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.

2. Role of Intermediate Representation (IR) in Compilation

Intermediate Representation (IR) plays a fundamental role in structuring the compilation process into clearly defined stages, improving both compiler efficiency and generated code quality. The IR acts as an intermediate abstraction between the front-end, which deals with language-specific processing, and the back-end, which manages architecture-specific optimizations and machine code generation. By clearly separating these concerns, IR allows compiler designers to simplify the process of adding new source languages or targeting new hardware architectures, such as RISC-V, without reinventing the entire compiler.

Compilers are generally structured into three primary phases:

- **Front-end:** The front-end analyzes source code by performing lexical analysis (scanning or tokenization), syntax analysis (parsing), and semantic analysis (type checking, scope resolution). The output of this phase is typically an Abstract Syntax Tree (AST), which captures the structure and semantic content of the source program.
- **Middle-end:** In this phase, the AST is transformed into an IR form that is well-suited for optimizations. The IR enables compilers to apply sophisticated optimizations such as loop unrolling, constant folding, redundancy elimination, and strength reduction. By optimizing at the IR level, these improvements become platform-independent, benefiting multiple target architectures simultaneously.
- **Back-end:** The back-end takes the optimized IR and generates machine-specific code that is efficient for the target architecture. This phase handles instruction selection, register allocation, and assembly code generation tailored explicitly for architectures such as RISC-V.

IRs facilitate portability by providing a common representation format for different languages and architectures. For instance, a single IR can serve as the compilation target for various front-end languages (e.g., C, C++, Rust) and can be used to generate optimized machine code for multiple architectures, including RISC-V, ARM, and x86. LLVM IR, which follows the Static Single Assignment (SSA) form, is particularly advantageous in compiler optimization, as it simplifies data flow analysis, enabling powerful transformations like global value numbering and conditional constant propagation. These transformations significantly enhance program efficiency and runtime performance.

---

3. Connection Between Scanners, Parsers, and IR

To effectively generate Intermediate Representation (IR), the compiler must first thoroughly understand the source program. This understanding is developed through a structured process involving scanners and parsers, which perform lexical and syntactic analysis. Each of these phases incrementally refines the program representation, preparing it for the final transformation into IR.

- **Scanning (Lexical Analysis):** The scanner (or lexer) takes the raw source code and converts it into a stream of tokens. Tokens are atomic units of meaning such as keywords, identifiers, literals, operators, and punctuation symbols. By removing irrelevant details such as whitespace and comments, the scanner produces a simplified sequence that eases subsequent stages of compilation.

- **Parsing (Syntax Analysis):** The parser uses these tokens to build an Abstract Syntax Tree (AST). The AST is a hierarchical representation of the program's structure based on grammatical rules. It captures the syntactical relationships between program constructs but does not yet encode instructions suitable for optimization or machine execution.
- **IR Generation:** Once the AST is formed, it undergoes a translation process into IR. This IR is more abstract and structured than machine code but lower-level and more explicitly defined than an AST. This stage simplifies subsequent optimization phases and facilitates efficient code generation.

For example, consider the following simple C function:

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

The scanning phase identifies tokens: `int`, `add`, `(`, `int`, `a`, `,`, `int`, `b`, `)`, `{`, `return`, `a`, `+`, `b`, `;`, and `}`. Next, parsing organizes these tokens into an AST, where the hierarchy clearly shows a function declaration with parameters (`a` and `b`) and an expression (`a + b`) within a return statement. Lastly, IR generation translates this AST into LLVM IR, resulting in a simplified and clearly defined set of instructions:

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

This explicit, structured IR clearly represents operations and data flow, enabling sophisticated optimizations and effective translation to machine-specific code for RISC-V or other architectures.

The modular connection among scanning, parsing, and IR generation is pivotal in modern compilers, allowing efficient and correct translation from high-level languages to optimized, executable machine code.

---

4. Taxonomy of IR

Intermediate Representations (IRs) can be broadly classified based on their form, structure, and the phase in which they are utilized in the compilation process. Understanding the taxonomy of IRs helps clarify their specific roles, benefits, and limitations within the compilation pipeline.

- **Graphical IR:** Graphical representations explicitly capture structural relationships within a program. Key examples include Abstract Syntax Trees (ASTs), Directed Acyclic Graphs (DAGs), and Control Flow Graphs (CFGs). Graphical IRs are crucial during the earlier stages of compilation because they help preserve and visualize structural and syntactic relationships necessary for semantic analysis and initial optimizations.

- **Linear IR:** Linear representations, such as Three-Address Code (TAC) and Stack-machine code, present instructions as sequential lists. These representations closely resemble machine instructions and are highly suitable for later-stage compiler optimizations like register allocation and instruction scheduling. Linear IRs make the program's data flow explicit, simplifying low-level analysis and transformation.

The choice between graphical and linear IR often depends on the specific optimization requirements and compilation stage. Graphical IRs facilitate high-level structural optimizations such as common subexpression elimination or loop transformations by clearly visualizing program structure. Linear IRs, in contrast, support detailed, instruction-level optimizations necessary for generating highly efficient machine code.

LLVM exemplifies the use of multiple IRs by adopting different levels of representation throughout its compilation pipeline:

1. **High-Level IR (HLLVM IR):** Maintains high-level program constructs like loops, function calls, and complex expressions to enable aggressive early optimizations.
2. **Mid-Level IR (LLVM IR):** Represents the program in a lower-level SSA form, facilitating extensive optimization such as dead code elimination and global value numbering.
3. **Low-Level IR (Machine IR):** This IR closely aligns with the specific architecture (e.g., RISC-V), assisting with hardware-specific optimizations such as instruction scheduling and register allocation, ultimately leading to efficient and effective assembly code.

Thus, understanding the taxonomy and roles of various IRs is crucial for compiler developers aiming to maximize efficiency, performance, and flexibility when targeting architectures such as RISC-V.

---

5. Graphical IR

Graphical Intermediate Representations (IRs) play a pivotal role in the compilation process by explicitly representing structural relationships within programs. These representations enable compiler designers to perform complex semantic analyses and structural optimizations effectively. Graphical IRs visually encode dependencies and relationships between different parts of a program, making it easier to reason about transformations and optimizations.

The primary types of graphical IR include Abstract Syntax Trees (ASTs), Directed Acyclic Graphs (DAGs), and Control Flow Graphs (CFGs). Each of these representations serves specific purposes and supports different optimization strategies.

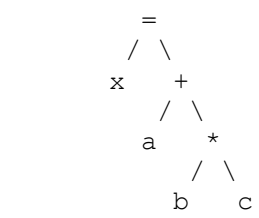
Abstract Syntax Trees (ASTs)

An Abstract Syntax Tree (AST) is a tree-based representation that captures the syntactic structure of source code. ASTs eliminate unnecessary syntax details (such as parentheses or punctuation), preserving only meaningful program structure. This simplification significantly eases semantic analysis, error checking, and initial transformations, such as constant folding and algebraic simplifications.

Consider the following example:

```
x = a + b * c;
```

This statement can be represented as an AST:



The AST explicitly encodes operator precedence, ensuring multiplication (`b * c`) occurs before addition (`a +`). Such representations simplify semantic analysis tasks, like type checking and identifying undeclared variables, making errors easier to detect and correct early in compilation.

ASTs also serve as a foundation for further IR transformations, enabling compilers to generate more optimized forms such as DAGs and CFGs.

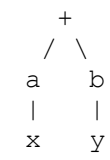
Directed Acyclic Graphs (DAGs)

Directed Acyclic Graphs (DAGs) improve upon ASTs by eliminating redundancy and explicitly representing shared subexpressions. By merging identical expressions into a single node, DAGs reduce computational overhead, optimize memory usage, and simplify subsequent optimization processes.

For example, consider the code snippet:

```
x = a + b;
y = a + b;
```

Rather than computing the expression `a + b` twice, a DAG consolidates it into a single shared node:



This representation highlights common subexpressions, facilitating optimization techniques such as common subexpression elimination (CSE). Reducing redundant calculations is particularly beneficial for architectures like RISC-V, where efficiency and minimal resource use significantly impact overall performance.

Moreover, DAGs enhance clarity and simplify optimization stages by explicitly capturing shared dependencies, which is crucial for effective register allocation and instruction scheduling.

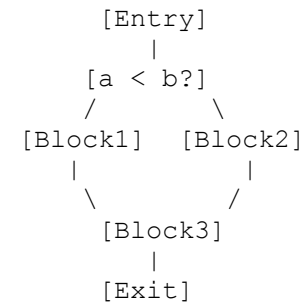
Control Flow Graphs (CFGs)

Control Flow Graphs (CFGs) represent the execution paths within a program. CFGs use nodes to represent basic blocks—sequences of instructions executed sequentially without branching—and edges to indicate potential execution flows between these blocks.

Consider a simple conditional statement:

```
if (a < b)
    x = a;
else
    x = b;
y = x;
```

The corresponding CFG clearly shows conditional branching:



CFGs enable compilers to analyze the execution flow efficiently, supporting advanced optimizations like loop unrolling, dead code elimination, and instruction reordering. The explicit control flow representation provided by CFGs also aids in identifying loops, detecting unreachable code, and optimizing branch prediction.

Furthermore, CFGs facilitate detailed data flow analyses, including dominance analysis, live-variable analysis, and reaching definitions. These analyses help compilers perform optimizations that enhance execution performance, reduce memory use, and efficiently allocate registers for architectures such as RISC-V.

In summary, graphical IRs—ASTs, DAGs, and CFGs—are indispensable tools for compiler developers. They provide structured, visual representations of program logic, enabling sophisticated, structure-sensitive optimizations that greatly enhance both compiler efficiency and generated code performance, particularly on platforms like RISC-V.

---

6. Linear IR

Linear Intermediate Representations (IRs) closely resemble the structure of machine instructions and are essential for compiler stages dealing directly with instruction-level optimizations and final code generation. Linear IR provides a simplified, sequential representation of program execution, significantly facilitating low-level optimizations such as instruction scheduling, register allocation, and detailed machine-specific analyses. Its explicit

and straightforward nature simplifies translation into the final assembly code for architectures like RISC-V.

Linear IRs strip away higher-level abstractions, explicitly describing each computational step and its operands. This explicitness is beneficial for performing optimizations that directly impact performance, especially in resource-constrained or performance-sensitive environments. Additionally, linear IR enables effective register allocation by clearly indicating the lifetimes and dependencies of variables, making it straightforward to minimize register spills and improve register utilization efficiency.

Stack-machine Code

Stack-machine code is a linear IR form that operates using an implicit stack mechanism. Instructions in stack-machine code typically manipulate the top elements of a stack for all arithmetic and logical operations. This approach simplifies instruction encoding, as no explicit operand addressing is required.

For example, the expression `a + b * c` is translated to stack-machine IR as follows:

```
push a
push b
push c
mul
add
```

Each instruction explicitly manipulates the implicit stack, thus simplifying the backend implementation, especially in virtual machines or interpreted environments like the Java Virtual Machine or WebAssembly. This simplicity enables compact code representations, making stack-machine IR efficient in embedded systems or virtual execution environments.

However, this implicit dependence on stack operations can introduce overhead in cases with extensive computation due to frequent stack manipulations, pushing and popping of values, potentially impacting performance compared to register-based IRs.

Three-address Code (TAC)

Three-address Code (TAC) is a more explicit linear IR form where each instruction contains at most three operands, typically representing two operands and a single result. TAC explicitly describes intermediate computations and is widely used in compilers due to its simplicity and ease of manipulation during optimization passes.

Consider the same arithmetic expression `x = a + b * c`. In TAC, this would be represented as:

```
t1 = b * c
t2 = a + t1
x = t2
```

The explicit naming of intermediate results (`t1`, `t2`) simplifies optimization passes like constant propagation, dead code elimination, and common subexpression elimination. TAC

facilitates detailed data flow analysis, which helps optimize register allocation and improve performance by clearly defining data dependencies and lifetimes of intermediate variables.

Additionally, TAC's straightforward nature makes it highly readable and maintainable, benefiting compiler development and debugging.

Constructing a Control Flow Graph from TAC

A Control Flow Graph (CFG) can be constructed directly from Three-address Code (TAC) by organizing sequences of TAC instructions into basic blocks. A basic block is defined as a straight-line code sequence with no internal branches; it has only one entry and one exit point.

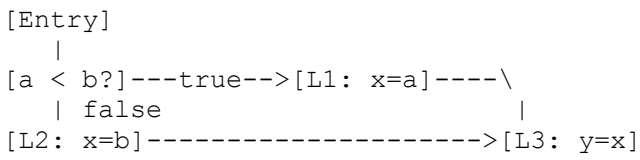
Consider the following example illustrating conditional branching:

```
if (a < b)
  x = a;
else
  x = b;
y = x;
```

This code translates into TAC as follows:

```
if a < b goto L1
goto L2
L1: x = a
goto L3
L2: x = b
L3: y = x
```

This TAC sequence translates directly into the following CFG:



Constructing a CFG from TAC provides an explicit representation of control flow, enabling powerful compiler optimizations such as dead code elimination, loop identification, and instruction scheduling. CFGs facilitate comprehensive analyses including dominance analysis, reaching definitions, and live-variable analysis. These analyses are crucial for advanced optimization strategies and efficient code generation for architectures like RISC-V.

In summary, Linear IR, encompassing Stack-machine Code, Three-address Code, and the construction of Control Flow Graphs, plays a crucial role in the backend compilation phases, significantly impacting the optimization and performance of compiled code for target architectures such as RISC-V.

7. Symbol Tables

Symbol tables are critical data structures utilized throughout the compilation process to store and manage information about identifiers (such as variables, functions, types, and other entities) encountered in the source code. These tables assist compilers in keeping track of crucial attributes associated with identifiers, including their scope, data types, and memory locations. Symbol tables play a significant role during semantic analysis, aiding in tasks such as name resolution, type checking, and memory allocation.

Name Resolution and Scope

Name resolution refers to the process of associating identifiers found in source code with their corresponding declarations or definitions. The symbol table facilitates name resolution by clearly organizing and maintaining identifiers based on their scope. Each programming language has specific rules for scope management (such as lexical scope or dynamic scope), and symbol tables are essential in enforcing these rules accurately.

Consider a simple example in C:

```
int global_var = 100;

void exampleFunction() {
    int local_var = 20;
    printf("%d, %d", global_var, local_var);
}
```

The symbol table distinguishes between the global variable `global_var` and the local variable `local_var` within the function. It ensures the correct identification and referencing of these variables during compilation.

Implementing Symbol Tables

Symbol tables are often implemented using efficient data structures such as hash tables or balanced trees (AVL trees, red-black trees). Hash tables are widely used due to their average-case constant-time complexity for insertion, deletion, and lookup operations, making them highly efficient for compiler implementations handling large codebases.

Example symbol table structure:

Identifier	Type	Scope	Memory Address
global_var	int	global	0x1000
local_var	int	local	0x2000

Such a simple representation allows easy tracking and retrieval of relevant attributes, facilitating semantic analysis and optimizations.

Namespaces and Memory Placement

Namespaces help organize identifiers into logical groups, preventing naming conflicts across different modules or scopes. Symbol tables manage namespaces by categorizing identifiers based on visibility and context. For example, two functions in separate modules can have identical names without conflict due to separate namespaces.

Memory placement decisions are guided by symbol tables during compilation. Symbol tables assist in determining where variables should be allocated in memory—whether on the stack, heap, or static/global memory regions. For example:

- **Stack Allocation:** Local variables and function parameters are usually allocated on the stack.
- **Heap Allocation:** Dynamically allocated variables (such as those created using `malloc` in C) reside on the heap.
- **Global/Static Allocation:** Variables defined at a global or static scope are stored in global or static memory.

Symbol tables provide compilers with essential information for managing these memory allocation decisions efficiently and correctly, thus ensuring optimal program execution.

---

8. LLVM and Intermediate Representation

LLVM (Low-Level Virtual Machine) has emerged as a leading compiler infrastructure, largely due to its powerful and flexible Intermediate Representation (LLVM IR). LLVM IR provides an independent, low-level programming language capable of representing programs from various source languages and targeting multiple hardware architectures, including RISC-V.

LLVM IR is designed in Static Single Assignment (SSA) form, a key feature where each variable is assigned exactly once throughout the program. SSA form simplifies data flow analysis, enabling compilers to efficiently perform sophisticated optimizations such as constant propagation, global value numbering, and dead-code elimination. This representation is crucial for compiler frameworks aiming to achieve portability and performance simultaneously.

LLVM IR Example

Consider the following LLVM IR snippet representing a multiply-and-add operation:

```
define i32 @multiply_add(i32 %a, i32 %b, i32 %c) {
entry:
    %mul = mul i32 %a, %b
    %add = add i32 %mul, %c
    ret i32 %add
}
```

In this snippet, the LLVM IR explicitly describes each computational step clearly, making the flow of data and dependencies transparent. This explicit nature facilitates compiler optimizations and direct mapping onto target architectures like RISC-V.

Common LLVM Commands for IR Manipulation

LLVM provides various command-line tools for manipulating IR, summarized in the table below:

Command	Description
`clang`	Front-end compiler that translates source code (e.g., C, C++) into LLVM IR.
`opt`	Applies various optimization passes on LLVM IR, improving code efficiency.
`llc`	Compiles LLVM IR into target-specific assembly code (e.g., RISC-V).
`llvm-dis`	Converts LLVM bitcode (`.bc`) files to human-readable LLVM IR (`.ll`).
`llvm-as`	Converts human-readable LLVM IR (`.ll`) back into bitcode format (`.bc`).
`lli`	Executes LLVM IR or bitcode directly without compiling it into machine code, using an interpreter.
`llvm-link`	Combines multiple LLVM IR bitcode files into one single bitcode file.
`llvm-extract`	Extracts individual functions or global variables from LLVM IR bitcode files.
`llvm-nm`	Lists symbols within LLVM IR bitcode files, useful for inspection and debugging.

These tools streamline the IR manipulation process, allowing developers to efficiently analyze, optimize, and generate executable code.

A typical LLVM IR workflow involves:

1. **Compilation to LLVM IR:**

```
clang -emit-llvm -S example.c -o example.ll
```

2. **IR Optimization:**

```
opt -O2 example.ll -o optimized.bc
```

3. **Code Generation (RISC-V Assembly):**

```
llc -march=riscv64 optimized.bc -o example.s
```

This process illustrates the seamless and modular nature of LLVM IR in supporting efficient compilation workflows tailored for RISC-V architectures.

9. Conclusion

Intermediate Representation (IR) is a foundational component in modern compiler design, bridging the gap between high-level languages and low-level machine-specific instructions. Its pivotal role in facilitating structured, efficient, and highly optimized compilation processes cannot be overstated. By providing an abstraction that separates concerns, IR greatly simplifies the compilation pipeline, enabling compilers to manage multiple source languages and multiple target architectures efficiently and effectively.

Graphical IR forms, such as Abstract Syntax Trees (ASTs), Directed Acyclic Graphs (DAGs), and Control Flow Graphs (CFGs), allow compilers to perform structure-aware optimizations, greatly enhancing code quality at the semantic level. Conversely, Linear IR forms such as Stack-machine Code and Three-address Code facilitate low-level optimizations, which are crucial for generating efficient and performant executable code tailored to specific hardware architectures like RISC-V.

LLVM's approach to IR exemplifies the ideal balance of structure, flexibility, and efficiency. Its Static Single Assignment (SSA) form simplifies complex optimization tasks, such as register allocation and constant propagation, providing powerful tools for compiler developers targeting RISC-V and other architectures. Understanding IR deeply equips compiler engineers with essential insights necessary for creating robust, portable, and highly optimized compilers.

In conclusion, the knowledge and proper use of Intermediate Representations are indispensable in compiler construction, significantly impacting the quality, efficiency, and versatility of generated code, particularly for advanced and performance-critical architectures such as RISC-V.

---

10. References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools.* Pearson Education.
- Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler (2nd Edition).* Elsevier Science.
- LLVM Compiler Infrastructure Documentation. Retrieved from: https://llvm.org/docs/
- Patterson, D., & Waterman, A. (2017). *The RISC-V Reader: An Open Architecture Atlas.* Strawberry Canyon.
- Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

These foundational texts and resources provide comprehensive information for those interested in further exploring compiler design, intermediate representations, and optimizations tailored specifically to modern architectures such as RISC-V.