Lecture 1:
Overview of Compilation

CESE4085 Modern Computer Architecture Course
Part 2, L5.1
Carlo Galuzzi

TUDelft

LECTURER

Dr. Carlo Galuzzi

c.galuzzi-2@tudelft.nl

Department of Quantum and Computer Engineering

LECTURE CONTENTS

- Introduction to Compilers and Interpreters
- Compiler Structure: Front-End, Optimizer, Back-End
- Language Translation Methods
- Phases of Compilation
- A Real-World Application of Compilers

LEARNING GOALS:

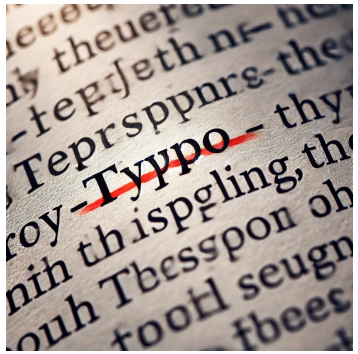
- Understand the role, importance, and structure of compilers in software engineering.
- Distinguish between compilers, interpreters, and different language translation methods.
- Describe the primary phases in compilation: lexical analysis, parsing, semantic analysis, intermediate representation, optimization, and code generation.
- Recognize practical applications and interdisciplinary connections within compiler design.

DISCLAIMER

- This (part of the) course on "***Compilers***" is limited to 8 lectures. It is more like an "***Introduction to Compilers***".
- Due to time constraints, not all compiler topics can be covered in depth.
- Some content may overlap with material from other related courses.
- While the course covers compilers in general, we will also focus on aspects of RISC-V architectures.
- The course is designed to provide a foundational overview; further study is recommended for a comprehensive understanding. See **Brightspace** for extra reading material.

TYPOS AND REWARDS

- These slides and the following ones may contain typos.
- If you spot any, send me an email.
- A free coffee (at the machine) for the first person who catches a typo.
- Stop by my office to get the card for your free coffee.



1st Person

EXTRA READING MATERIAL IN BRIGHSPACE

RISC-V and Its Benefits in Relation to LLVM

Table of Contents

1. Introduction

RISC-V is an open-source **Reduced Instruction Set Computing (RISC)** architecture that offers a flexible and extensible alternative to proprietary instruction sets like x86 and ARM. Since its inception, RISC-V has gained traction across industries, including **embedded systems, artificial intelligence, high-performance computing, and academia**. Its **modular, royalty-free** nature makes it an attractive choice for organizations looking to develop **custom, high-performance processors**.

One of the key differentiators of RISC-V is its **openness**, which fosters innovation and collaboration among companies, research institutions, and independent developers. Unlike traditional ISAs that require licensing fees and proprietary hardware, RISC-V allows unrestricted modification and implementation, making it particularly beneficial for emerging startups and low-budget projects.

LLVM (Low-Level Virtual Machine) plays a critical role in the software ecosystem of RISC-V by providing **compiler infrastructure, optimizations, and toolchain support**. By leveraging LLVM, developers can take advantage of modern compiler techniques, enabling high-performance execution and cross-platform compatibility. This document explores RISC-V's **features, advantages, and its integration with LLVM**, highlighting its growing impact in modern computing.

Introduction to LLVM for RISC-V Development

Table of Contents

1. History and Benefits of LLVM

1.1 History of LLVM

The LLVM (Low-Level Virtual Machine) project was originally developed in 2000 at the University of Illinois at Urbana-Champaign by Chris Lattner as part of his Ph.D. research. The project aimed to create a modular, reusable, and extensible compiler infrastructure that could optimize code at compile-time, link-time, and runtime. LLVM quickly gained popularity in both academia and industry due to its flexibility and strong optimization capabilities. Over time, it evolved from a research project into a widely-used production compiler framework that underpins many modern compilers, including Clang (the default C/C++ compiler for macOS) and Rust's `rustc`.

Initially designed as an alternative to traditional static compilers, LLVM introduced a novel approach that separated the compilation pipeline into distinct stages. This modularity allowed developers to reuse components for different tasks, such as Just-In-Time (JIT) compilation, ahead-of-time (AOT) compilation, and program analysis. As a result, LLVM became a key technology for programming language development, enabling the creation of compilers for Swift, Julia, and other modern languages.

LLVM's success led to industry-wide adoption, with companies such as Apple, Google, and Intel integrating LLVM into their development workflows. Its open-source nature, combined with an active development community, ensures that LLVM continues to evolve with advancements in computer architecture and software engineering.

Introduction to ILOC and Its Relationship to LLVM

Table of Contents

1. Introduction

ILOC (Intermediate Language for Optimizing Compilers) is a hypothetical assembly-like intermediate representation (IR) often used in academic settings to teach compiler design and optimization techniques. It is designed to be simple yet powerful enough to illustrate key concepts such as instruction selection, register allocation, and various optimization strategies. Unlike real-world machine languages, ILOC is machine-independent, making it an ideal teaching tool for students and researchers who need a clear and structured way to understand compiler internals.

On the other hand, LLVM (Low-Level Virtual Machine) is a widely used, industry-standard compiler infrastructure that includes its own intermediate representation, LLVM IR. LLVM IR serves as a bridge between high-level programming languages and machine code, enabling extensive optimizations and portability across multiple hardware architectures. While ILOC is primarily a conceptual and pedagogical tool, LLVM IR is a practical and robust system used in real-world compiler implementations.

Understanding ILOC provides a strong foundation for learning compiler design and optimization techniques, while LLVM IR extends these principles into an industry-grade framework. This document explores the structure of ILOC, its role in compiler construction, and its relationship with LLVM IR.

2. What is ILOC?

ILOC is an abstract, three-address code (TAC) representation often used in compiler courses to illustrate essential compiler principles. Unlike actual machine languages, which are tied to specific hardware architectures, ILOC remains abstract and machine-independent. This abstraction allows students and researchers to focus on high-level compiler optimizations without being distracted by low-level hardware details.

- RISC-V and Its Benefits in Relation to LLVM
- Introduction to LLVM for RISC-V Development
- Introduction to ILOC and Its Relationship to LLVM

INTRODUCTION TO COMPILERS AND INTERPRETERS

WHAT IS A COMPILER?

Before delving into the details of how compilers work, we must have a clear definition of the basic concept of compilers, focusing on their input, output, and high-level purpose.

Definition: A compiler is a program that translates code from a **source language** into a **target language**—most commonly into machine instructions for a specific processor (e.g., RISC V).

Core Purpose: A compiler preserves the *meaning* of the original code while adapting it to a new representation for efficient execution.

Examples:

- C/C++ compilers (translating C/C++ to x86 or ARM assembly).
- Source-to-source translators (e.g., high-level code to C for wider portability).

IMPORTANCE OF SOFTWARE & COMPILERS

Compilers are not just academic curiosities; they are integral to practically every system. Compilers deeply influence the entire software ecosystem, from operating systems to web browsers, and everything in between.

Pervasive Role of Software

- Computers appear in communications, entertainment, transportation, healthcare, etc.
- Nearly all this software depends on compilation, at least in some capacity.

Abstraction Layers

- High-level programming languages provide an efficient interface for developers.
- Compilers bridge the gap by translating these high-level abstractions to low-level operations.

AHEAD-OF-TIME (AOT) COMPILERS

One approach to generating executable code is to do all the work up front—called *ahead-of-time* (AOT) compilation.

Classic Model: The compiler translates source code into a complete, stand-alone machine-code binary *before* execution.

Advantages

- **Performance**: No runtime compilation overhead once the build is done.
- **Separate Build Step**: Developers catch errors and warnings in a dedicated compilation phase.

Use Cases: Traditional languages like C, C++, and Fortran rely on AOT compilation for speed and portability.

JUST-IN-TIME (JIT) COMPILERS

In contrast to AOT compilers, Just-In-Time (JIT) compilers handle translation dynamically during program execution.

Overview: A JIT compiler delays compilation until runtime, using execution profiles to guide optimization.

Dynamic Optimization: It identifies “hot” code paths and invests more effort optimizing them, improving performance where it matters most.

Trade-Off:

- **Costs**: Some overhead from compiling code during execution.
- **Benefits**: Potentially much faster code in performance-critical sections.

Examples: Java Virtual Machine and JavaScript .

INTERPRETERS VS. COMPILERS

While compilers produce an executable, **interpreters** directly execute code instructions. Many modern systems combine both methods in interesting ways.

Interpreter

- Executes code line by line or statement by statement.
- Often simpler to implement but may be slower at runtime.

Compiler

- Produces a final, separate artifact (e.g., machine code, bytecode).
- Typically yields higher execution speed once compilation is done.

Hybrid Approaches: Many modern runtimes (e.g., Java, Python) mix interpretation and compilation.

REAL-WORLD MIXED MODELS

Many prominent languages use a mixed strategy, generating an intermediate form (like bytecode) that can then be either interpreted or further compiled. How these methods balance portability and performance?

Bytecode + Virtual Machine

- Java: Source → Bytecode → Virtual Machine (interpreter or JIT).
- Python: Source → .pyc bytecode → Interpreted by CPython or JIT by alternative implementations.

Why Mix? Combine **portability** (bytecode can run anywhere) with **runtime optimization** (JIT can speed up frequently used parts).

WHY STUDY COMPILER CONSTRUCTION?

Compiler construction stands at the intersection of multiple computer science fields.

Pedagogical Perspective

- Integrates formal language theory, algorithms, data structures, software engineering, and more.
- A “capstone” project that demands a broad range of technical skills.

Practical Perspective

- Understanding compilers helps developers optimize performance and troubleshoot code-generation issues.
- Empowers developers to reason about the true costs of language features.

EVOLVING CHALLENGES & TRENDS

Compiler technology never stands still. Hardware evolves, languages add new features, and programmers push for better performance or portability. How compilers adapt and keep innovating?

Hardware Changes: Multi-core, out-of-order execution, complex memory hierarchies. Compilers must target parallelism, scheduling, and cache optimizations.

Language Features: Dynamic typing, late binding, reflection, and other complexities raise new challenges for compilers.

Diverse Environments: From embedded systems to large-scale data centers, each context demands specialized approaches.

KEY TAKEAWAYS

Compilers: Foundation of Modern Software

- Virtually all code is compiled or translated in some way.
- Compilers profoundly impact performance, portability, and security.

AOT vs. JIT vs. Interpretation: Each approach has unique strengths and trade-offs in speed, memory use, and developer convenience.

Ongoing Research & Innovation: New architectures, language paradigms, and optimization strategies keep compiler research moving forward.

COMPILER STRUCTURE

OVERVIEW OF COMPILER STRUCTURE (1)

A compiler is more than just a “black box” converting source code into machine code. Compilers are broken into distinct phases, each addressing different challenges.

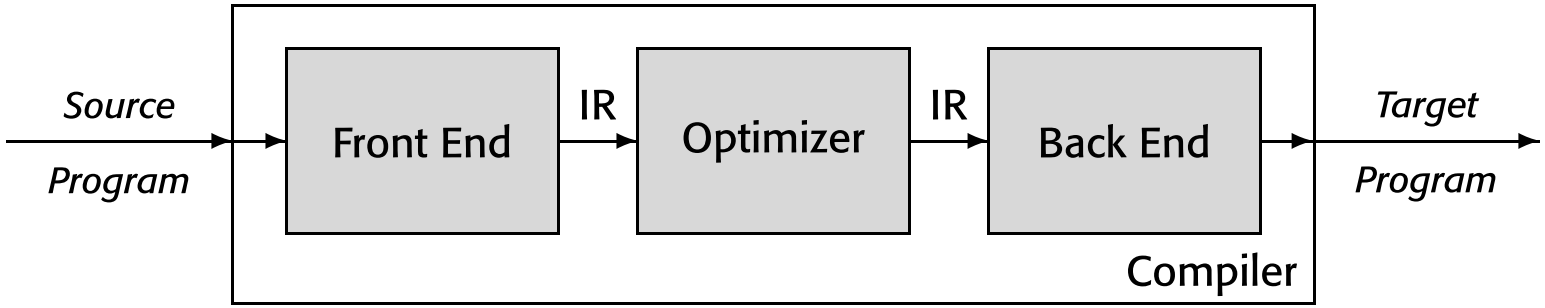
A Large, Complex Software System

- Must handle language-specific rules and machine-specific constraints.
- Leverages formal theories, algorithms, and heuristic approaches.

OVERVIEW OF COMPILER STRUCTURE (2)

Typical Three-Phase View

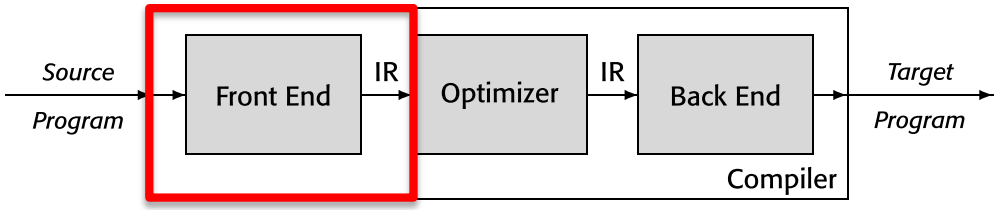
- **Front End**: Understands and validates the source language.
- **Optimizer (Middle End)**: Improves and rewrites code in an intermediate representation (IR).
- **Back End**: Adapts and generates code for the target machine.



Historical Evolution

- Decades of compiler development have led to a phased, pass-based structure.
- Separates concerns into manageable components.

FRONT END – SOURCE CODE ANALYSIS (1)



The front-end is where the compiler gains a deep understanding of the source program. It checks correctness and creates a structured representation of the code.

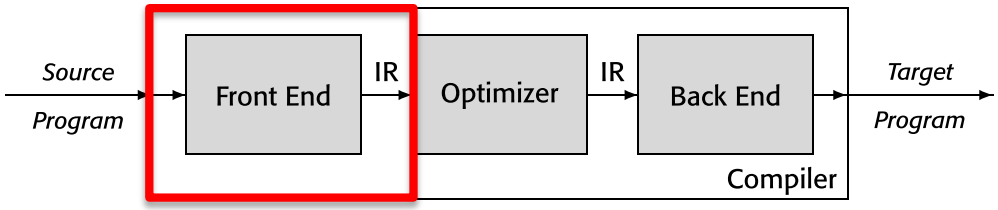
Lexical & Syntactic Checks

- **Scanning**: Converts raw text into tokens.
- **Parsing**: Builds a grammatical structure to verify syntax.

Semantic Elaboration

- Type checking, scope resolution, and consistency checks.
- Detects deeper errors that grammars alone can’t catch (e.g., type mismatches).

FRONT END – SOURCE CODE ANALYSIS (2)

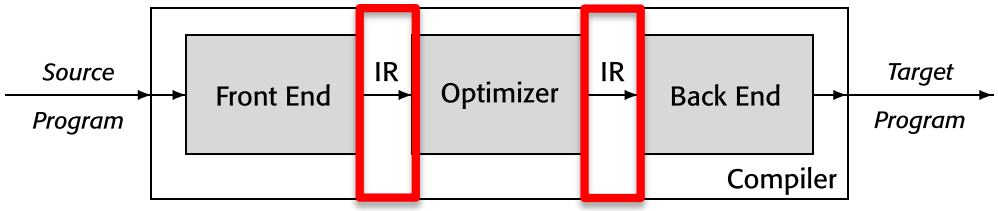


The front-end is where the compiler gains a deep understanding of the source program. It checks correctness and creates a structured representation of the code.

IR Generation

- Translates source constructs into an internal, language-independent form.
- Prepares the code for optimization and machine-dependent steps.

INTERMEDIATE REPRESENTATION (IR) (1)

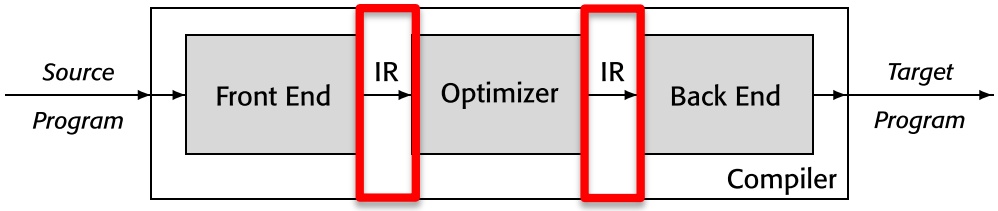


IR is the central data structure that bridges the front end, optimizer, and back end. It carries all the relevant details of the program in a form amenable to both analysis and transformation.

Language-Neutral

- Allows compilers to connect different front ends (languages) to a single back end (target).
- Encodes each operation or statement in a standardized format.

INTERMEDIATE REPRESENTATION (IR) (2)



IR is the central data structure that bridges the front end, optimizer, and back end. It carries all the relevant details of the program in a form amenable to both analysis and transformation.

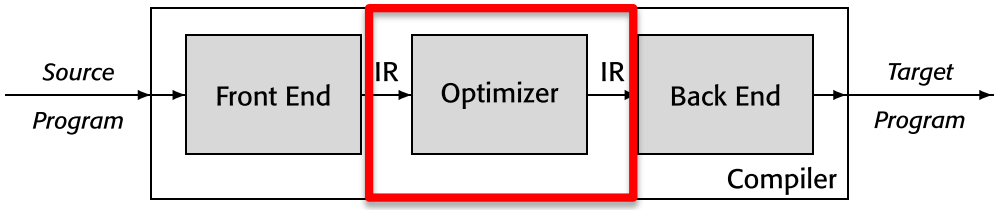
Multiple IR Forms

- Graph-based representations (e.g., control-flow graphs).
- Low-level, sequential forms (similar to assembly).
- High-level, abstract forms for advanced optimizations.

Importance in Collaboration

- Clear interfaces between phases.
- Simplifies debugging; each pass works on a well-defined IR structure.

THE OPTIMIZER – MIDDLE PHASE (1)

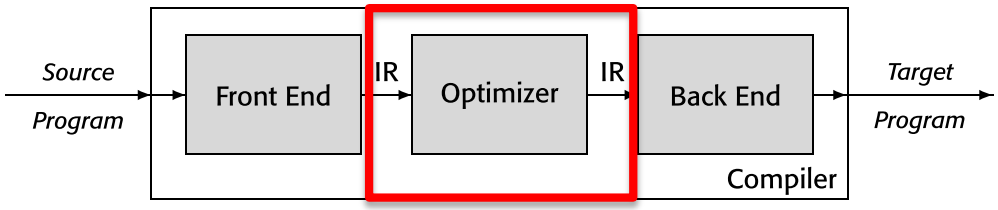


The middle end (or optimizer) tries to make the IR "better" by applying transformations that preserve correctness while improving execution speed, code size, or energy usage.

Analysis + Transformation

- **Analysis**: Data-flow analysis, dependence checks, pointer analysis.
- **Transformation**: Loop optimizations, constant folding, dead code elimination.

THE OPTIMIZER – MIDDLE PHASE (2)



The middle end (or optimizer) tries to make the IR "better" by applying transformations that preserve correctness while improving execution speed, code size, or energy usage.

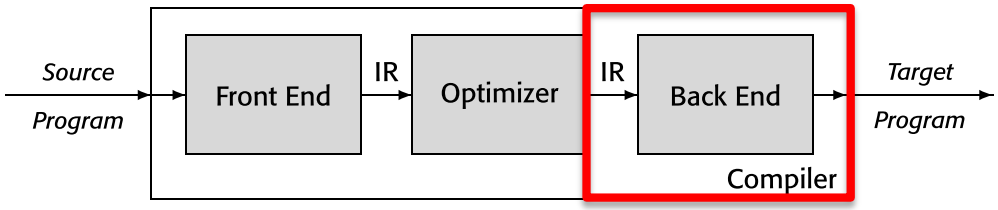
Multiple Passes

- Each pass refines the IR step by step.
- May reanalyse after certain transformations to discover new opportunities.

No Guaranteed Optimality

- Many optimization problems are **NP-complete** or too large to solve exactly.
- Heuristics/approximations often yield large performance gains.

THE BACK END – TARGET ADAPTATION (1)

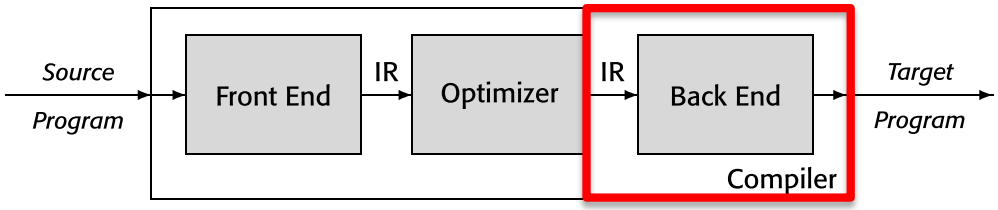


After optimization, the compiler must convert IR instructions into the final machine code. This phase grapples with hardware realities like finite registers and specific instruction sets.

Instruction Selection

- Maps IR operations to the most appropriate machine instructions.
- May exploit specialized instructions (e.g., multiply-accumulate).

THE BACK END – TARGET ADAPTATION (2)

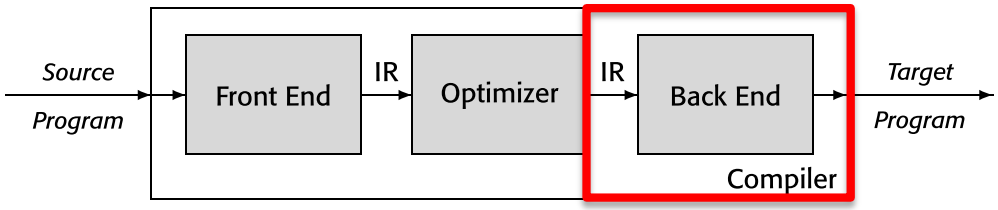


After optimization, the compiler must convert IR instructions into the final machine code. This phase grapples with hardware realities like finite registers and specific instruction sets.

Register Allocation

- Maps unlimited “virtual registers” in IR to the hardware’s limited physical registers.
- Spills extra values to memory when needed.

THE BACK END – TARGET ADAPTATION (3)

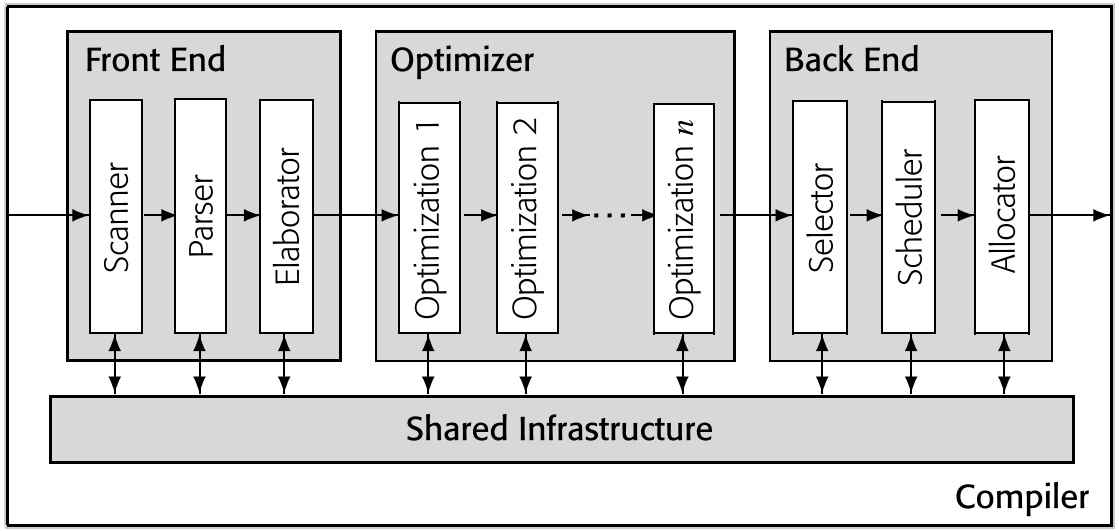


After optimization, the compiler must convert IR instructions into the final machine code. This phase grapples with hardware realities like finite registers and specific instruction sets.

Instruction Scheduling

- Reorders operations to hide latencies (e.g., memory access, pipeline stalls).
- Balances parallelism, data dependencies, and resource constraints.

PASSES VS. PHASES (1)

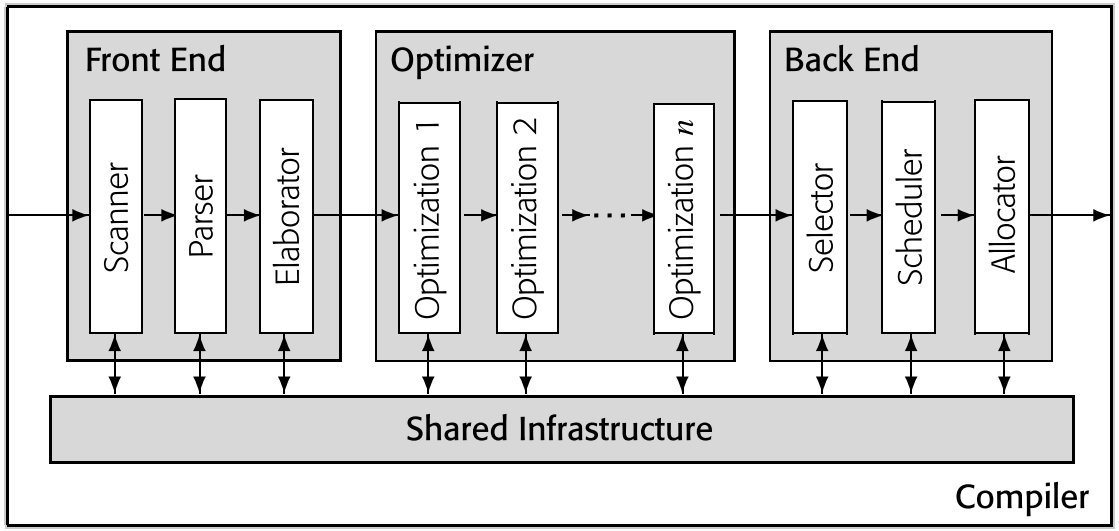


While the "front end–optimizer–back end" model is conceptually neat, each phase can be subdivided into multiple passes that perform specific tasks.

Phases

- **Front End**: Lexical analysis, parsing, semantic checks, IR building.
- **Optimizer**: Repetitive transformations, data-flow analyses.
- **Back End**: Instruction selection, register allocation, scheduling.

PASSES VS. PHASES (2)

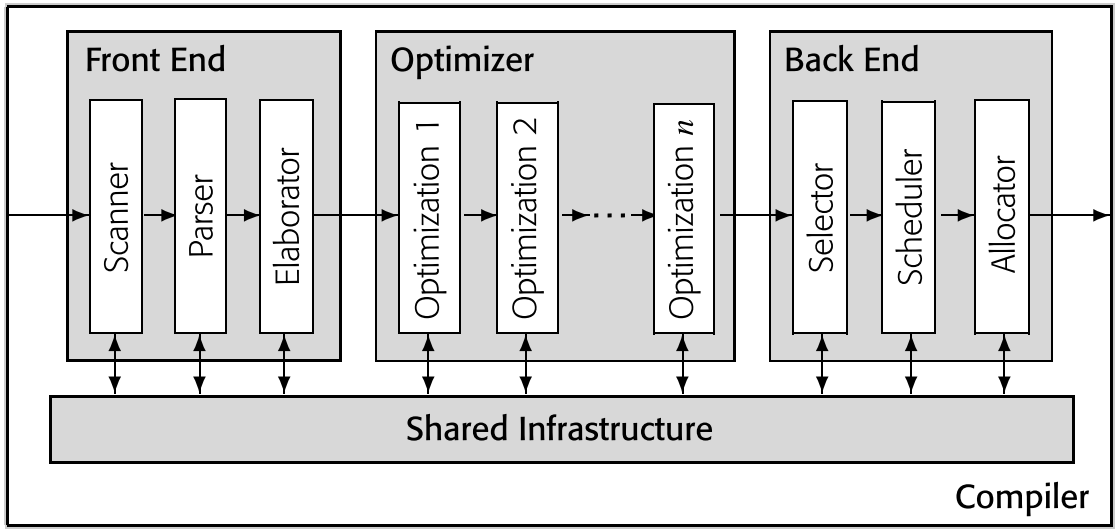


While the “front end–optimizer–back end” model is conceptually neat, each phase can be subdivided into multiple passes that perform specific tasks.

Passes

- Each phase can have many passes.
- Passes communicate via IR, ensuring each pass sees a consistent program representation.

PASSES VS. PHASES (3)



While the "front end–optimizer–back end" model is conceptually neat, each phase can be subdivided into multiple passes that perform specific tasks.

Benefits

- Easier to implement and debug each pass in isolation.
- Greater flexibility: can insert or remove passes without disturbing the entire compilation flow.

DEBUGGING & TESTING THROUGH IR

Because compilers are inherently complex, maintaining correctness across so many steps is challenging. The IR-centric design helps with incremental testing and validation.

Validating IR

- Checking each pass's output for consistency (type correctness, control-flow structure).
- Detecting transformations that introduce errors early in the pipeline.

Simplifies Debugging

- Each pass can be tested in isolation, producing or consuming IR.
- Tools can visualize or dump IR for inspection.

Ease of Maintenance

- Well-defined IR interfaces reduce unintended interactions between passes.
- Facilitates modular design and long-term maintainability.

SEPARATE COMPILATION (1)

Large software projects rarely compile their entire codebase in a single step. Separate compilation allows building and linking multiple modules independently.

Definition

- Compile distinct source files independently into intermediate or object files.
- Link the resulting object files to form the final executable.

Advantages

- Saves compile time: Only recompile changed files.
- Enables libraries: Precompiled code distributed in binary form.
- Multiple developers can work in parallel on different parts of a large project.

SEPARATE COMPILATION (2)

Large software projects rarely compile their entire codebase in a single step. Separate compilation allows building and linking multiple modules independently.

Implications for the Compiler

- Must manage partial knowledge of external function calls or global variables.
- Usually relies on a well-defined **Application Binary Interface (ABI).**

LIMITATIONS OF "OPTIMIZATION" (1)

Although compilers apply "optimizations," the term can be misleading because absolute optimality is seldom achievable. The compiler aims to make code "better," not perfectly optimal.

Why Not Truly Optimal?

- Problems like scheduling and register allocation are NP-complete or extremely large.
- Compiler writers rely on heuristics, approximations, or partial solutions.

Realistic Goals

- Achieve *noticeable* performance improvements or code-size reductions.
- Avoid increasing compile time to impractical levels.

LIMITATIONS OF “OPTIMIZATION” (2)

Although compilers apply “optimizations,” the term can be misleading because absolute optimality is seldom achievable. The compiler aims to make code “better,” not perfectly optimal.

Continuous Process

- As hardware and languages evolve, new optimization techniques are developed.
- Compiler research constantly refines these methods for better results.

TAKEAWAYS

A Phased, Pass-Based Approach

- Organizes a complex task into manageable segments: front end, optimizer, back end.
- Each segment has its own responsibilities and data structures.

Intermediate Representation (IR) as a Crucial Glue

- Standardizes communication between passes.
- Facilitates code analysis, transformation, and generation.

Balancing Complexity & Efficiency

- Compiler design involves trade-offs in speed, memory, and code quality.
- A well-structured compiler is easier to extend, debug, and maintain.

LANGUAGE TRANSLATION AND PHASES OF COMPILATION

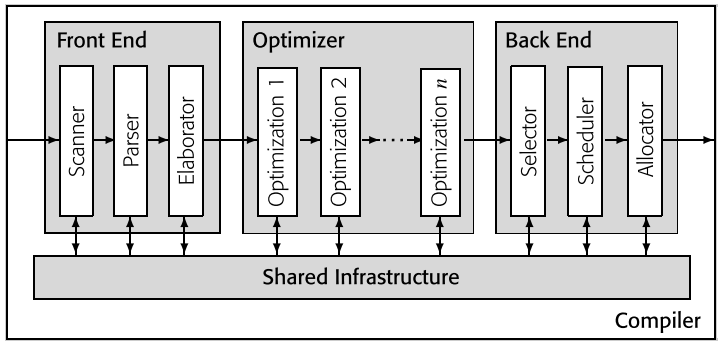
OVERVIEW OF TRANSLATION (1)

Let's see how a compiler takes high-level source code and processes it through several stages—front end, optimization, and back end—to produce efficient machine code.

This section uses a running example to illustrate how compilers transform and optimize code.

Main Concept

- Compilers systematically convert source language code into a form executable on a target machine.
- This "big picture" consists of multiple steps: validation, IR generation, optimization, and code generation.



OVERVIEW OF TRANSLATION (2)

Let's see how a compiler transforms high-level language constructs into optimized machine code. This "big-picture" view clarifies how the front end, optimizer, and back end work together on a single piece of code, using the recurring example

$$a \leftarrow a \times 2 \times b \times c \times d$$

and a loop scenario.

Key Themes

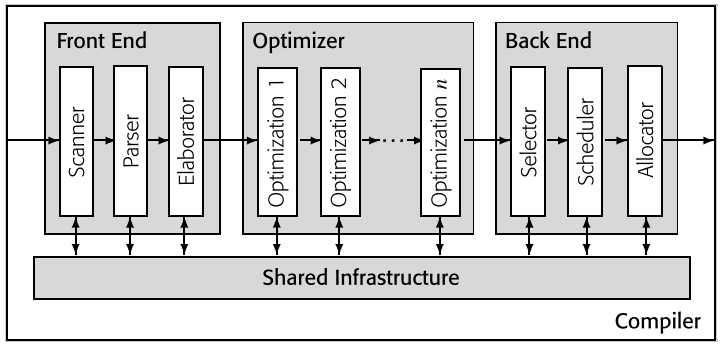
- The front end ensures the code is valid and converts it into an **intermediate representation (IR)**.
- The optimizer refines the IR to remove inefficiencies and improve performance.
- The back end generates machine-dependent code and handles final optimizations (e.g., instruction scheduling).

THE BIG PICTURE OF TRANSLATION (1)

A compiler's translation pipeline can be viewed as three major phases, each dealing with distinct tasks. These phases interact through well-defined representations and data structures.

Three Main Phases

1. **Front End**: Parses and semantically checks the source program, producing an IR.
2. **Optimizer** (Middle End): Analyzes and transforms the IR to make it "better" (faster, smaller, etc.).
3. **Back End**: Maps the IR onto the target machine's instruction set, accounting for hardware limits.

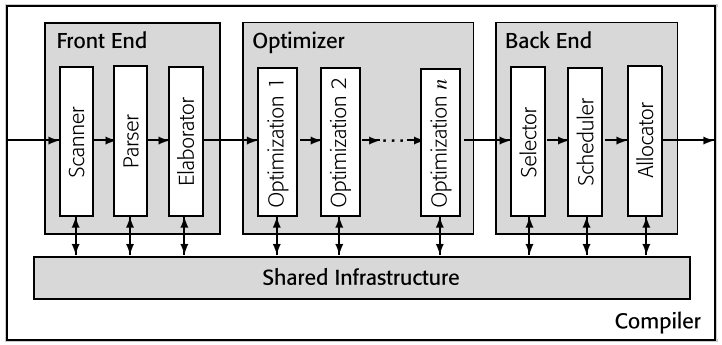


THE BIG PICTURE OF TRANSLATION (2)

A compiler's translation pipeline can be viewed as three major phases, each dealing with distinct tasks. These phases interact through well-defined representations and data structures.

Why This Structure?

- **Separation of Concerns**: Easier to maintain, debug, and evolve.
- **IR as a Bridge**: Allows front end and back end to remain relatively independent.
- **Multiple Passes**: Each sub-phase (e.g., instruction scheduling) can be tuned without disrupting others.



INITIAL EXPRESSION – a ← a × 2 × b × c × d

Even short code snippets can highlight a variety of compiler strategies!

Expression Details

- Source snippet: `a ← a × 2 × b × c × d`
- Involves multiple multiplications and an assignment back to `a`.

Front-End Challenges

- **Lexical analysis**: Distinguishes identifiers (`a`, `b`, `c`, `d`) from the operator `×` and constant `2`.
- **Parsing**: Ensures the expression conforms to the grammar.
- **Semantic checks**: Verifies `a`, `b`, `c`, `d` are valid variables, checks types, etc.

GENERATING AN IR (LOW-LEVEL SEQUENTIAL FORM) (1)

Once the front end deems the expression correct, it produces an **intermediate representation (IR)** to capture the semantics in a machine-neutral way.

$$
\begin{aligned}
t_0 &\leftarrow a \times 2 \\
t_1 &\leftarrow t_0 \times b \\
t_2 &\leftarrow t_1 \times c \\
t_3 &\leftarrow t_2 \times d \\
a &\leftarrow t_3
\end{aligned}
$$

Why This Format?

- Breaks down the expression into simpler steps (one operator per instruction).
- Uses temporary “virtual registers” (t0, t1, t2, t3) to store intermediate results.
- Facilitates subsequent compiler analyses (data-flow, optimization).

GENERATING AN IR (LOW-LEVEL SEQUENTIAL FORM) (2)

Once the front end deems the expression correct, it produces an **intermediate representation (IR)** to capture the semantics in a machine-neutral way.

$$
\begin{aligned}
t_0 &\leftarrow a \times 2 \\
t_1 &\leftarrow t_0 \times b \\
t_2 &\leftarrow t_1 \times c \\
t_3 &\leftarrow t_2 \times d \\
a &\leftarrow t_3
\end{aligned}
$$

Compiler Benefits

- IR is easier to inspect for redundant calculations.
- Language-agnostic (a stepping stone before generating code for different targets).

CONTEXT MATTERS – PLACING THE EXPRESSION IN A LOOP (1)

Where the expression appears in the program can expose additional optimization opportunities—especially in a loop.

```
b ← ···
c ← ···
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
    end
```

Observation

- `b`, `c`, and `2` do *not* change during the loop (loop-invariant).
- Recomputing them every iteration leads to extra instructions and overhead.

CONTEXT MATTERS – PLACING THE EXPRESSION IN A LOOP (2)

Where the expression appears in the program can expose additional optimization opportunities—especially in a loop.

```
b ← ···
c ← ···
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
    end
```

Implication

- An optimizer can detect and "hoist" loop-invariant expressions out of the loop to reduce total work.

LOOP-INVARIANT CODE MOTION – THE OPTIMIZED LOOP (1)

Loop-invariant code motion is a classic optimization that relocates calculations outside the loop if they remain constant across iterations.

Before (Inside the Loop)

- `a ← a × 2 × b × c × d`
- Results in four multiplications every iteration.

LOOP-INVARIANT CODE MOTION – THE OPTIMIZED LOOP (2)

Loop-invariant code motion is a classic optimization that relocates calculations outside the loop if they remain constant across iterations.

After (Partially Hoisted)

```
b ← ...
c ← ...
a ← 1
t ← 2 × b × c

for i = 1 to n
    read d
    a ← a × d × t
    end
```

Performance Gains

- **Original**: 4 multiplications × `n` iterations → `4n` total.
- **Optimized**: 2 multiplication outside, 2 per iteration → `2n + 2` total.
- Dramatic improvement for large `n`.

BACK END – MAPPING IR TO TARGET INSTRUCTIONS (1)

After optimization, the compiler translates the IR into an instruction set for a specific machine. This is illustrated with a low-level IR called ILOC (or a similar "assembly-like" format).

```
loadAI   r_arp, @a ⇒ r_a      // load 'a'
loadI    2         ⇒ r_2      // constant 2 into r_2
loadAI   r_arp, @b ⇒ r_b      // load 'b'
loadAI   r_arp, @c ⇒ r_c      // load 'c'
loadAI   r_arp, @d ⇒ r_d      // load 'd'
mult     r_a, r_2  ⇒ r_a      // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a      // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a      // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a      // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a // write r_a back to 'a'
```

ILOC Example (naïve mapping for $a \leftarrow a \times 2 \times b \times c \times d$):

ILOC Operation				Meaning
loadAI	r_1, c_2	⇒	r_3	Memory $(r_1 + c_2) \rightarrow r_3$
loadI	c_1	⇒	r_2	$c_1 \rightarrow r_2$
mult	r_1, r_2	⇒	r_3	$r_1 \times r_2 \rightarrow r_3$
storeAI	r_1	⇒	r_2, c_3	$r_1 \rightarrow$ Memory $(r_2 + c_3)$

BACK END – MAPPING IR TO TARGET INSTRUCTIONS (2)

After optimization, the compiler translates the IR into an instruction set for a specific machine. This is illustrated with a low-level IR called ILOC (or a similar "assembly-like" format).

```
loadAI   r_arp, @a ⇒ r_a      // load 'a'
loadI    2         ⇒ r_2      // constant 2 into r_2
loadAI   r_arp, @b ⇒ r_b      // load 'b'
loadAI   r_arp, @c ⇒ r_c      // load 'c'
loadAI   r_arp, @d ⇒ r_d      // load 'd'
mult     r_a, r_2  ⇒ r_a      // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a      // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a      // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a      // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a // write r_a back to 'a'
```

- Each `loadAI` or `storeAI` accesses memory at a known offset (`@a`, `@b`, etc.).
- `mult` is a two-operand instruction producing the result in a third register.
- Real machines have finite registers, so further steps handle allocation and scheduling.

REGISTER ALLOCATION & MINIMAL REGISTER USE (1)

Compilers initially pretend there's an **unlimited supply** of "virtual registers." Later, a **register allocator** decides which values must stay in real CPU registers and which must spill to memory.

REGISTER ALLOCATION & MINIMAL REGISTER USE (2)

Minimal Register Example

- Instead of using 5 or 6 registers, the compiler can reuse registers:

```
loadAI   r_arp, @a ⇒ r1      // load 'a'
add      r1, r1    ⇒ r1      // r1 ← a × 2
loadAI   r_arp, @b ⇒ r2      // load 'b'
mult     r1, r2    ⇒ r1      // r1 ← (a × 2) × b
loadAI   r_arp, @c ⇒ r2      // load 'c'
mult     r1, r2    ⇒ r1      // r1 ← (a × 2 × b) × c
loadAI   r_arp, @d ⇒ r2      // load 'd'
mult     r1, r2    ⇒ r1      // r1 ← (a × 2 × b × c) × d
storeAI  r1        ⇒ r_arp, @a // write r1 back to 'a'
```

- Fewer registers used, 3 instead of 6, but might hamper scheduling opportunities if we always overwrite r_1 and r_2.
- Minimizing register count can introduce more pipeline stalls later.
- A **sophisticated register allocator** balances **register usage** *and* **performance**.

INSTRUCTION SCHEDULING – DEALING WITH LATENCY

Let's see how reordering instructions can hide or reduce stalls on modern processors. Each operation (`load, mult, store`) might take several cycles; instruction scheduling weaves them together efficiently.

Start	End	Code	Naïve Schedule
1	3	loadAI	r_{arp}, @a ⇒ r_1 // load 'a'
4	4	add	r_1, r_1 ⇒ r_1 // $r_1 \leftarrow a \times 2$
5	7	loadAI	r_{arp}, @b ⇒ r_2 // load 'b'
8	9	mult	r_1, r_2 ⇒ r_1 // $r_1 \leftarrow (a \times 2) \times b$
10	12	loadAI	r_{arp}, @c ⇒ r_2 // load 'c'
13	14	mult	r_1, r_2 ⇒ r_1 // $r_1 \leftarrow (a \times 2 \times b) \times c$
15	17	loadAI	r_{arp}, @d ⇒ r_2 // load 'd'
18	19	mult	r_1, r_2 ⇒ r_1 // $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
20	22	storeAI	r_1 ⇒ r_{arp},@a // write r_1 back to 'a'

IMPROVED SCHEDULING – SHORTENING EXECUTION TIME (1)

By rearranging loads, multiplies, and adds to start earlier, the compiler can overlap operations, drastically cutting down total cycles to 13 (from 22, -40.91%).

Start	End	Code	Optimized Schedule
1	3	loadAI	r_{arp}, @a ⇒ r_1 // load 'a'
2	4	loadAI	r_{arp}, @b ⇒ r_2 // load 'b'
3	5	loadAI	r_{arp}, @c ⇒ r_3 // load 'c'
4	4	add	r_1, r_1 ⇒ r_1 // r_1 ← a × 2
5	6	mult	r_1, r_2 ⇒ r_1 // r_1 ← (a × 2) × b
6	8	loadAI	r_{arp}, @d ⇒ r_2 // load 'd'
7	8	mult	r_1, r_3 ⇒ r_1 // r_1 ← (a × 2 × b) × c
9	10	mult	r_1, r_2 ⇒ r_1 // r_1 ← (a × 2 × b × c) × d
11	13	storeAI	r_1 ⇒ r_{arp}, @a // write r_1 back to 'a'

Nevertheless, the code uses one more register than the minimal number.

IMPROVED SCHEDULING – SHORTENING EXECUTION TIME (2)

Start	End	Code
1	3	loadAI r_{arp}, @a ⇒ r_1 // load 'a'
2	4	loadAI r_{arp}, @b ⇒ r_2 // load 'b'
3	5	loadAI r_{arp}, @c ⇒ r_3 // load 'c'
4	4	add r_1, r_1 ⇒ r_1 // r_1 ← a × 2
5	6	mult r_1, r_2 ⇒ r_1 // r_1 ← (a × 2) × b
6	8	loadAI r_{arp}, @d ⇒ r_2 // load 'd'
7	8	mult r_1, r_3 ⇒ r_1 // r_1 ← (a × 2 × b) × c
9	10	mult r_1, r_2 ⇒ r_1 // r_1 ← (a × 2 × b × c) × d
11	13	storeAI r_1 ⇒ r_{arp},@a // write r_1 back to 'a'

Key Observations

- Loads for b and c start *before* the first multiplication finishes.
- Overlaps long-latency operations (`load`, `mult`) with others.
- Reduces idle time significantly.

Takeaway

- *Instruction scheduling* is crucial on pipelined or superscalar processors.
- It requires knowledge of operation latencies and data dependencies.

PUTTING IT ALL TOGETHER – FINAL CODE PATH (1)

After instruction selection, register allocation, and scheduling, the final code is a carefully woven sequence that balances minimal register usage with parallel execution on the hardware.

Final Step Sequence

1. **Front End**: Convert `a ← a × 2 × b × c × d` into IR.
2. **Optimizer**: Possibly hoist loop invariants, fold constants, or remove dead code, etc.
3. **Back End**:
    - **Instruction Selection**: Convert IR ops to target instructions.
    - **Register Allocation**: Assign real registers, spill if necessary.
    - **Scheduling**: Reorder instructions to reduce stall cycles.

PUTTING IT ALL TOGETHER – FINAL CODE PATH (2)

After instruction selection, register allocation, and scheduling, the final code is a carefully woven sequence that balances minimal register usage with parallel execution on the hardware.

Real-World Complexity

- Modern compilers can have dozens of passes for specialized optimizations.
- Each pass is tested with IR-level checks to ensure correctness.

LESSONS FROM THE EXAMPLE (1)

Even a small expression highlights multiple facets of compiler design—analysis, transformation, and code generation.

Front End

- Ensures correctness (syntax + semantics).
- Produces IR that is simpler to manipulate than raw source.

Loop Invariance & Optimization

- Real speedups often emerge by understanding code context (e.g., loop usage).
- Data-flow analysis identifies which parts of the expression are constant.

LESSONS FROM THE EXAMPLE (2)

Even a small expression highlights multiple facets of compiler design—analysis, transformation, and code generation.

Back-End Complexity

- Instruction selection, register allocation, scheduling are often NP-complete or near.
- Practical compilers rely on heuristics and approximations to produce good code in reasonable time.

TAKEAWAYS

Key Insights

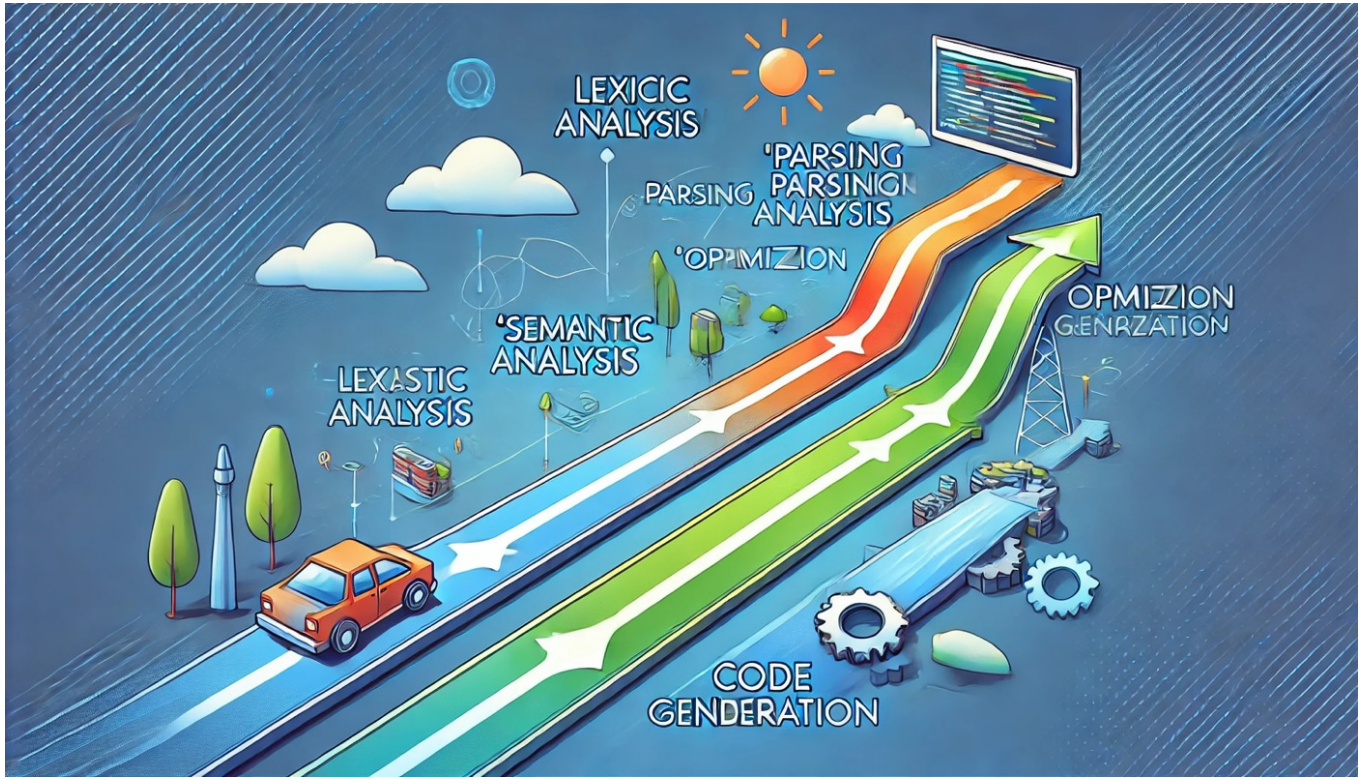
1. **Translation Steps**: Front end → IR → Optimizer → IR → Back end.
2. **Context-Specific Optimization**: Loop invariants can be hoisted to reduce overhead.
3. **Instruction Scheduling & Allocation**: Final code can differ drastically in performance, even if it's functionally identical.

Overall Takeaway

- Compiler design intertwines many techniques, from formal parsing to advanced scheduling.
- The entire process is about refining code while preserving its *meaning*, turning a short expression into *optimal or near-optimal* machine instructions.

AN APPLICATION JOURNEY:

COMPILING THE GCD C PROGRAM INTO RISC-V MACHINE CODE



STEP 1: THE SOURCE CODE (1)

All compilation starts from the programmer's high-level code—in this case, a **C program** that calculates the Greatest Common Divisor (GCD). High-level means human-readable constructs like functions, loops, conditionals, and standard library calls.

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main() {
    int num1 = 56, num2 = 98;
    int result = gcd(num1, num2);
    printf("GCD of %d and %d is %d\n", num1, num2, result);
    return 0;
}
```

STEP 1: THE SOURCE CODE (2)

#include <stdio.h> includes the standard input-output header file (e.g., printf, scanf, etc.)

Conditional: Checks if `b` is zero (if (`b == 0`)), then returns `a`

Recursion: Otherwise calls `gcd(b, a % b)` — the classic Euclidean algorithm.

Function main():

- Declares and initializes `num1` and `num2`.
- Calls `gcd(num1, num2)` and prints the result with `printf`.
- Returns `0` to indicate successful execution.

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main() {
    int num1 = 56, num2 = 98;
    int result = gcd(num1, num2);
    printf("GCD of %d and %d is
            %d\n", num1, num2,
            result);
    return 0;
}
```

Common Pitfall: If a user wrote if (b = 0) instead of if (b == 0), the code would compile incorrectly (assigning 0 to b) and produce a logical error—though the compiler's *semantic analysis* or a warning might catch it.

STEP 1: THE SOURCE CODE (3)

Why Use This GCD Program?

1. **Simplicity with Depth:** It's short and easy to read, but it demonstrates several key language features (e.g., recursion, arithmetic operations, conditionals, and function calls).

2. **Recursion:** The **gcd** function calls itself until `b` is zero, an interesting feature for the compiler to handle, especially when looking at optimization or how function calls are structured in assembly.

3. **Standard Library Call:** The `printf` function shows how the compiler must link in external routines from the C Standard Library. This matters for the final linking stage.

STEP 2: LEXICAL ANALYSIS (1)

Lexical analysis (or *tokenization*) is the first phase that transforms the raw source code into a sequence of **tokens**. A **token** is the smallest unit of meaning in a language—like a keyword (`int`), an identifier (`gcd`), a literal (`56`), an operator (`%`), or punctuation (`(`, `)`).

Reading the Source Text:

- The *lexer* (or *scanner*) reads characters from the source file (.c file).
- It groups these characters into tokens based on the language's rules (e.g., `int` is a keyword, `main` is an identifier).

STEP 2: LEXICAL ANALYSIS (2)

Categories of Tokens:

- **Keywords**: `int`, `return`, `if`—reserved words in the C language.
- **Identifiers**: Names for functions or variables, e.g., `gcd`, `num1`, `num2`.
- **Literals**: Constant values like `56`, `98` (integers here, but could be floats, chars, etc.).
- **Operators**: `=`, `%`, `==`, `,`—symbols that perform operations or comparisons.
- **Punctuation/Delimiters**: `(`, `)`, `{`, `}`, `;`—markers that structure the code.

STEP 2: LEXICAL ANALYSIS (3)

Removing Irrelevant Elements:

- **Whitespace** (spaces, tabs, newlines) is largely ignored unless it's part of distinguishing tokens.
- **Comments** are discarded so they don't appear in the token stream.

Example Token Stream for the gcd function:

```
[int] [gcd] [(] [int] [a] [,] [int] [b] [)] [{]
[if] [(] [b] [==] [0] [)] [return] [a] [;]
[return] [gcd] [(] [b] [,] [a] [%] [b] [)] [;]
[}]
...
```

And similarly for the main function, with tokens like `[int] [main] [(] [)] [{]`, etc.

STEP 2: LEXICAL ANALYSIS - NOTES

- **Invalid Characters**: If there's an unrecognized character (say, a typo or non-ASCII symbol), the lexer can raise an error.

- **Ambiguities**: For instance, the string `int` is recognized as a keyword only if it's isolated. If you had `integerVar`, it's seen as an identifier, not the keyword `int`.

- **Role of the Lexer**: It's purely mechanical—no deep checks of meaning or type correctness. For example, it won't notice if you used `if (b = 0)` instead of `if (b == 0)`; that error surfaces later.

After tokenization, the stream of tokens goes to **Syntax Analysis** (Step 3), which checks if the tokens form valid statements, expressions, and program structures according to the C grammar.

STEP 3: SYNTAX ANALYSIS (PARSING) (1)

Syntax analysis (commonly called *parsing*) is the phase that organizes the token stream into a hierarchical structure called an **Abstract Syntax Tree (AST)**. The AST reveals how tokens fit together according to the grammar rules of the C language—for example, how an if statement encloses its condition and body, or how function parameters are declared.

1. **Input: The Token Stream:** From Step 2, the compiler has a list of tokens. For the GCD code, some of these tokens include:

```
[int] [gcd] [(] [int] [a] [,] [int] [b] [)] [{] ...
```

and so on. These tokens, in sequence, are fed into the **parser**.

STEP 3: SYNTAX ANALYSIS (PARSING) (2)

2. **Grammar and Production Rules:** A *grammar* defines valid patterns of tokens. For instance, a simplified rule for a function definition in C might be:

```
function_definition
        ↓
type identifier "(" parameter_list ")" compound_statement
```

This tells the parser how tokens of certain types (e.g., `int` as the type, `gcd` as the identifier) must appear in the correct order, followed by parentheses and a body `({ ... })`.

STEP 3: SYNTAX ANALYSIS (PARSING) (3)

3. **Constructing the AST:** If the token stream matches valid grammar patterns, the parser builds an **Abstract Syntax Tree**. The AST for our GCD example typically includes:

```
Program
├── Function gcd(a, b)
│    ├── IfStatement (b == 0)
│    │    └── Return a
│    └── Return gcd(b, a % b)
└── Function main()
     ├── Declaration: int num1 = 56, num2 = 98
     ├── Declaration: int result = gcd(num1, num2)
     ├── FunctionCall: printf("GCD of %d and %d is %d\n",
     │                                num1, num2, result)
     └── Return 0
```

STEP 3: SYNTAX ANALYSIS (PARSING) (4)

```
Program
├── Function gcd(a, b)
│    ├── IfStatement (b == 0)
│    │    └── Return a
│    └── Return gcd(b, a % b)
└── Function main()
     ├── Declaration: int num1 = 56, num2 = 98
     ├── Declaration: int result = gcd(num1, num2)
     ├── FunctionCall: printf("GCD of %d and %d is %d\n",
     │                                  num1, num2, result)
     └── Return 0
```

- **Nodes** represent high-level constructs: Function, IfStatement, Return, FunctionCall, etc.
- **Children** show nesting and relationships: Within `gcd(a, b)`, there's an `IfStatement` node with its own child node for the return `a`.
- **Order Matters**: The AST preserves the sequence of operations and statements.

STEP 3: SYNTAX ANALYSIS (PARSING) (5)

4. Error Handling

- If the tokens don't match any rule (e.g., missing parentheses, extra commas), the parser detects a *syntax error*.
- This is different from semantic errors: a syntax error might be "return gcd(b a b);" missing commas, while a semantic error could be a type mismatch.

STEP 3: SYNTAX ANALYSIS (PARSING) (6)

5. Why the AST Matters

- **Intermediate Representation**: The AST is a structured map of the program, making further analyses (semantic checks, optimizations) more systematic.
- **Scope & Nesting**: The AST shows which statements belong together, clarifying scope boundaries (e.g., what variables are declared within a function).

Once the AST is successfully built, the compiler proceeds to **Semantic Analysis** (Step 4). There, it verifies that the constructs in the AST obey the language's deeper rules—like matching function signatures, types, and correct usage of library functions.

STEP 4: SEMANTIC ANALYSIS (1)

- Semantic analysis goes beyond syntax to check *logical correctness* in the AST.

- It ensures the program's constructs follow the rules of the C language—for instance, verifying types, function signatures, and variable declarations.

- By the end of semantic analysis, the compiler must confirm that every node in the AST "makes sense" according to language semantics.

STEP 4: SEMANTIC ANALYSIS (2)

1. **Type Checking**

   Example: `gcd(b, a % b)`

- We must confirm that `b` and `a % b` are both integers, matching `int gcd(int, int)`.
- **Expressions**: If we had something like int `x = 3.14;`, the compiler might allow it with a warning (truncation) or cast, but it's flagged semantically.

Why This Matters:

Ensures, for example, you're not trying to do arithmetic on pointers or pass a `float` to a function expecting an `int` (unless explicitly cast).

STEP 4: SEMANTIC ANALYSIS (3)

2. Scope and Declaration Checks

- Variables must be declared before use.
- If you reference `num1` in `gcd` without passing it as a parameter or declaring it globally, that's an error.
- Functions must be declared or have a prototype before invocation.
- For `gcd`, the definition precedes usage in `main`, so that's valid.

Why This Matters:

Prevents referencing identifiers that the compiler knows nothing about.

STEP 4: SEMANTIC ANALYSIS (4)

3. Function Signatures and Parameters

- **gcd** returns an integer and takes two integers. The compiler checks that the return type (`return a;`, `return gcd(...)`) is correct and that calls to `gcd` pass the proper arguments.
- **printf**: The compiler may do basic checks matching format specifiers (`%d`) to the argument types (integers in this case).

Why This Matters:
Makes sure function calls align with their definitions (e.g., correct number of arguments, correct argument types).

STEP 4: SEMANTIC ANALYSIS (5)

4. Logical Consistency

- **Conditional**: `if (b == 0) return a;`
  The compiler ensures `b` is a valid expression returning an integer or Boolean-like value.
- **Recursion**: `gcd(b, a % b)`
  The function signature and return type must match themselves recursively. Some languages also check for potential infinite recursion, though C compilers typically don't enforce that.

STEP 4: SEMANTIC ANALYSIS (6)

5. Errors vs. Warnings

- **Errors** stop compilation. For instance, calling `gcd` with three arguments or referencing an undeclared variable is typically an error.
- **Warnings** might point out suspicious behavior, like using the wrong format specifier in `printf`, but might not halt compilation.

Once the compiler confirms that everything is semantically valid, it proceeds to **Step 5: Intermediate Representation (IR) Generation**. At that point, the AST is translated into a more uniform, lower-level representation that paves the way for optimization and code generation.

STEP 5: INTERMEDIATE REPRESENTATION (IR) (1)

- Once the compiler knows your program is syntactically and semantically correct, it translates the **Abstract Syntax Tree (AST)** into an **Intermediate Representation (IR)**.

- The IR is often a simplified, lower-level, language-agnostic form of the program. Many modern compilers use forms like **three-address code** or **Static Single Assignment (SSA)** because these structures are easier to analyze and optimize systematically.

STEP 5: INTERMEDIATE REPRESENTATION (IR) - PURPOSE

Decoupling Front End and Back End

- After generating IR, the compiler doesn't need to worry about high-level C syntax anymore.
- Multiple front ends (e.g., C, C++, or even Fortran) can share the same IR, which then gets transformed by a common back end.

Facilitating Optimizations

- IR makes applying transformations easier (e.g., constant folding, dead code elimination, etc.).
- It's more uniform than high-level code, so optimizations can be implemented once and applied to multiple languages.

Machine Independence

- The IR is not tied to a specific CPU architecture. Later phases (instruction selection, register allocation) will tailor it to RISC-V or another target.

STEP 5: INTERMEDIATE REPRESENTATION (IR) (2)

Consider a **three-address code** style IR for the GCD function:

```
gcd:
    if b == 0 goto label_return_a
    t1 = a % b
    t2 = call gcd(b, t1)
    return t2
label_return_a:
    return a
```

Key Points:

- **Three-address code** typically uses at most three operands in each statement (e.g., `t1 = a % b`).
- **Branching** uses labels (`goto label_return_a`) rather than nested blocks or braces.
- **Function calls** appear as call `gcd(b, t1)` with an explicit mention of parameters, rather than the high-level `gcd(b, a % b)` syntax.

STEP 5: INTERMEDIATE REPRESENTATION (IR) (3)

Benefits for GCD

- **Explicit Branches**: IR shows exactly where the code jumps if `b == 0`.
- **Clear Representation of Recursion**: Recursively calling `gcd` becomes call `gcd(b, t1)`—making it easier to do transformations, like converting recursion to iteration (a potential optimization step).
- **Ease of Analysis**: Optimization phases don't have to parse C syntax again—everything is standardized in the IR.

STEP 5: INTERMEDIATE REPRESENTATION (IR) (4)

After IR generation, the compiler can apply various **optimizations** (Step 6). For instance:

- **Dead Code Elimination**: If a temporary “part of code” is never used, it can be removed.
- **Constant Folding**: Any computations with known constants are simplified.
- **Tail Recursion**: Recognizing that this GCD call is tail-recursive function, the compiler can transform it to an iterative form, if so desired.

Once optimized, the IR proceeds to instruction selection and register allocation (later steps) to finally produce RISC-V machine code.

STEP 6: OPTIMIZATION (1)

- In the optimization phase, the compiler takes the Intermediate Representation (IR) and refines it—removing redundancies, improving execution paths, and sometimes transforming the structure of the program for efficiency.

- The goal is to generate code that runs faster or uses fewer resources, without altering the program's observable behavior.

STEP 6: OPTIMIZATION – COMMON TECHNIQUES (1)

1. **Constant Folding**

- If an expression's operands are known at compile time (e.g., 2 + 3), the compiler replaces it with the computed result (5).
- This speeds up runtime by avoiding unnecessary calculations.

2. **Dead Code Elimination**

- Statements or branches that never affect the final output (e.g., unused variables, unreachable paths after a return) are removed.
- Reduces code size and improves efficiency by skipping instructions that do nothing.

STEP 6: OPTIMIZATION – COMMON TECHNIQUES (2)

3. Copy Propagation

If a temporary var. just holds another's value (e.g., `t2 = t1`), the compiler can often replace all uses of `t2` with `t1`, reducing overhead.

4. Loop Optimizations

- Techniques like loop unrolling, loop-invariant code motion, or strength reduction transform repetitive computations in loops.
- Although our GCD code doesn't explicitly use loops, compilers apply these optimizations in broader contexts.

5. Tail Recursion Removal

- Particularly relevant to GCD. If a function's last action is a recursive call to itself, some compilers convert it into a loop, eliminating function call overhead.
- The Euclidean algorithm's `gcd(a, b)` is a perfect example if recognized as tail-recursive.

STEP 6: OPTIMIZATION (2)

Transforming Recursive GCD to Iterative:

One of the best illustrations for GCD is **tail recursion elimination**. If the IR indicates a tail call like:

```
gcd:
    if b == 0 goto label_return_a
    t1 = a % b
    t2 = call gcd(b, t1)
    return t2
label_return_a:
    return a
```

STEP 6: OPTIMIZATION (3)

A compiler might detect that `gcd` calls itself last, which allows transformation into a loop:

```
// Conceptual iterative form the compiler might produce:
int optimized_gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

This iteration avoids repeated function call overhead and stack usage. Even if the compiler doesn't automatically do it, a programmer can do it manually—but many modern compilers will handle such optimizations in certain circumstances.

STEP 6: OPTIMIZATION (4)

Trade-Offs and Levels of Optimization

- **Compile-Time vs. Runtime**: More aggressive optimizations (like -O3 in GCC/Clang) can substantially improve runtime performance but increase compilation time.

- **Size vs. Speed**: An optimization might speed up execution but produce a larger binary, or vice versa. Different optimization levels (-Os for size, for instance) have different priorities.

STEP 6: OPTIMIZATION (5)

Outcome of Optimization:

- **Improved IR**: The IR is refined, so subsequent phases (instruction selection, register allocation) operate on a more efficient version.
- **Faster / Smaller Executables**: Less code, fewer instructions, or simpler data flow can lead to both performance gains and reduced memory usage.
- **Identical Program Semantics**: The program still outputs the same GCD results—optimizations can't change a program's functional behavior.

After optimizations are applied, the compiler transitions to **Step 7: Instruction Selection**, where each IR instruction is mapped to specific RISC-V assembly instructions. The optimized IR makes the instruction selection process more straightforward and yields better-performing final code.

STEP 7: INSTRUCTION SELECTION (1)

- The **Instruction Selection** phase translates the optimized Intermediate Representation (IR) into specific **machine instructions** for the target architecture—in this case, **RISC-V**.

- Each IR operation (like +, %, or if statements) must be matched to one or more RISC-V instructions.

STEP 7: INSTRUCTION SELECTION (2)

Mapping IR Operations to RISC-V

Arithmetic/Logic:

- If the IR has `t1 = a % b`, RISC-V provides a `rem` instruction: `rem t1, a0, a1`.
- For addition, IR might say `t2 = t1 + 5`, which maps to something like `addi t2, t1, 5` (if 5 fits in immediate form).

Branching:

- An IR statement like if `b == 0 goto label_return_a` becomes `beq a1, zero, label_return_a` in assembly.

Function Call:

- The IR might say `t2 = call gcd(b, t1)`; RISC-V compilers move `b` and `t1` into registers `a0` and `a1`, then perform `jal gcd`.

STEP 7: INSTRUCTION SELECTION (3)

From the GCD IR:

```
if b == 0 goto label_return_a
t1 = a % b
t2 = call gcd(b, t1)
return t2
```

A possible instruction selection outcome in RISC-V:

```
beq a1, zero, gcd_return  # if (b == 0), jump
rem t0, a0, a1            # t0 = a0 % a1
mv a0, a1                 # a0 = b
mv a1, t0                 # a1 = t0
jal gcd                   # call gcd
# (return from gcd call is already in a0)
```

STEP 7: INSTRUCTION SELECTION (4)

Why It's Crucial:

- **Architecture-Specific Details**: RISC-V might have different instructions and calling conventions than x86 or ARM. This phase ensures the generated code respects RISC-V's registers and instruction set.
- **Performance Implications**: A compiler can choose more efficient instructions or instruction combinations when multiple options exist. For example, using a specialized "remainder" instruction instead of a generic division if available.

Handling Complex IR Patterns: Some IR-level statements might need multiple assembly instructions. For example, if RISC-V didn't provide a native modulo (rem) instruction, the compiler would generate a sequence of instructions to replicate %. Fortunately, RISC-V does have rem, simplifying the mapping.

STEP 8: REGISTER ALLOCATION (1)

- **Register Allocation** is the phase where the compiler assigns variables and temporary values (from the IR) to the **physical registers** available on the target architecture—in our case, RISC-V.

- Since CPU registers are a scarce resource, how effectively the compiler allocates them can significantly impact performance.

STEP 8: REGISTER ALLOCATION (2)

The Need for Register Allocation

1. Limited Registers:

- RISC-V has a specific set of registers: `x0` through `x31`. Some of these are dedicated (e.g., `x0` is always 0, `sp` is the stack pointer, etc.).
- The calling convention also reserves specific registers for function arguments (`a0` through `a7`), return values, and temporary usage (`t0` through `t6`).

2. Memory vs. Register Access:

- Accessing a register is typically much faster than accessing memory.
- Good allocation minimizes the need to spill data to the stack (memory).

STEP 8: REGISTER ALLOCATION (3)

Mapping IR Variables to Registers

1. **Parameters & Return Values**:

- In our GCD example, `int gcd(int a, int b)` means `a` goes into `a0`, and `b` goes into `a1`.
- The function returns its result in `a0`.

2. **Temporaries**:

- When the IR says `t1 = a % b`, we might store `t1` in `t0` (a temporary register).
- If we need more temporary variables than available registers, we may have to spill some to memory.

3. **Allocation Strategy**:

- Different algorithms exist: *Graph coloring*, *linear scan*, etc.
- For simple examples, a basic approach is enough (e.g., a small number of variables mapped directly to known registers).

STEP 8: REGISTER ALLOCATION (4)

For the GCD Function:

```
# Suppose we've already selected instructions for gcd:
gcd:
    beq a1, zero, gcd_return      # if (b == 0)
    rem t0, a0, a1                # t0 = a0 % a1
    mv a0, a1                     # a0 = b
    mv a1, t0                     # a1 = t0
    jal gcd                       # call gcd
    ret

gcd_return:
    ret
```

a0: Holds `a` at the start and eventually the return value; **a1**: Holds `b`; **t0**: Temporary register for the `%` operation result.

If more var. or temporaries were needed than we have registers for, some would be “spilled” into memory and reloaded later.

STEP 8: REGISTER ALLOCATION (5)

Performance Implications

- **Fewer Spills**: The less the compiler has to write temporary values to the stack (and read them back), the faster the code generally runs.
- **Register Pressure**: If a function uses too many active variables, it can quickly exceed the available registers, forcing spillage. Optimizations can sometimes reduce live variables at any one time, lowering pressure on registers.

After **Register Allocation** is done, each IR instruction has a concrete set of registers where it reads/writes data. The compiler then proceeds to **Step 9: Code Generation**, where the final assembly is output (though we already have a near-final assembly at this point, it may just be annotated with or adjusted for final register decisions).

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (1)

- **Code Generation** transforms the optimized and register-allocated Intermediate Representation (IR) into the final **assembly language** specific to RISC-V.

- By this point, each IR operation has a chosen RISC-V instruction, and each variable or temporary has a register assignment.

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (2)

Outputting Assembly

- The compiler writes out assembly instructions in a .s file or directly passes them to the assembler.
- The assembly file typically includes function labels (e.g., `gcd:`), branch targets (e.g., `label_return_a:`), and directives that help manage data or align code in memory.

For our GCD function, the end result might look like:

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (3)

```
        .text
        .align  2
        .globl  gcd
gcd:
        beq a1, zero, gcd_return    # if (b == 0) jump
        rem t0, a0, a1              # t0 = a0 % a1
        mv a0, a1                   # a0 = b
        mv a1, t0                   # a1 = t0
        jal gcd                     # call gcd
        ret

gcd_return:
        ret
```

- **.text** indicates that what follows is code (not data).
- **.align 2** ensures proper alignment in memory (alignment in powers of 2).
- **.globl gcd**: Exports the label `gcd` so that other files can link against it.

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (4)

```
    .text
    .align  2
    .globl  gcd
gcd:
    beq a1, zero, gcd_return    # if (b == 0) jump
    rem t0, a0, a1              # t0 = a0 % a1
    mv a0, a1                   # a0 = b
    mv a1, t0                   # a1 = t0
    jal gcd                     # call gcd
    ret

gcd_return:
    ret
```

Instructions:

- `beq a1, zero, gcd_return` checks if `b == 0`.
- `rem t0, a0, a1` calculates the remainder.
- `mv a0, a1` and `mv a1, t0` set up registers for the next recursive call.
- `jal gcd` jumps to `gcd`, saving the return address automatically.
- `ret` returns to the caller.

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (5)

```
    .text
    .align  2
    .globl  gcd
gcd:
    beq a1, zero, gcd_return    # if (b == 0) jump
    rem t0, a0, a1              # t0 = a0 % a1
    mv a0, a1                   # a0 = b
    mv a1, t0                   # a1 = t0
    jal gcd                     # call gcd
    ret

gcd_return:
    ret
```

Calling Convention:

- `a0` and `a1` are the first two argument registers in RISC-V.
- `a0` also holds the return value.

STEP 9: CODE GENERATION (RISC-V ASSEMBLY) (6)

Comparing to the Original C Code

- The structure of recursive calls and the conditional check for `b == 0` remain logically identical.
- The high-level concept of "`returning a if b == 0, else calling gcd(b, a % b)`" translates directly to these assembly instructions.

Assembly File vs. Object File

- At this stage, the compiler might produce a .s assembly file.
- Typically, there's an internal or external **assembler** that then converts .s into a .o object file containing the actual machine code.

After code generation, we move on to **Step 10: Assembly and Linking**. This final step uses an assembler to make object files and a linker to combine them with libraries (e.g., the C standard library for `printf`). The end result is a complete executable that can run on RISC-V hardware or an emulator.

STEP 10: ASSEMBLY AND LINKING – ASSEMBLY (1)

After **Code Generation** (Step 9) produces the RISC-V assembly (either in a .s file or held in memory), the compiler (or an external toolchain) must assemble that code into machine instructions and then link it with other necessary components—most notably the C standard library, if the program uses functions like `printf`.

1. **Assembly File:** Often named something like gcd.s, containing the symbolic instructions.

2. **Assembler:**

- A tool that reads the assembly language file (.s) and translates each assembly instruction into a corresponding machine code instruction (opcodes, registers, addresses).
- Produces an **object file** (e.g., gcd.o) containing binary data and metadata like symbol information (names for functions like gcd).

STEP 10: ASSEMBLY AND LINKING – ASSEMBLY (2)

3. **Commands:** You might see or run something like:

```
riscv64-unknown-elf-gcc -c gcd.s -o gcd.o
```

The -c flag tells gcc (acting as a driver for the assembler) to compile/assemble but not link.

STEP 10: ASSEMBLY AND LINKING - LINKING (1)

1. **Combining Object Files**

- If you have multiple object files (e.g., gcd.o and main.o), the linker merges them into a single executable.
- The linker resolves symbol references (e.g., your main function referencing the gcd function, or calls to external library functions).

2. **Libraries**

- Calls to functions like `printf` need the standard C library.
- The linker pulls in the relevant parts of the C library (e.g., from libc.a or similar for RISC-V).

STEP 10: ASSEMBLY AND LINKING - LINKING (2)

3. **Commands:** A typical linker command might look like:

```
riscv64-unknown-elf-gcc gcd.o -o gcd
```

This links `gcd.o` with standard libraries, producing an executable named `gcd`.

4. **Executable Output**

- The final output file (e.g., gcd) is now in **RISC-V machine code**.
- This file can be loaded and executed on actual RISC-V hardware or a RISC-V emulator.

STEP 10: ASSEMBLY AND LINKING (1)

The Complete Executable

File Structure:
The linked binary contains:

- The machine instructions for your program (gcd, main).
- References and implementations of standard library calls (printf, etc.).
- Possibly a startup routine that calls main.

Running the Program: On a real RISC-V system or an emulator like `qemu-riscv64`, you can simply do (if permissions and environment are set up correctly):

```
./gcd
```

STEP 10: ASSEMBLY AND LINKING (2)

Why It Matters

- **Final Step to Execution**: Without assembling and linking, your high-level code can't become a standalone program.
- **External Dependencies**: Linking is where you incorporate any external libraries or static/dynamic linking strategies.
- **Modern Build Systems**: Tools like `make`, `CMake`, or others orchestrate these steps—compiling multiple source files, assembling them, and linking them in the correct order.

This final step cements your program into a form that the RISC-V architecture can directly load and execute, completing the entire compilation pipeline.

CONCLUSIONS - KEY TAKEAWAYS

- **Compilation vs. Interpretation**: Compilers produce executables; interpreters run code line by line.
- **Phased Structure**: Front end validates, the optimizer refines IR, and the back end emits machine code.
- **Intermediate Representation (IR)**: Language-independent bridge, enabling systematic optimizations.
- **Key Optimizations**: Loop-invariant code motion, constant folding, dead code elimination.
- **Constant Evolution**: Adapts to new architectures (e.g., RISC-V) and language features, fueling ongoing innovations.

See you at the next Lecture!

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