C Language Code to Assembly Language

Pseudo-Code Converter

Submitted By

Student Name	Student ID
Masum Billah	221-15-5571
Sabbir Islam	221-15-4657
Mahmudul Hasan	221-15-4990
Faisal Mollah	221-15-4718
Morshed khan Munan	221-15-4951

MINI LAB PROJECT REPORT

This Report Presented in Partial Fulfillment of the course **CSE-331: Compiler Design Lab in the Computer Science and Engineering Department**

DAFFODIL INTERNATIONAL UNIVERSITY

Dhaka, Bangladesh

April 13, 2025

DECLARATION

We hereby declare that this lab project has been done by us under the supervision of **Mushfiqur Rahman**, **Assistant Professor**, Department of Computer Science and Engineering, Daffodil International University. We also declare that neither this project nor any part of this project has been submitted elsewhere as lab projects.

Submitted To:

Mushfiqur Rahman
Assistant Professor
Department of Computer Science and Engineering Daffodil
International University

Submitted by

Masum Billah ID: 221-15-5571 Dept. of CSE, DIU	
Sabbir Islam ID: 221-15-4657 Dept. of CSE, DIU	Mahmudul Hasan ID: 221-15-4990 Dept. of CSE, DIU
Faisal Mollah ID: 221-15-4718 Dept. of CSE, DIU	Morshed khan Munan ID: 221-15-4951 Dept. of CSE, DIU

COURSE & PROGRAM OUTCOME

The following course have course outcomes as following:.

Table 1: Course Outcome Statements

CO's	Statements
CO1	**Define** and **Relate** classes, objects, members of the class, and relationships among them needed for solving specific problems
CO2	**Formulate** knowledge of object-oriented programming and Java in problem solving
CO3	**Analyze** Unified Modeling Language (UML) models to **Present** a specific problem
CO4	**Develop** solutions for real-world complex problems **applying** OOP concepts while evaluating their effectiveness based on industry standards.

Table 2: Mapping of CO, PO, Blooms, KP and CEP

CO	PO	Blooms	KP	CEP
CO1	PO1	C1, C2	KP3	EP1, EP3
CO2	PO2	C2	KP3	EP1, EP3
CO3	PO3	C4, A1	KP3	EP1, EP2
CO4	PO3	C3, C6, A3, P3	KP4	EP1, EP3

The mapping justification of this table is provided in section **4.3.1**, **4.3.2** and **4.3.3**.

Table of Contents

Chapter 1

Introduction

This chapter provides a brief introduction of this project.

1.1 Introduction

This project presents a simple compiler backend component that converts C language constructs into simplified assembly pseudocode. The problem addressed is the initial transformation from high-level code to low-level representation, which is a fundamental step in compiler design.

1.2 Motivation

This project implements a simplified compiler backend that converts basic C language constructs—such as variable declarations, arithmetic operations, and input/output statements—into assembly-style pseudocode. It addresses the fundamental compiler design task of transforming high-level code into low-level representations. The tool demonstrates how C code can be translated into basic instructions like MOV, ADD, and IN, providing an educational understanding of compiler behavior and laying the groundwork for more advanced code generation in the future.

1.3 Objectives

The objectives of this project are given below :

- Convert C variable declarations and expressions to pseudocode
- Handle user input/output via scanf/printf
- Translate simple arithmetic expressions into corresponding operations (ADD, SUB, etc.)

1.4 Feasibility Study

Many educational compiler projects and open-source tools are available that demonstrate various aspects of code generation and compiler functionality. These range from full-fledged compiler suites to lightweight tools aimed at helping students grasp core concepts. This project aligns with those educational efforts by offering a highly simplified yet practical approach to code generation. By focusing on translating basic C constructs into pseudocode, it serves as an accessible introduction to compiler backend processes. Its simplicity makes it especially suitable for classroom settings, lab exercises, and early-stage compiler education, where the primary goal is to understand the core principles without the complexity of full compiler design.

1.5 Gap Analysis

While modern compilers are equipped to handle a wide range of language features, complex control flows, and extensive optimizations, this project intentionally narrows its focus to the foundational aspects of code translation. It omits advanced constructs such as control structures, functions, and memory management in favor of simplicity and clarity. The primary gap this project addresses is the lack of beginner-friendly tools that demonstrate a minimum viable code generation engine. By concentrating on essential elements like declarations, assignments, and input/output, it provides a clear and digestible pathway for students to understand how source code begins its journey toward executable instructions.

1.6 Project Outcome

- Successfully translates basic C constructs (declarations, assignments, `scanf`, `printf`) into simplified pseudocode.
- Demonstrates key concepts of compiler backend design in an understandable way.
- Helps students visualize how high-level code is transformed into low-level operations.
- Provides a functional and extensible framework for further exploration of compiler components.
- Serves as an effective educational tool for learning code generation and syntax analysis.

Chapter 2

Proposed Methodology/Architecture

This chapter describes the system design and methodology used to build the converter.

2.1 Requirement Analysis & Design Specification

2.1.1 Overview

This project is built using C++, which handles the task of reading and processing C code line by line. It makes use of simple C++ features like string handling, condition checks, and stream operations to understand each statement and convert it into easy-to-read pseudocode. The goal is to simulate the early stages of how a real compiler works—breaking down code and translating it into something closer to machine instructions, but in a simplified and educational way.

2.1.2 Proposed Methodology/ System Design

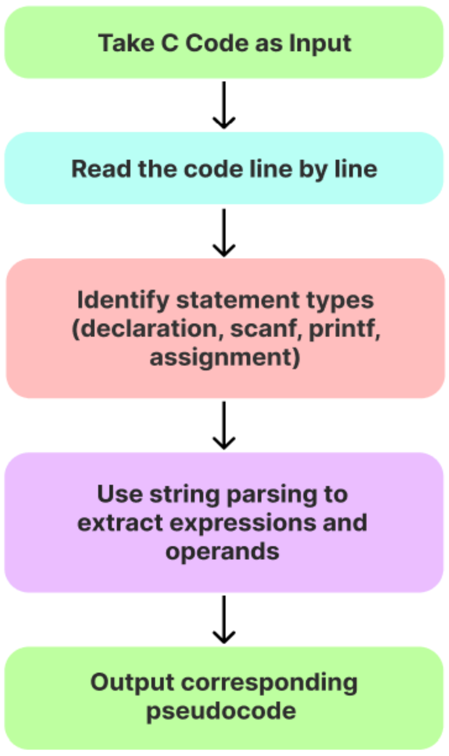

Take C Code as Input
Read the code line by line
Identify statement types (declaration, scanf, printf, assignment)
Use string parsing to extract expressions and operands
Output corresponding pseudocode

Figure 2.1: C to Assembly pseudocode converter workflow diagram.

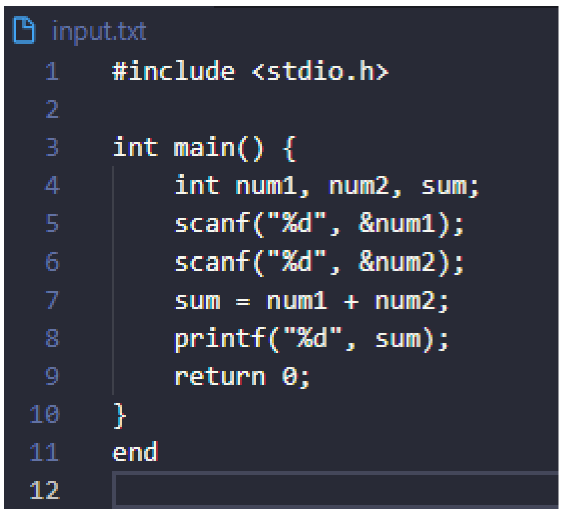
Proposed Methodology / System Design:

1. Read C code line by line
2. Identify statement types (declaration, scanf, printf, assignment)
3. Use string parsing to extract expressions and operands
4. Output corresponding pseudocode (e.g., MOV, ADD, SUB)

2.1.3 UI Design

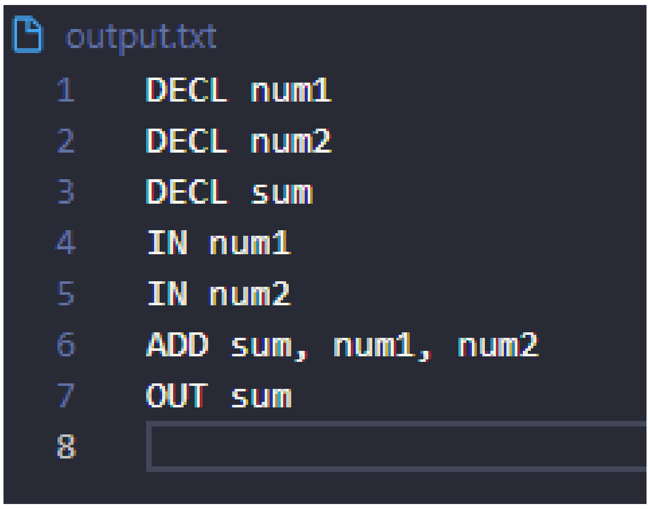
There is no graphical user interface. The input and output are handled through console (stdin/stdout).

UI for Input and Output:



input.txt

```
#include <stdio.h>

int main() {
    int num1, num2, sum;
    scanf("%d", &num1);
    scanf("%d", &num2);
    sum = num1 + num2;
    printf("%d", sum);
    return 0;
}
end
```



output.txt

```
DECL num1
DECL num2
DECL sum
IN num1
IN num2
ADD sum, num1, num2
OUT sum
```

2.2 Overall Project Plan

- Day 1-2: Requirement analysis and scope identification
- Day 3-4: Basic parsing and syntax handling
- Day 5-6: Code generation logic implementation
- Day 6-8: Testing and documentation

Chapter 3

Implementation and Results

This chapter outlines the technical implementation, performance, and discussion of results.

3.1 Implementation

The implementation is done in **C++**. It uses functions to handle each statement type (e.g., declarations, scanf, printf, assignments). The code uses basic string manipulation and stream parsing to translate each instruction to assembly-style pseudocode.

Here is the code implementation of this project using **C++**:

CODE:

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>

using namespace std;

int labelCounter = 0;

string trim(const string& str) {
    size_t first = str.find_first_not_of(" \t");
    if (first == string::npos) return "";
    size_t last = str.find_last_not_of(" \t");
    return str.substr(first, last - first + 1);
}

// --- Recognizers ---
bool isDeclaration(const string& line) {
    return line.find("int ") == 0 ||
        line.find("float ") == 0 ||
        line.find("double ") == 0 ||
        line.find("char ") == 0;
}
```

```cpp
bool isScanf(const string& line) {
    return line.find("scanf") != string::npos;
}

bool isPrintf(const string& line) {
    return line.find("printf") != string::npos;
}

bool isAssignment(const string& line) {
    return line.find('=') != string::npos && !isScanf(line) && !isDeclaration(line);
}

bool isIf(const string& line) {
    return line.find("if") == 0;
}

bool isElse(const string& line) {
    return line.find("else") == 0;
}

bool isWhile(const string& line) {
    return line.find("while") == 0;
}

bool isFor(const string& line) {
    return line.find("for") == 0;
}

// --- Handlers ---
void handleDeclaration(const string& line) {
    string type;
    size_t spacePos = line.find(' ');
    if (spacePos == string::npos) return;

    type = line.substr(0, spacePos);
    string content = line.substr(spacePos + 1);
    if (content.back() == ';') content.pop_back();

    stringstream ss(content);
    string var;
    while (getline(ss, var, ',')) {
        var = trim(var);

        if (var.find('=') != string::npos) {
            size_t eq = var.find('=');
            string name = trim(var.substr(0, eq));
            string expr = trim(var.substr(eq + 1));

            stringstream exprStream(expr);
            string lhs, rhs;
            char op;
            exprStream >> lhs;
            if (exprStream >> op >> rhs) {
                string instr;
                switch (op) {
                case '+': instr = "ADD"; break;
                case '-': instr = "SUB"; break;
                case '*': instr = "MUL"; break;
                case '/': instr = "DIV"; break;
                default: instr = "???"; break;
                }
                cout << instr << " " << name << ", " << lhs << ", " << rhs << endl;
            }
            else {
                cout << "MOV " << name << ", " << lhs << endl;
            }
        }
        else {
            cout << "DECL " << var << endl;
        }
    }
}

void handleScanf(const string& line) {
    size_t amp = line.find('&');
    size_t end = line.find(')', amp);
    if (amp != string::npos && end != string::npos) {
        string var = line.substr(amp + 1, end - amp - 1);
        cout << "IN " << trim(var) << endl;
    }
}

void handlePrintf(const string& line) {
    size_t comma = line.rfind(',');
    if (comma != string::npos) {
        size_t end = line.find(')', comma);
        string var = line.substr(comma + 1, end - comma - 1);
        cout << "OUT " << trim(var) << endl;
```

```cpp
    }
}

void handleAssignment(const string& line) {
    string code = line;
    if (code.back() == ';') code.pop_back();
    stringstream ss(code);
    string lhs, eq, rhs1, rhs2;
    char op;

    ss >> lhs >> eq >> rhs1;
    if (!(ss >> op >> rhs2)) {
        cout << "MOV " << lhs << ", " << rhs1 << endl;
    }
    else {
        string instr;
        switch (op) {
        case '+': instr = "ADD"; break;
        case '-': instr = "SUB"; break;
        case '*': instr = "MUL"; break;
        case '/': instr = "DIV"; break;
        default: instr = "???"; break;
        }
        cout << instr << " " << lhs << ", " << rhs1 << ", " << rhs2 << endl;
    }
}

void handleIf(const string& line) {
    size_t start = line.find('(');
    size_t end = line.find(')', start);
    if (start == string::npos || end == string::npos) return;

    string condition = trim(line.substr(start + 1, end - start - 1));
    vector<string> ops = { "==", "!=", ">=", "<=", ">", "<" };
    string opFound, lhs, rhs;

    for (const auto& op : ops) {
        size_t pos = condition.find(op);
        if (pos != string::npos) {
            lhs = trim(condition.substr(0, pos));
            rhs = trim(condition.substr(pos + op.length()));
            opFound = op;
            break;
        }
    }

    string instr;
    if (opFound == "<") instr = "IF_LT";
    else if (opFound == ">") instr = "IF_GT";
    else if (opFound == "==") instr = "IF_EQ";
    else if (opFound == "!=") instr = "IF_NE";
    else if (opFound == "<=") instr = "IF_LE";
    else if (opFound == ">=") instr = "IF_GE";

    cout << instr << " " << lhs << ", " << rhs << ", LABEL_" << labelCounter++ << endl;
}

void handleWhile(const string& line) {
    int loopLabel = labelCounter++;
    cout << "LABEL_" << loopLabel << ":" << endl;
    handleIf(line); // reuse the same condition logic
}

void handleFor(const string& line) {
    size_t start = line.find('(');
    size_t end = line.find(')', start);
    if (start == string::npos || end == string::npos) return;

    string content = line.substr(start + 1, end - start - 1);
    stringstream ss(content);
    string init, cond, inc;
    getline(ss, init, ';');
    getline(ss, cond, ';');
    getline(ss, inc);

    handleAssignment(init + ";");
    int loopLabel = labelCounter++;
    cout << "LABEL_" << loopLabel << ":" << endl;
    handleIf("if(" + cond + ")");
    cout << "// loop body here" << endl;
    handleAssignment(inc + ";");
    cout << "JMP LABEL_" << loopLabel << endl;
}

// --- Main Driver ---
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
```

```
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif

    string line;
    while (getline(cin, line)) {
        line = trim(line);
        if (line.empty() || line == "end") continue;

        if (line.find("#") == 0 || line.find("main") != string::npos ||
            line == "{" || line == "}" || line.find("return") != string::npos)
            continue;


        if (isDeclaration(line)) handleDeclaration(line);
        else if (isScanf(line)) handleScanf(line);
        else if (isPrintf(line)) handlePrintf(line);
        else if (isIf(line)) handleIf(line);
        else if (isElse(line)) cout << "ELSE BLOCK" << endl;
        else if (isWhile(line)) handleWhile(line);
        else if (isFor(line)) handleFor(line);
        else if (isAssignment(line)) handleAssignment(line);
        else cerr << "Unrecognized line: " << line << endl;
    }

    return 0;
}
```

3.2 Performance Analysis

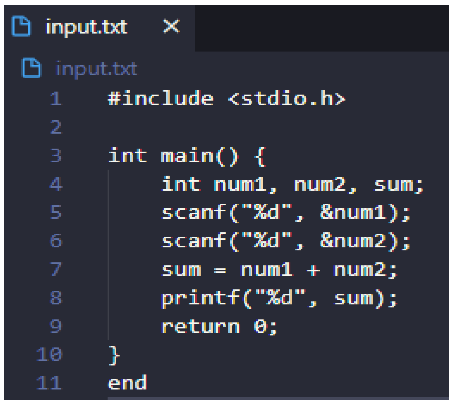
The program performs efficiently for its intended scope, processing input C code in a linear, line-by-line fashion. Since it deals with basic syntax and uses lightweight string operations, it exhibits fast execution with minimal memory usage. It is well-suited for small to medium-sized C programs focused on variable manipulation and I/O.

3.3 Results and Discussion

The C to Assembly Pseudocode Converter was designed to translate a subset of the C programming language into simplified assembly-like pseudocode. The converter successfully handles fundamental constructs including variable declarations, input/output operations, and arithmetic expressions. The transformation outputs pseudocode resembling low-level instructions like MOV, ADD, SUB, MUL, DIV, IN, and OUT, thereby mimicking the behavior of a basic compiler backend or an intermediate representation generator.

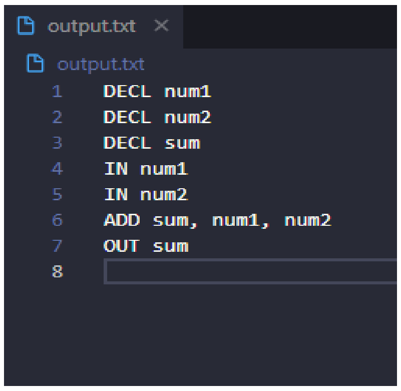
A sample input and output illustrates the transformation process:

INPUT: input calculations



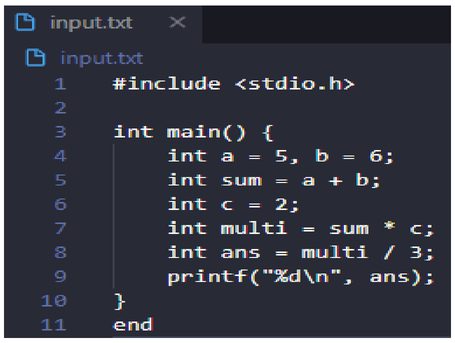
```
#include <stdio.h>

int main() {
    int num1, num2, sum;
    scanf("%d", &num1);
    scanf("%d", &num2);
    sum = num1 + num2;
    printf("%d", sum);
    return 0;
}
end
```

OUTPUT:



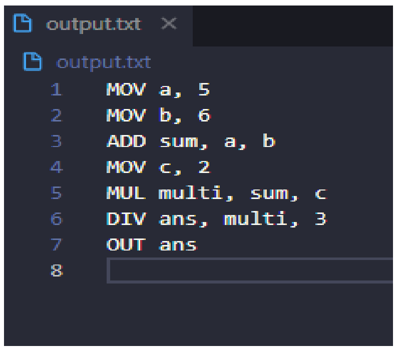
```
DECL num1
DECL num2
DECL sum
IN num1
IN num2
ADD sum, num1, num2
OUT sum
```

INPUT: arithmetic calculations



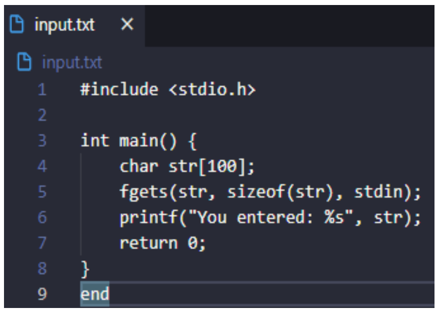
```
#include <stdio.h>

int main() {
    int a = 5, b = 6;
    int sum = a + b;
    int c = 2;
    int multi = sum * c;
    int ans = multi / 3;
    printf("%d\n", ans);
}
end
```

OUTPUT:



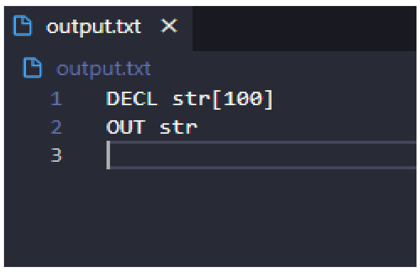
```
MOV a, 5
MOV b, 6
ADD sum, a, b
MOV c, 2
MUL multi, sum, c
DIV ans, multi, 3
OUT ans
```

INPUT: string input



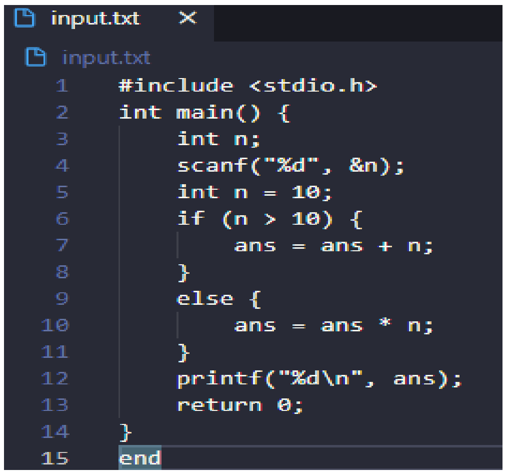
```
#include <stdio.h>

int main() {
    char str[100];
    fgets(str, sizeof(str), stdin);
    printf("You entered: %s", str);
    return 0;
}
end
```

OUTPUT:



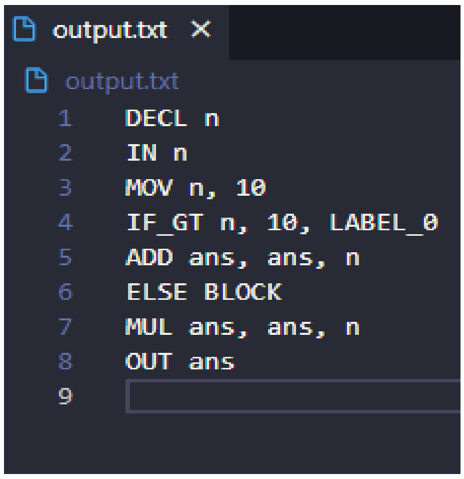
```
DECL str[100]
OUT str
```

INPUT: condition statements



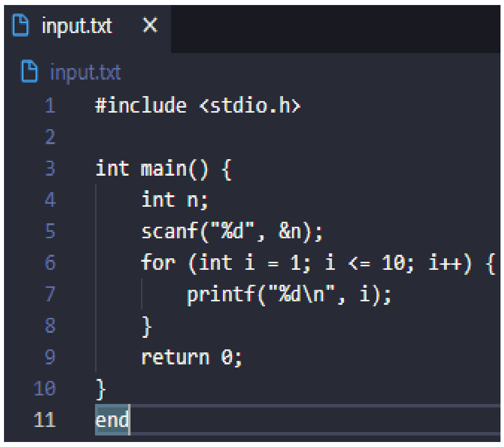
```
input.txt
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    int n = 10;
    if (n > 10) {
        ans = ans + n;
    }
    else {
        ans = ans * n;
    }
    printf("%d\n", ans);
    return 0;
}
end
```

OUTPUT:



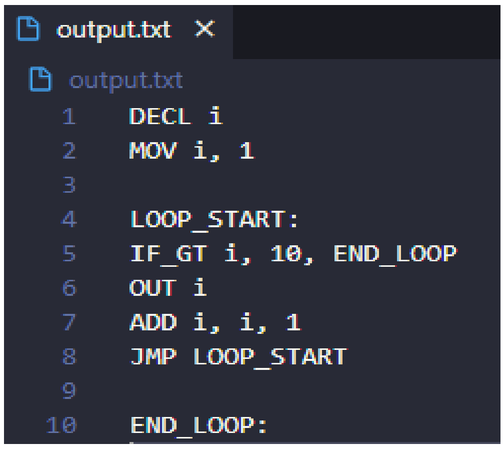
```
output.txt
DECL n
IN n
MOV n, 10
IF_GT n, 10, LABEL_0
ADD ans, ans, n
ELSE BLOCK
MUL ans, ans, n
OUT ans
```

INPUT: loop handle



```
input.txt
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
end
```

OUTPUT:



```
output.txt
DECL i
MOV i, 1

LOOP_START:
IF_GT i, 10, END_LOOP
OUT i
ADD i, i, 1
JMP LOOP_START

END_LOOP:
```

From analyzing the results, it has been shown that the converter can handle user input, variable declarations, basic if-else conditional statements, basic loops and basic data structures like Arrays and Strings properly. It takes the c code as input and then it reads it line by line to convert it into Assembly language pseudocode.

Chapter 4

Engineering Standards and Mapping

This chapter maps the project with societal impact, sustainability, and complex engineering problems.

4.1 Impact on Society, Environment and Sustainability

4.1.1 Impact on Life

- Enhances understanding of compiler functionality.
- Empowers both developers and students.
- Contributes to improved software quality.
- Leads to better performance in the long run.

4.1.2 Impact on Society & Environment

This project promotes better education, leading to more efficient software development. No direct environmental impact.

4.1.3 Ethical Aspects

The project promotes ethical coding by encouraging educational transparency and source understanding.

4.1.4 Sustainability Plan

The project is sustainable due to its simple and modular structure, making it easy to maintain and extend. It can be reused in future academic projects or enhanced with additional features like control flow, function handling, and arrays. This flexibility allows it to serve as a long-term educational tool for compiler design learning.

4.2 Project Management and Team Work

The project was managed through effective collaboration and task distribution among team members. Each member contributed to specific modules, including input parsing, pseudocode generation, testing, and documentation. Regular meetings and version control practices ensured consistency and progress tracking throughout the development process. The structured division of responsibilities not only improved productivity but also encouraged team members to take ownership of their individual contributions while aligning with the overall project goals.

4.3 Complex Engineering Problem

Translating C syntax into pseudocode required handling ambiguous syntax and edge cases accurately.

4.3.1 Mapping of Program Outcome

In this section, provide a mapping of the problem and provided solution with targeted Program Outcomes (PO's).

Table 4.1: Justification of Program Outcomes

PO's	Justification
PO1	Demonstrated understanding of compilation fundamentals
PO2	Designed and implemented the converter
PO3	Analyzed code patterns for translation

4.3.2 Complex Problem Solving

Requires design, abstraction, and real-time debugging to handle code translation effectively.

Table 4.2: Mapping with complex problem solving.

EP1 Dept of Knowledge	**EP2** Range of Conflicting Requiremen ts	**EP3** Depth of Analysis	**EP4** Familiarity of Issues	**EP5** Extent of Applicable Codes	**EP6** Extent Of Stakeholder Involvemen t	**EP7** Inter-dependence
Applies programming and compiler theory	Balances simplicity and extensibility	Breaks down statements and generates accurate pseudocode	Recognizes typical beginner challenges in parsing and translation	Follows syntax rules of C and principles of code generation	Low – Academic context	Depends on coordination between modules for parsing, processing, and generation

4.3.3 Engineering Activities

In this section, a mapping with engineering activities is provided. For each mapping add subsections to put rationale (Use Table 4.3).

Table 4.3: Mapping with complex engineering activities.

EA1 Range of resources	**EA2** Level of Interaction	**EA3** Innovation	**EA4** Consequences for society and environment	**EA5** Familiarity
Utilizes string parsing libraries and standard C++ I/O	Medium – Requires internal coordination between logic modules	Simplifies compiler concepts into a minimal working model	Low – Educational tool, indirect impact by improving learning outcomes	Educational settings are familiar with such tools for teaching compiler principles

Chapter 5

Conclusion

In this chapter the summary, limitation and the future works scope of the projects is discussed

5.1 Summary

This project successfully demonstrates the foundational concept of code translation by converting basic C language constructs into simplified assembly-style pseudocode. Developed in C++, the tool parses and interprets lines of C code—such as declarations, assignments, and input/output operations—and generates corresponding low-level representations. The implementation highlights the core principles of compiler back-end processing in a clear and accessible manner. Overall, the project fulfills its objective as an educational tool, offering valuable insights into how high-level programming languages are translated into machine-level instructions.

5.2 Limitation

- Supports only a limited set of C constructs (e.g., basic declarations, arithmetic operations, scanf, printf).
- Does not handle **Nested** control structures like while, or for loops.
- No support for functions, arrays, or complex expressions.
- Limited error handling for invalid or malformed C code.
- Pseudocode output is non-standard and meant for educational purposes only.

5.3 Future Work

- Extend support to control structures such as while, and for loops.
- Add functionality for functions, arrays, and nested expressions.
- Implement error detection and meaningful feedback for unsupported or incorrect syntax.
- Develop a graphical interface to visualize the translation process.
- Optimize pseudocode generation for better clarity and efficiency.

References

1. https://stackoverflow.com/questions/26091622/convert-c-program-into-assembly-code
2. https://www.tutorialspoint.com/assembly_programming/index.htm
3. https://www.w3schools.com/c/c_compiler.php
4. https://gist.github.com/lancejpollard/adf75b90137ef29e6f02

Github Source Code Link: https://github.com/masumbillah06/Compiler-Design-cse331/tree/main