

Compiler Design Lab Project Problem Statement

Team members:

1. Adithya Ananth (CS23B001)
2. Sudhanva Bharadwaj BM (CS23B051)
3. Srikrishna Madhusudhanan (CS23B056)

Design and Implementation of a Compiler for a Subset of the C Programming Language Targeting the RISC-V Instruction Set Architecture

Introduction

The C programming language remains one of the most widely used languages for systems software, embedded systems, and performance-critical applications. RISC-V is a widely adopted instruction set architecture (ISA) which is increasingly used in the industry. Hence, we have chosen RISC-V for learning compiler back-end design.

This project aims to build a working compiler that demonstrates how high-level C constructs are systematically lowered into RISC-V assembly. We will also apply optimisations while ensuring the source program's meaning remains unchanged.

Problem Definition

Our project is to develop a compiler that translates programs written in a custom language (whose allowed instructions are a subset of C) into executable RISC-V assembly code. We will also apply selected optimisations to improve the efficiency of the generated code.

Scope of the Source Language

- Primitive data types such as int and char
- Variables with local and global scope
- Arithmetic and logical expressions
- Control flow constructs (if, if-else, while, for)
- Functions with parameters and return values, including **recursion**
- One-dimensional arrays and array indexing
- Pointer variables and basic pointer dereferencing

Advanced C features such as preprocessor directives, floating-point arithmetic, unions, and inline assembly are **out of scope**.

Target Architecture

The compiler's target is the RISC-V RV32I instruction set architecture. The generated assembly code must:

- Follow the standard RISC-V calling convention
- Correctly manage stack frames for function calls
- Be compatible with standard RISC-V simulators

Compiler Phases

The compiler will be structured into the following major phases:

1. **Lexical Analysis:** Tokenisation of C source code using a lexer.
2. **Syntax Analysis:** Parsing of tokens to construct an abstract syntax tree (AST).
3. **Semantic Analysis:** Type checking, symbol table management, and scope resolution.
4. **Intermediate Representation (IR):** Generation of an intermediate, machine-independent representation.
5. **Optimisation:** Application of selected optimisations such as constant folding, dead code elimination, and basic control-flow simplification.
6. **Code Generation:** Translation of optimised IR into RISC-V assembly code.

External tools to be used:

1. Lex/flex for lexical analysis
2. Yacc/Bison for parser
3. RISC-V simulator such as RARS or Spike

Optimizations

The compiler will implement a limited set of classical optimisations, such as:

- Constant folding and propagation
- Dead code elimination
- Common subexpression elimination
- Strength reduction

Deliverables

- Compiler source code
- Language manual with some test programs written in the supported C subset
- Project report detailing design decisions, architecture, and limitations