MIMD Processing Unit Compiler Documentation

abaxor engineering GmbH

August 2, 2024

Model Diagram

The following diagram illustrates the overall architecture and flow of the project:

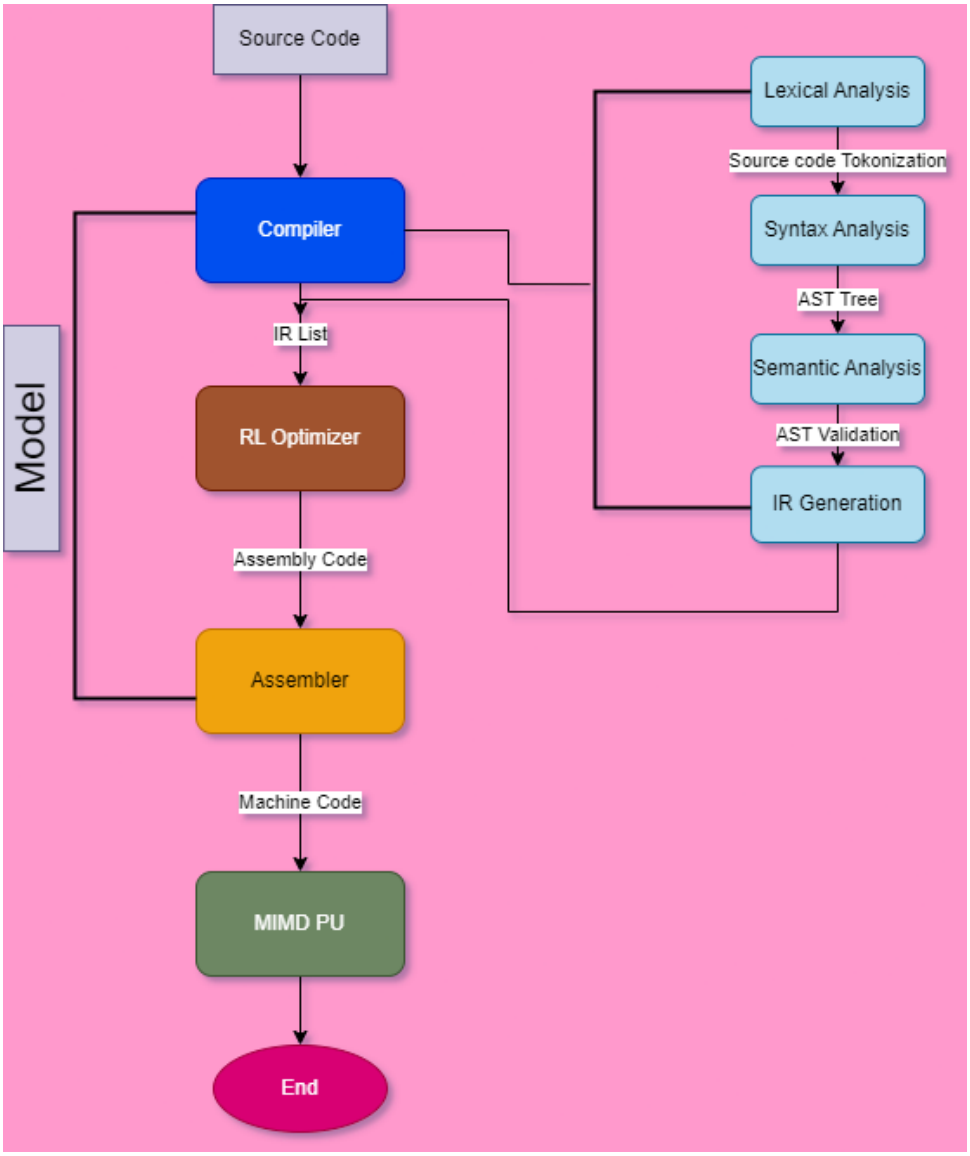

Figure 1: Description of what the diagram shows.

Objective

The primary goal of this project is to develop a compiler integrated with an assembler for a MIMD (Multiple Instruction, Multiple Data) Processing Unit architecture. The project aims to utilize Reinforcement Learning techniques to optimize the compilation process. Due to the current unavailability of a MIMD PU architecture, the initial implementation targets the x86 architecture, with plans to adapt it for the MIMD architecture in the future.

Current Development Status

The current implementation is a basic compiler framework which includes the following components:

Lexical Analysis

- **Purpose**: Tokenizes the input source code into a stream of tokens.
- **Details**: The lexical analyzer scans the source code and converts it into tokens using regular expressions. These tokens include identifiers, numbers, operators, assignment symbols, and statement terminators.

Syntax Analysis

- **Purpose**: Constructs an Abstract Syntax Tree (AST) from the token stream.
- **Details**: The syntax analyzer uses a parser to process the token stream and build an AST. It recognizes different types of expressions and statements, ensuring that the syntactic structure of the source code is correct.

Semantic Analysis

- **Purpose**: Performs basic type checking and variable declaration validation.
- **Details**: The semantic analyzer traverses the AST to validate the types and scopes of variables. It checks for type mismatches and ensures all variables are declared before use.

Intermediate Representation (IR) Generation

- **Purpose**: Produces an intermediate representation of the code.
- **Details**: The IR generator converts the AST into a lower-level intermediate representation, which is easier to optimize and translate into machine code.

Code Generation (x86 Assembly)

- **Purpose**: Translates the IR into x86 assembly code with basic register allocation.
- **Details**: The code generator translates the IR into x86 assembly instructions. It includes basic register allocation strategies to map variables and temporary values to CPU registers.

Assembly to Machine Code Translation

- **Purpose**: Converts the generated assembly code into a simplified machine code representation.
- **Details**: The assembler takes the x86 assembly code and converts it into a sequence of machine code instructions, ready for execution on an x86 processor.

Implemented Features

- **Lexical Analysis**: Tokenizes the input source code into a stream of tokens.
- **Syntax Analysis**: Constructs an Abstract Syntax Tree (AST) from the token stream.
- **Semantic Analysis**: Performs basic type checking and variable declaration validation.
- **IR Generation**: Produces an intermediate representation of the code.
- **Code Generation**: Translates the IR into x86 assembly code with basic register allocation.
- **Assembly**: Converts the generated assembly code into a simplified machine code representation.

Code Structure

The `BasicCompiler` class encapsulates all the compilation stages:

- `lexical_analysis`: Tokenizes the input source code.
- `syntax_analysis`: Builds the AST.
- `semantic_analysis`: Validates the AST for semantic correctness.
- `generate_ir`: Create an intermediate representation.
- `optimize_ir`: Placeholder for future IR optimization using RL.
- `code_generation`: Produces x86 assembly code.
- `assemble`: Converts assembly to basic machine code.

Limitations of the Current Model

- **Limited Language Support**:
  - The current implementation supports only a subset of C-like language features.
  - Lacks support for complex constructs such as loops, conditionals, and functions.
- **Basic Type System**:
  - Supports only integer types.
  - Lacks support for floating-point numbers, arrays, and user-defined types.
- **Simplified Assembly Generation**:
  - Produces basic x86 assembly without advanced optimizations or full instruction set support.
- **Naive Register Allocation**:
  - The current register allocation strategy is simplistic and may not efficiently utilize available registers.
- **No Optimization**:
  - While there's a placeholder for IR optimization, no actual optimizations are implemented yet. Optimization using RL will be implemented in Future
- **Limited Error Handling**:
  - The error reporting and recovery mechanisms are basic and may not provide detailed diagnostic information.
- **Lack of MIMD-specific Features**:

  - As the current implementation targets x86, it doesn't include any MIMD-specific optimizations or code generation strategies.

- **Simplified Assembler**:
  - The assembly to machine code translation is highly simplified and doesn't reflect a real-world assembler's complexity.

- **No Reinforcement Learning Integration**:
  - The RL Optimizer shown in the diagram is not yet implemented in the current code.

- **Limited Scope**:
  - The compiler currently handles only simple arithmetic expressions and variable assignments.

Future Work

- **Implement the Reinforcement Learning Optimizer**:
  - Develop and integrate RL techniques to optimize the IR.

- **Extend Language Support**:
  - Include more complex constructs such as loops, conditionals, and functions.

- **Enhance the Type System**:
  - Support a wider range of data types, including floating-point numbers, arrays, and user-defined types.

- **Advanced Optimization Techniques**:
  - Implement various optimization strategies in both IR and code generation phases to improve efficiency.

- **MIMD-specific Code Generation**:
  - Develop code generation and optimization strategies tailored for the MIMD PU architecture.

- **Improve Error Handling**:
  - Enhance error detection, reporting, and recovery mechanisms to provide detailed diagnostic information.

- **Sophisticated Register Allocation**:
  - Implement advanced register allocation algorithms to efficiently utilize available registers.

- **Extend the Assembler**:
  - Support a fuller range of x86 instructions and addressing modes, making the assembler more realistic and versatile.

- **Adaptation for MIMD PU Architecture**:
  - Begin the process of adapting the compiler for the target MIMD PU architecture once it becomes available.

Conclusion

This documentation provides an overview of the current state of the MIMD Processing Unit Compiler project, highlighting its objectives, current implementation, limitations, and future directions. As the project evolves, this documentation should be updated to reflect new developments and milestones achieved.