Lecture 2:
Intermediate Representation (IR)

CESE4085 Modern Computer Architecture Course
Part 2, L6.2
Carlo Galuzzi

LECTURE CONTENTS

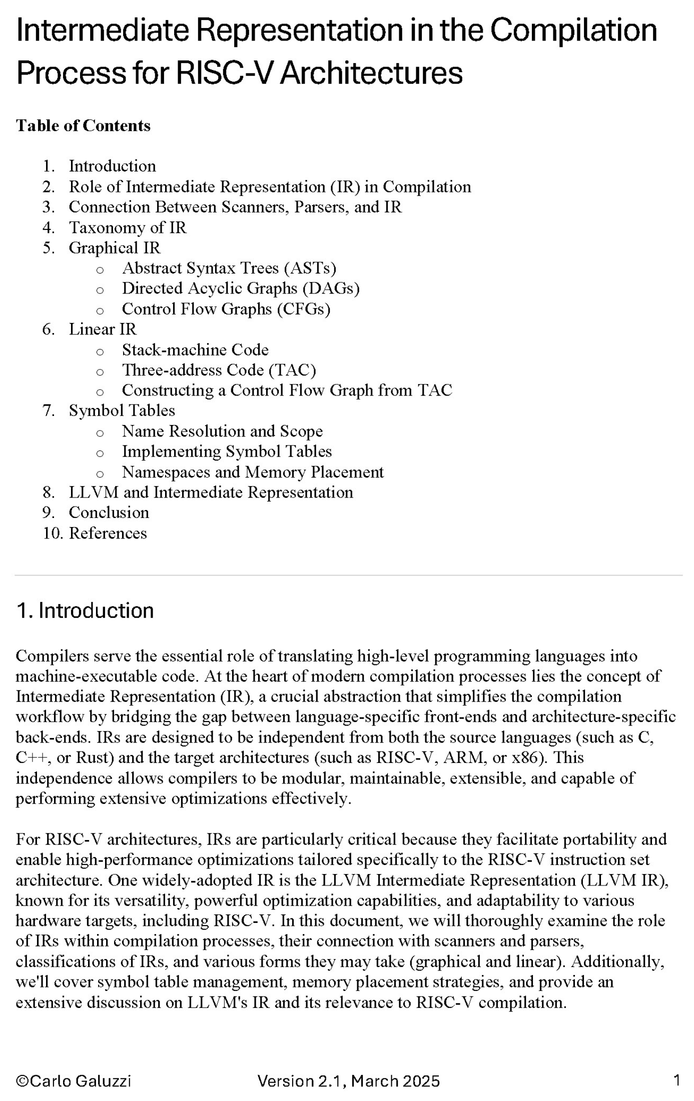
- Introduction to IR and its Importance in Compilation
- Graphical IR: AST, DAG, and CFG
- Linear IR: Stack-machine Code and Three-address Code (TAC)
- Constructing CFG from TAC
- Role of Symbol Tables, Namespaces, and Memory Allocation
- Overview of LLVM IR and its Optimization Capabilities
- Key Takeaways and Future Insights on Compiler Design

LEARNING GOALS:

Understanding Intermediate Representation (IR) is crucial for mastering compiler design and optimization techniques. By the end of this lecture, among other goals, you will be able to:

- **Understand the Role of IR:** Learn how IR bridges the gap between source code and machine code.
- **Differentiate Between IR Types:** Gain insights into Graphical IR (AST, DAG, CFG) and Linear IR (TAC, Stack-machine code).
- **Explore Symbol Tables and Memory Management:** See how compilers track and allocate identifiers.
- **Recognize the Importance of LLVM IR:** Discover how LLVM IR enhances compilation and optimization.

EXTRA READING MATERIAL IN BRIGHSPACE



Intermediate Representation in the Compilation Process for RISC-V Architectures

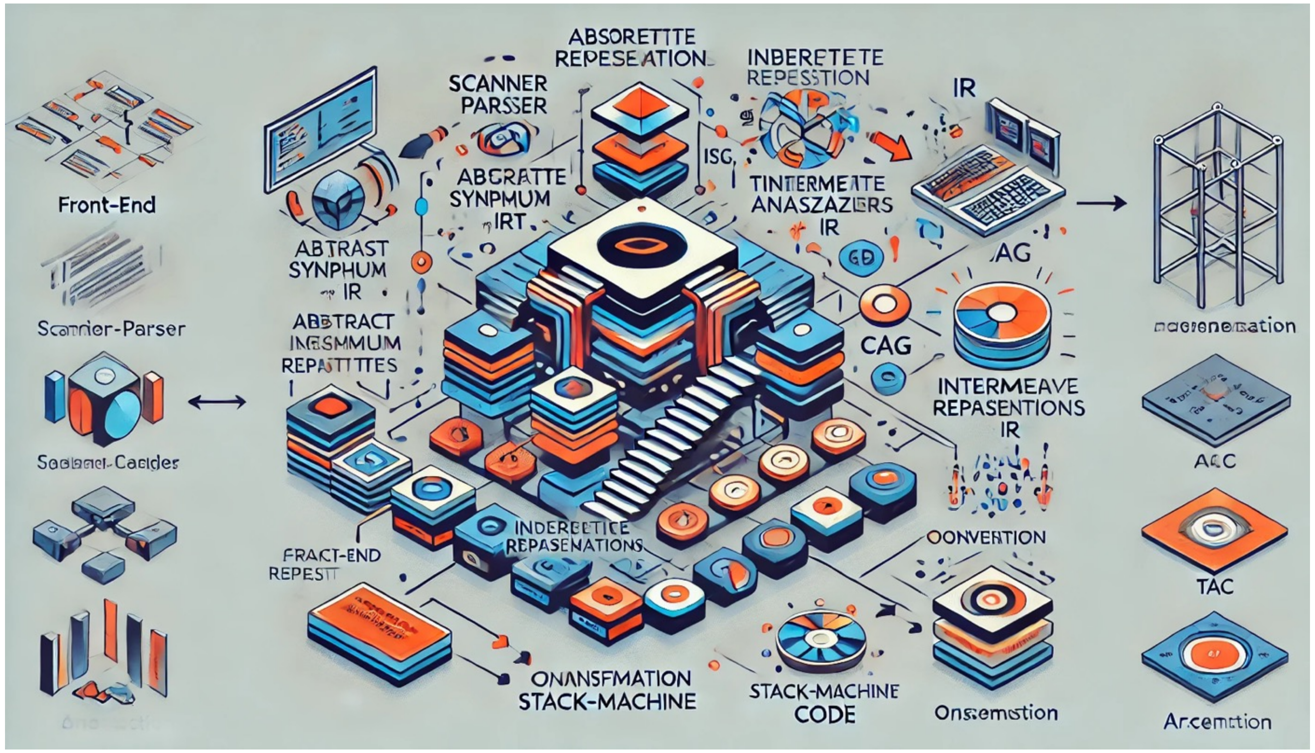
Table of Contents

1. Introduction

Compilers serve the essential role of translating high-level programming languages into machine-executable code. At the heart of modern compilation processes lies the concept of Intermediate Representation (IR), a crucial abstraction that simplifies the compilation workflow by bridging the gap between language-specific front-ends and architecture-specific back-ends. IRs are designed to be independent from both the source languages (such as C, C++, or Rust) and the target architectures (such as RISC-V, ARM, or x86). This independence allows compilers to be modular, maintainable, extensible, and capable of performing extensive optimizations effectively.

For RISC-V architectures, IRs are particularly critical because they facilitate portability and enable high-performance optimizations tailored specifically to the RISC-V instruction set architecture. One widely-adopted IR is the LLVM Intermediate Representation (LLVM IR), known for its versatility, powerful optimization capabilities, and adaptability to various hardware targets, including RISC-V. In this document, we will thoroughly examine the role of IRs within compilation processes, their connection with scanners and parsers, classifications of IRs, and various forms they may take (graphical and linear). Additionally, we'll cover symbol table management, memory placement strategies, and provide an extensive discussion on LLVM's IR and its relevance to RISC-V compilation.

Intermediate Representation



INTRODUCTION TO INTERMEDIATE REPRESENTATION

Compilers must translate high-level programming languages into executable machine code, but direct translation poses significant challenges due the diversity of programming languages and hardware architectures.

Intermediate Representation (IR) addresses this by providing an abstraction layer, simplifying the translation process.
IR acts as a bridge between source code and machine code, allowing compilers to optimize programs before they are translated into architecture-specific instructions.

- **Language-independent:** IR abstracts away details specific to source languages and target machines, facilitating portability.
- **Modular compiler design:** IR enables separation of concerns, supporting clear front-end and back-end compiler phases.
- **Optimization enablement:** IR allows extensive program optimizations before final code generation.

IR AND MODULARITY

Imagine building a compiler for every possible combination of programming languages and target architectures—it would be an overwhelming task!

Intermediate Representation allows compilers to efficiently manage complexity by separating language-specific front-end tasks from architecture-specific back-end tasks. This modular design helps maintainability and allows easy expansion to new languages and architectures.

- **Common Target Representation:** IR provides a universal format for diverse programming languages (e.g., C, Rust, C++).
- **Backend Reuse:** Enables reuse of backend optimization and code generation across multiple hardware platforms, such as RISC-V, ARM, and x86.
- **Enhanced Maintainability:** Facilitates easier debugging and development within compiler toolchains.

IR CLASSIFICATION PROBLEM

Not all IRs are the same, and choosing the right type is essential for effective compilation. Classifying Intermediate Representations (IRs) helps compiler designers select the most appropriate representation for each stage of compilation.

Different IR types are tailored for specific optimization tasks, affecting both compiler complexity and code quality.

- **Graphical vs. Linear IR:** Graphical IR represents program structures explicitly (useful for early optimizations), whereas Linear IR focuses on sequential instruction execution.
- **Optimization Suitability:** Graphical IR is preferred for high-level structural optimizations, and Linear IR is optimized for detailed, low-level optimizations and analysis.

OVERVIEW OF GRAPHICAL IR

Graphical IR provides a **structural representation** of the program, making it easier to visualize dependencies between different operations.

Examples include:

- **Abstract Syntax Trees (ASTs)**,
- **Directed Acyclic Graphs (DAGs)**,
- **Control Flow Graphs (CFGs)**.

- **Structural Representation:** Shows how different program components interact.
- **Optimizes High-Level Operations:** Enables optimizations such as common subexpression elimination.
- **Useful for Semantic Analysis:** Helps the compiler check variable usage, function calls, and dependencies efficiently.

LINEAR IR OVERVIEW

Once the high-level structure of the program is analyzed, the compiler needs a format that more closely resembles machine instructions.

Linear Intermediate Representations (IRs) simplify program instructions into sequential, low-level operations. Common forms of Linear IR include **Three-address Code (TAC)** and **Stack-machine Code**.

This simplification aids the compiler in efficiently managing detailed optimizations and generating machine-specific code.

- **Sequential Instructions:** Explicit sequence of simple operations.
- **Optimized for Backend Processing:** Facilitates low-level optimizations like instruction scheduling and register allocation.
- **Clear Operand Management:** Explicitly indicates the operands and results, aiding analysis and optimization.

INTRODUCTION TO ABSTRACT SYNTAX TREES (AST)

Abstract Syntax Trees (ASTs) provide a hierarchical, graphical representation of program syntax and they are one of the most fundamental forms of IR.

They strip away syntactic details unnecessary for semantic analysis, clarifying logical structures and simplifying error detection.

- **Hierarchical Representation:** Clearly depicts the syntactic hierarchy of code constructs.
- **Supports Semantic Analysis:** Simplifies tasks such as type checking and scope resolution.
- **Foundation for Optimizations:** Provides a base structure from which further graphical IR optimizations (e.g., DAGs, CFGs) can be derived.

SEMANTIC ANALYSIS WITH ABSTRACT SYNTAX TREES (AST)

Abstract Syntax Trees (ASTs) significantly simplify semantic analysis by clearly structuring code into logical components.

This hierarchical representation allows compilers to quickly identify and resolve semantic issues, improving compilation accuracy.

- **Early Error Detection:** Helps detect semantic errors early, such as undeclared variables, type mismatches, and incorrect function calls .
- **Simplified Semantic Checks:** Easier identification and correction of semantic issues due to structured representation.
- **Efficient Error Reporting:** Clearly pinpoint the exact location and nature of semantic errors.

DIRECTED ACYCLIC GRAPHS (DAGS)

Directed Acyclic Graphs (DAGs) take ASTs one step further and enhance program optimization by eliminating redundant subexpressions and ensuring efficient computation.

By transforming an Abstract Syntax Tree (AST) into a DAG, the compiler can **detect and reuse previously computed values**, reducing unnecessary operations (e.g., `(a + b) * (a + b)`: in an AST, the computation `a + b` would be stored twice, a DAG optimizes this by **sharing** the result across the tree).

- **Eliminates Redundant Computations:** Identifies and merges common subexpressions, reducing execution time.
- **Optimizes Resource Utilization:** Minimizes memory and CPU usage by preventing duplicate calculations.
- **Facilitates Advanced Optimizations:** Helps compilers with instruction scheduling and register allocation.

CONTROL FLOW GRAPHS (CFG)

Control Flow Graphs (CFGs) provide a **structural representation of execution flow** in a program. CFGs represent possible execution paths in a program explicitly through nodes and edges.

Each node represents a basic block (a sequence of instructions with a single entry and exit point), and edges indicate possible execution paths. CFGs are instrumental in **analyzing loops, conditionals, and optimizing control structures**.

- **Visualizes Program Execution Flow:** Clearly shows how the program moves through conditional statements and loops.
- **Essential for Optimizations:** Helps eliminate dead code, improve branch prediction, and optimize loops.
- **Supports Advanced Analysis:** Assists with reaching definitions, live-variable analysis, and dominance relations.

CFG EXAMPLE WITH CONDITIONAL STATEMENTS

Control Flow Graphs (CFGs) help visualize how execution flows through a program, especially in branching scenarios. This slide demonstrates how a conditional statement is transformed into a CFG, allowing the compiler to analyze control paths and optimize accordingly. For example:

```
if (x > y)
    x = y;
else
    x = z;
```

- **Branches Clearly Defined:** Shows explicit execution paths for true and false conditions.
- **Optimization Opportunities:** Helps detect unreachable code and redundant branching.
- **Improves Execution Flow Analysis:** Enables better branch prediction and loop optimization.

STACK-MACHINE CODE

Stack-machine Intermediate Representation (IR) processes computations using a stack-based approach rather than registers. Stack-machine IR differs from register-based IR by relying on a **last-in, first-out (LIFO) stack** to store operands and results.

It is widely used in virtual machines and interpreted languages because of its simplicity and compact encoding.

- **Implicit Operand Handling:** Operands are stored and retrieved from a stack instead of explicit register references.
- **Compact Encoding:** Requires fewer instructions compared to register-based IR.
- **Simplified Execution Model:** Easier to implement in virtual machines and embedded systems.

STACK-MACHINE CODE EXAMPLE

Stack-machine IR executes operations by pushing operands onto a stack, performing operations, and then popping the results. This approach simplifies execution but may introduce performance overhead due to frequent stack operations.

Expression:

```
z = x + y;
```

Stack-machine IR:

```
push x
push y
add
pop z
```

- **No Explicit Operands:** Operands are managed through stack operations instead of registers.
- **Simple Instruction Format:** Fewer instructions compared to register-based IR.
- **Potential Performance Overhead:** Excessive push/pop operations may slow execution in computation-heavy applications.

STACK-MACHINE IR: PROS AND CONS

Stack-machine IR has advantages and disadvantages that determine where it is best applied. It is particularly useful in virtual machines and embedded systems but may not be the most efficient for performance-critical applications.

Pros:

- Compact encoding reduces instruction set complexity.
- Simplicity of execution makes it ideal for interpreted languages and embedded devices.
- No need for explicit register management, simplifying compiler design.

Cons:

- Frequent stack operations may introduce execution delays.
- Less efficient than register-based architectures for arithmetic-heavy workloads.
- Harder to optimize compared to Three-address Code or SSA-based IR.

THREE-ADDRESS CODE (TAC) OVERVIEW

Three-address Code (TAC) is a form of Linear IR that explicitly represents operations with up to three operands per instruction.

This structure makes it easier for compilers to perform optimizations and translates more efficiently into machine code.

- **Explicit Intermediate Results:** Clearly defines and stores temporary computation values.
- **Simplifies Data Flow Analysis:** Helps compilers track dependencies between operations.
- **Improves Optimization Potential:** Enables constant folding, common subexpression elimination, and better register allocation.

THREE-ADDRESS CODE EXAMPLE

An example of TAC illustrates how operations are broken down into simple instructions, making it easy to analyze dependencies and optimize computations.

Example:

```
t1 = a + b;
t2 = t1 * c;
result = t2;
```

- **Step-by-step Computation:** Each operation is performed separately, reducing complexity.
- **Easier for Compiler Optimizations:** Explicit use of temporary variables allows better optimization.
- **Enhances Readability:** Debugging and analysis become more structured and manageable.

BENEFITS OF THREE-ADDRESS CODE

Three-address Code (TAC) enhances compiler optimizations by explicitly defining operand dependencies and computation steps. Its structured format allows efficient data flow analysis, making it easier for compilers to optimize performance.

- **Improved Optimization Potential:** TAC enables advanced optimizations like constant propagation and dead code elimination.
- **Clear Operand Dependencies:** Simplifies register allocation and instruction reordering.
- **Easier Debugging and Maintainability:** Explicit temporary variables make it easier to analyze intermediate computations and track program flow.

CONSTRUCTING CFG FROM TAC

Control Flow Graphs (CFGs) can be directly constructed from TAC, allowing compilers to analyze execution flow efficiently. CFGs help visualize branching, loops, and dead code, supporting critical optimizations.

- **Breaks TAC into Basic Blocks:** Each block contains a sequence of instructions with a single entry and exit.
- **Edges Represent Execution Paths:** Helps compilers understand possible execution flows.
- **Optimization Enabler:** CFGs are essential for loop unrolling, branch prediction, and eliminating redundant computations.

IMPORTANCE OF LINEAR IR

Linear Intermediate Representation (IR) plays a crucial role in the later stages of compilation, bridging the gap between high-level abstractions and machine instructions.

Unlike Graphical IR, which emphasizes program structure, Linear IR focuses on execution order, making it essential for **instruction scheduling, register allocation, and low-level optimizations**.

- **Explicit Execution Order:** Represents program flow in a way that closely resembles assembly code.
- **Facilitates Backend Optimizations:** Enables precise register allocation and instruction reordering for improved performance.
- **Essential for Target Code Generation:** Linear IR serves as a stepping stone between optimized IR and final machine code.

INTRODUCTION TO SYMBOL TABLES

Symbol tables are one of the most important data structures in a compiler, keeping track of **variables, functions, types, and memory locations** throughout the compilation process.

They ensure that identifiers are correctly linked to their declarations, enabling error-free semantic analysis.

- **Centralized Storage for Identifiers:** Tracks variable names, function names, and type information efficiently.
- **Supports Scope Resolution:** Ensures correct variable access and prevents naming conflicts.
- **Essential for Semantic Analysis:** Helps detect undeclared variables, type mismatches, and other errors early in compilation.

NAME RESOLUTION AND SCOPE*

Name resolution ensures that every identifier references the correct declaration within its scope.

This process is essential for avoiding errors and ensuring that variables and functions are accessed within their proper scope.

- **Correct Identifier Association:** Ensures that variables and functions are used in the correct context.
- **Scope Management:** Differentiates between local, global, and block scopes to prevent conflicts.
- **Prevents Naming Errors:** Avoids issues like variable shadowing and redeclaration conflicts.

IMPLEMENTING SYMBOL TABLES

Efficient symbol table implementation is crucial for compiler performance. Symbol tables can be implemented using different data structures, each with advantages and trade-offs.

Different data structures impact lookup speed, memory usage, and overall compilation efficiency.

- **Hash Tables:** Provide rapid insertion, deletion, and lookup operations (average constant time complexity) and are commonly used in modern compilers.
- **Balanced Trees (AVL, Red-Black):** Offer predictable and stable performance for operations, ideal for large-scale software projects.
- **Performance Considerations:** Choosing the right structure optimizes compilation speed and memory efficiency.

NAMESPACES*

Namespaces help organize identifiers logically, preventing naming conflicts and improving code modularity.

This is especially useful in large projects where multiple developers may define variables or functions with the same name in multiple libraries or modules.

- **Prevents Naming Conflicts:** Allows the same identifier to exist in different contexts without collisions.
- **Improves Code Organization:** Groups related functions and variables into logical units.
- **Enhances Code Maintainability:** Enables modular programming, making it easier to manage large projects.

NAMESPACES EXAMPLE*

A practical example shows how namespaces allow functions or variables with the same name to coexist in different modules without conflicts:

```
namespace Math {
  int add(int x, int y);
}
namespace Graphics {
  void add(Object a, Object b);
}
```

- **Eliminates Ambiguity:** Ensures functions with identical names do not interfere with each other.
- **Encourages Modular Development:** Allows developers to organize code logically.
- **Facilitates Code Reusability:** Reduces naming restrictions, making large-scale projects easier to manage.

MEMORY PLACEMENT STRATEGIES*

Memory allocation strategies influence a program’s efficiency. These strategies are critical in determining where variables and data structures reside in memory.

The symbol table helps compilers make efficient decisions regarding memory allocation, ensuring that variables are stored in the appropriate memory region.

- **Stack Allocation:** Used for local variables with automatic storage duration; fast and managed by function calls.
- **Heap Allocation:** Used for dynamically allocated variables that persist beyond function execution; requires manual memory management.
- **Global/Static Allocation:** Used for variables with program-wide lifetime; initialized at compile time and stored in a fixed memory location.

MEMORY PLACEMENT EXAMPLE

A practical illustration of memory placement guided by symbol tables: this example illustrates how variables are placed in memory based on their scope and lifetime, ensuring optimal runtime performance.

```
int global_counter = 0;  // Global memory

void increment() {
    int local_increment = 1;  // Stack memory
    global_counter += local_increment;
}
```

- **Clarifies Variable Storage:** Shows how the compiler determines memory placement.
- **Optimizes Runtime Efficiency:** Ensures proper memory use and deallocation.
- **Prevents Memory Leaks:** Helps manage memory effectively to avoid performance degradation.

LLVM INTERMEDIATE REPRESENTATION (BRIEF OVERVIEW)

LLVM Intermediate Representation (LLVM IR) is a widely used, platform-independent IR that enables extensive optimizations and efficient code generation across multiple architectures, including RISC-V.

It is designed to be both human-readable and suitable for low-level transformations.

- **Static Single Assignment (SSA) Form:** Ensures each variable is assigned only once, simplifying optimizations.
- **Platform Independence:** Enables code reuse and optimization across different architectures.
- **Extensive Optimization Support:** Allows for in-depth analysis and transformations before final code generation.

LLVM IR EXAMPLE*

A simple LLVM IR example shows how its structured format aids in optimization and machine-specific translation.

Example:

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

- **Explicit Instruction Representation:** Shows clear computation steps, aiding analysis.
- **Efficient Code Generation:** Facilitates optimization before target-specific translation.
- **Improves Data Flow Analysis:** SSA form simplifies dependency tracking and register allocation.

KEY TAKEAWAYS

Intermediate Representation (IR) is at the core of modern compiler design, providing a structured way to optimize and translate programs efficiently. This slide summarizes the most important aspects of IR and its impact on compilation.

- **IR is Essential for Compilation:** Bridges the gap between high-level languages and machine code.
- **Graphical vs. Linear IR:** Each serves a distinct purpose, with graphical IR aiding high-level analysis and linear IR facilitating low-level optimizations.
- **Symbol Tables and Memory Management:** Crucial for tracking identifiers, resolving names, and efficiently placing values in memory.
- **LLVM IR's Role in Modern Compilation:** A powerful IR that supports deep optimizations and cross-platform portability.

SUMMARY AND FUTURE INSIGHTS

As computing architectures continue to evolve, IR plays a vital role in shaping compiler efficiency and performance. Future research in IR design and optimization will drive advancements in **compiler performance, security, and adaptability to emerging architectures** such as RISC-V.

- **Future of IR Optimization:** New techniques in IR transformation will enhance compiler capabilities.
- **Adaptability for New Architectures:** The need for IR to efficiently support novel hardware paradigms, including quantum computing and AI-driven architectures.
- **The Growing Role of LLVM IR:** LLVM will likely remain a dominant framework, with ongoing improvements expanding its capabilities.

See you at the next Lecture!

Don’t forget to consult the Q&A blog on Brightspace!