**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHAINSTITUTEOFMEDICALANDTECHNICALSCIENCES**

**CHENNAI-602105**

# BACKEND OF THE COMPILER

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

## BACHELOROFENGINEERING

**IN**

**COMPUTER SCIENCE ENGINEERING**

**Submitted by**

**K.L.V.Jayaram 192210314**

**M.Mahesh 192210312**

**Under the Supervision of**

**Dr.G.MICHAEL**

**NOVEMBER 2024**



**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICALSCIENCES**

**CAPSTONE PROJECT REPORT**

**BACKEND OF COMPILER**

**CSA14 - COMPILER DESIGN**

**TEAM MEMBERS**

**K. L. V JAYARAM (192210314)**

**M. MAHESH (192210312)**

## DECLARATION

We,**M.Mahesh,K.L.V.Jayaram**, students of **‘Bachelor of Engineering in computer science Technology**, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences,

Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **Code Generation** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

(K.L.V.Jayaram 192210314)

(M.Mahesh 192210312)

Date:22/11/24

Place: chennai

## CERTIFICATE

This is to certify that the project entitled **“Code Generation”** submitted by **M.Mahesh, K.L.V.Jayaram** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Computer Science Engineering.

Course Faculty

Dr. G. MICHAEL

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TOPICS** | **PAGE NO** |
| 1 | **Abstract** | **05** |
| 2 | **Introduction** | **06** |
| 3 | **Literature Review** | **06** |
| 4. | **Problem Statement** | **07** |
| 5. | **Methodology** | **08** |
| 6. | **Proposed Design**  1. Requirement Gathering and Analysis 2. Tool selection criteria  3. Scanning and Testing Methodologies | **09** |
| 7. | **Functionality**   1. User Authentication and Role Based Access Control. 2. Tool Inventory and Management 3. Security and Compliance Control | **09-** |
| 8. | **UI Design**   1. Layout Design 2. Feasible Elements Used 3. Elements Positioning and Functionality | **11** |
| 9. | **OUTPUT** | **15** |
| 10. | **Conclusion** | **16** |

**ABSTRACT:**

Efficient code generation is fundamental in compiler design, ensuring the translation of high-level programming languages into executable machine code. This report delves into the development of a bespoke intermediate representation (IR), specifically tailored to proficient code generation within compiler frameworks. The project aims to address the critical need for an optimized IR solution by defining its structure and format to accurately represent source code semantics. By incorporating essential functionalities and adhering to best practices in UI design, the project lays the groundwork for enhancing compiler performance and optimizing code generation processes.

The proposed design follows a systematic approach, beginning with requirement gathering and analysis to identify the precise needs of the tailored IR. Stakeholder interviews and analysis of compiler design principles guide this process, ensuring alignment with project objectives. Subsequently, the project focuses on defining the structure and format of the IR, encompassing instructions, operands, control flow constructs, and data types to accurately capture source code semantics.

This includes arithmetic operations (addition, subtraction, multiplication, division), logical operations (AND, OR, NOT), memory operations (load, store), control flow operations (branching, function calls), etc.

In conclusion, the project endeavors to make a significant contribution to compiler technology by addressing the critical need for efficient code generation. By developing a tailored intermediate representation (IR) and laying the foundation for enhanced compiler performance, the project aims to advance the field of compiler design and optimization. Through rigorous design, implementation, and documentation efforts, the project seeks to provide a robust and adaptable solution applicable in diverse computational contexts.

**Introduction:**

Compiler design serves as a cornerstone in software development, facilitating the translation of high-level programming languages into machine-executable instructions. At the heart of this intricate process lies code generation, a pivotal phase wherein an intermediate representation (IR) acts as a crucial intermediary between human-readable source code and executable machine code. However, the effectiveness of code generation hinges on the quality and efficiency of the IR utilized.

This report delves into the development of a bespoke IR tailored explicitly for proficient code generation within compiler frameworks. Support control flow constructs like conditional branches (if-else), loops (while, for), function calls, and returns.

The introduction provides a comprehensive overview of the project's objectives, methodologies, and significance. It sets the stage for exploring the critical role of efficient code generation in compiler design and underscores the need for a tailored IR solution to meet the specific requirements of modern software development practices. Through a systematic approach encompassing requirement gathering, analysis, IR design, implementation, and integration, the project aims to address the challenges inherent in existing IR solutions and pave the way for advancements in compiler technology.

An Intermediate Representation (IR) serves as a critical component in modern compiler design, acting as a bridge between high-level source code and low-level machine code. The effectiveness of an IR directly influences the compiler's ability to perform accurate semantic analysis, implement advanced optimizations, and generate efficient target-specific code. However, developing a robust IR is a complex task that presents several challenges. These include defining a versatile structure that can represent diverse programming constructs, handling intricate control flow mechanisms, and managing a wide range of data types such as integers, floating-point numbers, arrays, and pointers. Furthermore, ensuring that the IR facilitates both language-agnostic transformations and hardware-specific code generation adds another layer of complexity. This chapter highlights the critical role of IR in the compilation process and identifies the key challenges and considerations necessary to guide its development, aiming to create an IR that is both semantically precise and optimization-friendly.

**LITERATURE REVIEW:**

Intermediate Representation (IR) serves as a crucial component in modern compiler design, bridging the gap between high-level source code and low-level machine code. Its development has been extensively studied to address challenges in representation, optimization, and translation. The following review explores key contributions to the field, categorized into its structural design, control flow handling, data type management, and optimization.

The development of an effective Intermediate Representation (IR) has been extensively explored in compiler design literature, with a focus on addressing the challenges of structure, control flow, data types, and optimization. Early works, such as Aho, Sethi, and Ullman (1986), emphasized the utility of Three-Address Code for its simplicity and human readability, while later advancements like Static Single Assignment (SSA) form by Cytron et al.

The evolution of Intermediate Representation (IR) has been driven by the need to balance expressiveness, efficiency, and adaptability. Researchers have explored various forms of IR to address these goals, such as linear representations like Three-Address Code (TAC) for simplicity and graph-based models like Control Flow Graphs (CFG) and SSA form for optimization flexibility. The LLVM IR, introduced by Lattner and Adve (2004), has emerged as a widely adopted framework, combining a strongly-typed structure with cross-platform adaptability to support diverse backends. Advanced IRs also integrate type systems to manage complex data types and memory constructs, as seen in SUIF (Amarasinghe et al., 1996) and LLVM.

**Problem Statement:**

Developing an effective intermediate representation (IR) poses challenges in defining its structure, handling control flow constructs, managing data types, such as (integers, floating-point numbers, characters, arrays, structures, and pointers) and optimizing code generation algorithms. This section discusses the need for a robust IR that accurately captures source code semantics while facilitating optimization and target-specific translation. Key challenges and considerations are identified to guide the project's development process.

**METHODOLOGY:**

The development of a robust Intermediate Representation (IR) involves systematic steps to ensure that it effectively captures source code semantics, handles various control flows and data types, and supports optimizations and target-specific code generation. The following methodology outlines the approach to address these objectives.

To effectively develop an Intermediate Representation (IR), the methodology emphasizes seamless integration within the compiler workflow. The process begins with a frontend module that translates high-level source code into IR while preserving semantic fidelity. The IR is then incorporated into middle-end optimization passes, enabling transformations such as inlining, loop unrolling, and instruction reordering. A feedback loop from backend performance metrics informs iterative refinements to ensure adaptability across diverse workloads and hardware targets.

To ensure the robustness and adaptability of the Intermediate Representation (IR), the methodology incorporates iterative design and evaluation phases. The IR's structure is initially prototyped and subjected to stress tests using diverse programming constructs and edge cases from multiple languages to validate its ability to capture complex semantics. Advanced analysis techniques, such as alias analysis and dependence graphs, are integrated into the IR framework to enhance its handling of memory and data dependencies.

Furthermore, the methodology includes the development of modular optimization passes that can be independently tested and reused, promoting scalability. A key focus is placed on designing a feedback-driven refinement loop, where runtime profiling and performance data from the generated code guide improvements in the IR design and optimization strategies.

**Proposed Design:**

**Requirement Gathering and Analysis:** Stakeholder interviews and analysis of compiler design principles guide the identification of requirements for the tailored intermediate representation (IR).

**IR Structure and Format Definition:** The project focuses on defining the structure and format of the IR, encompassing instructions, operands, control flow constructs, and data types, to accurately represent source code semantics.

**Implementation and Integration:** The selected IR design is implemented in a programming language of choice and seamlessly integrated into compiler frameworks for code generation.

**Functionality:**

**Efficient Code Generation Algorithms:**

The project focuses on implementing efficient code generation algorithms within the IR framework, optimizing performance and minimizing resource utilization during the compilation process. Through rigorous algorithmic analysis and optimization techniques, the software strives to generate highly optimized machine code that meets performance benchmarks and quality standards.

**Instructions**:

Instructions that represent the basic operations supported by the target architecture. This includes arithmetic operations (addition, subtraction, multiplication, division), logical operations (AND, OR, NOT), memory operations (load, store), control flow operations (branching, function calls), etc.

**Data Types:**

Define a set of data types supported by the source language and the target architecture, such as integers, floating-point numbers, characters, arrays, structures, and pointers

**IR Format:**

Each IR instruction should contain information about its opcode, operands, data types, and any associated metadata.Consider using an intermediate representation language (IRL) or an abstract syntax tree (AST) to represent the IR in a human-readable format.

**Optimization Strategies and Techniques:**

The project incorporates a range of optimization strategies and techniques within the IR framework to improve code quality, reduce execution time, and minimize memory usage. This includes but is not limited to, constant folding, loop optimization, register allocation, and dead code elimination, among others. By leveraging these optimization techniques, the software aims to generate highly efficient machine code that maximizes performance while minimizing resource overhead.

**Accurate Representation of Source Code Semantics:** The IR is meticulously designed to accurately capture the semantics of source code, ensuring faithful translation into executable machine code. It encompasses a comprehensive set of instructions, operands, control flow constructs, and data types to accommodate the diverse requirements of modern programming languages and optimization techniques.

**Architectural Design:**

**Intermediate Representation (IR):**

**Definition and Purpose:** Defines intermediate representation (IR) and explains its purpose in compiler design, serving as a bridge between the source code and the target machine code.

**Structure of IR:** Describes the structure and format of the IR, including instructions, operands, control flow constructs, and data types.

**Instructions, Operands, Control Flow Constructs, and Data Types:** Provides detailed explanations of each component of the IR, illustrating how they represent the semantics of the source code.

**Design Considerations:** Discusses various design considerations taken into account while designing the IR, such as simplicity, expressiveness, and ease of translation to machine code.

**.Integration Layer:** The integration layer facilitates communication and data exchange between different modules within the compiler system. This includes interfaces for connecting the lexer and parser modules, passing data between semantic analysis and optimization passes, and interfacing with external libraries and tools for code generation.

**Performance Optimization:** The architectural design incorporates performance optimization techniques to maximize the efficiency and speed of code generation. This includes optimizing algorithms and data structures, leveraging parallelism and concurrency where applicable, and employing compiler optimizations to minimize execution time and resource usage.

**Error Handling and Diagnostics:** The error handling and diagnostics layer provides mechanisms for detecting, reporting, and handling compilation errors and warnings. This includes generating informative error messages, highlighting syntax errors in source code, and providing diagnostic information to assist developers in debugging compilation issues..

**UI Design:**

**Layout Design:** For the layout design of the capstone project in compiler design focused on code generation, it's essential to create a user-friendly interface that facilitates the development of the intermediate representation (IR) and subsequent translation into executable machine code.

**Feasible Elements Used:**User-friendly controls enabling the initiation of the code generation process and customization of target architecture and optimization settings.

**Elements Positioning and Functionality:** Elements positioning and functionality within the layout design for the code generation capstone project in compiler design are crucial for optimizing workflow efficiency and ensuring intuitive access to essential functionalities.

**Feasible Element Used:**

**Dashboard:**

For the capstone project in compiler design, the development of a dashboard can be a valuable element for managing the code generation process and providing insights into the intermediate representation (IR) development.

**Summary Information:**

Display summary information about the current status of IR development, such as the number of instructions, operands, control flow constructs, and data types defined.

**Visualization Tools:**

Incorporate visualization tools like charts or graphs to represent the distribution of instructions, operands, and control flow constructs within the IR. This provides a visual understanding of the IR's structure and complexity.

**Code Generation Progress:**

Include a section to track the progress of code generation, showing the percentage completion or status of each stage in the process.

**CODE:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_EQ\_LENGTH 100

// Function to generate three-address code

void generateThreeAddressCode(char \*equations[], int num\_equations) {

char variables[MAX\_EQ\_LENGTH][MAX\_EQ\_LENGTH]; char operation[MAX\_EQ\_LENGTH]; int var\_count = 0;

for (int i = 0; i < num\_equations; i++) { char \*equation = equations[i]; char \*token = strtok(equation, " ");

// Get the left-hand side variable strcpy(variables[var\_count], token); var\_count++;

while (token != NULL) { token = strtok(NULL, " "); if (token != NULL) {

if (strcmp(token, "=") != 0) {

// Check if token is an operation if (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 || strcmp(token, "\*") == 0 ||

strcmp(token, "/") == 0 ||

strcmp(token, "&&") == 0 || strcmp(token, "||") == 0) { strcpy(operation, token);

} else {

// Store operands into variables array strcpy(variables[var\_count], token); var\_count++;

}

}

}

}

// Generate machine code if (strcmp(operation, "+") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]); printf("ADD %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

} else if (strcmp(operation, "-") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]); printf("SUB %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

} else if (strcmp(operation, "\*") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]);

printf("MUL %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

} else if (strcmp(operation, "/") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]); printf("DIV %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

} else if (strcmp(operation, "&&") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]); printf("AND %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

} else if (strcmp(operation, "||") == 0) {

printf("MOV %s, %s\n", variables[1], variables[0]); printf("OR %s, %s", variables[1], variables[2]); printf("STOR %s, %s\n\n", variables[0], variables[1]);

}

// Reset var\_count and operation for next equation var\_count = 0; memset(operation, 0, sizeof(operation));

} }

int main() {

char \*equations[MAX\_EQ\_LENGTH]; char input[MAX\_EQ\_LENGTH]; int num\_equations = 0; printf("Enter equations in the format 'left = right' (or 'quit' to exit):\n");

while (1) {

fgets(input, sizeof(input), stdin); if (strcmp(input, "quit\n") == 0) { break; } equations[num\_equations] = strdup(input); num\_equations++;

}

printf("\nThe output of this program is based on the input equations.\n");

printf("One thing to notice is that in each equation, there is only one variable on the left side.\n");

printf("The operators and operands used on the right side must be space-separated from

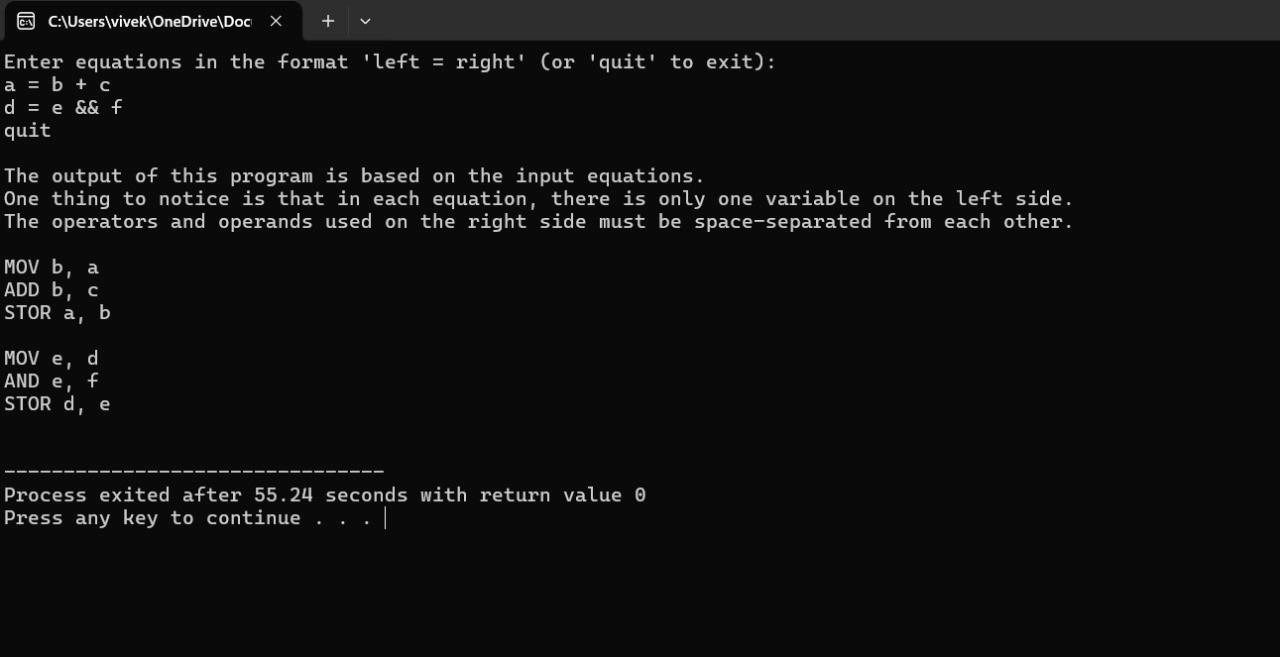
each other.\n\n");

// Generate machine code generateThreeAddressCode(equations, num\_equations);

return 0;

}

**OUTPUT:**



**Element Positioning and Functionality:**

**Syntax Highlighting and Error Handling:**

Implement syntax highlighting to differentiate between different components of the IR (instructions, operands, etc.).Include error handling mechanisms to detect and highlight syntax errors in real-time, aiding developers in debugging**.**

**Data Visualization:**

Position data visualization tools to provide graphical representations of the IR's structure and relationships between different components.Use visual cues such as charts or diagrams to enhance understanding and analysis of the IR.

**Code Generation Controls:**

Include controls for initiating the code generation process, such as buttons or dropdown menus to select the target architecture and optimization settings.Ensure clear labeling and intuitive design to streamline the code generation workflow.

**Conclusion:**

This project entails creating software capable of translating intermediate representation (IR) code into executable machine code for a specific target architecture. To accomplish this, it's crucial to meticulously design and implement an intermediate representation (IR) that accurately represents the semantics of the source code.

The defined IR structure should encompass instructions, operands, control flow constructs, and data types, providing a comprehensive framework for representing the source code's semantics. By defining the IR's format and ensuring its integrity, developers can facilitate the code generation process and ensure the resulting machine code's efficiency and correctness.

**REFERENCES:**

1. **Aho, A. V., Sethi, R., & Ullman, J. D. (1986)**  
   *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.
2. **Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K.(1991)** "Efficiently Computing Static Single Assignment Form and the ControlDependenceGraph." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4), 451–490.
3. ElongProgramAnalysis&Transformation." *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.
4. **Kildall,G.A.(1973)** "A Unified Approach to Global Program Optimization."  
   *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*.
5. **Allen,F.E.(1970)**"ControlFlowAnalysis." *Proceedings of a Symposium on Compiler Optimization.*
6. **Amarasinghe, S. P., Anderson, J. M., Liao, S.-W., Tseng, C.-W., O'Reilly, U.-M., & Lam, M. S. (1996)**  
   "The SUIF Compiler for Scalable Parallel Machines."  
   *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing.*
7. **McKinley, K. S., Carr, S., & Tseng, C.-W. (1996)**  
   "Improving Data Locality with Loop Transformations."