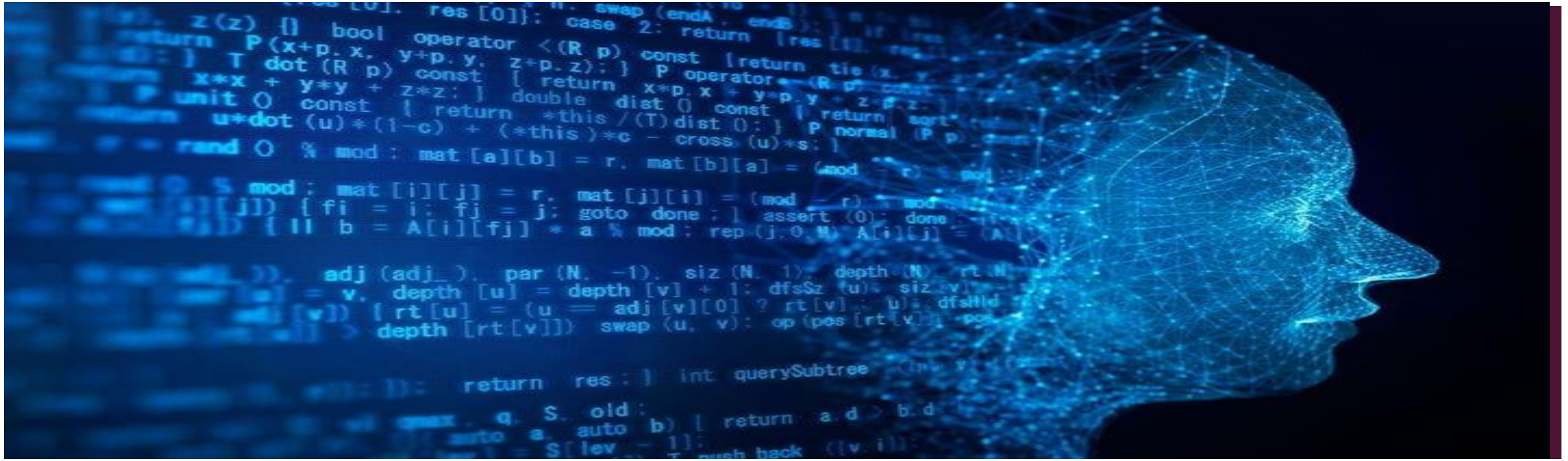
IMPLEMENTATION OF CODES FOR THE GIVEN EXPRESSION

BY
G .CHARAN
N.SRI KRISHNA CHAITANYA
K.POOJITHA



INTRODUCTION

- Code generation is the process in a compiler where high-level source code is translated into machine code or assembly language that can be executed by the target hardware. It bridges the semantic gap between the abstract representations of the source code and the specific instructions understood by the processor.
- Generating efficient machine code is crucial for achieving optimal performance of software applications. It directly impacts factors such as execution speed, memory usage, and power consumption. Thus, code generation plays a pivotal role in ensuring the effectiveness and competitiveness of software products.
- It takes the intermediate representation of the source code as input and produces executable machine code or assembly language as output. This presentation will delve into the intricacies of this phase, focusing on intermediate code representation, optimization techniques, instruction selection, scheduling, and register allocation.

INTERMEDIATE CODE REPRESENTATION

- Intermediate code serves as an intermediary representation between the high-level source code and the target machine code. It abstracts away language-specific details while preserving the essential semantics of the program.
- Various forms of intermediate representations exist, including Three-Address Code (TAC), Abstract Syntax Trees (AST), and Control Flow Graphs (CFG). Each representation has its advantages and is suited for different optimization and code generation tasks.
- Intermediate code simplifies the code generation process by providing a standardized format that facilitates analysis and transformation. It enables modular design, allowing compilers to focus on optimization and target-specific code generation independently of source languages.

CODE OPTIMIZATION TECHNIQUES

- Optimization Goals: The primary objectives of code optimization are to improve program efficiency, reduce resource consumption, and enhance maintainability. Optimization techniques aim to achieve these goals while preserving program correctness and semantics.
- Constant Folding: This technique involves evaluating constant expressions at compile time rather than at runtime, reducing the computational overhead during execution.
- Loop Optimization: Loops are a significant target for optimization due to their frequent occurrence in programs. Techniques such as loop unrolling, loop fusion, and loop interchange aim to minimize loop overhead and maximize iteration efficiency.

INSTRUCTION SELECTION AND SCHEDULING

- Instruction Selection: Intermediate code operations are mapped to machine instructions compatible with the target architecture. This process involves selecting suitable instructions that can efficiently implement the desired semantics.
- Operator Translation: Logical operators such as AND and OR are translated into bitwise operations (e.g., bitwise AND and OR) and Arthematic operators such as Add ,Sub, Mul and Div at the machine code level. This translation ensures that the program's logical behavior is preserved while utilizing the available hardware resources effectively.
- Instruction Scheduling: The order of instructions is optimized to minimize pipeline stalls, maximize instruction-level parallelism, and exploit hardware resources efficiently. Scheduling decisions consider factors such as data dependencies, resource conflicts, and processor pipeline characteristics.

PROPOSED DESIGN

- In the proposed design, the intermediate representation (IR) operations would encompass a range of basic operations that capture the semantics of the source language while being amenable to translation into machine code instructions. These operations might include arithmetic operations (addition, subtraction, multiplication, division), logical operations (AND, OR, NOT)assignments, function definitions, and any other language-specific constructs necessary to represent the source code's behavior. Each IR operation would be mapped to one or more corresponding machine code instructions during the code generation phase, ensuring that the resulting machine code faithfully executes the intended behavior of the source program.

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++;
```

```
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]);
```

CODE

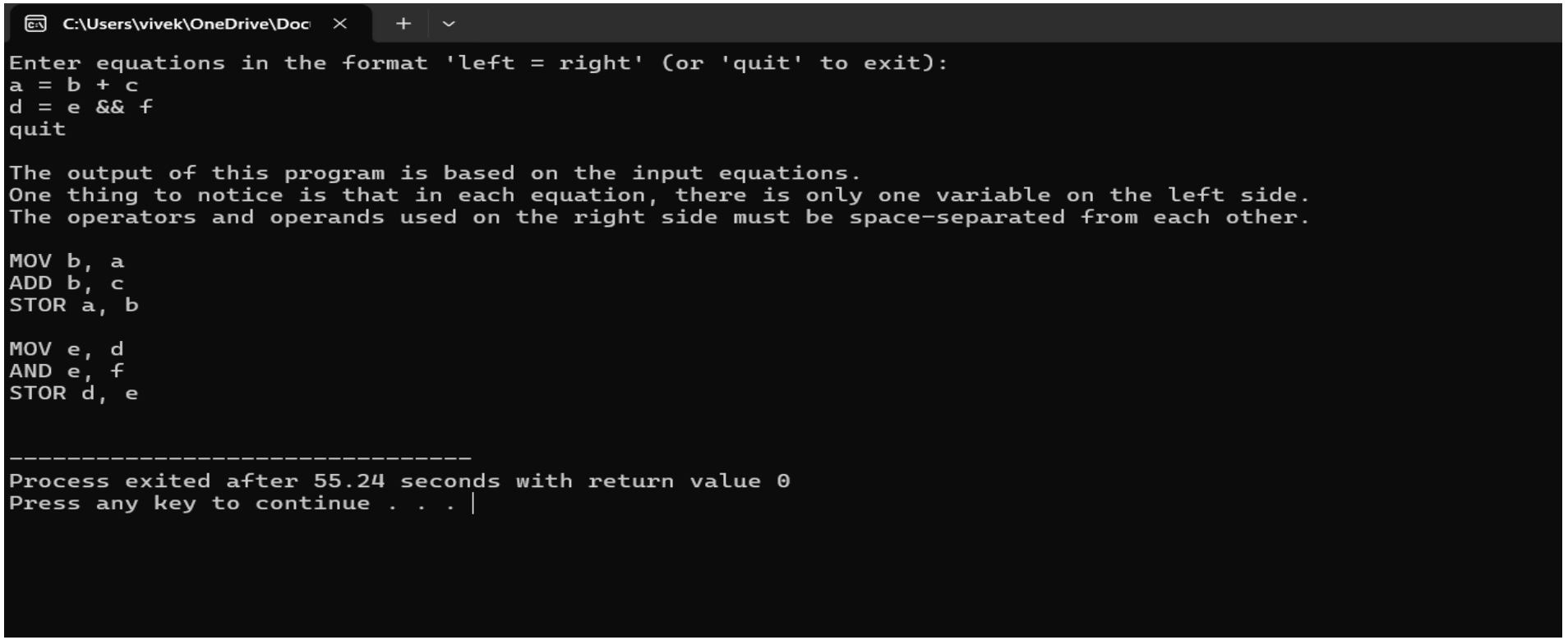
```
else if (strcmp(operation, "&&") == 0) {
    printf("MOV %s, %s\n", variables[1], variables[0]);
    printf("AND %s, %s", variables[1], variables[2]);
    printf("STOR %s, %s\n\n", variables[0], variables[1]);
}
        else if (strcmp(operation, "||") == 0) {
    printf("MOV %s, %s\n", variables[1], variables[0]);
    printf("OR %s, %s", variables[1], variables[2]);
    printf("STOR %s, %s\n\n", variables[0], variables[1]);
}
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]);
}
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


C:\Users\vivek\OneDrive\Doc
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 . . .

FUTURE SCOPE

- Looking ahead, the future of intermediate code to machine code conversion, or code generation, offers opportunities for advancements in efficiency, adaptability, and security. This includes refining optimization techniques to enhance performance, adapting seamlessly to emerging computing architectures, prioritizing energy-efficient code generation for sustainability, implementing robust security measures to mitigate vulnerabilities, and integrating with high-level abstractions for streamlined development. These endeavors aim to propel the field forward, ensuring that generated machine code is not only efficient and reliable but also adaptable to evolving technological landscapes and increasingly stringent security requirements.

CONCLUSION

- The code generation phase is essential for translating high-level source code into efficient machine code or assembly language. Intermediate representations, optimization techniques, instruction selection, scheduling, and register allocation play key roles in this process.
- Optimizing generated code is crucial for achieving high performance, reduced resource consumption, and improved software quality. It enables software developers to harness the full potential of modern computing hardware and deliver robust, efficient applications.
- As hardware architectures evolve and programming paradigms shift, code generation techniques must adapt to leverage new optimization opportunities and address emerging challenges. By staying abreast of advancements in compiler technology, developers can continue to refine code generation strategies and enhance the performance and reliability of software systems.

THANKYOU