Universidad de Ingeniería y Tecnología

Facultad de Computación

Compilador de C a x86-64

Proyecto Final

CS3402 - Compiladores

Salazar Hillenbrand, Mauricio
Alvarado Vargas, Fabian Martín

Noviembre 2025

Índice

1. Introducción

El presente documento describe el diseño e implementación de un compilador que traduce un subconjunto del lenguaje C a código ensamblador x86-64. El compilador fue desarrollado en C++ siguiendo una arquitectura modular que comprende las fases clásicas de compilación: análisis léxico, análisis sintáctico, análisis semántico y generación de código.

1.1. Objetivos

- Implementar un compilador funcional para un subconjunto de C
- Generar código ensamblador x86-64 ejecutable
- Aplicar técnicas de optimización en tiempo de compilación
- Soportar tipos de datos numéricos adicionales (float, long, unsigned int)
- Implementar extensiones del lenguaje (expresiones ternarias, typedef)

1.2. Alcance

El compilador soporta las siguientes características del lenguaje C:

- Declaración de variables con tipos int, long, unsigned int y float
- Operaciones aritméticas, relacionales y lógicas
- Estructuras de control: if-else, while, for
- Definición y llamada de funciones, incluyendo recursión
- Expresiones ternarias (operador ?:)
- Alias de tipos mediante typedef

2. Repositorio del Proyecto

El código fuente completo, casos de prueba y scripts adicionales se encuentran disponibles en el repositorio oficial del proyecto:

- **GitHub:** `https://github.com/mauriciosalazarsh/compialdoresfinal.git`

3. Arquitectura del Compilador

El compilador sigue una arquitectura de pipeline con cuatro fases principales, cada una implementada como un módulo independiente.

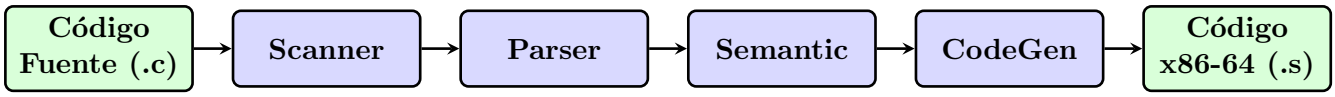

Figura 1: Pipeline del compilador

3.1. Estructura de Archivos

Listing 1: Estructura del proyecto

```
Final/
|-- src/
|   |-- main.cpp          # Punto de entrada
|   |-- scanner.cpp       # Analizador l xico
|   |-- parser.cpp        # Analizador sint ctico
|   |-- semantic.cpp      # Analizador sem ntico
|   |-- codegen.cpp       # Generador de c digo
|   |-- ast.cpp           # Nodos del AST
|   |-- symboltable.cpp   # Tabla de s mbolos
|-- include/
|   |-- scanner.h
|   |-- parser.h
|   |-- ast.h
|   |-- visitor.h
|   |-- semantic.h
|   |-- codegen.h
|   |-- symboltable.h
|-- tests/                # 18 casos de prueba
|-- visualizer/           # Bonus: visualizador web
```

4. Análisis Léxico (Scanner)

El analizador léxico convierte el código fuente en una secuencia de tokens. Está implementado en la clase `Scanner` utilizando un enfoque de análisis carácter por carácter.

4.1. Clase Scanner

Listing 2: Estructura principal del Scanner

```
class Scanner {
private:
    std::string input;
    size_t pos;
    int line, column;
    char current;

    void advance();
    char peek(int offset = 1);
    void skipWhitespace();
```

```
    void skipComment();
    void skipPreprocessor();
    Token scanNumber();
    Token scanIdentifier();
    Token scanString();

public:
    Scanner(const std::string& src);
    Token nextToken();
    std::vector<Token> tokenize();
};
```

4.2. Tipos de Tokens

El scanner reconoce los siguientes tipos de tokens:

Categoria	Tokens	Ejemplos
Palabras clave	int, long, float, unsigned, void	`int x;`
	if, else, while, for, return	`if (x >0)`
	typedef	`typedef int entero;`
Literales	NUM, FLOAT_LIT, STRING_LIT	`42`, `3.14`, `"hola"`
Identificadores	ID	`variable`, `función`
Operadores	+, -, *, /, %, ==, !=, ¡, ¿, ¡=, ¿=	`a + b`
Delimitadores	(,), {, }, [,], ;, ,	`func(a, b)`

Cuadro 1: Tipos de tokens reconocidos

4.3. Manejo de Números

El scanner detecta automáticamente el tipo de literal numérico:

Listing 3: Escaneo de literales numéricos

```
Token Scanner::scanNumber() {
    std::string num;
    bool isFloat = false;

    while (std::isdigit(current)) {
        num += current;
        advance();
    }

    // Detectar punto decimal
    if (current == '.' && std::isdigit(peek())) {
        isFloat = true;
        num += current;
        advance();
        while (std::isdigit(current)) {
            num += current;
            advance();
```

```
        }
    }

    return Token(isFloat ? TokenType::FLOAT_LIT
                         : TokenType::NUM, num);
}
```

5. Análisis Sintáctico (Parser)

El analizador sintáctico implementa un parser recursivo descendente que construye un Árbol de Sintaxis Abstracta (AST).

5.1. Clase Parser

Listing 4: Estructura del Parser

```
class Parser {
private:
    std::vector<Token> tokens;
    size_t current;
    std::map<std::string, DataType> typeAliases;

    Token peek(int offset = 0);
    Token advance();
    bool match(TokenType type);
    bool check(TokenType type);
    void expect(TokenType type, const std::string& msg);

    DataType parseType();
    void parseTypedef();

    std::unique_ptr<Expr> parseExpression();
    std::unique_ptr<Expr> parseTernary();
    std::unique_ptr<Expr> parseLogicalOr();

public:
    std::unique_ptr<Program> parse();
};
```

5.2. Jerarquía del AST

El AST utiliza el patrón Visitor para permitir múltiples recorridos:

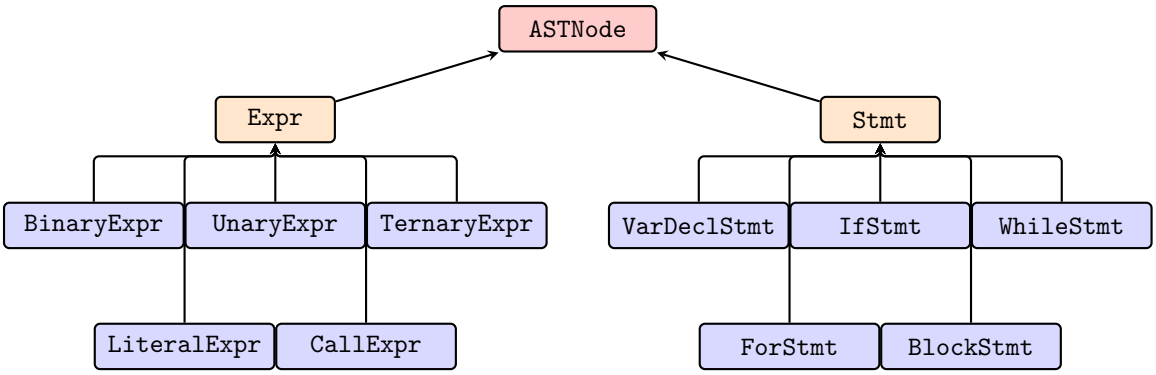

Figura 2: Jerarquía de clases del AST

5.3. Precedencia de Operadores

La precedencia se implementa mediante funciones de parsing separadas:

Nivel	Operadores	Función
1 (menor)	?: (ternario)	`parseTernary()`
2	——	`parseLogicalOr()`
3	&&	`parseLogicalAnd()`
4	==, !=	`parseEquality()`
5	¡, ¿, ¡=, ¿=	`parseRelational()`
6	+, -	`parseAdditive()`
7	*, /, %	`parseMultiplicative()`
8 (mayor)	-, ! (unarios)	`parseUnary()`

Cuadro 2: Niveles de precedencia

5.4. Soporte para Typedef

El parser mantiene un mapa de alias de tipos:

Listing 5: Implementación de typedef

```
void Parser::parseTypedef() {
    expect(TokenType::TYPEDEF, "Expected 'typedef'");
    DataType baseType = parseType();

    std::string aliasName = peek().lexeme;
    expect(TokenType::ID, "Expected type alias name");
    expect(TokenType::SEMICOLON, "Expected ';'");

    // Almacenar el alias
    typeAliases[aliasName] = baseType;
}

bool Parser::isTypeToken() {
    if (check(TokenType::INT) || check(TokenType::LONG) ||
        check(TokenType::FLOAT) || check(TokenType::UNSIGNED)) {
        return true;
```

```
    }
    // Verificar si es un alias de typedef
    if (check(TokenType::ID)) {
        return typeAliases.find(peek().lexeme)
               != typeAliases.end();
    }
    return false;
}
```

6. Análisis Semántico

El analizador semántico verifica la corrección del programa mediante el patrón Visitor.

6.1. Clase SemanticAnalyzer

Listing 6: Estructura del analizador semántico

```
class SemanticAnalyzer : public Visitor {
private:
    SymbolTable& symbolTable;
    DataType currentFunctionReturnType;
    bool hasErrors;
    std::string errors;

    bool areTypesCompatible(DataType expected,
                            DataType actual);
    DataType getCommonType(DataType t1, DataType t2);
    void error(const std::string& message);

public:
    void visit(BinaryExpr* node) override;
    void visit(TernaryExpr* node) override;
    void visit(CallExpr* node) override;
    void visit(VarDeclStmt* node) override;
    void visit(FunctionDecl* node) override;
};
```

6.2. Sistema de Tipos

El compilador soporta cuatro tipos numéricos con promoción automática:

Listing 7: Promoción de tipos

```
DataType SemanticAnalyzer::getCommonType(DataType t1,
                                         DataType t2) {
    if (t1 == t2) return t1;

    // Float domina sobre todos
    if (t1 == DataType::FLOAT || t2 == DataType::FLOAT)
        return DataType::FLOAT;
```

```
8
9       // Long domina sobre Int
10      if (t1 == DataType::LONG || t2 == DataType::LONG)
11          return DataType::LONG;
12
13      // UINT con INT -> LONG (para evitar overflow)
14      if ((t1 == DataType::UINT && t2 == DataType::INT) ||
15          (t1 == DataType::INT && t2 == DataType::UINT))
16          return DataType::LONG;
17
18      return t1;
19  }
```

6.3. Tabla de Símbolos

La tabla de símbolos maneja ámbitos anidados mediante una pila de scopes:

Listing 8: Estructura de la tabla de símbolos

```
1   struct Symbol {
2       std::string name;
3       DataType type;
4       bool isMutable;
5       bool isParameter;
6       int offset;  // Offset en el stack frame
7       std::vector<int> arrayDimensions;
8   };
9
10  class SymbolTable {
11  private:
12      std::vector<std::map<std::string, Symbol>> scopes;
13      std::map<std::string, FunctionSymbol> functions;
14      int currentOffset;
15
16  public:
17      void enterScope();
18      void exitScope();
19      bool declareVariable(const std::string& name,
20                           const Symbol& sym);
21      Symbol* lookup(const std::string& name);
22      int allocateStackSpace(int size);
23  };
```

7. Generación de Código

El generador de código produce ensamblador x86-64 en sintaxis Intel, siguiendo la convención de llamadas System V ABI.

7.1. Clase CodeGenerator

Listing 9: Estructura del generador de código

```
class CodeGenerator : public Visitor {
private:
    SymbolTable& symbolTable;
    std::stringstream code;
    std::stringstream dataSection;
    int labelCounter;
    int stringCounter;
    bool enableConstantFolding;
    bool enableDeadCodeElimination;

    std::string newLabel();
    void emit(const std::string& instruction);
    void emitLabel(const std::string& label);
    void generatePrologue(const std::string& func,
                          int stackSize);
    void generateEpilogue();

public:
    void visit(BinaryExpr* node) override;
    void visit(CallExpr* node) override;
    void visit(IfStmt* node) override;

    std::string getCode() const;
};
```

7.2. Convención de Llamadas (System V ABI)

Registro	**Uso**
rax	Valor de retorno, acumulador
rbx	Registro temporal (callee-saved)
rdi, rsi, rdx, rcx, r8, r9	Argumentos 1-6
rbp	Base pointer (marco de pila)
rsp	Stack pointer
xmm0-xmm1	Operaciones de punto flotante

Cuadro 3: Uso de registros x86-64

7.3. Estructura del Stack Frame

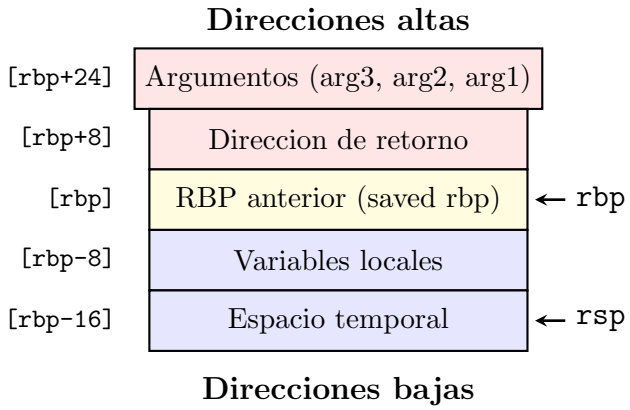

Figura 3: Estructura del stack frame en x86-64

7.4. Generación de Prólogo y Epílogo

Listing 10: Generación del prólogo de función

```
void CodeGenerator::generatePrologue(const std::string& func,
                                     int stackSize) {
    emitLabel(func);
    emit("push rbp");
    emit("mov rbp, rsp");

    // Alinear stack a 16 bytes (requerido por ABI)
    int alignedSize = ((stackSize + 15) & ~15);
    if (alignedSize > 0) {
        emit("sub rsp, " + std::to_string(alignedSize));
    }
}

void CodeGenerator::generateEpilogue() {
    emit("mov rsp, rbp");
    emit("pop rbp");
    emit("ret");
}
```

7.5. Generación de Expresiones Binarias

Listing 11: Generación de operaciones aritméticas

```
void CodeGenerator::visit(BinaryExpr* node) {
    // Evaluar operandos
    node->left->accept(this);
    emit("push rax");  // Guardar operando izquierdo

    node->right->accept(this);
```

```
    emit("mov rbx, rax");  // Operando derecho en rbx
    emit("pop rax");       // Operando izquierdo en rax

    // Generar operaci n
    if (node->op == "+") {
        emit("add rax, rbx");
    } else if (node->op == "-") {
        emit("sub rax, rbx");
    } else if (node->op == "*") {
        emit("imul rax, rbx");
    } else if (node->op == "/") {
        emit("cqo");         // Sign extend rax to rdx:rax
        emit("idiv rbx");    // Resultado en rax
    } else if (node->op == "%") {
        emit("cqo");
        emit("idiv rbx");
        emit("mov rax, rdx"); // Resto en rdx
    }
}
```

7.6. Generación de Expresiones Ternarias

Listing 12: Generación del operador ternario

```
void CodeGenerator::visit(TernaryExpr* node) {
    std::string falseLabel = newLabel();
    std::string endLabel = newLabel();

    // Evaluar condici n
    node->condition->accept(this);
    emit("test rax, rax");
    emit("jz " + falseLabel);

    // Rama verdadera
    node->trueExpr->accept(this);
    emit("jmp " + endLabel);

    // Rama falsa
    emitLabel(falseLabel);
    node->falseExpr->accept(this);

    emitLabel(endLabel);
}
```

7.7. Generación de Estructuras de Control

Listing 13: Generación de if-else

```
void CodeGenerator::visit(IfStmt* node) {
    std::string elseLabel = newLabel();
```

```
    std::string endLabel = newLabel();

    node->condition->accept(this);
    emit("test rax, rax");

    if (node->elseBranch) {
        emit("jz " + elseLabel);
        node->thenBranch->accept(this);
        emit("jmp " + endLabel);
        emitLabel(elseLabel);
        node->elseBranch->accept(this);
        emitLabel(endLabel);
    } else {
        emit("jz " + endLabel);
        node->thenBranch->accept(this);
        emitLabel(endLabel);
    }
}
```

Listing 14: Generación de bucle while

```
void CodeGenerator::visit(WhileStmt* node) {
    std::string startLabel = newLabel();
    std::string endLabel = newLabel();

    emitLabel(startLabel);
    node->condition->accept(this);
    emit("test rax, rax");
    emit("jz " + endLabel);

    node->body->accept(this);
    emit("jmp " + startLabel);

    emitLabel(endLabel);
}
```

8. Optimizaciones

El compilador implementa dos optimizaciones principales aplicadas durante la generación de código.

8.1. Constant Folding (Plegado de Constantes)

Esta optimización evalúa expresiones constantes en tiempo de compilación:

Listing 15: Implementación de constant folding

```
void CodeGenerator::visit(BinaryExpr* node) {
    if (enableConstantFolding) {
        auto leftLit = dynamic_cast<LiteralExpr*>(
                            node->left.get());
```

```
        auto rightLit = dynamic_cast<LiteralExpr*>(
                            node->right.get());

        if (leftLit && rightLit &&
            node->exprType != DataType::FLOAT) {
            long long leftVal = std::stoll(leftLit->value);
            long long rightVal = std::stoll(rightLit->value);
            long long result = 0;

            if (node->op == "+")
                result = leftVal + rightVal;
            else if (node->op == "-")
                result = leftVal - rightVal;
            else if (node->op == "*")
                result = leftVal * rightVal;
            else if (node->op == "/" && rightVal != 0)
                result = leftVal / rightVal;

            // Emitir resultado constante directamente
            emit("mov rax, " + std::to_string(result));
            return;
        }
    }
}
```

Ejemplo:

```
int x = 5 + 10 * 2;  // Se eval a como 25 en compilaci n
```

Código generado optimizado:

```
mov rax, 25          ; En lugar de m ltiples operaciones
```

8.2. Dead Code Elimination (Eliminación de Código Muerto)

El compilador detecta y omite código que nunca se ejecutará:

Listing 16: Eliminación de código muerto en if

```
void CodeGenerator::visit(IfStmt* node) {
    // Verificar si la condici n es constante
    if (auto lit = dynamic_cast<LiteralExpr*>(
                        node->condition.get())) {
        if (lit->value == "0") {
            // Condici n siempre falsa: solo generar else
            if (node->elseBranch) {
                node->elseBranch->accept(this);
            }
            return;  // Omitir rama then
        } else {
            // Condici n siempre verdadera: solo generar then
            node->thenBranch->accept(this);
            return;  // Omitir rama else
```

```
        }
    }
}
```

Ejemplo:

```
if (0) {
    z = 100;  // Este c digo nunca se ejecuta
}
```

El compilador no genera código para el bloque interno.

9. Extensiones Implementadas

9.1. Expresiones Ternarias

El operador ternario `?:` permite expresiones condicionales en línea:

Listing 17: Ejemplo de expresión ternaria

```
int max = (x > y) ? x : y;
int result = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

La implementación en el parser:

```
std::unique_ptr<Expr> Parser::parseTernary() {
    auto expr = parseLogicalOr();

    if (match(TokenType::QUESTION)) {
        auto trueExpr = parseExpression();
        expect(TokenType::COLON, "Expected ':'");
        auto falseExpr = parseExpression();
        return std::make_unique<TernaryExpr>(
            std::move(expr),
            std::move(trueExpr),
            std::move(falseExpr)
        );
    }

    return expr;
}
```

9.2. Alias de Tipo (typedef)

Permite crear nombres alternativos para tipos existentes:

Listing 18: Ejemplo de typedef

```
typedef int entero;
typedef long numero_grande;
typedef float decimal;

entero x = 10;
numero_grande big = 1000000;
```

```
decimal pi = 3.14;
```

El parser mantiene un mapa `typeAliases` que asocia nombres de alias con sus tipos base.

10. Casos de Prueba

El compilador incluye 18 casos de prueba organizados en cuatro categorías.

10.1. Tests de Funciones (3 casos)

Test	Descripción	Salida
test_func1.c	Función suma simple	15
test_func2.c	Función cuadrado anidada	49
test_func3.c	Factorial recursivo	120

Listing 19: test_func3.c - Factorial recursivo

```c
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int result = factorial(5);
    printf("%d\n", result);  // Output: 120
    return 0;
}
```

10.2. Tests de Implementación Base (5 casos)

Test	Descripción	Salida
test_base1.c	Aritmética básica	30, 10, 200, 2
test_base2.c	Sentencias if-else	10, 1
test_base3.c	Bucle while	45
test_base4.c	Bucle for	45
test_base5.c	Expresiones complejas	11, 16, 4

10.3. Tests de Extensiones (5 casos)

Test	Descripción	Salida
test_ext1.c	Expresiones ternarias	10, 5
test_ext2.c	Unsigned int	150
test_ext3.c	Long int	3000000
test_ext4.c	Float	6.280000
test_ext5.c	Typedef	30, 1000000, 3.140000

10.4. Tests de Optimización (5 casos)

Test	Descripción	Salida
test_opt1.c	Constant folding	5, 50, 25
test_opt2.c	Dead code elimination	30
test_opt3.c	Strength reduction	16, 32
test_opt4.c	Common subexpression	16
test_opt5.c	Loop optimization	100

11. Ejemplo de Código Generado

11.1. Código Fuente

Listing 20: Programa factorial

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int result = factorial(5);
    printf("%d\n", result);
    return 0;
}
```

11.2. Código Ensamblador Generado

Listing 21: Código x86-64 generado

```
.intel_syntax noprefix
.text
.global main

```

```
factorial:
    push rbp
    mov rbp, rsp
    mov rax, [rbp + 16]       ; Cargar par metro n
    push rax
    mov rax, 1
    mov rbx, rax
    pop rax
    cmp rax, rbx
    setle al
    movzx rax, al
    test rax, rax
    jz .L1                    ; Si n > 1, saltar
    mov rax, 1                ; return 1
    mov rsp, rbp
    pop rbp
    ret
.L1:
    mov rax, [rbp + 16]       ; Cargar n
    push rax
    sub rsp, 8                ; Alinear stack
    mov rax, [rbp + 16]
    push rax
    mov rax, 1
    mov rbx, rax
    pop rax
    sub rax, rbx              ; n - 1
    push rax
    call factorial            ; Llamada recursiva
    add rsp, 16
    mov rbx, rax              ; resultado en rbx
    pop rax                   ; n en rax
    imul rax, rbx             ; n * factorial(n-1)
    mov rsp, rbp
    pop rbp
    ret

main:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    sub rsp, 8
    mov rax, 5
    push rax
    call factorial
    add rsp, 16
    mov [rbp - 8], rax        ; result = factorial(5)
    mov rax, [rbp - 8]
    push rax
    lea rax, [.STR0]
    push rax
```

```
    pop rdi
    pop rsi
    xor eax, eax
    call printf
    mov rax, 0
    mov rsp, rbp
    pop rbp
    ret

.data
int_fmt: .asciz "%ld\n"
.STR0: .asciz "%d\n"
```

12. Instrucciones de Uso

12.1. Compilación del Compilador

```
# Compilar el compilador
make

# Limpiar archivos generados
make clean
```

12.2. Ejecución

Linux x86-64:

```
./compiler archivo.c
gcc -no-pie output.s -o program
./program
```

macOS (Apple Silicon):

```
./run_x86.sh archivo.c
```

Windows (WSL2):

```
./compiler archivo.c
gcc -no-pie output.s -o program
./program
```

13. Conclusiones

Se logró implementar exitosamente un compilador funcional que traduce un subconjunto significativo de C a código ensamblador x86-64. Los principales logros incluyen:

- Implementación completa de las cuatro fases del compilador
- Soporte para múltiples tipos de datos con promoción automática

- Manejo correcto de funciones recursivas y estructuras de control
- Optimizaciones efectivas que reducen el código generado
- Extensiones útiles como expresiones ternarias y typedef

El proyecto demuestra la aplicación práctica de los conceptos teóricos de compiladores, incluyendo análisis léxico, parsing recursivo descendente, verificación de tipos, generación de código y optimizaciones.

13.1. Trabajo Futuro

Posibles mejoras para versiones futuras:

- Soporte para estructuras (struct) y uniones
- Implementación de punteros y memoria dinámica
- Optimizaciones adicionales (register allocation, peephole)
- Soporte para más operadores de C
- Generación de código para otras arquitecturas

14. Referencias

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
2. Intel Corporation. (2021). *Intel 64 and IA-32 Architectures Software Developer's Manual*.
3. System V Application Binary Interface: AMD64 Architecture Processor Supplement.
4. Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.