Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп’ютерних наук та штучного інтелекту
Кафедра комп’ютерних систем та робототехніки

«Затверджую»
в.о. завідуючого кафедри
комп’ютерних систем та робототехніки
_____________ к. ф.-м. н., доцент Максим Хруслов
«___» грудня 2024 р.

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «**Модель оптимізації алгоритмів трансляції асемблерних програм**»

Спеціальність 123 – Комп’ютерна інженерія.
Галузь знань: 12 – Інформаційні технології.
Освітня програма «Комп’ютерна інженерія».

Захищено на засіданні
Екзаменаційної комісії № 42
протокол № __ від __.12.2024 р.
Оцінка _____ / ______________
Голова Екзаменаційної комісії
_____________ **СКОБ Ю. О.**

Виконав:
Студент(ка) групи КІ–61
Комеристий Владислав Сергійович

____

Керівник: доцент кафедри комп’ютерних систем та робототехніки, кандидат технічних наук
Рева Сергій Миколайович

____

Рецензент: доцент кафедри математичного моделювання та аналізу даних, к. ф.-м. н.
СПОРОВ Олександр Євгенович

____

Харків – 2024

АНОТАЦІЯ

Пояснювальна записка до магістерської атестаційної роботи складається зі вступу, трьох розділів, висновків, списку використаних джерел і двох додатків. Загальний обсяг роботи складає 74 сторінки, із яких 53 сторінок основної частини з 20 рисунками, 2 таблицями, списку використаних джерел із 20 найменувань та чотирма додатками.

Метою кваліфікаційної роботи є аналіз алгоритмів трансляції асемблерних програм, пошук варіантів їх адаптації та оптимізації і створення нового транслятора на основі знайдених рішень.

Об'єкт дослідження – транслятори та алгоритми трансляції асемблерних програм.

Предмет дослідження – способи управління процесом трансляції та методи адаптації алгоритмів транслятора до роботи із зовнішньою таблицею мнемонічних позначень команд та до взаємодії з різними видами операційних систем.

Проблема, яка вирішується у кваліфікаційній роботі полягає в тому, щоб на основі аналізу існуючих рішень та алгоритмів трансляції розробити власні алгоритми і створити програмне забезпечення для заміни існуючого застарілого та надати можливість студентам виконувати лабораторні роботи під різними операційними системами без використання віртуальної машини.

Область застосування – навчальний процес науково-навчального інституту комп'ютерних наук та штучного інтелекту Харківського національного університету імені В.Н.Каразіна. Нове програмне забезпечення вирішить питання мультиплатформеності, що дасть можливість більшості студентів використовувати дану розробку на своїх комп'ютерах, незалежно від типу операційної системи.

Ключові слова: транслятор асемблера, алгоритм, система команд, мнемонічне позначення, макропроцедура, C++.

ABSTRACT

The explanatory note to the master's certification work consists of an introduction, three sections, conclusions, a list of sources used and two appendices. The total volume of work is 74 pages, of which 53 pages of the main part with 20 figures, 2 tables, a list of used sources with 20 titles and four appendices.

The purpose of the qualification work is to analyze the algorithms for translating assembler programs, search for options for their adaptation and optimization, and create a new translator based on the solutions found.

Object of research: translators and algorithms for translating assembler programs.

The subject of research is the ways of controlling the translation process and methods of adapting the translator's algorithms to work with an external table of mnemonic designations of commands and to interact with various types of operating systems.

The problem that is solved in the qualification work is to develop own algorithms based on the analysis of existing solutions and translation algorithms and create software to replace the existing outdated one and enable students to perform laboratory work under different operating systems without using a virtual machine.

The field of application is the educational process of the Scientific and Educational Institute of Computer Science and Artificial Intelligence of V.N. Karazin Kharkiv National University. The new software will solve the issue of multiplatform, which will enable most students to use this development on their computers, regardless of the type of operating system.

Keywords: assembler translator, algorithm, command system, mnemonic designation, macro procedure, C++.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ.. 5
ВСТУП 6
РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ ЗАСОБІВ 9
1.1. Огляд TASM 9
1.2. Транслятор VASM 11
1.3. Транслятор GNU AS 12
1.4. Транслятор FASM 13
1.5. Транслятор NASM 14
1.6. LLVM with TableGen 15
Висновки за розділом 1... 16
РОЗДІЛ 2. РОЗРОБКА НОВОГО ТРАНСЛЯТОРА 18
2.1. Аналіз алгоритму трансляції... 18
2.2. Аналіз файлу із системою команд... 24
2.3. Створення алгоритму нового транслятора... 27
2.4. Розробка нового транслятора... 30
Висновки за розділом 2... 40
РОЗДІЛ 3. ВІДЛАГОДЖЕННЯ, ТЕСТУВАННЯ ТРАНСЛЯТОРА ТА РОЗРОБКА ОПИСУ КОРИСТУВАЧА... 42
3.1. Розробка методики випробувань... 42
3.2. Тестування та відлагодження транслятора 45
3.3. Опис користувача та синтаксичні правила 47
Висновки за розділом 3... 53
ВИСНОВКИ.. 55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.. 58
ДОДАТКИ... 61

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

ОС – операційна система

ПЗ – програмне забезпечення

VASM – Visual Assembler

FASM – Flat Assembler

NASM – Netwide Assembler

TASM – Telemark Assembler

ВСТУП

Кожен спеціаліст в області комп'ютерних наук знає, що всі електронні обчислювальні машини (ЕОМ) використовують двійкову систему числення, так звану машинну мову. Вона складається лише з 2 символів: 0 та 1. Різна послідовність цих значень вказує на різні команди, які комп'ютер повинен розуміти та виконувати. Проте програмісти пишуть програми на спеціально розроблених мовах програмування, а не на машинній мові. Щоб комп'ютер міг розуміти та виконувати поставлені задачі, весь написаний код необхідно перетворити на послідовність двійкових чисел. Для цієї цілі були розроблені компілятори та транслятори.

На факультеті комп'ютерних наук (нині ННІ комп'ютерних систем та штучного інтелекту) вже довгий час викладаються основи будови процесорів та їх функціонування, а також основи програмування, базуючись на використанні простої моделі процесора, створеної викладачами та студентами [1]. Рік тому було створено емулятор [2] – віртуальне обладнання, яке на сьогоднішній день допомагає студентам виконувати практичні та лабораторні роботи. В рамках вивчення основ програмування здобувачам освіти пропонується створювати програми мовою асемблер для вищезазначеної моделі процесора. Особливість полягає в тому, що студенти повинні розробити власний асемблер, створивши систему мнемонічних позначень для команд процесора. Асемблер – це мова програмування низького рівня, яка використовує мнемонічні позначення для інструкцій процесора замість двійкового коду [3].

Для трансляції асемблерного коду в машинну мову на сьогоднішній день використовується TASM (Telemark Assembler, v2.5) – програмне забезпечення, розроблене Томасом Андерсеном, який працював на той час у Speech Technology Incorporated, у 1987 році [4]. Всупереч тому, що цій програмі вже майже 40 років, вона доволі успішно виконує свої задачі і надає

функціональні можливості, які необхідні у навчальному процесі. Наразі існує безліч сучасних трансляторів, проте жоден з них не може повністю замінити TASM.

Актуальність роботи. Останні 5-7 років виявились складними для українців: спочатку пандемія, потім повномасштабна війна. В таких умовах online-навчання стало єдиним варіантом продовжувати здобувати освіту. Це призвело до необхідності замінювати фізичне обладнання, яке використовувалось під час аудиторних занять на програмні емулятори, на різні демонстраційні програми, відео лекції тощо. Для виконання студентами лабораторного практикуму потрібно було створювати програмну модель того обладнання, яке існувало раніше. Однією з проблем навчального процесу на сьогоднішній день є відсутність сучасного транслятора. Наразі використовується транслятор TASM, який має ряд обмежень. Для роботи з цим програмним забезпеченням потрібно мати 16- або 32-х розрядну систему. Через це виникає необхідність встановлювати віртуальну машину, щоб користуватись транслятором. Іншим обмеженням є не зовсім зручні синтаксичні правила для написання асемблерного коду. Ще одним важливим фактором є те, що транслятор перестав оновлюватись з 2003 року, тому він не має підтримки і подальшого розвитку. Все це призводить до необхідності створити нове програмне забезпечення, що замінить TASM і спростить виконання студентами лабораторних робіт під різними операційними системами.

Метою дослідження є аналіз алгоритмів трансляції асемблерних програм, пошук варіантів їх адаптації та оптимізації і створення нового транслятора на основі знайдених рішень.

Об'єкт дослідження – існуючі транслятори та алгоритми трансляції асемблерних програм.

Методи дослідження: аналіз, порівняння, вдосконалення, синтез, програмна реалізація, відлагодження, тестування.

Предмет дослідження – способи управління процесом трансляції та методи адаптації алгоритмів транслятора до роботи із зовнішньою таблицею мнемонічних позначень команд та до взаємодії з різними видами операційних систем.

Завдання дослідження

1. Виконати аналіз існуючих програмних засобів для пошуку найбільш вдалих алгоритмічних рішень та визначення недоліків.
2. Дослідити та узагальнити основні методи та алгоритми існуючих трансляторів з метою знаходження їх недоліків та розробки власних алгоритмів.
3. Розробити новий транслятор, який повинен виконувати функції трансляції асемблерних програм у відповідності із встановленими вимогами.
4. Розробити методику проведення випробувань, провести тестування та відлагодження розробленого програмного забезпечення.
5. Створити інструкцію користувача з описом синтаксичних правил та порядку роботи з транслятором.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ ЗАСОБІВ

На сьогоднішній день існує чимало трансляторів. Більшість їх функціональних можливостей схожі між собою, проте кожне ПЗ має свої особливості, які роблять його унікальним. Для пошуку сучасних варіантів, що могли б стати хорошою заміною TASM, було додатково використано можливості штучного інтелекту, що допомогло виконати більш розширений огляд. Серед великого різноманіття трансляторів для розгляду було обрано VASM, GNU AS, FASM, NASM, LLVM with TableGen, які будуть проаналізовані далі.

1.1. Огляд TASM

Перш ніж проводити аналіз існуючих програм-трансляторів, варто ознайомитись з самим TASM. Як було зазначено раніше, дане програмне забезпечення було розроблено ще у 1987 році. Остання його версія (v3.2) була випущена у 2002 році. TASM (Telemark Assembler) – табличний крос-асемблер для середовища MS-DOS [5]. Вихідний код асемблера можна писати на певному діалекті, що зазвичай є дуже близьким до мови асемблера виробника. TASM здатний його зібрати, відтранслювати і видати об’єктний код, який потім можна використовувати для обраної мікропроцесорної системи.

Даний транслятор підтримує достатньо велику кількість сімейств мікропроцесорів: 6502, 6800/6801/68HC11, 6805, 8048, 8051, 8080/8085, Z80 тощо. При цьому область застосування TASM не обмежується лише відомими сімействами. Якщо спеціаліст створив власну мікропроцесорну систему, він з легкістю може використати даний транслятор для трансляції коду, написаного під його розробку. Щоб TASM зміг коректно обробити вхідний код та сформувати об’єктний, необхідно створити власну систему мнемонічних позначень. Докладніше про це буде описано в розділі 2. Таким

чином, це є найголовнішою особливістю, яка робить TASM унікальним і корисним навіть через майже 40 років від його створення.

Функціональні можливості цього транслятора включають в себе потужний аналіз виразів, який здатний розпізнавати 17 різних операторів. TASM підтримує підмножину команд препроцесора C. Сюди можна віднести директиву **include**, що дозволяє включати додаткові файли; **define**, що дозволяє створювати макроси; **if, else, ifdef, ifndef, endif** – директиви, які перевіряють певні умови. Окрім цього, TASM має власні директиви, такі як **org, end, equ** тощо, які дозволяють йому правильно аналізувати різні фрагменти коду та виконувати відповідні дії.

Ще одна можливість, яку надає даний транслятор – створення макропроцедур. Дана конструкція схожа на функції в мовах програмування високого рівня. Іншими словами, створення макропроцедур дозволяє уникати повторного написання коду. Замість цього користувачу достатньо лише раз визначити назву цієї конструкції, за потреби вказати параметри, а потім описати тіло макропроцедури, яке буде виконуватись при кожному її виклику. Такі функціональна можливість є доволі зручною та ергономічною.

TASM може генерувати на виході об’єктний файл 4 різних форматів: Intel hex, MOS Technology hex, Motorola hex, binary. Перші три є лінійно орієнтованими і використовують певні правила для формування вихідного машинного коду. Приставка hex в кінці вказує на те, що кожен файл містить дані у шістнадцятковому форматі. Формат binary генерує звичайну послідовність даних, як вона розміщується в пам’яті.

Цей невеликий огляд показує, що TASM є дуже корисним та зручним програмним забезпеченням, не дивлячись на його вік. Як буде зрозуміло в подальшому, його функціональні можливості доволі обмежені в порівнянні із сучасними трансляторами. Проте цього достатньо не тільки для навчання, але і для невеликих проектів, наприклад програмування самостійно створених мікропроцесорів.

1.2. Транслятор VASM

VASM (Visual assembler) – транслятор, здатний створювати об’єкти, що зв’язуються, в різних форматах, а також абсолютний код. Підтримуються різні процесори, синтаксичні та вихідні модулі. Підтримуються найбільш поширені директиви/псевдо-опкоди (залежно від модуля синтаксису), а також специфічні для процесора розширення. Асемблер підтримує оптимізацію та релаксацію. Наприклад, вибір найкоротшої інструкції гілки або режиму адресації, а також перетворення гілки в абсолютний стрибок, якщо це необхідно. Він також підтримує цільові процесори з більш ніж 8 бітами на байт (адресація слів), але вимагає, щоб хост-система мала 8-бітні байти. Концепція полягає в тому, що на виході формується спеціальний бінарний код для будь-якої комбінації процесорів і синтаксичних модулів. Всі вихідні модулі, які мають сенс для обраного процесора, включені в VASM бінарний файл, і необхідно переконатися, що обрано потрібний формат вихідного файлу. Типовим є формат тестового виводу, корисний лише для зневадження або аналізу виводу [6].

VASM підтримує ті ж оператори, що і мова програмування С, але їх запис може відрізнятись. Наприклад, остача від ділення в С позначається символом «%», а у VASM – «//». Пріоритетність операторів у трансляторі дещо відрізняється, щоб відповідати загальній поведінці асемблера.

VASM надає можливість створювати власні символьні позначки. Є три типи таких позначень: вирази, мітки та імпортовані позначки. Вирази зазвичай не видимі поза кодом, якщо тільки вони не були явно експортовані. Мітки завжди є адресами всередині програми. За замовчуванням вони мають локальну область видимості для компонувальника. Імпортовані позначки визначаються поза програмою і повинні розпізнаватись компонувальником.

VASM транслятор є мультиплатформенним і охоплює доволі велику кількість ОС. Основна увага приділяється системам команд для конкретних платформ, таких як x86, ARM, MIPS та інші. Також даний транслятор

підтримує функціональність макросів, дозволяє створювати структури, може підтримувати умови, але це залежить від синтаксичних модулів. Якщо користувач потребує визначення власних змінних, він може використовувати директиви для цієї задачі. Загалом VASM має всі необхідні для трансляції функціональні можливості.

Основним недоліком даного транслятора є те, що він запрограмований на конкретні системи команд і використовується для конкретних процесорів. Іншими словами, у разі створення власного процесора із розробленою системою команд для нього VASM не зможе відтранслювати асемблерний код, оскільки мнемонічні позначення для нього будуть невідомі. Це призведе до помилок і в результаті не дасть очікуваного результату. В цьому випадку можна змінити вихідний код самого транслятора, додавши туди нові мнемоніки та інформацію про новий процесор. Але для цього необхідно мати достатньо досвіду в програмуванні такого типу програм, а також приділити час для аналізу коду та розуміння принципів роботи. У зв'язку з цим VASM не може в повній мірі замінити TASM.

1.3. Транслятор GNU AS

GNU AS (GNU assembler) – сімейство асемблерів, яке є частиною набору інструментів GNU Binutils. При використанні GNU assembler на одній архітектурі слід знайти схоже середовище при використанні на іншій архітектурі [7]. GNU AS дозволяє використовувати один інструмент для трансляції асемблерного коду на будь-якій підтримуваній архітектурі. Даний транслятор підтримує багато відомих архітектур, таких як x86, ARM, MIPS, RISC-V та інші. GNU AS, як і VASM, може працювати під різними ОС. Відмінністю є те, що цей транслятор більше розрахований на Unix-системи, тому для використання на Windows необхідно встановлювати середовища сумісності, наприклад Cygwin [8] або MinGW [9].

Оскільки GNU Binutils є цілою «екосистемою», то GNU AS добре інтегрується з іншими інструментами, що розширює можливості і дозволяє

виконувати лінковку та компіляцію. При цьому транслятор може працювати і окремо від цього набору інструментів, тоді він лише буде транслювати асемблерні файли і генерувати об'єктний файл на виході. GNU AS містить докладний опис користувача, чого немає у TASM. Як і VASM, даний транслятор підтримує створення макросів, використання директив, які передбачені розробниками, і може генерувати різні формати об'єктних файлів. GNU AS є швидким і ефективним транслятором. Він підходить для великих проектів, коли швидкість трансляції має значення.

GNU AS, на відміну від багатьох інших трансляторів, виконує трансляцію за 1 прохід. Даному транслятору можна передавати як один файл для трансляції, так і декілька підряд. В останньому випадку він прочитає всі файли в порядку їх запису, сформує один загальний вхідний файл і виконає трансляцію, сформувавши об'єктний файл на виході. GNU AS має власний синтаксис, але розглядати його немає сенсу, оскільки це великий обсяг інформації, і в більшості випадків синтаксис трансляторів схожий між собою.

Всупереч всім перевагам, GNU AS, як і VASM, не розрахований на те, щоб виконувати трансляцію для новостворених процесорів із власними системами команд. Усі інструкції та мнемоніки даного транслятора «зашиті» в нього, тому для додавання нових необхідно змінювати вихідний код програми. Додатковим недоліком, про що було зазначено раніше, є необхідність використання додаткового ПЗ для роботи під Windows.

1.4. Транслятор FASM

FASM (Flat assembler) – це швидкий транслятор мови асемблера для процесорів архітектури x86, який робить кілька проходів для оптимізації розміру згенерованого машинного коду. Він самокомпілюється і має версії для різних операційних систем. Всі версії призначені для використання з системного командного рядка і не повинні відрізнятися за поведінкою.

Для всіх версій потрібен 32-розрядний процесор архітектури x86 (щонайменше 80386), хоча вони можуть створювати програми і для 16-

розрядних процесорів архітектури x86. Для DOS-версії потрібна ОС, сумісна з MS DOS 2.0, а також справжнє середовище реального режиму або DPMI. Для версії для Windows потрібна консоль Win32, сумісна з версією 3.1 [10].

FASM є дуже швидким у порівнянні з іншими трансляторами асемблерів. Його оптимізований механізм трансляції дозволяє швидко збирати великий код без втрати продуктивності. FASM підтримує створення макропроцедур, користувацьких змінних, адресних міток, дозволяє створювати числові та строкові константи. Цей транслятор надає користувачам повний контроль над кожною операцією і результатом трансляції. Це дає можливість точного налаштування вихідного коду для досягнення оптимальних результатів, включаючи управління форматом машинного коду.

Говорячи про недоліки, можна виділити ті самі аспекти, що і в попередніх прикладах: обмежена підтримка архітектур та неможливість використання власних систем мнемонічних позначень. Через це область використання FASM значно звужується і обмежується лише тими процесорами, інформація про яких «вшита» у цей транслятор. Повертаючись до минулих програм, можна знову зазначити, що переписувати вихідний код є доволі важкою задачею, тому такий варіант не підходить для навчального процесу, де студентам можуть запропонувати створити власні процесори і системи команд для них.

1.5. Транслятор NASM

NASM (Netwide assembler) – транслятор асемблера для процесорів архітектури x86 та x86-64 [11]. Дана програма є мультиплатформенною, працює під системами Windows, Linux, MacOS та MS-DOS. На відміну від попередньо розглянутих варіантів, NASM має простий та легкий для розуміння синтаксис, що дозволяє значно швидше вивчити, як його використовувати. Окрім цього, можна зазначити, що даний транслятор є

проектом з відкритим кодом, що дозволяє користувачам ознайомлюватись з кодом та за потреби вносити зміни.

NASM підтримує використання макропроцедур. Такий підхід значно полегшує програмування на асемблері та дозволяє користувачам не писати постійно ті самі блоки коду, а створювати свого роду функції. Підтримка такої невеликої кількості архітектур є одночасно перевагою та недоліком NASM. Якщо казати про позитивну стороні, то як було зазначено раніше, це дозволяє значно швидше та легше навчитись використовувати даний транслятор.

Негативною стороною є те саме, що і в розглянутих минулих прикладах – нездатність працювати з новими або іншими існуючими процесорами без переписування вихідного коду програми. Іншим недоліком, знову ж таки, є відсутність можливості використання користувацьких систем мнемонічних позначень.

1.6. LLVM with TableGen

LLVM загалом є цілим фреймворком, призначеним для підтримки різних етапів компіляції: від генерації машинного коду та лінкування до оптимізації. Він підтримує такі мови програмування як C, C++, Objective C, Ada та Fortran [12]. LLVM здатний працювати з великою кількість різних архітектур, включаючи x86, x86-64, PowerPC та інші. Даний програмний продукт є потужним набором інструментів з великою кількістю переваг, проте все ще відсутня можливість працювати із власними системами команд.

На допомогу приходить TableGen – спеціалізована мова для опису інструкцій процесорів, регістрів та інших апаратних характеристик. Цей інструмент дозволяє описувати власні інструкції свого процесора або системи команд. З його допомогою також можна описувати регістри, генерувати необхідні компоненти для фронтенду та бекенду і багато іншого. TableGen має досить докладний опис користувача, де описано синтаксис, можливості, типи даних, які підтримуються, правила робота з ним тощо [13].

LLVM сумісно з TableGen працює наступним чином:

- Спочатку у файлах TableGen описуються всі необхідні інструкції, регістри та інші атрибути апаратної платформи;
- Далі запускається TableGen для створення коду, який потім використовує LLVM для підтримки обраної архітектури;
- Згенеровані TableGen файли інтегруються в компілятор LLVM. Це дозволяє компілятору підтримувати нову архітектуру й обробляти нові інструкції та оптимізувати їх для цільової платформи.

LLVM with TableGen є досить потужним інструментом, який може конкурувати з TASM. Проте, як було показано вище, для отримання необхідних для навчального процесу функціональних можливостей необхідно встановлювати додаткове програмне забезпечення. Більше того, враховуючи, що LLVM – це фреймворк, потрібно ще мати додатковий компілятор (наприклад, Clang). З одного боку, ця програма навіть надає можливість писати програми мовами програмування високого рівня, такими як C та C++, і у підсумку отримувати машинний код. З іншого боку, необхідність додаткового ПЗ, яке може створювати складнощі при роботі під різними ОС. Тому все ж таки в плані ергономічності TASM є кращим, оскільки все передбачено в одному пакеті. З цього можна зробити висновок, що LLVM with TableGen не є підходящим варіантом.

Висновки за розділом 1

Підсумовуючи все, що було сказано в 1 розділі, можна зробити висновок, що кожен із розглянутих трансляторів володіє своїми позитивними сторонами, має широкі функціональні можливості, що дозволяє йому виконувати більше задач, і працює під різними ОС. Якщо ж повернутись до TASM, то даний транслятор працює лише під MS-DOS, через що виникає необхідність налаштовувати додаткове ПЗ. Проте, можливості роботи цього транслятора із довільними системами команд роблять його незамінним навіть

після майже 40 років від його створення. Через це, в рамках навчального процесу, неможливо відмовитись від використання TASM і відсутня можливість замінити його програмним забезпеченням, яке буде доступне кожному студенту. Тому конче необхідно створити новий транслятор, який буде наслідувати можливості TASM і матиме доповнення у вигляді (як мінімум) мультиплатформеності. Окрім цього, потрібно створити доступний опис правил по використанню цього ПЗ, а також по створенню власних систем команд. Незважаючи на те, що користувач матиме можливість самостійно визначати імена команд, аргументів тощо, він повинен дотримуватись певних синтаксичних норм, щоб транслятор міг правильно опрацьовувати отримані дані.

РОЗДІЛ 2
РОЗРОБКА НОВОГО ТРАНСЛЯТОРА

Перш ніж розпочати розробку нового ПЗ, необхідно дослідити, як працює транслятор, обрати інструментарій та розробити алгоритм. За основу було вирішено взяти транслятор TASM, оскільки він має всі необхідні для вивчення основ програмування функціональні можливості.

2.1. Аналіз алгоритму трансляції

Транслятор працює наступним чином: приймає вхідний код, аналізує його та формує на виході об’єктний код. На рисунку 2.1 наведено приклад файлу, який передається для трансляції, а на рисунку 2.2 показано результат успішної трансляції правильно написаного коду:

```
        .ORG 000h           ; Призначення адреси трансляції
; -------------------------------------------------------------------------------------
        #Include Macros.asm ; Під'єднання бібліотеки макропроцедур
; -------------------------------------------------------------------------------------

        LD   D, #003h       ; Готуємо регістр D для зчитування кнопок M(D.0) і N(D.1)

; =========================  КОНТРОЛЬ МОЖЛИВИХ СТАНІВ ПРИСТРОЮ  =========================

ST00:
        Stat00()            ; Контроль стану N=0, M=0
        GOTO P1_01          ; Якщо натиснуто <M> -> додаємо C=C+1 та переходимо в стан 01
        GOTO P2_10          ; Якщо натиснуто <N>   -> На додавання 2
        GOTO P3_11          ; Якщо натиснуті <N+M> -> На додавання 3

ST01:   Stat01()            ; Контроль стану N=0, M=1
        GOTO ST00           ; Якщо відпущено <M>, переходимо до стану 00
        GOTO P2_11          ; Якщо натиснуто <N>, додаємо C=C+2 та переходимо до стану 11
        GOTO P2_10          ; Якщо змінились <N+M>, додаємо C=C+2 і переходимо до стану 10

ST10:   Stat10()            ; Контроль стану N=1, M=0
        GOTO P1_11          ; Якщо натиснуто <M>, додаємо C=C+1 та переходимо до стану 11
        GOTO ST00           ; Якщо відпущено <N>, переходимо до стану 00
        GOTO P1_01          ; Якщо змінились <N+M>, додаємо C=C+1 і переходимо до стану 01

ST11:   Stat11()            ; Контроль стану N=1, M=1
        GOTO ST10           ; Якщо відпущено <M>, переходимо до стану 10
        GOTO ST01           ; Якщо натиснуто <N>, переходимо до стану 01
        GOTO ST00           ; Якщо змінились <N+M>, переходимо до стану 00

; ==============================  ЗМІНА СТАНІВ ПРИСТРОЮ  ================================

P1_01:  AddC(1)             ; Додавання C=C+1
        GOTO ST01           ; Перехід до першого стану
P2_10:  AddC(2)             ; Додавання C=C+2
        GOTO ST10           ; Перехід до другого стану
P1_11:  AddC(1)             ; Додавання C=C+1
        GOTO ST11           ; Перехід до третього стану
P2_11:  AddC(2)             ; Додавання C=C+2
        GOTO ST11           ; Перехід до третього стану
P3_11:  AddC(3)             ; Додавання C=C+3
        GOTO ST11           ; Перехід до третього стану
;----------------------------
        .END
;----------------------------
```

Рисунок 2.1 – Вхідний файл для трансляції

```
:180000043030D40022CC00F40012CC118801640012CC0028062806883
:18001800807A0D40022CC02740012CC030802E40012CC11A80028074AB
:1800300080680D40022CC13F40012CC148804640012CC032806E80024A
:1800480080620D40022CC15740012CC060805E40012CC14A8032801AFC
:180060008002410108 3E801A4102083E8032410108 3E804A4102083ECE
:08007800804A4103083E804A62
:00000001FF
```

Рисунок 2.2 – Результат трансляції

Для того, щоб створити транслятор, який правильно буде обробляти вхідні файли, необхідно розуміти, як взагалі виконується перехід від асемблерного коду до машинного. Як видно з рисунку 2.1, вхідний файл містить різні синтаксичні конструкції. Транслятору необхідно розрізняти кожен елемент і правильно його обробляти. На прикладі TASM проведено огляд основних синтаксичних конструкцій:

- Директива. Директиви є зарезервованими словами, які закладені в коді транслятора на стадії розробки. Їх не можна використовувати як ім’я команд або аргументи. Ці конструкції є регістронезалежними. Це означає, що include та INCLUDE будуть розпізнані однаково, а не як різні ключові слова.

```
#Include Macros.asm
```

Рисунок 2.3 – Запис директиви, що буде розпізнаний транслятором TASM

- Ім’я команди. Імена команд створюються користувачем в окремому файлі, який містить список усіх інструкцій, які програміст може використовувати. Структура даного файлу буде розглянута дещо пізніше. Імена команд є регістронезалежними, як і директиви, але лише при написанні асемблерного коду для трансляції.
- Параметри. Параметри можуть бути як у команд, так і у директив. Якщо це назви регістрів, вони повинні співпадати з аргументами у файлі команд. Якщо це параметри у вигляді змінних,

необхідно надавати певні значення цим змінним і оголошувати їх до використання. Окрім того, параметр повинен відповідати шаблону аргументу. Наприклад, якщо в якості параметра передане число, то аргумент повинен вказувати на те, що необхідно використовувати не текстові значення.

```
LD    D,  003h
```

Рисунок 2.4 – Формат запису команди з параметрами

- Мітка. Мітка є спеціальною синтаксичною конструкцією, яка може спочатку і не мати значення під час використання. Оскільки вона зберігає лише адресу команди, що слідує після мітки, то можливий варіант, коли адреса ще не відома. В цьому випадку можна зробити ще один прохід після того, як всі адреси будуть відомі, і передивитись, чи немає міток, які не отримали свого адресного значення. Мітки є регістрозалежними, тобто label та LABEL будуть різними синтаксичними конструкціями.

```
ST00:
```

Рисунок 2.5 – Формат запису мітки, що буде розпізнана транслятором TASM

- Макропроцедура. Макропроцедурою називають іменований блок, який може включати в себе декілька команд. Це є аналог функцій у мовах програмування високого рівня, що дозволяє уникати повторного написання коду. Макропроцедура має своє ім’я, може приймати аргументи, а також має тіло, яке складається з 1 або більше команд. Кількість інструкцій, які можна помістити всередину даної конструкції є обмеженою, аби не перевантажувати її. Якщо необхідно створити велику макропроцедуру, краще розбити її на кілька невеликих і спростити обробку. На рисунку 2.6 представлено спосіб запису макропроцедури для TASM.

```
#DEFINE   Stat01()            MOV    B,   D
#DEFCONT \                    LD     A,   #00000010b
#DEFCONT \                    LD     A,   AND
#DEFCONT \                    JZ     *+9
#DEFCONT \                    LD     A,   #00000001b
#DEFCONT \                    LD     A,   AND
#DEFCONT \                    JZ     *+13
#DEFCONT \                    GOTO   *+9
#DEFCONT \                    LD     A,   #00000001b
#DEFCONT \                    LD     A,   AND
#DEFCONT \                    JNZ    *-16
```

Рисунок 2.6 – Коректний запис макропроцедури, який буде розпізнаний TASM.

- Коментар. Коментар є синтаксичною конструкцією, яка відкидається в процесі трансляції. Зазвичай це певні підказки, які програміст може писати як для себе, так і для тих, хто буде працювати з його кодом. Як тільки транслятор зустрічає позначення коментаря, він ігнорує все, що написано далі. Коментарі можуть бути як однорядкові, так і багаторядкові.

```
; ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
; ||||||||||||||||||||  ПРОЦЕДУРА ДОДАВАННЯ ЧИСЛОВИХ ЗНАЧЕНЬ ДО РЕГІСТРА С  ||||||||||||||||||||
; ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

;    Процедура дозволяє за допомогою клавіатури,  під'єднаної до порту D моделі процесора,
; додавати різні числові значення до регістра  С  при натисканні кнопок,  що під'єднані до
; розрядів D.0 та D.1. Кнопки можуть натискатися та відпускатися в будь-який момент часу у
; довільному порядку.
; ________________________________________________________________________________________
```

Рисунок 2.7 – Позначення коментарів, які розпізнаються TASM.

- Користувацька змінна. Користувацька змінна є елементом, який створюється користувачем. Вона не повинна бути зарезервованим словом, ім’ям команди або ім’ям аргументу (наприклад, якщо є аргумент – регістр А, то не можна створити змінну А). Користувацькій змінній можна присвоїти значення іншої змінної, якщо воно існує.

Це були представлені основні синтаксичні конструкції асемблера, які повинні правильно розпізнаватись трансляторами. Оскільки для майбутньої

розробки нового програмного забезпечення було взято за основу TASM, то деякі правила запису тих чи інших елементів будуть схожими.

Кожна команда, створена користувачем, повинна мати код операції. Зазвичай це число у шістнадцятковому форматі. Коли транслятор зустрічає інструкцію, він шукає її у файлі із системою команд. Для цього вона повинна відповідати 2 вимогам: мати те саме ім’я та її список параметрів повинен задовольняти списку аргументів. Якщо обидві умови виконуються, тоді транслятор отримує код операції цієї команди та записує його до масиву доступної пам’яті. Наприклад, на рисунку 2.4 було представлено інструкцію LD, яка приймає 2 параметри: регістр, куди потрібно записати дані, та число, яке необхідно записати в цей регістр. Транслятор, зустрівши цю команду, звернеться до системи команд, яка була завантажена на початку, виконає пошук і знайде запис, який має аналогічну мнемоніку і його аргументи співпадають з переданими параметрами. Код операції дорівнює 43h. Це значення буде записано в масив за адресою, на яку вказує транслятор у поточний момент часу. Оскільки дана команда є двобайтною, після коду операції в масив буде записано число 3h, яке було передане другим параметром.

Окремо варто розглянути директиви асемблера, оскільки це є команди, закладені розробником в сам транслятор, які користувач не може змінювати. Загалом TASM може розпізнавати велику кількість директив, але в рамках аналізу для розробки майбутнього транслятора було обрано ті, які найчастіше використовуються студентами під час виконання лабораторних та практичних робіт. Нижче наведене перелік та невеликий опис кожної директиви:

- Include – директива, що дозволяє підключати додаткові файли. Це можуть бути бібліотеки або інші файли з кодом (наприклад, файл для макропроцедур). В якості параметра приймає ім’я файлу, який необхідно включити.

- DEFINE – директива, яка вказує на те, що далі буде створено макропроцедуру. Після вказання цього ключового слова необхідно записати ім’я майбутньої макропроцедури, а потім вказати список аргументів у круглих дужках (або залишити порожніми, якщо аргументи відсутні). Різні транслятори мають різний синтаксис, тому десь можна зустріти інший спосіб ініціалізації макропроцедури, але, як було зазначено раніше, основою майбутнього транслятора стане саме TASM. Після списку аргументів даний транслятор дозволяє записати першу команду тіла цієї конструкції;
- DEFCONT – директива, яка вказує на продовження тіла макропроцедури, визначеної перед цим директивою DEFINE. В якості параметра їй передається звичайний слеш «\». Потім, встановивши пробіл або табуляцію після параметра, можна вказувати інструкцію;
- ORG – директива, що вказує на адресу, з якої необхідно починати запис байтів програми. В якості параметра приймає одне число. Якщо не викликати цю директиву явно на початку програми, тоді початок буде встановлено неявно на адресу 0;
- EQU – директива, що дозволяє створювати користувацькі змінні. Приймає 2 параметри: ім’я змінної та значення;
- NOLIST – директива, що вказує транслятору не додавати весь наступний текст до файлу-лістингу. Цей файл містить докладний звіт трансляції з адресами, помилками (якщо вони є), кодами операцій, які були отримані після трансляції, а також загальний лістинг програми, разом з усіма включеними файлами;
- LIST – директива, яка вказує транслятору додавати весь наступний текст до файлу-лістингу. Зазвичай ці дві директиви використовують в парі: непотрібну частину коду приховують за допомогою NOLIST, а все інше відображають через LIST;

- END – директива, яка вказує на місце завершення трансляції. Все, що написано після цієї директиви, ігнорується транслятором.
- DB – директива, яка дозволяє занести байт до поточної адреси.
- DW – директива, аналогічна до DB, але замість байту дозволяє занести двобайтове слово до поточної адреси.

Цієї невеликої кількості розглянутих директив цілком достатньо для виконання завдань, які передбачені в лабораторних роботах.

2.2. Аналіз файлу із системою команд

На рисунку 2.8 наведено фрагмент файлу із системою команд:

```
NOP   ""      00 1 NOP 1
MOV   B,A     01 1 NOP 1
MOV   C,A     02 1 NOP 1
MOV   D,A     03 1 NOP 1
MOV   A,B     04 1 NOP 1
MOV   C,B     06 1 NOP 1
MOV   D,B     07 1 NOP 1
MOV   A,C     08 1 NOP 1
MOV   B,C     09 1 NOP 1
MOV   D,C     0B 1 NOP 1
MOV   A,D     0C 1 NOP 1
MOV   B,D     0D 1 NOP 1
MOV   C,D     0E 1 NOP 1
```

Рисунок 2.8 – Файл із системою команд

При створенні власної системи команд програміст повинен дотримуватись структури файлу, щоб транслятор міг правильно його прочитати. Як видно з рисунку 2.8, для TASM файл повинен містити 6 стовпців. Далі наведено опис кожного компоненту:

- Перший стовпчик – ім’я команди. Зазвичай прийнято, що це повинна бути скорочена назва, яка дозволяє зрозуміти, що робить дана інструкція. Ім’я команди не повинно бути зарезервованим словом.

Окрім цього, TASM підтримує лише символи латинського алфавіту, цифри, якщо вони не на початку назви, і деякі спеціальні символи;

- Другий стовпчик – список аргументів. Аргументи бувають різних типів, і зазвичай розробниками процесорів задається певний синтаксис для кожного типу. Тому транслятор повинен їх розрізняти відповідно до тих позначень, які були передбачені на стадії розробки. Далі невеликий огляд цих аргументів:
  - Іменований аргумент. Це позначення різних регістрів. На рисунку 2.8 якраз наведено приклад іменованих аргументів.
  - Адреса. Даний аргумент позначається символом «*» і вказує на те, що передане значення буде сприйматись як адреса. В якості параметра можна передавати або мітку, або числове значення. Також можна передавати простий математичний вираз. Всі інші синтаксичні конструкції будуть розглядатись як помилка.
  - Числовий аргумент. В якості параметра можна передавати число або змінну, яка містить числове значення. Якщо явно не вказано систему числення, параметр сприймається в десятковій системі. Позначається числовий аргумент комбінацією символів «#*».
  - Порожній аргумент. Якщо інструкція не містить ніяких аргументів (команда NOP на рисунку 2.8), в цьому випадку програміст повинен вказати порожні подвійні лапки. Якщо просто залишити порожнє місце, транслятор видасть помилку.

Даний список є неповним, оскільки будь-який процесор може мати інші варіанти аргументів, як наприклад аргумент непрямої адресації, або інвертоване значення аргументу. Проте поки що для розробки майбутнього транслятора розглянутих варіантів буде достатньо.

- Третій стовпчик – код операції. Це число у шістнадцятковому форматі, яке потрібне для того, щоб процесор сприймав її як

інструкцію для виконання. Код операції є основним елементом, оскільки імена команд та аргументи потрібні для програміста, аби правильно написати код програми, а також для транслятора, аби коректно обробити ці команди, тоді як ЕОМ виконувати лише машинний код.

- Четвертий стовпчик – кількість байт. Команда може бути однобайтною, двобайтною та навіть трьохбайтною. Кількість байт вказує транслятору на те, скільки адресного простору повинна займати та чи інша команда.
- П'ятий та шостий стовпчики були внесені автором транслятора Томасом Андерсеном для того, щоб правильно розрізняти типи команд та процесори, і в залежності від цього виконувати різні розрахунки. Це було потрібно, оскільки TASM був створений для роботи з 20 різними процесорами. Різні команди в різних процесорах можуть під час трансляції аналізуватися та перетворюватися на машинний код за різними алгоритмами. Хорошим прикладом є команди, які виконують перехід на певну адресу. Адресація може бути відносною або абсолютною. Кожен тип адресації використовує власний алгоритм обробки, тому для коректної обробки транслятору необхідно надати додаткові вказівки. Ця можливість налаштування транслятора також збережена у новій версії.

Аналіз файлу із системою команд надає можливість зрозуміти, які синтаксичні конструкції повинні бути розпізнані транслятором, як правильно визначати коректність параметрів та аргументів і як синтаксично коректна команда повинна бути представлена на машинній мові. В цьому розділі опущено огляд синтаксичних правил. Вони будуть представлені у розділі 3 для розробленого нового транслятора. Це допоможе студентам не допускати

помилок при створенні власних систем команд та написанні асемблерного коду.

2.3. Створення алгоритму нового транслятора

Після проведення аналізу транслятора TASM перед початком розробки необхідно створити алгоритм, який допоможе визначитись з тим, які інструменти обрати, яким чином реалізувати необхідні функціональні можливості, що можна модифікувати та покращити. На рисунку 2.9 показано UML діаграму активності [14], яка відображає основний алгоритм роботи майбутнього транслятора.

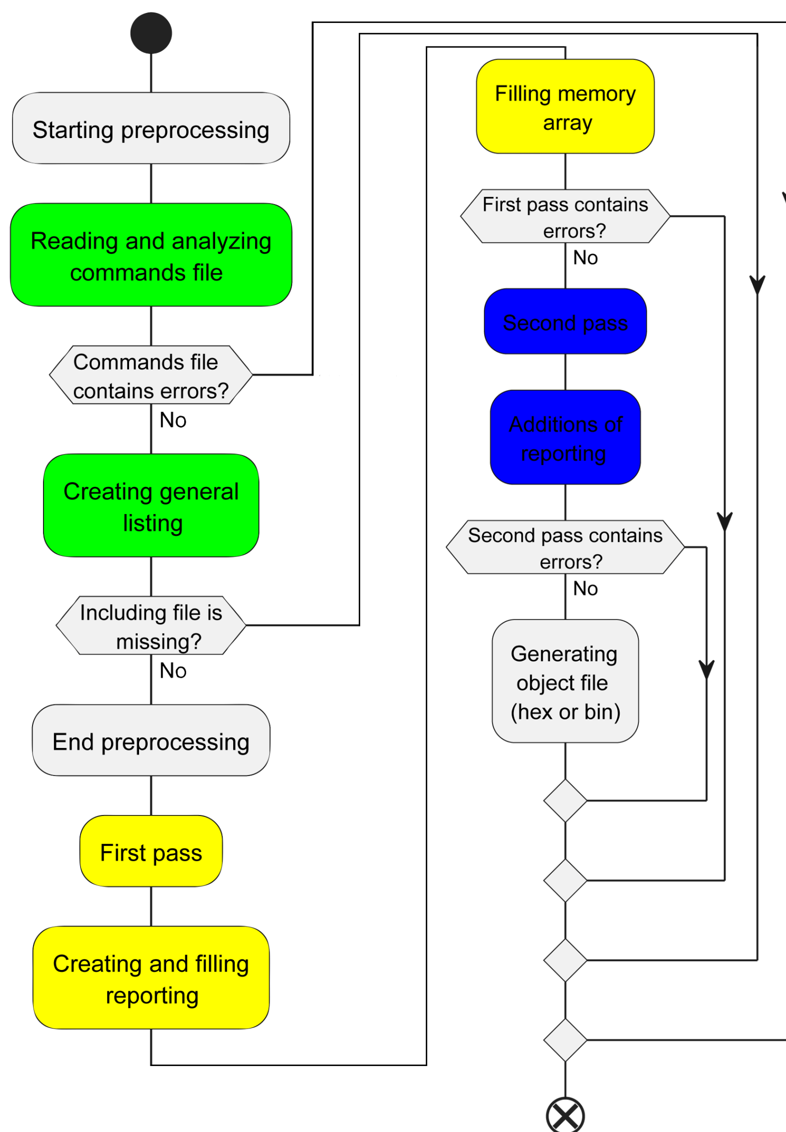

Рисунок 2.9 – UML діаграма активності

Робота транслятора буде починатися з препроцесінгу. Це частина, де відбувається підготовка до самого процесу трансляції. Вона включає в себе 2 етапи: аналіз переданого файлу із системою команд та створення загального лістингу програми. Далі представлено більш докладний опис кожного етапу.

Аналіз файлу із системою команд включає в себе повний перегляд цього файлу та перевірку кожного рядка на відповідність синтаксичним правилам. Як було описано в розділі 2.2, файл повинен мати певну структуру. Новий транслятор буде підтримувати таку саму структуру файлу із системою команд, оскільки вона є достатньо зручною та зрозумілою. Під час аналізу кожен рядок повинен перевірятись на наявність усіх елементів. Далі повинна бути перевірка валідності кожного компонента. Якщо всі команди пройшли перевірку, тоді необхідно сформувати масив, аби не звертатись щоразу до файлу. В іншому випадку треба сформувати звіт про знайдені помилки і завершити на цьому процес трансляції. Загалом, на кожному етапі якщо знайдено помилки, трансляція буде завершуватись, а користувач отримуватиме вказівки про те, де переглянути інформацію про них.

Створення загального лістингу програми передбачатиме пошук директив include, які вказують на включення додаткових файлів. В цьому випадку виникає необхідність перевіряти, чи файл, який вказаний як параметр до директиви, знаходиться в тій же директорії. Якщо так, тоді цей файл буде повністю прочитаний та доданий до лістингу звітності після виклику директиви. В іншому випадку користувачу треба повідомити, що файлу не існує, і як результат, трансляція на цьому завершиться. Немає необхідності продовжувати цей процес, оскільки основний код програми може містити синтаксичні конструкції, які булі ініціалізовані в іншому файлі. А оскільки файл відсутній, транслятор визначатиме ці компоненти як невідомі, генеруючи нові помилки. Окрім пошуку директив include на етапі препроцесінгу ще потрібно буде знаходити ініціалізацію макропроцедур,

якщо вони є. Для зручності процесу трансляції простіше буде зберігати тіло кожної макропроцедури. Потім, коли транслятор зустрічатиме її виклик, замість нього буде вставлятися блок коду, що належить цій конструкції. Це позбавить транслятор необхідності шукати оголошення та тіло макропроцедури. Також такий підхід може спростити визначення невідомих синтаксичних конструкцій.

Після завершення препроцесінгу транслятор повинен почати перший прохід. На цьому етапі він повинен читати загальний лістинг програми, сформований препроцесором, рядок за рядком і проводити аналіз кожного елементу. Докладний опис аналізу буде представлено в розділі 2.4. Як видно з рисунків 2.9 та 2.10 під час першого проходу транслятора буде формуватися загальний звіт процесу трансляції. Окрім цього, повинен формуватися масив, який є прообразом пам’яті процесора. Він буде містити коди операцій кожної інструкції, що зустрінеться у вхідному файлі. Якщо під час першого проходу транслятор не знайде помилок, він видасть повідомлення користувачу про те, що перший прохід успішно завершено, і перейде до виконання другого проходу. В іншому випадку програміст отримає інформацію про знайдені помилки, а трансляція завершиться.

Другий прохід матиме задачу ще раз переглянути весь лістинг програми у пошуках міток, які не отримали адресних значень під час першого проходу. Як було описано у розділі 2.1, мітка є особливим компонентом, який може не одразу отримати числове значення. Після першого проходу адресний простір вже буде частково або повністю заповнений кодами операцій. Це означає, що всі мітки отримають свої значення адрес, і під час другого проходу вони будуть використанні там, де це необхідно. Наприклад, програміст виконує команду переходу, і в якості параметра вказує мітку, яка ще не зустрічалась транслятором. Тоді або ця вона вказана далі, або можливо її взагалі забули додати. В першому випадку після завершення першого проходу мітка вже буде вказувати на адресу

команди, що слідує за нею. Під час другого проходу, коли транслятор знову зустріне команду переходу, він зможе замінити мітку її значенням, і додати це значення до пам'яті у відповідне місце. Якщо на цьому етапі не було знайдено помилок, тоді транслятор повідомить користувачу, що другий прохід успішно завершений і перейде до фінальної стадії. В іншому випадку аналогічно до першого проходу.

Перед завершенням роботи транслятор повинен сформувати об'єктний вихідний код, оскільки це основний результат його роботи. Формат цього коду може бути як Intel HEX, так і binary. Від чого це залежить буде описано у розділі 2.4. На даному етапі транслятор буде працювати з масивом пам'яті, виконуючи перенос даних до об'єктного файлу. Після цього робота транслятора вважається завершеною. Також користувач отримає повідомлення про кількість помилок, які були знайдені. Очевидно, що при успішній трансляції це число буде 0.

Розроблений алгоритм допомагає зрозуміти, яку структуру повинен мати транслятор, яким чином він має працювати, які особливості трансляції необхідно передбачити та як з асемблерного коду отримати машинний.

2.4. Розробка нового транслятора

Для розробки такого програмного забезпечення як транслятор на думку розробника чудово підходить мова програмування C++ [15]. Вона є швидшою та ефективнішою за інші мови програмування, наприклад Python або Java. Окрім цього, розробник має досвід роботи з C++, оскільки розробка програмної моделі процесора виконувалась також з її використанням. Стандарт C++17 [16] надає достатньо потужні функціональні можливості, які можуть стати у нагоді під час аналізу асемблерного коду, тому було обрано саме його. Для компіляції програми спершу було вирішено використовувати MinGW, оскільки це є базовий компілятор для C++ під ОС Windows. Але згодом його було замінено на CygWin. Причини будуть описані пізніше.

Згідно з розробленим алгоритмом, програмування нового програмного забезпечення почалось із реалізації препроцесінга. Перший етап – аналіз файлу системи команд. Для цього було створено функцію, що виконує цю задачу. Вона приймає в якості параметра файл із системою команд, а також змінну, яка буде зберігати максимальний об'єм пам'яті, який можна задіяти під час трансляції. Це значення можна отримати з того ж файлу із мнемонічними позначками. Для цього користувач при створенні таблиці команд повинен вказати змінну MAX_SIZE = *максимальний розмір пам'яті*. Це повинно бути число в десятковій системі числення. В навчальному процесі використовуються процесори із досить обмеженим об'ємом пам'яті, тому потрібен контроль, щоб не допустити створення програм, які не зможуть потім бути завантажені в пам'ять процесора. Система команд, розроблена Томасом Андерсеном, не передбачала такого параметру, це було доповнення від розробника транслятора. У випадку, якщо користувач не вказав цей параметр, транслятор не несе відповідальності за можливе переповнення доступного об'єму пам'яті. На цьому етапі вже було використано нові можливості стандарту C++17, а саме бібліотека optional. Вона надає можливість створити порожнє значення. Якщо параметр MAX_SIZE було розпізнано у файлі команд, тоді змінна отримає значення, яке там міститься. В іншому випадку, вона залишиться порожньою.

```
struct instruction {
    char name[11];
    vector<string> args;
    bool isUnknownSeq;
    short opcode;
    short bytes;
    char mod[11];
    short _class;
};
```

Рисунок 2.10 – Склад структури для зберігання компонент системи команд під час трансляції

Для аналізу файлу команд кожен рядок необхідно було розбити на складові. Всі компоненти необхідно десь зберігати. Було прийнято рішення використати можливості мови програмування С, які також доступні у С++, та створити структуру [17]. На рисунку 2.10 показано її склад. Вона вміщує в себе сім компонентів, шість з яких пов'язані з таблицею, а один додатковий для визначення наявності зайвих символів у рядку.

Згідно із синтаксичним правилом, кожен елемент повинен відділятись від інших хоча б одним пробілом або символом табуляції. Це стало підказкою для розробника, як правильно розбити рядок на компоненти. Аби не вдаватись в дрібні подробиці, на виході після розбиття зберігається об'єкт, який містить кожен елемент рядка з таблиці. Якщо після розбиття залишились додаткові символи, встановлюється флаг (один з компонентів структури), який потім вказує на те, що у файлі містяться помилки. Далі починається аналіз кожного компонента відповідно до синтаксичних правил. Спершу перевіряється ім'я команди. Воно не повинно починатися з цифр або спеціальних символів, окрім «_». Ім'я команди може містити в собі літери не тільки латиниці, але й кирилиці, цифри та деякі спеціальні символи (будуть описані в розділі 3). Воно не повинно бути зарезервованим словом, оскільки це призведе до помилки. Ім'я команди при написанні асемблерного коду є регістронезалежним, тому команда MOV та mov будуть розпізнані транслятором однаково. Наступним елементом перевіряється список аргументів. Якщо аргумент є символьним, він не повинен бути ім'ям команди або зарезервованим словом. Інший варіант, коли даний компонент є спеціальним символом або їх комбінацією. В цьому випадку потрібно строго дотримуватись синтаксичних правил, інакше транслятор визначить некоректний запис як помилку. Якщо аргумент містить символи, які непередбачені синтаксисом, це є помилка. Для перевірки наявності невалідних символів було використано функцію **find** та різні її модифікації.

Дана функція повертає позицію, коли символ було знайдено, або –1 в іншому випадку. Окремо виконується перевірка аргументів, які можуть бути адресою, числом, вказівником, порожнім аргументом. Далі аналізується код операції. Він повинен бути числом у шістнадцятковому форматі. Для зручності було створено функцію, яка повертає булеве значення **true** або **false**. Дана функція представлена на рисунку 2.11.

```
bool isHexString(const string &str)
{
    auto it = str.begin();
    while(it != str.end() && isxdigit(*it)) ++it;
    return !str.empty() && it == str.end();
}
```

Рисунок 2.11 – Функція, що перевіряє, чи рядок є шістнадцятковим числом

Після цього також перевіряється, щоб кількість байтів була десятковим числом. Для цього була створена функція, схожа на функцію **isHexString**, яка перевіряє, що всі символи в підрядку є десятковими цифрами. Щодо полів структури **mod** та **_class**, то оскільки вони не мають поки що призначення, їх перевірка не виконується. При подальшій розробці ці поля можуть містити іншу інформацію. Після перевірки всіх компонентів якщо не було знайдено помилок, екземпляр структури додається до масиву таких структур, який буде містити весь список команд. Коли функція завершує свою роботу, виконується аналіз, чи були наявні помилки. Якщо так, в цьому випадку масив команд просто очищується, оскільки в подальшому він вже не потрібен – транслятор завершить роботу.

Наступна розроблена функція стосується формування загального лістингу програми. Як було описано раніше, на цьому етапі виконується пошук директиви **include**. Для цього транслятор починає читати вхідний файл рядок за рядком. Тут вже додано аналіз кожного рядка та розбиття його на складові. Загалом він може мати наступні елементи: мітка, інструкція з параметрами, коментар. Коментар ігнорується транслятором, тому якщо він

зустрічається, ця частина відсікається. Якщо було знайдено мітку (за синтаксичним правилом, вона повинна завершуватись двокрапкою), то вона також відсікається. Після цього весь рядок, що завершився, аналізується на наявність в ньому директиви **include**. Якщо вона була знайдена, і файл, що переданий в якості параметра, знаходиться в тій же директорії, що і транслятор, тоді цей файл відкривається, і на цьому етапі функція рекурсивно викликається для аналізу вже вкладеного файлу. При цьому кожен рядок нового файлу вставляється в загальний лістинг програми після виклику директиви **include**. Щоб програмісту було простіше зрозуміти, що даний файл є вкладеним, у звіті номери рядків цього файлу мають символ «+» в кінці. Якщо в цьому файлі буде ще один успішний **include**, тоді рядки будуть мати два символи «+» на кінці, і так далі. Окрім пошуку вкладених файлів ця функція також шукає оголошення макропроцедур. Розробником було прийнято рішення залишити використання директиви **DEFINE**, проте тіло цієї конструкції має дещо інший синтаксис. У TASM потрібно було використовувати директиву **DEFCONT** на кожному рядку, поки тіло макропроцедури не завершиться. Проте було вирішено відкинути такий формат, замінивши його більш знайомим синтаксисом, який використовується багатьма мовами програмування. Мова йде про заключення тіла макропроцедури у фігурні дужки. Коли транслятор зустрічає директиву **DEFINE**, він спочатку перевіряє, чи оголошення макропроцедури відповідає синтаксичним правилам. Потім аналізує, чи слідує після оголошення тіло функції. Якщо так, то вся макропроцедура зберігається у тимчасовому масиві. Це потрібно для того, щоб коли при подальшому аналізі файлу транслятор зустріне виклик макропроцедури, він мав можливість замінити його її тілом. На рисунках 2.12 та 2.13 показано, як це працює. Такий підхід спрощує подальший аналіз загального лістингу програми, оскільки транслятору потрібно буде лише перевіряти валідність команд, не виконуючи додаткових пошуків якогось блоку коду.

```
        LD    D, #003h          ; Готуємо регістр D для зчитування кнопок M(D.0) і N(D.1)
; ========================  КОНТРОЛЬ МОЖЛИВИХ СТАНІВ ПРИСТРОЮ  ==========================
ST00:
        Stat00()                ; Контроль стану N=0, M=0
```

Рисунок 2.12 – Виклик макропроцедури

```
        LD    D, #003h          ; Готуємо регістр D для зчитування кнопок M(D.0) і N(D.1)
; ========================  КОНТРОЛЬ МОЖЛИВИХ СТАНІВ ПРИСТРОЮ  ==========================
ST00:
        MOV   B, D
        LD    A, #00000010b
        LD    A, AND
        JZ    *+9
        LD    A, #00000001b
        LD    A, AND
        JNZ   *+13
        GOTO  *+9
        LD    A, #00000001b
        LD    A, AND
        JZ    *-16
```

Рисунок 2.13 – Заміна виклику макропроцедури її тілом

Після перевірки списку команд та формування загального лістингу, якщо помилок не було знайдено, транслятор переходить до наступного етапу.

Основна, найбільш об’ємна та складна функція – це функція першого проходу **firstPass**. В якості параметрів вона приймає словник ієрархії файлів та змінну адреси. Ієрархія файлів потрібна для того, щоб у загальному лістингу правильно обробляти рядки вкладених файлів, і у випадку знаходження помилок, повідомити програміста, в якому файлі ця помилка виникла. Такий підхід спростить пошук негараздів. Приклад помилки наведено на рисунку 2.14:

```
Translation failed: file Count-2.asm contains errors. Check REPORT.LST for details.
```

Рисунок 2.14 – Повідомлення про помилки під час трансляції.

Починається функція з того, що весь лістинг програми перебирається рядок за рядком. Процес аналізу рядка частково схожий на той, що відбувався під час формування загального лістингу. Транслятор спершу знаходить та

відсікає коментар, потім шукає мітки. Відмінністю є те, що знайдена мітка не тільки відсікається від рядку, але ще й заноситься до тимчасового масиву для подальшої з нею роботи. Можна відмітити, що масив міток та користувацьких змінних також використовує можливості стандарту C++17. З додаванням бібліотеки **variant** з’явилась можливість в одному масиві зберігати дані різних типів. Така необхідність виникла через те, що користувач може створювати змінні, яким присвоювати символьні значення або інші змінні, що були ініціалізовані раніше. В цьому випадку, треба зберігати або ціле число, або символьне значення. Тому **variant** є підходящим варіантом.

Після того, як рядок було оброблено та видалено зайві елементи, транслятор починає аналізувати інструкції. Це може бути або команда з файлу із системою команд, або директива. Директиви препроцесінгу, такі як **include** та **define**, ігноруються, оскільки вони вже були оброблені раніше. Інші перевіряються відповідно до синтаксичних правил, встановлених розробником транслятора. Директиви не мають власних кодів операцій, тому у випадку успішної перевірки до пам’яті нічого не записується (якщо це не директива **db** або **dw**, які власне і записують числа до пам’яті). Проте вони дозволяють управляти процесом трансляції. Якщо синтаксична конструкція не є директивою, тоді виконується пошук відповідності у списку команд. На цьому етапі важливо знайти не лише співпадіння імені команди, але і перевірити, що список параметрів та список аргументів відповідають одне одному. Цикл перевірки достатньо великий, тому наводити його тут не зовсім зручно, але якщо коротко, то спершу виконується пошук команди за її ім’ям. Якщо таке мнемонічне позначення було знайдено, проводиться порівняння аргументів та параметрів. При цьому обробляються всі можливі варіанти, оскільки аргументи можуть бути різних типів (опис можливих аргументів було наведено у розділі 2.2). Якщо всі перевірки були успішні, тоді це означає, що команда знайдена, і в цьому випадку можна виділити її

код операції і занести до масиву пам’яті. Після цього поточна адреса збільшується на кількість байтів, яку займає дана команда. Такий цикл перевірок відбувається до того моменту, поки не знайдено директиви **END** або поки не досягнуто кінця лістингу програми (якщо **END** явно не вказано). Під час цього аналізу додатково формується файл протоколу трансляції, який містить інформацію про процес трансляції. На рисунку 2.15 наведено фрагмент цього протоколу:

```
0128+   0000            #DEFINE  AddC(N)
0129+   0000            {
0130+   0000                LD   B, #N          ; Завантажуємо число N до регістра B
0131+   0000                MOV  A, C           ; Читаємо в регістр A стан регістра C
0132+   0000                LD   C, SUM         ; Та завантажуємо результат додавання A+B в C
0133+   0000            }
0134+   0000            ;_________________________________________________________________________
0135+   0000            ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0136+   0000
0137+   0000            ;       .LIST
0138+   0000            ;*************************************************************************
0019    0000            ; ------------------------------------------------------------------------
0020    0000
0021    0000    43 03           LD   D, #003h       ; Готуємо регістр D для зчитування кнопок M(D.0) і N(D.1)
0022    0002
0023    0002            ; =========================  КОНТРОЛЬ МОЖЛИВИХ СТАНІВ ПРИСТРОЮ  =========================
0024    0002
0025    0002            ST00:
0026    0002    0D              MOV  B, D
0026    0003    40 02           LD   A, #00000010b
0026    0005    2C              LD   A, AND
0026    0006    C0 0F           JZ   *+9
0026    0008    40 01           LD   A, #00000001b
0026    000A    2C              LD   A, AND
0026    000B    C1 18           JNZ  *+13
0026    000D    80 16           GOTO *+9
0026    000F    40 01           LD   A, #00000001b
0026    0011    2C              LD   A, AND
0026    0012    C0 02           JZ   *-16
0027    0014    80 62           GOTO P1_01          ; Якщо натиснуто <M> -> додаємо C=C+1 та переходимо в стан 01
0028    0016    80 68           GOTO P2_10          ; Якщо натиснуто <N>   -> На додавання 2
0029    0018    80 7A           GOTO P3_11          ; Якщо натиснуті <N+M> -> На додавання 3
0030    001A
```

Рисунок 2.15 – Протокол трансляції

Даний протокол складається з декількох полів. Перший стовпчик показує номер рядка файлу. Рядки 0128-0138 записані із символом «+». Як було описано раніше, це позначення вкладеного файлу. Після того, як було оброблено директиву **include** та відкрито файл, весь код додається до загального лістингу програми, а номери рядків включають в себе додатковий символ на кінці. Другий стовпчик показує поточну адресу. У рядку 0021 записано двобайтну команду. У рядку 0022 можна помітити, що адреса збільшилась на 2. Це означає, що двобайтна команда була записана до масиву, починаючи з адреси 0000 в даному випадку, і наступна адреса була

отримана шляхом зміщення початкової адреси на 2 байти. Окрім цього, варто звернути увагу, що адресація записується у шістнадцятковому форматі. Третій стовпчик містить запис команди на машинній мові. Якщо команда однобайтна, тоді записується лише код операції. Якщо команда складається з більшої кількості байтів, то окрім коду операції вказується параметр, який було передано. Якщо це команда переходу, то в якості параметра передається адреса, на яку потрібно виконати перехід. Якщо це команда запису числа, тоді другий параметр відповідно показує число, яке необхідно записати до приймача. Останнє поле в цьому протоколі – це повний лістинг програми. У разі виникнення помилок користувач зможе швидко знайти ці місця, а не переглядати всі файли і дивитись, де саме він помилився. Такий формат є доволі зручним та легким.

Протокол трансляції сформовано, команди перевірено, помилок не знайдено. В цьому випадку завершується перший прохід трансляції. На цьому етапі вже більшість задач виконано. Проте якщо були знайдені мітки, їх треба обробити окремо. Тому далі транслятор починає виконувати другий прохід.

Функція другого проходу, **secondPass,** виконує ще один аналіз лістингу програми, але тепер перевіряються лише мітки. Ця задача є дещо простішою, оскільки в кожному рядку необхідно лише знайти мітку, що передана як параметр, і якщо у масиві вона має числове значення, додати до масиву пам'яті. В іншому випадку транслятор видасть помилку, оскільки ця синтаксична конструкція буде невідомою.

Після успішного завершення другого проходу виконується одна з двох функцій формування об'єктного файлу. Вибір залежить від параметру, який вказує користувач під час запуску трансляції. Розроблений транслятор може формувати на виході файл з розширенням .bin та .hex. Функція формування бінарного файлу доволі проста, оскільки вона просто записує масив пам'яті. Приклад бінарного файлу наведено на рисунку 2.16:

```
00000000: 43 03 0D 40 02 2C C0 0F|40 01 2C C1 18 80 16 40  | C..@.,А.@.,Б.Ђ.@
00000010: 01 2C C0 02 80 62 80 68|80 7A 0D 40 02 2C C0 27  | .,А.ЂbЂhЂz.@.,А'
00000020: 40 01 2C C0 30 80 2E 40|01 2C C1 1A 80 02 80 74  | @.,А0Ђ.@.,Б.Ђ.Ђt
00000030: 80 68 0D 40 02 2C C1 3F|40 01 2C C1 48 80 46 40  | Ђh.@.,Б?@.,БHЂF@
00000040: 01 2C C0 32 80 6E 80 02|80 62 0D 40 02 2C C1 57  | .,А2ЂnЂ.Ђb.@.,БW
00000050: 40 01 2C C0 60 80 5E 40|01 2C C1 4A 80 32 80 1A  | @.,А`Ђ^@.,БJЂ2Ђ.
00000060: 80 02 41 01 08 3E 80 1A|41 02 08 3E 80 32 41 01  | Ђ.A..>Ђ.A..>Ђ2A.
00000070: 08 3E 80 4A 41 02 08 3E|80 4A 41 03 08 3E 80 4A  | .>ЂJA..>ЂJA..>ЂJ
```

Рисунок 2.16 – Приклад вихідного коду у форматі BIN

Даний набір символів є незрозумілим для програміста, але зрозумілим для процесора. І модель розглянутого процесора, і його програмна реалізація вміють розпізнавати цей машинний код та правильно виконувати програму, написану програмістом.

Функція формування об’єктного файлу у форматі Intel HEX дещо складніша. Приклад такого файлу було наведено на рисунку 2.2. З першого погляду також незрозуміло, що це за набір символів, звідки він взявся та що означає. Щоб правильно створити такий файл необхідно було звернутись до стандарту Intel HEX та проаналізувати його структуру [18]. Вона наведена у таблиці 2.1:

Таблиця 2.1

Структура Intel HEX файлу

Record Mark ‘:’	RECLEN	LOAD OFFSET	RECTYPE	DATA	CHKSUM
1 байт	1 байт	2 байти	1 байт	n байт	1 байт
:	18	0000	00	430D…68	86

Для прикладу було взято перший рядок Intel HEX файлу з рисунку 2.2. На початку ставиться двокрапка – вказівник на початок рядку. Наступні 2 цифри є кількість байт, які потрібно записати. Цифри записуються у шістнадцятковому форматі, тому 18h = 24 байти. Після цього вказується адреса, з якої необхідно почати запис байтів. В даному випадку, початкова адреса 0000. Адреса на наступному рядку вже становить 0018h, оскільки було

записано 24 байти. Наступні 2 цифри вказують на типу запису. 00 – запис, що містить дані, 01 – запис, що сигналізує про завершення файлу. Після цього йде послідовність байтів у шістнадцятковому форматі. В розглянутому випадку кількість байт у DATA становить 24, оскільки таке значення було записано до RECLEN. Останні 2 цифри вказують на контрольну суму. Байт контрольної суми CHKSUM для рядка у файлі HEX обчислюється таким чином, щоб сума байтів усіх корисних даних у рядку та самої контрольної суми з відкиданням переповнення дорівнювала нулю. При цьому додаються не самі ASCII символи, тільки ті дані, які вони представляють. Якщо контрольна сума розрахована правильно, то при складані всіх значень, починаючи від RECLEN, повинно вийти 0. Враховуючи всі ці фактори, функція була побудована таким чином, щоб правильно обчислювати контрольні суми, записувати потрібну кількість байтів, коректно змінювати адреси та завершувати цей файл. В результаті отриманий файл було перевірено через програмну модель процесора DPM-08, яка правильно розмістила всю програму в пам’яті.

На цьому транслятор завершує свою роботу. На виході користувач отримує декілька файлів: протокол трансляції у форматі **.lst** та об’єктний файл у форматі **.bin** або **.hex**. Якщо транслятор визначив помилки на будь-якому етапі трансляції, на виході буде сформовано лише протокол трансляції, в якому будуть вказані усі проблеми.

Висновки за розділом 2

Підводячи підсумки по цьому розділу, можна сказати, що було проведено достатньо об’ємну роботу по аналізу існуючого алгоритму трансляції, розробці власного алгоритму та створенню нового програмного забезпечення. Транслятор є доволі складною програмою, яка виконує велику кількість перевірок, аналізує різноманітні ситуації та визначає всі помилки, допущені користувачем при написанні як асемблерного коду, так і при

створенні системи власних мнемонічних позначень. У зв’язку із цим перед розробкою необхідно було:

- створити основні механізми та алгоритми трансляції;
- визначитись із синтаксисом, який буде використовуватись;

Результатом розробки став новий транслятор, прототипом якого є TASM, але який має багато нових елементів:

- контроль пам’яті, аби попередити користувача, якщо код програми виходить за межі доступної пам’яті. В цьому випадку програма видасть не помилку, а попередження, яке не вплине на процес трансляції;
- використання більш сучасного синтаксису для визначення макропроцедур. Наприклад, застарілий синтаксис з використанням директиви DEFCONT було замінено звичним для мов високого рівня синтаксисом з використанням фігурних дужок, всередині яких розміщується тіло;
- мультиплатформенність, що надає можливість працювати як під операційними системами Windows, так і під Unix системами, використовуючи при цьому відкриту систему команд. Це перший транслятор, який дозволяє розробляти власний асемблер для різних процесорів на різних ОС

Створене програмне забезпечення має усі функціональні можливості, які необхідні для зручного навчання. Розроблений алгоритм має запозичення від існуючих трансляторів, проте не є повною їх копією. В ході розробки було змінено деякі синтаксичні правила, модифіковано вимоги як до створення системи мнемонічних позначень, так і до написання асемблерного коду. Подальша розробка передбачає додаткове підвищення ергономічності, спрощення користування транслятором та поєднання його з іншим програмним забезпеченням.

РОЗДІЛ 3
ВІДЛАГОДЖЕННЯ, ТЕСТУВАННЯ ТРАНСЛЯТОРА ТА РОЗРОБКА ОПИСУ КОРИСТУВАЧА

Як правило, після розробки програмного забезпечення необхідно його протестувати, виправити баги, помилки в алгоритмі або змінити логіку реалізації. Неправильно працююча програма викликає незадоволення серед користувачів і характеризує програміста як поганого спеціаліста. Також є потреба у створенні інструкції для користувачів, яким саме чином користуватись розробленою програмою. Коли мова йде про таке програмне забезпечення, як транслятор, правилом хорошого тону буде сформувати синтаксичні правила, щоб зменшити кількість помилок і надати програмісту більше інформації про те, яким чином він може створити синтаксично правильну програму, зрозумілу транслятору.

3.1. Розробка методики випробувань

Для перевірки розробленого транслятора необхідно мати програму, яка буде включати в себе всі директиви та команди користувача на прикладі конкретної системи мнемонічних позначень. Всупереч тому, що різні системи команд можуть містити різні інструкції, для перевірки транслятора достатньо того, щоб всі конструкції відповідали синтаксичним вимогам і правильно оброблялись програмою. Однією із задач в рамках лабораторного практикуму студентам пропонується розробити програму, яка додає числові значення для регістра С. Для аналізу асемблерного коду розробнику було надано приклад, який виконує описану задачу. Ця програма є чудовим варіантом для проведення тестування транслятора, оскільки вона містить в собі більшість синтаксичних конструкцій:

- Включення додаткових файлів за допомогою директиви include;
- Використання директиви .ORG для встановлення початкової адреси;
- Створення макропроцедур та використання макропроцедур;

- Використання команд перезапису даних LD, MOV;
- Використання команд відносного переходу;
- Використання команд абсолютного переходу;
- Використання директиви .END, що вказує транслятору, де він повинен завершити аналіз коду.

На рисунку 3.1 показано, як виглядає цей файл, а на рисунку 3.2 наведено фрагмент вкладеного файлу, який містить створення макропроцедур.

```
        .ORG 000h           ; Призначення адреси трансляції
; ------------------------------------------------------------------------------------------
        #Include Macros.asm ; Під'єднання бібліотеки макропроцедур
; ------------------------------------------------------------------------------------------

        LD   D, #003h       ; Готуємо регістр D для зчитування кнопок M(D.0) і N(D.1)

; =========================  КОНТРОЛЬ МОЖЛИВИХ СТАНІВ ПРИСТРОЮ  =========================

ST00:
        Stat00()            ; Контроль стану N=0, M=0
        GOTO P1_01          ; Якщо натиснуто <M> -> додаємо C=C+1 та переходимо в стан 01
        GOTO P2_10          ; Якщо натиснуто <N>   -> На додавання 2
        GOTO P3_11          ; Якщо натиснуті <N+M> -> На додавання 3

ST01:   Stat01()            ; Контроль стану N=0, M=1
        GOTO ST00           ; Якщо відпущено <M>, переходимо до стану 00
        GOTO P2_11          ; Якщо натиснуто <N>, додаємо C=C+2 та переходимо до стану 11
        GOTO P2_10          ; Якщо змінились <N+M>, додаємо C=C+2 і переходимо до стану 10

ST10:   Stat10()            ; Контроль стану N=1, M=0
        GOTO P1_11          ; Якщо натиснуто <M>, додаємо C=C+1 та переходимо до стану 11
        GOTO ST00           ; Якщо відпущено <N>, переходимо до стану 00
        GOTO P1_01          ; Якщо змінились <N+M>, додаємо C=C+1 і переходимо до стану 01

ST11:   Stat11()            ; Контроль стану N=1, M=1
        GOTO ST10           ; Якщо відпущено <M>, переходимо до стану 10
        GOTO ST01           ; Якщо натиснуто <N>, переходимо до стану 01
        GOTO ST00           ; Якщо змінились <N+M>, переходимо до стану 00

; ===============================  ЗМІНА СТАНІВ ПРИСТРОЮ  ================================

P1_01:  AddC(1)             ; Додавання C=C+1
        GOTO ST01           ; Перехід до першого стану
P2_10:  AddC(2)             ; Додавання C=C+2
        GOTO ST10           ; Перехід до другого стану
P1_11:  AddC(1)             ; Додавання C=C+1
        GOTO ST11           ; Перехід до третього стану
P2_11:  AddC(2)             ; Додавання C=C+2
        GOTO ST11           ; Перехід до третього стану
P3_11:  AddC(3)             ; Додавання C=C+3
        GOTO ST11           ; Перехід до третього стану
;----------------------------
        .END
;----------------------------
```

Рисунок 3.1 – Асемблерний файл, який дозволить протестувати роботу транслятора

```
#DEFINE  Stat00()
{
    MOV  B, D            ; Читаємо в B стан обох кнопок
    LD   A, #00000010b   ; Готуємо маску для аналіза кнопки <N>
    LD   A, AND          ; Виділяємо стан кнопки <N>
    JZ   *+9             ; Якщо кнопка <N> не натиснута -> аналіз <M>
    LD   A, #00000001b   ; Готуємо маску для аналіза кнопки <M>
    LD   A, AND          ; Виділяємо стан кнопки <M>
    JNZ  *+13            ; Якщо кнопка <M> натиснута -> вихід <Push NM>
    GOTO *+9             ; Інакше -> вихід з результатом <Push N>
    LD   A, #00000001b   ; Готуємо маску для аналіза кнопки <M>
    LD   A, AND          ; Виділяємо стан кнопки <M>
    JZ   *-16            ; Якщо кнопка <M> не натисн. -> на початок
                         ; Інакше -> вихід з результатом <Push M>
}
;______________________________________________________________________________
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;______________________________________________________________________________
;||||||||||||||||||||||||||||  ОЧІКУВАННЯ ЗМІНИ ДАНИХ ДЛЯ STAT01  ||||||||||||||||||||||||||||

;   Функція Stat00() контролює зміну стану кнопок клавіатури,  під'єднаної до порту D, при
; умові, якщо до цього жодна кнопка в розрядах D.0 та D.1 не була натиснута. Процедура пе-
; редає управління на адресу A (це адреса наступної команди), якщо буде натиснуто на кноп-
; ку <M> (D.0=1). Якщо буде натиснуто кнопку <N> (D.1=1), то управління передається на ад-
; ресу A+2,  а якщо будуть натиснуті обидві кнопки одночасно,  то управління буде передано
; на адресу A+4.
;------------------------------------------------------------------------------

#DEFINE  Stat01()
{
    MOV  B, D            ; Читаємо в B стан обох кнопок
    LD   A, #00000010b   ; Готуємо маску для аналіза кнопки <N>
    LD   A, AND          ; Виділяємо стан кнопки <N>
    JZ   *+9             ; Якщо кнопка <N> не натиснута -> аналіз <M>
    LD   A, #00000001b   ; Готуємо маску для аналіза кнопки <M>
    LD   A, AND          ; Виділяємо стан кнопки <M>
    JZ   *+13            ; Якщо кнопка <M> натиснута -> вихід <Push NM>
    GOTO *+9             ; Інакше -> вихід з результатом <Push N>
    LD   A, #00000001b   ; Готуємо маску для аналіза кнопки <M>
    LD   A, AND          ; Виділяємо стан кнопки <M>
    JNZ  *-16            ; Якщо кнопка <M> не натисн. -> на початок
                         ; Інакше -> вихід з результатом <Push M>
}
```

Рисунок 3.2 – Вкладений файл, що містить створення макропроцедур

Як видно у прикладі програми для тестування, використовуються і директиви, і різноманітні команди перезапису даних з різними аргументами, і запис числа до регістрів, і відносна та абсолютна адресація, а також представлено створення та виклик макропроцедур.

Окрім цього, необхідно також мати програму, яка буде містити помилки, аби відстежити їх обробку транслятором. Для цього можна в розглянутому файлі внести невеликі зміни, які призведуть до помилок. Також бувають нештатні ситуації, як наприклад запуск транслятора з неправильними параметрами або їх кількістю. В цьому випадку також ПЗ

повинно обробляти ці помилки, виводити відповідні повідомлення на екран, аби користувач розумів, що призвело до аварійного завершення. Частина тестів наведена у додатку В.

3.2. Тестування та відлагодження транслятора

Однією з вимог до нового транслятора є підтримка мультиплатформеності. TASM працював лише під ОС MS-DOS, що призводило до необхідності встановлювати віртуальну машину, оскільки зазначена система є застарілою і майже не використовується. Натомість велика кількість користувачів працює з ОС Windows, а певний відсоток взагалі працює під Unix-системами, наприклад Linux або MacOS. Тому новий транслятор повинен бути адаптований до потреб студентів, щоб не ускладнювати виконання лабораторного практикуму можливими проблема при встановленні віртуальної машини. На сьогоднішній день транслятор вже має підтримку Windows різних версій, починаючи з Windows XP x32 і завершуючи сучасним Windows 11 x64. Також він працює під ОС Linux Ubuntu x64. Підтримка MacOS поки що перебуває на стадії розробки.

В розділі 2.4 було вказано, що спершу для компіляції було обрано компілятор MinGW, але потім замінено на CygWin. Причиною стало виникнення проблем щодо сумісності транслятора з Windows XP та Windows 7 розрядності x32. Windows 7 x32 спершу вказувала на відсутність вхідних DLL бібліотек. Користуючись підказками ОС крок за кроком було додано всі необхідні компоненти. Проте проблеми це не вирішило. Тоді було прийнято рішення спробувати запустити транслятор під ОС Windows XP x32. Повідомлення про помилку вказало на те, що дана програма взагалі не є Win32 застосунком. Пошук варіантів вирішення цих проблем не дав результатів, тому було прийнято рішення пошукати інший компілятор, який скомпілює програму, сумісну із зазначеними системами. CygWin став чудовим варіантом. Він може компілювати файли як для систем x32 розрядності, так і для x64. Отриманий виконуваний файл спершу було

запущено на Windows 7 x32. Знову помилки про відсутність DLL бібліотек. Проте цього разу ці бібліотеки можна було знайти у директорії компілятора. Додавши до директорії проекту всі необхідні бібліотеки, транслятор повністю виконав та завершив процес трансляції. Тестування під Windows XP x32 також дало позитивний результат.

Для забезпечення сумісності транслятора з ОС Linux розглядалися різні варіанти: крос-компіляція за допомогою CygWin, використання docker контейнерів [19] або крос-компіляція через WSL [20] (Windows Subsystem for Linux). Перший варіант вимагав встановлення та налаштування додаткових пакетів. Використання docker контейнерів є дещо складним, оскільки розробник не має досвіду роботи з цим інструментом, а терміни вже були обмежені. Тому було прийнято рішення використати WSL. Він дозволяє використовувати оболонку Linux, працюючи при цьому через Windows. Для цього в командному рядку необхідно ввести команду, що встановить WSL. Після встановлення користувач отримує можливість використовувати команди Linux, тому необхідно встановити компілятор gcc. WSL надає доступ до всіх файлів системи Windows, тому достатньо перейти до директорії проекту та скомпілювати його. На виході сформується виконуваний файл, який підтримується системою Linux.

Сумісність транслятора з різними ОС була не єдиною проблемою. Після завершення основної роботи на ПЗ було розпочато тестування на правильність роботи. Однією з проблем було правильно замінити у лістингу програми параметри, які передані макропроцедурі. На рисунку 3.3 наведено приклад макропроцедури з аргументами:

```
#DEFINE  AddC(N)
{
     LD    B,  #N
     MOV   A,  C
     LD    C,  SUM
}
```

Рисунок 3.3 – Макропроцедура з аргументами

З прикладу видно, що макропроцедура приймає один параметр N, який потім використовується в її тілі як число. Оскільки під час формування загального лістингу виклик макропроцедури замінюється її тілом, необхідно було якимось чином замінювати N на параметр. Проте все тіло є текстом, і знайти серед нього потрібний символ доволі непросто. Для цього було реалізовано функцію, яка виконує поставлену задачу. У підсумку дана проблема була вирішена, наразі всі параметри правильно підставляються замість аргументів.

Під час тестування також були знайдені помилки у визначенні синтаксичних помилок. Наприклад, користувач вводить невалідний аргумент, проте транслятор не відмічає це як помилку. Або помилка була допущена в одному рядку, а вказано інший. Це все результат неправильної логіки обробки тих чи інших ситуацій. Наприклад, неправильно використаний флаг, або відсутній лічильник помилок, або лічильник неправильно підраховує кількість помилок. Також були знайдені недоліки у вирівнюванні тексту в протоколі трансляції. Під час компіляції транслятора під ОС Linux виникла проблема, спричинена однією з функцій C++, а саме **find**. Точніше було б сказати, що причиною була неповна логіка обробки випадків, коли результат, повернутий цією функцією, є негативним. Під час компіляції під Windows цей випадок був пропущений, проте Linux зосередив увагу на цьому. На сьогоднішній день всі помічені помилки були виправлені і транслятор працює правильно в більшості випадків. Не можна відкидати ймовірності, що при експлуатації будуть знайдені додаткові недоліки, проте в цьому випадку вони будуть швидко виправлені. Тож перша бета версія транслятора може бути надана студентам для тестового використання.

3.3. Опис користувача та синтаксичні правила

Транслятор поки що є консольним застосунком, оскільки він не потребує наочності, як наприклад програмна модель процесора. Це лише ПЗ, яке приймає асемблерний файл та видає об’єктний. Для запуску транслятора необхідно виконати наступні дії:

- Відкрити командний рядок вашої операційної системи;
- Переміститись до директорії, де знаходиться виконуваний файл. Асемблерний код, а також файл із системою команд повинні знаходитись в тій самій директорії. Якщо файл з асемблерним кодом включає додаткові файли, їх також необхідно розмістити в тій же директорії;
- Вказати виконуваний файл Translator.exe;
- Вказати файл з асемблерним кодом (має розширення .asm);
- Вказати файл із системою команд (має розширення .tab);
- Вказати флаг трансляції. Якщо на виході потрібно сформувати бінарний файл, треба записати «-b» або «-B». Якщо потрібен Intel HEX файл, тоді вказати «-h» або «-H».

На рисунку 3.4 показано, як повинен виглядати повноцінний рядок для запуску транслятора:

```
> Translator.exe Count-2.asm TASM08.TAB -b
```

Рисунок 3.4 – Запуск транслятора

Якщо один з параметрів буде відсутнім, транслятор не запуститься і повідомить про помилку. Запуск під ОС Linux такий самий, за винятком того, що виконуваний файл має інше розширення, і на початку треба вказати послідовність символів «./».

Оскільки щоразу виконувати ці дії незручно, можна створити BAT файл для Windows або SH скрипт для Linux. Достатньо лише помістити той самий рядок до файлу, і зберегти його з потрібним розширенням. Після цього можна запускати транслятор подвійним кліком миші по цьому файлу.

При створенні програм необхідно дотримуватись певних синтаксичних правил, аби транслятор міг правильно обробляти код та виконувати свою задачу. Тож нижче наведено перелік цих правил, які стануть у нагоді кожному користувачеві.

Правила створення файлу із системою команд:

1. Файл із системою команд повинен мати параметр, що вказує на доступний об’єм пам’яті, який можна заповнювати. Для цього треба вказати «MAX_SIZE = *об’єм пам’яті*», де об’єм пам’яті – це десяткове число. Якщо даний параметр відсутній, транслятор не несе відповідальності за можливе переповнення.
2. Таблиця мнемонічних позначень повинна містити шість полів: ім’я команди, список аргументів, код операції, кількість байт, тип команди та клас. Розділятись ці компоненти повинні мінімум одним пробілом або табуляцією. В іншому випадку транслятор некоректно визначить елементи, що призведе до помилок. Кожна наступна команда повинна бути введена з нового рядка.
3. Ім’я команди – це набір символів латинського алфавіту або кирилиці. Ім’я команди не повинно бути зарезервованим словом. Не може починатись з цифр та спеціальних символів, окрім символу «_». Може містити цифри в середині або кінці імені; серед спеціальних символів можна використовувати наступні: «-_». Вводити ім’я команди потрібно символами у верхньому регістрі. Максимально допустима довжина 10 символів. Якщо ім’я команди є довшим, воно буде обрізано до 10 символів.
4. Список аргументів розділяється комами, якщо їх кількість більша за один. Порожній список аргументів вказується парою подвійних лапок без символів всередині. Ім’я аргументу не повинно збігатися із ключовими словами або іменами команд. Так само, як і ім’я команди, не може починатися з цифр. Підтримує наступні символи на початку: _ # / * @. Може містити всередині або на кінці цифри та символи «. _». Для позначення аргументу, яке приймає число, потрібно вказати комбінацію символів «#*». Аргумент, що приймає адресу, позначається символом «*». Символ «@» використовується для непрямої адресації.

Символ «/» позначає інвертоване значення. Символьні аргументи потрібно вказувати у верхньому регістрі.

5. Код операції представлений шістнадцятковим числом. Це означає, що можна вводити лише цифри та символи латинського алфавіту, які використовуються в шістнадцятковій системі числення (A-F). Інші символи алфавіту або спеціальні символи не підтримуються.
6. Кількість байт вказується десятковим числом. Підтримуються лише цифри. Неправильно вказана кількість байтів може призвести до помилок в адресації.
7. Тип команди використовує символьні позначення. Даний компонент необхідний для правильної обробки різних команд. Наприклад, при використанні команд відносного або абсолютного переходу, транслятору необхідно вказати тип цієї команди JMP. Це буде вказувати на те, що для цих команд використовується окремий алгоритм обробки. А для команд переміщення даних, логічних та арифметичних операцій, використовується позначення NOP.
8. Клас команди є числовим значенням, поки що дорівнює просто одиниці. В даній версії програми це поле не використовується. Проте алгоритм трансляції передбачає подальше вдосконалення з можливістю долучення інших типів команд, які можуть виникнути при розробці нових процесорів.

При написанні асемблерного коду також необхідно дотримуватись певних синтаксичних правил, інакше транслятор не зможе правильно обробити вхідний код. Тому нижче описані вимоги до написання асемблерної програми:

1. Кожна команда та директива повинна записуватись в окремому рядку.
2. Коментарі можна записувати в будь-якому місці програми. Для позначення коментарів використовується символ «;». Коментарі

можуть бути багаторядковими. В цьому випадку синтаксис схожий як у С та С++: треба використовувати сполучення символів «/*» для позначення початку багаторядкового коментаря та «*/» для відображення його завершення. Треба бути уважним з багаторядковими коментарями, оскільки якщо забути закрити його, тоді весь код буде вважатись коментарем і буде проігноровано транслятором. Все, що записано у коментарі, не обробляється транслятором.

3. Ім'я директив є регістронезалежними, тому неважливо в якому регістрі їх записувати. Кожна директива має свій синтаксис, який представлений у таблиці 3.2. Неправильний запис синтаксичної конструкції призведе до того, що вона не буде розпізнана як директива.

Таблиця 3.1

Опис директив

Ім'я директиви	Список аргументів	Правильний запис	Призначення
include	ім'я включаємого файлу	#INCLUDE Macros.asm	Використовується для включення додаткових файлів до основного коду
define	Ім'я макропроцедури + список аргументів в дужках	#DEFINE macro() #define macro(a, b)	Використовується для створення макропроцедур
org	Число із вказанням системи числення	.ORG 000h .ORG 010b .ORG 000	Вказує транслятору з якої адреси записувати наступну команду
nolist	відсутні	.NOLIST	Вказує транслятору не додавати наступні рядки до протоколу трансляції

Завершення таблиці 3.2

list	відсутні	.LIST	Вказує транслятору додавати наступні рядки до протоколу трансляції
equ =	Ім’я змінної Значення змінної	X EQU 1 Y = 2	Використовується для створення користувацьких змінних
end	відсутні	.END	Вказує транслятору на завершення коду для трансляції.

4. Мітки виділяються двокрапкою на кінці. Ім’я мітки не повинно збігатися з ключовими словами або іменами команд та аргументами. Не може починатися з цифр або спеціальних символів, окрім «_». Мітки є регістрозалежними, тому **label** та **LABEL** будуть різними. Не можна оголошувати одну мітку двічі. Мітка може містити літери латинського алфавіту та кирилиці, цифри, та спеціальні символи « _». Якщо мітка передається як аргумент, її ім’я повинно співпадати з оголошеним.
5. Макропроцедури повинні бути оголошені до того, як зустрінеться їх виклик. При створенні макропроцедури потрібно використати директиву define, потім вказати ім’я макропроцедури. Після імені необхідно вказати круглі дужки незалежно від того чи є аргументи, чи ні. Далі на наступному рядку відкривається фігурна дужка, і, починаючи з наступного рядка, формується тіло макропроцедури. В кінці необхідно позначити завершення макропроцедури, записавши закриваючу фігурну дужку на наступному рядку. При виклику макропроцедури необхідно передати стільки параметрів, скільки було

вказано аргументів. В іншому випадку транслятор видасть помилку через невідомі елементи.

6. При створенні користувацьких змінних треба переконатись, що вони не є зарезервованими словами або іменами команд. Можна використовувати як директиву EQU, так і символ «=», якщо змінній присвоюється числове значення. У випадку символьної ініціалізації можна використовувати лише директиву. Змінна не може бути порожньою. Також не можна використовувати змінну, якщо вона не була оголошена перед цим. За потреби можна присвоювати змінній іншу змінну, якщо вона була ініціалізована. Присвоюючи число можна використовувати десяткову, двійкову та шістнадцяткову системи, вказуючи при цьому на кінці або b, або h. Для десяткової системи можна не вказувати нічого. Числові змінні можна передавати як параметри тим командам, які мають числовий аргумент.
7. Команди з системи мнемонічних позначень є регістронезалежними. Числові параметри необхідно передавати із вказанням системи числення. Мітки можна передавати навіть якщо вони ще не були оголошені. Бажано не плутати символи латиниці та кирилиці, оскільки транслятор сприймає їх як різні. Параметри команди розділяються комами.

Цей перелік синтаксичних правил допоможе користувачам створювати файли системи команд та асемблерні коди, дотримуючись основних вимог і зменшуючи кількість помилок.

Висновки за розділом 3

В даному розділі була розроблена методика та створено спеціальні тестові програми, за допомогою яких перевірялись 3 наступні фактори:

- Робота під різними операційними системами;
- Перевірка на виконання необхідних команд;

- Перевірка на знаходження позаштатних ситуацій та формування відповідних повідомлень під час трансляції.

Після цього описано процес тестування, в ході якого були знайдені деякі суттєві та несуттєві помилки. В результаті тестування було виконано доопрацювання програми та виправлення знайдених проблем. Етапи тестування та відлагодження є важливими, оскільки програмне забезпечення з купою помилок не матиме застосування. Окрім цього, результати роботи програми будуть спотворені або взагалі відсутні.

Після тестування було створено інструкцію для користувачів, яка має на меті показати, яким чином можна запустити транслятор, щоб він працював. Докладно описано всі основні моменти, наведено приклад того, як виглядає запуск транслятора та які параметри необхідно вказати. Потім було докладно описано синтаксичні правила, яку допоможуть користувачам створювати правильний код, зменшуючи кількість можливих помилок. Спершу було описано вимоги до файлу мнемонічних позначень, потім до асемблерного коду. Таким чином, транслятор готовий до роботи.

ВИСНОВКИ

В ході виконання дипломної роботи всі поставлені задачі були успішно виконані.

1. Було проведено аналіз декількох сучасних програм-трансляторів. Кожен має свої переваги, деякі утворюють цілі системи, які не обмежуються лише процесом трансляції. Проте всі транслятори мають загальний недолік – відсутня можливість працювати із власними системами команд для різних процесорів, не лише з відомих сімейств. Ця функціональна можливість є дуже важливою у навчальному процесі, оскільки дозволяє студентам ознайомитись з принципами програмування процесорів на рівні мнемонічних позначень. В результаті аналізу було запозичено деякі корисні алгоритмічні рішення, що були покладені в майбутню розробку.
2. Було досліджено та узагальнено основні методи та алгоритми існуючих трансляторів з метою знаходження їх недоліків та розробки власних алгоритмів. Основним прототипом майбутньої розробки став транслятор TASM – застаріле програмне забезпечення, яке стало у нагоді при проведенні навчального процесу. Даний транслятор може працювати як з відомими процесорами, так і підтримує власні розробки. Наприклад, TASM може транслювати код для моделі процесора DPM-08, який був розроблений в ХНУ ім. В.Н.Каразіна, та інших моделей процесорів, використовуючи команди, які їм властиві. На основі дослідження було розроблено власний алгоритм трансляції, який має модифікації, такі як підтримка різних ОС, підтримка кирилиці.
3. На основі розробленого алгоритму трансляції було створено новий транслятор. Це перший сучасний транслятор, який надає можливість створювати власні системи мнемонічних позначень і при цьому працює

під різними операційними системами без встановлення додаткового ПЗ. Процес розробки був доволі складним, виникало багато суперечливих моментів, функції були переписані по декілька разів. Проте у підсумку було досягнуто необхідного результату. Транслятор працює у відповідності до встановлених вимог, має свої синтаксичні правила та виконує поставлені задачі.

4. Після розробки було проведено тестування, в результаті якого знайдено та виправлено проблеми, пов'язані як з кросплатформеністю, так і з алгоритмічними рішеннями. Сюди можна віднести коректну обробку параметрів, правильне визначення помилок трансляції, правильне формування вихідного файлу. Всі ці етапи були протестовані та відлагоджені.
5. Для нового транслятора було створено інструкцію по експлуатації, щоб користувачам було одразу зрозуміло, як правильно запустити транслятор та які параметри вказати. Окрім цього описано синтаксичні правила, яких необхідно дотримуватись, аби робота транслятора давала позитивний результат, а не перелік повідомлень про помилки.

Загалом нове ПЗ має потенціал для подальшого вдосконалення. Наприклад, додатково була поставлена, але не реалізована задача по створенню віконного інтерфейсу для транслятора. Це зробить його більш зручним та ергономічним, додасть нових можливостей і зробить all-in-one. Можна інтегрувати транслятор з програмною модель процесора DPM-08 та редактором, наприклад Notepad++. Таким чином достатньо відкрити один застосунок, який об'єднає декілька інших. Це може стати основою для подальшої розробки повноцінного інтегрованого середовища проектування. Проте цього не було реалізовано в рамках даної роботи, оскільки подібна розробка потребує значно більше часу.

Підсумками проведеного дослідження є наступні основні результати:

1) Вперше розроблено транслятор асемблера, алгоритми трансляції якого дозволяють використання зовнішнього файлу із системою команд і можуть працювати під різними видами операційних систем;
2) Вдосконалено алгоритми трансляції, що надало можливість контролювати об’єм доступних ресурсів під час трансляції;
3) Отримав подальший розвиток набір синтаксичних правил, що використовувався у трансляторі-прототипі за рахунок уніфікації деяких процедур відповідно до синтаксису мов високого рівня.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Журавель Ю. А. Модель цифрового процесора / Ю. А. Журавель, С. Н. Рева // Вісник Харківського національного університету імені В.Н. Каразіна. Серія : Математичне моделювання. Інформаційні технології. Автоматизовані системи управління. 2011. № 987, Вип. 18. С. 5-18 [Електронний ресурс]. – режим доступу: URL: http://nbuv.gov.ua/UJRN/VKhIMAM_2011_987_18_3 (дата звернення: 15.09.2024).
2. Рева С.М., Комеристий В.С. Програмний емулятор навчальної моделі цифрового процесора. Вісник Харківського національного університету імені В. Н. Каразіна, сер. «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління». 2023. т. 58. С.54-63. https://doi.org/10.26565/2304-6201-2023-58-06 (дата звернення: 15.09.2024)
3. What is Assembly Language? [Електронний ресурс] – режим доступу: URL: https://www.geeksforgeeks.org/what-is-assembly-language/ (дата звернення: 15.09.2024)
4. The Telemark Assembler (TASM) User's Manual [Електронний ресурс] – режим доступу: URL: https://www.mikrocontroller.net/attachment/339967/TASMMAN.pdf (дата звернення: 15.09.2024)
5. MS-DOS operating system [Електронний ресурс] – режим доступу: URL: https://www.britannica.com/technology/MS-DOS (дата звернення: 15.09.2024)
6. VASM assembler system [Електронний ресурс] – режим доступу: URL: http://sun.hasenbraten.de/vasm/release/vasm.pdf (дата звернення: 15.09.2024)

7. GNU assembler manual [Електронний ресурс] – режим доступу: URL: http://microelectronics.esa.int/erc32/doc/as.pdf (дата звернення: 15.09.2024)
8. MinGW-w64 [Електронний ресурс] – режим доступу: URL: https://www.mingw-w64.org/ (дата звернення: 16.09.2024)
9. Cygwin: Documentation [Електронний ресурс] – режим доступу: URL: https://www.cygwin.com/docs.html (дата звернення: 16.09.2024)
10. Flat assembler: documentation and tutorials [Електронний ресурс] – режим доступу: URL: https://flatassembler.net/docs.php?article=manual#1.1 (дата звернення: 16.09.2024)
11. NASM documentation [Електронний ресурс] – режим доступу: URL: http://www.nasm.us/xdoc/2.16.03/nasmdoc.pdf (дата звернення: 16.09.2024)
12. LLVM documentation [Електронний ресурс] – режим доступу: URL: https://llvm.org/docs/index.html (дата звернення: 17.09.2024)
13. TableGen Programmer’s Reference [Електронний ресурс] – режим доступу: URL: https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/TableGen/ProgRef.html#introduction (дата звернення: 17.09.2024)
14. Activity Diagrams – Unified Modeling Language (UML) [Електронний ресурс] – режим доступу: URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/ (дата звернення: 05.10.2024)
15. Огляд і основи мови програмування С++ [Електронний ресурс]. – режим доступу: URL: http://www.znannya.org/?view=Cpp_basics (дата звернення : 05.10.2024)
16. C++17 [Електронний ресурс] – режим доступу: URL: https://en.cppreference.com/w/cpp/17 (дата звернення 05.10.2024)

17. Structures in C++ [Електронний ресурс] – режим доступу: URL: https://www.geeksforgeeks.org/structures-in-cpp/ (дата звернення 05.10.2024)
18. General: Intel HEX File Format [Електронний ресурс] – режим доступу: URL: https://developer.arm.com/documentation/ka003292/latest/ (дата звернення: 10.10.2024)
19. Use containers to Build, Share and Run your applications [Електронний ресурс] – режим доступу: URL: https://www.docker.com/resources/what-container/ (дата звернення: 10.10.2024)
20. Ubuntu WSL [Електронний ресурс] – режим доступу: URL: https://documentation.ubuntu.com/wsl/en/latest/ (дата звернення: 10.10.2024)

ДОДАТКИ

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп’ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **Магістр**
Галузь знань: 12 – Інформаційні технології
Спеціальність: 123 – Комп’ютерна інженерія
Освітня програма «Комп’ютерна інженерія»

ЗАТВЕРДЖУЮ
в.о. завідувача кафедри комп’ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«12» вересня 2024року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Комеристого Владислава Сергійовича
(прізвище, ім’я, по батькові студента)

1. Тема роботи: **«Модель оптимізації алгоритмів трансляції асемблерних програм»**

керівник роботи: Рева Сергій Миколайович, к.т.н.
затверджені наказом по університету ***№ 4101-5/3657 від 12 листопада 2024 року***

2. Строк подання студентом роботи: **30 листопада 2024 року**
3. Перелік питань, які потрібно розробити

1) Провести аналіз предметної області
2) Дослідити принципи роботи трансляторів на основі існуючих рішень
3) Визначити проблемні місця в існуючих рішеннях та розглянути варіанти їх вирішення
4) Розробити загальний алгоритм для побудови транслятора
5) Створити оптимізовану модель транслятора
6) Провести тестування розробленої моделі
7) Написати вимоги до асемблерних програм, створених студентами
8) Підготувати пояснювальну записку, доповідь та презентацію

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Проведення аналізу предметної області, пошук та вивчення літератури	Вересень 2024
2	Дослідження трансляції асемблерних програм, пошук слабких місць	Вересень 2024
3	Пошук рішень покращення процесу трансляції	Вересень 2024
4	Розробка загального алгоритму транслятора	Жовтень 2024
5	Розробка транслятора асемблерних програм, відлагодження готового рішення	Жовтень 2024 – Листопад 2024
6	Написання синтаксичних правил щодо створення асемблерних програм для коректної обробки транслятором	Листопад 2024
7	Оформлення пояснювальної записки	Листопад 2024
8	Створення презентації, підготовка до доповіді	Листопад 2024

5. Дата видачі завдання ***0 5вересня 2024 року.***

Студент В.С.Комеристий
ініціали, прізвище — підпис

Керівник роботи С.М.Рева
ініціали, прізвище — підпис

Додаток Б

Затверджую

____

«_____»__________2024 р.

Технічне завдання

на розробку програмного виробу «Модель оптимізації алгоритмів трансляції асемблерних програм**»**

1.	Введення	1.1. Назва: Модель оптимізації алгоритмів трансляції асемблерних програм. 1.2. Галузь застосування: навчальний процес
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 123 – Комп'ютерна інженерія 2.2. Завдання на кваліфікаційну роботу магістра № ***4101-5/3657*** від ***12 листопада 2024 року*** (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета розробки: покращення навчального процесу. 3.2. Призначення розробки: надання можливості використовувати програмне забезпечення для виконання лабораторного практикуму без встановлення додаткових засобів. 3.3. Вихідні дані розробки: прототип TASM, сучасні програми-транслятори.
4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: сумісність з операційними системами, правильність обробки, підтримка власної системи мнемонічних позначень. 4.2. Вимоги до надійності: стабільність роботи, здатність коректно завершити роботу у випадку знайдених помилок. 4.3.Вимоги до умов експлуатації: комп'ютер із встановленою операційною системою Windows, Linux або MacOS. Потреб у високих обчислювальних потужностях немає. 4.4. Вимоги до складу і параметрів технічних засобів: комп'ютер або ноутбук із встановленою операційною системою 4.5. Вимоги до інформаційної та програмної сумісності: програмне забезпечення має сумісність із

<table>
<tr><td></td><td></td><td>різними версіями та розрядностями Windows, також підтримує Linux та на стадії розробки підтримка MacOS.
4.6. Вимоги до маркування та упаковки: відсутні
4.7. Вимоги до транспортування і зберігання: відсутні
4.8. Спеціальні вимоги: немає.</td></tr>
<tr><td>5.</td><td>Вимоги до програмної документації</td><td>Програмною документацією до виробу «Метод аналізу інформативності змінних стану при діагностиці систем з використанням інформаційних критеріїв» вважати:
1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи).
2) Інструкція по використанню (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи)
3) Опис синтаксичних правил та вимог (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи)</td></tr>
<tr><td>6.</td><td>Вимоги до техніко-економічних показників</td><td>Техніко-економічні показники відсутні.</td></tr>
<tr><td>7.</td><td>Стадії і етапи розробки</td><td>
<table>
<tr><th>№ з/п</th><th>Назви етапів роботи</th><th>Термін виконання етапів роботи</th></tr>
<tr><td>1</td><td>Проведення аналізу предметної області, пошук та вивчення літератури</td><td>Вересень 2024</td></tr>
<tr><td>2</td><td>Дослідження трансляції асемблерних програм, пошук слабких місць</td><td>Вересень 2024</td></tr>
<tr><td>3</td><td>Пошук рішень покращення процесу трансляції</td><td>Вересень 2024</td></tr>
<tr><td>4</td><td>Розробка загального алгоритму транслятора</td><td>Жовтень 2024</td></tr>
</table>
</td></tr>
</table>

		5	Розробка транслятора асемблерних програм, відлагодження готового рішення	Жовтень 2024 – Листопад 2024
		6	Написання синтаксичних правил щодо створення асемблерних програм для коректної обробки транслятором	Листопад 2024
		7	Оформлення пояснювальної записки	Листопад 2024
		8	Створення презентації, підготовка до доповіді	Листопад 2024
8.	Порядок контролю і приймання програмного продукту (моделі)	1. Перевірку ходу розробки програми виконувати раз в 3 тижні. 2. Захист розробленої моделі провести на засіданні Атестаційної комісії. 3. Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді в 1 примірнику.		

Виконавець
студент групи КІ- 61
Комеристий В.С.

Замовник
к.т.н., доцент
Рева С.М.

Додаток В

Програма і методика випробувань програмного виробу
«Модель оптимізації алгоритмів трансляції асемблерних програм»

1. Об'єкт випробувань

1. Назва програмного виробу: «Модель оптимізації алгоритмів трансляції асемблерних програм»
2. Галузь застосування: навчальний процес
3. Перераховані відомості запозичуються з відповідних розділів Технічного завдання.

2. Мета випробувань

Перевірка відповідності функціональності програмної реалізації системи заявленим функціональним можливостям в технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи).

3. Загальні положення

1) **Підстави для проведення випробувань**

Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

2) **Місце і тривалість випробувань**

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри в період роботи атестаційної комісії.

3) **Обсяг випробувань**

Приймальні випробування програмного вироби проводяться в обсязі відповідному цієї програми і методики випробувань.

4) **Організації, які беруть участь у випробуваннях**

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

1) працювати на основних операційних системах: Windows, Linux, MacOS;
2) вимоги до надійності;
3) передбачити захист від некоректних дій користувача;
4) сумісність з іншими програмними продуктами;
5) бути легко розширюваною;
6) елементи програми повинні бути ізольовані одне від одного для зменшення їх впливу на роботи програми під час редагування програмного коду;
7) вимоги до складу і параметрів технічних засобів;

8) вимоги до функціональності;
9) вимоги до маркування та упаковки (не висуваються);
10) вимоги до транспортування і зберігання (не висуваються).

Спеціальні вимоги (не пред’являються).

5. Вимоги до програмної документації

Програмною документацією до виробу «Модель оптимізації алгоритмів трансляції асемблерних програм» вважати:

1. справжнє технічне завдання на розробку програми (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи);
2. Програму і методику випробувань розробленої програми (представити як Додаток В до пояснювальної записки до кваліфікаційної роботи);
3. Опис користувача з використання створеної програми (представити в Розділі 3 пояснювальної записки до кваліфікаційної роботи).
4. Текст програми (представити як Додаток Г до пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань необхідно створити асемблерний файл, в якому вказати усі можливі синтаксичні конструкції та протестувати їх роботу.

6.2 Порядок проведення випробувань

Як правило, випробування проводяться в два етапи:
-ознайомчий (1-й етап);
-випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає в себе:

1. Перевірку комплектності програмної документації.
   1.1. Перевірка комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в ТЗ документації.
2. Перевірку комплектності складу технічних і програмних засобів.
3. Методику проведення перевірок на 1 етапі випробувань.
4. Якість програмної документації перевіряється на відповідність вимогам стандартів ЕСПД.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає в себе:

1. перевірку відповідності технічних характеристик програми вимогам технічного завдання;
2. перевірку ступеня виконання функціональних вимог до програми;
3. методику проведення перевірок, що входять до переліку по 2 етапу випробувань.

1. Програма працює відповідно до умов експлуатації операційних систем Windows та Linux.
2. Для роботи необхідний набір тестових файлів для перевірки функціональних можливостей транслятора та їх відповідності технічному завданню.
3. Порядок проведення випробувань:
   3.1. Запуск програми здійснюється за допомогою командного рядка. Для цього необхідно вказати ім’я виконуваного файлу Translator.exe, після чого передати 3 параметри: ім’я файлу для трансляції, файл із системою команд, та флаг для визначення формату об’єктного файлу (-b або -h). У випадку некоректного запуску буде виведено повідомлення про помилки.
   3.2. Після запуску програми у разі успішного виконання транслятор вказує, що обидва проходи були успішні і було знайдено 0 помилок. Формується файл протоколу трансляції, а також об’єктний файл;
   3.3. У разі виникнення помилок користувач отримає повідомлення про них. Транслятор завершить свою роботу і сформує файл протоколу із детальним описом помилок;
   3.4. Для завершення роботи транслятора необхідно натиснути кнопку «Enter».

Для проведення випробувань пропонується тест 1, тест 2, тест 3, тест 4, тест 5, тест 6.

Тест 1

1. Запуск транслятора з командного рядка;
2. Перевірка працездатності програми при коректних параметрах.

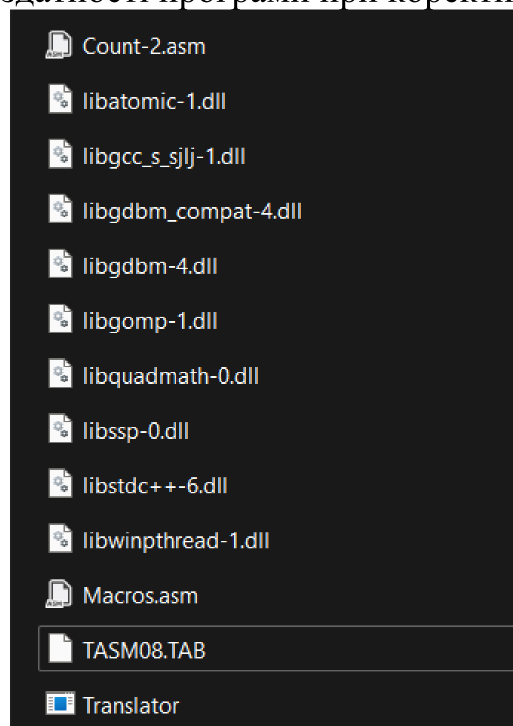

Рис. В.1 Тест 1 – Вміст директорії перед трансляцією

```
>Translator.exe Count-2.asm TASM08.TAB -h
First pass complete.
Second pass complete.
Number of errors: 0
```

Рис. В.2 Тест 1 – Успішна трансляція

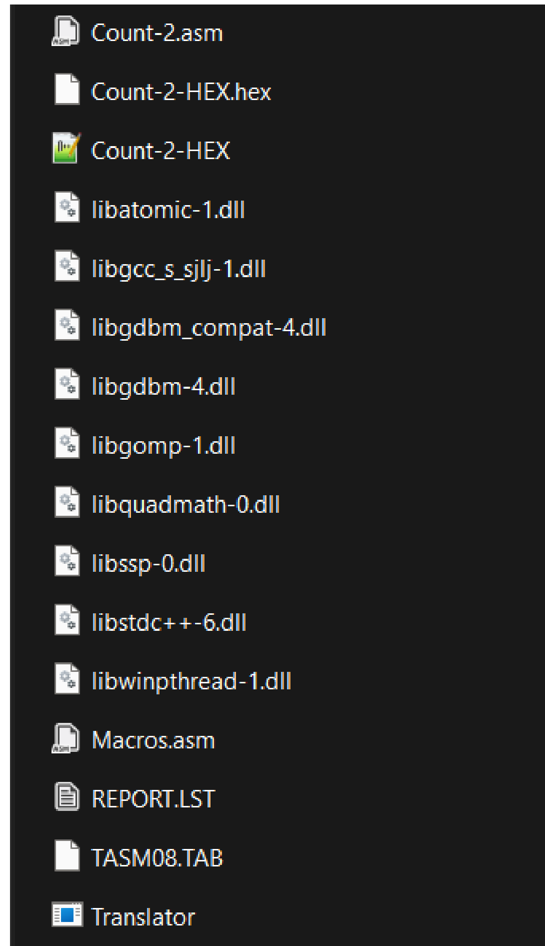

Рис. В.3 Тест 1 – Вміст директорії після трансляції

Тест 2

1. Запуск транслятора з командного рядка;
2. Перевірка працездатності програми при некоректних вхідних параметрах.

```
>Translator.exe Count-2.asm TASM08.TAB
Error: not enough arguments.
```

Рис. В.4 Тест 2 – Запуск з недостатньою кількістю параметрів

```
>Translator.exe Count-2.b TASM08.TAB -h
Error: file doesn't contain.
```

Рис. В.5 Тест 2 – Запуск із вказанням файлу, якого не існує в директорії

Тест 3

3. Запуск транслятора з командного рядка;

4. Перевірка роботи програми при навмисному допущенні синтаксичних помилок у директиві.

```
>Translator.exe Count-2.asm TASM08.TAB -h
Translation failed: file Count-2.asm contains errors. Check REPORT.LST for details.
```

Рис. В.6 Тест 3 – Запуск транслятора при наявних помилках у коді

```
0017    ; -----------------------------------------------------------------
0018          #Include Macros ; Під'єднання бібліотеки макропроцедур
Line 0018 in Count-2.asm: Error: file doesn't exist.
0019    ; -----------------------------------------------------------------
```

Рис. В.7 Тест 3 – Протокол із вказанням помилки

Тест 4

1. Запуск транслятора з командного рядка;
2. Перевірка роботи програми при навмисному допущенні синтаксичних помилок у макропроцедурі.

```
0016+   #DEFINE  Stat00(
Line 0016 in Macros.asm: Error: macro procedure must has parenthesis even if arguments not be contained.
0017+
```

Рис. В.8 Тест 4 – Помилка при ініціалізації макропроцедури

Тест 5

1. Запуск транслятора з командного рядка;
2. Перевірка роботи програми при навмисному допущенні синтаксичних помилок при введені інструкції.

```
0022     0002                    LD   M, #001
Line 0022 in Count-2.asm: ArgumentsError: unrecognized arguments.
```

Рис. В.9 Тест 5 – Помилка при виклику команди

Тест 6

1. Запуск транслятора під операційною системою Linux;
2. Перевірка працездатності програми при коректних параметрах

```
root@AcerAspire7:/mnt/d/Documents/CLionProjects/Translator_V2# ./Translator Count-2.asm TASM08.TAB -b
First pass complete.
Second pass complete.
Number of errors: 0
```

Рис. В.10 Тест 6 – Успішна робота транслятора під Linux

Тест вважається пройденим, якщо відбуваються вказані операції і їх відображення у програмному продукті.

Висновки: всі тести були проведені. Успішні тести показали відповідний результат. Далі було навмисно допущено помилки, щоб показати коректність їх обробки та сповіщення користувача про них. Випробування проведено успішно.

Виконавець: студент групи КІ-61, Комеристий В.С.

Додаток Г

Фрагменти коду розробленого транслятора

```
#include "MyFunctions.h"

int main(const int argc, char* argv[])
{
    system( Command: "cls");
    SetConsoleCP( wCodePageID: 1251);
    SetConsoleOutputCP( wCodePageID: 1251);
    if(argc <= 1) {
        cout << "You must enter arguments!\n";
        cin.get();
        exit( Code: -1);
    }

    if(argc != 4) {
        cout << "Error: not enough arguments.\n";
        cin.get();
        exit( Code: -1);
    }

    if(toUpperStr( str: argv[3]) != "-B" && toUpperStr( str: argv[3]) != "-H") {
        cout << "Error: undefined extension for output file.\n";
        cin.get();
        exit( Code: -4);
    }

    string filename = argv[2];

    ifstream in_file;
    in_file.open( s: argv[2]);
    if(!in_file.is_open()) {
        cout << "Error: file doesn't contain.\n";
```

```
        cin.get();
        exit( Code: -2);
    }

    optional<int> max_size = nullopt;
    analyzeCommandsFile( &: in_file,  &: max_size);
    in_file.close();

    if(commands_list.empty()){
        cout << "Translation failed: File with commands contains mistakes. All mistakes you can see in file REPORT.LST\n";
        cin.get();
        exit( Code: -3);
    }

    in_file.open( s: argv[1]);
    if(!(in_file.is_open())) {
        cout << "Error: file doesn't contain.\n";
        cin.get();
        exit( Code: -2);
    }

    map<string, string> fileHierarchy;
    createGeneralListing( &: in_file,  filename: argv[1],  &: fileHierarchy);
    in_file.close();

    if(gen_list.empty()) {
        cout << "Translation failed: file Count-2.asm contains errors. Check REPORT.LST for details.\n";
        cin.get();
        exit( Code: -3);
    }
```

```
63      int address = 0;
64      firstPass(fileHierarchy, &: address);
65
66      if(final_report.empty()) {
67          cout << "Translation failed: errors were found during the translation. Check REPORT.LST for details.\n";
68          cin.get();
69          exit( Code: -3);
70      }
71
72      if(max_size) {
73          if(address > max_size.value()) {
74              cout << "Warning: size of program bigger than maximum available memory. Your program will cropped.\n";
75          }
76      } else {
77          cout << "Warning: undefined memory size. Your program may be cropped.\n";
78      }
79
80      secondPass(fileHierarchy);
81
82      if(final_report.empty()) {
83          cout << "Translation failed: errors were found during the translation. Check REPORT.LST for details.\n";
84          cin.get();
85          exit( Code: -3);
86      }
```

```
88      int memory[256];
89      fill_n( first: memory,  n: 256,  value: -1);
90      if(toUpperStr( str: argv[3]) == "-B")
91          binaryGenerating(memory,  size: 256,  filename: argv[1]);
92      else
93          intelHEXGenerating(memory,  size: 256,  filename: "Count-2.asm");
94
95
96      gen_list.clear();
97      fileHierarchy.clear();
98      final_report.clear();
99      cin.get();
100     return 0;
101 }
```

```
12  void analyzeCommandsFile(ifstream &com_file, optional<int> &max_memory)
13  {
14      optional<int> num_value;
15      vector<pair<optional<int>, string>> report;
16      set<string> command_names;
17      instruction command;
18
19      string temp_str;
20      int line = 1;
21      string str_report;
22      bool isComment = false;
23
24      while(getline( &: com_file,  &: temp_str))
25      {
26          report.emplace_back( a: line++,  b: temp_str);
27
28          // check if the string is empty or it's comment
29          temp_str = removeLeadingSpaces( str: temp_str);
30          if(temp_str.empty() || temp_str.at( n: 0) == ';') {
31              continue;
32          }
33          if(temp_str.substr( pos: 0,  n: 2) == "/*") {
34              isComment = true;
35              continue;
36          }
37          if((temp_str.substr( pos: temp_str.length() - 2,  n: 2) == "*/") && isComment) {
38              isComment = false;
39              continue;
40          }
```

```
58      for(int count = 1; it != report.end(); count++, it++) {
59          temp_str = removeLeadingSpaces(it->second);
60          temp_str.erase(temp_str.find_last_not_of(" \t\n\r\f\v") + 1);
61          stringstream line_to_str;
62          line_to_str << setw(4) << setfill('0') << count;
63          string error_info = "Line " + line_to_str.str() + ": ";
64          string error_msg;
65          if(temp_str.empty() || temp_str.at(0) == ';') {
66              continue;
67          }
68          if(temp_str.substr(0, 2) == "/*") {
69              isComment = true;
70              continue;
71          }
72          if(isComment && temp_str.length() > 2) {
73              if(temp_str.substr(temp_str.length() - 2, 2) == "*/") {
74                  isComment = false;
75                  continue;
76              }
77          }
78          if(temp_str.find("MAX_SIZE") != string::npos) {
79              istringstream iss(temp_str);
80              iss >> temp_str;
81              iss >> temp_str;
82              if(temp_str == "=") {
83                  iss >> temp_str;
84                  num_value = strToInt(temp_str);
85                  if(num_value) {
86                      max_memory = *num_value;
```

```
308 void createGeneralListing(ifstream &main_file, const string &filename, map<string, string> &fileHierarchy)
310     string temp_str;
311     string substr;
312     int line = 0;
313     bool isMacro = false;
314 
315     static int num_of_errors = 0;
316     static int depth = 0;
317     static string degree_of_nesting;
318     string current_macro_name;
319 
320     fileHierarchy[degree_of_nesting] = filename;
321 
322     while(getline(main_file, temp_str)) {
323         line++;
324         stringstream ss;
325         ss << setw(4) << setfill('0') << to_string(line);
326         string num_of_line = ss.str();
327         num_of_line += degree_of_nesting;
328         gen_list.emplace_back(num_of_line, temp_str);
329         auto pos = num_of_line.find('+');
330         if(pos != string::npos) num_of_line = num_of_line.substr(0, pos);
331         string error_info = "Line " + num_of_line + " in ";
332         error_info += filename + ": ";
333         string error_msg;
334         auto position = temp_str.find(';');
335         if(position != string::npos)
336             temp_str = temp_str.substr(0, position);
337         position = temp_str.find(':');
338         string label = "";
```

```
348         if(toUpperStr( str: substr).find( s: "#INCLUDE") != string::npos) {
349             while(iss >> substr) words.push_back(substr);
350             if(words.size() > 2) {
351                 error_msg = error_info + "Error: string contains superfluous characters.";
352                 gen_list.emplace_back( a: "",  b: error_msg);
353                 num_of_errors++;
354             } else {
355                 ifstream temp_file( s: words.at( n: 1));
356                 if (!temp_file.is_open()) {
357                     error_msg = error_info + "Error: file doesn't exist.";
358                     gen_list.emplace_back( a: "",  b: error_msg);
359                     num_of_errors++;
360                 } else {
361                     depth++;
362                     degree_of_nesting += '+';
363                     createGeneralListing( &: temp_file,  filename: words.at( n: 1),  &: fileHierarchy);
364                     depth--;
365                     degree_of_nesting.pop_back();
366                 }
367             }
368             words.clear();
369         }
```

```
370         if(toUpperStr( str: substr).find( s: "#DEFINE") != string::npos) {
371             iss >> substr;
372             getline( &: iss,  &: substr);
373             substr = removeLeadingSpaces( str: substr);
374             macroProcedure temp;
375             pos = substr.find( c: '(');
376             position = substr.find( c: ')');
377             if(pos != string::npos && position != string::npos) {
378                 macro_name = substr.substr( pos: 0,  n: pos);
379                 macro_name = removeLeadingSpaces( str: macro_name);
380                 macro_name.erase( pos: macro_name.find_last_not_of( s: " \t\n\r\f\v") + 1);
381                 temp.name = macro_name;
382                 auto result :iterator<...>  = find_if( first: macros.begin(),  last: macros.end(),
383                                             pred: [&macro_name](macroProcedure& m) -> bool  {
384                                                 return m.name == macro_name;
385                                             });
386                 bool isErrors = false;
387                 if(result == macros.end()) {
388                     vector<string> arguments = split( str: substr.substr( pos: pos + 1,  n: position - pos - 1),  delimiter: ',');
389                     if(!arguments.empty()) {
390                         for (const auto &args : string const & : arguments) {
391                             auto isKeyword : const_iterator<string>  = keywords.find( x: args);
392                             if (isKeyword != keywords.end()) {
393                                 isErrors = true;
394                             }
395                             for (const auto &cmd : instruction const & : commands_list) {
396                                 if (strcmp(args.c_str(), cmd.name) == 0) {
397                                     isErrors = true;
398                                     break;
```

```
626                 // checking of correctness var_name
627                 if ((ispunct( C: var_name.at( n: 0)) && var_name.at( n: 0) != '_') || isdigit( C: var_name.at( n: 0))) {
628 //                  error_msg = error_info + "\033[31m" + "Error: " + "\033[0m";
629                     error_msg += error_info + "VariableNameError: variable name cannot begin with digits or special "
630                                              "characters except '_'.";
631                     error_report.emplace_back( a: temp_element.second,  b: error_msg);
632                     num_of_errors++;
633                     isErrorsEQU = true;
634                 }
635                 bool isCorrectName = all_of( first: var_name.begin() + 1,  last: var_name.end(),  pred: [](unsigned char c) -> bool  {
636                     return isalnum(c) || (c == '_');
637                 });
638                 if (!isCorrectName) {
639                     error_msg = error_info + "VariableNameError: name contains forbidden characters.";
640                     error_report.emplace_back( a: temp_element.second,  b: error_msg);
641                     num_of_errors++;
642                     isErrorsEQU = true;
643                 }
```