



SIMATS SCHOOL OF ENGINEERING
SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
CHENNAI-602105



Code Generator

A CAPSTONE PROJECT REPORT

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

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE ENGINEERING

Submitted by
Vishnu Sanjeev.G 192210101
Vivek.K 192210103
Gowrinadh.B 192211043

Under the Supervision of
Dr.W.Deva Priya

March 2024

DECLARATION

We, **Vishnu Sanjeev.G, Vivek Reddy.K, Gowrinadh.B**, 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.

(Vishnu Sanjeev.G 192210101)

(Vivek.K 192210103)

(Gowrinadh.B 192211043)

Date:

Place:

CERTIFICATE

This is to certify that the project entitled “**Code Generation**” submitted by **Vishnu Sanjeev.G, Vivek.K, Gowrinadh.B** 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.W.Deva Priya

Table of Contents

S.NO	TOPICS
1	Abstract
2	Introduction
3	Problem Statement
4	Proposed Design <ul style="list-style-type: none">1. Requirement Gathering and Analysis2. Tool selection criteria3. Scanning and Testing Methodologies
5.	Functionality <ul style="list-style-type: none">1. User Authentication and Role Based Access Control.2. Tool Inventory and Management3. Security and Compliance Control
6	UI Design <ul style="list-style-type: none">1. Layout Design2. Feasible Elements Used3. Elements Positioning and Functionality
7	Conclusion

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.

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..

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("MOV %s, %s", variables[1], variables[0]);
    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++;
    }

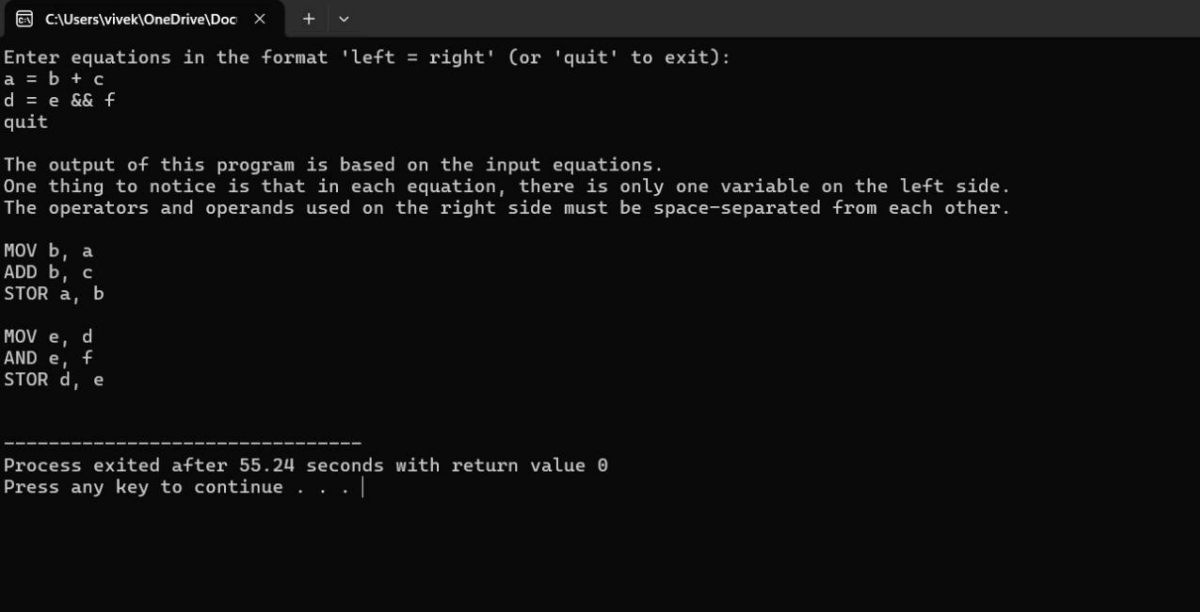
    printf("\n\nThe output of this program is based on the input equations.\n");
    printf("One thing to notice is that in each equation, there is only one variable on the left side.\n");
}

```

```
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:



```
C:\Users\vivek\OneDrive\Doc x + v  
Enter equations in the format 'left = right' (or 'quit' to exit):  
a = b + c  
d = e && f  
quit  
  
The output of this program is based on the input equations.  
One thing to notice is that in each equation, there is only one variable on the left side.  
The operators and operands used on the right side must be space-separated from each other.  
  
MOV b, a  
ADD b, c  
STOR a, b  
  
MOV e, d  
AND e, f  
STOR d, e  
  
-----  
Process exited after 55.24 seconds with return value 0  
Press any key to continue . . . |
```

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.