https://www.simplifycpp.org

LLVM IR
Quick Reference

First Edition

DRAFT

LLVM
COMPILER INFRASTRUCTURE

Prepared by: Ayman Alheraki

LLVM IR Quick Reference

Prepared by Ayman Alheraki

Target Audience: professionals

simplifycpp.org

March 2025

Contents

Introduction

In the world of compiler development and complex software systems, **LLVM IR (Intermediate Representation)** stands as one of the most powerful tools a developer can master. Whether you're designing a new compiler, optimizing the performance of an existing program, or even building advanced operating systems, understanding LLVM IR opens endless possibilities. This reference is not just an ordinary book; it is **your gateway to deep-level programming**, where you control every detail of the code that runs on any known processor.

Why Should You Read This Reference More Than Once?

1. **Because LLVM IR is Not Just a Language, It's a Philosophy:**
   LLVM IR is not just an intermediate representation of code; it's a way of thinking. Every time you read this reference, you'll uncover a new layer of understanding. You'll learn how to think like a compiler, how to control data flow, and how to transform complex ideas into simple, efficient instructions.

2. **Because It's the Language All Processors Speak:**
   Whether you're working on x86, ARM, RISC-V, or any other processor, LLVM IR is the common language that connects them all. This reference will help you understand how to transform the code you write into instructions that any processor can understand,

making you a more versatile and powerful developer.

3. **Because Mastery Requires Repetition:**
   LLVM IR is not something you can learn overnight. Every time you read this reference, you'll discover new details, grasp deeper concepts, and become more capable of applying this knowledge to your real-world projects. Mastery requires repetition, and this reference is designed to be your lifelong companion.

Why is This Reference Different?

1. **Practical and Quick Reference:** This reference is not a lengthy theoretical book; it's a **quick, practical guide** that delivers information clearly and concisely, with examples and practical applications to help you grasp concepts quickly.

2. **It Will Be Continuously Updated:** Technology evolves rapidly, and this reference will keep pace. We will update it regularly to include any new developments in the world of LLVM IR, whether they are performance improvements, new features, or corrections to any parts that can be enhanced. This reference is **alive and evolving**, just like the technology it covers.

3. **Designed to Be Your Permanent Companion:** Whether you're a beginner or an expert, this reference will always be on your desk (or your device) as a quick guide you can turn to at any time. You won't need to search through dozens of sources; everything you need is here.

How Will LLVM IR Change the Way You Work?

When you master LLVM IR, you'll be able to:

- **Optimize your programs' performance** like never before, because you'll control every detail of the code.
- **Build your own compilers** that support new programming languages or specialized processors.
- **Understand how compilers** like Clang and GCC work under the hood, making you a more efficient developer.
- **Work with any processor** effortlessly, because LLVM IR is the common language among them all.

A Final Word: This Reference is the Beginning of Your Journey to Mastery

Don't treat this reference as a book you read once only. Treat it as a lifelong friend, a reference you return to whenever you face new challenges in the world of compiler development and software systems. Every time you read this reference, you'll discover something new, and you'll become more proficient in this powerful intermediate language.

Stay Connected

For more discussions and valuable content about **LLVM IR Quick Reference**, I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Part I

beginners Users: to understand LLVM IR syntax and structure.

Chapter 1

Introduction to LLVM IR

1.1 Overview of LLVM Intermediate Representation (IR)

1.1.1 What is LLVM IR?

LLVM Intermediate Representation (IR) is a low-level, platform-independent programming language that serves as the intermediate step between high-level source code (e.g., C, C++, Rust) and machine-specific assembly code. It is a critical component of the LLVM (Low-Level Virtual Machine) project, which is a collection of modular and reusable compiler and toolchain technologies. LLVM IR is designed to be both human-readable and machine-optimizable, making it a powerful tool for code analysis, transformation, and optimization. LLVM IR is often referred to as a ”portable assembly language” because it abstracts away the details of specific hardware architectures while retaining enough low-level information to enable efficient code generation. This allows developers to write optimizations and analyses that work across multiple target architectures, such as x86, ARM, and RISC-V.

1.1.2 Key Characteristics of LLVM IR

LLVM IR has several defining characteristics that make it unique and versatile:

1. **Static Single Assignment (SSA) Form**:
   LLVM IR uses SSA form, which means that every variable is assigned exactly once, and every use of a variable is defined by a single assignment. This simplifies data flow analysis and enables powerful optimizations, such as dead code elimination and constant propagation.

2. **Strong Typing**:
   LLVM IR is strongly typed, meaning that every value and operation has a well-defined type. This helps catch errors early and ensures that optimizations are performed safely. Common types include integers, floating-point numbers, pointers, and aggregates (e.g., arrays and structures).

3. **Three-Address Code**:
   LLVM IR is based on a three-address code format, where most instructions operate on two operands and produce a single result. This format is easy to optimize and translate into machine code.

4. **Extensible and Modular**:
   LLVM IR is designed to be extensible, allowing new instructions, types, and metadata to be added without breaking existing tools. This modularity makes it suitable for a wide range of applications, from traditional compilers to domain-specific languages.

5. **Human-Readable Text Format**:
   LLVM IR can be represented in two forms: a human-readable text format (`.ll` files) and a compact binary format (`.bc` files). The text format is particularly useful for debugging and learning, as it closely resembles assembly language.

1.1.3 Structure of LLVM IR

LLVM IR is organized into a hierarchy of components, each serving a specific purpose:

1. **Module**:
   A module is the top-level container in LLVM IR. It represents an entire compilation unit, such as a single source file. A module contains global variables, functions, and metadata.

2. **Function**:
   A function is a callable unit of code that contains a list of basic blocks. Functions can have parameters, return types, and attributes (e.g., `inline`, `noinline`).

3. **Basic Block**:
   A basic block is a sequence of non-branching instructions that ends with a terminator instruction (e.g., `ret`, `br`, `switch`). Basic blocks are the building blocks of control flow graphs (CFGs), which represent the flow of execution within a function.

4. **Instruction**:
   An instruction is a single operation, such as an arithmetic operation, memory access, or control flow instruction. Instructions operate on values, which can be constants, variables, or the results of other instructions.

5. **Value**:
   A value is a piece of data that can be used as an operand to an instruction. Values can be constants (e.g., `42`, `3.14`), variables, or the results of instructions.

6. **Type**:
   A type defines the kind of data that a value can hold. LLVM IR supports a wide range of types, including integers, floating-point numbers, pointers, arrays, structures, and vectors.

7. **Metadata**:
   Metadata is additional information attached to LLVM IR constructs, such as debug information, profile data, or optimization hints. Metadata does not affect the semantics of the program but is useful for tools and optimizations.

1.1.4 Advantages of LLVM IR

LLVM IR offers several advantages over traditional intermediate representations:

1. **Portability**:
   LLVM IR is designed to be platform-independent, making it easy to target multiple architectures with a single codebase.

2. **Optimizability**:
   The SSA form and strong typing of LLVM IR enable a wide range of optimizations, from simple peephole optimizations to complex loop transformations.

3. **Interoperability**:
   LLVM IR can be generated from multiple high-level languages (e.g., C, C++, Rust, Swift) and can be translated into machine code for various architectures. This makes it a universal backend for compilers.

4. **Extensibility**:
   LLVM IR is highly extensible, allowing developers to add custom instructions, types, and metadata for specialized use cases.

5. **Tooling Support**:
   LLVM provides a rich set of tools for working with IR, including optimizers, analyzers, and debuggers. These tools make it easier to develop, test, and optimize code at the IR level.

1.1.5 Use Cases of LLVM IR

LLVM IR is used in a variety of applications, including:

1. **Compilers**:
   LLVM IR is the backbone of many modern compilers, such as Clang (C/C++), Rustc (Rust), and Swiftc (Swift). These compilers generate LLVM IR from high-level source code and then optimize and lower it to machine code.

2. **JIT Compilation**:
   LLVM IR is used in Just-In-Time (JIT) compilers, such as those found in the Julia programming language and the LLVM-based JIT in the V8 JavaScript engine. JIT compilers generate and optimize code at runtime, enabling dynamic performance improvements.

3. **Static Analysis**:
   LLVM IR is a popular target for static analysis tools, which analyze code for bugs, security vulnerabilities, and performance issues. The SSA form and strong typing make it easier to reason about program behavior.

4. **Domain-Specific Languages (DSLs)**:
   LLVM IR is often used as a backend for DSLs, allowing developers to create custom languages without worrying about low-level code generation.

5. **Research and Prototyping**:
   LLVM IR is widely used in academic research and prototyping due to its flexibility and ease of use. Researchers can experiment with new optimizations, language features, and compiler techniques using LLVM IR.

1.1.6 Example of LLVM IR

Here is a simple example of LLVM IR for a function that adds two integers:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

In this example:

- `define i32 @add(i32 %a, i32 %b)` declares a function named `add` that takes two 32-bit integer arguments (`%a` and `%b`) and returns a 32-bit integer.
- `%result = add i32 %a, %b` adds the two arguments and stores the result in a new variable `%result`.
- `ret i32 %result` returns the result of the addition.

1.1.7 Conclusion

LLVM IR is a powerful and versatile intermediate representation that bridges the gap between high-level source code and machine code. Its SSA form, strong typing, and modular design make it an ideal target for compilers, optimizers, and analysis tools. By understanding LLVM IR, developers can gain deeper insights into how their code is compiled and optimized, enabling them to write more efficient and portable software.

1.2 LLVM IR as a Static Single Assignment (SSA) Form

1.2.1 Introduction to Static Single Assignment (SSA)

Static Single Assignment (SSA) is a fundamental concept in compiler design and intermediate representations (IRs). It is a property of code where each variable is assigned exactly once, and every use of a variable is defined by a single assignment. This property simplifies data flow analysis and enables powerful optimizations, making SSA a cornerstone of modern compiler technology.

LLVM IR is designed around the SSA form, which is one of the reasons it is so effective for optimization and analysis. In this section, we will explore what SSA is, how it works in LLVM IR, and why it is so important for compilers and tools.

1.2.2 What is SSA?

In SSA form, every variable in the program is assigned a value exactly once. This means that if a variable is reassigned, it is treated as a new variable with a unique name. For example, consider the following pseudo-code:

```
x = 1
x = x + 2
y = x * 3
```

In SSA form, this code would be transformed into:

```
x1 = 1
x2 = x1 + 2
y1 = x2 * 3
```

Here, each assignment to `x` is given a unique name (`x1`, `x2`), ensuring that every use of `x` refers to a single, unambiguous definition.

1.2.3 Benefits of SSA

The SSA form provides several key benefits:

1. **Simplified Data Flow Analysis**:
   Since each variable has a single definition, tracking the flow of data through the program becomes much easier. This simplifies algorithms for optimizations like constant propagation, dead code elimination, and copy propagation.

2. **Efficient Optimizations**:
   Many optimizations are more efficient or even trivial to implement in SSA form. For example, detecting unused variables or redundant computations is straightforward because each variable is defined only once.

3. **Improved Debugging**:
   SSA form makes it easier to track the origin of values, which can be helpful for debugging and understanding program behavior.

4. **Parallelism and Vectorization**:
   SSA form simplifies the identification of independent computations, making it easier to parallelize or vectorize code.

1.2.4 SSA in LLVM IR

LLVM IR is built on the SSA form, and this design choice is one of the reasons LLVM is so powerful and flexible. In LLVM IR, every value is defined exactly once, and every use of a value refers to a single definition. This applies to both local variables (temporary values) and global variables.

1. **SSA Variables in LLVM IR**

   In LLVM IR, SSA variables are represented as numbered temporaries, often prefixed with a `%` symbol. For example:

```
%x1 = add i32 1, 2
%x2 = add i32 %x1, 3
%y1 = mul i32 %x2, 4
```

   Here, `%x1`, `%x2`, and `%y1` are SSA variables, each defined exactly once.

2. **Phi Nodes: Handling Control Flow**

   One of the challenges of SSA form is handling control flow, such as branches and loops. When a variable can be assigned different values along different paths, SSA form uses a special construct called a **Phi node** (ϕ) to merge the values.

   For example, consider the following pseudo-code:

```
if (condition) {
  x = 1
} else {
  x = 2
}
y = x + 3
```

   In SSA form, this would be represented as:

   llvm

   Copy

```
  br i1 %condition, label %if_true, label %if_false

if_true:
  %x1 = i32 1
  br label %merge

if_false:
  %x2 = i32 2
  br label %merge

merge:
  %x3 = phi i32 [%x1, %if_true], [%x2, %if_false]
  %y1 = add i32 %x3, 3
```

Here, the `phi` instruction in the `merge` block selects the value of `x` based on the control flow path taken. The `phi` node ensures that `%x3` has a single definition, even though it can come from multiple sources.

1.2.5 Why SSA is Important in LLVM IR

The use of SSA form in LLVM IR provides several advantages:

1. **Enables Advanced Optimizations**:
   Many LLVM optimizations, such as global value numbering (GVN), loop-invariant code motion, and dead store elimination, rely on the SSA form to work effectively.

2. **Simplifies Analysis**:
   SSA form makes it easier to analyze data dependencies and identify opportunities for optimization. For example, determining whether a variable is used or unused is trivial in SSA form.

3. **Improves Code Generation**:
   The SSA form simplifies the process of generating efficient machine code, as the compiler can easily track the lifetime and usage of variables.

4. **Facilitates Debugging and Profiling**:
   SSA form makes it easier to map high-level source code to low-level IR, which is useful for debugging and profiling tools.

1.2.6 Challenges of SSA

While SSA form is powerful, it also introduces some challenges:

1. **Complexity of Phi Nodes**:
   Phi nodes can make the IR more complex, especially in programs with deeply nested control flow.

2. **Memory Usage**:
   SSA form can increase memory usage, as each assignment creates a new variable. However, this is typically offset by the benefits of simplified analysis and optimization.

3. **Translation to Machine Code**:
   SSA form must be converted back to a non-SSA form (e.g., using register allocation) before generating machine code. This process can be complex but is well-handled by LLVM's backend.

1.2.7 Example of SSA in LLVM IR

Let’s look at a complete example of SSA in LLVM IR. Consider the following C code:

```
int foo(int a, int b) {
  int x;
  if (a > b) {
    x = a;
  } else {
    x = b;
  }
  return x + 1;
}
```

The corresponding LLVM IR in SSA form would look like this:

```
define i32 @foo(i32 %a, i32 %b) {
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %if_true, label %if_false

if_true:
  %x1 = add i32 %a, 0  ; x = a
  br label %merge

if_false:
  %x2 = add i32 %b, 0  ; x = b
  br label %merge

merge:
  %x3 = phi i32 [%x1, %if_true], [%x2, %if_false]
  %result = add i32 %x3, 1
  ret i32 %result
}
```

In this example:

- The `phi` instruction in the `merge` block selects the value of `x` based on whether the `if_true` or `if_false` block was executed.
- The SSA variables `%x1` and `%x2` are merged into `%x3`, which is then used to compute the result.

1.2.8 Conclusion

LLVM IR's use of SSA form is one of its most powerful features, enabling efficient and effective optimizations, analyses, and code generation. By ensuring that every variable is assigned exactly once, SSA simplifies data flow analysis and makes it easier to reason about program behavior. While SSA introduces some complexity, particularly with phi nodes, the benefits far outweigh the challenges.

1.3 LLVM IR vs. Assembly and Other IRs

1.3.1 Introduction

Intermediate Representations (IRs) are a critical part of modern compilers, serving as a bridge between high-level source code and low-level machine code. LLVM IR is one of the most widely used IRs, but it is not the only one. In this section, we will compare LLVM IR with assembly language and other IRs, highlighting their similarities, differences, and use cases. This comparison will help beginners understand why LLVM IR is unique and how it fits into the broader landscape of compiler technology.

1.3.2 LLVM IR vs. Assembly Language

Assembly language is a low-level programming language that is closely tied to the architecture of a specific machine. It provides a human-readable representation of machine code, with instructions that directly correspond to the CPU's operations. In contrast, LLVM IR is a higher-level, platform-independent representation that abstracts away many hardware-specific details.

1. **Key Differences**

   (a) **Abstraction Level**:

   - **Assembly**: Assembly language is very low-level, with instructions that map directly to machine code. It includes details like registers, memory addresses, and CPU-specific instructions.
   - **LLVM IR**: LLVM IR is higher-level and platform-independent. It abstracts away hardware details, such as registers and instruction sets, making it portable across different architectures.

(b) **Portability**:

- **Assembly**: Assembly code is specific to a particular CPU architecture (e.g., x86, ARM). Code written for one architecture cannot run on another without significant modification.
- **LLVM IR**: LLVM IR is designed to be portable. The same IR code can be optimized and compiled for multiple target architectures using LLVM's backend.

(c) **Readability and Maintainability**:

- **Assembly**: Assembly code is difficult to read and maintain due to its low-level nature and lack of abstraction.
- **LLVM IR**: LLVM IR is more readable and maintainable, with a structured format and support for high-level constructs like functions, loops, and types.

(d) **Optimization**:

- **Assembly**: Optimizing assembly code is challenging and requires deep knowledge of the target architecture.
- **LLVM IR**: LLVM IR is designed for optimization. Its SSA form and strong typing enable a wide range of optimizations that are difficult or impossible to perform at the assembly level.

(e) **Tooling**:

- **Assembly**: Tools for working with assembly language are often limited and architecture-specific.
- **LLVM IR**: LLVM provides a rich set of tools for analyzing, optimizing, and transforming IR code, making it easier to work with.

2. **Example Comparison**

Consider a simple addition operation in C:

```c
int add(int a, int b) {
  return a + b;
}
```

Assembly (x86):

```
add:
  mov eax, edi  ; Move argument a into register eax
  add eax, esi  ; Add argument b to eax
  ret           ; Return the result
```

LLVM IR:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

The LLVM IR version is more concise and abstract, while the assembly version includes low-level details like registers and instructions.

1.3.3 LLVM IR vs. Other IRs

LLVM IR is not the only intermediate representation used in compilers. Other popular IRs include:

1. **GIMPLE (GNU)**:
    - Used by the GNU Compiler Collection (GCC).

- GIMPLE is a tree-based IR that represents high-level constructs like loops and conditionals.
- Unlike LLVM IR, GIMPLE is not in SSA form by default, though it can be converted to SSA for optimization.

2. **Java Bytecode**:

- Used by the Java Virtual Machine (JVM).
- Java bytecode is a stack-based IR that is designed for portability and security.
- It is lower-level than LLVM IR and lacks many of the optimization opportunities provided by SSA form.

3. **CIL (Common Intermediate Language)**:

- Used by the .NET framework.
- CIL is a stack-based IR similar to Java bytecode but with additional features like support for value types and tail call optimization.
- Like Java bytecode, it is not in SSA form.

4. **SPIR-V (Standard Portable Intermediate Representation)**:

- Used for GPU programming in Vulkan and OpenCL.
- SPIR-V is a binary IR designed for portability across different GPU architectures.
- It is lower-level than LLVM IR and focuses on graphics and compute workloads.

1. **Comparison with LLVM IR**

   (a) **SSA Form**:

- LLVM IR uses SSA form, which simplifies optimization and analysis.
- Many other IRs (e.g., GIMPLE, Java bytecode, CIL) do not use SSA form by default, though some can be converted to SSA for optimization.

(b) **Portability**:

- LLVM IR is designed to be portable across multiple architectures.
- Other IRs like Java bytecode and CIL are also portable but are tied to specific runtime environments (JVM, .NET).

(c) **Optimization**:

- LLVM IR is highly optimizable due to its SSA form and strong typing.
- Other IRs may have limited optimization capabilities, depending on their design and use case.

(d) **Tooling**:

- LLVM provides a comprehensive set of tools for working with IR, including optimizers, analyzers, and debuggers.
- Other IRs may have less mature or less flexible tooling.

1.3.4 Advantages of LLVM IR Over Other IRs

1. **Flexibility**:
   LLVM IR is designed to be extensible and modular, making it suitable for a wide range of applications, from traditional compilers to domain-specific languages.

2. **Optimization**:
   The SSA form and strong typing of LLVM IR enable a wide range of optimizations that are difficult or impossible to perform with other IRs.

3. **Portability**:

LLVM IR is platform-independent, allowing the same code to be optimized and compiled for multiple architectures.

4. **Tooling**:
   LLVM provides a rich set of tools for working with IR, making it easier to develop, test, and optimize code.

1.3.5 When to Use LLVM IR vs. Other IRs

1. **Use LLVM IR When**:
   - You need a portable, optimizable IR for multiple target architectures.
   - You are building a compiler or toolchain and want access to LLVM's powerful optimization and analysis capabilities.
   - You are working on a project that requires extensibility and modularity.
2. **Use Other IRs When**:
   - You are working within a specific ecosystem (e.g., JVM for Java bytecode, .NET for CIL).
   - You need an IR tailored to a specific use case (e.g., SPIR-V for GPU programming).
   - You are constrained by existing tools or infrastructure that use a particular IR.

1.3.6 Conclusion

LLVM IR stands out among intermediate representations due to its SSA form, strong typing, portability, and powerful tooling. While assembly language and other IRs have their uses, LLVM IR is uniquely suited for modern compiler development, optimization, and analysis. By

understanding the differences between LLVM IR and other representations, developers can make informed decisions about which tools and technologies to use for their projects.

Chapter 2

LLVM IR Syntax & Structure

2.1 Modules, Functions, and Basic Blocks

2.1.1 Introduction

LLVM IR is a structured and hierarchical intermediate representation that organizes code into modules, functions, and basic blocks. Understanding these components is essential for working with LLVM IR, as they form the foundation of its syntax and structure. In this section, we will explore each of these components in detail, providing examples and explanations to help beginners grasp their roles and relationships.

2.1.2 Modules

A **module** is the top-level container in LLVM IR. It represents an entire compilation unit, such as a single source file or a collection of related functions and data. A module contains all the information needed to compile and optimize a program, including functions, global variables, and metadata.

1. **Structure of a Module**

   A module in LLVM IR typically includes the following components:

   (a) **Target Information**: Specifies the target architecture, data layout, and other platform-specific details.

   (b) **Global Variables**: Variables that are accessible throughout the module.

   (c) **Functions**: The code for each function in the module.

   (d) **Metadata**: Additional information, such as debug information or optimization hints.

2. **Example of a Module**

   Here is an example of a simple LLVM IR module:

```
; Target information
target datalayout =
 ↪  "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Global variable
@global_var = global i32 42, align 4

; Function declaration
declare i32 @printf(i8*, ...)

; Function definition
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

In this example:

- The `target datalayout` and `target triple` specify the target architecture.
- `@global_var` is a global variable initialized to the value `42`.
- `@printf` is a function declaration for the standard C library function `printf`.
- `@add` is a function definition that adds two integers and returns the result.

2.1.3 Functions

A **function** is a callable unit of code in LLVM IR. It contains a list of basic blocks, each representing a sequence of non-branching instructions. Functions can have parameters, return types, and attributes that describe their behavior.

1. **Structure of a Function**

   A function in LLVM IR consists of the following components:

   (a) **Function Signature**: Specifies the function's name, return type, parameter types, and attributes.

   (b) **Basic Blocks**: A list of basic blocks that make up the function's body.

   (c) **Terminator Instructions**: Instructions that end a basic block, such as `ret` (return) or `br` (branch).

2. **Example of a Function**

   Here is an example of a function in LLVM IR:

```
define i32 @multiply(i32 %a, i32 %b) {
entry:
  %result = mul i32 %a, %b
  ret i32 %result
}
```

In this example:

- The function `@multiply` takes two integer parameters (`%a` and `%b`) and returns an integer.
- The `entry` block contains a single instruction (`mul`) that multiplies the two parameters and stores the result in `%result`.
- The `ret` instruction returns the result.

3. **Function Attributes**

   Functions can have attributes that provide additional information about their behavior. For example:

   - `nounwind`: Indicates that the function does not throw exceptions.
   - `readonly`: Indicates that the function does not modify memory.
   - `inlinehint`: Suggests that the function should be inlined.

   Example:

```
define i32 @square(i32 %x) #0 {
  %result = mul i32 %x, %x
  ret i32 %result
}
```

```
attributes #0 = { nounwind readonly }
```

2.1.4 Basic Blocks

A **basic block** is a sequence of non-branching instructions that ends with a terminator instruction. Basic blocks are the building blocks of control flow graphs (CFGs), which represent the flow of execution within a function.

1. **Structure of a Basic Block**

   A basic block in LLVM IR consists of the following components:

   (a) **Label**: A unique identifier for the block (e.g., `entry`, `if_true`, `if_false`).

   (b) **Instructions**: A list of non-branching instructions that perform computations or memory operations.

   (c) **Terminator Instruction**: An instruction that ends the block, such as `ret`, `br`, or `switch`.

2. **Example of a Basic Block**

   Here is an example of a basic block in LLVM IR:

```
entry:
  %x = add i32 1, 2
  %y = mul i32 %x, 3
  ret i32 %y
```

   In this example:

- The `entry` block contains two instructions (`add` and `mul`) that perform computations.
- The `ret` instruction terminates the block and returns the result.

3. **Control Flow with Basic Blocks**

   Basic blocks are connected by terminator instructions to form control flow graphs. For example, consider a function with an `if` statement:

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %if_true, label %if_false

if_true:
  ret i32 %a

if_false:
  ret i32 %b
}
```

   In this example:

   - The `entry` block compares `%a` and `%b` and branches to either `if_true` or `if_false`.
   - The `if_true` and `if_false` blocks return the appropriate value.

2.1.5 Relationship Between Modules, Functions, and Basic Blocks

- A **module** contains one or more **functions**.

- A **function** contains one or more **basic blocks**.
- A **basic block** contains a sequence of **instructions**.

This hierarchical structure allows LLVM IR to represent complex programs in a clear and organized manner.

2.1.6 Example: A Complete Module

Here is an example of a complete LLVM IR module that includes a function with multiple basic blocks:

```
target datalayout =
↪  "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

define i32 @max(i32 %a, i32 %b) {
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %if_true, label %if_false

if_true:
  ret i32 %a

if_false:
  ret i32 %b
}
```

In this example:

- The module contains a single function `@max`.
- The function has three basic blocks: `entry`, `if_true`, and `if_false`.

- The `entry` block performs a comparison and branches to one of the other blocks.
- The `if_true` and `if_false` blocks return the appropriate value.

2.1.7 Conclusion

Modules, functions, and basic blocks are the fundamental building blocks of LLVM IR. By understanding their structure and relationships, beginners can start to write, analyze, and optimize LLVM IR code effectively. In the next section, we will explore the instructions and operands that make up the body of basic blocks, providing a deeper understanding of how LLVM IR represents computations and control flow.

2.2 Instruction Format and Operand Types

2.2.1 Introduction

Instructions are the fundamental units of computation in LLVM IR. They represent operations such as arithmetic, memory access, and control flow. Each instruction operates on one or more operands, which are the values used as inputs to the operation. Understanding the format of instructions and the types of operands they use is essential for working with LLVM IR. In this section, we will explore the structure of LLVM IR instructions, the types of operands they can take, and how they are used in practice.

2.2.2 Instruction Format

LLVM IR instructions follow a consistent format that makes them easy to read and understand. Each instruction typically consists of the following components:

1. **Result Variable (Optional)**:
   Many instructions produce a result, which is stored in a variable. The result variable is optional for instructions that do not produce a value (e.g., `store`, `br`).

2. **Operation**:
   The operation specifies what the instruction does, such as addition, subtraction, or memory access.

3. **Operands**:
   The operands are the values used as inputs to the operation. They can be constants, variables, or the results of other instructions.

4. **Type**:
   The type of the result and operands is explicitly specified in LLVM IR. This ensures type safety and enables optimizations.

1. **General Syntax**

   The general syntax for an LLVM IR instruction is:

   ```
   <result> = <operation> <type> <operand1>, <operand2>, ...
   ```

   For example:

   ```
   %sum = add i32 %a, %b
   ```

   Here:

   - `%sum` is the result variable.
   - `add` is the operation.
   - `i32` is the type of the operands and result.
   - `%a` and `%b` are the operands.

2. **Terminator Instructions**

   Terminator instructions are special instructions that end a basic block. They include `ret`, `br`, and `switch`. These instructions do not produce a result and have a slightly different syntax.

   Example:

   ```
   ret i32 %result
   ```

   Here:

   - `ret` is the terminator instruction.
   - `i32 %result` is the operand.

2.2.3 Operand Types

Operands in LLVM IR can be of various types, depending on the instruction and context. The most common operand types include:

1. **Constants**:
   Constants are fixed values that do not change during program execution. They can be integers, floating-point numbers, or special constants like `null`.

   Examples:

```
%x = add i32 5, 10        ; Integer constants
%y = fadd float 3.14, 2.71 ; Floating-point constants
```

2. **Variables**:
   Variables are named values that represent the result of previous instructions or function parameters. They are prefixed with a `%` symbol.

   Examples:

```
%sum = add i32 %a, %b     ; %a and %b are variables
```

3. **Temporary Values**:
   Temporary values are unnamed results of instructions. They are often used in intermediate computations.

   Example:

```
%tmp = mul i32 %x, %y
%result = add i32 %tmp, %z
```

4. **Global Variables**:
   Global variables are accessible throughout the module and are prefixed with a `@` symbol.

   Example:

```
@global_var = global i32 42
%x = load i32, i32* @global_var
```

5. **Labels**:
   Labels are used to identify basic blocks and are often used as operands in branch instructions.

   Example:

```
br label %next_block
```

2.2.4 Common Instruction Types

LLVM IR supports a wide range of instructions, which can be broadly categorized into the following types:

1. **Arithmetic Instructions**

   Arithmetic instructions perform mathematical operations on integer or floating-point operands.

   Examples:

```
%sum = add i32 %a, %b       ; Integer addition
%dif = sub i32 %a, %b       ; Integer subtraction
%prod = mul i32 %a, %b      ; Integer multiplication
%quot = sdiv i32 %a, %b     ; Signed integer division
```

2. **Memory Instructions**

   Memory instructions are used to load from or store to memory.

   Examples:

```
%val = load i32, i32* %ptr    ; Load a value from memory
store i32 %val, i32* %ptr     ; Store a value to memory
```

3. **Control Flow Instructions**

   Control flow instructions manage the flow of execution within a function.

   Examples:

```
br label %next_block          ; Unconditional branch
br i1 %cond, label %true, label %false ; Conditional branch
ret i32 %result               ; Return from function
```

4. **Comparison Instructions**

   Comparison instructions compare two values and produce a boolean result.

   Examples:

```
%cmp = icmp eq i32 %a, %b     ; Integer equality comparison
%fcmp = fcmp ult float %x, %y ; Floating-point less-than comparison
```

5. **Conversion Instructions**

   Conversion instructions convert values from one type to another.

   Examples:

```
%trunc = trunc i64 %x to i32  ; Truncate a 64-bit integer to 32 bits
%zext = zext i32 %x to i64    ; Zero-extend a 32-bit integer to 64
↪  bits
%fptrunc = fptrunc double %x to float ; Truncate a double to a float
```

2.2.5 Example: Using Instructions and Operands

Here is an example of a function that uses various instructions and operand types:

```
define i32 @compute(i32 %a, i32 %b, float %c) {
entry:
  %sum = add i32 %a, %b           ; Integer addition
  %cmp = icmp sgt i32 %sum, 100   ; Integer comparison
  br i1 %cmp, label %if_true, label %if_false

if_true:
  %result = sitofp i32 %sum to float ; Convert integer to float
  %final = fadd float %result, %c    ; Floating-point addition
  ret i32 1

if_false:
  ret i32 0
}
```

In this example:

- The `add` instruction performs integer addition.
- The `icmp` instruction compares the result to `100`.
- The `br` instruction branches based on the comparison result.

- The `sitofp` instruction converts an integer to a floating-point value.
- The `fadd` instruction performs floating-point addition.

2.2.6 Conclusion

Instructions and operands are the building blocks of LLVM IR. By understanding their format and types, beginners can start to write and analyze LLVM IR code effectively. In the next section, we will explore the types and type system in LLVM IR, providing a deeper understanding of how values are represented and manipulated.

2.3 Comments and Metadata

2.3.1 Introduction

Comments and metadata are essential components of LLVM IR that enhance readability, maintainability, and functionality. Comments provide human-readable explanations within the code, while metadata attaches additional information to LLVM IR constructs for debugging, optimization, and analysis purposes. In this section, we will explore the syntax and usage of comments and metadata in LLVM IR, providing detailed explanations and examples to help beginners understand their importance and application.

2.3.2 Comments in LLVM IR

Comments in LLVM IR are used to annotate the code with human-readable explanations. They are ignored by the LLVM compiler and tools but are invaluable for developers who need to understand or modify the IR.

1. **Syntax of Comments**

   LLVM IR supports two types of comments:

   (a) **Single-Line Comments**:
   Single-line comments begin with a semicolon (`;`) and continue to the end of the line.

   Example:

```
; This is a single-line comment
%x = add i32 1, 2  ; Add 1 and 2, store result in %x
```

(b) **Multi-Line Comments**:
Multi-line comments are enclosed within /* and */. They can span multiple lines.
Example:

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
%y = sub i32 %x, 3  ; Subtract 3 from %x, store result in %y
```

2. **Best Practices for Using Comments**

- **Explain Complex Logic**: Use comments to explain complex or non-obvious parts of the code.
- **Document Assumptions**: Document any assumptions or constraints that the code relies on.
- **Avoid Redundancy**: Avoid commenting on obvious or self-explanatory code.
- **Keep Comments Updated**: Ensure that comments are updated when the code changes to avoid confusion.

2.3.3 Metadata in LLVM IR

Metadata in LLVM IR is used to attach additional information to IR constructs, such as functions, instructions, and basic blocks. This information does not affect the semantics of the program but is useful for debugging, optimization, and analysis.

1. **Types of Metadata**

Metadata in LLVM IR can be categorized into the following types:

(a) **Debug Information**:

Debug information is used to map LLVM IR constructs back to the original source code, enabling debugging tools to provide meaningful information.

Example:

```
!llvm.dbg.cu = !{!0}
!0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1,
↪  producer: "clang", isOptimized: true)
```

(b) **Profile Data**:

Profile data provides information about the runtime behavior of the program, such as branch probabilities and execution counts. This data is used by optimizers to make informed decisions.

Example:

```
!prof = !{!1}
!1 = !{!"branch_weights", i32 30, i32 70}
```

(c) **Optimization Hints**:

Optimization hints provide guidance to the optimizer, such as loop unrolling factors or alignment preferences.

Example:

```
!llvm.loop = !{!0}
!0 = distinct !{!0, !1}
!1 = !{!"llvm.loop.unroll.count", i32 4}
```

(d) **Custom Metadata**:

Custom metadata can be defined by developers to attach any additional information to IR constructs.

Example:

```
!custom = !{!0}
!0 = !{!"custom_info", i32 42}
```

2. **Syntax of Metadata**

   Metadata in LLVM IR is represented using metadata nodes, which are identified by a `!` prefix. Metadata nodes can be attached to instructions, functions, and other IR constructs using metadata attachments.

   Example:

```
define i32 @add(i32 %a, i32 %b) !dbg !5 {
  %result = add i32 %a, %b, !dbg !6
  ret i32 %result
}

!llvm.dbg.cu = !{!0}
!0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1, producer:
↪  "clang", isOptimized: true)
!5 = distinct !DISubprogram(name: "add", scope: !1, file: !1, line: 1,
↪  type: !2)
!6 = !DILocation(line: 2, column: 3, scope: !5)
```

   In this example:

   - `!dbg !5` attaches debug information to the function `@add`.
   - `!dbg !6` attaches debug information to the `add` instruction.

- `!llvm.dbg.cu`, `!0`, `!5`, and `!6` are metadata nodes that define the debug information.

3. **Using Metadata for Debugging**

   Metadata is particularly useful for debugging, as it allows developers to map LLVM IR constructs back to the original source code. This is done using the **Debug Information (DI)** metadata, which includes information such as source file names, line numbers, and variable locations.

   Example:

```
define i32 @multiply(i32 %a, i32 %b) !dbg !10 {
entry:
  %result = mul i32 %a, %b, !dbg !11
  ret i32 %result, !dbg !12
}

!llvm.dbg.cu = !{!0}
!0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1, producer:
↪  "clang", isOptimized: true)
!1 = !DIFile(filename: "example.c", directory: "/path/to/source")
!10 = distinct !DISubprogram(name: "multiply", scope: !1, file: !1,
↪  line: 3, type: !2)
!11 = !DILocation(line: 4, column: 10, scope: !10)
!12 = !DILocation(line: 4, column: 3, scope: !10)
```

   In this example:

   - `!dbg !10` attaches debug information to the function `@multiply`.
   - `!dbg !11` and `!dbg !12` attach debug information to the `mul` and `ret` instructions, respectively.

- The metadata nodes (`!0`, `!1`, `!10`, `!11`, `!12`) provide detailed debug information, including the source file, line numbers, and column numbers.

2.3.4 Example: Combining Comments and Metadata

Here is an example that combines comments and metadata in LLVM IR:

```
; Function to compute the factorial of a number
define i32 @factorial(i32 %n) !dbg !5 {
entry:
  ; Check if n is 0 or 1
  %cmp = icmp sle i32 %n, 1, !dbg !6
  br i1 %cmp, label %base_case, label %recursive_case, !dbg !7

base_case:
  ; Return 1 for base case
  ret i32 1, !dbg !8

recursive_case:
  ; Compute factorial(n-1)
  %n_minus_1 = sub i32 %n, 1, !dbg !9
  %factorial_n_minus_1 = call i32 @factorial(i32 %n_minus_1), !dbg !10
  %result = mul i32 %n, %factorial_n_minus_1, !dbg !11
  ret i32 %result, !dbg !12
}

; Debug information metadata
!llvm.dbg.cu = !{!0}
!0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1, producer:
↪  "clang", isOptimized: true)
!1 = !DIFile(filename: "factorial.c", directory: "/path/to/source")
!5 = distinct !DISubprogram(name: "factorial", scope: !1, file: !1, line:
↪  1, type: !2)
```

```
!6 = !DILocation(line: 2, column: 10, scope: !5)
!7 = !DILocation(line: 2, column: 3, scope: !5)
!8 = !DILocation(line: 3, column: 5, scope: !5)
!9 = !DILocation(line: 5, column: 20, scope: !5)
!10 = !DILocation(line: 5, column: 10, scope: !5)
!11 = !DILocation(line: 5, column: 5, scope: !5)
!12 = !DILocation(line: 6, column: 3, scope: !5)
```

In this example:

- Comments are used to explain the logic of the factorial function.
- Metadata is used to attach debug information to the function and instructions.

2.3.5 Conclusion

Comments and metadata are powerful tools for enhancing the readability, maintainability, and functionality of LLVM IR. Comments provide human-readable explanations, while metadata attaches additional information for debugging, optimization, and analysis. By mastering the use of comments and metadata, beginners can write more effective and understandable LLVM IR code.

Chapter 3

Data Types in LLVM IR

3.1 Primitive Types: `iN`, `float`, `double`, `void`, `ptr`

3.1.1 Introduction

Data types are fundamental to any programming language or intermediate representation (IR). They define the kind of data that can be stored, manipulated, and operated on. In LLVM IR, data types are categorized into **primitive types** and **derived types**. Primitive types are the basic building blocks, while derived types are constructed from primitive types. In this section, we will focus on **primitive types**, which include integer types (`iN`), floating-point types (`float`, `double`), the `void` type, and the pointer type (`ptr`). Understanding these types is essential for writing and analyzing LLVM IR code.

3.1.2 Integer Types (`iN`)

Integer types in LLVM IR are used to represent whole numbers. They are denoted by the prefix `i` followed by the number of bits (`N`). For example, `i32` represents a 32-bit integer, and

`i64` represents a 64-bit integer.

1. **Syntax**

   The syntax for integer types is:

   ```
   iN
   ```

   where `N` is the number of bits.

2. **Common Integer Types**

   - `i1`: A 1-bit integer, often used for boolean values (`true` or `false`).
   - `i8`: An 8-bit integer, commonly used for byte-sized data.
   - `i16`: A 16-bit integer.
   - `i32`: A 32-bit integer, commonly used for standard integers.
   - `i64`: A 64-bit integer, often used for larger integers or pointers on 64-bit architectures.

3. **Example Usage**

   ```
   %x = add i32 5, 10        ; Add two 32-bit integers
   %y = sub i64 100, 50      ; Subtract two 64-bit integers
   %z = and i1 1, 0          ; Perform a bitwise AND on two 1-bit
    ↪  integers
   ```

4. **Key Points**

   - Integer types can represent both signed and unsigned values, but the interpretation depends on the instructions used (e.g., `add` vs. `add nsw` for signed addition with no signed wrap).

- The size of an integer type (`N`) can be any positive integer, but common sizes are powers of 2 (e.g., 1, 8, 16, 32, 64).

3.1.3 Floating-Point Types (`float`, `double`)

Floating-point types in LLVM IR are used to represent real numbers. The two most common floating-point types are `float` (32-bit) and `double` (64-bit).

1. **Syntax**

   - `float`: A 32-bit floating-point number.
   - `double`: A 64-bit floating-point number.

2. **Example Usage**

```
%a = fadd float 3.14, 2.71  ; Add two 32-bit floating-point numbers
%b = fmul double 1.5, 2.0   ; Multiply two 64-bit floating-point
↪  numbers
```

3. **Key Points**

   - Floating-point types follow the IEEE 754 standard for floating-point arithmetic.
   - Operations on floating-point types use specific instructions, such as `fadd`, `fsub`, `fmul`, and `fdiv`.

3.1.4 Void Type (`void`)

The `void` type in LLVM IR is used to represent the absence of a value. It is typically used as the return type for functions that do not return a value.

1. **Syntax**

```
void
```

2. **Example Usage**

```
define void @print_hello() {
  call void @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]*
  ↪  @.str, i32 0, i32 0))
  ret void
}
```

In this example:

- The function `@print_hello` has a return type of `void`, indicating that it does not return a value.
- The `ret void` instruction is used to return from a `void` function.

3. **Key Points**

- The `void` type cannot be used as the type of a variable or operand.
- It is primarily used for function return types and certain instructions (e.g., `ret void`).

3.1.5 Pointer Type (`ptr`)

The pointer type in LLVM IR is used to represent memory addresses. It is denoted by the `ptr` keyword, which is a shorthand for a pointer to a generic memory location. In earlier versions of LLVM, pointers were explicitly typed (e.g., `i32*` for a pointer to an `i32`), but modern LLVM IR uses the `ptr` type for simplicity and flexibility.

1. **Syntax**

```
ptr
```

2. **Example Usage**

```
%x = alloca i32, align 4        ; Allocate memory for an i32 and
↪  return a pointer
%y = load i32, ptr %x           ; Load the value from the memory
↪  location pointed to by %x
store i32 42, ptr %x            ; Store the value 42 at the memory
↪  location pointed to by %x
```

3. **Key Points**

   - The `ptr` type is used to represent pointers to any type of data.
   - Pointer arithmetic and memory operations (e.g., `load`, `store`, `getelementptr`) are performed using the `ptr` type.
   - The `ptr` type is opaque, meaning it does not specify the type of data it points to. This simplifies the IR and makes it more flexible.

3.1.6 Example: Combining Primitive Types

Here is an example that demonstrates the use of primitive types in LLVM IR:

```
define i32 @compute(i32 %a, float %b, ptr %c) {
entry:
  %x = add i32 %a, 10           ; Integer addition
  %y = fadd float %b, 3.14      ; Floating-point addition
```

```
  %z = load i32, ptr %c          ; Load an integer from the memory location
  ↪  pointed to by %c
  %result = add i32 %x, %z       ; Add the loaded value to %x
  ret i32 %result                ; Return the result
}
```

In this example:

- `i32` is used for integer values.
- `float` is used for floating-point values.
- `ptr` is used for pointers to memory locations.

3.1.7 Conclusion

Primitive types are the foundation of LLVM IR, providing the basic building blocks for representing data. By understanding the `iN`, `float`, `double`, `void`, and `ptr` types, beginners can start to write and analyze LLVM IR code effectively. In the next section, we will explore **derived types**, which are constructed from primitive types and include arrays, structures, and vectors.

3.2 Composite Types: Structs, Arrays, Vectors

3.2.1 Introduction

Composite types in LLVM IR are used to group or aggregate multiple values into a single entity. They are constructed from primitive types and other composite types, enabling the representation of complex data structures such as arrays, structures, and vectors. In this section, we will explore the three main composite types in LLVM IR: **structs**, **arrays**, and **vectors**. Understanding these types is essential for working with complex data in LLVM IR.

3.2.2 Struct Types

Struct types in LLVM IR are used to represent a collection of fields, each of which can have a different type. Structs are similar to structures in high-level programming languages like C or C++.

1. **Syntax**

   The syntax for defining a struct type is:

   ```
   %struct_name = type { <type1>, <type2>, ... }
   ```

   where `<type1>`, `<type2>`, etc., are the types of the fields in the struct.

2. **Example Usage**

   Here is an example of a struct type and its usage:

   ```
   %Person = type { i32, float, ptr }  ; Define a struct with three
   ↪  fields: i32, float, and ptr
   ```

```
define void @print_person(ptr %p) {
  %age = load i32, ptr %p                          ; Load the first
  ↪  field (i32)
  %height = load float, ptr getelementptr inbounds (%Person, ptr %p,
  ↪  i32 0, i32 1) ; Load the second field (float)
  %name = load ptr, ptr getelementptr inbounds (%Person, ptr %p, i32
  ↪  0, i32 2)     ; Load the third field (ptr)
  ; (Assume a function @print exists to print these values)
  call void @print(i32 %age, float %height, ptr %name)
  ret void
}
```

In this example:

- `%Person` is a struct type with three fields: an `i32` (age), a `float` (height), and a `ptr` (name).
- The `getelementptr` instruction is used to access fields within the struct.

3. **Key Points**

- Struct types can contain any combination of primitive and composite types.
- Fields in a struct are accessed using the `getelementptr` instruction, which computes the memory address of a field.
- Structs are useful for representing complex data structures, such as records or objects.

3.2.3 Array Types

Array types in LLVM IR are used to represent a sequence of elements of the same type. Arrays are fixed in size, meaning the number of elements is specified at compile time.

1. **Syntax**

   The syntax for defining an array type is:

```
[<size> x <element_type>]
```

   where <size> is the number of elements, and <element_type> is the type of each element.

2. **Example Usage**

   Here is an example of an array type and its usage:

```
@global_array = global [4 x i32] [i32 1, i32 2, i32 3, i32 4]  ;
↪  Define a global array of 4 i32 elements

define i32 @sum_array(ptr %arr) {
  %sum = alloca i32, align 4
  store i32 0, ptr %sum
  br label %loop

loop:
  %i = phi i32 [0, %entry], [%next_i, %loop_body]  ; Loop counter
  %exit_cond = icmp eq i32 %i, 4                   ; Check if the loop
  ↪  should exit
  br i1 %exit_cond, label %exit, label %loop_body

loop_body:
  %element_ptr = getelementptr inbounds [4 x i32], ptr %arr, i32 0,
  ↪  i32 %i  ; Get pointer to the i-th element
  %element = load i32, ptr %element_ptr
  ↪  ; Load the i-th element
  %current_sum = load i32, ptr %sum
```

```
  %new_sum = add i32 %current_sum, %element
  store i32 %new_sum, ptr %sum
  %next_i = add i32 %i, 1
  br label %loop

exit:
  %final_sum = load i32, ptr %sum
  ret i32 %final_sum
}
```

In this example:

- `@global_array` is a global array of 4 `i32` elements.
- The `getelementptr` instruction is used to access elements within the array.
- A loop is used to sum the elements of the array.

3. **Key Points**

- Array types are fixed in size, and the size must be known at compile time.
- Elements in an array are accessed using the `getelementptr` instruction.
- Arrays are useful for representing sequences of data, such as buffers or matrices.

3.2.4 Vector Types

Vector types in LLVM IR are used to represent a fixed number of elements of the same type, similar to arrays. However, vectors are designed for SIMD (Single Instruction, Multiple Data) operations, making them ideal for parallel computations.

1. **Syntax**

The syntax for defining a vector type is:

```
<<num_elements> x <element_type>>
```

where <num_elements> is the number of elements, and <element_type> is the type of each element.

2. **Example Usage**

   Here is an example of a vector type and its usage:

```
define <4 x float> @add_vectors(<4 x float> %a, <4 x float> %b) {
  %result = fadd <4 x float> %a, %b  ; Perform element-wise addition
  ret <4 x float> %result
}
```

   In this example:

   - <4 x float> is a vector type with 4 float elements.
   - The fadd instruction performs element-wise addition on the vectors.

3. **Key Points**

   - Vector types are designed for SIMD operations, enabling parallel computations on multiple data elements.
   - The number of elements in a vector must be known at compile time.
   - Vectors are useful for performance-critical applications, such as graphics processing or scientific computing.

3.2.5 Example: Combining Composite Types

Here is an example that demonstrates the use of structs, arrays, and vectors in LLVM IR:

```
%Matrix = type { i32, i32, [4 x <4 x float>] }  ; Define a struct
↪  representing a 4x4 matrix

define float @matrix_trace(ptr %m) {
  %sum = alloca float, align 4
  store float 0.0, ptr %sum
  br label %loop

loop:
  %i = phi i32 [0, %entry], [%next_i, %loop_body]  ; Loop counter
  %exit_cond = icmp eq i32 %i, 4                   ; Check if the loop
  ↪  should exit
  br i1 %exit_cond, label %exit, label %loop_body

loop_body:
  %element_ptr = getelementptr inbounds %Matrix, ptr %m, i32 0, i32 2, i32
  ↪  %i, i32 %i  ; Get pointer to the (i,i)-th element
  %element = load float, ptr %element_ptr
  ↪  ; Load the (i,i)-th element
  %current_sum = load float, ptr %sum
  %new_sum = fadd float %current_sum, %element
  store float %new_sum, ptr %sum
  %next_i = add i32 %i, 1
  br label %loop

exit:
  %final_sum = load float, ptr %sum
  ret float %final_sum
}
```

In this example:

- `%Matrix` is a struct type representing a 4x4 matrix, with fields for the number of rows, columns, and a 4x4 array of vectors.
- The `getelementptr` instruction is used to access elements within the matrix.
- A loop is used to compute the trace of the matrix (the sum of diagonal elements).

3.2.6 Conclusion

Composite types in LLVM IR—structs, arrays, and vectors—enable the representation of complex data structures and facilitate efficient data manipulation. By understanding these types, beginners can write more expressive and optimized LLVM IR code. In the next section, we will explore **function types** and **pointer types**, which are essential for working with functions and memory in LLVM IR.

3.3 Opaque & Pointer Types

3.3.1 Introduction

In LLVM IR, **opaque types** and **pointer types** play a crucial role in representing incomplete or abstract data structures and memory addresses, respectively. Opaque types allow for the declaration of types whose details are not yet known, while pointer types are used to reference memory locations. These types are essential for working with complex data structures, dynamic memory, and modular code. In this section, we will explore the syntax, usage, and key characteristics of opaque and pointer types in LLVM IR.

3.3.2 Opaque Types

Opaque types in LLVM IR are used to declare types whose structure is not fully defined. They are often used in situations where the details of a type are not yet known or are intentionally hidden, such as in forward declarations or modular programming.

1. **Syntax**

   The syntax for defining an opaque type is:

   ```
   %opaque_type_name = type opaque
   ```

   where `%opaque_type_name` is the name of the opaque type.

2. **Example Usage**

   Here is an example of an opaque type and its usage:

```
%MyStruct = type opaque  ; Declare an opaque type

define void @use_opaque_type(ptr %p) {
  ; Assume %p is a pointer to an instance of %MyStruct
  ; Perform operations on %p (details of %MyStruct are not known)
  ret void
}
```

In this example:

- `%MyStruct` is an opaque type, meaning its internal structure is not defined.
- The function `@use_opaque_type` takes a pointer to an instance of `%MyStruct` but does not access its internal fields (since they are unknown).

3. **Key Points**

- Opaque types are useful for forward declarations or when the details of a type are not yet available.
- They can be replaced with a concrete type later in the IR using `type` definitions.
- Opaque types are often used in modular programming to hide implementation details.

3.3.3 Pointer Types

Pointer types in LLVM IR are used to represent memory addresses. They are essential for working with dynamic memory, arrays, and data structures. In modern LLVM IR, the `ptr` type is used as a generic pointer type, simplifying the representation of pointers.

1. **Syntax**

The syntax for a pointer type is:

```
ptr
```

In earlier versions of LLVM, pointers were explicitly typed (e.g., `i32*` for a pointer to an `i32`). However, modern LLVM IR uses the `ptr` type, which is a generic pointer type.

2. **Example Usage**

   Here is an example of pointer types and their usage:

```
define i32 @dereference_pointer(ptr %p) {
  %value = load i32, ptr %p  ; Load the value from the memory
   ↪  location pointed to by %p
  ret i32 %value
}

define void @store_value(ptr %p, i32 %value) {
  store i32 %value, ptr %p   ; Store the value at the memory location
   ↪  pointed to by %p
  ret void
}
```

   In this example:

   - `ptr` is used to represent a pointer to a memory location.
   - The `load` instruction reads a value from the memory location pointed to by `%p`.
   - The `store` instruction writes a value to the memory location pointed to by `%p`.

3. **Pointer Arithmetic**

Pointer arithmetic is performed using the `getelementptr` (GEP) instruction, which computes the address of a specific element within a data structure.

Example:

```
define i32 @access_array_element(ptr %arr, i32 %index) {
  %element_ptr = getelementptr inbounds i32, ptr %arr, i32 %index  ;
   ↪  Compute the address of the element at %index
  %element = load i32, ptr %element_ptr                            ;
   ↪  Load the element
  ret i32 %element
}
```

In this example:

- The `getelementptr` instruction computes the address of the element at the specified index in the array.
- The `load` instruction reads the value from the computed address.

4. **Key Points**

- The `ptr` type is a generic pointer type that simplifies the representation of pointers in LLVM IR.
- Pointers are used to reference memory locations, enabling dynamic memory access and manipulation.
- Pointer arithmetic is performed using the `getelementptr` instruction, which is essential for working with arrays, structs, and other data structures.

3.3.4 Combining Opaque and Pointer Types

Opaque and pointer types are often used together to represent abstract data structures and dynamic memory. Here is an example:

```
%MyStruct = type opaque  ; Declare an opaque type

define void @manipulate_opaque_struct(ptr %p) {
  ; Assume %p is a pointer to an instance of %MyStruct
  ; Perform operations on %p (details of %MyStruct are not known)
  ret void
}

define ptr @create_opaque_struct() {
  %instance = alloca i8, i64 16, align 8  ; Allocate memory for an
  ↪  instance of %MyStruct
  ret ptr %instance
}
```

In this example:

- `%MyStruct` is an opaque type, meaning its internal structure is not defined.
- The function `@manipulate_opaque_struct` takes a pointer to an instance of `%MyStruct` but does not access its internal fields.
- The function `@create_opaque_struct` allocates memory for an instance of `%MyStruct` and returns a pointer to it.

3.3.5 Example: Using Opaque and Pointer Types

Here is a complete example that demonstrates the use of opaque and pointer types in LLVM IR:

```
%MyStruct = type opaque  ; Declare an opaque type

define ptr @create_struct() {
  %instance = alloca i8, i64 16, align 8  ; Allocate memory for an
  ↪  instance of %MyStruct
  ret ptr %instance
}

define void @use_struct(ptr %p) {
  ; Assume %p is a pointer to an instance of %MyStruct
  ; Perform operations on %p (details of %MyStruct are not known)
  ret void
}

define i32 @main() {
  %struct_ptr = call ptr @create_struct()  ; Create an instance of
  ↪  %MyStruct
  call void @use_struct(ptr %struct_ptr)   ; Use the instance
  ret i32 0
}
```

In this example:

- `%MyStruct` is an opaque type, and its internal structure is not defined.
- The function `@create_struct` allocates memory for an instance of `%MyStruct` and returns a pointer to it.
- The function `@use_struct` takes a pointer to an instance of `%MyStruct` but does not access its internal fields.
- The `@main` function demonstrates the creation and use of an opaque struct.

3.3.6 Conclusion

Opaque and pointer types are powerful tools in LLVM IR for representing abstract data structures and memory addresses. Opaque types allow for the declaration of types whose details are not yet known, while pointer types enable dynamic memory access and manipulation. By understanding these types, beginners can write more flexible and modular LLVM IR code.

Chapter 4

Constants & Literals

4.1 Integer Constants (`i32 42`)

4.1.1 Introduction

Constants in LLVM IR are immutable values that are known at compile time. They are used to represent fixed values, such as numbers, strings, or addresses, and are essential for performing computations and initializing variables. In this section, we will focus on **integer constants**, which are used to represent whole numbers in LLVM IR. Understanding integer constants is fundamental for writing and analyzing LLVM IR code.

4.1.2 What Are Integer Constants?

Integer constants in LLVM IR are fixed values of integer types. They are represented using the syntax `<type> <value>`, where `<type>` is the integer type (e.g., `i32`, `i64`) and `<value>` is the constant value. For example, `i32 42` represents the integer value `42` of type `i32`.

1. **Syntax**

   The syntax for integer constants is:

   ```
   <type> <value>
   ```

   where:

   - `<type>` is the integer type (e.g., `i32`, `i64`, `i1`).
   - `<value>` is the constant value (e.g., `42`, `0`, `-1`).

2. **Example Usage**

   Here are some examples of integer constants:

   ```
   i32 42          ; 32-bit integer constant with value 42
   i64 -100        ; 64-bit integer constant with value -100
   i1 1            ; 1-bit integer constant with value 1 (true)
   i8 255          ; 8-bit integer constant with value 255
   ```

4.1.3 Types of Integer Constants

Integer constants can be of any integer type, including:

1. **Single-Bit Integers (`i1`)**:
   Used for boolean values, where `1` represents `true` and `0` represents `false`.

   Example:

   ```
   i1 1   ; true
   i1 0   ; false
   ```

2. **Small Integers (`i8`, `i16`)**:
   Used for small integers, such as byte-sized values or short integers.

   Example:

```
i8 255   ; 8-bit integer with maximum value
i16 1000 ; 16-bit integer with value 1000
```

3. **Standard Integers (`i32`, `i64`)**:
   Used for standard integers, such as those used in arithmetic operations or loop counters.

   Example:

```
i32 42   ; 32-bit integer with value 42
i64 -1   ; 64-bit integer with value -1
```

4. **Arbitrary-Width Integers (`iN`)**:
   LLVM IR supports integers of arbitrary width, where `N` can be any positive integer.

   Example:

```
i128 123456789012345678901234567890 ; 128-bit integer
```

4.1.4 Using Integer Constants in LLVM IR

Integer constants are used in various contexts in LLVM IR, including:

1. **Arithmetic Operations**

   Integer constants are often used as operands in arithmetic operations, such as addition, subtraction, multiplication, and division.

Example:

```
%sum = add i32 5, 10        ; Add two integer constants
%dif = sub i32 100, 50      ; Subtract two integer constants
%prod = mul i32 6, 7        ; Multiply two integer constants
%quot = sdiv i32 100, 25    ; Divide two integer constants
```

2. **Comparisons**

   Integer constants are used in comparison operations to compare values.

   Example:

```
%cmp = icmp eq i32 %x, 42  ; Compare %x with the integer constant 42
```

3. **Initialization**

   Integer constants are used to initialize variables, global variables, and arrays.

   Example:

```
@global_var = global i32 42  ; Initialize a global variable with the
↪  integer constant 42
%local_var = alloca i32, align 4
store i32 10, ptr %local_var ; Initialize a local variable with the
↪  integer constant 10
```

4. **Control Flow**

   Integer constants are used in control flow instructions, such as branches and switches.

   Example:

```
br i1 1, label %true, label %false  ; Branch based on a boolean
↪  constant
switch i32 %x, label %default [i32 1, label %case1, i32 2, label
↪  %case2]  ; Switch based on integer constants
```

4.1.5 Example: Using Integer Constants

Here is a complete example that demonstrates the use of integer constants in LLVM IR:

```
define i32 @compute(i32 %x) {
entry:
  %cmp = icmp eq i32 %x, 42  ; Compare %x with the integer constant 42
  br i1 %cmp, label %if_true, label %if_false

if_true:
  %result = add i32 %x, 10   ; Add 10 to %x
  ret i32 %result

if_false:
  %result2 = sub i32 %x, 5   ; Subtract 5 from %x
  ret i32 %result2
}
```

In this example:

- The integer constant `42` is used in a comparison operation.
- The integer constants `10` and `5` are used in arithmetic operations.

4.1.6 Key Points

- Integer constants are fixed values of integer types, represented using the syntax `<type> <value>`.
- They can be of any integer type, including `i1`, `i8`, `i16`, `i32`, `i64`, and arbitrary-width integers.
- Integer constants are used in arithmetic operations, comparisons, initialization, and control flow.
- They are essential for performing computations and representing fixed values in LLVM IR.

4.1.7 Conclusion

Integer constants are a fundamental part of LLVM IR, enabling the representation of fixed values and the performance of computations. By understanding how to use integer constants, beginners can write more expressive and efficient LLVM IR code. In the next section, we will explore **floating-point constants**, which are used to represent real numbers in LLVM IR.

4.2 Floating-Point Constants (`float 3.14`)

4.2.1 Introduction

Floating-point constants in LLVM IR are used to represent real numbers, such as `3.14` or `-0.5`. They are essential for performing computations involving fractional values, scientific calculations, and graphics processing. In this section, we will explore the syntax, types, and usage of floating-point constants in LLVM IR. Understanding floating-point constants is crucial for writing and analyzing LLVM IR code that involves real numbers.

4.2.2 What Are Floating-Point Constants?

Floating-point constants in LLVM IR are fixed values of floating-point types. They are represented using the syntax `<type> <value>`, where `<type>` is the floating-point type (e.g., `float`, `double`) and `<value>` is the constant value. For example, `float 3.14` represents the floating-point value `3.14` of type `float`.

1. **Syntax**

   The syntax for floating-point constants is:

   ```
   <type> <value>
   ```

   where:

   - `<type>` is the floating-point type (e.g., `float`, `double`).
   - `<value>` is the constant value (e.g., `3.14`, `-0.5`, `1.0e-10`).

2. **Example Usage**

   Here are some examples of floating-point constants:

```
float 3.14        ; 32-bit floating-point constant with value 3.14
double -0.5       ; 64-bit floating-point constant with value -0.5
float 1.0e-10     ; 32-bit floating-point constant with value 1.0 ×
↪  10¹
double 2.71828    ; 64-bit floating-point constant with value 2.71828
```

4.2.3 Types of Floating-Point Constants

Floating-point constants can be of the following types:

1. **Single-Precision (`float`)**:
   A 32-bit floating-point type, commonly used for real numbers in most applications.

   Example:

   ```
   float 3.14  ; 32-bit floating-point constant
   ```

2. **Double-Precision (`double`)**:
   A 64-bit floating-point type, used for higher precision and a wider range of values.

   Example:

   ```
   double 2.71828  ; 64-bit floating-point constant
   ```

3. **Half-Precision (`half`)**:
   A 16-bit floating-point type, used for reduced precision and memory savings, often in graphics and machine learning.

   Example:

```
half 1.0  ; 16-bit floating-point constant
```

4. **Quad-Precision (`fp128`)**:
   A 128-bit floating-point type, used for extremely high precision calculations.

   Example:

```
fp128 1.189731495357231765085759326628007 0e+4932  ; 128-bit
↪  floating-point constant
```

4.2.4 Using Floating-Point Constants in LLVM IR

Floating-point constants are used in various contexts in LLVM IR, including:

1. **Arithmetic Operations**

   Floating-point constants are often used as operands in arithmetic operations, such as addition, subtraction, multiplication, and division.

   Example:

```
%sum = fadd float 3.14, 2.71       ; Add two floating-point constants
%dif = fsub float 10.0, 5.5        ; Subtract two floating-point
↪  constants
%prod = fmul float 2.0, 1.5        ; Multiply two floating-point
↪  constants
%quot = fdiv float 10.0, 2.0       ; Divide two floating-point
↪  constants
```

2. **Comparisons**

   Floating-point constants are used in comparison operations to compare values.

   Example:

   llvm

   Copy

```
%cmp = fcmp oeq float %x, 3.14  ; Compare %x with the floating-point
↪  constant 3.14
```

3. **Initialization**

   Floating-point constants are used to initialize variables, global variables, and arrays.

   Example:

```
@global_var = global float 3.14  ; Initialize a global variable with
↪  the floating-point constant 3.14
%local_var = alloca float, align 4
store float 1.0, ptr %local_var  ; Initialize a local variable with
↪  the floating-point constant 1.0
```

4. **Control Flow**

   Floating-point constants are used in control flow instructions, such as branches and switches, when combined with comparison operations.

   Example:

   llvm

   Copy

```
%cmp = fcmp olt float %x, 0.0  ; Compare %x with the floating-point
↪  constant 0.0
br i1 %cmp, label %negative, label %non_negative
```

4.2.5 Example: Using Floating-Point Constants

Here is a complete example that demonstrates the use of floating-point constants in LLVM IR:

```
define float @compute(float %x) {
entry:
  %cmp = fcmp oeq float %x, 3.14  ; Compare %x with the floating-point
  ↪  constant 3.14
  br i1 %cmp, label %if_true, label %if_false

if_true:
  %result = fadd float %x, 1.0   ; Add 1.0 to %x
  ret float %result

if_false:
  %result2 = fsub float %x, 0.5  ; Subtract 0.5 from %x
  ret float %result2
}
```

In this example:

- The floating-point constant `3.14` is used in a comparison operation.
- The floating-point constants `1.0` and `0.5` are used in arithmetic operations.

4.2.6 Key Points

- Floating-point constants are fixed values of floating-point types, represented using the syntax `<type> <value>`.
- They can be of types `float`, `double`, `half`, or `fp128`, depending on the required precision.
- Floating-point constants are used in arithmetic operations, comparisons, initialization, and control flow.
- They are essential for performing computations involving real numbers in LLVM IR.

4.2.7 Conclusion

Floating-point constants are a fundamental part of LLVM IR, enabling the representation of real numbers and the performance of computations involving fractional values. By understanding how to use floating-point constants, beginners can write more expressive and efficient LLVM IR code. In the next section, we will explore **string constants**, which are used to represent text data in LLVM IR.

4.3 String Constants (`c"Hello\00"`)

4.3.1 Introduction

String constants in LLVM IR are used to represent text data, such as messages, file paths, or other character sequences. They are essential for tasks like printing output, logging, and interacting with external libraries that require string inputs. In this section, we will explore the syntax, representation, and usage of string constants in LLVM IR. Understanding string constants is crucial for working with text data in LLVM IR.

4.3.2 What Are String Constants?

String constants in LLVM IR are sequences of characters stored in memory. They are typically represented as arrays of `i8` (8-bit integers) and are often null-terminated, meaning they end with a `\00` (null character) to indicate the end of the string.

1. **Syntax**

   The syntax for string constants is:

   ```
   c"<string_content>\00"
   ```

   where:

   - `c` indicates that the string is a constant.
   - `<string_content>` is the sequence of characters in the string.
   - `\00` is the null terminator, which marks the end of the string.

2. **Example Usage**

   Here are some examples of string constants:

```
@hello = constant [6 x i8] c"Hello\00"  ; A string constant "Hello"
@path = constant [12 x i8] c"/usr/bin\00" ; A string constant
↪  "/usr/bin"
@empty = constant [1 x i8] c"\00"        ; An empty string
```

4.3.3 Representation of String Constants

String constants are represented as arrays of `i8` (8-bit integers) in LLVM IR. Each character in the string is stored as an `i8` value, and the string is null-terminated to indicate its end.

1. **Array Representation**

   A string constant is typically declared as a global constant array of `i8`. For example, the string `"Hello"` is represented as:

   ```
   @hello = constant [6 x i8] c"Hello\00"
   ```

   Here:

   - `[6 x i8]` indicates an array of 6 `i8` elements.
   - `c"Hello\00"` is the string content, where `\00` is the null terminator.

2. **Null Termination**

   The null terminator (`\00`) is essential for C-style strings, as it indicates the end of the string. Without it, functions like `printf` or `strlen` would not know where the string ends.

   Example:

```
@hello = constant [6 x i8] c"Hello\00"  ; Correctly null-terminated
@invalid = constant [5 x i8] c"Hello"    ; Missing null terminator
↪  (incorrect)
```

4.3.4 Using String Constants in LLVM IR

String constants are used in various contexts in LLVM IR, including:

1. **Global Variables**

   String constants are often stored in global variables for easy access throughout the program.

   Example:

```
@message = constant [13 x i8] c"Hello, World\00"  ; Global string
↪  constant
```

2. **Function Calls**

   String constants are frequently passed as arguments to functions, such as `printf` or custom functions.

   Example:

```
declare i32 @printf(ptr, ...)  ; Declare the printf function

define i32 @main() {
  %str = getelementptr inbounds [13 x i8], ptr @message, i32 0, i32 0
  ↪  ; Get pointer to the string
  call i32 (ptr, ...) @printf(ptr %str)  ; Call printf with the
  ↪  string
```

```
  ret i32 0
}
```

In this example:

- `@message` is a global string constant.
- The `getelementptr` instruction computes the address of the first character in the string.
- The `printf` function is called with the string as an argument.

3. **Initialization**

   String constants can be used to initialize arrays or structures.

   Example:

```
%MyStruct = type { ptr, i32 }

@my_struct = global %MyStruct { ptr @message, i32 13 }  ; Initialize
↪  a struct with a string constant
```

4.3.5 Example: Using String Constants

Here is a complete example that demonstrates the use of string constants in LLVM IR:

```
@hello = constant [6 x i8] c"Hello\00"  ; Global string constant
@world = constant [6 x i8] c"World\00"  ; Global string constant

declare i32 @printf(ptr, ...)  ; Declare the printf function
```

```
define i32 @main() {
  %hello_ptr = getelementptr inbounds [6 x i8], ptr @hello, i32 0, i32 0  ;
  ↪  Get pointer to "Hello"
  %world_ptr = getelementptr inbounds [6 x i8], ptr @world, i32 0, i32 0  ;
  ↪  Get pointer to "World"

  call i32 (ptr, ...) @printf(ptr %hello_ptr)  ; Print "Hello"
  call i32 (ptr, ...) @printf(ptr %world_ptr)  ; Print "World"

  ret i32 0
}
```

In this example:

- Two global string constants, `@hello` and `@world`, are defined.
- The `getelementptr` instruction is used to compute pointers to the strings.
- The `printf` function is called to print the strings.

4.3.6 Key Points

- String constants in LLVM IR are represented as arrays of `i8` and are typically null-terminated.
- They are declared using the syntax `c"<string_content>\00"`.
- String constants are often stored in global variables and passed as arguments to functions.
- The null terminator (`\00`) is essential for C-style strings to indicate the end of the string.

4.3.7 Conclusion

String constants are a fundamental part of LLVM IR, enabling the representation and manipulation of text data. By understanding how to use string constants, beginners can write LLVM IR code that interacts with strings effectively, such as printing messages or handling file paths. In the next section, we will explore **global constants**, which are used to represent immutable values that are accessible throughout a module.

4.4 Global and Local Constants

4.4.1 Introduction

Constants in LLVM IR are immutable values that are known at compile time. They can be either **global** or **local**, depending on their scope and lifetime. Global constants are accessible throughout the entire module, while local constants are limited to a specific function or basic block. In this section, we will explore the syntax, usage, and key characteristics of global and local constants in LLVM IR. Understanding these constants is essential for writing efficient and maintainable LLVM IR code.

4.4.2 Global Constants

Global constants are immutable values that are accessible throughout the entire LLVM module. They are typically used for values that need to be shared across multiple functions or for large data structures that should not be duplicated.

1. **Syntax**

   The syntax for defining a global constant is:

   ```
   @<name> = constant <type> <value>
   ```

   where:

   - `@<name>` is the name of the global constant.
   - `<type>` is the type of the constant (e.g., `i32`, `float`, `[N x i8]`).
   - `<value>` is the constant value.

2. **Example Usage**

   Here are some examples of global constants:

```
@int_constant = constant i32 42           ; Global integer constant
@float_constant = constant float 3.14     ; Global floating-point
↪  constant
@string_constant = constant [13 x i8] c"Hello, World\00"  ; Global
↪  string constant
@array_constant = constant [3 x i32] [i32 1, i32 2, i32 3]  ; Global
↪  array constant
```

3. **Key Points**

   - Global constants are immutable and cannot be modified at runtime.
   - They are stored in the global memory space and are accessible throughout the module.
   - Global constants are often used for shared data, such as lookup tables, configuration values, or string literals.

4.4.3 Local Constants

Local constants are immutable values that are limited to a specific function or basic block. They are typically used for values that are only needed within a small scope, such as loop counters or intermediate results.

1. **Syntax**

   Local constants are defined using the `constant` keyword within a function or basic block. The syntax is:

```
<name> = constant <type> <value>
```

where:

- <name> is the name of the local constant.
- <type> is the type of the constant.
- <value> is the constant value.

2. **Example Usage**

   Here are some examples of local constants:

```
define i32 @example() {
  %local_int = constant i32 10          ; Local integer constant
  %local_float = constant float 2.71    ; Local floating-point
  ↪  constant
  %local_array = constant [2 x i32] [i32 1, i32 2]  ; Local array
  ↪  constant

  %sum = add i32 %local_int, 5          ; Use the local constant in
  ↪  an operation
  ret i32 %sum
}
```

3. **Key Points**

   - Local constants are immutable and cannot be modified at runtime.
   - They are limited to the scope of the function or basic block in which they are defined.
   - Local constants are often used for temporary values or loop counters.

4.4.4 Differences Between Global and Local Constants

Feature	Global Constants	Local Constants
Scope	Accessible throughout the module	Limited to a function or basic block
Lifetime	Persistent for the entire program	Limited to the scope of definition
Memory Location	Stored in global memory	Stored in stack or registers
Usage	Shared data, lookup tables, strings	Temporary values, loop counters

4.4.5 Example: Using Global and Local Constants

Here is a complete example that demonstrates the use of global and local constants in LLVM IR:

```
@global_int = constant i32 42          ; Global integer constant
@global_string = constant [13 x i8] c"Hello, World\00"  ; Global string
↪  constant

define i32 @main() {
  %local_int = constant i32 10          ; Local integer constant
  %local_float = constant float 3.14    ; Local floating-point constant

  %sum = add i32 @global_int, %local_int  ; Add global and local constants
  %result = sitofp i32 %sum to float      ; Convert sum to float
  %final_result = fadd float %result, %local_float  ; Add local float
  ↪  constant
```

```
  ; Print the global string
  %str_ptr = getelementptr inbounds [13 x i8], ptr @global_string, i32 0,
  ↪  i32 0
  call i32 (ptr, ...) @printf(ptr %str_ptr)

  ret i32 0
}

declare i32 @printf(ptr, ...)  ; Declare the printf function
```

In this example:

- `@global_int` and `@global_string` are global constants.
- `%local_int` and `%local_float` are local constants.
- The global and local constants are used in arithmetic operations and function calls.

4.4.6 Key Points

- **Global constants** are accessible throughout the module and are stored in global memory.
- **Local constants** are limited to a specific function or basic block and are stored in stack or registers.
- Both global and local constants are immutable and cannot be modified at runtime.
- Global constants are often used for shared data, while local constants are used for temporary values.

4.4.7 Conclusion

Global and local constants are essential tools in LLVM IR for representing immutable values. By understanding their syntax, usage, and differences, beginners can write more efficient and maintainable LLVM IR code. In the next section, we will explore **metadata**, which is used to attach additional information to LLVM IR constructs for debugging, optimization, and analysis.

Chapter 5

Memory Management

5.1 Stack Allocation (`alloca`)

5.1.1 Introduction

Memory management is a critical aspect of any programming language or intermediate representation (IR). In LLVM IR, memory can be allocated on the stack or the heap. Stack allocation is managed using the `alloca` instruction, which allocates memory on the stack for local variables. This section will provide a detailed explanation of stack allocation in LLVM IR, focusing on the `alloca` instruction, its syntax, usage, and key characteristics. Understanding stack allocation is essential for writing efficient and correct LLVM IR code.

5.1.2 What is Stack Allocation?

Stack allocation refers to the process of reserving memory on the stack for local variables. The stack is a region of memory that is automatically managed by the compiler and is used for storing temporary data, such as function parameters, local variables, and return addresses.

Stack-allocated memory is automatically deallocated when the function returns, making it efficient and easy to use.
In LLVM IR, stack allocation is performed using the `alloca` instruction. This instruction reserves a block of memory on the stack and returns a pointer to the allocated memory.

5.1.3 The `alloca` Instruction

The `alloca` instruction is used to allocate memory on the stack. It reserves a block of memory of a specified type and returns a pointer to the allocated memory.

1. **Syntax**

   The syntax for the `alloca` instruction is:

   ```
   <result> = alloca <type> [, <num_elements>] [, align <alignment>]
   ```

   where:

   - `<result>` is the pointer to the allocated memory.
   - `<type>` is the type of the elements to be allocated.
   - `<num_elements>` (optional) is the number of elements to allocate (default is 1).
   - `align <alignment>` (optional) specifies the alignment of the allocated memory.

2. **Example Usage**

   Here are some examples of the `alloca` instruction:

```
%x = alloca i32                              ; Allocate memory for a
↪  single i32
%y = alloca i32, align 4                     ; Allocate memory for a
↪  single i32 with 4-byte alignment
%arr = alloca [10 x i32]                     ; Allocate memory for an
↪  array of 10 i32 elements
%struct_ptr = alloca %MyStruct               ; Allocate memory for a
↪  struct of type %MyStruct
```

5.1.4 Using `alloca` for Local Variables

The `alloca` instruction is typically used to allocate memory for local variables within a function. The allocated memory is automatically deallocated when the function returns, making it ideal for temporary storage.

1. **Example: Allocating and Using Local Variables**

   Here is an example of using `alloca` to allocate and use local variables:

```
define i32 @example() {
  %x = alloca i32, align 4                   ; Allocate memory for a
  ↪  local i32 variable
  store i32 42, ptr %x                       ; Store the value 42 in the
  ↪  allocated memory
  %y = load i32, ptr %x                      ; Load the value from the
  ↪  allocated memory
  %sum = add i32 %y, 10                      ; Perform an operation using
  ↪  the loaded value
  ret i32 %sum
}
```

In this example:

- The `alloca` instruction allocates memory for a local `i32` variable.
- The `store` instruction stores the value `42` in the allocated memory.
- The `load` instruction retrieves the value from the allocated memory.
- The `add` instruction performs an operation using the loaded value.

5.1.5 Allocating Arrays and Structs

The `alloca` instruction can also be used to allocate memory for arrays and structs. This is useful for working with complex data structures.

1. **Example: Allocating an Array**

   Here is an example of using `alloca` to allocate memory for an array:

```
define void @array_example() {
  %arr = alloca [10 x i32], align 4      ; Allocate memory for an
  ↪  array of 10 i32 elements
  %first_element_ptr = getelementptr inbounds [10 x i32], ptr %arr,
  ↪  i32 0, i32 0  ; Get pointer to the first element
  store i32 100, ptr %first_element_ptr  ; Store the value 100 in the
  ↪  first element
  ret void
}
```

   In this example:

   - The `alloca` instruction allocates memory for an array of 10 `i32` elements.
   - The `getelementptr` instruction computes the address of the first element in the array.

- The `store` instruction stores the value `100` in the first element.

2. **Example: Allocating a Struct**

   Here is an example of using `alloca` to allocate memory for a struct:

```
%MyStruct = type { i32, float }

define void @struct_example() {
  %s = alloca %MyStruct, align 8          ; Allocate memory for a
  ↪  struct of type %MyStruct
  %int_field_ptr = getelementptr inbounds %MyStruct, ptr %s, i32 0,
  ↪  i32 0  ; Get pointer to the first field (i32)
  %float_field_ptr = getelementptr inbounds %MyStruct, ptr %s, i32 0,
  ↪  i32 1 ; Get pointer to the second field (float)
  store i32 42, ptr %int_field_ptr        ; Store the value 42 in the
  ↪  first field
  store float 3.14, ptr %float_field_ptr ; Store the value 3.14 in
  ↪  the second field
  ret void
}
```

   In this example:

   - The `alloca` instruction allocates memory for a struct of type `%MyStruct`.
   - The `getelementptr` instruction computes the addresses of the struct fields.
   - The `store` instruction stores values in the struct fields.

5.1.6 Alignment in `alloca`

The `align` keyword can be used with the `alloca` instruction to specify the alignment of the allocated memory. Proper alignment can improve performance by ensuring that memory

accesses are aligned with the hardware's requirements.

1. **Example: Specifying Alignment**

   Here is an example of using the `align` keyword with `alloca`:

```
define void @alignment_example() {
  %x = alloca i32, align 8                ; Allocate memory for an i32
   ↪  with 8-byte alignment
  store i32 42, ptr %x                    ; Store the value 42 in the
   ↪  allocated memory
  ret void
}
```

   In this example:

   - The `alloca` instruction allocates memory for an `i32` with 8-byte alignment.

5.1.7 Key Points

- The `alloca` instruction is used to allocate memory on the stack for local variables.
- It returns a pointer to the allocated memory, which can be used with `load` and `store` instructions.
- The `alloca` instruction can allocate memory for single values, arrays, and structs.
- The `align` keyword can be used to specify the alignment of the allocated memory.
- Stack-allocated memory is automatically deallocated when the function returns.

5.1.8 Conclusion

Stack allocation using the `alloca` instruction is a fundamental concept in LLVM IR. It provides an efficient and easy-to-use mechanism for managing local variables and temporary data. By understanding how to use `alloca`, beginners can write LLVM IR code that effectively manages memory on the stack.

5.2 Load and Store Instructions (`load`, `store`)

5.2.1 Introduction

Memory management in LLVM IR involves not only allocating memory but also reading from and writing to memory. The `load` and `store` instructions are fundamental for interacting with memory in LLVM IR. The `load` instruction reads a value from a memory location, while the `store` instruction writes a value to a memory location. Understanding these instructions is crucial for working with variables, arrays, structs, and other data structures in LLVM IR. This section will provide a detailed explanation of the `load` and `store` instructions, their syntax, usage, and key characteristics.

5.2.2 The `load` Instruction

The `load` instruction is used to read a value from a memory location. It takes a pointer to a memory location and returns the value stored at that location.

1. **Syntax**

   The syntax for the `load` instruction is:

   ```
   <result> = load <type>, ptr <pointer> [, align <alignment>]
   ```

   where:

   - `<result>` is the value loaded from memory.
   - `<type>` is the type of the value to be loaded.
   - `<pointer>` is the pointer to the memory location.
   - `align <alignment>` (optional) specifies the alignment of the memory access.

2. **Example Usage**

   Here are some examples of the `load` instruction:

```
%x = load i32, ptr %ptr                 ; Load an i32 value from the
↪  memory location pointed to by %ptr
%y = load float, ptr %float_ptr, align 4 ; Load a float value with
↪  4-byte alignment
%z = load i64, ptr %arr_ptr             ; Load an i64 value from an
↪  array element
```

3. **Key Points**

   - The `load` instruction reads a value from a memory location specified by a pointer.
   - The type of the loaded value must match the type of the memory location.
   - The `align` keyword can be used to specify the alignment of the memory access, which can improve performance.

5.2.3 The `store` Instruction

The `store` instruction is used to write a value to a memory location. It takes a value and a pointer to a memory location and stores the value at that location.

1. **Syntax**

   The syntax for the `store` instruction is:

```
store <type> <value>, ptr <pointer> [, align <alignment>]
```

   where:

- `<type>` is the type of the value to be stored.
- `<value>` is the value to be stored.
- `<pointer>` is the pointer to the memory location.
- `align <alignment>` (optional) specifies the alignment of the memory access.

2. **Example Usage**

   Here are some examples of the `store` instruction:

```
store i32 42, ptr %ptr                    ; Store the value 42 at the
↪  memory location pointed to by %ptr
store float 3.14, ptr %float_ptr, align 4 ; Store the value 3.14 with
↪  4-byte alignment
store i64 %val, ptr %arr_ptr              ; Store an i64 value in an
↪  array element
```

3. **Key Points**

   - The `store` instruction writes a value to a memory location specified by a pointer.
   - The type of the stored value must match the type of the memory location.
   - The `align` keyword can be used to specify the alignment of the memory access, which can improve performance.

5.2.4 Using `load` and `store` with Local Variables

The `load` and `store` instructions are commonly used with local variables allocated on the stack using the `alloca` instruction.

1. **Example: Loading and Storing Local Variables**

Here is an example of using `load` and `store` with local variables:

```
define i32 @example() {
  %x = alloca i32, align 4                ; Allocate memory for a
   ↪  local i32 variable
  store i32 42, ptr %x                    ; Store the value 42 in the
   ↪  allocated memory
  %y = load i32, ptr %x                   ; Load the value from the
   ↪  allocated memory
  %sum = add i32 %y, 10                   ; Perform an operation using
   ↪  the loaded value
  ret i32 %sum
}
```

In this example:

- The `alloca` instruction allocates memory for a local `i32` variable.
- The `store` instruction stores the value `42` in the allocated memory.
- The `load` instruction retrieves the value from the allocated memory.
- The `add` instruction performs an operation using the loaded value.

5.2.5 Using `load` and `store` with Arrays

The `load` and `store` instructions are also used to access elements of arrays. The `getelementptr` instruction is often used to compute the address of array elements.

1. **Example: Loading and Storing Array Elements**

   Here is an example of using `load` and `store` with arrays:

```
define void @array_example() {
  %arr = alloca [10 x i32], align 4      ; Allocate memory for an
   ↪  array of 10 i32 elements
  %first_element_ptr = getelementptr inbounds [10 x i32], ptr %arr,
   ↪  i32 0, i32 0  ; Get pointer to the first element
  store i32 100, ptr %first_element_ptr  ; Store the value 100 in the
   ↪  first element
  %loaded_value = load i32, ptr %first_element_ptr  ; Load the value
   ↪  from the first element
  ret void
}
```

In this example:

- The `alloca` instruction allocates memory for an array of 10 `i32` elements.
- The `getelementptr` instruction computes the address of the first element in the array.
- The `store` instruction stores the value `100` in the first element.
- The `load` instruction retrieves the value from the first element.

5.2.6 Using `load` and `store` with Structs

The `load` and `store` instructions are used to access fields of structs. The `getelementptr` instruction is often used to compute the address of struct fields.

1. **Example: Loading and Storing Struct Fields**

   Here is an example of using `load` and `store` with structs:

```
%MyStruct = type { i32, float }

define void @struct_example() {
  %s = alloca %MyStruct, align 8          ; Allocate memory for a
   ↪  struct of type %MyStruct
  %int_field_ptr = getelementptr inbounds %MyStruct, ptr %s, i32 0,
   ↪  i32 0  ; Get pointer to the first field (i32)
  %float_field_ptr = getelementptr inbounds %MyStruct, ptr %s, i32 0,
   ↪  i32 1 ; Get pointer to the second field (float)
  store i32 42, ptr %int_field_ptr        ; Store the value 42 in the
   ↪  first field
  store float 3.14, ptr %float_field_ptr ; Store the value 3.14 in
   ↪  the second field
  %loaded_int = load i32, ptr %int_field_ptr  ; Load the value from
   ↪  the first field
  %loaded_float = load float, ptr %float_field_ptr  ; Load the value
   ↪  from the second field
  ret void
}
```

In this example:

- The `alloca` instruction allocates memory for a struct of type `%MyStruct`.
- The `getelementptr` instruction computes the addresses of the struct fields.
- The `store` instruction stores values in the struct fields.
- The `load` instruction retrieves values from the struct fields.

5.2.7 Alignment in `load` and `store`

The `align` keyword can be used with the `load` and `store` instructions to specify the alignment of the memory access. Proper alignment can improve performance by ensuring

that memory accesses are aligned with the hardware's requirements.

1. **Example: Specifying Alignment**

   Here is an example of using the `align` keyword with `load` and `store`:

```
define void @alignment_example() {
  %x = alloca i32, align 8                ; Allocate memory for an i32
   ↪  with 8-byte alignment
  store i32 42, ptr %x, align 8           ; Store the value 42 with
   ↪  8-byte alignment
  %y = load i32, ptr %x, align 8          ; Load the value with 8-byte
   ↪  alignment
  ret void
}
```

   In this example:

   - The `alloca` instruction allocates memory for an `i32` with 8-byte alignment.
   - The `store` instruction stores the value `42` with 8-byte alignment.
   - The `load` instruction retrieves the value with 8-byte alignment.

5.2.8 Key Points

- The `load` instruction reads a value from a memory location specified by a pointer.
- The `store` instruction writes a value to a memory location specified by a pointer.
- Both `load` and `store` instructions can be used with local variables, arrays, and structs.
- The `align` keyword can be used to specify the alignment of memory accesses, improving performance.

5.2.9 Conclusion

The `load` and `store` instructions are fundamental for interacting with memory in LLVM IR. By understanding how to use these instructions, beginners can effectively manage and manipulate data in LLVM IR.

5.3 Global Variables (`@globalVar = global i32 42`)

5.3.1 Introduction

Global variables in LLVM IR are variables that are accessible throughout the entire module. They are stored in the global memory space and are typically used for data that needs to be shared across multiple functions or persists for the entire lifetime of the program. In this section, we will explore the syntax, usage, and key characteristics of global variables in LLVM IR. Understanding global variables is essential for managing shared data and maintaining state across function calls.

5.3.2 What Are Global Variables?

Global variables are variables that are declared outside of any function and are accessible to all functions within the module. They are stored in the global memory space, which is distinct from the stack or heap. Global variables can be initialized with constant values or left uninitialized.
In LLVM IR, global variables are declared using the `global` keyword. They are prefixed with the `@` symbol to distinguish them from local variables, which are prefixed with `%`.

5.3.3 Syntax of Global Variables

The syntax for declaring a global variable is:

```
@<name> = [linkage] [visibility] [thread_local] [unnamed_addr]
 ↪  [addrspace(<num>)] <type> [<initializer>] [, section "<section_name>"]
 ↪  [, align <alignment>]
```

where:

- `@<name>` is the name of the global variable.
- `linkage` (optional) specifies the linkage type (e.g., `private`, `internal`, `external`).
- `visibility` (optional) specifies the visibility (e.g., `default`, `hidden`, `protected`).
- `thread_local` (optional) indicates that the variable is thread-local.
- `unnamed_addr` (optional) indicates that the address of the variable is not significant.
- `addrspace(<num>)` (optional) specifies the address space for the variable.
- `<type>` is the type of the global variable.
- `<initializer>` (optional) is the initial value of the global variable.
- `section "<section_name>"` (optional) specifies the section in the object file where the variable should be placed.
- `align <alignment>` (optional) specifies the alignment of the variable.

1. **Example Usage**

   Here are some examples of global variable declarations:

```
@global_int = global i32 42                ; Global integer variable
↪  initialized to 42
@global_float = global float 3.14          ; Global float variable
↪  initialized to 3.14
@global_array = global [3 x i32] [i32 1, i32 2, i32 3]  ; Global array
↪  variable
```

```
@global_string = global [13 x i8] c"Hello, World\00"  ; Global string
↪  variable
@uninitialized_global = global i32       ; Uninitialized global
↪  integer variable
```

5.3.4 Using Global Variables

Global variables can be accessed and modified by any function within the module. They are typically used for shared data, configuration values, or constants that need to be accessed globally.

1. **Example: Accessing and Modifying Global Variables**

   Here is an example of accessing and modifying global variables:

```
@global_int = global i32 42              ; Global integer variable
↪  initialized to 42

define i32 @get_global() {
  %value = load i32, ptr @global_int     ; Load the value of the
  ↪  global variable
  ret i32 %value
}

define void @set_global(i32 %new_value) {
  store i32 %new_value, ptr @global_int  ; Store a new value in the
  ↪  global variable
  ret void
}
```

   In this example:

- The `@global_int` global variable is initialized to `42`.
- The `get_global` function loads the value of `@global_int` and returns it.
- The `set_global` function stores a new value in `@global_int`.

5.3.5 Linkage and Visibility

Global variables can have different linkage and visibility attributes, which control how they are shared across modules and how they are optimized.

1. **Linkage Types**

   - `private`: The global variable is only accessible within the current module.
   - `internal`: Similar to `private`, but the variable cannot be accessed by other modules.
   - `external`: The global variable is accessible from other modules.
   - `weak`: The global variable can be overridden by another definition in a different module.

2. **Visibility Types**

   - `default`: The global variable is visible to other modules.
   - `hidden`: The global variable is not visible to other modules.
   - `protected`: The global variable is visible to other modules but cannot be overridden.

3. **Example: Specifying Linkage and Visibility**

   Here is an example of specifying linkage and visibility for a global variable:

```
@private_global = private global i32 42  ; Private global variable
@internal_global = internal global i32 42  ; Internal global variable
@external_global = external global i32 42  ; External global variable
@weak_global = weak global i32 42        ; Weak global variable
```

5.3.6 Thread-Local Global Variables

Thread-local global variables are variables that have a separate instance for each thread. They are declared using the `thread_local` keyword.

1. **Example: Thread-Local Global Variable**

   Here is an example of a thread-local global variable:

```
@thread_local_global = thread_local global i32 42  ; Thread-local
↪  global variable
```

5.3.7 Address Spaces

Global variables can be placed in different address spaces, which are used to represent different memory regions (e.g., global memory, local memory, constant memory).

1. **Example: Specifying Address Space**

   Here is an example of specifying an address space for a global variable:

```
@global_in_addrspace = addrspace(1) global i32 42  ; Global variable
↪  in address space 1
```

5.3.8 Alignment and Sections

Global variables can be aligned and placed in specific sections of the object file using the `align` and `section` keywords.

1. **Example: Specifying Alignment and Section**

   Here is an example of specifying alignment and section for a global variable:

```
@aligned_global = global i32 42, align 8  ; Global variable with
↪  8-byte alignment
@section_global = global i32 42, section ".data"  ; Global variable
↪  placed in the .data section
```

5.3.9 Example: Comprehensive Use of Global Variables

Here is a complete example that demonstrates the use of global variables with various attributes:

```
@global_int = global i32 42               ; Global integer variable
↪  initialized to 42
@global_array = global [3 x i32] [i32 1, i32 2, i32 3]  ; Global array
↪  variable
@private_global = private global i32 42  ; Private global variable
@thread_local_global = thread_local global i32 42  ; Thread-local global
↪  variable
@aligned_global = global i32 42, align 8  ; Global variable with 8-byte
↪  alignment

define i32 @get_global() {
  %value = load i32, ptr @global_int     ; Load the value of the global
  ↪  variable
```

```
  ret i32 %value
}

define void @set_global(i32 %new_value) {
  store i32 %new_value, ptr @global_int  ; Store a new value in the global
  ↪  variable
  ret void
}
```

In this example:

- `@global_int` is a global integer variable initialized to `42`.
- `@global_array` is a global array variable.
- `@private_global` is a private global variable.
- `@thread_local_global` is a thread-local global variable.
- `@aligned_global` is a global variable with 8-byte alignment.
- The `get_global` function loads the value of `@global_int`.
- The `set_global` function stores a new value in `@global_int`.

5.3.10 Key Points

- Global variables are accessible throughout the entire module and are stored in the global memory space.
- They are declared using the `global` keyword and are prefixed with `@`.
- Global variables can have linkage, visibility, thread-local, and address space attributes.

- They can be initialized with constant values or left uninitialized.
- Global variables are useful for shared data, configuration values, and constants.

5.3.11 Conclusion

Global variables are a powerful feature of LLVM IR, enabling the management of shared data and state across functions. By understanding how to declare, initialize, and use global variables, beginners can write more modular and efficient LLVM IR code.

5.4 Address Spaces and Pointers

5.4.1 Introduction

In LLVM IR, memory is organized into different **address spaces**, which represent distinct regions of memory with unique properties. Address spaces are used to model various memory hierarchies, such as global memory, local memory, constant memory, and private memory. Pointers in LLVM IR can reference memory in specific address spaces, enabling fine-grained control over memory access and optimization. In this section, we will explore the concept of address spaces, how they are used in LLVM IR, and how pointers interact with them. Understanding address spaces and pointers is essential for writing efficient and correct LLVM IR code, especially for targets like GPUs and specialized hardware.

5.4.2 What Are Address Spaces?

Address spaces are logical divisions of memory that represent different regions with distinct properties. For example:

- **Address Space 0**: Default address space (generic memory).
- **Address Space 1**: Global memory (used for GPU programming).
- **Address Space 2**: Local memory (used for GPU programming).
- **Address Space 3**: Constant memory (read-only memory).

Each address space has its own characteristics, such as access speed, scope, and visibility. By using address spaces, LLVM IR can optimize memory access patterns and ensure correctness for specific hardware architectures.

5.4.3 Syntax for Address Spaces

In LLVM IR, address spaces are specified using the `addrspace(<num>)` keyword, where `<num>` is the address space identifier. Pointers and global variables can be associated with specific address spaces.

1. **Syntax for Pointers in Address Spaces**

   The syntax for a pointer in a specific address space is:

   ```
   ptr addrspace(<num>)
   ```

   where `<num>` is the address space identifier.

2. **Syntax for Global Variables in Address Spaces**

   The syntax for a global variable in a specific address space is:

   ```
   @<name> = addrspace(<num>) global <type> <initializer>
   ```

   where `<num>` is the address space identifier.

5.4.4 Example: Using Address Spaces

Here are some examples of using address spaces in LLVM IR:

1. **Global Variable in Address Space 1**

   ```
   @global_var = addrspace(1) global i32 42  ; Global variable in
   ↪  address space 1
   ```

2. **Pointer to Address Space 2**

```
%local_ptr = alloca i32, addrspace(2)     ; Pointer to local memory
↪  (address space 2)
```

3. **Function Parameter in Address Space 3**

```
define void @example(ptr addrspace(3) %ptr) {
  %value = load i32, ptr addrspace(3) %ptr  ; Load a value from
  ↪  constant memory (address space 3)
  ret void
}
```

5.4.5 Pointers and Address Spaces

Pointers in LLVM IR can reference memory in specific address spaces. This allows for fine-grained control over memory access and enables optimizations tailored to the properties of each address space.

1. **Pointer Types in Address Spaces**

   A pointer type in LLVM IR can specify an address space. For example:

   - `ptr addrspace(0)`: Pointer to the default address space (generic memory).
   - `ptr addrspace(1)`: Pointer to global memory.
   - `ptr addrspace(2)`: Pointer to local memory.

2. **Example: Pointer to Global Memory**

   Here is an example of using a pointer to global memory:

```
@global_var = addrspace(1) global i32 42  ; Global variable in
↪  address space 1

define i32 @get_global() {
  %ptr = getelementptr i32, ptr addrspace(1) @global_var, i32 0  ;
  ↪  Get pointer to global variable
  %value = load i32, ptr addrspace(1) %ptr  ; Load the value from
  ↪  global memory
  ret i32 %value
}
```

In this example:

- `@global_var` is a global variable in address space 1 (global memory).
- The `getelementptr` instruction computes the address of the global variable.
- The `load` instruction reads the value from global memory.

5.4.6 Address Space Casting

LLVM IR provides the `addrspacecast` instruction to convert pointers between different address spaces. This is useful when interacting with functions or memory regions that expect pointers in specific address spaces.

1. **Syntax for Address Space Casting**

   The syntax for the `addrspacecast` instruction is:

   ```
   <result> = addrspacecast <type> <value> to <type2>
   ```

   where:

- `<result>` is the pointer in the new address space.
- `<type>` is the original pointer type.
- `<value>` is the original pointer.
- `<type2>` is the target pointer type.

2. **Example: Address Space Casting**

   Here is an example of casting a pointer from address space 1 to address space 0:

```
@global_var = addrspace(1) global i32 42  ; Global variable in
↪  address space 1

define i32 @get_global() {
  %ptr = getelementptr i32, ptr addrspace(1) @global_var, i32 0  ;
  ↪  Get pointer to global variable
  %generic_ptr = addrspacecast ptr addrspace(1) %ptr to ptr  ; Cast
  ↪  to default address space
  %value = load i32, ptr %generic_ptr  ; Load the value from generic
  ↪  memory
  ret i32 %value
}
```

   In this example:

   - The `addrspacecast` instruction converts the pointer from address space 1 to the default address space (0).
   - The `load` instruction reads the value from the generic memory address space.

5.4.7 Practical Use Cases for Address Spaces

Address spaces are particularly useful in the following scenarios:

1. **GPU Programming**

   In GPU programming, different memory regions (e.g., global, local, constant) are mapped to distinct address spaces. Using address spaces allows the compiler to optimize memory access patterns and ensure correctness.

   Example:

```
@global_mem = addrspace(1) global i32 42  ; Global memory (address
↪  space 1)
@local_mem = addrspace(3) global i32 0    ; Local memory (address
↪  space 3)

define void @gpu_kernel(ptr addrspace(1) %input, ptr addrspace(1)
↪  %output) {
  %local_ptr = alloca i32, addrspace(3)  ; Allocate local memory
  ↪  (address space 3)
  store i32 0, ptr addrspace(3) %local_ptr  ; Store a value in local
  ↪  memory
  %value = load i32, ptr addrspace(1) %input  ; Load a value from
  ↪  global memory
  store i32 %value, ptr addrspace(1) %output  ; Store a value in
  ↪  global memory
  ret void
}
```

2. **Specialized Hardware**

   For specialized hardware, such as FPGAs or DSPs, address spaces can represent distinct memory banks or regions with unique access properties.

   Example:

```
@mem_bank_a = addrspace(4) global i32 42  ; Memory bank A (address
↪  space 4)
@mem_bank_b = addrspace(5) global i32 0   ; Memory bank B (address
↪  space 5)

define void @hardware_function() {
  %value_a = load i32, ptr addrspace(4) @mem_bank_a  ; Load from
  ↪  memory bank A
  store i32 %value_a, ptr addrspace(5) @mem_bank_b   ; Store in memory
  ↪  bank B
  ret void
}
```

5.4.8 Key Points

- Address spaces represent distinct regions of memory with unique properties.
- Pointers and global variables can be associated with specific address spaces using the `addrspace(<num>)` keyword.
- The `addrspacecast` instruction is used to convert pointers between different address spaces.
- Address spaces are particularly useful for GPU programming and specialized hardware.

5.4.9 Conclusion

Address spaces and pointers are powerful features of LLVM IR that enable fine-grained control over memory access and optimization. By understanding how to use address spaces, beginners can write LLVM IR code that is tailored to specific hardware architectures and memory hierarchies.

Chapter 6

Control Flow & Branching

6.1 Unconditional Branch (`br label %target`)

6.1.1 Introduction

Control flow is a fundamental concept in programming that determines the order in which instructions are executed. In LLVM IR, control flow is managed using **branch instructions**, which direct the execution of the program to different parts of the code. The simplest form of branch instruction is the **unconditional branch**, which transfers control to a specified label without any condition. In this section, we will explore the syntax, usage, and key characteristics of the unconditional branch instruction (`br label %target`). Understanding this instruction is essential for writing and analyzing LLVM IR code that involves jumps and loops.

6.1.2 What is an Unconditional Branch?

An unconditional branch is a control flow instruction that transfers execution to a specified label without evaluating any condition. It is represented by the `br` instruction followed by the `label` keyword and the target label.

1. **Syntax**

   The syntax for an unconditional branch is:

   ```
   br label %<target>
   ```

   where:

   - `br` is the branch instruction.
   - `label` indicates that the branch is unconditional.
   - `%<target>` is the label to which control is transferred.

2. **Example Usage**

   Here is an example of an unconditional branch:

   ```
   define void @example() {
     br label %next_block  ; Unconditionally branch to %next_block
   next_block:
     ret void
   }
   ```

   In this example:

   - The `br label %next_block` instruction unconditionally transfers control to the `%next_block` label.
   - The `ret void` instruction is executed after the branch.

6.1.3 Key Characteristics of Unconditional Branches

1. **No Condition**: Unconditional branches do not evaluate any condition. They always transfer control to the specified label.

2. **Target Label**: The target of an unconditional branch must be a valid label within the same function.

3. **Terminator Instruction**: Unconditional branches are **terminator instructions**, meaning they must appear at the end of a basic block.

6.1.4 Using Unconditional Branches

Unconditional branches are commonly used in the following scenarios:

1. **Jumping to a Label**

   The most straightforward use of an unconditional branch is to jump to a specific label within the same function.

   Example:

```
define void @jump_example() {
  br label %target_block  ; Jump to %target_block
target_block:
  ret void
}
```

2. **Creating Loops**

   Unconditional branches are often used to create loops by jumping back to the beginning of a loop body.

Example:

```
define void @loop_example() {
entry:
  br label %loop_body  ; Jump to the start of the loop
loop_body:
  ; Loop body instructions go here
  br label %loop_body  ; Jump back to the start of the loop
}
```

In this example:

- The `br label %loop_body` instruction creates an infinite loop by repeatedly jumping back to the `%loop_body` label.

3. **Combining with Conditional Branches**

   Unconditional branches are often used in conjunction with conditional branches to implement more complex control flow, such as `if-else` statements.

   Example:

```
define void @if_else_example(i32 %x) {
  %cmp = icmp eq i32 %x, 0  ; Compare %x with 0
  br i1 %cmp, label %if_true, label %if_false  ; Conditional branch
if_true:
  ; Instructions for the true case
  br label %merge  ; Jump to the merge block
if_false:
  ; Instructions for the false case
  br label %merge  ; Jump to the merge block
merge:
```

```
  ret void
}
```

In this example:

- The `br i1 %cmp, label %if_true, label %if_false` instruction is a conditional branch that chooses between two labels.
- The unconditional branches (`br label %merge`) ensure that control flows to the `%merge` block after executing the `if_true` or `if_false` blocks.

6.1.5 Example: Unconditional Branch in a Loop

Here is a complete example that demonstrates the use of an unconditional branch to create a loop:

```
define void @count_to_10() {
entry:
  %i = alloca i32, align 4  ; Allocate memory for the loop counter
  store i32 0, ptr %i        ; Initialize the loop counter to 0
  br label %loop_condition   ; Jump to the loop condition
loop_condition:
  %current_i = load i32, ptr %i  ; Load the current value of the loop
  ↪  counter
  %cmp = icmp slt i32 %current_i, 10  ; Compare the loop counter with 10
  br i1 %cmp, label %loop_body, label %loop_exit  ; Conditional branch
loop_body:
  ; Print the current value of the loop counter (assuming a print function
  ↪  exists)
  call void @print_i32(i32 %current_i)
  %next_i = add i32 %current_i, 1  ; Increment the loop counter
```

```
  store i32 %next_i, ptr %i         ; Store the updated loop counter
  br label %loop_condition          ; Jump back to the loop condition
loop_exit:
  ret void
}

declare void @print_i32(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `br label %loop_condition` instruction creates a loop by repeatedly jumping back to the `%loop_condition` label.
- The loop continues until the loop counter (`%current_i`) reaches 10.

6.1.6 Key Points

- Unconditional branches transfer control to a specified label without evaluating any condition.
- They are represented by the `br label %<target>` instruction.
- Unconditional branches are terminator instructions and must appear at the end of a basic block.
- They are commonly used to create loops, implement jumps, and combine with conditional branches for complex control flow.

6.1.7 Conclusion

The unconditional branch instruction (`br label %target`) is a fundamental tool for managing control flow in LLVM IR. By understanding how to use unconditional branches,

beginners can write LLVM IR code that implements jumps, loops, and other control flow constructs.

6.2 Conditional Branch (`br i1 %cond, label %if, label %else`)

6.2.1 Introduction

Conditional branches are a cornerstone of control flow in LLVM IR, enabling programs to make decisions based on runtime conditions. Unlike unconditional branches, which always transfer control to a specified label, conditional branches evaluate a boolean condition and choose between two possible paths. In this section, we will explore the syntax, usage, and key characteristics of the conditional branch instruction (`br i1 %cond, label %if, label %else`). Understanding conditional branches is essential for implementing `if-else` statements, loops, and other dynamic control flow constructs in LLVM IR.

6.2.2 What is a Conditional Branch?

A conditional branch is a control flow instruction that evaluates a boolean condition (`i1` type) and transfers execution to one of two specified labels based on the result of the condition. It is represented by the `br` instruction followed by a condition and two target labels.

1. **Syntax**

   The syntax for a conditional branch is:

   ```
   br i1 <condition>, label <if_true>, label <if_false>
   ```

   where:

   - `br` is the branch instruction.
   - `i1` is the type of the condition (a 1-bit integer representing `true` or `false`).

- `<condition>` is the boolean condition to evaluate.
- `<if_true>` is the label to which control is transferred if the condition is `true`.
- `<if_false>` is the label to which control is transferred if the condition is `false`.

2. **Example Usage**

   Here is an example of a conditional branch:

```
define void @example(i32 %x) {
  %cmp = icmp eq i32 %x, 0  ; Compare %x with 0
  br i1 %cmp, label %if_true, label %if_false  ; Conditional branch
if_true:
  ; Instructions for the true case
  ret void
if_false:
  ; Instructions for the false case
  ret void
}
```

   In this example:

   - The `icmp eq i32 %x, 0` instruction compares `%x` with `0` and produces a boolean result (`true` or `false`).
   - The `br i1 %cmp, label %if_true, label %if_false` instruction evaluates the condition and transfers control to either `%if_true` or `%if_false`.

6.2.3 Key Characteristics of Conditional Branches

1. **Condition Evaluation**: Conditional branches evaluate a boolean condition (`i1` type) to determine the control flow path.

2. **Two Targets**: Conditional branches have two target labels: one for the `true` case and one for the `false` case.

3. **Terminator Instruction**: Like unconditional branches, conditional branches are **terminator instructions** and must appear at the end of a basic block.

6.2.4 Using Conditional Branches

Conditional branches are commonly used in the following scenarios:

1. **Implementing `if-else` Statements**

   Conditional branches are the primary mechanism for implementing `if-else` statements in LLVM IR.

   Example:

```
define void @if_else_example(i32 %x) {
  %cmp = icmp sgt i32 %x, 0  ; Compare %x with 0
  br i1 %cmp, label %if_true, label %if_false  ; Conditional branch
if_true:
  ; Instructions for the true case (x > 0)
  call void @print(i32 1)  ; Print 1 (assuming a print function
  ↪  exists)
  br label %merge  ; Jump to the merge block
if_false:
  ; Instructions for the false case (x <= 0)
  call void @print(i32 0)  ; Print 0
  br label %merge  ; Jump to the merge block
merge:
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `icmp sgt i32 %x, 0` instruction checks if `%x` is greater than `0`.
- The `br i1 %cmp, label %if_true, label %if_false` instruction implements the `if-else` logic.
- The unconditional branches (`br label %merge`) ensure that control flows to the `%merge` block after executing the `if_true` or `if_false` blocks.

2. **Creating Loops with Conditions**

   Conditional branches are often used to create loops by evaluating a condition at the beginning or end of the loop.

   Example:

```
define void @loop_example(i32 %n) {
entry:
  %i = alloca i32, align 4  ; Allocate memory for the loop counter
  store i32 0, ptr %i       ; Initialize the loop counter to 0
  br label %loop_condition  ; Jump to the loop condition
loop_condition:
  %current_i = load i32, ptr %i  ; Load the current value of the loop
  ↪ counter
  %cmp = icmp slt i32 %current_i, %n  ; Compare the loop counter with
  ↪ %n
  br i1 %cmp, label %loop_body, label %loop_exit  ; Conditional
  ↪ branch
loop_body:
  ; Loop body instructions go here
  call void @print_i32(i32 %current_i)  ; Print the current loop
  ↪ counter
```

```
  %next_i = add i32 %current_i, 1        ; Increment the loop counter
  store i32 %next_i, ptr %i              ; Store the updated loop
   ↪  counter
  br label %loop_condition               ; Jump back to the loop
   ↪  condition
loop_exit:
  ret void
}

declare void @print_i32(i32)  ; Declare a function to print an i32
 ↪  value
```

In this example:

- The `icmp slt i32 %current_i, %n` instruction checks if the loop counter (`%current_i`) is less than `%n`.
- The `br i1 %cmp, label %loop_body, label %loop_exit` instruction implements the loop condition.
- The loop continues until the loop counter reaches `%n`.

3. **Combining with Unconditional Branches**

   Conditional branches are often combined with unconditional branches to implement more complex control flow, such as nested `if-else` statements or multi-way branches.

   Example:

```
define void @nested_if_example(i32 %x, i32 %y) {
  %cmp1 = icmp sgt i32 %x, 0  ; Compare %x with 0
  br i1 %cmp1, label %if_x_positive, label %if_x_non_positive  ;
   ↪  First conditional branch
```

```
if_x_positive:
  %cmp2 = icmp sgt i32 %y, 0  ; Compare %y with 0
  br i1 %cmp2, label %if_both_positive, label
  ↪ %if_x_positive_y_non_positive  ; Second conditional branch
if_both_positive:
  call void @print(i32 1)  ; Print 1 (both x and y are positive)
  br label %merge  ; Jump to the merge block
if_x_positive_y_non_positive:
  call void @print(i32 2)  ; Print 2 (x is positive, y is
  ↪ non-positive)
  br label %merge  ; Jump to the merge block
if_x_non_positive:
  call void @print(i32 3)  ; Print 3 (x is non-positive)
  br label %merge  ; Jump to the merge block
merge:
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The first conditional branch (`br i1 %cmp1, label %if_x_positive, label %if_x_non_positive`) checks if `%x` is positive.
- The second conditional branch (`br i1 %cmp2, label %if_both_positive, label %if_x_positive_y_non_positive`) checks if `%y` is positive.
- Unconditional branches (`br label %merge`) ensure that control flows to the `%merge` block after executing the appropriate case.

6.2.5 Example: Conditional Branch in a Loop

Here is a complete example that demonstrates the use of a conditional branch to implement a loop:

```
define void @count_to_n(i32 %n) {
entry:
  %i = alloca i32, align 4  ; Allocate memory for the loop counter
  store i32 0, ptr %i        ; Initialize the loop counter to 0
  br label %loop_condition   ; Jump to the loop condition
loop_condition:
  %current_i = load i32, ptr %i  ; Load the current value of the loop
  ↪  counter
  %cmp = icmp slt i32 %current_i, %n  ; Compare the loop counter with %n
  br i1 %cmp, label %loop_body, label %loop_exit  ; Conditional branch
loop_body:
  ; Print the current value of the loop counter (assuming a print function
  ↪  exists)
  call void @print_i32(i32 %current_i)
  %next_i = add i32 %current_i, 1  ; Increment the loop counter
  store i32 %next_i, ptr %i        ; Store the updated loop counter
  br label %loop_condition         ; Jump back to the loop condition
loop_exit:
  ret void
}

declare void @print_i32(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `icmp slt i32 %current_i, %n` instruction checks if the loop counter (`%current_i`) is less than `%n`.

- The `br i1 %cmp, label %loop_body, label %loop_exit` instruction implements the loop condition.
- The loop continues until the loop counter reaches `%n`.

6.2.6 Key Points

- Conditional branches evaluate a boolean condition (`i1` type) and transfer control to one of two specified labels.
- They are represented by the `br i1 <condition>, label <if_true>, label <if_false>` instruction.
- Conditional branches are terminator instructions and must appear at the end of a basic block.
- They are commonly used to implement `if-else` statements, loops, and other dynamic control flow constructs.

6.2.7 Conclusion

The conditional branch instruction (`br i1 %cond, label %if, label %else`) is a powerful tool for managing dynamic control flow in LLVM IR. By understanding how to use conditional branches, beginners can write LLVM IR code that implements decision-making logic, loops, and other complex control flow constructs.

6.3 Switch Statements (`switch i32 %val, label %default`)

6.3.1 Introduction

Switch statements are a powerful control flow construct that allow multi-way branching based on the value of a variable. In LLVM IR, switch statements are implemented using the `switch` instruction, which compares a value against a set of constants and transfers control to the corresponding label. This section will provide a detailed explanation of the syntax, usage, and key characteristics of switch statements in LLVM IR. Understanding switch statements is essential for implementing complex control flow logic, such as state machines or multi-way decision-making.

6.3.2 What is a Switch Statement?

A switch statement is a control flow instruction that evaluates a value and transfers control to one of several possible labels based on the value. It is similar to a series of `if-else` statements but is more efficient for handling multiple cases. In LLVM IR, the `switch` instruction is used to implement switch statements.

1. **Syntax**

   The syntax for a switch statement is:

   ```
   switch <type> <value>, label <default> [ <case1>, label <target1>,
   ↪  <case2>, label <target2>, ... ]
   ```

   where:

   - `<type>` is the type of the value being compared (e.g., `i32`).

- `<value>` is the value being compared.
- `<default>` is the label to which control is transferred if no cases match.
- `<case1>`, `<case2>`, etc., are the constant values to compare against.
- `<target1>`, `<target2>`, etc., are the labels to which control is transferred if the corresponding case matches.

2. **Example Usage**

   Here is an example of a switch statement:

```
define void @switch_example(i32 %x) {
  switch i32 %x, label %default [
    i32 1, label %case1
    i32 2, label %case2
    i32 3, label %case3
  ]
case1:
  ; Instructions for case 1
  ret void
case2:
  ; Instructions for case 2
  ret void
case3:
  ; Instructions for case 3
  ret void
default:
  ; Instructions for the default case
  ret void
}
```

   In this example:

- The `switch i32 %x, label %default [ ... ]` instruction compares `%x` against the values `1`, `2`, and `3`.
- If `%x` matches one of the cases, control is transferred to the corresponding label (`%case1`, `%case2`, or `%case3`).
- If `%x` does not match any of the cases, control is transferred to the `%default` label.

6.3.3 Key Characteristics of Switch Statements

1. **Multi-Way Branching**: Switch statements allow for multi-way branching based on the value of a variable.
2. **Default Case**: Switch statements must include a default case, which is executed if no other cases match.
3. **Terminator Instruction**: Like other branch instructions, switch statements are **terminator instructions** and must appear at the end of a basic block.
4. **Constant Cases**: The cases in a switch statement must be constant values.

6.3.4 Using Switch Statements

Switch statements are commonly used in the following scenarios:

1. **Implementing Multi-Way Decision-Making**

   Switch statements are ideal for implementing multi-way decision-making logic, such as handling different states in a state machine.

   Example:

```
define void @state_machine(i32 %state) {
  switch i32 %state, label %default [
    i32 0, label %state0
    i32 1, label %state1
    i32 2, label %state2
  ]
state0:
  ; Instructions for state 0
  call void @print(i32 0)  ; Print 0 (assuming a print function
  ↪  exists)
  ret void
state1:
  ; Instructions for state 1
  call void @print(i32 1)  ; Print 1
  ret void
state2:
  ; Instructions for state 2
  call void @print(i32 2)  ; Print 2
  ret void
default:
  ; Instructions for the default state
  call void @print(i32 -1)  ; Print -1
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `switch i32 %state, label %default [ ... ]` instruction implements a state machine with three states (`0`, `1`, and `2`).
- The `default` case handles any unexpected state values.

2. **Handling Enumerations**

   Switch statements are often used to handle enumerations, where each case corresponds to a specific enumerated value.

   Example:

```
define void @handle_enum(i32 %enum_value) {
  switch i32 %enum_value, label %default [
    i32 0, label %case_red
    i32 1, label %case_green
    i32 2, label %case_blue
  ]
case_red:
  ; Instructions for the red case
  call void @print(i32 0)  ; Print 0 (red)
  ret void
case_green:
  ; Instructions for the green case
  call void @print(i32 1)  ; Print 1 (green)
  ret void
case_blue:
  ; Instructions for the blue case
  call void @print(i32 2)  ; Print 2 (blue)
  ret void
default:
  ; Instructions for the default case
  call void @print(i32 -1)  ; Print -1 (invalid)
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `switch i32 %enum_value, label %default [ ... ]` instruction handles an enumeration with three values (`0`, `1`, and `2`).
- The `default` case handles any invalid enumeration values.

3. **Combining with Other Control Flow Constructs**

   Switch statements can be combined with other control flow constructs, such as loops and conditional branches, to implement more complex logic.

   Example:

```
define void @complex_logic(i32 %x) {
entry:
  %cmp = icmp sgt i32 %x, 0  ; Compare %x with 0
  br i1 %cmp, label %positive, label %non_positive  ; Conditional
   ↪  branch
positive:
  switch i32 %x, label %default [
    i32 1, label %case1
    i32 2, label %case2
    i32 3, label %case3
  ]
case1:
  ; Instructions for case 1
  call void @print(i32 1)  ; Print 1
  br label %merge  ; Jump to the merge block
case2:
  ; Instructions for case 2
  call void @print(i32 2)  ; Print 2
  br label %merge  ; Jump to the merge block
case3:
```

```
  ; Instructions for case 3
  call void @print(i32 3)  ; Print 3
  br label %merge  ; Jump to the merge block
default:
  ; Instructions for the default case
  call void @print(i32 -1)  ; Print -1
  br label %merge  ; Jump to the merge block
non_positive:
  ; Instructions for the non-positive case
  call void @print(i32 0)  ; Print 0
  br label %merge  ; Jump to the merge block
merge:
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `icmp sgt i32 %x, 0` instruction checks if `%x` is positive.
- The `br i1 %cmp, label %positive, label %non_positive` instruction implements the conditional branch.
- The `switch i32 %x, label %default [ ... ]` instruction handles the positive cases.
- Unconditional branches (`br label %merge`) ensure that control flows to the `%merge` block after executing the appropriate case.

6.3.5 Example: Switch Statement in a Loop

Here is a complete example that demonstrates the use of a switch statement within a loop:

```
define void @loop_with_switch(i32 %n) {
entry:
  %i = alloca i32, align 4  ; Allocate memory for the loop counter
  store i32 0, ptr %i        ; Initialize the loop counter to 0
  br label %loop_condition   ; Jump to the loop condition
loop_condition:
  %current_i = load i32, ptr %i  ; Load the current value of the loop
  ↪  counter
  %cmp = icmp slt i32 %current_i, %n  ; Compare the loop counter with %n
  br i1 %cmp, label %loop_body, label %loop_exit  ; Conditional branch
loop_body:
  switch i32 %current_i, label %default [
    i32 0, label %case0
    i32 1, label %case1
    i32 2, label %case2
  ]
case0:
  ; Instructions for case 0
  call void @print(i32 0)  ; Print 0
  br label %loop_increment  ; Jump to the loop increment
case1:
  ; Instructions for case 1
  call void @print(i32 1)  ; Print 1
  br label %loop_increment  ; Jump to the loop increment
case2:
  ; Instructions for case 2
  call void @print(i32 2)  ; Print 2
  br label %loop_increment  ; Jump to the loop increment
default:
  ; Instructions for the default case
  call void @print(i32 -1)  ; Print -1
  br label %loop_increment  ; Jump to the loop increment
```

```
loop_increment:
  %next_i = add i32 %current_i, 1  ; Increment the loop counter
  store i32 %next_i, ptr %i        ; Store the updated loop counter
  br label %loop_condition         ; Jump back to the loop condition
loop_exit:
  ret void
}

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `switch i32 %current_i, label %default [ ... ]` instruction handles different cases based on the loop counter (`%current_i`).
- The loop continues until the loop counter reaches `%n`.

6.3.6 Key Points

- Switch statements allow for multi-way branching based on the value of a variable.
- They are represented by the `switch <type> <value>, label <default> [ ... ]` instruction.
- Switch statements must include a default case and are terminator instructions.
- They are commonly used for implementing state machines, handling enumerations, and combining with other control flow constructs.

6.3.7 Conclusion

The switch statement (`switch i32 %val, label %default`) is a powerful tool for managing multi-way branching in LLVM IR. By understanding how to use switch statements,

beginners can write LLVM IR code that implements complex control flow logic, such as state machines and multi-way decision-making.

6.4 Indirect Branching (`indirectbr i8* %addr, [label %1, label %2]`)

6.4.1 Introduction

Indirect branching is a powerful and advanced control flow mechanism in LLVM IR that allows for dynamic and runtime-determined jumps. Unlike direct branches (e.g., `br` or `switch`), which transfer control to a fixed set of labels, indirect branches use a pointer to determine the target label at runtime. This makes indirect branching particularly useful for implementing features like jump tables, virtual function dispatch, and other dynamic control flow patterns. In this section, we will explore the syntax, usage, and key characteristics of indirect branching in LLVM IR. Understanding indirect branching is essential for advanced LLVM IR programming, especially for performance-critical or low-level applications.

6.4.2 What is Indirect Branching?

Indirect branching is a control flow instruction that transfers execution to a label determined by a pointer value. The target label is not known at compile time but is computed at runtime. This is achieved using the `indirectbr` instruction, which takes a pointer to a block address and a list of possible target labels.

1. **Syntax**

   The syntax for an indirect branch is:

   ```
   indirectbr ptr <address>, [label <target1>, label <target2>, ...]
   ```

   where:

   - `indirectbr` is the indirect branch instruction.

- `ptr <address>` is a pointer to a block address (typically of type `i8*`).
- `[label <target1>, label <target2>, ...]` is a list of possible target labels.

2. **Example Usage**

   Here is an example of an indirect branch:

```
define void @indirect_branch_example(ptr %addr) {
  indirectbr ptr %addr, [label %label1, label %label2]  ; Indirect
  ↪ branch
label1:
  ; Instructions for label1
  ret void
label2:
  ; Instructions for label2
  ret void
}
```

   In this example:

   - The `indirectbr ptr %addr, [label %label1, label %label2]` instruction transfers control to either `%label1` or `%label2`, depending on the value of `%addr`.
   - The value of `%addr` must be a valid block address corresponding to one of the labels in the list.

6.4.3 Key Characteristics of Indirect Branching

1. **Dynamic Target**: The target of an indirect branch is determined at runtime, making it highly flexible but also harder to analyze and optimize.

2. **Block Address**: The pointer used in an indirect branch must be a valid block address, typically obtained using the `blockaddress` function.

3. **Terminator Instruction**: Like other branch instructions, indirect branches are **terminator instructions** and must appear at the end of a basic block.

4. **Target List**: The list of possible target labels must include all valid targets for the indirect branch.

6.4.4 Using Indirect Branching

Indirect branching is commonly used in the following scenarios:

1. **Implementing Jump Tables**

   Jump tables are a common use case for indirect branching. They allow for efficient multi-way branching based on an index or key.

   Example:

```
define void @jump_table_example(i32 %index) {
  %addr = getelementptr inbounds [3 x ptr], ptr @jump_table, i32 0,
   ↪  i32 %index  ; Compute the address in the jump table
  %target = load ptr, ptr %addr  ; Load the target address from the
   ↪  jump table
  indirectbr ptr %target, [label %case0, label %case1, label %case2]
   ↪  ; Indirect branch
case0:
  ; Instructions for case 0
  ret void
case1:
  ; Instructions for case 1
  ret void
```

```
case2:
  ; Instructions for case 2
  ret void
}

@jump_table = global [3 x ptr] [ptr blockaddress(@jump_table_example,
↪  %case0),
                                ptr blockaddress(@jump_table_example,
                                ↪  %case1),
                                ptr blockaddress(@jump_table_example,
                                ↪  %case2)]
```

In this example:

- The `@jump_table` global variable contains the addresses of the target labels (`%case0`, `%case1`, and `%case2`).
- The `getelementptr` instruction computes the address of the target label based on the `%index` value.
- The `indirectbr` instruction transfers control to the target label specified by the jump table.

2. **Virtual Function Dispatch**

   Indirect branching can be used to implement virtual function dispatch, where the target function is determined at runtime based on the object's type.

   Example:

```
define void @virtual_dispatch(ptr %obj) {
  %vtable = load ptr, ptr %obj  ; Load the virtual table pointer from
  ↪  the object
```

```
  %func_ptr = load ptr, ptr %vtable  ; Load the function pointer from
  ↪  the virtual table
  indirectbr ptr %func_ptr, [label %func1, label %func2]  ; Indirect
  ↪  branch
func1:
  ; Instructions for func1
  ret void
func2:
  ; Instructions for func2
  ret void
}
```

In this example:

- The `%obj` pointer points to an object with a virtual table.
- The `%vtable` pointer is loaded from the object, and the `%func_ptr` pointer is loaded from the virtual table.
- The `indirectbr` instruction transfers control to the function specified by `%func_ptr`.

3. **State Machines**

   Indirect branching can be used to implement state machines, where the next state is determined dynamically at runtime.

   Example:

```
define void @state_machine(ptr %state) {
  %next_state = load ptr, ptr %state  ; Load the next state address
  indirectbr ptr %next_state, [label %state0, label %state1, label
  ↪  %state2]  ; Indirect branch
```

```
state0:
  ; Instructions for state0
  ret void
state1:
  ; Instructions for state1
  ret void
state2:
  ; Instructions for state2
  ret void
}
```

In this example:

- The `%state` pointer points to the current state of the state machine.
- The `%next_state` pointer is loaded from the state, and the `indirectbr` instruction transfers control to the next state.

6.4.5 Example: Indirect Branching in a Loop

Here is a complete example that demonstrates the use of indirect branching within a loop:

```
define void @loop_with_indirect_branch(ptr %initial_state) {
entry:
  %state = alloca ptr, align 8  ; Allocate memory for the state pointer
  store ptr %initial_state, ptr %state  ; Initialize the state pointer
  br label %loop  ; Jump to the loop
loop:
  %current_state = load ptr, ptr %state  ; Load the current state address
  indirectbr ptr %current_state, [label %state0, label %state1, label
  ↪  %state2]  ; Indirect branch
state0:
```

```
  ; Instructions for state0
  call void @print(i32 0)  ; Print 0 (assuming a print function exists)
  %next_state0 = getelementptr inbounds [3 x ptr], ptr @state_table, i32 0,
  ↪  i32 1  ; Compute the next state
  store ptr %next_state0, ptr %state  ; Update the state pointer
  br label %loop  ; Jump back to the loop
state1:
  ; Instructions for state1
  call void @print(i32 1)  ; Print 1
  %next_state1 = getelementptr inbounds [3 x ptr], ptr @state_table, i32 0,
  ↪  i32 2  ; Compute the next state
  store ptr %next_state1, ptr %state  ; Update the state pointer
  br label %loop  ; Jump back to the loop
state2:
  ; Instructions for state2
  call void @print(i32 2)  ; Print 2
  ret void  ; Exit the loop
}

@state_table = global [3 x ptr] [ptr
↪  blockaddress(@loop_with_indirect_branch, %state0),
                                 ptr
                                  ↪  blockaddress(@loop_with_indirect_branch,
                                  ↪  %state1),
                                 ptr
                                  ↪  blockaddress(@loop_with_indirect_branch,
                                  ↪  %state2)]

declare void @print(i32)  ; Declare a function to print an i32 value
```

In this example:

- The `@state_table` global variable contains the addresses of the state labels

(`%state0`, `%state1`, and `%state2`).

- The `indirectbr` instruction transfers control to the current state.
- The loop continues until the `%state2` state is reached, at which point the function returns.

6.4.6 Points

- Indirect branching allows for dynamic and runtime-determined jumps using a pointer to a block address.
- It is represented by the `indirectbr ptr <address>, [label <target1>, label <target2>, ...]` instruction.
- Indirect branching is commonly used for jump tables, virtual function dispatch, and state machines.
- It is a terminator instruction and must appear at the end of a basic block.

6.4.7 Conclusion

Indirect branching (`indirectbr i8* %addr, [label %1, label %2]`) is a powerful and flexible control flow mechanism in LLVM IR. By understanding how to use indirect branching, beginners can implement advanced control flow patterns, such as jump tables and state machines, in their LLVM IR code.

Part II

Intermediate Users: Focus to optimize LLVM IR usage.

Chapter 7

Arithmetic & Logical Operations

7.1 Integer Arithmetic (`add`, `sub`, `mul`, `sdiv`, `udiv`)

7.1.1 Introduction

Integer arithmetic is a fundamental aspect of LLVM IR, enabling the manipulation of integer values through operations such as addition, subtraction, multiplication, and division. These operations are essential for implementing mathematical computations, control flow logic, and data manipulation in LLVM IR. In this section, we will explore the syntax, usage, and key characteristics of integer arithmetic instructions in LLVM IR, including `add`, `sub`, `mul`, `sdiv`, and `udiv`. Understanding these instructions is crucial for intermediate users who want to write efficient and correct LLVM IR code.

7.1.2 Integer Arithmetic Instructions

LLVM IR provides a set of integer arithmetic instructions that operate on integer types (`iN`, where `N` is the number of bits). These instructions include:

1. **Addition (`add`)**
2. **Subtraction (`sub`)**
3. **Multiplication (`mul`)**
4. **Signed Division (`sdiv`)**
5. **Unsigned Division (`udiv`)**

Each of these instructions has specific behavior and constraints, which we will explore in detail.

7.1.3 Addition (`add`)

The `add` instruction performs integer addition on two operands of the same integer type.

1. **Syntax**

   The syntax for the `add` instruction is:

   ```
   <result> = add <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the addition.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands to be added.

2. **Example Usage**

   Here is an example of the `add` instruction:

```
%sum = add i32 5, 10  ; %sum = 5 + 10 = 15
```

3. **Key Points**

   - The `add` instruction performs integer addition.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Overflow behavior is undefined for signed integers.

7.1.4 Subtraction (`sub`)

The `sub` instruction performs integer subtraction on two operands of the same integer type.

1. **Syntax**

   The syntax for the `sub` instruction is:

   ```
   <result> = sub <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the subtraction.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` is the minuend (the value from which `<op2>` is subtracted).
   - `<op2>` is the subtrahend (the value to subtract).

2. **Example Usage**

   Here is an example of the `sub` instruction:

```
%difference = sub i32 20, 7  ; %difference = 20 - 7 = 13
```

3. **Key Points**

   - The `sub` instruction performs integer subtraction.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Overflow behavior is undefined for signed integers.

7.1.5 Multiplication (`mul`)

The `mul` instruction performs integer multiplication on two operands of the same integer type.

1. **Syntax**

   The syntax for the `mul` instruction is:

   ```
   <result> = mul <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the multiplication.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands to be multiplied.

2. **Example Usage**

   Here is an example of the `mul` instruction:

```
%product = mul i32 6, 7  ; %product = 6 * 7 = 42
```

3. **Key Points**

   - The `mul` instruction performs integer multiplication.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Overflow behavior is undefined for signed integers.

7.1.6 Signed Division (`sdiv`)

The `sdiv` instruction performs signed integer division on two operands of the same integer type.

1. **Syntax**

   The syntax for the `sdiv` instruction is:

   ```
   <result> = sdiv <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the division.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` is the dividend (the value to be divided).
   - `<op2>` is the divisor (the value by which `<op1>` is divided).

2. **Example Usage**

   Here is an example of the `sdiv` instruction:

```
%quotient = sdiv i32 -20, 4  ; %quotient = -20 / 4 = -5
```

3. **Key Points**

   - The `sdiv` instruction performs signed integer division.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Division by zero is undefined behavior.
   - The result is rounded towards zero.

7.1.7 Unsigned Division (`udiv`)

The `udiv` instruction performs unsigned integer division on two operands of the same integer type.

1. **Syntax**

   The syntax for the `udiv` instruction is:

```
<result> = udiv <type> <op1>, <op2>
```

   where:

   - `<result>` is the variable that stores the result of the division.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).

- `<op1>` is the dividend (the value to be divided).
- `<op2>` is the divisor (the value by which `<op1>` is divided).

2. **Example Usage**

   Here is an example of the `udiv` instruction:

```
%quotient = udiv i32 20, 4  ; %quotient = 20 / 4 = 5
```

3. **Key Points**

   - The `udiv` instruction performs unsigned integer division.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Division by zero is undefined behavior.
   - The result is rounded towards zero.

7.1.8 Example: Combining Integer Arithmetic Instructions

Here is a complete example that demonstrates the use of integer arithmetic instructions in LLVM IR:

```
define i32 @arithmetic_example(i32 %a, i32 %b) {
  %sum = add i32 %a, %b           ; %sum = %a + %b
  %difference = sub i32 %a, %b    ; %difference = %a - %b
  %product = mul i32 %a, %b       ; %product = %a * %b
  %quotient_signed = sdiv i32 %a, %b  ; %quotient_signed = %a / %b
  ↪  (signed)
  %quotient_unsigned = udiv i32 %a, %b  ; %quotient_unsigned = %a / %b
  ↪  (unsigned)
```

```
  %result = add i32 %sum, %difference  ; %result = %sum + %difference
  %final_result = mul i32 %result, %product  ; %final_result = %result *
  ↪  %product
  ret i32 %final_result
}
```

In this example:

- The function `@arithmetic_example` takes two integer arguments (`%a` and `%b`).
- It performs addition, subtraction, multiplication, signed division, and unsigned division on the arguments.
- The results are combined to compute a final result, which is returned.

7.1.9 Key Points

- Integer arithmetic instructions (`add`, `sub`, `mul`, `sdiv`, `udiv`) operate on integer types (`iN`).
- Both operands must be of the same integer type.
- The result is of the same type as the operands.
- Division by zero is undefined behavior.
- Overflow behavior is undefined for signed integers.

7.1.10 Conclusion

Integer arithmetic instructions are essential for performing mathematical computations and data manipulation in LLVM IR. By understanding the syntax, usage, and behavior of these instructions, intermediate users can write efficient and correct LLVM IR code.

7.2 Floating-Point Arithmetic (`fadd`, `fsub`, `fmul`, `fdiv`)

7.2.1 Introduction

Floating-point arithmetic is a critical aspect of LLVM IR, enabling the manipulation of real numbers through operations such as addition, subtraction, multiplication, and division. These operations are essential for implementing scientific computations, graphics processing, and other applications that require precise handling of fractional values. In this section, we will explore the syntax, usage, and key characteristics of floating-point arithmetic instructions in LLVM IR, including `fadd`, `fsub`, `fmul`, and `fdiv`. Understanding these instructions is crucial for intermediate users who want to write efficient and accurate LLVM IR code for floating-point computations.

7.2.2 Floating-Point Arithmetic Instructions

LLVM IR provides a set of floating-point arithmetic instructions that operate on floating-point types (`float`, `double`, etc.). These instructions include:

1. **Floating-Point Addition (`fadd`)**
2. **Floating-Point Subtraction (`fsub`)**
3. **Floating-Point Multiplication (`fmul`)**
4. **Floating-Point Division (`fdiv`)**

Each of these instructions has specific behavior and constraints, which we will explore in detail.

7.2.3 Floating-Point Addition (`fadd`)

The `fadd` instruction performs floating-point addition on two operands of the same floating-point type.

1. **Syntax**

   The syntax for the `fadd` instruction is:

   ```
   <result> = fadd <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the addition.
   - `<type>` is the floating-point type of the operands (e.g., `float`, `double`).
   - `<op1>` and `<op2>` are the operands to be added.

2. **Example Usage**

   Here is an example of the `fadd` instruction:

   ```
   %sum = fadd float 3.14, 2.71  ; %sum = 3.14 + 2.71 = 5.85
   ```

3. **Key Points**

   - The `fadd` instruction performs floating-point addition.
   - Both operands must be of the same floating-point type.
   - The result is of the same type as the operands.
   - The operation follows IEEE 754 floating-point arithmetic rules.

7.2.4 Floating-Point Subtraction (`fsub`)

The `fsub` instruction performs floating-point subtraction on two operands of the same floating-point type.

1. **Syntax**

   The syntax for the `fsub` instruction is:

   ```
   <result> = fsub <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the subtraction.
   - `<type>` is the floating-point type of the operands (e.g., `float`, `double`).
   - `<op1>` is the minuend (the value from which `<op2>` is subtracted).
   - `<op2>` is the subtrahend (the value to subtract).

2. **Example Usage**

   Here is an example of the `fsub` instruction:

   ```
   %difference = fsub float 10.5, 3.2  ; %difference = 10.5 - 3.2 = 7.3
   ```

3. **Key Points**

   - The `fsub` instruction performs floating-point subtraction.
   - Both operands must be of the same floating-point type.
   - The result is of the same type as the operands.
   - The operation follows IEEE 754 floating-point arithmetic rules.

7.2.5 Floating-Point Multiplication (`fmul`)

The `fmul` instruction performs floating-point multiplication on two operands of the same floating-point type.

1. **Syntax**

   The syntax for the `fmul` instruction is:

   ```
   <result> = fmul <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the multiplication.
   - `<type>` is the floating-point type of the operands (e.g., `float`, `double`).
   - `<op1>` and `<op2>` are the operands to be multiplied.

2. **Example Usage**

   Here is an example of the `fmul` instruction:

   ```
   %product = fmul float 2.5, 4.0  ; %product = 2.5 * 4.0 = 10.0
   ```

3. **Key Points**

   - The `fmul` instruction performs floating-point multiplication.
   - Both operands must be of the same floating-point type.
   - The result is of the same type as the operands.
   - The operation follows IEEE 754 floating-point arithmetic rules.

7.2.6 Floating-Point Division (`fdiv`)

The `fdiv` instruction performs floating-point division on two operands of the same floating-point type.

1. **Syntax**

   The syntax for the `fdiv` instruction is:

   ```
   <result> = fdiv <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the division.
   - `<type>` is the floating-point type of the operands (e.g., `float`, `double`).
   - `<op1>` is the dividend (the value to be divided).
   - `<op2>` is the divisor (the value by which `<op1>` is divided).

2. **Example Usage**

   Here is an example of the `fdiv` instruction:

   ```
   %quotient = fdiv float 10.0, 2.5  ; %quotient = 10.0 / 2.5 = 4.0
   ```

3. **Key Points**

   - The `fdiv` instruction performs floating-point division.
   - Both operands must be of the same floating-point type.
   - The result is of the same type as the operands.
   - The operation follows IEEE 754 floating-point arithmetic rules.
   - Division by zero produces a special floating-point value (e.g., `inf` or `NaN`).

7.2.7 Example: Combining Floating-Point Arithmetic Instructions

Here is a complete example that demonstrates the use of floating-point arithmetic instructions in LLVM IR:

```
define float @floating_point_example(float %a, float %b) {
  %sum = fadd float %a, %b          ; %sum = %a + %b
  %difference = fsub float %a, %b   ; %difference = %a - %b
  %product = fmul float %a, %b      ; %product = %a * %b
  %quotient = fdiv float %a, %b     ; %quotient = %a / %b
  %result = fadd float %sum, %difference  ; %result = %sum + %difference
  %final_result = fmul float %result, %product  ; %final_result = %result *
  ↪  %product
  ret float %final_result
}
```

In this example:

- The function `@floating_point_example` takes two floating-point arguments (`%a` and `%b`).
- It performs addition, subtraction, multiplication, and division on the arguments.
- The results are combined to compute a final result, which is returned.

7.2.8 Key Points

- Floating-point arithmetic instructions (`fadd`, `fsub`, `fmul`, `fdiv`) operate on floating-point types (`float`, `double`).
- Both operands must be of the same floating-point type.
- The result is of the same type as the operands.

- The operations follow IEEE 754 floating-point arithmetic rules.
- Division by zero produces special floating-point values (e.g., `inf` or `NaN`).

7.2.9 Advanced Topics

1. **Fast-Math Flags**

   LLVM IR supports **fast-math flags**, which allow for optimizations that may violate strict IEEE 754 semantics but can improve performance. These flags include:

   - `fast`: Enables all fast-math optimizations.
   - `nnan`: No NaNs (assumes no NaN values).
   - `ninf`: No infinities (assumes no infinite values).
   - `nsz`: No signed zeros (assumes no distinction between +0.0 and -0.0).
   - `arcp`: Allow reciprocal (replaces division with multiplication by reciprocal).

   Example:

   ```
   %sum = fadd fast float %a, %b  ; Perform fast-math addition
   ```

2. **Special Floating-Point Values**

   Floating-point arithmetic in LLVM IR can produce special values, such as:

   - **NaN (Not a Number)**: Represents an undefined or unrepresentable value.
   - **Infinity**: Represents an infinitely large value.
   - **Denormal**: Represents very small values close to zero.

   These values are handled according to IEEE 754 rules.

7.2.10 Conclusion

Floating-point arithmetic instructions are essential for performing precise computations involving real numbers in LLVM IR. By understanding the syntax, usage, and behavior of these instructions, intermediate users can write efficient and accurate LLVM IR code for floating-point computations.

7.3 Bitwise Operations (`and`, `or`, `xor`, `shl`, `lshr`, `ashr`)

7.3.1 Introduction

Bitwise operations are fundamental to low-level programming, enabling the manipulation of individual bits within integer values. These operations are essential for tasks such as data encoding, cryptography, hardware control, and performance optimizations. In this section, we will explore the syntax, usage, and key characteristics of bitwise operations in LLVM IR, including `and`, `or`, `xor`, `shl`, `lshr`, and `ashr`. Understanding these operations is crucial for intermediate users who want to write efficient and precise LLVM IR code for bit-level manipulations.

7.3.2 Bitwise Operations in LLVM IR

LLVM IR provides a set of bitwise operations that operate on integer types (`iN`, where `N` is the number of bits). These operations include:

1. **Bitwise AND (`and`)**
2. **Bitwise OR (`or`)**
3. **Bitwise XOR (`xor`)**
4. **Shift Left (`shl`)**
5. **Logical Shift Right (`lshr`)**
6. **Arithmetic Shift Right (`ashr`)**

Each of these operations has specific behavior and constraints, which we will explore in detail.

7.3.3 Bitwise AND (`and`)

The `and` instruction performs a bitwise AND operation on two integer operands.

1. **Syntax**

   The syntax for the `and` instruction is:

   ```
   <result> = and <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the AND operation.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands to be ANDed.

2. **Example Usage**

   Here is an example of the `and` instruction:

   ```
   %result = and i32 5, 3  ; %result = 5 & 3 = 1
   ```

   Explanation:

   - Binary representation of `5`: `0101`
   - Binary representation of `3`: `0011`
   - Bitwise AND result: `0001` (decimal `1`)

3. **Key Points**

   - The `and` instruction performs a bitwise AND operation.

- Both operands must be of the same integer type.
- The result is of the same type as the operands.
- Each bit in the result is `1` if both corresponding bits in the operands are `1`; otherwise, it is `0`.

7.3.4 Bitwise OR (`or`)

The `or` instruction performs a bitwise OR operation on two integer operands.

1. **Syntax**

   The syntax for the `or` instruction is:

   ```
   <result> = or <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the OR operation.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands to be ORed.

2. **Example Usage**

   Here is an example of the `or` instruction:

   ```
   %result = or i32 5, 3  ; %result = 5 | 3 = 7
   ```

   Explanation:

   - Binary representation of `5`: `0101`

- Binary representation of `3`: `0011`
- Bitwise OR result: `0111` (decimal `7`)

3. **Key Points**

   - The `or` instruction performs a bitwise OR operation.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Each bit in the result is `1` if at least one of the corresponding bits in the operands is `1`; otherwise, it is `0`.

7.3.5 Bitwise XOR (`xor`)

The `xor` instruction performs a bitwise XOR (exclusive OR) operation on two integer operands.

1. **Syntax**

   The syntax for the `xor` instruction is:

   ```
   <result> = xor <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the XOR operation.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands to be XORed.

2. **Example Usage**

   Here is an example of the `xor` instruction:

```
%result = xor i32 5, 3  ; %result = 5 ^ 3 = 6
```

   Explanation:

   - Binary representation of `5`: `0101`
   - Binary representation of `3`: `0011`
   - Bitwise XOR result: `0110` (decimal `6`)

3. **Key Points**

   - The `xor` instruction performs a bitwise XOR operation.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Each bit in the result is `1` if the corresponding bits in the operands are different; otherwise, it is `0`.

7.3.6 Shift Left (`shl`)

The `shl` instruction performs a left shift operation on an integer operand.

1. **Syntax**

   The syntax for the `shl` instruction is:

```
<result> = shl <type> <op1>, <op2>
```

where:

- `<result>` is the variable that stores the result of the left shift.
- `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
- `<op1>` is the value to be shifted.
- `<op2>` is the number of bits to shift.

2. **Example Usage**

   Here is an example of the `shl` instruction:

```
%result = shl i32 1, 3  ; %result = 1 << 3 = 8
```

   Explanation:

   - Binary representation of `1`: `0001`
   - After shifting left by `3` bits: `1000` (decimal `8`)

3. **Key Points**

   - The `shl` instruction performs a left shift operation.
   - Both operands must be of the same integer type.
   - The result is of the same type as the operands.
   - Bits shifted out of the most significant bit (MSB) are discarded.
   - Zeros are shifted into the least significant bit (LSB).

7.3.7 Logical Shift Right (`lshr`)

The `lshr` instruction performs a logical right shift operation on an integer operand.

1. **Syntax**

   The syntax for the `lshr` instruction is:

   ```
   <result> = lshr <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the logical right shift.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` is the value to be shifted.
   - `<op2>` is the number of bits to shift.

2. **Example Usage**

   Here is an example of the `lshr` instruction:

   ```
   %result = lshr i32 8, 2  ; %result = 8 >> 2 = 2
   ```

   Explanation:

   - Binary representation of `8`: `1000`
   - After shifting right by `2` bits: `0010` (decimal `2`)

3. **Key Points**

   - The `lshr` instruction performs a logical right shift operation.

- Both operands must be of the same integer type.
- The result is of the same type as the operands.
- Bits shifted out of the least significant bit (LSB) are discarded.
- Zeros are shifted into the most significant bit (MSB).

7.3.8 Arithmetic Shift Right (`ashr`)

The `ashr` instruction performs an arithmetic right shift operation on an integer operand.

1. **Syntax**

   The syntax for the `ashr` instruction is:

   ```
   <result> = ashr <type> <op1>, <op2>
   ```

   where:

   - `<result>` is the variable that stores the result of the arithmetic right shift.
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` is the value to be shifted.
   - `<op2>` is the number of bits to shift.

2. **Example Usage**

   Here is an example of the `ashr` instruction:

   ```
   %result = ashr i32 -8, 2  ; %result = -8 >> 2 = -2
   ```

   Explanation:

- Binary representation of `-8` (in two's complement): `11111000`
- After shifting right by `2` bits: `11111110` (decimal `-2`)

3. **Key Points**

- The `ashr` instruction performs an arithmetic right shift operation.
- Both operands must be of the same integer type.
- The result is of the same type as the operands.
- Bits shifted out of the least significant bit (LSB) are discarded.
- The sign bit (MSB) is preserved and shifted into the new MSB positions.

7.3.9 Example: Combining Bitwise Operations

Here is a complete example that demonstrates the use of bitwise operations in LLVM IR:

```
define i32 @bitwise_example(i32 %a, i32 %b) {
  %and_result = and i32 %a, %b          ; %and_result = %a & %b
  %or_result = or i32 %a, %b            ; %or_result = %a | %b
  %xor_result = xor i32 %a, %b          ; %xor_result = %a ^ %b
  %shl_result = shl i32 %a, 2           ; %shl_result = %a << 2
  %lshr_result = lshr i32 %a, 2         ; %lshr_result = %a >> 2 (logical)
  %ashr_result = ashr i32 %a, 2         ; %ashr_result = %a >> 2
  ↪  (arithmetic)
  %final_result = add i32 %and_result, %or_result  ; %final_result =
  ↪  %and_result + %or_result
  ret i32 %final_result
}
```

In this example:

- The function `@bitwise_example` takes two integer arguments (`%a` and `%b`).

- It performs bitwise AND, OR, XOR, left shift, logical right shift, and arithmetic right shift operations on the arguments.
- The results are combined to compute a final result, which is returned.

7.3.10 Key Points

- Bitwise operations (`and`, `or`, `xor`, `shl`, `lshr`, `ashr`) operate on integer types (`iN`).
- Both operands must be of the same integer type.
- The result is of the same type as the operands.
- Shift operations (`shl`, `lshr`, `ashr`) require the shift amount to be of the same type as the value being shifted.
- Logical shifts (`lshr`) fill with zeros, while arithmetic shifts (`ashr`) preserve the sign bit.

7.3.11 Conclusion

Bitwise operations are essential for low-level programming tasks that require precise control over individual bits in integer values. By understanding the syntax, usage, and behavior of these operations, intermediate users can write efficient and accurate LLVM IR code for bit-level manipulations.

7.4 Overflow Handling (`nsw`, `nuw`)

7.4.1 Introduction

Overflow handling is a critical aspect of integer arithmetic in LLVM IR, especially when working with fixed-width integer types. Overflow occurs when the result of an arithmetic operation exceeds the range that can be represented by the integer type. LLVM IR provides mechanisms to handle overflow explicitly using **flags** such as `nsw` (No Signed Wrap) and `nuw` (No Unsigned Wrap). These flags allow developers to specify the expected behavior of arithmetic operations and enable optimizations based on those assumptions. In this section, we will explore the syntax, usage, and key characteristics of overflow handling in LLVM IR, focusing on the `nsw` and `nuw` flags.

7.4.2 What is Overflow?

Overflow occurs when the result of an arithmetic operation cannot be represented within the range of the integer type used. For example:

- **Signed Overflow**: Occurs when the result of a signed arithmetic operation exceeds the maximum or minimum value that can be represented by the signed integer type.
- **Unsigned Overflow**: Occurs when the result of an unsigned arithmetic operation exceeds the maximum value that can be represented by the unsigned integer type.

In LLVM IR, overflow behavior is undefined by default. However, developers can use the `nsw` and `nuw` flags to specify that overflow should not occur, enabling optimizations and ensuring predictable behavior.

7.4.3 No Signed Wrap (`nsw`)

The `nsw` (No Signed Wrap) flag indicates that a signed arithmetic operation does not overflow. If overflow occurs, the behavior is undefined.

1. **Syntax**

   The syntax for using the `nsw` flag is:

   ```
   <result> = <operation> nsw <type> <op1>, <op2>
   ```

   where:

   - `<operation>` is the arithmetic operation (e.g., `add`, `sub`, `mul`).
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands.

2. **Example Usage**

   Here is an example of the `nsw` flag:

   ```
   %result = add nsw i32 %a, %b  ; %result = %a + %b (no signed
   ↪  overflow)
   ```

3. **Key Points**

   - The `nsw` flag applies to signed arithmetic operations (`add`, `sub`, `mul`).
   - It indicates that the operation does not overflow.
   - If overflow occurs, the behavior is undefined.
   - The `nsw` flag enables optimizations, such as simplifying expressions or eliminating redundant checks.

7.4.4 No Unsigned Wrap (`nuw`)

The `nuw` (No Unsigned Wrap) flag indicates that an unsigned arithmetic operation does not overflow. If overflow occurs, the behavior is undefined.

1. **Syntax**

   The syntax for using the `nuw` flag is:

   ```
   <result> = <operation> nuw <type> <op1>, <op2>
   ```

   where:

   - `<operation>` is the arithmetic operation (e.g., `add`, `sub`, `mul`).
   - `<type>` is the integer type of the operands (e.g., `i32`, `i64`).
   - `<op1>` and `<op2>` are the operands.

2. **Example Usage**

   Here is an example of the `nuw` flag:

   ```
   %result = add nuw i32 %a, %b  ; %result = %a + %b (no unsigned
   ↪  overflow)
   ```

3. **Key Points**

   - The `nuw` flag applies to unsigned arithmetic operations (`add`, `sub`, `mul`).
   - It indicates that the operation does not overflow.
   - If overflow occurs, the behavior is undefined.
   - The `nuw` flag enables optimizations, such as simplifying expressions or eliminating redundant checks.

7.4.5 Combining `nsw` and `nuw`

The `nsw` and `nuw` flags can be combined to indicate that an arithmetic operation does not overflow in either signed or unsigned contexts.

1. **Example Usage**

   Here is an example of combining `nsw` and `nuw`:

```
%result = add nsw nuw i32 %a, %b  ; %result = %a + %b (no signed or
↪  unsigned overflow)
```

2. **Key Points**

   - Combining `nsw` and `nuw` ensures that the operation does not overflow in either signed or unsigned contexts.
   - If overflow occurs, the behavior is undefined.
   - This combination enables even more aggressive optimizations.

7.4.6 Example: Using `nsw` and `nuw` in Arithmetic Operations

Here is a complete example that demonstrates the use of `nsw` and `nuw` in arithmetic operations:

```
define i32 @overflow_example(i32 %a, i32 %b) {
  %sum_nsw = add nsw i32 %a, %b          ; %sum_nsw = %a + %b (no signed
  ↪  overflow)
  %diff_nsw = sub nsw i32 %a, %b         ; %diff_nsw = %a - %b (no signed
  ↪  overflow)
  %prod_nsw = mul nsw i32 %a, %b         ; %prod_nsw = %a * %b (no signed
  ↪  overflow)
```

```
  %sum_nuw = add nuw i32 %a, %b          ; %sum_nuw = %a + %b (no unsigned
  ↪  overflow)
  %diff_nuw = sub nuw i32 %a, %b         ; %diff_nuw = %a - %b (no
  ↪  unsigned overflow)
  %prod_nuw = mul nuw i32 %a, %b         ; %prod_nuw = %a * %b (no
  ↪  unsigned overflow)
  %final_result = add i32 %sum_nsw, %sum_nuw  ; %final_result = %sum_nsw +
  ↪  %sum_nuw
  ret i32 %final_result
}
```

In this example:

- The function `@overflow_example` takes two integer arguments (`%a` and `%b`).
- It performs addition, subtraction, and multiplication with both `nsw` and `nuw` flags.
- The results are combined to compute a final result, which is returned.

7.4.7 Key Points

- The `nsw` flag indicates that a signed arithmetic operation does not overflow.
- The `nuw` flag indicates that an unsigned arithmetic operation does not overflow.
- Both flags enable optimizations and ensure predictable behavior.
- If overflow occurs with `nsw` or `nuw`, the behavior is undefined.
- The flags can be combined for operations that should not overflow in either signed or unsigned contexts.

7.4.8 Practical Use Cases

1. **Loop Induction Variables**

   Using `nsw` or `nuw` with loop induction variables can enable optimizations, such as loop unrolling or strength reduction.

   Example:

```
define void @loop_example(i32 %n) {
entry:
  %i = alloca i32, align 4
  store i32 0, ptr %i
  br label %loop
loop:
  %current_i = load i32, ptr %i
  %cmp = icmp slt i32 %current_i, %n
  br i1 %cmp, label %loop_body, label %loop_exit
loop_body:
  ; Loop body instructions
  %next_i = add nsw i32 %current_i, 1  ; No signed overflow
  store i32 %next_i, ptr %i
  br label %loop
loop_exit:
  ret void
}
```

2. **Bounds Checking**

   Using `nsw` or `nuw` can eliminate the need for explicit bounds checking in some cases, improving performance.

   Example:

```
define i32 @safe_add(i32 %a, i32 %b) {
  %sum = add nsw i32 %a, %b  ; No signed overflow
  ret i32 %sum
}
```

7.4.9 Conclusion

Overflow handling using the `nsw` and `nuw` flags is a powerful feature of LLVM IR that enables optimizations and ensures predictable behavior for integer arithmetic operations. By understanding how to use these flags, intermediate users can write efficient and correct LLVM IR code that avoids undefined behavior due to overflow.

Chapter 8

Function Definitions & Calls

8.1 Declaring and Defining Functions (`define i32 @func()`)

8.1.1 Introduction

Functions are a fundamental building block of LLVM IR, enabling modular and reusable code. A function in LLVM IR consists of a declaration (signature) and a definition (implementation). The declaration specifies the function's name, return type, and parameter types, while the definition includes the function's body, which is a sequence of instructions. In this section, we will explore the syntax, usage, and key characteristics of declaring and defining functions in LLVM IR. Understanding how to work with functions is essential for intermediate users who want to write modular and efficient LLVM IR code.

8.1.2 Function Declaration

A function declaration in LLVM IR specifies the function's signature, including its name, return type, and parameter types. It does not include the function's body. Function declarations are used to inform the compiler about functions that are defined elsewhere (e.g., in another module or external library).

Syntax

The syntax for a function declaration is:

```
declare <return_type> @<function_name>(<parameter_types>)
```

where:

- `<return_type>` is the type of the value returned by the function (e.g., `i32`, `void`).
- `<function_name>` is the name of the function, prefixed with `@`.
- `<parameter_types>` is a comma-separated list of the types of the function's parameters.

Example Usage

Here is an example of a function declaration:

```
declare i32 @printf(ptr, ...)  ; Declare the printf function
```

In this example:

- The function `@printf` returns an `i32` value.
- It takes a `ptr` (pointer) as its first argument and accepts additional arguments via the `...` syntax (varargs).

Key Points

- Function declarations are used to declare functions that are defined elsewhere.
- They do not include a function body.
- Function names are prefixed with `@`.
- Parameter types are specified as a comma-separated list.

8.1.3 Function Definition

A function definition in LLVM IR includes both the function's signature and its body. The body consists of a sequence of instructions that implement the function's logic.

Syntax

The syntax for a function definition is:

```
define <return_type> @<function_name>(<parameter_list>) {
  <function_body>
}
```

where:

- `<return_type>` is the type of the value returned by the function (e.g., `i32`, `void`).
- `<function_name>` is the name of the function, prefixed with `@`.
- `<parameter_list>` is a comma-separated list of parameters, each specified as `<type> <name>`.
- `<function_body>` is a sequence of instructions that implement the function's logic.

Example Usage

Here is an example of a function definition:

```
define i32 @add(i32 %a, i32 %b) {
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

In this example:

- The function `@add` returns an `i32` value.
- It takes two parameters: `%a` (type `i32`) and `%b` (type `i32`).
- The function body adds the two parameters and returns the result.

Key Points

- Function definitions include both the function's signature and its body.
- Function names are prefixed with `@`.
- Parameters are specified as `<type> <name>`.
- The function body consists of a sequence of instructions, ending with a terminator instruction (e.g., `ret`).

8.1.4 Function Attributes

Function attributes provide additional information about a function's behavior, such as its visibility, linkage, or optimization hints. Attributes are specified using the `attributes` keyword.

Syntax

The syntax for specifying function attributes is:

```
define <return_type> @<function_name>(<parameter_list>)
↪  #<attribute_number> {
  <function_body>
}

attributes #<attribute_number> = { <attribute_list> }
```

where:

- `<attribute_number>` is a unique identifier for the attribute set.
- `<attribute_list>` is a list of attributes, separated by spaces.

Example Usage

Here is an example of a function with attributes:

```
define i32 @square(i32 %x) #0 {
  %result = mul i32 %x, %x
  ret i32 %result
}

attributes #0 = { nounwind readonly }
```

In this example:

- The function `@square` has the attributes `nounwind` (indicates that the function does not throw exceptions) and `readonly` (indicates that the function does not modify memory).

Common Attributes

- `nounwind`: The function does not throw exceptions.
- `readonly`: The function does not modify memory.
- `inlinehint`: Suggests that the function should be inlined.
- `noinline`: Prevents the function from being inlined.
- `optnone`: Disables optimizations for the function.

8.1.5 Parameter Attributes

Parameter attributes provide additional information about a function's parameters, such as their alignment or whether they are read-only.

Syntax
The syntax for specifying parameter attributes is:

```
define <return_type> @<function_name>(<type> <name> <parameter_attributes>)
↪ {
  <function_body>
}
```

where:

- `<parameter_attributes>` is a list of attributes, separated by spaces.

Example Usage
Here is an example of a function with parameter attributes:

```
define i32 @increment(i32 %x align 4) {
  %result = add i32 %x, 1
  ret i32 %result
}
```

In this example:

- The parameter `%x` has the `align 4` attribute, indicating that it should be aligned to 4 bytes.

Common Parameter Attributes

- `align <n>`: Specifies the alignment of the parameter.
- `readonly`: Indicates that the parameter is read-only.
- `nocapture`: Indicates that the parameter is not captured by the function.

8.1.6 Example: Declaring and Defining Functions

Here is a complete example that demonstrates declaring and defining functions in LLVM IR:

```
declare i32 @printf(ptr, ...)  ; Declare the printf function

define i32 @add(i32 %a, i32 %b) {
  %sum = add i32 %a, %b
  ret i32 %sum
}

define void @print_sum(i32 %a, i32 %b) {
  %sum = call i32 @add(i32 %a, i32 %b)  ; Call the add function
  %format = getelementptr inbounds [4 x i8], ptr @.str, i32 0, i32 0
```

```
  call i32 (ptr, ...) @printf(ptr %format, i32 %sum)  ; Call printf
  ret void
}

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
```

In this example:

- The `@printf` function is declared.
- The `@add` function is defined to add two integers and return the result.
- The `@print_sum` function calls `@add` and prints the result using `@printf`.
- The `@.str` global variable stores the format string for `printf`.

8.1.7 Key Points

- Function declarations specify the function's signature without a body.
- Function definitions include both the function's signature and its body.
- Function attributes provide additional information about the function's behavior.
- Parameter attributes provide additional information about the function's parameters.
- Functions are prefixed with `@` and can be called using the `call` instruction.

8.1.8 Conclusion

Declaring and defining functions is a fundamental aspect of LLVM IR, enabling modular and reusable code. By understanding the syntax, usage, and key characteristics of functions, intermediate users can write efficient and maintainable LLVM IR code.

8.2 Calling Functions (`call i32 @add(i32 %a, i32 %b)`)

8.2.1 Introduction

Function calls are a fundamental mechanism in LLVM IR for invoking functions and passing arguments. They allow modular and reusable code by enabling one function to call another. In this section, we will explore the syntax, usage, and key characteristics of function calls in LLVM IR, focusing on the `call` instruction. Understanding how to call functions is essential for intermediate users who want to write modular and efficient LLVM IR code.

8.2.2 The `call` Instruction

The `call` instruction is used to invoke a function in LLVM IR. It specifies the function to be called, the arguments to pass, and the return value (if any).

Syntax

The syntax for the `call` instruction is:

```
<result> = call <return_type> @<function_name>(<argument_list>)
```

where:

- `<result>` is the variable that stores the return value of the function (if any).
- `<return_type>` is the type of the value returned by the function (e.g., `i32`, `void`).
- `<function_name>` is the name of the function to call, prefixed with `@`.
- `<argument_list>` is a comma-separated list of arguments to pass to the function.

For functions that do not return a value (i.e., `void` functions), the syntax is:

```
call void @<function_name>(<argument_list>)
```

Example Usage

Here are some examples of the `call` instruction:

1. Calling a function that returns a value:

```
%sum = call i32 @add(i32 %a, i32 %b)  ; Call the @add function and store
↪  the result in %sum
```

1. Calling a function that does not return a value:

```
call void @print(i32 %x)  ; Call the @print function with argument %x
```

Key Points

- The `call` instruction is used to invoke a function.
- The return value (if any) is stored in a variable.
- Arguments are passed as a comma-separated list.
- Functions that do not return a value are called using `call void`.

8.2.3 Passing Arguments to Functions

Arguments are passed to functions in LLVM IR by specifying them in the `<argument_list>` of the `call` instruction. Each argument must match the type expected by the function's parameter.

Example Usage

Here is an example of passing arguments to a function:

```
define i32 @add(i32 %a, i32 %b) {
  %sum = add i32 %a, %b
  ret i32 %sum
}

define i32 @main() {
  %result = call i32 @add(i32 5, i32 10)  ; Call @add with arguments 5 and
  ↪  10
  ret i32 %result
}
```

In this example:

- The `@add` function is called with the arguments `5` and `10`.
- The result of the function call is stored in `%result`.

Key Points

- Arguments must match the types expected by the function's parameters.
- Constants, variables, and expressions can be passed as arguments.

8.2.4 Calling External Functions

External functions (e.g., library functions) can be called in LLVM IR by declaring them first and then using the `call` instruction.

Example Usage

Here is an example of calling an external function:

```
declare i32 @printf(ptr, ...)  ; Declare the printf function

define void @print_message() {
  %message = getelementptr inbounds [13 x i8], ptr @.str, i32 0, i32 0
  call i32 (ptr, ...) @printf(ptr %message)  ; Call printf with the
  ↪  message
  ret void
}

@.str = private unnamed_addr constant [13 x i8] c"Hello, World\00", align
↪  1
```

In this example:

- The `@printf` function is declared as an external function.
- The `@print_message` function calls `@printf` with a string argument.

Key Points

- External functions must be declared before they can be called.
- The declaration specifies the function's signature, including its return type and parameter types.

8.2.5 Tail Calls

A tail call is a function call that occurs as the last action in a function. Tail calls can be optimized by the compiler to reuse the current function's stack frame, reducing memory usage and improving performance.

Syntax

The syntax for a tail call is:

```
tail call <return_type> @<function_name>(<argument_list>)
```

Example Usage

Here is an example of a tail call:

```
define i32 @factorial(i32 %n, i32 %acc) {
  %cmp = icmp eq i32 %n, 0
  br i1 %cmp, label %base_case, label %recursive_case
base_case:
  ret i32 %acc
recursive_case:
  %new_n = sub i32 %n, 1
  %new_acc = mul i32 %n, %acc
  %result = tail call i32 @factorial(i32 %new_n, i32 %new_acc)  ; Tail
  ↪  call
  ret i32 %result
}
```

In this example:

- The `@factorial` function uses a tail call to optimize the recursive call.

Key Points

- Tail calls occur as the last action in a function.
- They can be optimized to reuse the current function's stack frame.
- The `tail` keyword is used to indicate a tail call.

8.2.6 Example: Function Calls in a Program

Here is a complete example that demonstrates function calls in LLVM IR:

```
declare i32 @printf(ptr, ...)  ; Declare the printf function

define i32 @add(i32 %a, i32 %b) {
  %sum = add i32 %a, %b
  ret i32 %sum
}

define void @print_sum(i32 %a, i32 %b) {
  %sum = call i32 @add(i32 %a, i32 %b)  ; Call the add function
  %format = getelementptr inbounds [4 x i8], ptr @.str, i32 0, i32 0
  call i32 (ptr, ...) @printf(ptr %format, i32 %sum)  ; Call printf
  ret void
}

define i32 @main() {
  call void @print_sum(i32 5, i32 10)  ; Call print_sum with arguments 5
  ↪  and 10
  ret i32 0
}

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
```

In this example:

- The `@printf` function is declared as an external function.
- The `@add` function adds two integers and returns the result.
- The `@print_sum` function calls `@add` and prints the result using `@printf`.
- The `@main` function calls `@print_sum` with the arguments `5` and `10`.

8.2.7 Key Points

- The `call` instruction is used to invoke functions in LLVM IR.
- Arguments are passed as a comma-separated list.
- External functions must be declared before they can be called.
- Tail calls can be optimized to reuse the current function's stack frame.
- Function calls enable modular and reusable code.

8.2.8 Conclusion

Function calls are a powerful mechanism in LLVM IR for invoking functions and passing arguments. By understanding the syntax, usage, and key characteristics of function calls, intermediate users can write modular and efficient LLVM IR code.

8.3 Inline Assembly (`call void asm "nop", ""()`)

8.3.1 Introduction

Inline assembly is a powerful feature in LLVM IR that allows developers to embed architecture-specific assembly code directly within LLVM IR. This is particularly useful for low-level programming tasks, such as accessing hardware-specific features, optimizing performance-critical code, or implementing functionality that cannot be expressed in LLVM IR. In this section, we will explore the syntax, usage, and key characteristics of inline assembly in LLVM IR. Understanding inline assembly is essential for intermediate users who need to write highly optimized or hardware-specific code.

8.3.2 What is Inline Assembly?

Inline assembly enables the embedding of assembly code within LLVM IR. The assembly code is written as a string and is passed to the `asm` keyword, which is used in conjunction with the `call` instruction. The LLVM compiler does not interpret or optimize the inline assembly code; instead, it passes it directly to the target assembler.

Syntax

The syntax for inline assembly is:

```
call <return_type> asm <assembly_code>, <constraints>(<arguments>)
```

where:

- `<return_type>` is the type of the value returned by the assembly code (e.g., `void`, `i32`).
- `<assembly_code>` is a string containing the assembly instructions.

- `<constraints>` is a string specifying the constraints for the assembly code (e.g., input/output operands).
- `<arguments>` is a list of arguments passed to the assembly code.

For assembly code that does not return a value, the syntax is:

```
call void asm <assembly_code>, <constraints>(<arguments>)
```

Example Usage

Here is an example of inline assembly:

```
call void asm "nop", ""()  ; Inline assembly for a no-op instruction
```

In this example:

- The `nop` instruction is a no-op (no operation) that does nothing.
- The empty string `""` indicates that there are no constraints or arguments.

8.3.3 Key Characteristics of Inline Assembly

1. **Assembly Code**

   The assembly code is written as a string and must be valid for the target architecture. For example, `nop` is a valid instruction for x86 architectures but may not be valid for other architectures.

2. **Constraints**

   Constraints specify how the assembly code interacts with LLVM IR values. They define:

   - Input operands: Values passed to the assembly code.

- Output operands: Values returned by the assembly code.
- Clobbered registers: Registers that the assembly code modifies.

Constraints are specified using a syntax similar to GCC inline assembly.

3. **Arguments**

   Arguments are LLVM IR values passed to the assembly code. They must match the constraints specified in the inline assembly statement.

8.3.4 Using Inline Assembly

Basic Example

Here is a basic example of inline assembly that performs a no-op:

```
call void asm "nop", ""()  ; Inline assembly for a no-op instruction
```

Example with Input and Output Operands

Here is an example of inline assembly with input and output operands:

```
define i32 @add(i32 %a, i32 %b) {
  %result = call i32 asm "addl $1, $0", "=r,r"(i32 %a, i32 %b)
  ret i32 %result
}
```

In this example:

- The `addl` instruction adds the two input operands (`%a` and `%b`).
- The constraint `"=r,r"` specifies that the first operand is an output (`=r`) and the second and third operands are inputs (`r`).

Example with Clobbered Registers

Here is an example of inline assembly that clobbers a register:

```
define void @example() {
  call void asm "movl $$0, %eax", "~{eax}"()  ; Clobber the %eax register
  ret void
}
```

In this example:

- The `movl` instruction sets the `%eax` register to `0`.
- The constraint `"~{eax}"` indicates that the `%eax` register is clobbered.

8.3.5 Constraints Syntax

Constraints are specified using a string with the following syntax:

- `=r`: Output operand (write-only).
- `r`: Input operand (read-only).
- `+r`: Input/output operand (read-write).
- `~{<register>}`: Clobbered register.

Example: Multiple Constraints

Here is an example of inline assembly with multiple constraints:

```
define i32 @multiply(i32 %a, i32 %b) {
  %result = call i32 asm "imull $1, $0", "=r,r,~{eax}"(i32 %a, i32 %b)
  ret i32 %result
}
```

In this example:

- The `imull` instruction multiplies the two input operands (`%a` and `%b`).
- The constraint `"=r,r,~{eax}"` specifies that the first operand is an output, the second and third operands are inputs, and the `%eax` register is clobbered.

8.3.6 Example: Inline Assembly in a Program

Here is a complete example that demonstrates the use of inline assembly in LLVM IR:

```
define i32 @add(i32 %a, i32 %b) {
  %result = call i32 asm "addl $1, $0", "=r,r"(i32 %a, i32 %b)
  ret i32 %result
}

define i32 @multiply(i32 %a, i32 %b) {
  %result = call i32 asm "imull $1, $0", "=r,r,~{eax}"(i32 %a, i32 %b)
  ret i32 %result
}

define void @example() {
  call void asm "nop", ""()  ; No-op instruction
  ret void
}

define i32 @main() {
  %sum = call i32 @add(i32 5, i32 10)  ; Call the add function
  %product = call i32 @multiply(i32 %sum, i32 2)  ; Call the multiply
  ↪  function
  call void @example()  ; Call the example function
  ret i32 %product
}
```

In this example:

- The `@add` function uses inline assembly to add two integers.
- The `@multiply` function uses inline assembly to multiply two integers.
- The `@example` function uses inline assembly to execute a no-op instruction.
- The `@main` function calls the other functions and returns the result.

8.3.7 Key Points

- Inline assembly allows embedding architecture-specific assembly code within LLVM IR.
- The `asm` keyword is used to specify the assembly code and constraints.
- Constraints define input/output operands and clobbered registers.
- Inline assembly is useful for low-level programming and performance optimizations.
- The LLVM compiler does not interpret or optimize inline assembly code.

8.3.8 Conclusion

Inline assembly is a powerful feature in LLVM IR that enables developers to write highly optimized or hardware-specific code. By understanding the syntax, usage, and key characteristics of inline assembly, intermediate users can leverage this feature to achieve fine-grained control over their programs.

8.4 Function Attributes (`alwaysinline`, `noreturn`, `nounwind`)

8.4.1 Introduction

Function attributes in LLVM IR provide additional information about a function's behavior, enabling optimizations and ensuring correctness. Attributes can specify properties such as whether a function should always be inlined, whether it never returns, or whether it does not throw exceptions. In this section, we will explore the syntax, usage, and key characteristics of function attributes in LLVM IR, focusing on commonly used attributes like `alwaysinline`, `noreturn`, and `nounwind`. Understanding function attributes is essential for intermediate users who want to write efficient and correct LLVM IR code.

8.4.2 What Are Function Attributes?

Function attributes are metadata attached to function declarations or definitions that describe specific properties or behaviors of the function. These attributes are used by the LLVM optimizer to make informed decisions about code transformations and optimizations.

Syntax

Function attributes are specified using the `attributes` keyword and are attached to a function definition or declaration. The syntax is:

```
define <return_type> @<function_name>(<parameter_list>)
 ↪  #<attribute_number> {
  <function_body>
}

attributes #<attribute_number> = { <attribute_list> }
```

where:

- `<attribute_number>` is a unique identifier for the attribute set.
- `<attribute_list>` is a list of attributes, separated by spaces.

Alternatively, attributes can be specified directly in the function definition:

```
define <return_type> @<function_name>(<parameter_list>) <attribute_list> {
  <function_body>
}
```

Example Usage

Here is an example of a function with attributes:

```
define i32 @square(i32 %x) #0 {
  %result = mul i32 %x, %x
  ret i32 %result
}

attributes #0 = { alwaysinline nounwind }
```

In this example:

- The `@square` function has the attributes `alwaysinline` and `nounwind`.

8.4.3 Commonly Used Function Attributes

1. **`alwaysinline`**

   The `alwaysinline` attribute forces the LLVM optimizer to inline the function at every call site, regardless of the optimization level.

Syntax

```
define <return_type> @<function_name>(<parameter_list>) alwaysinline
↪ {
  <function_body>
}
```

Example Usage

Here is an example of the `alwaysinline` attribute:

```
define i32 @add(i32 %a, i32 %b) alwaysinline {
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

In this example:

- The `@add` function is marked with the `alwaysinline` attribute, ensuring it is inlined at every call site.

Key Points

- The `alwaysinline` attribute forces inlining of the function.
- It is useful for small, frequently called functions where inlining improves performance.

2. **`noreturn`**

The `noreturn` attribute indicates that the function does not return to its caller. This is typically used for functions that terminate the program or enter an infinite loop.

Syntax

```
define void @<function_name>(<parameter_list>) noreturn {
  <function_body>
}
```

Example Usage

Here is an example of the `noreturn` attribute:

```
define void @exit_program() noreturn {
  call void @exit(i32 1)  ; Call the exit function
  unreachable  ; Indicate that control never reaches this point
}
```

In this example:

- The `@exit_program` function is marked with the `noreturn` attribute, indicating it does not return.
- The `unreachable` instruction is used to indicate that control never reaches this point.

Key Points

- The `noreturn` attribute indicates that the function does not return.
- It is used for functions that terminate the program or enter an infinite loop.

3. **`nounwind`**

The `nounwind` attribute indicates that the function does not throw exceptions. This allows the optimizer to eliminate unnecessary exception-handling code.

Syntax

```
define <return_type> @<function_name>(<parameter_list>) nounwind {
  <function_body>
}
```

Example Usage

Here is an example of the `nounwind` attribute:

```
define i32 @safe_add(i32 %a, i32 %b) nounwind {
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

In this example:

- The `@safe_add` function is marked with the `nounwind` attribute, indicating it does not throw exceptions.

Key Points

- The `nounwind` attribute indicates that the function does not throw exceptions.
- It enables optimizations by eliminating unnecessary exception-handling code.

8.4.4 Other Common Function Attributes

1. **`readonly`**

   The `readonly` attribute indicates that the function does not modify memory.

   Example Usage

```
define i32 @get_value(ptr %ptr) readonly {
  %value = load i32, ptr %ptr
  ret i32 %value
}
```

2. **`readnone`**

   The `readnone` attribute indicates that the function does not read or modify memory.

   Example Usage

```
define i32 @pure_function(i32 %a, i32 %b) readnone {
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

3. **`noinline`**

   The `noinline` attribute prevents the function from being inlined.

   Example Usage

```
define i32 @large_function(i32 %a, i32 %b) noinline {
  ; Complex logic here
  ret i32 %result
}
```

4. **optnone**

   The `optnone` attribute disables optimizations for the function.

   Example Usage

```
define i32 @unoptimized_function(i32 %a, i32 %b) optnone {
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

8.4.5 Example: Using Function Attributes

Here is a complete example that demonstrates the use of function attributes in LLVM IR:

```
define i32 @add(i32 %a, i32 %b) alwaysinline nounwind {
  %sum = add i32 %a, %b
  ret i32 %sum
}

define void @exit_program() noreturn {
  call void @exit(i32 1)  ; Call the exit function
  unreachable  ; Indicate that control never reaches this point
}
```

```
define i32 @get_value(ptr %ptr) readonly {
  %value = load i32, ptr %ptr
  ret i32 %value
}

define i32 @main() {
  %sum = call i32 @add(i32 5, i32 10)  ; Call the add function
  %value = call i32 @get_value(ptr @global_var)  ; Call the get_value
  ↪  function
  call void @exit_program()  ; Call the exit_program function
  ret i32 0
}

@global_var = global i32 42, align 4
```

In this example:

- The `@add` function is marked with `alwaysinline` and `nounwind`.
- The `@exit_program` function is marked with `noreturn`.
- The `@get_value` function is marked with `readonly`.
- The `@main` function calls the other functions and returns a value.

8.4.6 Key Points

- Function attributes provide additional information about a function's behavior.
- Common attributes include `alwaysinline`, `noreturn`, and `nounwind`.
- Attributes enable optimizations and ensure correctness.

- Attributes can be specified using the `attributes` keyword or directly in the function definition.

8.4.7 Conclusion

Function attributes are a powerful feature in LLVM IR that enable developers to provide additional information about a function's behavior. By understanding the syntax, usage, and key characteristics of function attributes, intermediate users can write efficient and correct LLVM IR code.

Chapter 9

Exception Handling (EH) & Unwinding

9.1 Landing Pads (`landingpad { i8*, i32 } catch i8* null`)

9.1.1 Introduction

Exception handling is a critical feature in many programming languages, enabling developers to manage runtime errors and unexpected conditions gracefully. In LLVM IR, exception handling is implemented using **landing pads**, which are special blocks of code that handle exceptions thrown during program execution. Landing pads are part of the LLVM IR's exception handling mechanism, which includes `landingpad` instructions, `invoke` instructions, and personality functions. In this section, we will explore the syntax, usage, and key characteristics of landing pads in LLVM IR. Understanding landing pads is essential for intermediate users who want to implement or analyze exception handling in LLVM IR.

9.1.2 What is a Landing Pad?

A landing pad is a basic block in LLVM IR that handles exceptions thrown during the execution of an `invoke` instruction. It is responsible for catching exceptions, performing cleanup actions, and resuming normal execution or propagating the exception further up the call stack.
The `landingpad` instruction is used to define a landing pad. It specifies the types of exceptions that can be caught and provides a mechanism for accessing the exception object and selector value.

9.1.3 Syntax of the `landingpad` Instruction

The syntax for the `landingpad` instruction is:

```
<result> = landingpad <type> <clause_list>
```

where:

- `<result>` is a value of type `{ i8*, i32 }`, which represents the exception object and selector value.
- `<type>` is the type of the result, typically `{ i8*, i32 }`.
- `<clause_list>` is a list of clauses that specify the types of exceptions that can be caught.

The `<clause_list>` can include:

- `catch <type> <value>`: Specifies a catch clause for a specific exception type.
- `filter <type> <value>`: Specifies a filter clause for a set of exception types.

Example Usage

Here is an example of a `landingpad` instruction:

```
%lp = landingpad { i8*, i32 } catch i8* null
```

In this example:

- The `landingpad` instruction catches all exceptions (`catch i8* null`).
- The result is a struct of type `{ i8*, i32 }`, where:
  - `i8*` is a pointer to the exception object.
  - `i32` is the selector value, which identifies the type of exception.

9.1.4 Key Components of Landing Pads

1. **Exception Object**

   The exception object is a pointer to the thrown exception. It is typically of type `i8*` and can be cast to the appropriate exception type for further processing.

2. **Selector Value**

   The selector value is an integer (`i32`) that identifies the type of exception. It is used to determine which catch clause should handle the exception.

3. **Personality Function**

   The personality function is a runtime function that handles the mechanics of exception handling, such as unwinding the stack and invoking the appropriate landing pad. It is specified using the `personality` keyword in the function definition.

9.1.5 Example: Using Landing Pads

Here is a complete example that demonstrates the use of landing pads in LLVM IR:

```
declare void @throw_exception()  ; Declare a function that throws an
↪  exception

define i32 @main() personality ptr @__gxx_personality_v0 {
entry:
  invoke void @throw_exception()
          to label %normal unwind label %landing_pad

normal:
  ret i32 0

landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* null  ; Catch all exceptions
  %exception_ptr = extractvalue { i8*, i32 } %lp, 0  ; Extract the
  ↪  exception object
  %selector = extractvalue { i8*, i32 } %lp, 1  ; Extract the selector
  ↪  value
  call void @handle_exception(i8* %exception_ptr)  ; Handle the exception
  ret i32 1
}

declare i32 @__gxx_personality_v0(...)  ; Personality function for C++
↪  exceptions

declare void @handle_exception(i8*)  ; Declare a function to handle
↪  exceptions
```

In this example:

- The `@main` function uses an `invoke` instruction to call `@throw_exception`.
- If an exception is thrown, control is transferred to the `landing_pad` block.
- The `landingpad` instruction catches all exceptions (`catch i8* null`).
- The exception object and selector value are extracted using `extractvalue`.
- The `@handle_exception` function is called to handle the exception.

9.1.6 Key Points

- Landing pads are basic blocks that handle exceptions thrown during the execution of an `invoke` instruction.
- The `landingpad` instruction specifies the types of exceptions that can be caught.
- The result of a `landingpad` instruction is a struct of type `{ i8*, i32 }`, representing the exception object and selector value.
- The personality function is a runtime function that handles the mechanics of exception handling.

9.1.7 Advanced Topics

1. **Catch Clauses**

   Catch clauses specify the types of exceptions that can be caught by the landing pad. Each catch clause is associated with a specific exception type.

   Example:

```
%lp = landingpad { i8*, i32 }
        catch i8* @ExceptionType1
        catch i8* @ExceptionType2
```

2. **Filter Clauses**

   Filter clauses specify a set of exception types that should be filtered out. If an exception matches one of the types in the filter, it is not caught by the landing pad.

   Example:

```
%lp = landingpad { i8*, i32 }
        filter [1 x i8*] [i8* @ExceptionType1]
```

3. **Cleanup Actions**

   Landing pads can also include cleanup actions, which are executed regardless of whether an exception is caught. Cleanup actions are typically used to release resources or perform other necessary cleanup.

   Example:

```
landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* null
  call void @cleanup()  ; Perform cleanup actions
  resume { i8*, i32 } %lp  ; Resume exception propagation
```

9.1.8 Example: Advanced Landing Pad Usage

Here is an example that demonstrates advanced landing pad usage, including catch clauses and cleanup actions:

```
declare void @throw_exception()  ; Declare a function that throws an
↪  exception

define i32 @main() personality ptr @__gxx_personality_v0 {
entry:
  invoke void @throw_exception()
          to label %normal unwind label %landing_pad

normal:
  ret i32 0

landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* @ExceptionType1
          catch i8* @ExceptionType2
          filter [1 x i8*] [i8* @ExceptionType3]
  %exception_ptr = extractvalue { i8*, i32 } %lp, 0  ; Extract the
  ↪  exception object
  %selector = extractvalue { i8*, i32 } %lp, 1  ; Extract the selector
  ↪  value
  call void @cleanup()  ; Perform cleanup actions
  %is_filtered = icmp eq i32 %selector, 0  ; Check if the exception is
  ↪  filtered
  br i1 %is_filtered, label %resume, label %handle_exception

handle_exception:
  call void @handle_exception(i8* %exception_ptr)  ; Handle the exception
  ret i32 1
```

```
resume:
  resume { i8*, i32 } %lp  ; Resume exception propagation
}

declare i32 @__gxx_personality_v0(...)  ; Personality function for C++
↪  exceptions

declare void @cleanup()  ; Declare a function to perform cleanup

declare void @handle_exception(i8*)  ; Declare a function to handle
↪  exceptions

@ExceptionType1 = external global i8  ; Exception type 1
@ExceptionType2 = external global i8  ; Exception type 2
@ExceptionType3 = external global i8  ; Exception type 3
```

In this example:

- The `landingpad` instruction catches exceptions of type `@ExceptionType1` and `@ExceptionType2`.
- Exceptions of type `@ExceptionType3` are filtered out.
- Cleanup actions are performed using the `@cleanup` function.
- The exception is either handled or propagated further up the call stack.

9.1.9 Key Points

- Landing pads handle exceptions thrown during the execution of an `invoke` instruction.
- The `landingpad` instruction specifies catch and filter clauses.

- The result of a `landingpad` instruction is a struct of type `{ i8*, i32 }`.
- Cleanup actions can be performed in the landing pad.
- The personality function handles the mechanics of exception handling.

9.1.10 Conclusion

Landing pads are a critical component of exception handling in LLVM IR, enabling developers to manage runtime errors and unexpected conditions gracefully. By understanding the syntax, usage, and key characteristics of landing pads, intermediate users can implement and analyze exception handling in LLVM IR effectively.

9.2 Personality Functions (`personality i32 (...)* @__gxx_personality_v0`)

9.2.1 Introduction

Personality functions are a critical component of exception handling in LLVM IR.
They are runtime functions that handle the mechanics of exception handling, such as unwinding the stack, invoking landing pads, and managing exception objects. In LLVM IR, personality functions are specified using the `personality` keyword in function definitions. Understanding personality functions is essential for intermediate users who want to implement or analyze exception handling in LLVM IR. In this section, we will explore the syntax, usage, and key characteristics of personality functions, focusing on their role in exception handling and unwinding.

9.2.2 What is a Personality Function?

A personality function is a runtime function that manages the process of exception handling. It is responsible for:

1. **Unwinding the Stack**: When an exception is thrown, the personality function unwinds the stack to find the appropriate landing pad.
2. **Invoking Landing Pads**: The personality function transfers control to the landing pad that handles the exception.
3. **Managing Exception Objects**: The personality function provides access to the exception object and selector value.

In LLVM IR, personality functions are typically provided by the runtime environment (e.g.,

the C++ runtime for C++ exceptions). The most commonly used personality function for C++ exceptions is `@__gxx_personality_v0`.

9.2.3 Syntax of Personality Functions

The syntax for specifying a personality function in LLVM IR is:

```
define <return_type> @<function_name>(<parameter_list>) personality i32
↪  (...)* @<personality_function> {
  <function_body>
}
```

where:

- `<return_type>` is the type of the value returned by the function.
- `<function_name>` is the name of the function.
- `<parameter_list>` is a list of parameters.
- `@<personality_function>` is the personality function (e.g., `@__gxx_personality_v0`).
- `<function_body>` is the body of the function.

Example Usage

Here is an example of a function with a personality function:

```
define i32 @main() personality i32 (...)* @__gxx_personality_v0 {
  ; Function body with exception handling
  ret i32 0
}
```

In this example:

- The `@main` function uses the `@__gxx_personality_v0` personality function for exception handling.

9.2.4 Role of Personality Functions in Exception Handling

1. **Unwinding the Stack**

   When an exception is thrown, the personality function unwinds the stack to find the appropriate landing pad. This involves:

   - Identifying the call stack frames that have landing pads.
   - Transferring control to the landing pad that can handle the exception.

2. **Invoking Landing Pads**

   The personality function transfers control to the landing pad that handles the exception. The landing pad is responsible for catching the exception, performing cleanup actions, and resuming normal execution or propagating the exception further up the call stack.

3. **Managing Exception Objects**

   The personality function provides access to the exception object and selector value, which are used by the landing pad to determine how to handle the exception.

9.2.5 Example: Using Personality Functions

Here is a complete example that demonstrates the use of personality functions in LLVM IR:

```
declare void @throw_exception()  ; Declare a function that throws an
↪  exception

define i32 @main() personality i32 (...)* @__gxx_personality_v0 {
entry:
  invoke void @throw_exception()
          to label %normal unwind label %landing_pad

normal:
  ret i32 0

landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* null  ; Catch all exceptions
  %exception_ptr = extractvalue { i8*, i32 } %lp, 0  ; Extract the
  ↪  exception object
  %selector = extractvalue { i8*, i32 } %lp, 1  ; Extract the selector
  ↪  value
  call void @handle_exception(i8* %exception_ptr)  ; Handle the exception
  ret i32 1
}

declare i32 @__gxx_personality_v0(...)  ; Personality function for C++
↪  exceptions

declare void @handle_exception(i8*)  ; Declare a function to handle
↪  exceptions
```

In this example:

- The `@main` function uses the `@__gxx_personality_v0` personality function for exception handling.

- The `invoke` instruction calls `@throw_exception` and specifies a landing pad for exception handling.
- The `landingpad` instruction catches all exceptions (`catch i8* null`).
- The exception object and selector value are extracted using `extractvalue`.
- The `@handle_exception` function is called to handle the exception.

9.2.6 Key Points

- Personality functions are runtime functions that manage the mechanics of exception handling.
- They are specified using the `personality` keyword in function definitions.
- The most commonly used personality function for C++ exceptions is `@__gxx_personality_v0`.
- Personality functions handle stack unwinding, invoke landing pads, and manage exception objects.

9.2.7 Advanced Topics

1. **Custom Personality Functions**

   While `@__gxx_personality_v0` is the standard personality function for C++ exceptions, custom personality functions can be used for other exception handling mechanisms.

   Example:

```
define i32 @main() personality i32 (...)* @custom_personality {
  ; Function body with exception handling
  ret i32 0
}

declare i32 @custom_personality(...)  ; Custom personality function
```

2. **Multiple Personality Functions**

   A single LLVM IR module can use multiple personality functions for different exception handling mechanisms.

   Example:

```
define i32 @cpp_function() personality i32 (...)*
↪  @__gxx_personality_v0 {
  ; Function body with C++ exception handling
  ret i32 0
}

define i32 @custom_function() personality i32 (...)*
↪  @custom_personality {
  ; Function body with custom exception handling
  ret i32 0
}
```

3. **Interaction with Landing Pads**

   The personality function interacts with landing pads to determine how to handle exceptions. It provides the exception object and selector value to the landing pad, which uses this information to decide whether to catch the exception or propagate it further.

9.2.8 Example: Advanced Personality Function Usage

Here is an example that demonstrates advanced usage of personality functions, including custom personality functions and multiple exception handling mechanisms:

```
declare void @throw_cpp_exception()  ; Declare a function that throws a C++
 ↪  exception
declare void @throw_custom_exception()  ; Declare a function that throws a
 ↪  custom exception

define i32 @cpp_function() personality i32 (...)* @__gxx_personality_v0 {
entry:
  invoke void @throw_cpp_exception()
          to label %normal unwind label %landing_pad

normal:
  ret i32 0

landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* null  ; Catch all C++ exceptions
  %exception_ptr = extractvalue { i8*, i32 } %lp, 0  ; Extract the
   ↪  exception object
  %selector = extractvalue { i8*, i32 } %lp, 1  ; Extract the selector
   ↪  value
  call void @handle_cpp_exception(i8* %exception_ptr)  ; Handle the C++
   ↪  exception
  ret i32 1
}

define i32 @custom_function() personality i32 (...)* @custom_personality {
entry:
  invoke void @throw_custom_exception()
```

```
          to label %normal unwind label %landing_pad

normal:
  ret i32 0

landing_pad:
  %lp = landingpad { i8*, i32 }
          catch i8* null  ; Catch all custom exceptions
  %exception_ptr = extractvalue { i8*, i32 } %lp, 0  ; Extract the
  ↪  exception object
  %selector = extractvalue { i8*, i32 } %lp, 1  ; Extract the selector
  ↪  value
  call void @handle_custom_exception(i8* %exception_ptr)  ; Handle the
  ↪  custom exception
  ret i32 1
}

declare i32 @__gxx_personality_v0(...)  ; Personality function for C++
↪  exceptions
declare i32 @custom_personality(...)  ; Custom personality function

declare void @handle_cpp_exception(i8*)  ; Declare a function to handle C++
↪  exceptions
declare void @handle_custom_exception(i8*)  ; Declare a function to handle
↪  custom exceptions
```

In this example:

- The `@cpp_function` uses the `@__gxx_personality_v0` personality function for C++ exception handling.
- The `@custom_function` uses a custom personality function for custom exception handling.

- Each function has its own landing pad to handle exceptions.

9.2.9 Key Points

- Personality functions manage the mechanics of exception handling, including stack unwinding and invoking landing pads.
- The `@__gxx_personality_v0` personality function is commonly used for C++ exceptions.
- Custom personality functions can be used for other exception handling mechanisms.
- Personality functions interact with landing pads to handle exceptions.

9.2.10 Conclusion

Personality functions are a critical component of exception handling in LLVM IR, enabling developers to manage runtime errors and unexpected conditions gracefully. By understanding the syntax, usage, and key characteristics of personality functions, intermediate users can implement and analyze exception handling in LLVM IR effectively.

9.3 Stack Unwinding (Cleanup)

Exception handling in LLVM IR is a critical aspect of writing robust and fault-tolerant programs. One of the most important components of exception handling is **stack unwinding**, which ensures that resources are properly cleaned up and the program state is restored when an exception is thrown. This section delves into the details of stack unwinding, its mechanisms, and how it is implemented in LLVM IR.

9.3.1 Overview of Stack Unwinding

Stack unwinding is the process of dismantling the call stack when an exception is thrown. This involves:

1. **Destroying local objects**: Ensuring that destructors for local variables are called.
2. **Releasing resources**: Freeing memory, closing files, and releasing other acquired resources.
3. **Restoring the program state**: Returning control to an appropriate exception handler (catch block).

In LLVM IR, stack unwinding is tightly integrated with the exception handling framework, which relies on the **Itanium C++ ABI** for exception handling on many platforms. This ABI defines how exceptions are thrown, caught, and how the stack is unwound.

9.3.2 Key Concepts in Stack Unwinding

1. **Landing Pads**

   A **landing pad** is a block of code in LLVM IR that is executed when an exception is thrown. It is responsible for:

- Identifying the type of exception.
- Invoking the appropriate cleanup actions.
- Transferring control to the corresponding catch block.

Landing pads are generated using the `landingpad` instruction in LLVM IR. This instruction specifies the types of exceptions it can handle and the cleanup actions to perform.

2. **Personality Function**

   The **personality function** is a runtime function that assists in exception handling. It is specified using the `personality` attribute in LLVM IR. The personality function is responsible for:

   - Determining the type of exception.
   - Guiding the unwinding process.
   - Invoking the appropriate landing pad.

   On many platforms, the personality function is provided by the C++ runtime (e.g., `__gxx_personality_v0` for the Itanium C++ ABI).

3. **Cleanup Actions**

   Cleanup actions are operations performed during stack unwinding to release resources and destroy local objects. These actions are typically implemented using:

   - **Destructor calls**: For local objects with automatic storage duration.
   - **Resource release code**: For manually managed resources like memory or file handles.

   In LLVM IR, cleanup actions are often embedded within landing pads or invoked using the `invoke` instruction.

9.3.3 Stack Unwinding Process

The stack unwinding process in LLVM IR can be broken down into the following steps:

1. **Exception Thrown**: An exception is thrown using the `throw` instruction or an equivalent runtime function.
2. **Search for Handler**: The runtime searches the call stack for an appropriate exception handler.
3. **Unwind the Stack**: The stack is unwound, and cleanup actions are performed for each frame.
4. **Transfer Control**: Control is transferred to the landing pad of the matching exception handler.
5. **Execute Cleanup**: Cleanup actions are executed, and the exception is handled.

9.3.4 LLVM IR Instructions for Stack Unwinding

1. **`landingpad` Instruction**

   The `landingpad` instruction is used to define a landing pad block. It has the following syntax:

   ```
   <result> = landingpad <type> <clause>+ <filter>?
   ```

   - `<type>`: The type of the exception object.
   - `<clause>`: Specifies the types of exceptions the landing pad can handle (e.g., `catch` or `filter` clauses).
   - `<filter>`: Optional filter clause for exception filtering.

Example:

```
%lp = landingpad { i8*, i32 }
         catch i8* @ExceptionType
         filter [1 x i8*] [i8* null]
```

2. **`invoke` Instruction**

   The `invoke` instruction is used to call a function that may throw an exception. It has the following syntax:

```
<result> = invoke <return_type> <function>(<args>)
           to <normal_dest> unwind <unwind_dest>
```

   - `<normal_dest>`: The block to execute if the function returns normally.
   - `<unwind_dest>`: The landing pad block to execute if the function throws an exception.

   Example:

```
%result = invoke i32 @may_throw(i32 %arg)
           to label %normal unwind label %unwind
```

3. **`resume` Instruction**

   The `resume` instruction is used to continue unwinding the stack after a landing pad has performed its cleanup actions. It has the following syntax:

```
resume <type> <value>
```

- <value>: The exception object to resume unwinding with.

Example:

```
resume { i8*, i32 } %exception
```

9.3.5 Example of Stack Unwinding in LLVM IR

Consider the following C++ code:

```
void foo() {
    MyClass obj;
    throw std::exception();
}
```

The corresponding LLVM IR might look like this:

```
define void @foo() personality i8* bitcast (i32 (...)*
 ↪  @__gxx_personality_v0 to i8*) {
entry:
    %obj = alloca %MyClass, align 8
    call void @MyClass_ctor(%MyClass* %obj)
    invoke void @__cxa_throw(i8* null, i8* null, i8* null)
            to label %unreachable unwind label %lpad

lpad:
    %lp = landingpad { i8*, i32 }
            catch i8* null
```

```
    call void @MyClass_dtor(%MyClass* %obj)
    resume { i8*, i32 } %lp

unreachable:
    unreachable
}
```

In this example:

1. The `invoke` instruction is used to call `__cxa_throw`, which throws an exception.
2. The `landingpad` instruction defines the landing pad block (`%lpad`).
3. The destructor for `MyClass` is called in the landing pad.
4. The `resume` instruction continues unwinding the stack.

9.3.6 Advanced Topics

1. **Exception Handling in Multithreaded Environments**

   In multithreaded programs, stack unwinding must handle exceptions across multiple threads. LLVM IR relies on the runtime to ensure thread-safe exception handling.

2. **Custom Personality Functions**

   Advanced users can define custom personality functions to implement specialized exception handling logic. This is useful for non-C++ languages or unique runtime environments.

3. **Optimizations and Debugging**

   LLVM provides tools like `llvm-objdump` and `llvm-dwarfdump` to analyze exception handling metadata. Optimizations like inlining and tail call elimination must preserve exception handling semantics.

9.3.7 Conclusion

Stack unwinding is a fundamental part of exception handling in LLVM IR. By understanding the role of landing pads, personality functions, and cleanup actions, developers can write efficient and reliable code that gracefully handles errors. The integration of stack unwinding with the Itanium C++ ABI ensures compatibility with existing C++ runtimes, while LLVM's flexibility allows for custom implementations in specialized environments.

This section has provided a comprehensive overview of stack unwinding in LLVM IR, equipping intermediate users with the knowledge to implement and debug exception handling in their projects.

Chapter 10

Type Conversions & Casting

10.1 Integer to Float (`sitofp`, `uitofp`)

Type conversion is a fundamental operation in programming, allowing developers to transform data from one type to another. In LLVM IR, type conversions are explicitly represented using specific instructions. This section focuses on **integer-to-floating-point conversions**, which are essential for operations involving mixed types, such as arithmetic computations or data transformations. The two primary instructions for this purpose are `sitofp` (signed integer to floating-point) and `uitofp` (unsigned integer to floating-point).

10.1.1 Overview of Integer-to-Float Conversions

Integer-to-floating-point conversions are used to transform integer values into their corresponding floating-point representations. This is particularly useful in scenarios where:

1. **Arithmetic operations** involve both integer and floating-point operands.
2. **Data processing** requires converting integer data into floating-point format for further

computation.

3. **Interfacing with APIs** that expect floating-point inputs but receive integer data.

In LLVM IR, these conversions are performed using the `sitofp` and `uitofp` instructions, which handle signed and unsigned integers, respectively.

10.1.2 Key Concepts

1. **Signed vs. Unsigned Integers**

   - **Signed Integers**: Represent both positive and negative values using a sign bit.
   - **Unsigned Integers**: Represent only non-negative values, with no sign bit.

   The distinction between signed and unsigned integers is crucial because it affects how the conversion to floating-point is performed.

2. **Floating-Point Representation**

   Floating-point numbers in LLVM IR are typically represented using the `float` (32-bit) or `double` (64-bit) types. These types adhere to the IEEE 754 standard, which defines their format and behavior.

3. **Precision and Rounding**

   When converting integers to floating-point values, precision and rounding must be considered:

   - **Precision**: Floating-point types have limited precision, which may result in loss of accuracy for large integers.
   - **Rounding**: The conversion process may involve rounding if the integer value cannot be represented exactly in the target floating-point format.

10.1.3 LLVM IR Instructions

1. **`sitofp` Instruction**

   The `sitofp` instruction converts a **signed integer** to a floating-point value. Its syntax is as follows:

   ```
   <result> = sitofp <source_type> <value> to <destination_type>
   ```

   - `<source_type>`: The type of the integer value (e.g., `i32`, `i64`).
   - `<value>`: The integer value to convert.
   - `<destination_type>`: The target floating-point type (e.g., `float`, `double`).

   Example:

   ```
   %result = sitofp i32 %int_value to float
   ```

   This converts the signed 32-bit integer `%int_value` to a 32-bit floating-point value.

2. **`uitofp` Instruction**

   The `uitofp` instruction converts an **unsigned integer** to a floating-point value. Its syntax is as follows:

   ```
   <result> = uitofp <source_type> <value> to <destination_type>
   ```

   - `<source_type>`: The type of the integer value (e.g., `i32`, `i64`).
   - `<value>`: The integer value to convert.

- `<destination_type>`: The target floating-point type (e.g., `float`, `double`).

Example:

```
%result = uitofp i32 %uint_value to double
```

This converts the unsigned 32-bit integer `%uint_value` to a 64-bit floating-point value.

10.1.4 Detailed Behavior

1. **Conversion Process**

   The conversion process involves the following steps:

   (a) **Bit Interpretation**: The integer value is interpreted based on its signedness (signed or unsigned).

   (b) **Mapping to Floating-Point**: The integer value is mapped to the closest representable floating-point value.

   (c) **Rounding**: If the integer value cannot be represented exactly, rounding is performed according to the default rounding mode.

2. **Handling Large Integers**

   For large integers that exceed the precision of the target floating-point type, the conversion may result in a loss of precision. For example:

   - Converting the 64-bit integer `9223372036854775807` (maximum value for `i64`) to a `float` may result in a rounded value.

- The `double` type, with its higher precision, can represent larger integers more accurately.

3. **Special Cases**

   - **Zero**: Both `sitofp` and `uitofp` convert zero to `0.0`.
   - **Negative Values**: The `sitofp` instruction correctly handles negative integers, converting them to negative floating-point values.
   - **Overflow**: If the integer value is too large to be represented in the target floating-point type, the result is infinity.

10.1.5 Examples

1. **Simple Conversion**

   Consider the following C code:

```c
int x = 42;
float y = (float)x;
```

   The corresponding LLVM IR is:

```
%x = alloca i32, align 4
store i32 42, i32* %x, align 4
%x_value = load i32, i32* %x, align 4
%y = sitofp i32 %x_value to float
```

2. **Unsigned Conversion**

   Consider the following C code:

```
unsigned int x = 1000;
double y = (double)x;
```

The corresponding LLVM IR is:

```
%x = alloca i32, align 4
store i32 1000, i32* %x, align 4
%x_value = load i32, i32* %x, align 4
%y = uitofp i32 %x_value to double
```

3. **Handling Negative Values**

   Consider the following C code:

```
int x = -42;
float y = (float)x;
```

The corresponding LLVM IR is:

```
%x = alloca i32, align 4
store i32 -42, i32* %x, align 4
%x_value = load i32, i32* %x, align 4
%y = sitofp i32 %x_value to float
```

10.1.6 Advanced Topics

1. **Rounding Modes**

By default, LLVM uses the **round-to-nearest** rounding mode for integer-to-float conversions. However, advanced users can specify alternative rounding modes (e.g., round-toward-zero) using additional LLVM intrinsics or metadata.

2. **Vectorized Conversions**

   LLVM supports vectorized integer-to-float conversions, allowing multiple values to be converted in parallel. For example:

```
%result = sitofp <4 x i32> %int_vector to <4 x float>
```

   This converts a vector of four 32-bit signed integers to a vector of four 32-bit floating-point values.

3. **Performance Considerations**

   - **Hardware Support**: Modern CPUs provide efficient hardware instructions for integer-to-float conversions, making these operations relatively fast.
   - **Precision Loss**: Developers should be aware of potential precision loss when converting large integers to floating-point values.

10.1.7 Conclusion

The `sitofp` and `uitofp` instructions in LLVM IR provide a robust mechanism for converting integer values to floating-point representations. By understanding their behavior, developers can ensure accurate and efficient type conversions in their programs. Whether working with signed or unsigned integers, these instructions enable seamless integration of integer and floating-point data, making them indispensable for intermediate and advanced LLVM IR programming.

This section has provided a comprehensive exploration of integer-to-float conversions, equipping users with the knowledge to implement and optimize these operations in their LLVM-based projects.

10.2 Float to Integer (`fptosi`, `fptoui`)

Floating-point to integer conversion is a critical operation in many programs, especially when dealing with data that requires truncation or rounding to fit into integer types. In LLVM IR, these conversions are explicitly represented using the `fptosi` (floating-point to signed integer) and `fptoui` (floating-point to unsigned integer) instructions. This section provides a detailed exploration of these instructions, their behavior, and their applications in intermediate-level LLVM IR programming.

10.2.1 Overview of Float-to-Integer Conversions

Floating-point to integer conversions are used to transform floating-point values into their corresponding integer representations. This is particularly useful in scenarios such as:

1. **Data truncation**: Converting floating-point values to integers for storage or further processing.
2. **Mathematical operations**: Performing integer-specific operations on floating-point data.
3. **API compatibility**: Interfacing with functions or systems that expect integer inputs but receive floating-point data.

In LLVM IR, these conversions are performed using the `fptosi` and `fptoui` instructions, which handle signed and unsigned integers, respectively.

10.2.2 Key Concepts

1. **Floating-Point Representation**

Floating-point numbers in LLVM IR are typically represented using the `float` (32-bit) or `double` (64-bit) types, adhering to the IEEE 754 standard. These types can represent a wide range of values, including fractions and very large or small numbers.

2. **Integer Representation**

   - **Signed Integers**: Represent both positive and negative values using a sign bit.
   - **Unsigned Integers**: Represent only non-negative values, with no sign bit.

   The distinction between signed and unsigned integers is crucial because it affects how floating-point values are truncated or rounded during conversion.

3. **Truncation and Rounding**

   When converting floating-point values to integers, the fractional part is discarded, effectively truncating the value. However, the behavior for out-of-range values or special cases (e.g., NaN, infinity) must be carefully considered.

10.2.3 LLVM IR Instructions

1. **`fptosi` Instruction**

   The `fptosi` instruction converts a **floating-point value** to a **signed integer**. Its syntax is as follows:

```
<result> = fptosi <source_type> <value> to <destination_type>
```

   - `<source_type>`: The type of the floating-point value (e.g., `float`, `double`).
   - `<value>`: The floating-point value to convert.
   - `<destination_type>`: The target integer type (e.g., `i32`, `i64`).

Example:

```
%result = fptosi float %float_value to i32
```

This converts the 32-bit floating-point value `%float_value` to a 32-bit signed integer.

2. **`fptoui` Instruction**

   The `fptoui` instruction converts a **floating-point value** to an **unsigned integer**. Its syntax is as follows:

   ```
   <result> = fptoui <source_type> <value> to <destination_type>
   ```

   - `<source_type>`: The type of the floating-point value (e.g., `float`, `double`).
   - `<value>`: The floating-point value to convert.
   - `<destination_type>`: The target integer type (e.g., `i32`, `i64`).

   Example:

   ```
   %result = fptoui double %double_value to i64
   ```

   This converts the 64-bit floating-point value `%double_value` to a 64-bit unsigned integer.

10.2.4 Detailed Behavior

1. **Conversion Process**

   The conversion process involves the following steps:

(a) **Truncation**: The fractional part of the floating-point value is discarded.

(b) **Range Checking**: The value is checked to ensure it falls within the representable range of the target integer type.

(c) **Special Cases**: Handling of special floating-point values (e.g., NaN, infinity) is performed.

2. **Handling Out-of-Range Values**

   - If the floating-point value exceeds the maximum representable value of the target integer type, the result is **undefined behavior**.
   - If the floating-point value is negative and the target type is unsigned (`fptoui`), the result is **undefined behavior**.

3. **Special Cases**

   - **NaN (Not a Number)**: Converting NaN to an integer results in **undefined behavior**.
   - **Infinity**: Converting infinity to an integer results in **undefined behavior**.
   - **Zero**: Both `fptosi` and `fptoui` convert zero to `0`.

10.2.5 Examples

1. **Simple Conversion**

   Consider the following C code:

```c
float x = 42.7f;
int y = (int)x;
```

The corresponding LLVM IR is:

```
%x = alloca float, align 4
store float 42.7, float* %x, align 4
%x_value = load float, float* %x, align 4
%y = fptosi float %x_value to i32
```

2. **Unsigned Conversion**

   Consider the following C code:

```
double x = 1000.9;
unsigned int y = (unsigned int)x;
```

   The corresponding LLVM IR is:

```
%x = alloca double, align 8
store double 1000.9, double* %x, align 8
%x_value = load double, double* %x, align 8
%y = fptoui double %x_value to i32
```

3. **Handling Negative Values**

   Consider the following C code:

```
float x = -42.7f;
int y = (int)x;
```

   The corresponding LLVM IR is:

```
%x = alloca float, align 4
store float -42.7, float* %x, align 4
%x_value = load float, float* %x, align 4
%y = fptosi float %x_value to i32
```

10.2.6 Advanced Topics

1. **Undefined Behavior**

   Developers must be cautious when converting floating-point values to integers, as out-of-range values or special cases (e.g., NaN, infinity) result in undefined behavior. To avoid this, range checks should be performed before conversion.

2. **Vectorized Conversions**

   LLVM supports vectorized floating-point-to-integer conversions, allowing multiple values to be converted in parallel. For example:

   ```
   %result = fptosi <4 x float> %float_vector to <4 x i32>
   ```

   This converts a vector of four 32-bit floating-point values to a vector of four 32-bit signed integers.

3. **Performance Considerations**

   - **Hardware Support**: Modern CPUs provide efficient hardware instructions for floating-point-to-integer conversions, making these operations relatively fast.
   - **Range Checks**: Explicit range checks can add overhead but are necessary to avoid undefined behavior.

10.2.7 Conclusion

The `fptosi` and `fptoui` instructions in LLVM IR provide a robust mechanism for converting floating-point values to integer representations. By understanding their behavior and limitations, developers can ensure accurate and efficient type conversions in their programs. Whether working with signed or unsigned integers, these instructions enable seamless integration of floating-point and integer data, making them indispensable for intermediate and advanced LLVM IR programming.

This section has provided a comprehensive exploration of floating-point-to-integer conversions, equipping users with the knowledge to implement and optimize these operations in their LLVM-based projects.

10.3 Pointer Conversions (`bitcast`, `inttoptr`, `ptrtoint`)

Pointer conversions are a powerful feature in LLVM IR, enabling developers to manipulate memory addresses and reinterpret data types. These conversions are essential for low-level programming tasks, such as interfacing with hardware, implementing custom memory allocators, or working with type-agnostic data structures. This section explores the three primary pointer conversion instructions in LLVM IR: `bitcast`, `inttoptr`, and `ptrtoint`. Each instruction serves a distinct purpose and has specific use cases, which will be discussed in detail.

10.3.1 Overview of Pointer Conversions

Pointer conversions in LLVM IR allow developers to:

1. **Reinterpret data types**: Change the type of a pointer without altering the underlying data.
2. **Convert between pointers and integers**: Treat memory addresses as integer values and vice versa.
3. **Enable low-level memory manipulation**: Perform operations that require direct control over memory addresses.

These conversions are explicitly represented in LLVM IR using the `bitcast`, `inttoptr`, and `ptrtoint` instructions. Each instruction has specific semantics and constraints, which must be understood to avoid undefined behavior.

10.3.2 Key Concepts

1. **Pointer Types in LLVM IR**

In LLVM IR, pointers are represented using the `ptr` type (or explicit types like `i8*`, `i32*`, etc.). Pointers can point to any data type, including integers, floats, structs, and arrays.

2. **Type Safety**

   LLVM IR is strongly typed, meaning that operations on pointers must respect their types. Pointer conversions allow developers to bypass some of these restrictions, but they must be used carefully to avoid undefined behavior.

3. **Address Space**

   LLVM IR supports multiple address spaces, which are used to represent different memory regions (e.g., global, local, or device memory). Pointer conversions must respect address space constraints.

10.3.3 LLVM IR Instructions

1. **`bitcast` Instruction**

   The `bitcast` instruction reinterprets a value as a different type without changing its bits. It is commonly used for pointer-to-pointer conversions. Its syntax is as follows:

```
<result> = bitcast <source_type> <value> to <destination_type>
```

   - `<source_type>`: The type of the input value (e.g., `i32*`, `float*`).
   - `<value>`: The value to reinterpret.
   - `<destination_type>`: The target type (e.g., `i8*`, `double*`).

   Example:

```
%ptr = bitcast i32* %int_ptr to i8*
```

This reinterprets the `i32*` pointer `%int_ptr` as an `i8*` pointer.

2. **`inttoptr` Instruction**

   The `inttoptr` instruction converts an integer value to a pointer. Its syntax is as follows:

   ```
   <result> = inttoptr <source_type> <value> to <destination_type>
   ```

   - `<source_type>`: The type of the integer value (e.g., `i64`).
   - `<value>`: The integer value to convert.
   - `<destination_type>`: The target pointer type (e.g., `i32*`).

   Example:

   ```
   %ptr = inttoptr i64 %address to i32*
   ```

   This converts the 64-bit integer `%address` to a pointer to a 32-bit integer.

3. **`ptrtoint` Instruction**

   The `ptrtoint` instruction converts a pointer to an integer value. Its syntax is as follows:

   ```
   <result> = ptrtoint <source_type> <value> to <destination_type>
   ```

   - `<source_type>`: The type of the pointer (e.g., `i32*`).

- `<value>`: The pointer value to convert.
- `<destination_type>`: The target integer type (e.g., `i64`).

Example:

```
%address = ptrtoint i32* %ptr to i64
```

This converts the `i32*` pointer `%ptr` to a 64-bit integer.

10.3.4 Detailed Behavior

1. **`bitcast` Semantics**
   - The `bitcast` instruction does not modify the bits of the input value; it only changes the type.
   - It is typically used for pointer-to-pointer conversions or to reinterpret data types (e.g., casting a `float*` to an `i32*`).
   - The source and destination types must have the same size.
2. **`inttoptr` Semantics**
   - The `inttoptr` instruction converts an integer to a pointer, treating the integer as a memory address.
   - The resulting pointer must point to a valid memory location; otherwise, dereferencing it may result in undefined behavior.
   - This instruction is often used in low-level programming, such as implementing memory allocators or interfacing with hardware.
3. **`ptrtoint` Semantics**

- The `ptrtoint` instruction converts a pointer to an integer, representing the memory address as an integer value.
- The resulting integer can be used for arithmetic operations or stored in data structures.
- Care must be taken to ensure that the integer value is not used in a way that violates memory safety.

10.3.5 Examples

1. **Using `bitcast` for Type Reinterpretation**

   Consider the following C code:

   ```c
   int x = 42;
   float* y = (float*)&x;
   ```

   The corresponding LLVM IR is:

   ```
   %x = alloca i32, align 4
   store i32 42, i32* %x, align 4
   %y = bitcast i32* %x to float*
   ```

2. **Using `inttoptr` for Address Manipulation**

   Consider the following C code:

   ```c
   uintptr_t address = 0x1000;
   int* ptr = (int*)address;
   ```

   The corresponding LLVM IR is:

```
%address = alloca i64, align 8
store i64 4096, i64* %address, align 8  ; 0x1000 in decimal
%ptr = inttoptr i64 %address to i32*
```

3. **Using `ptrtoint` for Address Arithmetic**

   Consider the following C code:

```
int* ptr = ...;
uintptr_t address = (uintptr_t)ptr;
```

   The corresponding LLVM IR is:

```
%ptr = alloca i32*, align 8
%address = ptrtoint i32* %ptr to i64
```

10.3.6 Advanced Topics

1. **Address Space Considerations**

   When working with multiple address spaces, pointer conversions must respect the address space of the source and destination types. For example:

```
%ptr = bitcast i32 addrspace(1)* %global_ptr to i8 addrspace(1)*
```

2. **Undefined Behavior**

   Pointer conversions can lead to undefined behavior if not used carefully. For example:

   - Using `inttoptr` with an invalid address.

- Dereferencing a pointer obtained from `bitcast` if the types are incompatible.

3. **Vectorized Conversions**

   LLVM supports vectorized pointer conversions, allowing multiple pointers or integers to be converted in parallel. For example:

```
%ptr_vector = bitcast <4 x i32*> %int_ptr_vector to <4 x i8*>
```

10.3.7 Conclusion

Pointer conversions are a powerful tool in LLVM IR, enabling developers to manipulate memory addresses and reinterpret data types. The `bitcast`, `inttoptr`, and `ptrtoint` instructions provide the flexibility needed for low-level programming tasks, but they must be used with care to avoid undefined behavior. By understanding the semantics and constraints of these instructions, intermediate users can leverage them effectively in their LLVM-based projects.

This section has provided a comprehensive exploration of pointer conversions, equipping users with the knowledge to implement and optimize these operations in their programs.

Chapter 11

Atomic & Concurrency Instructions

11.1 Atomic Load & Store (`atomicrmw add i32* %ptr, i32 1 seq_cst`)

Atomic operations are fundamental to concurrent programming, ensuring that multiple threads can safely manipulate shared data without causing race conditions. In LLVM IR, atomic operations are explicitly represented using specialized instructions, such as `atomicrmw` and `cmpxchg`. This section focuses on **atomic load and store operations**, particularly the `atomicrmw` instruction, which performs atomic read-modify-write operations. We will explore its syntax, semantics, and use cases in detail, with a focus on the `add` operation and the `seq_cst` (sequentially consistent) memory ordering.

11.1.1 Overview of Atomic Operations

Atomic operations ensure that certain memory operations are performed indivisibly, even in the presence of multiple threads. This is crucial for implementing synchronization primitives,

such as locks, barriers, and atomic counters. In LLVM IR, atomic operations are supported through the following key instructions:

1. **`atomicrmw`**: Performs an atomic read-modify-write operation.
2. **`cmpxchg`**: Performs an atomic compare-and-swap operation.
3. **`load atomic` and `store atomic`**: Perform atomic loads and stores.

This section focuses on the `atomicrmw` instruction, which is commonly used for operations like atomic addition, subtraction, and bitwise operations.

11.1.2 Key Concepts

1. **Atomicity**

   Atomicity ensures that an operation is executed as a single, indivisible unit. For example, an atomic addition operation reads the value at a memory location, adds a specified value to it, and writes the result back to memory without any intermediate steps being interrupted.

2. **Memory Ordering**

   Memory ordering defines the visibility of memory operations across threads. LLVM IR supports several memory ordering constraints, including:

   - **`unordered`**: No ordering guarantees.
   - **`monotonic`**: Ensures that operations on the same memory location are ordered.
   - **`acquire`**: Ensures that subsequent operations are not reordered before the atomic operation.
   - **`release`**: Ensures that previous operations are not reordered after the atomic operation.

- **`acq_rel`**: Combines `acquire` and `release` semantics.
- **`seq_cst`**: Ensures sequential consistency, the strongest memory ordering.

3. **Read-Modify-Write Operations**

   A read-modify-write operation performs three steps atomically:

   (a) **Read**: Load the value from memory.

   (b) **Modify**: Perform an operation (e.g., addition) on the value.

   (c) **Write**: Store the result back to memory.

11.1.3 LLVM IR Instruction: `atomicrmw`

The `atomicrmw` instruction performs an atomic read-modify-write operation. Its syntax is as follows:

```
<result> = atomicrmw <operation> <type>* <pointer>, <type> <value>
 ↪  <ordering>
```

- `<operation>`: The operation to perform (e.g., `add`, `sub`, `and`, `or`, `xor`, `xchg`).
- `<type>`: The type of the value (e.g., `i32`, `i64`).
- `<pointer>`: The pointer to the memory location.
- `<value>`: The value to use in the operation.
- `<ordering>`: The memory ordering constraint (e.g., `seq_cst`, `acq_rel`, `monotonic`).

Example:

```
%result = atomicrmw add i32* %ptr, i32 1 seq_cst
```

This performs an atomic addition of `1` to the value at the memory location pointed to by `%ptr`, using sequentially consistent memory ordering.

11.1.4 Detailed Behavior

1. **Supported Operations**

   The `atomicrmw` instruction supports the following operations:

   - **`add`**: Atomic addition.
   - **`sub`**: Atomic subtraction.
   - **`and`**: Atomic bitwise AND.
   - **`or`**: Atomic bitwise OR.
   - **`xor`**: Atomic bitwise XOR.
   - **`xchg`**: Atomic exchange (swap).

2. **Memory Ordering Semantics**

   The memory ordering constraint determines how the atomic operation interacts with other memory operations. For example:

   - **`seq_cst`**: Ensures that all threads see all operations in the same order.
   - **`acq_rel`**: Ensures that the operation acts as both an acquire and a release barrier.
   - **`monotonic`**: Provides minimal ordering guarantees.

3. **Return Value**

   The `atomicrmw` instruction returns the original value at the memory location before the operation was performed. This is useful for implementing algorithms that require knowledge of the previous value.

11.1.5 Examples

1. **Atomic Addition**

   Consider the following C code:

```c
#include <stdatomic.h>
atomic_int counter = ATOMIC_VAR_INIT(0);
atomic_fetch_add(&counter, 1);
```

   The corresponding LLVM IR is:

```
%counter = alloca i32, align 4
store i32 0, i32* %counter, align 4
%result = atomicrmw add i32* %counter, i32 1 seq_cst
```

2. **Atomic Subtraction**

   Consider the following C code:

```c
#include <stdatomic.h>
atomic_int counter = ATOMIC_VAR_INIT(10);
atomic_fetch_sub(&counter, 1);
```

   The corresponding LLVM IR is:

```
%counter = alloca i32, align 4
store i32 10, i32* %counter, align 4
%result = atomicrmw sub i32* %counter, i32 1 seq_cst
```

3. **Atomic Bitwise AND**

   Consider the following C code:

```
#include <stdatomic.h>
atomic_int flags = ATOMIC_VAR_INIT(0xFF);
atomic_fetch_and(&flags, 0xF0);
```

   The corresponding LLVM IR is:

```
%flags = alloca i32, align 4
store i32 255, i32* %flags, align 4  ; 0xFF in decimal
%result = atomicrmw and i32* %flags, i32 240 seq_cst  ; 0xF0 in
↪  decimal
```

11.1.6 Advanced Topics

1. **Memory Ordering Trade-offs**

   - **`seq_cst`**: Provides the strongest guarantees but may incur higher overhead.
   - **`acq_rel`**: Balances performance and ordering guarantees.
   - **`monotonic`**: Offers minimal guarantees but is the fastest.

2. **Atomic Operations on Non-Integral Types**

While `atomicrmw` primarily supports integer types, atomic operations on non-integral types (e.g., pointers) can be achieved using `cmpxchg` or `load atomic`/`store atomic`.

3. **Vectorized Atomic Operations**

   LLVM does not natively support vectorized atomic operations. However, developers can emulate them using loops and scalar atomic operations.

11.1.7 Conclusion

The `atomicrmw` instruction is a powerful tool for implementing atomic read-modify-write operations in LLVM IR. By understanding its syntax, semantics, and memory ordering constraints, intermediate users can effectively write concurrent programs that are both correct and efficient. Whether implementing atomic counters, flags, or custom synchronization primitives, the `atomicrmw` instruction provides the necessary functionality to ensure thread-safe operations.

This section has provided a comprehensive exploration of atomic load and store operations, equipping users with the knowledge to leverage these instructions in their LLVM-based projects.

11.2 Thread Synchronization (`fence seq_cst`)

Thread synchronization is a critical aspect of concurrent programming, ensuring that multiple threads coordinate their operations correctly and avoid race conditions. In LLVM IR, thread synchronization is achieved using **memory fences** (also known as memory barriers), which enforce ordering constraints on memory operations. This section focuses on the `fence` instruction, particularly with the `seq_cst` (sequentially consistent) memory ordering, and explores its role in thread synchronization.

11.2.1 Overview of Thread Synchronization

Thread synchronization ensures that memory operations performed by one thread are visible to other threads in a predictable and controlled manner. Without proper synchronization, concurrent programs can suffer from race conditions, data corruption, and undefined behavior. In LLVM IR, thread synchronization is primarily achieved using:

1. **Memory Fences (`fence`)**: Enforce ordering constraints on memory operations.
2. **Atomic Operations (`atomicrmw`, `cmpxchg`)**: Provide atomicity and ordering guarantees.
3. **Synchronization Primitives**: Built using the above instructions (e.g., locks, barriers).

This section focuses on the `fence` instruction, which is used to establish memory ordering constraints between threads.

11.2.2 Key Concepts

1. **Memory Fences**

A memory fence (or memory barrier) is an instruction that enforces ordering constraints on memory operations. It ensures that certain memory operations are not reordered across the fence, providing guarantees about the visibility of memory operations to other threads.

2. **Memory Ordering**

   Memory ordering defines the visibility of memory operations across threads. LLVM IR supports several memory ordering constraints, including:

   - **`unordered`**: No ordering guarantees.
   - **`monotonic`**: Ensures that operations on the same memory location are ordered.
   - **`acquire`**: Ensures that subsequent operations are not reordered before the fence.
   - **`release`**: Ensures that previous operations are not reordered after the fence.
   - **`acq_rel`**: Combines `acquire` and `release` semantics.
   - **`seq_cst`**: Ensures sequential consistency, the strongest memory ordering.

3. **Sequential Consistency (`seq_cst`)**

   Sequential consistency is the strongest memory ordering model. It ensures that all threads see all memory operations in the same order. This simplifies reasoning about concurrent programs but may incur higher overhead.

11.2.3 LLVM IR Instruction: `fence`

The `fence` instruction enforces memory ordering constraints. Its syntax is as follows:

```
fence <ordering>
```

- `<ordering>`: The memory ordering constraint (e.g., `seq_cst`, `acq_rel`, `release`, `acquire`).

Example:

```
fence seq_cst
```

This enforces a sequentially consistent memory fence.

11.2.4 Detailed Behavior

1. **Memory Ordering Semantics**

   The `fence` instruction ensures that memory operations before and after the fence are not reordered in a way that violates the specified memory ordering. For example:

   - **`seq_cst`**: Ensures that all memory operations before the fence are visible to all threads before any memory operations after the fence.
   - **`acq_rel`**: Combines `acquire` and `release` semantics, ensuring that previous operations are not reordered after the fence and subsequent operations are not reordered before the fence.
   - **`release`**: Ensures that previous operations are not reordered after the fence.
   - **`acquire`**: Ensures that subsequent operations are not reordered before the fence.

2. **Use Cases**

   Memory fences are used in various synchronization scenarios, such as:

   - **Implementing Locks**: Ensuring that critical sections are properly synchronized.
   - **Building Barriers**: Coordinating the execution of multiple threads.
   - **Enforcing Ordering**: Guaranteeing that memory operations are visible in a specific order.

11.2.5 Examples

1. **Sequential Consistency Fence**

   Consider the following C code:

```c
#include <stdatomic.h>
atomic_int flag = ATOMIC_VAR_INIT(0);
atomic_store_explicit(&flag, 1, memory_order_seq_cst);
```

   The corresponding LLVM IR is:

```
%flag = alloca i32, align 4
store i32 0, i32* %flag, align 4
store atomic i32 1, i32* %flag seq_cst, align 4
fence seq_cst
```

2. **Acquire-Release Fence**

   Consider the following C code:

```c
#include <stdatomic.h>
atomic_int data = ATOMIC_VAR_INIT(0);
atomic_int flag = ATOMIC_VAR_INIT(0);

// Thread 1
atomic_store_explicit(&data, 42, memory_order_relaxed);
atomic_store_explicit(&flag, 1, memory_order_release);

// Thread 2
while (atomic_load_explicit(&flag, memory_order_acquire) == 0);
int value = atomic_load_explicit(&data, memory_order_relaxed);
```

The corresponding LLVM IR for Thread 1 is:

```
%data = alloca i32, align 4
%flag = alloca i32, align 4
store i32 0, i32* %data, align 4
store i32 0, i32* %flag, align 4
store atomic i32 42, i32* %data monotonic, align 4
fence release
store atomic i32 1, i32* %flag monotonic, align 4
```

The corresponding LLVM IR for Thread 2 is:

```
%flag = alloca i32, align 4
%data = alloca i32, align 4
br label %loop

loop:
  %flag_value = load atomic i32, i32* %flag acquire, align 4
  %cmp = icmp eq i32 %flag_value, 0
  br i1 %cmp, label %loop, label %exit

exit:
  %value = load atomic i32, i32* %data monotonic, align 4
```

11.2.6 Advanced Topics

1. **Performance Considerations**

   - **`seq_cst`**: Provides the strongest guarantees but may incur higher overhead due to stricter ordering constraints.

- **`acq_rel`**: Balances performance and ordering guarantees, making it suitable for many synchronization scenarios.
- **`release` and `acquire`**: Provide weaker guarantees but are more efficient.

2. **Fences vs. Atomic Operations**

   - **Fences**: Enforce ordering constraints on all memory operations.
   - **Atomic Operations**: Provide atomicity and ordering guarantees for specific memory locations.

3. **Hardware-Specific Behavior**

   The behavior of memory fences may vary across hardware architectures. For example, x86 provides strong memory ordering guarantees, while ARM requires explicit fences for weaker memory models.

11.2.7 Conclusion

The `fence` instruction is a powerful tool for enforcing memory ordering constraints in LLVM IR. By understanding its semantics and use cases, intermediate users can effectively implement thread synchronization in their concurrent programs. Whether building locks, barriers, or custom synchronization primitives, the `fence` instruction provides the necessary functionality to ensure correct and efficient concurrent execution.
This section has provided a comprehensive exploration of thread synchronization using memory fences, equipping users with the knowledge to leverage these instructions in their LLVM-based projects.

11.3 Compare and Swap (`cmpxchg`)

The **Compare and Swap (CAS)** operation is a fundamental building block for concurrent programming, enabling lock-free and wait-free algorithms. In LLVM IR, the `cmpxchg` instruction provides atomic compare-and-swap functionality, allowing developers to implement sophisticated synchronization primitives and data structures. This section delves into the `cmpxchg` instruction, its syntax, semantics, and use cases, with a focus on its role in concurrent programming.

11.3.1 Overview of Compare and Swap

The Compare and Swap operation is an atomic instruction that performs the following steps indivisibly:

1. **Compare**: Check if the value at a memory location matches an expected value.
2. **Swap**: If the comparison succeeds, update the memory location with a new value.

This operation is widely used in concurrent programming for implementing:

- **Lock-free data structures**: Such as stacks, queues, and linked lists.
- **Synchronization primitives**: Such as spinlocks and semaphores.
- **Atomic counters and flags**: For managing shared state.

In LLVM IR, the `cmpxchg` instruction provides this functionality, ensuring atomicity and memory ordering guarantees.

11.3.2 Key Concepts

1. **Atomicity**

   The `cmpxchg` instruction ensures that the compare-and-swap operation is performed atomically, meaning no other thread can observe or modify the memory location during the operation.

2. **Memory Ordering**

   The `cmpxchg` instruction supports memory ordering constraints, which determine how the operation interacts with other memory operations. Supported memory orderings include:

   - **`monotonic`**: Minimal ordering guarantees.
   - **`acquire`**: Ensures that subsequent operations are not reordered before the `cmpxchg`.
   - **`release`**: Ensures that previous operations are not reordered after the `cmpxchg`.
   - **`acq_rel`**: Combines `acquire` and `release` semantics.
   - **`seq_cst`**: Ensures sequential consistency, the strongest memory ordering.

3. **Return Value**

   The `cmpxchg` instruction returns a structure containing two values:

   (a) **The original value**: The value at the memory location before the operation.

   (b) **A success flag**: Indicates whether the comparison succeeded and the swap was performed.

11.3.3 LLVM IR Instruction: `cmpxchg`

The `cmpxchg` instruction performs an atomic compare-and-swap operation. Its syntax is as follows:

```
<result> = cmpxchg <type>* <pointer>, <type> <expected>, <type> <new>
↪  <success_ordering> <failure_ordering>
```

- `<type>`: The type of the value (e.g., `i32`, `i64`).
- `<pointer>`: The pointer to the memory location.
- `<expected>`: The expected value at the memory location.
- `<new>`: The new value to store if the comparison succeeds.
- `<success_ordering>`: The memory ordering constraint if the comparison succeeds.
- `<failure_ordering>`: The memory ordering constraint if the comparison fails.

Example:

```
%result = cmpxchg i32* %ptr, i32 42, i32 100 seq_cst seq_cst
```

This performs an atomic compare-and-swap operation on the `i32` value at `%ptr`. If the value is `42`, it is replaced with `100`.

11.3.4 Detailed Behavior

1. **Operation Steps**

   The `cmpxchg` instruction performs the following steps atomically:

(a) **Load**: Read the value at the memory location.

(b) **Compare**: Check if the loaded value matches the expected value.

(c) **Swap**: If the comparison succeeds, store the new value at the memory location.

2. **Memory Ordering Semantics**

   - **Success Ordering**: Applies if the comparison succeeds and the swap is performed.
   - **Failure Ordering**: Applies if the comparison fails and no swap is performed.

   For example, using `seq_cst` for both success and failure ensures the strongest memory ordering guarantees.

3. **Return Value Structure**

   The `cmpxchg` instruction returns a structure with two fields:

   (a) **Original Value**: The value at the memory location before the operation.

   (b) **Success Flag**: A boolean indicating whether the swap was performed.

11.3.5 Examples

1. **Simple Compare and Swap**

   Consider the following C code:

```
#include <stdatomic.h>
atomic_int value = ATOMIC_VAR_INIT(42);
atomic_compare_exchange_strong(&value, &expected, 100);
```

   The corresponding LLVM IR is:

```
%value = alloca i32, align 4
store i32 42, i32* %value, align 4
%expected = alloca i32, align 4
store i32 42, i32* %expected, align 4
%result = cmpxchg i32* %value, i32 42, i32 100 seq_cst seq_cst
```

2. **Loop Until Success**

   Consider the following C code, which repeatedly attempts to update a value using CAS:

```
#include <stdatomic.h>
atomic_int value = ATOMIC_VAR_INIT(0);
int expected = 0;
while (!atomic_compare_exchange_weak(&value, &expected, expected +
↪  1));
```

   The corresponding LLVM IR is:

```
%value = alloca i32, align 4
store i32 0, i32* %value, align 4
%expected = alloca i32, align 4
store i32 0, i32* %expected, align 4
br label %loop

loop:
  %expected_value = load i32, i32* %expected, align 4
  %new_value = add i32 %expected_value, 1
  %result = cmpxchg i32* %value, i32 %expected_value, i32 %new_value
  ↪  seq_cst seq_cst
  %success = extractvalue { i32, i1 } %result, 1
  br i1 %success, label %exit, label %loop
```

```
exit:
  ; Continue execution
```

11.3.6 Advanced Topics

1. **Weak vs. Strong CAS**

   - **Strong CAS**: Ensures that the operation succeeds only if the comparison is true.
   - **Weak CAS**: May fail spuriously, even if the comparison is true. This can improve performance in certain scenarios.

2. **Memory Ordering Trade-offs**

   - **`seq_cst`**: Provides the strongest guarantees but may incur higher overhead.
   - **`acq_rel`**: Balances performance and ordering guarantees.
   - **`monotonic`**: Offers minimal guarantees but is the fastest.

3. **Use in Lock-Free Data Structures**

   The `cmpxchg` instruction is essential for implementing lock-free data structures, such as:

   - **Lock-free stacks**: Using CAS to update the head pointer.
   - **Lock-free queues**: Using CAS to manage head and tail pointers.
   - **Atomic counters**: Using CAS to increment or decrement values.

11.3.7 Conclusion

The `cmpxchg` instruction is a powerful tool for implementing atomic compare-and-swap operations in LLVM IR. By understanding its syntax, semantics, and use cases, intermediate users can effectively write concurrent programs that are both correct and efficient. Whether implementing lock-free data structures, synchronization primitives, or custom algorithms, the `cmpxchg` instruction provides the necessary functionality to ensure thread-safe operations. This section has provided a comprehensive exploration of the Compare and Swap operation, equipping users with the knowledge to leverage this instruction in their LLVM-based projects.

Chapter 12

Vectorization & SIMD Operations

12.1 Vector Data Types (<4 x i32>)

Vectorization is a powerful technique for improving the performance of computationally intensive programs by leveraging Single Instruction, Multiple Data (SIMD) capabilities of modern processors. In LLVM IR, vector data types and operations are first-class citizens, enabling developers to express and optimize parallel computations directly in the intermediate representation. This section provides a detailed exploration of **vector data types** in LLVM IR, focusing on their syntax, semantics, and use cases.

12.1.1 Overview of Vector Data Types

Vector data types in LLVM IR represent collections of elements of the same type, which can be processed in parallel using SIMD instructions. These types are particularly useful for:

1. **Parallel computations**: Performing the same operation on multiple data elements simultaneously.

2. **Performance optimization**: Exploiting hardware-level parallelism to speed up computations.

3. **Data alignment**: Ensuring that data is aligned for efficient memory access.

In LLVM IR, vector types are explicitly represented using the syntax `<N x T>`, where `N` is the number of elements and `T` is the type of each element. For example, `<4 x i32>` represents a vector of four 32-bit integers.

12.1.2 Key Concepts

1. **SIMD (Single Instruction, Multiple Data)**

   SIMD is a parallel processing technique where a single instruction is applied to multiple data elements simultaneously. Modern CPUs and GPUs provide SIMD instructions (e.g., SSE, AVX, NEON) to accelerate vectorized computations.

2. **Vector Length**

   The length of a vector (`N`) determines the number of elements it contains. Common vector lengths include 2, 4, 8, and 16, depending on the hardware capabilities.

3. **Element Type**

   The element type (`T`) defines the type of each element in the vector. Supported element types include:

   - **Integer types**: `i8`, `i16`, `i32`, `i64`, etc.
   - **Floating-point types**: `float`, `double`.
   - **Pointer types**: `i8*`, `i32*`, etc.

4. **Alignment and Padding**

Vector types often require specific memory alignment to ensure efficient access. LLVM IR provides mechanisms to specify alignment and handle padding.

12.1.3 LLVM IR Syntax for Vector Types

Vector types in LLVM IR are defined using the following syntax:

```
<N x T>
```

- `<N>`: The number of elements in the vector.
- `<T>`: The type of each element.

Examples:

- `<4 x i32>`: A vector of four 32-bit integers.
- `<8 x float>`: A vector of eight 32-bit floating-point numbers.
- `<2 x i64*>`: A vector of two 64-bit integer pointers.

12.1.4 Operations on Vector Types

LLVM IR supports a wide range of operations on vector types, including:

1. **Arithmetic Operations**

   Arithmetic operations (e.g., addition, subtraction, multiplication) can be performed on vectors element-wise.

   Example:

```
%v1 = add <4 x i32> %a, %b
```

This adds the corresponding elements of vectors `%a` and `%b`.

2. **Bitwise Operations**

   Bitwise operations (e.g., AND, OR, XOR) can also be performed element-wise.

   Example:

```
%v2 = and <4 x i32> %a, %b
```

   This performs a bitwise AND operation on the corresponding elements of vectors `%a` and `%b`.

3. **Comparison Operations**

   Comparison operations (e.g., equal, greater than) return a vector of boolean values.

   Example:

```
%v3 = icmp eq <4 x i32> %a, %b
```

   This compares the corresponding elements of vectors `%a` and `%b` for equality.

4. **Load and Store Operations**

   Vectors can be loaded from and stored to memory using `load` and `store` instructions.

   Example:

```
%vec = load <4 x i32>, <4 x i32>* %ptr, align 16
store <4 x i32> %vec, <4 x i32>* %ptr, align 16
```

5. **Shuffle Operations**

   The `shufflevector` instruction creates a new vector by rearranging elements from two input vectors.

   Example:

```
%v4 = shufflevector <4 x i32> %a, <4 x i32> %b, <4 x i32> <i32 0, i32
↪  4, i32 1, i32 5>
```

   This creates a new vector by selecting elements from `%a` and `%b`.

12.1.5 Examples

1. **Vector Addition**

   Consider the following C code, which adds two arrays element-wise:

```
int a[4] = {1, 2, 3, 4};
int b[4] = {5, 6, 7, 8};
int c[4];
for (int i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
}
```

   The corresponding LLVM IR using vector operations is:

```
%a = load <4 x i32>, <4 x i32>* %a_ptr, align 16
%b = load <4 x i32>, <4 x i32>* %b_ptr, align 16
%c = add <4 x i32> %a, %b
store <4 x i32> %c, <4 x i32>* %c_ptr, align 16
```

2. **Vector Comparison**

   Consider the following C code, which compares two arrays element-wise:

```
int a[4] = {1, 2, 3, 4};
int b[4] = {2, 2, 3, 5};
bool result[4];
for (int i = 0; i < 4; i++) {
    result[i] = (a[i] == b[i]);
}
```

   The corresponding LLVM IR using vector operations is:

   llvm

   Copy

```
%a = load <4 x i32>, <4 x i32>* %a_ptr, align 16
%b = load <4 x i32>, <4 x i32>* %b_ptr, align 16
%result = icmp eq <4 x i32> %a, %b
```

12.1.6 Advanced Topics

1. **Vector Length and Hardware Compatibility**

   The choice of vector length (`N`) should align with the hardware's SIMD capabilities. For example:

   - **SSE**: Supports 128-bit vectors (e.g., `<4 x float>`).
   - **AVX**: Supports 256-bit vectors (e.g., `<8 x float>`).
   - **AVX-512**: Supports 512-bit vectors (e.g., `<16 x float>`).

2. **Vectorization and Loop Unrolling**

   Vectorization is often combined with loop unrolling to maximize parallelism. LLVM's auto-vectorization pass can automatically transform scalar loops into vectorized code.

3. **Masked Vector Operations**

   Some hardware supports masked vector operations, where only certain elements of a vector are processed. LLVM IR provides intrinsics for masked operations.

12.1.7 Conclusion

Vector data types in LLVM IR provide a powerful mechanism for expressing and optimizing parallel computations. By understanding their syntax, semantics, and use cases, intermediate users can leverage SIMD capabilities to improve the performance of their programs. Whether performing arithmetic operations, comparisons, or memory accesses, vector types enable efficient and scalable parallel processing.

This section has provided a comprehensive exploration of vector data types, equipping users with the knowledge to implement and optimize vectorized code in their LLVM-based projects.

12.2 SIMD Arithmetic (`add <4 x i32> %vec1, %vec2`)

SIMD (Single Instruction, Multiple Data) arithmetic is a cornerstone of high-performance computing, enabling parallel processing of multiple data elements with a single instruction. In LLVM IR, SIMD arithmetic operations are explicitly supported through vector instructions, allowing developers to harness the power of modern CPUs and GPUs. This section provides a detailed exploration of **SIMD arithmetic** in LLVM IR, focusing on its syntax, semantics, and practical applications.

12.2.1 Overview of SIMD Arithmetic

SIMD arithmetic operations perform the same computation on multiple data elements simultaneously. This is particularly useful for:

1. **Parallel computations**: Accelerating tasks like matrix multiplication, image processing, and scientific simulations.

2. **Performance optimization**: Reducing the number of instructions and improving throughput.

3. **Data-level parallelism**: Exploiting the parallel processing capabilities of modern hardware.

In LLVM IR, SIMD arithmetic is expressed using vector types and instructions, such as `add`, `sub`, `mul`, and `div`. These instructions operate element-wise on vectors, enabling efficient parallel computation.

12.2.2 Key Concepts

1. **Element-Wise Operations**

   SIMD arithmetic operations are performed element-wise, meaning that each element of the input vectors is processed independently. For example, adding two `<4 x i32>` vectors results in a new vector where each element is the sum of the corresponding elements from the input vectors.

2. **Vector Length and Alignment**

   The length of the vector (`N`) determines the number of elements processed in parallel. Proper alignment of vector data in memory is crucial for efficient SIMD operations.

3. **Supported Arithmetic Operations**

   LLVM IR supports a wide range of SIMD arithmetic operations, including:

   - **Addition (`add`)**: Element-wise addition.
   - **Subtraction (`sub`)**: Element-wise subtraction.
   - **Multiplication (`mul`)**: Element-wise multiplication.
   - **Division (`div`)**: Element-wise division.
   - **Bitwise operations (`and`, `or`, `xor`)**: Element-wise bitwise operations.

4. **Floating-Point and Integer Operations**

   SIMD arithmetic supports both integer and floating-point element types, enabling a wide range of applications.

12.2.3 LLVM IR Syntax for SIMD Arithmetic

SIMD arithmetic operations in LLVM IR are expressed using the following syntax:

```
<result> = <operation> <vector_type> <operand1>, <operand2>
```

- `<operation>`: The arithmetic operation (e.g., `add`, `sub`, `mul`, `div`).
- `<vector_type>`: The type of the vectors (e.g., `<4 x i32>`, `<8 x float>`).
- `<operand1>` and `<operand2>`: The input vectors.

Examples:

- **Addition**:

```
%result = add <4 x i32> %vec1, %vec2
```

- **Subtraction**:

```
%result = sub <4 x float> %vec1, %vec2
```

- **Multiplication**:

```
%result = mul <8 x i16> %vec1, %vec2
```

- **Division**:

```
%result = div <4 x double> %vec1, %vec2
```

12.2.4 Detailed Behavior

1. **Element-Wise Computation**

   Each SIMD arithmetic operation computes the result for each pair of corresponding elements in the input vectors. For example:

```
%vec1 = <i32 1, i32 2, i32 3, i32 4>
%vec2 = <i32 5, i32 6, i32 7, i32 8>
%result = add <4 x i32> %vec1, %vec2  ; Result: <i32 6, i32 8, i32 10,
↪  i32 12>
```

2. **Handling Different Element Types**

   SIMD arithmetic operations support various element types, including:

   - **Integer types**: `i8`, `i16`, `i32`, `i64`, etc.
   - **Floating-point types**: `float`, `double`.

   The operation is performed based on the type of the elements. For example, floating-point addition uses the `fadd` instruction:

```
%result = fadd <4 x float> %vec1, %vec2
```

3. **Overflow and Undefined Behavior**

   - **Integer operations**: Overflow behavior depends on the type (e.g., wrapping for unsigned integers, undefined for signed integers).
   - **Floating-point operations**: Follow IEEE 754 rules for special cases (e.g., NaN, infinity).

12.2.5 Examples

1. **Vector Addition**

   Consider the following C code, which adds two arrays element-wise:

```c
int a[4] = {1, 2, 3, 4};
int b[4] = {5, 6, 7, 8};
int c[4];
for (int i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
}
```

   The corresponding LLVM IR using SIMD arithmetic is:

```
%a = load <4 x i32>, <4 x i32>* %a_ptr, align 16
%b = load <4 x i32>, <4 x i32>* %b_ptr, align 16
%c = add <4 x i32> %a, %b
store <4 x i32> %c, <4 x i32>* %c_ptr, align 16
```

2. **Vector Multiplication**

   Consider the following C code, which multiplies two arrays element-wise:

```c
float a[4] = {1.0, 2.0, 3.0, 4.0};
float b[4] = {5.0, 6.0, 7.0, 8.0};
float c[4];
for (int i = 0; i < 4; i++) {
    c[i] = a[i] * b[i];
}
```

   The corresponding LLVM IR using SIMD arithmetic is:

```
%a = load <4 x float>, <4 x float>* %a_ptr, align 16
%b = load <4 x float>, <4 x float>* %b_ptr, align 16
%c = fmul <4 x float> %a, %b
store <4 x float> %c, <4 x float>* %c_ptr, align 16
```

12.2.6 Advanced Topics

1. **Vector Length and Hardware Compatibility**

   The choice of vector length (`N`) should align with the hardware's SIMD capabilities. For example:

   - **SSE**: Supports 128-bit vectors (e.g., `<4 x float>`).
   - **AVX**: Supports 256-bit vectors (e.g., `<8 x float>`).
   - **AVX-512**: Supports 512-bit vectors (e.g., `<16 x float>`).

2. **Masked SIMD Operations**

   Some hardware supports masked SIMD operations, where only certain elements of a vector are processed. LLVM IR provides intrinsics for masked operations.

3. **Vector Reduction**

   Vector reduction operations (e.g., summing all elements of a vector) can be implemented using a combination of SIMD arithmetic and shuffle operations.

12.2.7 Conclusion

SIMD arithmetic in LLVM IR provides a powerful mechanism for expressing and optimizing parallel computations. By understanding its syntax, semantics, and use cases, intermediate users can leverage SIMD capabilities to improve the performance of their programs. Whether

performing addition, subtraction, multiplication, or division, SIMD arithmetic enables efficient and scalable parallel processing.

This section has provided a comprehensive exploration of SIMD arithmetic, equipping users with the knowledge to implement and optimize vectorized code in their LLVM-based projects.

12.3 LLVM Intrinsics for SIMD (`llvm.fma.f32`)

LLVM intrinsics are built-in functions that provide direct access to low-level operations, often mapping to hardware-specific instructions. In the context of SIMD (Single Instruction, Multiple Data), LLVM intrinsics enable developers to leverage advanced hardware features, such as fused multiply-add (FMA) operations, without writing platform-specific code. This section explores **LLVM intrinsics for SIMD**, with a focus on the `llvm.fma.f32` intrinsic and its role in high-performance computing.

12.3.1 Overview of LLVM Intrinsics for SIMD

LLVM intrinsics are designed to expose hardware capabilities in a portable and efficient manner. For SIMD operations, intrinsics provide:

1. **Access to specialized instructions**: Such as FMA, which performs a multiply-add operation in a single step.
2. **Performance optimization**: By reducing the number of instructions and improving throughput.
3. **Portability**: By abstracting hardware-specific details, enabling code to run efficiently across different platforms.

The `llvm.fma.f32` intrinsic is a key example, enabling fused multiply-add operations on floating-point vectors.

12.3.2 Key Concepts

1. **Fused Multiply-Add (FMA)**

   FMA is a common operation in scientific computing and graphics, defined as:

```
result = (a * b) + c
```

FMA performs the multiplication and addition in a single step, reducing rounding errors and improving performance.

2. **Intrinsic Functions**

   Intrinsic functions are built into LLVM and provide direct access to low-level operations. They are often optimized for specific hardware architectures.

3. **Vectorization with Intrinsics**

   LLVM intrinsics can operate on vector types, enabling SIMD parallelism. For example, `llvm.fma.v4f32` performs FMA on a vector of four 32-bit floating-point numbers.

4. **Memory Ordering and Alignment**

   SIMD intrinsics often require aligned memory access for optimal performance. LLVM provides mechanisms to ensure proper alignment.

12.3.3 LLVM Intrinsic: `llvm.fma.f32`

The `llvm.fma.f32` intrinsic performs a fused multiply-add operation on 32-bit floating-point values. Its syntax is as follows:

```
declare float @llvm.fma.f32(float %a, float %b, float %c)
```

- **`%a`**: The first operand (multiplier).
- **`%b`**: The second operand (multiplicand).
- **`%c`**: The third operand (addend).

- **Return value**: The result of `(a * b) + c`.

Example:

```
%result = call float @llvm.fma.f32(float %a, float %b, float %c)
```

12.3.4 Vectorized FMA Intrinsics

LLVM provides vectorized versions of the FMA intrinsic for SIMD operations. For example:

- **`llvm.fma.v4f32`**: FMA for a vector of four 32-bit floating-point numbers.
- **`llvm.fma.v8f32`**: FMA for a vector of eight 32-bit floating-point numbers.

Syntax:

```
declare <4 x float> @llvm.fma.v4f32(<4 x float> %a, <4 x float> %b, <4 x
↪  float> %c)
```

Example:

```
%result = call <4 x float> @llvm.fma.v4f32(<4 x float> %a, <4 x float> %b,
↪  <4 x float> %c)
```

12.3.5 Detailed Behavior

1. **Precision and Rounding**

   FMA operations reduce rounding errors by performing the multiplication and addition in a single step, without intermediate rounding. This is particularly important for numerical stability in scientific computing.

2. **Hardware Support**

   Modern CPUs and GPUs provide dedicated FMA instructions, which are mapped directly to LLVM intrinsics. For example:

   - **Intel**: FMA3 instructions (e.g., `vfmadd132ps`).
   - **ARM**: FMA instructions in the NEON and SVE instruction sets.
   - **NVIDIA GPUs**: FMA instructions in CUDA.

3. **Performance Benefits**

   FMA intrinsics improve performance by:

   - Reducing the number of instructions.
   - Minimizing rounding errors.
   - Leveraging hardware acceleration.

12.3.6 Examples

1. **Scalar FMA**

   Consider the following C code, which performs a fused multiply-add operation:

```c
#include <math.h>
float a = 2.0f, b = 3.0f, c = 4.0f;
float result = fmaf(a, b, c);  // Result: (2.0 * 3.0) + 4.0 = 10.0
```

   The corresponding LLVM IR using the `llvm.fma.f32` intrinsic is:

```
%a = float 2.0
%b = float 3.0
%c = float 4.0
%result = call float @llvm.fma.f32(float %a, float %b, float %c)
```

2. **Vectorized FMA**

   Consider the following C code, which performs FMA on arrays:

```
#include <math.h>
float a[4] = {1.0, 2.0, 3.0, 4.0};
float b[4] = {5.0, 6.0, 7.0, 8.0};
float c[4] = {9.0, 10.0, 11.0, 12.0};
float result[4];
for (int i = 0; i < 4; i++) {
    result[i] = fmaf(a[i], b[i], c[i]);
}
```

   The corresponding LLVM IR using the `llvm.fma.v4f32` intrinsic is:

```
%a = load <4 x float>, <4 x float>* %a_ptr, align 16
%b = load <4 x float>, <4 x float>* %b_ptr, align 16
%c = load <4 x float>, <4 x float>* %c_ptr, align 16
%result = call <4 x float> @llvm.fma.v4f32(<4 x float> %a, <4 x
↪  float> %b, <4 x float> %c)
store <4 x float> %result, <4 x float>* %result_ptr, align 16
```

12.3.7 Advanced Topics

1. **Masked FMA Operations**

Some hardware supports masked FMA operations, where only certain elements of a vector are processed. LLVM provides intrinsics for masked operations, such as `llvm.masked.fma.v4f32`.

2. **Mixed-Precision FMA**

   LLVM supports mixed-precision FMA operations, where the operands and result may have different precisions. For example, `llvm.fma.v4f16` performs FMA on 16-bit floating-point vectors.

3. **Auto-Vectorization with Intrinsics**

   LLVM's auto-vectorization pass can automatically generate SIMD code using intrinsics, reducing the need for manual vectorization.

12.3.8 Conclusion

LLVM intrinsics for SIMD, such as `llvm.fma.f32`, provide a powerful mechanism for leveraging hardware-specific features in a portable and efficient manner. By understanding their syntax, semantics, and use cases, intermediate users can optimize their programs for high-performance computing. Whether performing scalar or vectorized FMA operations, LLVM intrinsics enable efficient and accurate parallel processing.
This section has provided a comprehensive exploration of LLVM intrinsics for SIMD, equipping users with the knowledge to implement and optimize vectorized code in their LLVM-based projects.

Part III

Advanced Users: for metadata, JIT, and advanced optimizations.

Chapter 13

Metadata & Debug Information

13.1 Debug Info (`!llvm.dbg.cu = !{!0}`)

Debug information is a critical component of modern software development, enabling developers to inspect and debug their programs effectively. In LLVM IR, debug information is represented using **metadata**, which provides a structured way to associate source-level information with the intermediate representation. This section provides a detailed exploration of **debug information** in LLVM IR, focusing on its representation, semantics, and practical applications.

13.1.1 Overview of Debug Information

Debug information in LLVM IR serves the following purposes:

1. **Source-level debugging**: Enables developers to step through their code, inspect variables, and set breakpoints at the source level.
2. **Error reporting**: Provides meaningful error messages that reference the original source

code.

3. **Optimization feedback**: Helps developers understand how optimizations affect their code.

LLVM uses metadata to represent debug information, which is embedded directly in the IR. This metadata is preserved throughout the compilation process and can be used by debugging tools, such as GDB or LLDB.

13.1.2 Key Concepts

1. **Metadata**

   Metadata in LLVM IR is used to attach additional information to instructions, functions, and other IR entities. Debug information is represented using specialized metadata nodes.

2. **Debug Information Metadata**

   Debug information metadata includes:

   - **Compilation unit metadata**: Represents a source file being compiled.
   - **Function metadata**: Describes functions and their source locations.
   - **Variable metadata**: Describes variables and their types.
   - **Type metadata**: Describes source-level types.

3. **Debug Information Format**

   LLVM uses the **DWARF** (Debugging With Attributed Record Formats) standard to represent debug information. DWARF is a widely used format for debugging information in executable files.

4. **Debug Information Intrinsics**

   LLVM provides intrinsics, such as `llvm.dbg.declare` and `llvm.dbg.value`, to associate debug information with IR instructions.

13.1.3 LLVM Metadata Syntax

Metadata in LLVM IR is represented using the following syntax:

```
!metadata_name = !{!metadata_node}
```

- **`!metadata_name`**: The name of the metadata node.
- **`!metadata_node`**: The content of the metadata node, which can include other metadata nodes or constants.

Example:

```
!llvm.dbg.cu = !{!0}
```

This defines a metadata node named `!llvm.dbg.cu` with the content `!{!0}`.

13.1.4 Debug Information Metadata Nodes

1. **Compilation Unit Metadata (`!llvm.dbg.cu`)**

   The `!llvm.dbg.cu` metadata node represents a compilation unit, which corresponds to a source file being compiled. It contains references to other metadata nodes, such as functions and types.

   Example:

```
!llvm.dbg.cu = !{!0}
!0 = !DICompileUnit(language: DW_LANG_C, file: !1, producer: "clang",
↪  isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug)
```

- **`language`**: The source language (e.g., `DW_LANG_C` for C).
- **`file`**: The source file metadata node.
- **`producer`**: The compiler that generated the IR.
- **`isOptimized`**: Indicates whether the code is optimized.
- **`runtimeVersion`**: The runtime version (if applicable).
- **`emissionKind`**: The level of debug information (e.g., `FullDebug`).

2. **File Metadata (`!DIFile`)**

   The `!DIFile` metadata node represents a source file.

   Example:

```
!1 = !DIFile(filename: "example.c", directory: "/path/to/source")
```

   - **`filename`**: The name of the source file.
   - **`directory`**: The directory containing the source file.

3. **Function Metadata (`!DISubprogram`)**

   The `!DISubprogram` metadata node represents a function and its source-level information.

   Example:

```
!2 = !DISubprogram(name: "main", scope: !1, file: !1, line: 5, type:
↪  !3, isLocal: false, isDefinition: true, scopeLine: 5, flags:
↪  DIFlagPrototyped, isOptimized: false)
```

- **`name`**: The name of the function.
- **`scope`**: The scope of the function (e.g., the file).
- **`file`**: The source file metadata node.
- **`line`**: The line number where the function is defined.
- **`type`**: The function type metadata node.
- **`isLocal`**: Indicates whether the function is local.
- **`isDefinition`**: Indicates whether the function is defined (not just declared).
- **`scopeLine`**: The line number of the function's scope.
- **`flags`**: Additional flags (e.g., `DIFlagPrototyped`).
- **`isOptimized`**: Indicates whether the function is optimized.

4. **Variable Metadata (`!DILocalVariable`)**

   The `!DILocalVariable` metadata node represents a local variable.

   Example:

```
!4 = !DILocalVariable(name: "x", scope: !2, file: !1, line: 6, type:
↪  !5, arg: 1)
```

- **`name`**: The name of the variable.
- **`scope`**: The scope of the variable (e.g., the function).

- **`file`**: The source file metadata node.
- **`line`**: The line number where the variable is defined.
- **`type`**: The variable type metadata node.
- **`arg`**: The argument number (if the variable is a function parameter).

5. **Type Metadata (`!DIBasicType`)**

   The `!DIBasicType` metadata node represents a source-level type.

   Example:

```
!5 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
```

   - **`name`**: The name of the type.
   - **`size`**: The size of the type in bits.
   - **`encoding`**: The encoding of the type (e.g., `DW_ATE_signed` for signed integers).

13.1.5 Debug Information Intrinsics

1. **`llvm.dbg.declare`**

   The `llvm.dbg.declare` intrinsic associates a variable with its debug information.

   Example:

```
call void @llvm.dbg.declare(metadata i32* %x, metadata !4, metadata
↪  !DIExpression())
```

   - **`%x`**: The variable to associate with debug information.

- **`!4`**: The variable metadata node.
- **`!DIExpression()`**: An optional expression for complex variable locations.

2. **`llvm.dbg.value`**

   The `llvm.dbg.value` intrinsic associates a value with its debug information.

   Example:

```
call void @llvm.dbg.value(metadata i32 %x, metadata !4, metadata
↪  !DIExpression())
```

   - **`%x`**: The value to associate with debug information.
   - **`!4`**: The variable metadata node.
   - **`!DIExpression()`**: An optional expression for complex value locations.

13.1.6 Examples

1. **Debug Information for a Function**

   Consider the following C code:

```
int main() {
    int x = 42;
    return x;
}
```

   The corresponding LLVM IR with debug information is:

```
!llvm.dbg.cu = !{!0}
!0 = !DICompileUnit(language: DW_LANG_C, file: !1, producer: "clang",
↪  isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug)
!1 = !DIFile(filename: "example.c", directory: "/path/to/source")
!2 = !DISubprogram(name: "main", scope: !1, file: !1, line: 1, type:
↪  !3, isLocal: false, isDefinition: true, scopeLine: 1, flags:
↪  DIFlagPrototyped, isOptimized: false)
!3 = !DISubroutineType(types: !{!4})
!4 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!5 = !DILocalVariable(name: "x", scope: !2, file: !1, line: 2, type:
↪  !4)

define i32 @main() !dbg !2 {
  %x = alloca i32, align 4
  call void @llvm.dbg.declare(metadata i32* %x, metadata !5, metadata
  ↪  !DIExpression()), !dbg !6
  store i32 42, i32* %x, align 4, !dbg !6
  %1 = load i32, i32* %x, align 4, !dbg !7
  ret i32 %1, !dbg !7
}
```

13.1.7 Advanced Topics

1. **Debug Information and Optimizations**

   Debug information must be preserved during optimizations to ensure accurate debugging. LLVM provides mechanisms to maintain debug information, such as `llvm.dbg.value` for tracking variable values.

2. **Debug Information for Inlined Functions**

Inlined functions require special handling to preserve debug information. LLVM uses `!DILexicalBlock` and `!DILocation` metadata to represent inlined code.

3. **Debug Information for Complex Types**

   Complex types, such as structs and arrays, are represented using `!DICompositeType` metadata.

13.1.8 Conclusion

Debug information in LLVM IR provides a powerful mechanism for source-level debugging and error reporting. By understanding its representation, semantics, and use cases, advanced users can effectively debug and optimize their programs. Whether working with compilation units, functions, variables, or types, debug information metadata enables accurate and meaningful debugging.

This section has provided a comprehensive exploration of debug information in LLVM IR, equipping users with the knowledge to implement and optimize debug information in their LLVM-based projects.

13.2 Module Flags (`!llvm.module.flags = !{!0}`)

Module flags in LLVM IR are a form of metadata used to convey important information about a module, such as its target architecture, optimization level, or debugging settings. These flags are essential for guiding the behavior of LLVM tools, such as the optimizer, code generator, and linker. This section provides a detailed exploration of **module flags** in LLVM IR, focusing on their syntax, semantics, and practical applications.

13.2.1 Overview of Module Flags

Module flags are metadata nodes that describe properties of an LLVM module. They serve the following purposes:

1. **Guiding optimizations**: Specifying optimization levels or target-specific behavior.
2. **Enabling debugging**: Indicating whether debug information is present.
3. **Controlling linking**: Specifying how modules should be linked together.
4. **Target-specific settings**: Providing information about the target architecture or ABI.

Module flags are represented using the `!llvm.module.flags` metadata node, which contains a list of flag definitions.

13.2.2 Key Concepts

1. **Module Flag Metadata**

   Module flags are represented as metadata nodes with the following structure:

```
!llvm.module.flags = !{!0, !1, !2, ...}
```

Each flag (`!0`, `!1`, etc.) is a metadata node that defines a specific property of the module.

2. **Flag Behavior**

   Each module flag has a **behavior** that determines how it should be interpreted. The behavior is specified as an integer value, with the following common options:

   - **`1` (Error)**: The flag must have the same value in all modules being linked. If not, an error is raised.
   - **`2` (Warning)**: The flag should have the same value in all modules being linked. If not, a warning is issued.
   - **`3` (Require)**: The flag must have the same value in all modules being linked. If not, the linker will override the value.
   - **`4` (Override)**: The flag's value in the current module overrides any conflicting values in other modules.
   - **`5` (Append)**: The flag's value is appended to the list of values from other modules.
   - **`6` (AppendUnique)**: The flag's value is appended to the list of values from other modules, but duplicates are removed.

3. **Common Module Flags**

   Some commonly used module flags include:

   - **`wchar_size`**: The size of the `wchar_t` type.
   - **`PIC Level`**: The level of position-independent code (PIC).

- **`Debug Info Version`**: The version of debug information being used.
- **`Objective-C Image Info`**: Information about the Objective-C runtime.

13.2.3 LLVM Metadata Syntax for Module Flags

Module flags are defined using the following syntax:

```
!llvm.module.flags = !{!0, !1, !2, ...}
!0 = !{i32 <behavior>, <flag_name>, <flag_value>}
```

- **`<behavior>`**: The behavior of the flag (e.g., `1` for Error, `2` for Warning).
- **`<flag_name>`**: The name of the flag (e.g., `"wchar_size"`, `"PIC Level"`).
- **`<flag_value>`**: The value of the flag (e.g., `i32 4`, `i32 2`).

Example:

```
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"wchar_size", i32 4}
```

This defines a module flag named `"wchar_size"` with a value of `4` and a behavior of `1` (Error).

13.2.4 Detailed Behavior

1. **Flag Behavior and Linking**

   The behavior of a module flag determines how it is handled during linking:

   - **Error (`1`)**: Ensures that all modules have the same value for the flag. If not, linking fails.

- **Warning (`2`)**: Issues a warning if modules have different values for the flag.
- **Require (`3`)**: Ensures that all modules have the same value for the flag. If not, the linker overrides the value.
- **Override (`4`)**: The flag's value in the current module takes precedence over conflicting values in other modules.
- **Append (`5`)**: Combines the flag's value with values from other modules.
- **AppendUnique (`6`)**: Combines the flag's value with values from other modules, removing duplicates.

2. **Common Flag Values**

   - **`wchar_size`**: Specifies the size of the `wchar_t` type (e.g., `4` for 32-bit, `2` for 16-bit).
   - **`PIC Level`**: Specifies the level of position-independent code (e.g., `1` for PIC, `2` for PIE).
   - **`Debug Info Version`**: Specifies the version of debug information (e.g., `3` for DWARF version 3).
   - **`Objective-C Image Info`**: Specifies information about the Objective-C runtime (e.g., version, ABI).

13.2.5 Examples

1. **Specifying `wchar_size`**

   Consider a module that requires `wchar_t` to be 4 bytes:

```
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"wchar_size", i32 4}
```

This flag ensures that all linked modules use a 4-byte `wchar_t`.

2. **Specifying PIC Level**

   Consider a module that requires position-independent code at level 2:

```
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"PIC Level", i32 2}
```

   This flag ensures that all linked modules use PIC level 2.

3. **Specifying Debug Info Version**

   Consider a module that uses DWARF version 4 for debug information:

```
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"Debug Info Version", i32 4}
```

   This flag ensures that all linked modules use DWARF version 4.

13.2.6 Advanced Topics

1. **Custom Module Flags**

   Developers can define custom module flags to convey module-specific information. For example:

```
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"CustomFlag", i32 42}
```

This defines a custom flag named `"CustomFlag"` with a value of `42`.

2. **Module Flags and LTO (Link-Time Optimization)**

   Module flags play a crucial role in Link-Time Optimization (LTO), ensuring that all modules have consistent settings for optimization, debugging, and target-specific behavior.

3. **Module Flags and Target Triples**

   Module flags often work in conjunction with the target triple, which specifies the target architecture, vendor, and operating system. For example:

```
target triple = "x86_64-unknown-linux-gnu"
!llvm.module.flags = !{!0}
!0 = !{i32 1, !"wchar_size", i32 4}
```

13.2.7 Conclusion

Module flags in LLVM IR provide a powerful mechanism for conveying important information about a module, such as its target architecture, optimization level, and debugging settings. By understanding their syntax, semantics, and use cases, advanced users can effectively guide the behavior of LLVM tools and ensure consistent behavior across linked modules.

This section has provided a comprehensive exploration of module flags, equipping users with the knowledge to implement and optimize module-specific settings in their LLVM-based projects.

13.3 Source Locations & Variables (`!DILocation`, `!DILocalVariable`)

Source locations and variable metadata are essential components of debug information in LLVM IR, enabling developers to map intermediate representation (IR) constructs back to their original source code. This section provides a detailed exploration of **source location metadata** (`!DILocation`) and **local variable metadata** (`!DILocalVariable`), focusing on their syntax, semantics, and practical applications.

13.3.1 Overview of Source Locations & Variables

Debug information in LLVM IR serves the following purposes:

1. **Source-level debugging**: Enables developers to step through their code, inspect variables, and set breakpoints at the source level.
2. **Error reporting**: Provides meaningful error messages that reference the original source code.
3. **Optimization feedback**: Helps developers understand how optimizations affect their code.

Source locations and variable metadata are represented using specialized metadata nodes, such as `!DILocation` and `!DILocalVariable`. These nodes provide a structured way to associate IR instructions with their corresponding source-level constructs.

13.3.2 Key Concepts

1. **Source Location Metadata (`!DILocation`)**

The `!DILocation` metadata node represents a specific location in the source code, such as a line number or column number. It is used to associate IR instructions with their corresponding source-level constructs.

2. **Local Variable Metadata (`!DILocalVariable`)**

   The `!DILocalVariable` metadata node represents a local variable in the source code. It describes the variable's name, type, scope, and location.

3. **Debug Information Intrinsics**

   LLVM provides intrinsics, such as `llvm.dbg.declare` and `llvm.dbg.value`, to associate debug information with IR instructions.

4. **Debug Information Format**

   LLVM uses the **DWARF** (Debugging With Attributed Record Formats) standard to represent debug information. DWARF is a widely used format for debugging information in executable files.

13.3.3 LLVM Metadata Syntax

Metadata in LLVM IR is represented using the following syntax:

```
!metadata_name = !{!metadata_node}
```

- **`!metadata_name`**: The name of the metadata node.
- **`!metadata_node`**: The content of the metadata node, which can include other metadata nodes or constants.

13.3.4 Source Location Metadata (`!DILocation`)

The `!DILocation` metadata node represents a specific location in the source code. Its syntax is as follows:

```
!DILocation(line: <line>, column: <column>, scope: <scope>, inlinedAt:
↪  <inlinedAt>)
```

- **`line`**: The line number in the source file.
- **`column`**: The column number in the source file.
- **`scope`**: The scope of the location (e.g., a function or lexical block).
- **`inlinedAt`**: The location where the code was inlined (if applicable).

Example:

```
!10 = !DILocation(line: 5, column: 10, scope: !2)
```

This defines a source location at line 5, column 10, within the scope of `!2`.

13.3.5 Local Variable Metadata (`!DILocalVariable`)

The `!DILocalVariable` metadata node represents a local variable in the source code. Its syntax is as follows:

```
!DILocalVariable(name: <name>, scope: <scope>, file: <file>, line: <line>,
↪  type: <type>, arg: <arg>)
```

- **`name`**: The name of the variable.

- **`scope`**: The scope of the variable (e.g., a function or lexical block).
- **`file`**: The source file metadata node.
- **`line`**: The line number where the variable is defined.
- **`type`**: The variable type metadata node.
- **`arg`**: The argument number (if the variable is a function parameter).

Example:

```
!4 = !DILocalVariable(name: "x", scope: !2, file: !1, line: 6, type: !5,
↪  arg: 1)
```

This defines a local variable named `"x"` at line 6, within the scope of `!2`, with type `!5`.

13.3.6 Debug Information Intrinsics

1. **`llvm.dbg.declare`**

   The `llvm.dbg.declare` intrinsic associates a variable with its debug information.

   Example:

   ```
   call void @llvm.dbg.declare(metadata i32* %x, metadata !4, metadata
   ↪  !DIExpression())
   ```

   - **`%x`**: The variable to associate with debug information.
   - **`!4`**: The variable metadata node.
   - **`!DIExpression()`**: An optional expression for complex variable locations.

2. **`llvm.dbg.value`**

   The `llvm.dbg.value` intrinsic associates a value with its debug information.

   Example:

```
call void @llvm.dbg.value(metadata i32 %x, metadata !4, metadata
↪  !DIExpression())
```

   - **`%x`**: The value to associate with debug information.
   - **`!4`**: The variable metadata node.
   - **`!DIExpression()`**: An optional expression for complex value locations.

13.3.7 Examples

1. **Debug Information for a Function**

   Consider the following C code:

```
int main() {
    int x = 42;
    return x;
}
```

   The corresponding LLVM IR with debug information is:

```
!llvm.dbg.cu = !{!0}
!0 = !DICompileUnit(language: DW_LANG_C, file: !1, producer: "clang",
↪  isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug)
!1 = !DIFile(filename: "example.c", directory: "/path/to/source")
!2 = !DISubprogram(name: "main", scope: !1, file: !1, line: 1, type:
↪  !3, isLocal: false, isDefinition: true, scopeLine: 1, flags:
↪  DIFlagPrototyped, isOptimized: false)
```

```
!3 = !DISubroutineType(types: !{!4})
!4 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!5 = !DILocalVariable(name: "x", scope: !2, file: !1, line: 2, type:
↪  !4)

define i32 @main() !dbg !2 {
  %x = alloca i32, align 4
  call void @llvm.dbg.declare(metadata i32* %x, metadata !5, metadata
  ↪  !DIExpression()), !dbg !6
  store i32 42, i32* %x, align 4, !dbg !6
  %1 = load i32, i32* %x, align 4, !dbg !7
  ret i32 %1, !dbg !7
}
```

2. **Debug Information for Inlined Code**

   Consider the following C code with an inlined function:

```
inline int add(int a, int b) {
    return a + b;
}

int main() {
    int x = add(1, 2);
    return x;
}
```

   The corresponding LLVM IR with debug information is:

```
!llvm.dbg.cu = !{!0}
!0 = !DICompileUnit(language: DW_LANG_C, file: !1, producer: "clang",
↪  isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug)
```

```
!1 = !DIFile(filename: "example.c", directory: "/path/to/source")
!2 = !DISubprogram(name: "main", scope: !1, file: !1, line: 1, type:
↪  !3, isLocal: false, isDefinition: true, scopeLine: 1, flags:
↪  DIFlagPrototyped, isOptimized: false)
!3 = !DISubroutineType(types: !{!4})
!4 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!5 = !DILocalVariable(name: "x", scope: !2, file: !1, line: 5, type:
↪  !4)
!6 = !DILocation(line: 5, column: 10, scope: !2)

define i32 @main() !dbg !2 {
  %x = alloca i32, align 4
  call void @llvm.dbg.declare(metadata i32* %x, metadata !5, metadata
  ↪  !DIExpression()), !dbg !6
  store i32 3, i32* %x, align 4, !dbg !6
  %1 = load i32, i32* %x, align 4, !dbg !7
  ret i32 %1, !dbg !7
}
```

13.3.8 Advanced Topics

1. **Debug Information and Optimizations**

   Debug information must be preserved during optimizations to ensure accurate debugging. LLVM provides mechanisms to maintain debug information, such as `llvm.dbg.value` for tracking variable values.

2. **Debug Information for Inlined Functions**

   Inlined functions require special handling to preserve debug information. LLVM uses `!DILexicalBlock` and `!DILocation` metadata to represent inlined code.

3. **Debug Information for Complex Types**

   Complex types, such as structs and arrays, are represented using `!DICompositeType` metadata.

13.3.9 Conclusion

Source locations and variable metadata in LLVM IR provide a powerful mechanism for source-level debugging and error reporting. By understanding their representation, semantics, and use cases, advanced users can effectively debug and optimize their programs. Whether working with source locations, local variables, or inlined code, debug information metadata enables accurate and meaningful debugging.

This section has provided a comprehensive exploration of source locations and variable metadata, equipping users with the knowledge to implement and optimize debug information in their LLVM-based projects.

Chapter 14

LLVM Passes & Optimizations

14.1 Running Passes with `opt`

LLVM is renowned for its powerful and flexible optimization framework, which allows developers to analyze, transform, and optimize intermediate representation (IR) code. The `opt` tool is a central component of this framework, enabling users to apply **passes**—modular units of optimization or analysis—to LLVM IR. This section provides a detailed exploration of running passes with `opt`, focusing on its usage, capabilities, and practical applications for advanced users.

14.1.1 Overview of LLVM Passes

LLVM passes are the building blocks of the LLVM optimization pipeline. They perform specific tasks, such as:

1. **Optimizations**: Improving the performance or size of the generated code (e.g., dead code elimination, loop unrolling).

2. **Analyses**: Gathering information about the IR (e.g., alias analysis, dominance analysis).

3. **Transformations**: Modifying the IR (e.g., inlining functions, simplifying instructions).

Passes can be combined into **pipelines** to perform a sequence of optimizations and analyses. The `opt` tool is the primary interface for applying these passes to LLVM IR files.

14.1.2 Key Concepts

1. **The `opt` Tool**

   The `opt` tool is a command-line utility that applies passes to LLVM IR files. It reads IR, runs the specified passes, and outputs the transformed IR or analysis results.

2. **Types of Passes**

   - **Analysis Passes**: Collect information about the IR but do not modify it.
   - **Transformation Passes**: Modify the IR to optimize or transform it.
   - **Utility Passes**: Perform auxiliary tasks, such as printing IR or verifying its correctness.

3. **Pass Pipelines**

   A pass pipeline is a sequence of passes that are executed in a specific order. LLVM provides predefined pipelines (e.g., `-O1`, `-O2`, `-O3`) as well as the ability to define custom pipelines.

4. **Pass Managers**

   LLVM uses **pass managers** to schedule and execute passes. The `opt` tool provides interfaces for interacting with pass managers, such as the **legacy pass manager** and the **new pass manager**.

14.1.3 Using the `opt` Tool

The `opt` tool is invoked from the command line with the following syntax:

```
opt [options] <input.ll> -o <output.ll>
```

- **`<input.ll>`**: The input LLVM IR file.
- **`<output.ll>`**: The output LLVM IR file (optional).
- **`[options]`**: Flags and arguments to control the behavior of `opt`.

14.1.4 Common `opt` Options

1. **Optimization Levels**

   - **`-O0`**: No optimizations (default).
   - **`-O1`**: Minimal optimizations.
   - **`-O2`**: Moderate optimizations.
   - **`-O3`**: Aggressive optimizations.
   - **`-Os`**: Optimize for size.
   - **`-Oz`**: Aggressively optimize for size.

   Example:

   ```
   opt -O2 input.ll -o output.ll
   ```

2. **Individual Passes**

   Individual passes can be specified using the `-pass-name` syntax. For example:

- **`-instcombine`**: Combine redundant instructions.
- **`-dce`**: Perform dead code elimination.
- **`-inline`**: Inline functions.

Example:

```
opt -instcombine -dce input.ll -o output.ll
```

3. **Analysis Passes**

   Analysis passes can be run to gather information about the IR. For example:

   - **`-print-alias-sets`**: Print alias analysis results.
   - **`-print-dom-info`**: Print dominance information.

   Example:

   ```
   opt -print-alias-sets input.ll
   ```

4. **Debugging and Diagnostics**

   - **`-debug`**: Enable debug output.
   - **`-stats`**: Print statistics about the IR.
   - **`-verify`**: Verify the IR after each pass.

   Example:

```
opt -verify -stats input.ll -o output.ll
```

14.1.5 Examples

1. **Running a Predefined Optimization Pipeline**

   To apply the `-O2` optimization pipeline to an IR file:

   ```
   opt -O2 input.ll -o output.ll
   ```

2. **Running Individual Passes**

   To combine instructions and eliminate dead code:

   ```
   opt -instcombine -dce input.ll -o output.ll
   ```

3. **Running Analysis Passes**

   To print alias analysis results:

   ```
   opt -print-alias-sets input.ll
   ```

4. **Debugging with `opt`**

   To verify the IR and print statistics:

   ```
   opt -verify -stats input.ll -o output.ll
   ```

14.1.6 Advanced Topics

1. **Custom Pass Pipelines**

   Advanced users can define custom pass pipelines using the `-passes` option with the new pass manager. For example:

   ```
   opt -passes='instcombine,dce' input.ll -o output.ll
   ```

2. **Legacy vs. New Pass Manager**

   - **Legacy Pass Manager**: The older pass manager, which uses the `-pass-name` syntax.
   - **New Pass Manager**: The modern pass manager, which uses the `-passes` syntax and provides better performance and flexibility.

   Example (New Pass Manager):

   ```
   opt -passes='inline,loop-unroll' input.ll -o output.ll
   ```

3. **Writing Custom Passes**

   Advanced users can write custom passes in C++ and load them into `opt` using the `-load` option. For example:

   ```
   opt -load ./MyPass.so -mypass input.ll -o output.ll
   ```

14.1.7 Conclusion

The `opt` tool is a powerful and versatile utility for applying LLVM passes to IR code. By understanding its usage, options, and capabilities, advanced users can effectively optimize, analyze, and transform their programs. Whether running predefined optimization pipelines, individual passes, or custom analyses, `opt` provides the flexibility and control needed to work with LLVM IR at an advanced level.

This section has provided a comprehensive exploration of running passes with `opt`, equipping users with the knowledge to leverage LLVM's optimization framework in their projects.

14.2 Common Optimization Passes

LLVM provides a rich set of optimization passes that transform and improve intermediate representation (IR) code. These passes are modular, allowing developers to apply specific optimizations tailored to their needs. This section explores four common optimization passes in detail:

1. **`-mem2reg`**: Promote memory to registers.
2. **`-instcombine`**: Combine redundant instructions.
3. **`-loop-unroll`**: Unroll loops.
4. **`-gvn`**: Perform Global Value Numbering.

Each pass is explained with its purpose, behavior, and practical examples, providing advanced users with the knowledge to apply these optimizations effectively.

14.2.1 `-mem2reg`: Promote Memory to Registers

1. **Overview**

   The `-mem2reg` pass converts memory-based variables (allocated with `alloca`) into register-based variables. This optimization is crucial for improving performance by reducing memory accesses and enabling further optimizations.

2. **Key Concepts**

   - **`alloca` Instruction**: Allocates memory on the stack for local variables.
   - **SSA Form**: Static Single Assignment (SSA) form is a property of IR where each variable is assigned exactly once.

- **Phi Nodes**: Used in SSA form to merge values from different control flow paths.

3. **Behavior**

   - **Input**: IR with `alloca` instructions for local variables.
   - **Output**: IR with `alloca` instructions replaced by registers and Phi nodes where necessary.
   - **Limitations**: Only works on variables with a single dominating definition and no complex memory operations.

4. **Example**

   Consider the following C code:

```c
int foo(int x) {
    int y = x + 1;
    return y;
}
```

   The initial IR might look like this:

```
define i32 @foo(i32 %x) {
  %y = alloca i32
  %1 = add i32 %x, 1
  store i32 %1, i32* %y
  %2 = load i32, i32* %y
  ret i32 %2
}
```

   After applying `-mem2reg`, the IR is optimized to:

```
define i32 @foo(i32 %x) {
  %1 = add i32 %x, 1
  ret i32 %1
}
```

14.2.2 `-instcombine`: Combine Redundant Instructions

1. **Overview**

   The `-instcombine` pass simplifies and combines redundant instructions, reducing the complexity of the IR and improving performance.

2. **Key Concepts**

   - **Peephole Optimization**: Local optimizations that replace small sequences of instructions with more efficient equivalents.
   - **Constant Folding**: Evaluating expressions at compile time.
   - **Algebraic Simplification**: Applying mathematical identities to simplify expressions.

3. **Behavior**

   - **Input**: IR with redundant or suboptimal instructions.
   - **Output**: IR with simplified and combined instructions.
   - **Examples**:
     - Replacing `add x, 0` with `x`.
     - Folding `mul 2, 3` into `6`.
     - Combining `add x, (sub y, z)` into `add (sub y, z), x`.

4. **Example**

   Consider the following IR:

```
%1 = add i32 %x, 0
%2 = mul i32 %1, 2
%3 = add i32 %2, 3
```

   After applying `-instcombine`, the IR is optimized to:

```
%1 = shl i32 %x, 1
%2 = add i32 %1, 3
```

14.2.3 `-loop-unroll`: Unroll Loops

1. **Overview**

   The `-loop-unroll` pass duplicates the body of a loop to reduce the overhead of loop control and enable further optimizations.

2. **Key Concepts**

   - **Loop Unrolling Factor**: The number of times the loop body is duplicated.
   - **Partial Unrolling**: Unrolling a loop by a factor that does not divide the iteration count evenly.
   - **Runtime Unrolling**: Unrolling loops with a non-constant iteration count.

3. **Behavior**

   - **Input**: IR with loops.

- **Output**: IR with unrolled loops.
- **Limitations**: May increase code size and register pressure.

4. **Example**

   Consider the following C code:

```
for (int i = 0; i < 4; i++) {
    a[i] = b[i] + c[i];
}
```

   The initial IR might look like this:

```
for.body:
  %i = phi i32 [ 0, %entry ], [ %i.next, %for.body ]
  %a.i = getelementptr i32, i32* %a, i32 %i
  %b.i = getelementptr i32, i32* %b, i32 %i
  %c.i = getelementptr i32, i32* %c, i32 %i
  %b.val = load i32, i32* %b.i
  %c.val = load i32, i32* %c.i
  %sum = add i32 %b.val, %c.val
  store i32 %sum, i32* %a.i
  %i.next = add i32 %i, 1
  %cmp = icmp slt i32 %i.next, 4
  br i1 %cmp, label %for.body, label %for.end
```

   After applying `-loop-unroll`, the IR is optimized to:

```
  %a.0 = getelementptr i32, i32* %a, i32 0
  %b.0 = getelementptr i32, i32* %b, i32 0
  %c.0 = getelementptr i32, i32* %c, i32 0
```

```
%b.val.0 = load i32, i32* %b.0
%c.val.0 = load i32, i32* %c.0
%sum.0 = add i32 %b.val.0, %c.val.0
store i32 %sum.0, i32* %a.0

%a.1 = getelementptr i32, i32* %a, i32 1
%b.1 = getelementptr i32, i32* %b, i32 1
%c.1 = getelementptr i32, i32* %c, i32 1
%b.val.1 = load i32, i32* %b.1
%c.val.1 = load i32, i32* %c.1
%sum.1 = add i32 %b.val.1, %c.val.1
store i32 %sum.1, i32* %a.1

; Repeat for i = 2 and i = 3
```

14.2.4 `-gvn`: Perform Global Value Numbering

1. **Overview**

   The `-gvn` pass eliminates redundant computations by identifying and replacing duplicate expressions with a single computed value.

2. **Key Concepts**

   - **Value Numbering**: Assigning unique identifiers to expressions based on their operands and operations.
   - **Redundancy Elimination**: Replacing duplicate expressions with previously computed values.
   - **Memory Dependencies**: Handling memory operations to ensure correctness.

3. **Behavior**

- **Input**: IR with redundant computations.
- **Output**: IR with redundant computations eliminated.
- **Examples**:
  - Replacing `%1 = add i32 %x, %y` and `%2 = add i32 %x, %y` with a single computation.
  - Eliminating redundant loads and stores.

4. **Example**

   Consider the following IR:

```
%1 = add i32 %x, %y
%2 = add i32 %x, %y
%3 = mul i32 %1, %2
```

   After applying `-gvn`, the IR is optimized to:

```
%1 = add i32 %x, %y
%3 = mul i32 %1, %1
```

14.2.5 Conclusion

The `-mem2reg`, `-instcombine`, `-loop-unroll`, and `-gvn` passes are powerful tools for optimizing LLVM IR. By understanding their behavior and use cases, advanced users can apply these passes effectively to improve the performance and efficiency of their programs. Whether promoting memory to registers, simplifying instructions, unrolling loops, or eliminating redundant computations, these passes form the foundation of LLVM's optimization framework.

This section has provided a comprehensive exploration of common optimization passes, equipping users with the knowledge to leverage LLVM's optimization capabilities in their projects.

Chapter 15

Target-Specific Code Generation

15.1 Specifying the Target (`target triple = "x86_64-pc-linux-gnu"`)

Target-specific code generation is a critical aspect of LLVM's compilation process, enabling the generation of machine code tailored to specific hardware architectures and operating systems. The **target triple** is a key mechanism for specifying the target platform in LLVM IR. This section provides a detailed exploration of target triples, their syntax, semantics, and practical applications for advanced users.

15.1.1 Overview of Target Triples

A **target triple** is a string that describes the target architecture, vendor, and operating system for which code is being generated. It is used by LLVM to select the appropriate backend, instruction set, and ABI (Application Binary Interface) for code generation.
The target triple is specified in LLVM IR using the following syntax:

```
target triple = "<architecture>-<vendor>-<os>-<abi>"
```

- **<architecture>**: The target CPU architecture (e.g., `x86_64`, `arm`, `aarch64`).
- **<vendor>**: The vendor of the target platform (e.g., `pc`, `apple`, `unknown`).
- **<os>**: The target operating system (e.g., `linux`, `windows`, `darwin`).
- **<abi>**: The target ABI (e.g., `gnu`, `msvc`, `eabi`).

Example:

```
target triple = "x86_64-pc-linux-gnu"
```

This specifies a target with an x86_64 architecture, a generic PC vendor, the Linux operating system, and the GNU ABI.

15.1.2 Key Concepts

1. **Architecture**

   The architecture component of the target triple specifies the CPU architecture for which code is being generated. Common architectures include:

   - **`x86_64`**: 64-bit x86 architecture.
   - **`i386`**: 32-bit x86 architecture.
   - **`arm`**: 32-bit ARM architecture.
   - **`aarch64`**: 64-bit ARM architecture.
   - **`riscv32`**: 32-bit RISC-V architecture.
   - **`riscv64`**: 64-bit RISC-V architecture.

2. **Vendor**

   The vendor component specifies the manufacturer or platform vendor. Common vendors include:

   - **`pc`**: Generic PC platform.
   - **`apple`**: Apple platforms (e.g., macOS, iOS).
   - **`unknown`**: Unknown or unspecified vendor.

3. **Operating System**

   The operating system component specifies the target OS. Common operating systems include:

   - **`linux`**: Linux-based systems.
   - **`windows`**: Microsoft Windows.
   - **`darwin`**: Apple macOS and iOS.
   - **`freebsd`**: FreeBSD.
   - **`android`**: Android.

4. **ABI**

   The ABI component specifies the application binary interface, which defines calling conventions, data types, and other low-level details. Common ABIs include:

   - **`gnu`**: GNU ABI (used on Linux and other Unix-like systems).
   - **`msvc`**: Microsoft Visual C++ ABI (used on Windows).
   - **`eabi`**: Embedded ABI (used on embedded systems).

15.1.3 Specifying the Target Triple in LLVM IR

The target triple is specified at the module level in LLVM IR using the `target triple` attribute.
Example:

```
target triple = "x86_64-pc-linux-gnu"
```

This attribute informs LLVM that the code should be generated for an x86_64 architecture, a generic PC vendor, the Linux operating system, and the GNU ABI.

15.1.4 Practical Applications

1. **Cross-Compilation**

   Target triples are essential for cross-compilation, where code is generated for a platform different from the one on which the compiler is running. For example, compiling for ARM on an x86_64 machine:

   ```
   target triple = "arm-unknown-linux-gnueabi"
   ```

2. **Platform-Specific Optimizations**

   The target triple enables LLVM to apply platform-specific optimizations. For example, targeting `x86_64-pc-windows-msvc` allows LLVM to use Windows-specific calling conventions and optimizations.

3. **ABI Compatibility**

   The target triple ensures that the generated code adheres to the correct ABI, enabling interoperability with other binaries and libraries on the target platform.

15.1.5 Examples

1. **Targeting x86_64 Linux**

```
target triple = "x86_64-pc-linux-gnu"
```

2. **Targeting ARM macOS**

```
target triple = "arm64-apple-darwin20.1.0"
```

3. **Targeting RISC-V Embedded Systems**

```
target triple = "riscv32-unknown-elf"
```

4. **Targeting Windows x86_64**

```
target triple = "x86_64-pc-windows-msvc"
```

15.1.6 Advanced Topics

1. **Overriding the Target Triple**

   The target triple can be overridden at compile time using the `-mtriple` flag with `clang` or `llc`. For example:

```
clang -mtriple=arm-unknown-linux-gnueabi -c input.c -o output.o
```

2. **Target-Specific Intrinsics**

   LLVM provides target-specific intrinsics that can be used in IR to leverage architecture-specific features. For example, x86_64 intrinsics for SIMD instructions:

```
%result = call <4 x float> @llvm.x86.sse.add.ss(<4 x float> %a, <4 x
↪  float> %b)
```

3. **Target-Specific Metadata**

   LLVM allows target-specific metadata to be attached to IR constructs, enabling fine-grained control over code generation. For example:

```
!0 = !{!"target-features", !"+avx2"}
```

15.1.7 Conclusion

The target triple is a fundamental mechanism for specifying the target platform in LLVM IR. By understanding its syntax, semantics, and practical applications, advanced users can effectively generate code for a wide range of architectures and operating systems. Whether targeting x86_64 Linux, ARM macOS, or RISC-V embedded systems, the target triple ensures that LLVM generates optimized and ABI-compliant code.

This section has provided a comprehensive exploration of target triples, equipping users with the knowledge to specify and customize target-specific code generation in their LLVM-based projects.

15.2 Architecture-Specific Instructions

Target-specific code generation in LLVM involves leveraging architecture-specific instructions to optimize performance and functionality. These instructions are tailored to the capabilities of the target hardware, such as specialized registers, SIMD (Single Instruction, Multiple Data) operations, and hardware-specific features. This section provides a detailed exploration of **architecture-specific instructions** in LLVM IR, focusing on their representation, usage, and practical applications for advanced users.

15.2.1 Overview of Architecture-Specific Instructions

Architecture-specific instructions are low-level operations that directly map to the capabilities of the target hardware. They are used to:

1. **Optimize performance**: Exploit hardware features like SIMD, vector processing, and specialized instructions.
2. **Enable functionality**: Access hardware-specific features, such as cryptographic extensions or atomic operations.
3. **Reduce code size**: Replace sequences of generic instructions with a single specialized instruction.

In LLVM IR, architecture-specific instructions are represented using **intrinsics** or **inline assembly**.

15.2.2 Key Concepts

1. **Intrinsics**

Intrinsics are built-in functions that map directly to target-specific instructions. They provide a portable way to access hardware features without writing platform-specific code.

2. **Inline Assembly**

   Inline assembly allows developers to embed target-specific assembly code directly in LLVM IR. This provides fine-grained control over code generation but sacrifices portability.

3. **Target Features**

   Target features are attributes that describe the capabilities of the target hardware, such as support for SIMD extensions or specific instruction sets.

4. **Instruction Selection**

   LLVM's instruction selection process maps generic IR instructions to target-specific machine instructions during code generation.

15.2.3 Architecture-Specific Intrinsics

LLVM provides a wide range of intrinsics for accessing architecture-specific instructions. These intrinsics are defined in the `llvm` namespace and are prefixed with the target architecture.

1. **x86/x86_64 Intrinsics**

   x86 and x86_64 architectures provide intrinsics for SIMD instructions (e.g., SSE, AVX), cryptographic extensions, and more.

   Example:

```
%result = call <4 x float> @llvm.x86.sse.add.ss(<4 x float> %a, <4 x
↪  float> %b)
```

This intrinsic performs a scalar addition of the first elements of two 128-bit vectors using the SSE instruction set.

2. **ARM/AArch64 Intrinsics**

   ARM and AArch64 architectures provide intrinsics for NEON (SIMD) instructions, cryptographic extensions, and more.

   Example:

```
%result = call <4 x i32> @llvm.arm.neon.vadd.v4i32(<4 x i32> %a, <4 x
↪  i32> %b)
```

   This intrinsic performs a vector addition of two 128-bit vectors using the NEON instruction set.

3. **RISC-V Intrinsics**

   RISC-V architectures provide intrinsics for vector extensions (e.g., RVV) and other hardware features.

   Example:

```
%result = call <4 x i32> @llvm.riscv.vadd.v4i32(<4 x i32> %a, <4 x
↪  i32> %b)
```

   This intrinsic performs a vector addition of two 128-bit vectors using the RISC-V vector extension.

15.2.4 Inline Assembly

Inline assembly allows developers to embed target-specific assembly code directly in LLVM IR. It is represented using the `asm` keyword.

1. **Syntax**

```
<result> = call <return_type> asm "<assembly code>",
↪  "<constraints>"(<operands>)
```

   - **<assembly code>**: The target-specific assembly code.
   - **<constraints>**: Constraints on the operands and result.
   - **<operands>**: The input operands.

2. **Example**

   Consider the following x86_64 inline assembly to add two integers:

```
%result = call i32 asm "addl $1, $0", "=r,r"(i32 %a, i32 %b)
```

   This inline assembly adds the values of `%a` and `%b` using the `addl` instruction and stores the result in `%result`.

15.2.5 Target Features

Target features are attributes that describe the capabilities of the target hardware. They are specified using the `target-features` attribute.

1. **Syntax**

```
!0 = !{!"target-features", !"+<feature>"}
```

- **<feature>**: The target feature to enable (e.g., `+avx2`, `+neon`).

2. **Example**

   To enable AVX2 instructions on x86_64:

```
!0 = !{!"target-features", !"+avx2"}
```

15.2.6 Practical Applications

1. **SIMD Optimization**

   Architecture-specific intrinsics enable SIMD optimizations, such as vectorized arithmetic operations, memory accesses, and reductions.

   Example:

```
%result = call <4 x float> @llvm.x86.avx.add.ps.256(<8 x float> %a,
↪  <8 x float> %b)
```

   This intrinsic performs a vector addition of two 256-bit vectors using the AVX instruction set.

2. **Cryptographic Extensions**

   Architecture-specific intrinsics provide access to cryptographic instructions, such as AES encryption and SHA hashing.

   Example:

```
%result = call <2 x i64> @llvm.x86.aesni.aesenc(<2 x i64> %a, <2 x
↪  i64> %b)
```

This intrinsic performs a single round of AES encryption using the AES-NI instruction set.

3. **Atomic Operations**

   Architecture-specific intrinsics enable atomic operations, such as compare-and-swap and atomic loads/stores.

   Example:

```
%result = call i32 @llvm.x86.atomic.load.add.i32(i32* %ptr, i32
↪  %value)
```

This intrinsic performs an atomic addition using the x86 instruction set.

15.2.7 Advanced Topics

1. **Custom Intrinsics**

   Advanced users can define custom intrinsics to expose hardware-specific features not covered by LLVM's built-in intrinsics.

2. **Instruction Selection and Lowering**

   LLVM's instruction selection process maps generic IR instructions to target-specific machine instructions. Advanced users can customize this process using TableGen and custom backends.

3. **Target-Specific Metadata**

LLVM allows target-specific metadata to be attached to IR constructs, enabling fine-grained control over code generation.

15.2.8 Conclusion

Architecture-specific instructions are a powerful tool for optimizing performance and enabling hardware-specific functionality in LLVM IR. By leveraging intrinsics, inline assembly, and target features, advanced users can generate highly optimized and platform-specific code. Whether targeting x86, ARM, RISC-V, or other architectures, LLVM provides the flexibility and control needed to exploit the full capabilities of the target hardware.
This section has provided a comprehensive exploration of architecture-specific instructions, equipping users with the knowledge to implement and optimize target-specific code generation in their LLVM-based projects.

15.3 Linking & Interfacing with C/C++ (`declare i32 @printf(i8*, ...)`)

Linking and interfacing with C/C++ code is a common requirement when working with LLVM IR, especially for advanced users who need to integrate low-level optimizations with high-level language constructs. This section provides a detailed exploration of **linking and interfacing with C/C++** in LLVM IR, focusing on how to declare and use external functions, handle variadic arguments, and manage symbol resolution.

15.3.1 Overview of Linking & Interfacing with C/C++

LLVM IR is designed to interoperate seamlessly with C/C++ code, enabling developers to:

1. **Call external functions**: Use functions defined in C/C++ libraries, such as the C standard library.
2. **Export functions**: Make LLVM IR functions available to C/C++ code.
3. **Handle variadic arguments**: Work with functions that accept a variable number of arguments, such as `printf`.
4. **Manage symbol resolution**: Control how symbols are linked and resolved during compilation.

This section focuses on declaring and using external functions, with a particular emphasis on variadic functions like `printf`.

15.3.2 Key Concepts

1. **External Function Declarations**

External functions are declared in LLVM IR using the `declare` keyword. This informs the compiler that the function is defined externally and will be resolved during linking.

2. **Variadic Functions**

   Variadic functions accept a variable number of arguments. In LLVM IR, variadic functions are declared with an ellipsis (`...`) to indicate the variable argument list.

3. **Calling Conventions**

   Calling conventions define how function arguments are passed and how the stack is managed. LLVM supports multiple calling conventions, such as `cdecl`, `fastcc`, and `tailcc`.

4. **Symbol Resolution**

   Symbol resolution ensures that external functions are correctly linked to their definitions in libraries or other modules.

15.3.3 Declaring External Functions

External functions are declared in LLVM IR using the following syntax:

```
declare <return_type> @<function_name>(<argument_types>, ...)
```

- **<return_type>**: The return type of the function.
- **<function_name>**: The name of the function.
- **<argument_types>**: The types of the function's arguments.
- `...`: Indicates a variadic function (optional).

Example:

```
declare i32 @printf(i8*, ...)
```

This declares the `printf` function, which returns an `i32` and accepts a variable number of arguments.

15.3.4 Calling External Functions

External functions are called using the `call` instruction. For variadic functions, additional arguments are passed after the fixed arguments.

1. **Syntax**

   ```
   <result> = call <return_type> @<function_name>(<arguments>, ...)
   ```

   - **<result>**: The result of the function call (optional).
   - **<arguments>**: The arguments to pass to the function.

2. **Example**

   Consider the following C code:

   ```
   #include <stdio.h>
   int main() {
       printf("Hello, World!\n");
       return 0;
   }
   ```

   The corresponding LLVM IR is:

```
@.str = private unnamed_addr constant [14 x i8] c"Hello, World!\00",
↪  align 1

declare i32 @printf(i8*, ...)

define i32 @main() {
  %1 = getelementptr inbounds [14 x i8], [14 x i8]* @.str, i32 0, i32
  ↪  0
  %2 = call i32 (i8*, ...) @printf(i8* %1)
  ret i32 0
}
```

- **`@.str`**: A global string constant containing the message.
- **`declare i32 @printf(i8\*, ...)`**: Declares the `printf` function.
- **`call i32 (i8\*, ...) @printf(i8\* %1)`**: Calls `printf` with the string as an argument.

15.3.5 Handling Variadic Arguments

Variadic functions like `printf` require special handling in LLVM IR. The `va_list` type and related intrinsics are used to manage variable argument lists.

1. **`va_list` Type**

   The `va_list` type represents a variable argument list. It is typically implemented as a pointer to a structure that holds the arguments.

2. **Variadic Intrinsics**

   LLVM provides intrinsics for working with variadic arguments, such as:

- **`llvm.va_start`**: Initializes a `va_list`.
- **`llvm.va_copy`**: Copies a `va_list`.
- **`llvm.va_end`**: Cleans up a `va_list`.

Example:

```
declare void @llvm.va_start(i8*)
declare void @llvm.va_copy(i8*, i8*)
declare void @llvm.va_end(i8*)
```

3. **Example**

Consider the following C code:

```
#include <stdarg.h>
#include <stdio.h>

void print_numbers(int count, ...) {
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; i++) {
        int num = va_arg(args, int);
        printf("%d\n", num);
    }
    va_end(args);
}
```

The corresponding LLVM IR is:

```
declare void @llvm.va_start(i8*)
declare i32 @printf(i8*, ...)

define void @print_numbers(i32 %count, ...) {
  %args = alloca i8*, align 8
  %args1 = bitcast i8** %args to i8*
  call void @llvm.va_start(i8* %args1)
  br label %loop

loop:
  %i = phi i32 [ 0, %entry ], [ %i.next, %loop.body ]
  %cmp = icmp slt i32 %i, %count
  br i1 %cmp, label %loop.body, label %exit

loop.body:
  %num = va_arg i8** %args, i32
  %fmt = getelementptr inbounds [4 x i8], [4 x i8]* @.str, i32 0, i32
  ↪  0
  call i32 (i8*, ...) @printf(i8* %fmt, i32 %num)
  %i.next = add i32 %i, 1
  br label %loop

exit:
  call void @llvm.va_end(i8* %args1)
  ret void
}
```

15.3.6 Exporting Functions to C/C++

LLVM IR functions can be exported to C/C++ by ensuring they have the correct linkage and calling convention.

1. **Syntax**

```
define <linkage> <return_type> @<function_name>(<argument_types>) {
  ; Function body
}
```

   - **<linkage>**: The linkage type (e.g., `external`, `internal`).

2. **Example**

   To export an LLVM IR function to C/C++:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

   This function can be called from C/C++ code as follows:

```
extern int add(int a, int b);
```

15.3.7 Advanced Topics

1. **Symbol Resolution**

   LLVM provides mechanisms to control symbol resolution, such as `weak` linkage and `alias` declarations.

2. **Calling Conventions**

   Advanced users can specify custom calling conventions using the `cc <calling_convention>` attribute.

3. **Interfacing with C++ Name Mangling**

   C++ functions must be declared with their mangled names in LLVM IR. Tools like `clang` can generate the correct mangled names.

15.3.8 Conclusion

Linking and interfacing with C/C++ code is a fundamental aspect of working with LLVM IR. By understanding how to declare and use external functions, handle variadic arguments, and manage symbol resolution, advanced users can seamlessly integrate LLVM IR with high-level language constructs. Whether calling `printf`, exporting functions, or working with variadic argument lists, LLVM provides the flexibility and control needed to build robust and efficient systems.

This section has provided a comprehensive exploration of linking and interfacing with C/C++ in LLVM IR, equipping users with the knowledge to implement and optimize cross-language interoperability in their projects.

Chapter 16

LLVM Intrinsics & Built-in Functions

16.1 Math Operations (`llvm.sqrt.f32`)

LLVM provides a rich set of **intrinsics**—built-in functions that map directly to low-level operations or hardware instructions. These intrinsics enable advanced users to perform complex operations, such as mathematical computations, with high efficiency and precision. This section focuses on **math operations** in LLVM IR, with a detailed exploration of the `llvm.sqrt.f32` intrinsic and its role in performing square root calculations.

16.1.1 Overview of Math Intrinsics

Math intrinsics in LLVM IR are designed to provide efficient implementations of common mathematical operations, such as:

1. **Square roots**: `llvm.sqrt.f32`, `llvm.sqrt.f64`.
2. **Trigonometric functions**: `llvm.sin.f32`, `llvm.cos.f32`.
3. **Exponential and logarithmic functions**: `llvm.exp.f32`, `llvm.log.f32`.

4. **Minimum and maximum**: `llvm.minnum.f32`, `llvm.maxnum.f32`.

These intrinsics are optimized for performance and accuracy, often leveraging hardware support for mathematical operations.

16.1.2 Key Concepts

1. **Intrinsics**

   Intrinsics are built-in functions that map directly to low-level operations or hardware instructions. They provide a portable way to access advanced functionality without writing platform-specific code.

2. **Floating-Point Precision**

   Math intrinsics support various floating-point precisions, such as:

   - **`f32`**: 32-bit floating-point (single precision).
   - **`f64`**: 64-bit floating-point (double precision).

3. **Fast-Math Flags**

   Fast-math flags enable aggressive optimizations for floating-point operations, such as assuming no NaNs or infinities.

4. **Hardware Support**

   Many math intrinsics are mapped directly to hardware instructions, such as SSE or AVX on x86 architectures.

16.1.3 The `llvm.sqrt.f32` Intrinsic

The `llvm.sqrt.f32` intrinsic computes the square root of a 32-bit floating-point value. Its syntax is as follows:

```
declare float @llvm.sqrt.f32(float %val)
```

- **`%val`**: The input value.
- **Return value**: The square root of `%val`.

1. **Behavior**

   - **Input**: A 32-bit floating-point value.
   - **Output**: The square root of the input value.
   - **Precision**: Follows IEEE 754 rules for floating-point arithmetic.

2. **Example**

   Consider the following C code:

   ```
   #include <math.h>
   float sqrt_value(float x) {
       return sqrtf(x);
   }
   ```

   The corresponding LLVM IR using the `llvm.sqrt.f32` intrinsic is:

   ```
   declare float @llvm.sqrt.f32(float)

   define float @sqrt_value(float %x) {
     %result = call float @llvm.sqrt.f32(float %x)
     ret float %result
   }
   ```

16.1.4 Other Math Intrinsics

1. **Trigonometric Functions**

   - **`llvm.sin.f32`**: Computes the sine of a 32-bit floating-point value.
   - **`llvm.cos.f32`**: Computes the cosine of a 32-bit floating-point value.

   Example:

```
declare float @llvm.sin.f32(float)
declare float @llvm.cos.f32(float)

define float @compute_sin(float %x) {
  %result = call float @llvm.sin.f32(float %x)
  ret float %result
}
```

2. **Exponential and Logarithmic Functions**

   - **`llvm.exp.f32`**: Computes the exponential of a 32-bit floating-point value.
   - **`llvm.log.f32`**: Computes the natural logarithm of a 32-bit floating-point value.

   Example:

```
declare float @llvm.exp.f32(float)
declare float @llvm.log.f32(float)

define float @compute_exp(float %x) {
  %result = call float @llvm.exp.f32(float %x)
  ret float %result
}
```

3. **Minimum and Maximum**

   - **`llvm.minnum.f32`**: Computes the minimum of two 32-bit floating-point values.
   - **`llvm.maxnum.f32`**: Computes the maximum of two 32-bit floating-point values.

   Example:

```
declare float @llvm.minnum.f32(float, float)
declare float @llvm.maxnum.f32(float, float)

define float @compute_min(float %a, float %b) {
  %result = call float @llvm.minnum.f32(float %a, float %b)
  ret float %result
}
```

16.1.5 Fast-Math Flags

Fast-math flags enable aggressive optimizations for floating-point operations by relaxing certain constraints, such as:

- **`nnan`**: No NaNs.
- **`ninf`**: No infinities.
- **`nsz`**: No signed zeros.
- **`arcp`**: Allow reciprocal operations.

Example:

```
declare float @llvm.sqrt.f32(float)

define float @sqrt_value_fast(float %x) {
  %result = call fast float @llvm.sqrt.f32(float %x)
  ret float %result
}
```

16.1.6 Practical Applications

1. **Scientific Computing**

   Math intrinsics are widely used in scientific computing for tasks such as numerical simulations, signal processing, and machine learning.

2. **Graphics Programming**

   Math intrinsics enable efficient implementations of graphics algorithms, such as vector normalization and lighting calculations.

3. **Performance Optimization**

   By leveraging hardware support for math operations, intrinsics can significantly improve the performance of computationally intensive applications.

16.1.7 Advanced Topics

1. **Vectorized Math Intrinsics**

   LLVM supports vectorized math intrinsics for SIMD operations. For example:

   ```
   declare <4 x float> @llvm.sqrt.v4f32(<4 x float>)
   ```

2. **Custom Math Intrinsics**

   Advanced users can define custom math intrinsics to expose hardware-specific features not covered by LLVM's built-in intrinsics.

3. **Precision Control**

   LLVM provides mechanisms to control the precision of math operations, such as using `fptrunc` and `fpext` to convert between floating-point precisions.

16.1.8 Conclusion

Math intrinsics in LLVM IR provide a powerful mechanism for performing efficient and accurate mathematical computations. By understanding their syntax, semantics, and use cases, advanced users can leverage these intrinsics to optimize performance and enable advanced functionality in their programs. Whether computing square roots, trigonometric functions, or exponential operations, LLVM's math intrinsics offer the flexibility and control needed to build high-performance systems.

This section has provided a comprehensive exploration of math operations in LLVM IR, equipping users with the knowledge to implement and optimize mathematical computations in their projects.

16.2 Memory Intrinsics (`llvm.memcpy.p0i8.p0i8.i32`)

Memory intrinsics in LLVM IR provide efficient and portable ways to perform common memory operations, such as copying, moving, and setting memory regions. These intrinsics are essential for optimizing performance and ensuring correctness in low-level code. This section focuses on **memory intrinsics**, with a detailed exploration of the `llvm.memcpy.p0i8.p0i8.i32` intrinsic and its role in memory operations.

16.2.1 Overview of Memory Intrinsics

Memory intrinsics are built-in functions that map directly to low-level memory operations. They are designed to:

1. **Optimize performance**: Leverage hardware-specific instructions for memory operations.
2. **Ensure correctness**: Handle alignment, overlapping regions, and other edge cases.
3. **Simplify code**: Provide a portable and high-level interface for memory operations.

Common memory intrinsics include:

- **`llvm.memcpy`**: Copy a memory region.
- **`llvm.memmove`**: Move a memory region (handles overlapping regions).
- **`llvm.memset`**: Set a memory region to a specific value.

16.2.2 Key Concepts

1. **Memory Operations**

Memory operations involve reading from or writing to a region of memory. These operations must handle alignment, overlapping regions, and other low-level details.

2. **Alignment**

   Alignment specifies the byte alignment of memory addresses. Proper alignment improves performance and ensures correctness on some architectures.

3. **Volatile Memory**

   Volatile memory operations are not optimized away and are always executed, ensuring that memory-mapped I/O and other side effects are preserved.

4. **Overlapping Regions**

   Memory operations must handle overlapping regions correctly. For example, `llvm.memmove` ensures correct behavior when source and destination regions overlap.

16.2.3 The `llvm.memcpy.p0i8.p0i8.i32` Intrinsic

The `llvm.memcpy.p0i8.p0i8.i32` intrinsic copies a memory region from a source address to a destination address. Its syntax is as follows:

```
declare void @llvm.memcpy.p0i8.p0i8.i32(i8* <dest>, i8* <src>, i32 <len>,
↪  i1 <is_volatile>)
```

- **<dest>**: The destination memory address.
- **<src>**: The source memory address.
- **<len>**: The number of bytes to copy.
- **<is_volatile>**: Indicates whether the operation is volatile.

1. **Behavior**

   - **Input**: A source address, a destination address, a length, and a volatile flag.
   - **Output**: The memory region is copied from the source to the destination.
   - **Alignment**: The intrinsic assumes proper alignment of the source and destination addresses.

2. **Example**

   Consider the following C code:

```c
#include <string.h>
void copy_memory(void* dest, void* src, size_t len) {
    memcpy(dest, src, len);
}
```

   The corresponding LLVM IR using the `llvm.memcpy.p0i8.p0i8.i32` intrinsic is:

```
declare void @llvm.memcpy.p0i8.p0i8.i32(i8*, i8*, i32, i1)

define void @copy_memory(i8* %dest, i8* %src, i32 %len) {
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %dest, i8* %src, i32 %len,
  ↪  i1 false)
  ret void
}
```

16.2.4 Other Memory Intrinsics

1. **`llvm.memmove`**

The `llvm.memmove` intrinsic moves a memory region, handling overlapping regions correctly.

Example:

```
declare void @llvm.memmove.p0i8.p0i8.i32(i8*, i8*, i32, i1)

define void @move_memory(i8* %dest, i8* %src, i32 %len) {
  call void @llvm.memmove.p0i8.p0i8.i32(i8* %dest, i8* %src, i32 %len,
  ↪  i1 false)
  ret void
}
```

2. **`llvm.memset`**

   The `llvm.memset` intrinsic sets a memory region to a specific value.

   Example:

```
declare void @llvm.memset.p0i8.i32(i8*, i8, i32, i1)

define void @set_memory(i8* %dest, i8 %value, i32 %len) {
  call void @llvm.memset.p0i8.i32(i8* %dest, i8 %value, i32 %len, i1
  ↪  false)
  ret void
}
```

16.2.5 Practical Applications

1. **Data Structures**

   Memory intrinsics are used to implement data structures, such as arrays, strings, and buffers.

2. **Performance Optimization**

   By leveraging hardware support for memory operations, intrinsics can significantly improve the performance of memory-intensive applications.

3. **Low-Level Programming**

   Memory intrinsics are essential for low-level programming tasks, such as implementing memory allocators or interacting with hardware.

16.2.6 Advanced Topics

1. **Alignment**

   Memory intrinsics assume proper alignment of memory addresses. Advanced users can specify alignment using the `align` attribute.

   Example:

```
call void @llvm.memcpy.p0i8.p0i8.i32(i8* align 4 %dest, i8* align 4
↪  %src, i32 %len, i1 false)
```

2. **Volatile Memory**

   Volatile memory operations are not optimized away and are always executed. This is useful for memory-mapped I/O and other side effects.

   Example:

```
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %dest, i8* %src, i32 %len, i1
↪  true)
```

3. **Vectorized Memory Operations**

   LLVM supports vectorized memory operations for SIMD architectures. For example:

```
declare void @llvm.memcpy.p0i8.p0i8.v4i32(i8*, i8*, <4 x i32>, i1)
```

16.2.7 Conclusion

Memory intrinsics in LLVM IR provide a powerful mechanism for performing efficient and correct memory operations. By understanding their syntax, semantics, and use cases, advanced users can leverage these intrinsics to optimize performance and ensure correctness in their programs. Whether copying, moving, or setting memory regions, LLVM's memory intrinsics offer the flexibility and control needed to build high-performance systems.
This section has provided a comprehensive exploration of memory intrinsics, equipping users with the knowledge to implement and optimize memory operations in their projects.

16.3 Synchronization & Atomic (`llvm.fence`)

Synchronization and atomic operations are critical for writing correct and efficient concurrent programs. LLVM provides a set of intrinsics to support these operations, ensuring that memory accesses are properly ordered and that atomicity is maintained across threads. This section focuses on **synchronization and atomic intrinsics**, with a detailed exploration of the `llvm.fence` intrinsic and its role in memory ordering and synchronization.

16.3.1 Overview of Synchronization & Atomic Intrinsics

Synchronization and atomic intrinsics in LLVM IR are designed to:

1. **Enforce memory ordering**: Ensure that memory operations are executed in the correct order across threads.
2. **Provide atomicity**: Guarantee that certain operations are performed indivisibly.
3. **Support lock-free algorithms**: Enable the implementation of efficient concurrent data structures.

Key intrinsics include:

- **`llvm.fence`**: Enforces memory ordering constraints.
- **`llvm.atomic.\*`**: Provides atomic read-modify-write operations.
- **`llvm.memory.barrier`**: Ensures memory consistency across threads.

16.3.2 Key Concepts

1. **Memory Ordering**

Memory ordering defines the visibility of memory operations across threads. Common memory ordering constraints include:

- **Sequentially Consistent (`seq_cst`)**: The strongest ordering, ensuring a total order of operations.
- **Acquire-Release (`acq_rel`)**: Ensures that operations before a release are visible after an acquire.
- **Relaxed (`monotonic`)**: No ordering guarantees.

2. **Atomicity**

   Atomic operations are indivisible, meaning they are executed as a single, uninterruptible unit. This is essential for implementing lock-free algorithms and ensuring correctness in concurrent programs.

3. **Fences**

   A memory fence (or barrier) enforces ordering constraints on memory operations, ensuring that certain operations are not reordered across the fence.

4. **Volatile Memory**

   Volatile memory operations are not optimized away and are always executed, ensuring that memory-mapped I/O and other side effects are preserved.

16.3.3 The `llvm.fence` Intrinsic

The `llvm.fence` intrinsic enforces memory ordering constraints. Its syntax is as follows:

```
declare void @llvm.fence(i32 <ordering>, i1 <is_volatile>)
```

- **<ordering>**: The memory ordering constraint (e.g., `seq_cst`, `acq_rel`, `monotonic`).
- **<is_volatile>**: Indicates whether the operation is volatile.

1. **Behavior**

   - **Input**: A memory ordering constraint and a volatile flag.
   - **Output**: A memory fence that enforces the specified ordering constraints.
   - **Use Cases**: Ensuring visibility of memory operations across threads, implementing synchronization primitives.

2. **Example**

   Consider the following C code using a memory fence:

```c
#include <stdatomic.h>
void synchronize() {
    atomic_thread_fence(memory_order_seq_cst);
}
```

   The corresponding LLVM IR using the `llvm.fence` intrinsic is:

```llvm
declare void @llvm.fence(i32, i1)

define void @synchronize() {
  call void @llvm.fence(i32 5, i1 false)  ; 5 corresponds to seq_cst
  ret void
}
```

16.3.4 Other Synchronization & Atomic Intrinsics

1. **`llvm.atomic.*` Intrinsics**16.3.4.1 llvm.atomic.* Intrinsics

   The `llvm.atomic.*` intrinsics provide atomic read-modify-write operations, such as compare-and-swap and atomic addition.

   Example:

```
declare i32 @llvm.atomic.load.add.i32.p0i32(i32*, i32)

define i32 @atomic_add(i32* %ptr, i32 %value) {
  %result = call i32 @llvm.atomic.load.add.i32.p0i32(i32* %ptr, i32
  ↪  %value)
  ret i32 %result
}
```

2. **`llvm.memory.barrier`**

   The `llvm.memory.barrier` intrinsic ensures memory consistency across threads.

   Example:

```
declare void @llvm.memory.barrier(i32, i32, i32, i32, i1)

define void @barrier() {
  call void @llvm.memory.barrier(i32 1, i32 2, i32 3, i32 4, i1
  ↪  false)
  ret void
}
```

16.3.5 Practical Applications

1. **Lock-Free Data Structures**

   Synchronization and atomic intrinsics are essential for implementing lock-free data structures, such as queues, stacks, and hash tables.

2. **Synchronization Primitives**

   These intrinsics are used to implement synchronization primitives, such as mutexes, condition variables, and barriers.

3. **Performance Optimization**

   By leveraging hardware support for atomic operations, intrinsics can significantly improve the performance of concurrent programs.

16.3.6 Advanced Topics

1. **Memory Ordering Trade-offs**

   Different memory ordering constraints offer trade-offs between performance and correctness. Advanced users must choose the appropriate ordering for their use case.

2. **Volatile Memory**

   Volatile memory operations are not optimized away and are always executed. This is useful for memory-mapped I/O and other side effects.

   Example:

```
call void @llvm.fence(i32 5, i1 true)  ; Volatile seq_cst fence
```

3. **Custom Synchronization**

   Advanced users can define custom synchronization mechanisms using a combination of fences, atomic operations, and memory barriers.

16.3.7 Conclusion

Synchronization and atomic intrinsics in LLVM IR provide a powerful mechanism for writing correct and efficient concurrent programs. By understanding their syntax, semantics, and use cases, advanced users can leverage these intrinsics to implement lock-free data structures, synchronization primitives, and high-performance concurrent algorithms. Whether enforcing memory ordering, ensuring atomicity, or implementing custom synchronization, LLVM's synchronization and atomic intrinsics offer the flexibility and control needed to build robust and scalable systems.

This section has provided a comprehensive exploration of synchronization and atomic intrinsics, equipping users with the knowledge to implement and optimize concurrent programs in their projects.

Chapter 17

Debugging & Profiling LLVM IR

17.1 Debugging with `llvm-dis`, `llvm-dwarfdump`

Debugging is a critical aspect of software development, and LLVM provides powerful tools to analyze and debug intermediate representation (IR) code. This section focuses on **debugging LLVM IR** using two essential tools: `llvm-dis` and `llvm-dwarfdump`. These tools enable advanced users to inspect and understand the structure, metadata, and debug information embedded in LLVM IR.

17.1.1 Overview of Debugging Tools

LLVM provides a suite of tools for debugging and analyzing IR code:

1. **`llvm-dis`**: Converts LLVM bitcode (`.bc`) to human-readable LLVM IR (`.ll`).

2. **`llvm-dwarfdump`**: Extracts and displays DWARF debug information from LLVM IR or object files.

These tools are essential for understanding the structure of IR, inspecting debug information, and diagnosing issues in the compilation process.

17.1.2 Key Concepts

1. **LLVM Bitcode vs. LLVM IR**

   - **LLVM Bitcode (`.bc`)**: A binary representation of LLVM IR, optimized for storage and processing.
   - **LLVM IR (`.ll`)**: A human-readable text representation of LLVM IR.

2. **DWARF Debug Information**

   DWARF is a widely used format for encoding debug information in executable files. It includes details such as source file locations, variable names, and type information.

3. **Debug Metadata**

   LLVM IR embeds debug metadata to associate IR constructs with their corresponding source-level information. This metadata is used by debugging tools like GDB and LLDB.

17.1.3 The `llvm-dis` Tool

The `llvm-dis` tool converts LLVM bitcode (`.bc`) to human-readable LLVM IR (`.ll`). This is useful for inspecting the IR and understanding its structure.

1. **Syntax**

```
llvm-dis [options] <input.bc> -o <output.ll>
```

- **<input.bc>**: The input LLVM bitcode file.
- **<output.ll>**: The output LLVM IR file (optional).

2. **Common Options**

   - **-o <output.ll>**: Specifies the output file.
   - **-f**: Overwrites the output file if it exists.
   - **-help**: Displays help information.

3. **Example**

   Consider an LLVM bitcode file `example.bc`. To convert it to human-readable IR:

   ```
   llvm-dis example.bc -o example.ll
   ```

   This generates an `example.ll` file containing the LLVM IR in text format.

4. **Use Cases**

   - **Inspecting IR**: View the structure and content of LLVM IR.
   - **Debugging**: Identify issues in the IR, such as incorrect optimizations or missing metadata.
   - **Learning**: Study the IR generated by different frontends or optimization passes.

17.1.4 The `llvm-dwarfdump` Tool

The `llvm-dwarfdump` tool extracts and displays DWARF debug information from LLVM IR or object files. This is useful for inspecting debug metadata and understanding how IR constructs map to source-level information.

1. **Syntax**

```
llvm-dwarfdump [options] <input>
```

   - **<input>**: The input file (LLVM IR, object file, or executable).

2. **Common Options**

   - **-debug-info**: Displays the `.debug_info` section (source-level information).
   - **-debug-line**: Displays the `.debug_line` section (line number information).
   - **-debug-loc**: Displays the `.debug_loc` section (location information).
   - **-debug-abbrev**: Displays the `.debug_abbrev` section (abbreviation tables).
   - **-help**: Displays help information.

3. **Example**

   Consider an LLVM IR file `example.ll` with embedded DWARF debug information. To inspect the debug information:

```
llvm-dwarfdump example.ll
```

   This displays the DWARF debug information, including source file locations, variable names, and type information.

4. **Use Cases**

   - **Debugging**: Inspect source-level information to diagnose issues in the IR.
   - **Verification**: Ensure that debug metadata is correctly embedded in the IR.
   - **Optimization**: Analyze how optimizations affect debug information.

17.1.5 Practical Applications

1. **Debugging Optimized Code**

   Optimized code can be challenging to debug due to transformations like inlining and loop unrolling. `llvm-dwarfdump` helps inspect debug metadata to understand how optimized IR maps to source code.

2. **Verifying Debug Information**

   Debug information must be accurate and complete for effective debugging. `llvm-dwarfdump` verifies that debug metadata is correctly embedded in the IR.

3. **Learning and Analysis**

   By inspecting IR and debug information, developers can learn how frontends and optimization passes generate and transform code.

17.1.6 Advanced Topics

1. **Custom Debug Metadata**

   Advanced users can define custom debug metadata to provide additional information for debugging and analysis.

2. **Debugging with LLDB**

   LLDB uses DWARF debug information to provide source-level debugging. `llvm-dwarfdump` helps ensure that the debug information is compatible with LLDB.

3. **Debugging JIT Code**

   Just-In-Time (JIT) compilation can complicate debugging. `llvm-dwarfdump` helps inspect debug information generated by JIT compilers.

17.1.7 Conclusion

Debugging LLVM IR is a critical skill for advanced users, and tools like `llvm-dis` and `llvm-dwarfdump` provide powerful capabilities for inspecting and analyzing IR code. By understanding how to use these tools, developers can diagnose issues, verify debug information, and gain deeper insights into the structure and behavior of their programs. This section has provided a comprehensive exploration of debugging with `llvm-dis` and `llvm-dwarfdump`, equipping users with the knowledge to effectively debug and analyze LLVM IR in their projects.

17.2 Profiling with `perf` and LLVM Sanitizers (`asan`, `msan`)

Profiling and runtime analysis are essential for optimizing performance and ensuring the correctness of programs. LLVM provides powerful tools for profiling and runtime error detection, including integration with the `perf` profiler and the use of sanitizers like AddressSanitizer (`asan`) and MemorySanitizer (`msan`). This section provides a detailed exploration of these tools, focusing on their usage, capabilities, and practical applications for advanced users.

17.2.1 Overview of Profiling and Sanitizers

Profiling and sanitizers serve the following purposes:

1. **Performance profiling**: Identify performance bottlenecks and optimize code.
2. **Runtime error detection**: Detect memory errors, data races, and undefined behavior.
3. **Code correctness**: Ensure that programs are free from common runtime errors.

This section focuses on two key tools:

- **`perf`**: A Linux profiling tool for performance analysis.
- **LLVM Sanitizers**: Runtime error detection tools, including AddressSanitizer (`asan`) and MemorySanitizer (`msan`).

17.2.2 Key Concepts

1. **Profiling**

   Profiling involves measuring the performance of a program to identify bottlenecks, such as slow functions or inefficient memory usage.

2. **Sanitizers**

   Sanitizers are runtime tools that detect errors such as memory corruption, use-after-free, and uninitialized memory access.

3. **AddressSanitizer (`asan`)**

   AddressSanitizer detects memory errors, including buffer overflows, use-after-free, and memory leaks.

4. **MemorySanitizer (`msan`)**

   MemorySanitizer detects uninitialized memory access, ensuring that all memory reads are from initialized locations.

5. **`perf`**

   `perf` is a Linux profiling tool that collects and analyzes performance data, such as CPU cycles, cache misses, and function call counts.

17.2.3 Profiling with `perf`

`perf` is a powerful profiling tool that provides detailed insights into program performance. It is integrated with LLVM, allowing users to profile optimized IR and machine code.

1. **Usage**

   The `perf` tool is invoked from the command line with the following syntax:

   ```
   perf <command> [options]
   ```

   Common commands include:

   - **`record`**: Collect performance data.

- **`report`**: Analyze collected data.
- **`stat`**: Display performance statistics.

2. **Example: Profiling a Program**

   To profile a program `example` using `perf`:

   ```
   perf record ./example
   ```

   This collects performance data and stores it in a file (`perf.data`).

   To analyze the collected data:

   ```
   perf report
   ```

   This displays a detailed report of performance metrics, such as function call counts and CPU usage.

3. **Advanced Features**

   - **Hardware Counters**: `perf` can access hardware performance counters to measure metrics like cache misses and branch mispredictions.
   - **Call Graphs**: `perf` can generate call graphs to visualize function call relationships and identify performance bottlenecks.
   - **Annotations**: `perf` can annotate source code with performance metrics, helping developers pinpoint inefficiencies.

17.2.4 Using LLVM Sanitizers

LLVM sanitizers are runtime tools that detect errors in programs. They are integrated with LLVM IR and can be enabled using compiler flags.

1. **AddressSanitizer (`asan`)**

   AddressSanitizer detects memory errors, such as buffer overflows, use-after-free, and memory leaks.

   Enabling AddressSanitizer

   To enable AddressSanitizer, compile the program with the `-fsanitize=address` flag:

   ```
   clang -fsanitize=address -o example example.c
   ```

   Example: Detecting a Buffer Overflow

   Consider the following C code with a buffer overflow:

   ```c
   #include <stdlib.h>
   int main() {
       int* arr = (int*)malloc(10 * sizeof(int));
       arr[10] = 42;  // Buffer overflow
       free(arr);
       return 0;
   }
   ```

   When run with AddressSanitizer, the program produces an error message:

   ```
   ==12345==ERROR: AddressSanitizer: heap-buffer-overflow on addres
   ```

2. **MemorySanitizer (`msan`)**

   MemorySanitizer detects uninitialized memory access, ensuring that all memory reads are from initialized locations.

   Enabling MemorySanitizer

To enable MemorySanitizer, compile the program with the `-fsanitize=memory` flag:

```
clang -fsanitize=memory -o example example.c
```

Example: Detecting Uninitialized Memory

Consider the following C code with uninitialized memory access:

```c
#include <stdio.h>
int main() {
    int x;
    printf("%d\n", x);  // Uninitialized memory access
    return 0;
}
```

When run with MemorySanitizer, the program produces an error message:

```
==12345==WARNING: MemorySanitizer: use-of-uninitialized-value
```

17.2.5 Practical Applications

1. **Performance Optimization**

   Profiling with `perf` helps identify performance bottlenecks, such as slow functions or inefficient memory usage.

2. **Debugging Memory Errors**

   Sanitizers like AddressSanitizer and MemorySanitizer detect memory errors that can lead to crashes or undefined behavior.

3. **Ensuring Code Correctness**

   By detecting uninitialized memory access and other runtime errors, sanitizers help ensure that programs are correct and reliable.

17.2.6 Advanced Topics

1. **Combining Profiling and Sanitizers**

   Advanced users can combine profiling and sanitizers to optimize performance while ensuring correctness. For example, use `perf` to identify bottlenecks and AddressSanitizer to detect memory errors.

2. **Custom Sanitizers**

   LLVM allows advanced users to define custom sanitizers to detect specific types of errors or enforce custom runtime checks.

3. **Integration with CI/CD**

   Profiling and sanitizers can be integrated into continuous integration/continuous deployment (CI/CD) pipelines to automatically detect performance regressions and runtime errors.

17.2.7 Conclusion

Profiling with `perf` and using LLVM sanitizers like AddressSanitizer and MemorySanitizer are essential techniques for optimizing performance and ensuring the correctness of programs. By understanding how to use these tools, advanced users can diagnose performance bottlenecks, detect runtime errors, and build robust and efficient systems.
This section has provided a comprehensive exploration of profiling and sanitizers, equipping users with the knowledge to effectively debug, profile, and optimize their LLVM-based projects.

Chapter 18

Interfacing LLVM IR with External Tools

18.1 Using `clang -emit-llvm` to Generate IR

Generating LLVM Intermediate Representation (IR) from high-level source code is a fundamental step in leveraging LLVM's optimization and code generation capabilities. The `clang` compiler, part of the LLVM project, provides a straightforward way to generate LLVM IR from C/C++ source code using the `-emit-llvm` flag. This section provides a detailed exploration of how to use `clang -emit-llvm` to generate LLVM IR, focusing on its usage, options, and practical applications for advanced users.

18.1.1 Overview of `clang -emit-llvm`

The `clang` compiler is a frontend for the LLVM project, supporting multiple programming languages, including C, C++, and Objective-C. The `-emit-llvm` flag instructs `clang` to generate LLVM IR instead of machine code. This is useful for:

1. **Analyzing IR**: Inspecting the IR to understand how high-level code is translated.

2. **Optimizing IR**: Applying LLVM's optimization passes to the generated IR.

3. **Interfacing with tools**: Using the IR as input for custom tools or further processing.

18.1.2 Key Concepts

1. **LLVM IR**

   LLVM IR is a low-level, typed, and SSA (Static Single Assignment) based representation of code. It serves as an intermediate step between high-level source code and machine code.

2. **Bitcode vs. Textual IR**

   - **Bitcode (`.bc`)**: A binary representation of LLVM IR, optimized for storage and processing.
   - **Textual IR (`.ll`)**: A human-readable text representation of LLVM IR.

3. **Compilation Pipeline**

   The `clang` compilation pipeline consists of multiple stages, including preprocessing, parsing, IR generation, optimization, and code generation. The `-emit-llvm` flag stops the pipeline after IR generation.

18.1.3 Using `clang -emit-llvm`

The `clang` compiler is invoked from the command line with the following syntax:

```
clang -emit-llvm [options] <input_file> -o <output_file>
```

- **`<input_file>`**: The input source file (e.g., `example.c`).

- **<output_file>**: The output file containing LLVM IR (e.g., `example.ll` or `example.bc`).

1. **Generating Textual IR**

   To generate human-readable textual IR, use the `-S` flag:

   ```
   clang -emit-llvm -S -o example.ll example.c
   ```

   This generates an `example.ll` file containing the LLVM IR in text format.

2. **Generating Bitcode**

   To generate binary bitcode, omit the `-S` flag:

   ```
   clang -emit-llvm -c -o example.bc example.c
   ```

   This generates an `example.bc` file containing the LLVM IR in bitcode format.

3. **Common Options**

   - **-O<level>**: Specifies the optimization level (e.g., `-O0`, `-O1`, `-O2`, `-O3`).
   - **-g**: Generates debug information.
   - **-I<path>**: Adds an include path for header files.
   - **-D<macro>**: Defines a preprocessor macro.

18.1.4 Practical Examples

1. **Generating IR for a Simple C Program**

   Consider the following C program (`example.c`):

```c
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

To generate textual IR:

```
clang -emit-llvm -S -o example.ll example.c
```

The generated `example.ll` file contains LLVM IR similar to:

```
@.str = private unnamed_addr constant [14 x i8] c"Hello, World!\00",
 ↪  align 1

declare i32 @printf(i8*, ...)

define i32 @main() {
  %1 = getelementptr inbounds [14 x i8], [14 x i8]* @.str, i32 0, i32
   ↪  0
  %2 = call i32 (i8*, ...) @printf(i8* %1)
  ret i32 0
}
```

2. **Generating IR with Debug Information**

   To generate IR with debug information, use the `-g` flag:

   ```
   clang -emit-llvm -S -g -o example.ll example.c
   ```

   The generated IR includes debug metadata, such as source file locations and variable names.

3. **Generating IR with Optimizations**

   To generate optimized IR, specify an optimization level:

   ```
   clang -emit-llvm -S -O2 -o example.ll example.c
   ```

   The generated IR is optimized, with redundant instructions removed and loops unrolled.

18.1.5 Advanced Topics

1. **Linking Multiple IR Files**

   LLVM IR files can be linked together using the `llvm-link` tool. For example:

   ```
   llvm-link file1.ll file2.ll -o combined.ll
   ```

2. **Customizing the Compilation Pipeline**

   Advanced users can customize the `clang` compilation pipeline using flags like `-Xclang` to pass options directly to the LLVM backend.

3. **Generating IR for C++ Code**

   The `clang` compiler also supports generating IR for C++ code. For example:

   ```
   clang++ -emit-llvm -S -o example.ll example.cpp
   ```

4. **Using IR with Custom Tools**

   Generated IR can be used as input for custom tools, such as static analyzers, optimizers, or code generators.

18.1.6 Conclusion

Using `clang -emit-llvm` to generate LLVM IR is a powerful technique for analyzing, optimizing, and interfacing with high-level source code. By understanding its usage, options, and practical applications, advanced users can leverage LLVM IR to build robust and efficient systems. Whether generating textual IR, bitcode, or optimized IR, `clang` provides the flexibility and control needed to work with LLVM IR effectively.

This section has provided a comprehensive exploration of using `clang -emit-llvm` to generate LLVM IR, equipping users with the knowledge to interface LLVM IR with external tools and processes in their projects.

18.2 Linking with `llvm-link`

Linking is a critical step in the compilation process, where multiple object files or intermediate representations (IR) are combined into a single executable or library. In the context of LLVM, the `llvm-link` tool is used to link multiple LLVM IR files into a single IR module. This section provides a detailed exploration of **linking with `llvm-link`**, focusing on its usage, options, and practical applications for advanced users.

18.2.1 Overview of `llvm-link`

The `llvm-link` tool is part of the LLVM project and is used to combine multiple LLVM IR files into a single IR module. This is useful for:

1. **Combining IR files**: Linking multiple IR files generated from different source files.
2. **Creating libraries**: Building IR-based libraries that can be linked with other IR modules.
3. **Optimizing linked IR**: Applying optimizations to the combined IR module.

`llvm-link` supports both textual IR (`.ll`) and bitcode (`.bc`) formats.

18.2.2 Key Concepts

1. **LLVM IR Modules**

   An LLVM IR module is a self-contained unit of IR code, typically corresponding to a single source file. Modules contain functions, global variables, and metadata.

2. **Linking Process**

   Linking combines multiple IR modules into a single module, resolving external references and ensuring consistency.

3. **Symbol Resolution**

   During linking, `llvm-link` resolves symbols (e.g., functions and global variables) across modules, ensuring that references are correctly linked to their definitions.

4. **Optimization**

   Linking provides an opportunity to apply optimizations to the combined IR module, improving performance and reducing code size.

18.2.3 Using `llvm-link`

The `llvm-link` tool is invoked from the command line with the following syntax:

```
llvm-link [options] <input_files> -o <output_file>
```

- **<input_files>**: The input IR files to be linked.
- **<output_file>**: The output file containing the combined IR module.

1. **Basic Usage**

   To link multiple IR files into a single module:

   ```
   llvm-link file1.ll file2.ll -o combined.ll
   ```

   This generates a `combined.ll` file containing the linked IR.

2. **Linking Bitcode Files**

   To link bitcode files (`.bc`):

```
llvm-link file1.bc file2.bc -o combined.bc
```

This generates a `combined.bc` file containing the linked bitcode.

3. **Common Options**

   - **`-o <output_file>`**: Specifies the output file.
   - **`-S`**: Writes the output in textual IR format (`.ll`).
   - **`-internalize`**: Internalizes all symbols except those specified.
   - **`-only-needed`**: Links only the needed symbols.

18.2.4 Practical Examples

1. **Linking Two IR Files**

   Consider two IR files, `file1.ll` and `file2.ll`:

   `file1.ll`:

```
@global_var = global i32 42

define void @func1() {
  ret void
}
```

   `file2.ll`:

```
declare void @func1()

define void @func2() {
  call void @func1()
```

```
  ret void
}
```

To link these files:

```
llvm-link file1.ll file2.ll -o combined.ll
```

The resulting `combined.ll` file contains:

```
@global_var = global i32 42

define void @func1() {
  ret void
}

define void @func2() {
  call void @func1()
  ret void
}
```

2. **Linking with Optimization**

   To link and optimize the combined IR module:

   ```
   llvm-link file1.ll file2.ll -o combined.ll
   opt -O2 combined.ll -o optimized.ll
   ```

   This applies LLVM's `-O2` optimizations to the linked IR.

3. **Creating an IR Library**

To create an IR library that can be linked with other modules:

```
llvm-link lib1.ll lib2.ll -o libcombined.bc
```

This generates a `libcombined.bc` file that can be linked with other IR modules.

18.2.5 Advanced Topics

1. **Symbol Internalization**

   The `-internalize` flag internalizes all symbols except those specified, making them private to the module. This is useful for creating libraries.

   Example:

   ```
   llvm-link -internalize file1.ll file2.ll -o combined.ll
   ```

2. **Linking with Debug Information**

   When linking IR files with debug information, `llvm-link` preserves the debug metadata, ensuring that the combined module can be debugged effectively.

3. **Custom Linking Scripts**

   Advanced users can create custom linking scripts to automate the linking process, specifying which symbols to include or exclude.

4. **Linking with External Libraries**

   `llvm-link` can be used to link IR modules with external libraries, such as the C standard library, by including their IR or bitcode files.

18.2.6 Conclusion

Linking with `llvm-link` is a powerful technique for combining multiple LLVM IR files into a single module, enabling advanced users to build complex systems and libraries. By understanding its usage, options, and practical applications, developers can effectively manage and optimize their IR code. Whether linking IR files, creating libraries, or applying optimizations, `llvm-link` provides the flexibility and control needed to work with LLVM IR at an advanced level.

This section has provided a comprehensive exploration of linking with `llvm-link`, equipping users with the knowledge to interface LLVM IR with external tools and processes in their projects.

18.3 Optimizing with `opt`

Optimization is a cornerstone of the LLVM infrastructure, enabling developers to improve the performance, size, and efficiency of their code. The `opt` tool is a powerful utility for applying optimization passes to LLVM Intermediate Representation (IR). This section provides a detailed exploration of **optimizing with `opt`**, focusing on its usage, capabilities, and practical applications for advanced users.

18.3.1 Overview of `opt`

The `opt` tool is a command-line utility that applies optimization passes to LLVM IR. It is part of the LLVM project and is used to:

1. **Apply optimizations**: Improve performance, reduce code size, and enhance efficiency.
2. **Analyze IR**: Gather information about the IR, such as function call graphs and memory usage.
3. **Transform IR**: Modify the IR to enable further optimizations or meet specific requirements.

`opt` supports both textual IR (`.ll`) and bitcode (`.bc`) formats.

18.3.2 Key Concepts

1. **Optimization Passes**

   Optimization passes are modular units of optimization that perform specific tasks, such as dead code elimination, loop unrolling, and function inlining.

2. **Pass Pipelines**

A pass pipeline is a sequence of optimization passes that are executed in a specific order. LLVM provides predefined pipelines (e.g., `-O1`, `-O2`, `-O3`) as well as the ability to define custom pipelines.

3. **Analysis Passes**

   Analysis passes gather information about the IR without modifying it. This information is used by transformation passes to guide optimizations.

4. **Transformation Passes**

   Transformation passes modify the IR to improve performance, reduce code size, or enable further optimizations.

18.3.3 Using `opt`

The `opt` tool is invoked from the command line with the following syntax:

```
opt [options] <input_file> -o <output_file>
```

- **<input_file>**: The input IR file (`.ll` or `.bc`).
- **<output_file>**: The output file containing the optimized IR (optional).

1. **Basic Usage**

   To apply optimizations to an IR file:

   ```
   opt -O2 input.ll -o output.ll
   ```

   This applies the `-O2` optimization pipeline to `input.ll` and writes the optimized IR to `output.ll`.

2. **Common Options**

   - **`-O<level>`**: Specifies the optimization level (e.g., `-O0`, `-O1`, `-O2`, `-O3`).
   - **`-S`**: Writes the output in textual IR format (`.ll`).
   - **`-passes=<passes>`**: Specifies a custom sequence of passes (new pass manager).
   - **`-stats`**: Prints statistics about the IR.
   - **`-debug`**: Enables debug output.

3. **Example: Applying Optimizations**

   Consider an IR file `input.ll`:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

   To apply the `-O2` optimization pipeline:

```
opt -O2 input.ll -o output.ll
```

   The optimized IR in `output.ll` might look like this:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

   In this simple example, the IR remains unchanged, but more complex IR would show significant transformations.

18.3.4 Practical Examples

1. **Applying Individual Passes**

   To apply specific optimization passes, use the `-passes` option (new pass manager):

   ```
   opt -passes='instcombine,dce' input.ll -o output.ll
   ```

   This applies the `instcombine` (instruction combining) and `dce` (dead code elimination) passes.

2. **Analyzing IR**

   To analyze the IR without modifying it, use analysis passes:

   ```
   opt -passes='print<cfg>' input.ll
   ```

   This prints the control flow graph (CFG) of the IR.

3. **Custom Pass Pipelines**

   Advanced users can define custom pass pipelines to apply specific sequences of optimizations. For example:

   ```
   opt -passes='inline,loop-unroll' input.ll -o output.ll
   ```

   This inlines functions and unrolls loops.

18.3.5 Advanced Topics

1. **Legacy vs. New Pass Manager**

   - **Legacy Pass Manager**: The older pass manager, which uses the `-pass-name` syntax.

- **New Pass Manager**: The modern pass manager, which uses the `-passes` syntax and provides better performance and flexibility.

Example (New Pass Manager):

```
opt -passes='inline,loop-unroll' input.ll -o output.ll
```

2. **Custom Optimization Passes**

   Advanced users can write custom optimization passes in C++ and load them into `opt` using the `-load` option. For example:

```
opt -load ./MyPass.so -mypass input.ll -o output.ll
```

3. **Debugging Optimizations**

   To debug optimizations, use the `-debug` flag:

```
opt -O2 -debug input.ll -o output.ll
```

   This enables debug output, providing detailed information about the optimization process.

4. **Profile-Guided Optimization (PGO)**

   LLVM supports profile-guided optimization, where optimization decisions are based on runtime profiling data. Use the `-fprofile-instr-generate` and `-fprofile-instr-use` flags with `clang` to generate and use profiling data.

18.3.6 Conclusion

Optimizing with `opt` is a powerful technique for improving the performance, size, and efficiency of LLVM IR. By understanding its usage, options, and practical applications, advanced users can leverage `opt` to apply a wide range of optimizations, analyze IR, and transform code to meet specific requirements. Whether using predefined optimization pipelines, custom pass sequences, or advanced features like PGO, `opt` provides the flexibility and control needed to optimize LLVM IR effectively.

This section has provided a comprehensive exploration of optimizing with `opt`, equipping users with the knowledge to interface LLVM IR with external tools and processes in their projects.

18.4 JIT Compilation with LLVM's ORC

Just-In-Time (JIT) compilation is a powerful technique that allows code to be compiled and executed at runtime, enabling dynamic optimization and flexibility. LLVM's **On-Request Compilation (ORC)** API provides a modern and modular framework for JIT compilation, making it easier to integrate JIT capabilities into applications. This section provides a detailed exploration of **JIT compilation with LLVM's ORC**, focusing on its architecture, usage, and practical applications for advanced users.

18.4.1 Overview of JIT Compilation

JIT compilation bridges the gap between interpreted and statically compiled languages by compiling code at runtime. This approach offers several advantages:

1. **Dynamic Optimization**: Code can be optimized based on runtime information.
2. **Flexibility**: New code can be generated and executed dynamically.
3. **Performance**: JIT-compiled code can achieve performance close to statically compiled code.

LLVM's ORC API is designed to be modular, extensible, and easy to use, making it suitable for a wide range of JIT compilation scenarios.

18.4.2 Key Concepts

1. **On-Request Compilation (ORC)**

   ORC is LLVM's modern JIT compilation framework, designed to replace the older MCJIT (Machine Code JIT) framework. It provides a modular and layered architecture for JIT compilation.

2. **Layers in ORC**

   ORC is built around the concept of **layers**, which are modular components that handle specific tasks in the JIT compilation process. Common layers include:

   - **IRCompileLayer**: Compiles LLVM IR to machine code.
   - **ObjectLinkingLayer**: Links compiled object files into memory.
   - **CompileOnDemandLayer**: Compiles functions on demand.

3. **ExecutionSession**

   The `ExecutionSession` class manages the execution of JIT-compiled code, including symbol resolution and memory management.

4. **JITDylib**

   The `JITDylib` class represents a dynamic library in the JIT context, managing symbols and their definitions.

5. **Lazy Compilation**

   Lazy compilation defers the compilation of functions until they are called, reducing startup time and memory usage.

18.4.3 Using LLVM's ORC API

The ORC API is used to create and manage JIT compilation sessions, compile LLVM IR or bitcode, and execute the compiled code.

1. **Basic Workflow**

   (a) **Create an ExecutionSession**: Initialize the JIT environment.

   (b) **Create a JITDylib**: Define a dynamic library for the JIT session.

(c) **Add IR or Bitcode**: Provide the code to be JIT-compiled.

(d) **Lookup Symbols**: Resolve symbols (e.g., function addresses) for execution.

(e) **Execute Code**: Call the JIT-compiled functions.

2. **Example: JIT Compiling a Simple Function**

   Consider the following C code:

```c
int add(int a, int b) {
    return a + b;
}
```

   The corresponding LLVM IR is:

```
define i32 @add(i32 %a, i32 %b) {
  %result = add i32 %a, %b
  ret i32 %result
}
```

   To JIT-compile and execute this function using ORC:

   (a) **Create an ExecutionSession**:

```c
auto ES = std::make_unique<llvm::orc::ExecutionSession>();
```

   (b) **Create a JITDylib**:

```c
auto JD = ES->createJITDylib("main");
```

(c) **Add IR or Bitcode**:

```
llvm::orc::ThreadSafeContext
↪  TSCtx(std::make_unique<llvm::LLVMContext>());
auto M = llvm::parseIRFile("add.ll", *TSCtx.getContext());
if (!M) {
    // Handle error
}
```

(d) **Compile and Link**:

```
llvm::orc::RTDyldObjectLinkingLayer ObjectLayer(*ES, []() {
    return std::make_unique<llvm::SectionMemoryManager>();
});
llvm::orc::IRCompileLayer CompileLayer(*ES, ObjectLayer,
↪  std::make_unique<llvm::orc::SimpleCompiler>());
auto Handle = cantFail(CompileLayer.add(JD,
↪  llvm::orc::ThreadSafeModule(std::move(M), TSCtx));
```

(e) **Lookup and Execute**:

```
auto AddSym = cantFail(ES->lookup({&JD}, "add"));
auto Add = (int(*)(int, int))AddSym.getAddress();
int Result = Add(2, 3);  // Result = 5
```

18.4.4 Practical Applications

1. **Dynamic Code Generation**

   JIT compilation enables dynamic code generation, where new code is generated and executed at runtime. This is useful for scripting languages, dynamic optimization, and runtime code specialization.

2. **Lazy Compilation**

   Lazy compilation improves startup time and reduces memory usage by deferring the compilation of functions until they are called.

3. **Custom Optimization Passes**

   JIT-compiled code can be optimized dynamically based on runtime information, such as profiling data or input characteristics.

4. **Interactive Development**

   JIT compilation supports interactive development environments, where code can be modified and executed on the fly.

18.4.5 Advanced Topics

1. **Custom Layers**

   Advanced users can define custom layers to extend ORC's functionality, such as adding custom optimizations or integrating with external tools.

2. **Lazy Compilation with CompileOnDemandLayer**

   The `CompileOnDemandLayer` enables lazy compilation by deferring the compilation of functions until they are called.

   Example:

```
llvm::orc::CompileOnDemandLayer CODLayer(*ES, CompileLayer);
auto Handle = cantFail(CODLayer.add(JD,
 ↪  llvm::orc::ThreadSafeModule(std::move(M), TSCtx));
```

3. **Profile-Guided Optimization (PGO)**

   JIT-compiled code can be optimized dynamically using runtime profiling data, improving performance based on actual usage patterns.

4. **Cross-Platform Support**

   ORC supports cross-platform JIT compilation, enabling the same code to be JIT-compiled and executed on different architectures.

18.4.6 Conclusion

JIT compilation with LLVM's ORC API provides a powerful and flexible framework for dynamic code generation, optimization, and execution. By understanding its architecture, usage, and practical applications, advanced users can leverage ORC to build high-performance, dynamic, and interactive systems. Whether implementing scripting engines, dynamic optimizers, or interactive development environments, ORC offers the modularity and extensibility needed to meet a wide range of JIT compilation needs.
This section has provided a comprehensive exploration of JIT compilation with LLVM's ORC, equipping users with the knowledge to interface LLVM IR with external tools and processes in their projects.

Appendices for LLVM IR Quick Reference Booklet

Appendix A: LLVM IR Instruction Set Summary

- **Arithmetic Instructions:** `add`, `sub`, `mul`, `sdiv`, `udiv`, `fadd`, `fsub`, `fmul`, `fdiv`
- **Bitwise Instructions:** `and`, `or`, `xor`, `shl`, `lshr`, `ashr`
- **Memory Instructions:** `alloca`, `load`, `store`, `getelementptr`
- **Control Flow Instructions:** `br`, `switch`, `indirectbr`, `ret`, `invoke`
- **Conversion Instructions:** `trunc`, `zext`, `sext`, `fptrunc`, `fpext`, `bitcast`, `inttoptr`, `ptrtoint`
- **Atomic Instructions:** `atomicrmw`, `cmpxchg`, `fence`
- **Vector Instructions:** `extractelement`, `insertelement`, `shufflevector`
- **Aggregate Instructions:** `extractvalue`, `insertvalue`

Appendix B: LLVM IR Intrinsics Cheat Sheet

- **Math Intrinsics:** `llvm.sqrt`, `llvm.sin`, `llvm.cos`, `llvm.pow`
- **Memory Intrinsics:** `llvm.memcpy`, `llvm.memmove`, `llvm.memset`
- **Atomic Intrinsics:** `llvm.atomic.load`, `llvm.atomic.store`, `llvm.atomic.cmp.xchg`
- **Exception Handling Intrinsics:** `llvm.eh.typeid.for`, `llvm.eh.sjlj.setjmp`
- **Debug Intrinsics:** `llvm.dbg.declare`, `llvm.dbg.value`
- **Vector Intrinsics:** `llvm.vector.reduce.add`, `llvm.vector.reduce.fmax`

Appendix C: LLVM IR Metadata Tags

- **Debug Info Metadata:** `!DILocation`, `!DILocalVariable`, `!DIGlobalVariable`
- **Module Flags Metadata:** `!llvm.module.flags`
- **Function Metadata:** `!llvm.loop`, `!llvm.mem.parallel_loop_access`
- **Type Metadata:** `!llvm.type`, `!llvm.dbg.typedef`
- **Profiling Metadata:** `!llvm.prof`, `!llvm.loop.unroll.disable`

Appendix D: LLVM IR Optimization Flags

- **Common Optimization Flags:**

  - **`-O0`**: No optimization
  - **`-O1`**: Basic optimizations
  - **`-O2`**: Moderate optimizations
  - **`-O3`**: Aggressive optimizations
  - **`-Os`**: Optimize for size
  - **`-Oz`**: Optimize for size aggressively

- **Specific Optimization Flags:**
  - **`-mem2reg`**: Promote memory to registers
  - **`-instcombine`**: Combine redundant instructions
  - **`-loop-unroll`**: Unroll loops
  - **`-gvn`**: Global Value Numbering
  - **`-inline`**: Function inlining

Appendix E: LLVM IR Target Triples

- **Common Target Triples:**
  - **x86_64:** `x86_64-pc-linux-gnu`, `x86_64-apple-darwin`, `x86_64-w64-mingw32`
  - **ARM:** `arm-linux-gnueabi`, `arm-none-eabi`, `armv7-apple-ios`
  - **AArch64:** `aarch64-linux-gnu`, `aarch64-apple-darwin`
  - **RISC-V:** `riscv64-unknown-elf`, `riscv32-unknown-linux-gnu`
  - **MIPS:** `mips-linux-gnu`, `mipsel-linux-gnu`

Appendix F: LLVM IR Tools & Utilities

- **IR Generation Tools:**
  - **`clang`**: Generate IR from C/C++ (`clang -emit-llvm -S -o output.ll input.c`)
  - **`llvm-as`**: Convert textual IR to bitcode (`llvm-as input.ll -o output.bc`)
  - **`llvm-dis`**: Convert bitcode to textual IR (`llvm-dis input.bc -o output.ll`)
- **Optimization Tools:**
  - **`opt`**: Run optimization passes (`opt -O3 -S input.ll -o output.ll`)
  - **`llvm-link`**: Link multiple IR files (`llvm-link file1.ll file2.ll -o output.ll`)
- **Debugging Tools:**
  - **`llvm-dwarfdump`**: Dump DWARF debug info (`llvm-dwarfdump input.bc`)
  - **`llvm-objdump`**: Disassemble object files (`llvm-objdump -d input.o`)
- **JIT Tools:**
  - **`lli`**: Execute IR directly (`lli input.ll`)
  - **`llvm-mc`**: Assemble and disassemble machine code (`llvm-mc --assemble input.s -o output.o`)

Appendix G: LLVM IR Examples

- **Simple Arithmetic Function:**

```
define i32 @add(i32 %a, i32 %b) {
    %sum = add i32 %a, %b
    ret i32 %sum
}
```

- **Control Flow with Conditional Branch:**

```
define i32 @max(i32 %a, i32 %b) {
    %cmp = icmp sgt i32 %a, %b
    br i1 %cmp, label %if, label %else
if:
    ret i32 %a
else:
    ret i32 %b
}
```

- **Memory Allocation and Access:**

```
define i32 @main() {
    %ptr = alloca i32
    store i32 42, i32* %ptr
    %val = load i32, i32* %ptr
    ret i32 %val
}
```

- **Vector Operations:**

```
define <4 x i32> @add_vec(<4 x i32> %a, <4 x i32> %b) {
    %result = add <4 x i32> %a, %b
    ret <4 x i32> %result
}
```

Appendix H: LLVM IR Resources & Further Reading

- **Official Documentation:**
  - LLVM Language Reference Manual
  - LLVM Programmer's Manual
- **Books:**
  - *"LLVM Essentials"* by Suyog Sarda and Mayur Pandey
  - *"Getting Started with LLVM Core Libraries"* by Bruno Cardoso Lopes and Rafael Auler
- **Online Tutorials:**
  - LLVM Tutorial
  - Kaleidoscope: Implementing a Language with LLVM
- **Community & Forums:**
  - LLVM Discourse
  - LLVM Mailing Lists

Appendix I: LLVM IR Glossary

- **Basic Block (BB):** A sequence of instructions with a single entry and exit point.
- **SSA (Static Single Assignment):** A form where each variable is assigned exactly once.
- **IR (Intermediate Representation):** A low-level, platform-independent representation of code.
- **Module:** The top-level container in LLVM IR, containing functions, global variables, and metadata.
- **Function:** A callable unit of code with a set of arguments and a return type.
- **Instruction:** A single operation in LLVM IR, such as `add`, `load`, or `store`.
- **Metadata:** Additional information attached to LLVM IR for debugging, optimization, or analysis.
- **Intrinsic:** A built-in function provided by LLVM for specific operations like math or memory management.

Appendix J: LLVM IR Version History

- **LLVM 1.0 (2003):** Initial release with basic IR and optimizations.
- **LLVM 2.0 (2005):** Introduction of SSA form and more advanced optimizations.
- **LLVM 3.0 (2011):** Major improvements in IR stability and target support.
- **LLVM 4.0 (2017):** Introduction of new intrinsics and better support for modern architectures.

- **LLVM 10.0 (2020):** Enhanced support for C++20, new optimization passes, and improved debugging tools.
- **LLVM 15.0 (2022):** Introduction of opaque pointers, improved vectorization, and better support for heterogeneous computing.

Appendix K: LLVM IR Frequently Asked Questions (FAQ)

- **Q: How do I generate LLVM IR from C/C++ code?**
  - **A:** Use `clang -emit-llvm -S -o output.ll input.c`.
- **Q: What is the difference between `i32` and `i64`?**
  - **A:** `i32` is a 32-bit integer, while `i64` is a 64-bit integer.
- **Q: How do I optimize LLVM IR?**
  - **A:** Use the `opt` tool with optimization flags like `-O3` or specific passes like `-mem2reg`.
- **Q: Can I execute LLVM IR directly?**
  - **A:** Yes, using `lli` or by compiling it to machine code with `llc`.
- **Q: What is the purpose of metadata in LLVM IR?**
  - **A:** Metadata provides additional information for debugging, optimization, or analysis, such as source locations or variable names.

Appendix L: LLVM IR Quick Reference Tables

- **Data Types:**

Type	Description
`iN`	Integer of N bits
`float`	32-bit floating-point
`double`	64-bit floating-point
`void`	No type (used for functions)
`ptr`	Pointer type
`<N x T>`	Vector of N elements of type T

- **Common Instructions:**

Instruction	Description
`add`	Integer addition
`fadd`	Floating-point addition
`load`	Load from memory
`store`	Store to memory
`br`	Branch (conditional/unconditional)
`ret`	Return from function

- **Optimization Passes:**

Pass	Description
`-mem2reg`	Promote memory to registers
`-instcombine`	Combine redundant instructions
`-gvn`	Global Value Numbering
`-inline`	Function inlining