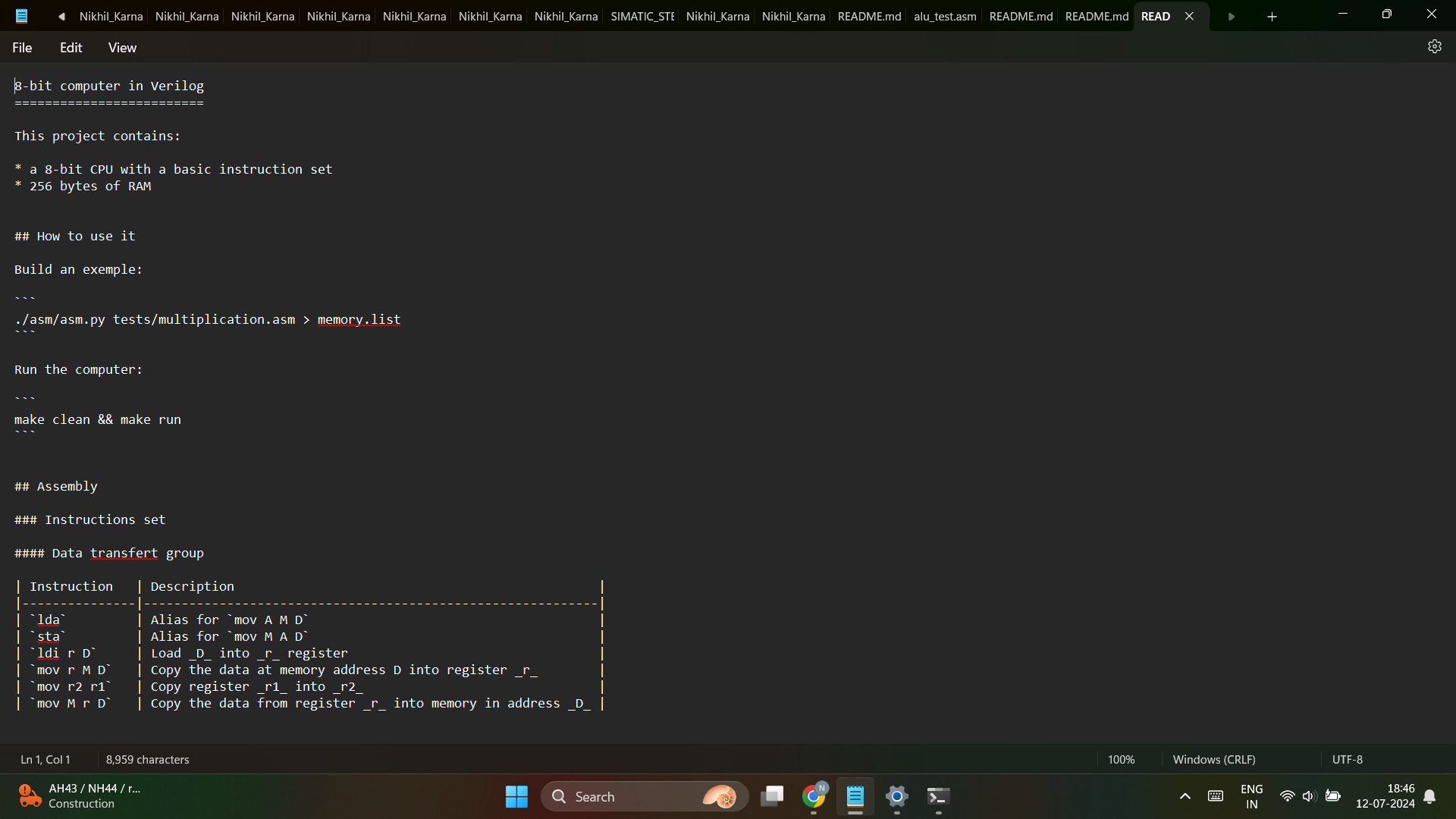
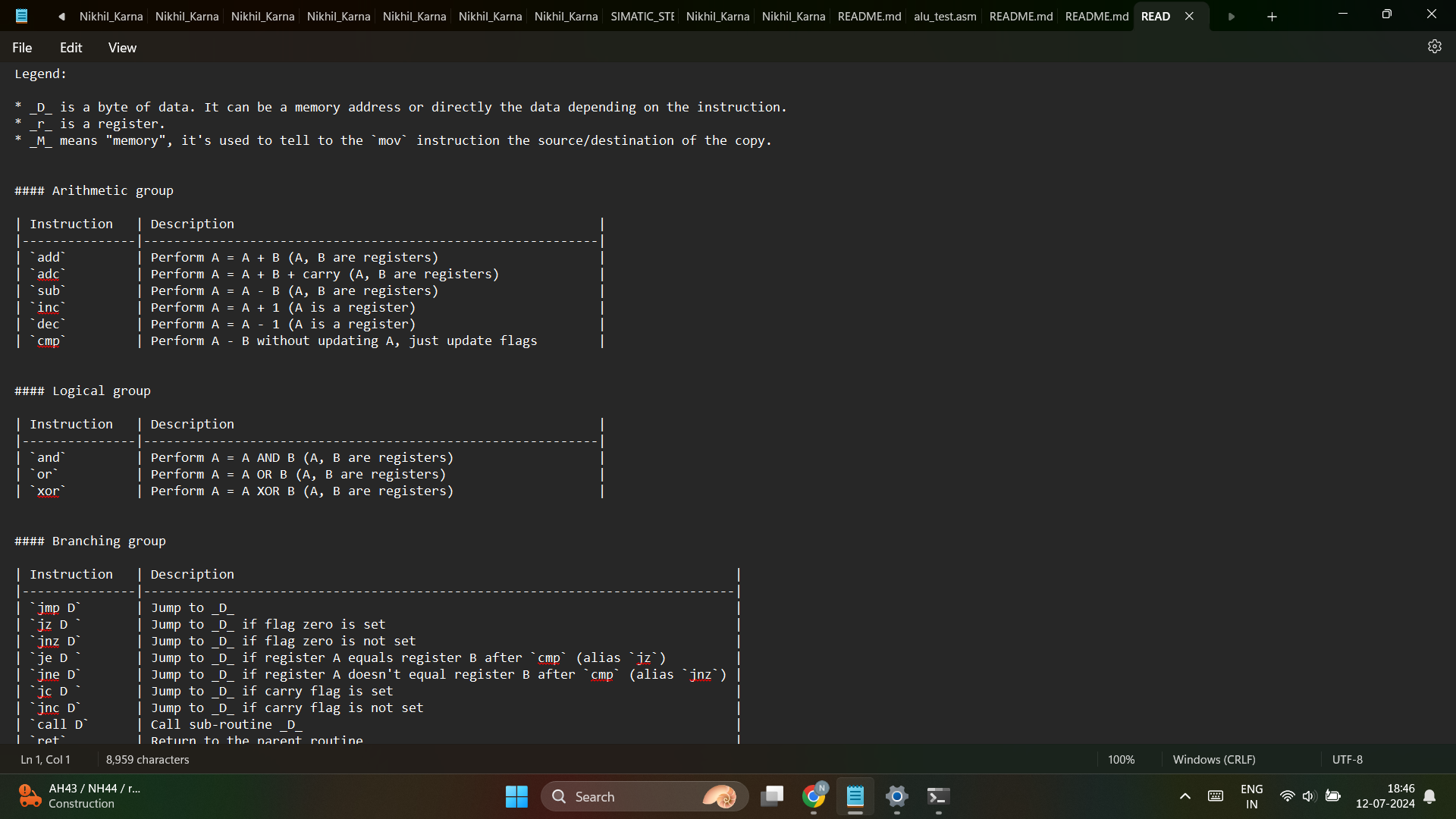
Documentation for SimpleLang Compiler

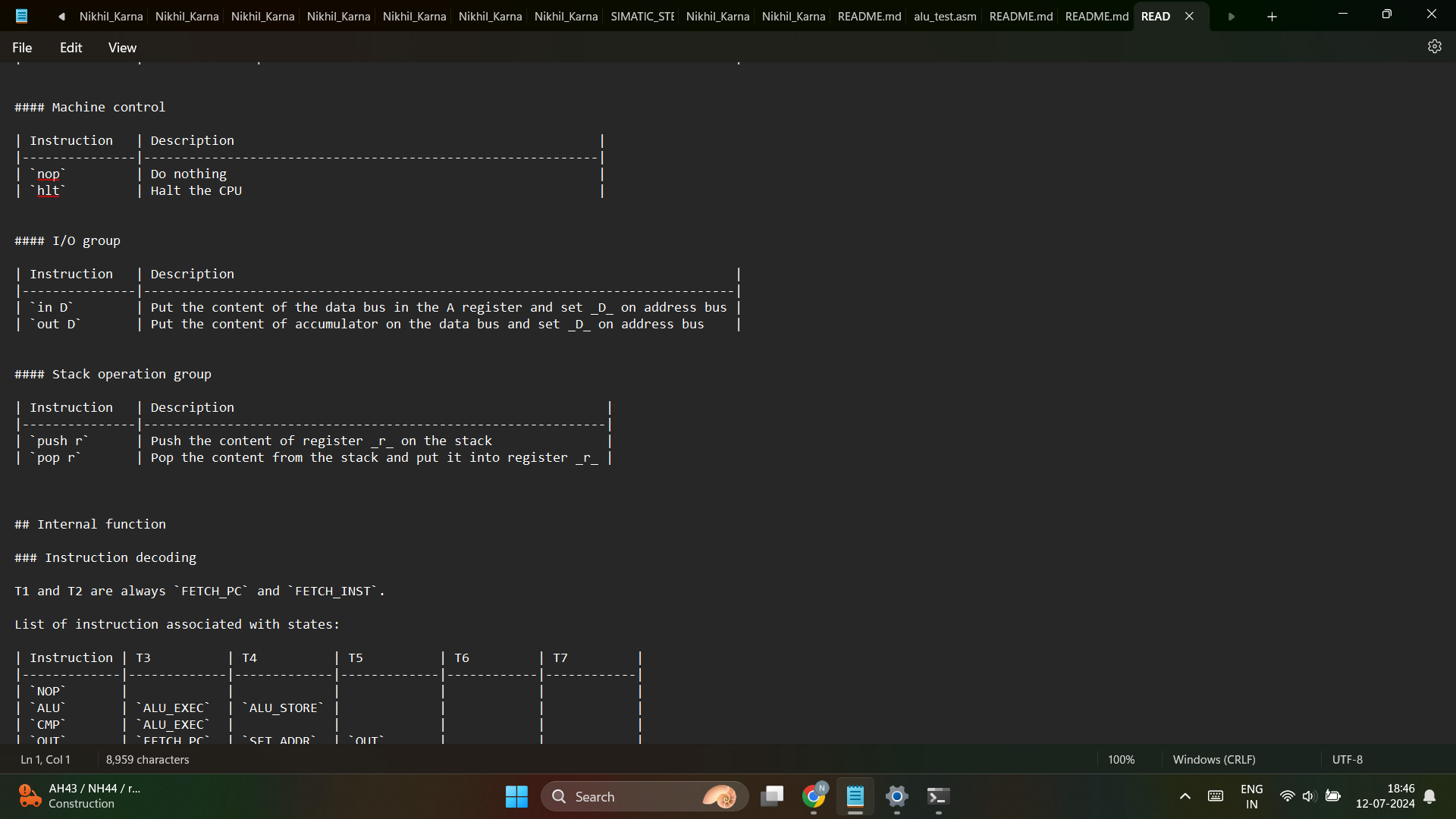
**Introduction**

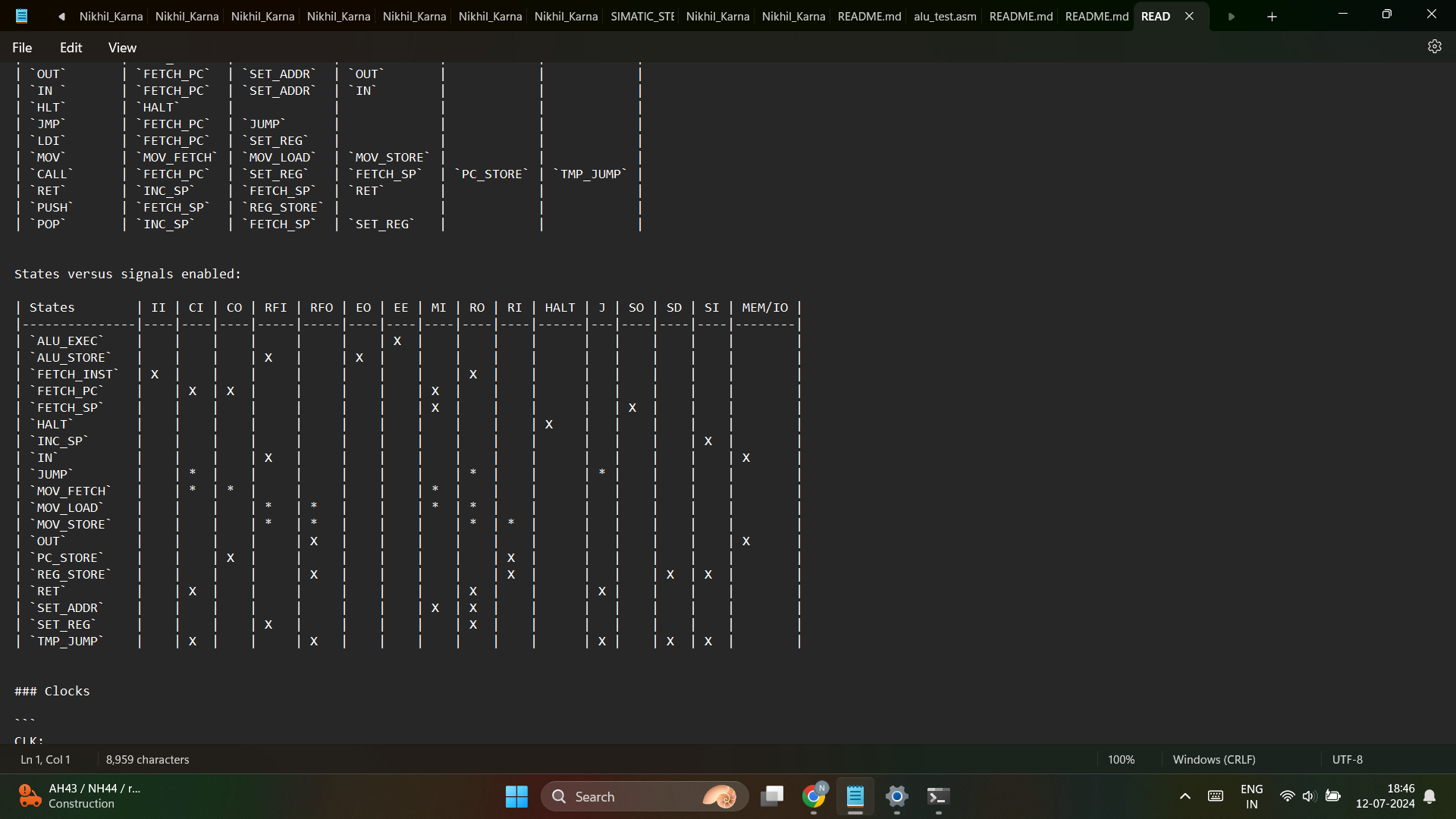
SimpleLang is a minimalistic high-level language designed to run on an 8-bit CPU. It includes basic constructs such as variable declarations, assignments, arithmetic operations, and conditional statements, but it does not include loops. This language aims to be easy to understand and implement for educational purposes.

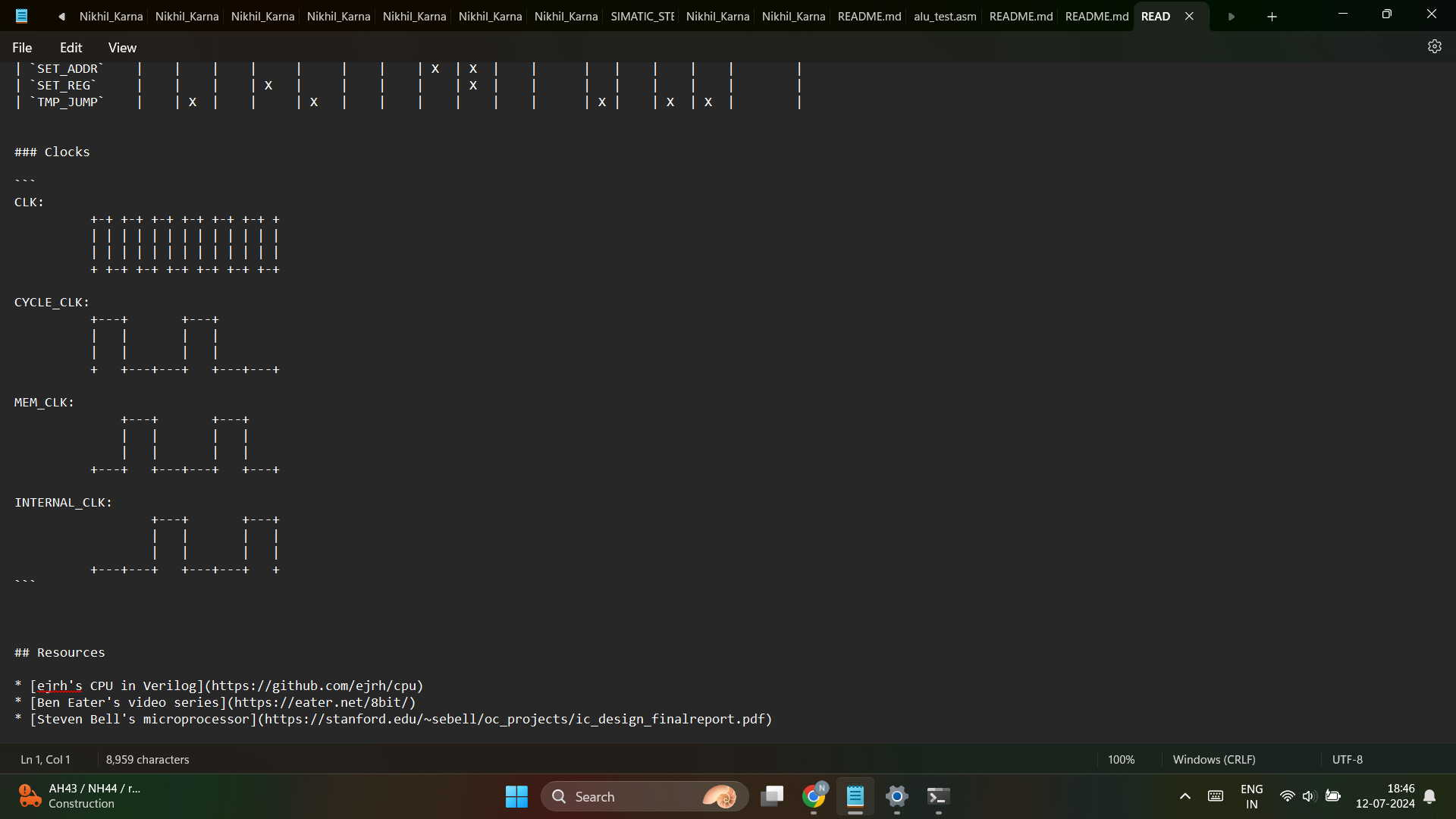
The SimpleLang compiler is designed to translate SimpleLang code into assembly code for an 8-bit CPU. This involves several stages, including lexical analysis, parsing, and code generation. The following documentation outlines the design and implementation details of each stage.











Design Overview

* **Lexer:** Tokenizes the input SimpleLang source code into a series of tokens.
* **Parser:** Analyzes the token sequence to generate an Abstract Syntax Tree (AST).
* **Code Generator:** Traverses the AST to produce assembly code for the 8-bit CPU.

Implementation Details

Lexer

* Purpose of the Lexer:

The lexer reads the source code and converts it into a sequence of tokens, which are the basic building blocks of the language.

* Token Types and Example Tokens:

Token Types: KEYWORD, IDENTIFIER, NUMBER, OPERATOR, DELIMITER, etc.

* Example Tokens:

int x; would be tokenized into [TOKEN\_KEYWORD, TOKEN\_IDENTIFIER, TOKEN\_DELIMITER]

lexer.h code :

#ifndef LEXER\_H

#define LEXER\_H

typedef enum {

TOKEN\_INT,

TOKEN\_IDENTIFIER,

TOKEN\_SEMICOLON,

TOKEN\_EOF

} TokenType;

typedef struct {

TokenType type;

char value[64];

} Token;

int tokenize(const char \*filename, Token \*tokens);

#endif

lexer.c code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "lexer.h"

int tokenize(const char \*filename, Token \*tokens) {

FILE \*file = fopen(filename, "r");

if (!file) {

fprintf(stderr, "Could not open file %s\n", filename);

return 0;

}

int index = 0;

while (fscanf(file, "%s", tokens[index].value) != EOF) {

tokens[index].type = TOKEN\_IDENTIFIER; // Simplified for example

index++;

}

fclose(file);

return index

Parser

The parser takes the tokens produced by the lexer and generates an Abstract Syntax Tree (AST). The AST represents the hierarchical structure of the source code.

**parser.h code:**

#ifndef PARSER\_H

#define PARSER\_H

#include "lexer.h"

typedef struct AstNode {

Token token;

struct AstNode \*left;

struct AstNode \*right;

} AstNode;

AstNode\* parse(Token \*tokens, int num\_tokens);

#endif

parser.c code:

#include <stdlib.h>

#include "parser.h"

AstNode\* parse(Token \*tokens, int num\_tokens) {

if (num\_tokens == 0) return NULL;

AstNode \*root = malloc(sizeof(AstNode));

root->token = tokens[0];

root->left = NULL;

root->right = NULL;

// Simplified example, usually more complex logic

if (num\_tokens > 1) {

root->right = parse(&tokens[1], num\_tokens - 1);

}

return root;

}

Code Generator

The code generator traverses the AST and generates the corresponding assembly code.

code\_generator.h**:-**

#ifndef CODE\_GENERATOR\_H

#define CODE\_GENERATOR\_H

#include "parser.h"

void generate\_assembly(AstNode \*ast, char \*assembly\_code);

#endif

code\_generator.c:

#include <stdio.h>

#include <string.h>

#include "code\_generator.h"

void generate\_assembly(AstNode \*ast, char \*assembly\_code) {

if (!ast) return;

// Simplified example

if (ast->token.type == TOKEN\_IDENTIFIER) {

strcat(assembly\_code, "LOAD ");

strcat(assembly\_code, ast->token.value);

strcat(assembly\_code, "\n");

}

generate\_assembly(ast->left, assembly\_code);

generate\_assembly(ast->right, assembly\_code);

}

Integration

integratecode.c code**:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "lexer.h"

#include "parser.h"

#include "code\_generator.h"

#define MAX\_TOKENS 100

#define MAX\_ASSEMBLY\_CODE 1000

int main() {

Token tokens[MAX\_TOKENS];

int num\_tokens = tokenize("example.sl", tokens);

if (num\_tokens == 0) {

fprintf(stderr, "Failed to tokenize the source code.\n");

return 1;

}

AstNode \*ast = parse(tokens, num\_tokens);

if (!ast) {

fprintf(stderr, "Failed to parse the tokens.\n");

return 1;

}

char assembly\_code[MAX\_ASSEMBLY\_CODE] = "";

generate\_assembly(ast, assembly\_code);

printf("Generated Assembly Code:\n%s\n", assembly\_code);

return 0;}

Conclusion:-

 Summary of Design Choices**:**

* Simplified lexer and parser for demonstration purposes.
* Focus on mapping high-level constructs to a minimal set of assembly instructions.

 Challenges Faced and Solutions**:**

* Handling syntax errors: Implemented basic error handling in the parser.
* Memory management: Ensured proper allocation and deallocation of AST nodes.

 Future Improvements and Extensions**:**

* Expand the SimpleLang syntax to include more complex constructs.
* Optimize the generated assembly code.
* Implement additional error handling and reporting features.