Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА АСЕМБЛЕРА ДЛЯ СИМУЛЯТОРА ВОСЬМИБІТНОГО ПРОЦЕСОРА

ASSEMBLER DEVELOPMENT FOR THE EIGHT-BIT PROCESSOR SIMULATOR

спеціальність 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

освітня програма «Інженерія програмного забезпечення»
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи ІПЗ-41
Міщук Олександр Сергійович
_____
(підпис)

Керівник: к.т.н., доцент
Суринович Олена Миколаївна
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«8» 06 2023 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна
_____
(підпис)

Луцьк – 2023 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *121 Інженерія програмного забезпечення*
Освітня програма: *«Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«02» 02 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Міщуку Олександру Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Розробка асемблера для симулятора восьмибітного процесора
Керівник роботи: к.т.н., доцент Суринович Олена Миколаївна
затверджені наказом закладу вищої освіти від «*23*» *грудня* 2022 р. № *961-01-02*

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи « *8* » *червня* 2023 р.

3. Вихідні дані до роботи: *технічне та програмне забезпечення для розробки ПЗ, вимоги до фінального продукту, використовувані технології: мови програмування C++ та Python, перелік технічних специфікацій процесора MOS Technology 6502, література, статті та нормативи за темою дослідження.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *Огляд природи асемблерів та сучасних тенденцій їх розвитку; огляд базової архітектури асемблерів; постановка завдання на кваліфікаційну роботу; аналіз вимог до розроблюваної системи та технічних специфікацій процесора, під який даний асемблер розроблятиметься; проектування архітектури продукту; вибір засобів та методів реалізації ПЗ; розробка асемблера та його тестування.*

5. Перелік графічного матеріалу: *5 рисунків, з них: 3 схеми, 2 діаграми.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Суринович О.М.*		
Специфікація вимог до розроблюваної системи	*Суринович О.М.*		
Розробка об'єкта проектування	*Суринович О.М.*		
Нормоконтроль	*Суринович О.М.*		
Гарант ОП	*Ліщина Н.М.*		
Показник запозичень тексту	152 %		
Академічна доброчесність	*Ящук А.А.*		

7. Дата видачі завдання «2» *лютого* 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Огляд літературних джерел по темі кваліфікаційної роботи бакалавра*	*05.02.2023 р.*	вик.
2	*Аналіз проблеми розробки та впровадження об'єкту проектування*	*27.02.2023 р.*	вик.
3	*Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання*	*10.03.2023 р.*	вик.
4	*Розробка функціонально-структурної схеми роботи об'єкта проектування*	*17.04.2023 р.*	вик.
5	*Практична реалізація об'єкта проектування*	*18.05.2023 р.*	вик.
6	*Тестування і налагодження інформаційної системи*	*01.06.2023 р.*	вик.
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*08.06.2023 р.*	вик.

Здобувач вищої освіти _____ Міщук О.С.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Суринович О.М.
(підпис) (прізвище, ініціали)

Міністерство освіти і науки України
Луцький національний технічний університет

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач вищої освіти: *Міщук Олександр Сергійович*
(ПІБ)

Тема кваліфікаційної роботи: *Розробка асемблера для симулятора восьмибітного процесора/ Assembler development for the eight-bit processor simulator.*
(повна назва згідно наказу)

Коротка характеристика кваліфікаційної роботи та прийнятих рішень: *Робота містить три розділи, у яких розглядаються теоретичні відомості щодо будови трансляторів, зокрема, асемблерів, а також, здійснюється проектування та розробка власного асемблера. Для практичної реалізації завдання використовуються мови програмування C++ та Python.*

Самостійні розробки і пропозиції автора: *Дипломантом був розроблений асемблер для набору інструкцій процесора MOS Technology 6502. Після реалізації автором роботи були запропоновані ідеї для розвитку продукту в майбутньому у вигляді розширення як функціоналу, так і кількості підтримуваних платформ.*

Практичне значення роботи: *Розроблений дипломантом продукт може використовуватись, як за прямим призначенням – для збірки програм під процесор MOS Technology 6502, так і у демонстраційних цілях для наочного навчання майбутніх спеціалістів, що займатимуться розробкою трансляторів.*

Недоліки: *Суттєвих недоліків не виявлено.*

Загальний висновок: *У кваліфікаційній роботі бакалавра дипломантом продемонстрований високий рівень теоретичних та практичних знань і навичок, як в сфері предметної області, так і в інженерії програмного забезпечення в цілому.*
Здобувач освіти в повній мірі реалізував усі поставлені завдання. Кваліфікаційна робота виконана на високому рівні та рекомендується до захисту Екзаменаційній комісії та заслуговує на оцінку «відмінно».

Рецензент _____ (*к.т.н., доц., доцент кафедри КН, Лук'янчук Ю.А.*)
(підпис) (науковий ступінь, вчене звання, посада, ПІБ)

«8» червня 2023 р.

АНОТАЦІЯ

Міщук О.С. Розробка асемблера для симулятора восьмибітного процесора. Рукопис.

Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності 121 Інженерія програмного забезпечення. Луцький національний технічний університет. Луцьк, 2023.

Кваліфікаційна робота бакалавра складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків (згідно структури кваліфікаційної роботи, затвердженої кафедрою).

Робота присвячена дослідженню асемблерів, принципів їх побудови та функціонування, а також, реалізації власного асемблера для набору інструкцій процесора MOS Technology 6502 (згідно змісту кваліфікаційної роботи бакалавра).

Ключові слова: асемблер, транслятор, компілятор, інтерпретатор, лексер, парсер, мова програмування C++, машинний код, набір інструкцій, процесор MOS Technology 6502.

ABSTRACT

Mishchuk O.S. Assembler development for the eight-bit processor simulator. – Manuscript.

Bachelor thesis of the educational program «Software Engineering» in specialty 121 Software Engineering. Lutsk National Technical University. Lutsk, 2023.

Bachelor thesis consists of introduction, three sections, conclusions, references and appendixes (according to the bachelor thesis structure, approved by the department).

The thesis is devoted to the research of assemblers, the principles of their construction and functioning, as well as the development of own assembler for the MOS Technology 6502 processor instruction set (according to the content of the bachelor's thesis).

Keywords: assembler, translator, compiler, interpreter, lexer, parser, C++ programming language, machine code, instruction set, MOS Technology 6502 processor.

ЗМІСТ

ВСТУП

Транслятори програмного коду сьогодні відіграють ключову роль у розробці усіх видів програмного забезпечення. Оскільки, написання програм здійснюється з використанням високорівневих мов програмування, добре зрозумілих людині, але незрозумілих комп'ютеру, постає питання щодо їх максимально ефективної трансляції у машинні команди, котрі уже зможе виконувати обчислювальна машина. Для цього існують спеціальні програмні засоби, названі трансляторами. Вони відрізняються за принципами своєї роботи, внутрішнім влаштуванням, продуктивністю, швидкістю, переносимістю, призначенням, тощо. Трохи окремо тримаються асемблери – транслятори коду на мові асемблера. Їх головна мета – перетворення низькорівневого коду у машинні команди.

Важливим аспектом при розгляді проблеми побудови асемблерів є низька переносимість їх коду. Так, кожна апаратна архітектура (будь то x86, ARM, PIC, чи інша) має свій власний набір інструкцій, що, в свою чергу, піднімає питання щодо необхідності розробки асемблера під кожен новий за архітектурою пристрій. Це, з однієї сторони, робить проблему ще більш актуальною, а з іншої – значно ускладнює роботу з використанням мови асемблера. Що, відповідно, вимагає високої кваліфікації людини, котра працюватиме над вирішенням такого роду задач.

Тим не менше, основні підходи до побудови всіх видів трансляторів, в цілому, схожі. В даному контексті, асемблер необхідно розглядати як підвид транслятора типу компілятор.

Актуальність теми розробки як асемблерів, так і машинних трансляторів в цілому є вкрай високою, що позитивно позначається на перспективах розвитку інженерів у даній сфері. Цьому сприяють де-кілька факторів. По-перше, нові мови програмування як з'являлися, так і продовжать з'являтися, особливо, враховуючи попит на no-code та low-code рішення. По-друге, рівень знань та компетенцій сучасних розробників став набагато нижчим ніж 15–20 років тому,

а відповідно, попит на кваліфікованих інженерів у таких, не зовсім простих сферах, як розробка трансляторів, буде лише зростати. По-третє, виходячи з попереднього пункту, конструювання трансляторів є одним з найкращих способів знайти високооплачувану роботу в одній із топових компаній світу. Саме тому, вивчення теорії трансляції та принципів побудови мов програмування є досить перспективним напрямком росту спеціаліста як інженера.

Дослідження підходів до конструювання трансляторів на прикладі розробки власного асемблера під час виконання даної кваліфікаційної роботи повинне стати чудовою можливістю отримання досвіду розробок такого типу для подальшого його використання у більш прикладних з точки зору широкого ринку задачах.

Мета даної роботи – навчитися розробляти прості транслятори мов програмування шляхом проведення теоретичного дослідження та отримання практичного досвіду.

Об’єктом дослідження у даній кваліфікаційній роботі є машинні транслятори, зокрема, асемблери. При цьому, акцент зроблений саме на останніх як на найбільш простих для вивчення та наглядних за принципами роботи.

Предметом проведення дослідження є принципи функціонування та внутрішня структура трансляторів, а особливо, компіляторів, одним з підвидів яких є асемблери.

Завдання на кваліфікаційну роботу полягає у:

– аналізі предметної області об’єкта проектування та сучасного стану проблеми;

– дослідженні основних підходів та методів конструювання машинних трансляторів, зокрема, асемблерів;

– визначенні вимог, проектуванні, конструюванні та тестуванні програмного продукту.

РОЗДІЛ 1
АНАЛІЗ ПРИНЦИПІВ БУДОВИ АСЕМБЛЕРІВ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Природа асемблерів та сучасні тенденції

Асемблери представляють собою один із підвидів трансляторів, якими, в свою чергу, у термінах Computer Science називають спеціальні програми, що використовуються для перетворення програмного коду, написаного на, зазвичай, високорівневій мові програмування у машинний код, що зрозумілий процесору [1]. Справа в тім, що сама по собі обчислювальна машина (якою по суті є будь-який комп'ютер, хоч сам термін «обчислювальна машина» і звучить застаріло) не може розпізнавати та виконувати ніякі інструкції, окрім тих, що були закодовані у вигляді підтримуваних нею команд.

Сама по собі машина – логічний пристрій, що виражається не лише у «логічності» його роботи, а й у способі представлення у ньому будь-якої інформації. Так, якщо людина може сприймати інформацію різного роду за допомогою зору, нюху, слуху, відчуття смаку, вестибулярного апарату, тощо, то комп'ютер, у наших реаліях вміє працювати лише з однією властивістю – напругою електричного струму. Історія знає безліч спроб реалізації кодування інформації для електронних обчислювальних машин. Були спроби реалізації двійкового (розуміючого лише два стани напруги), трійкового, та інших видів процесорів. Проте, саме перший, двійковий, став найбільш популярним, і, за виключенням винаходів ентузіастів-експериментаторів, єдиним стандартом кодування інформації для комп'ютера. Із назви зрозуміло, що «розуміє» двійкова машина лише два стани – високий та низький рівні напруги. Відповідно, низький можна представити нулем, а високий – одиницею. Відношення типу: «є» – «немає», «високий» – «низький», і т.д., можна привести до вигляду «правда» – «брехня», де обидва стани є класичними значеннями в логіці. Звідси, і визначення комп'ютера як «логічної» машини.

У контексті даної роботи, занурюватися глибше немає сенсу, тому зачіпляти тему архітектури процесорів, передачі даних між пристроями комп’ютера, та інших подробиць роботи на рівні «заліза» ми не будемо. Відмітимо лише один вкрай важливий для нас факт: і команди, і дані для обчислювальних машин можна представити як послідовність нулів та одиничок.

Також, не заглиблюючись в принципи кодування символів, приймемо, що, код букви «F» у таблиці ASCII має вигляд «01000110». Як бачимо, єдине, з чим може працювати комп’ютер – це послідовність одиниць та нулів, відповідно до чого, можливість нативної реалізації десяткової системи числення зникає. Тут і з’являється принцип збереження та обробки всієї інформації у вигляді двійкової системи числення. Проте, оскільки записувати послідовності одиничок та нулів не зручно, зазвичай, працюючи на рівні машинної архітектури, люди використовують шістнадцяткову систему числення, що дозволяє значно скоротити записи. Тепер, букву «F» можна представити як «46». Кожне з таких дворозрядних шістнадцяткових чисел (від 00 до FF) можна представити максимум вісьмома двійковими розрядами, відповідно – бітами, що за розміром складає один байт.

Відштовхуючись він раніше описаної інформації, неважко зрозуміти, що за допомогою шістнадцяткових кодів можна легко та відносно елегантно описати будь-яку комп’ютерну програму, при чому, у зрозумілому для машини вигляді. Так, для прикладу, знаменита програма «Hello, World!» для процесора на архітектурі x86 під операційною системою MS DOS з використанням BIOS переривань виглядатиме як на лістингу 1.1 [2].

Лістинг 1.1 – «Hello, World!» на машинних кодах

```
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C 6F 2C 20 57 6F 72
6C 64 21
```

Кінець лістингу 1.1

Як бачимо із наведеного вище лістингу, код програми хоч і буде зрозумілим комп’ютеру, все ж абсолютно незрозумілий для людини. Ба більше того, оскільки все кодується числами, то ні границь між командами, ніж відмінностей між опкодом та даними, ні навіть повідомлення «Hello, World!» ми вирізнити простим поглядом не можемо. Проте, для простоти, на даному лістингу кожна команда позначена звичайним, або жирним шрифтом через одну, а рядок – виділений похилим.

Певний час люди так і писали програми: у вигляді двійкових, вісімкових, чи шістнадцяткових чисел. Тогочасним програмістам необхідно було або запам’ятовувати таблиці кодів, або постійно користуватися ними під час роботи. Зрозуміло, що це спричиняло величезну кількість проблем. По-перше, така розробка займала надзвичайно велику кількість часу, через явну незручність. По-друге, така «неконкретність», якщо можна так висловитися, коли команди та дані «викладалися» у вигляді звичайних чисел у рядок, не могла не спричиняти велику кількість помилок з переплутаними десь номером операції, чи порядком цифр в числі. Відповідно, це спричиняло третю проблему – вкрай важку відладку програмного забезпечення, коли знайти, де в коді помилка було надзвичайно затратно в плані часу та зусиль.

Звичайно, довго це тривати не могло. Через певний час з’явилися так звані асемблери та мова для них – мова асемблера. Асемблер – це свого роду транслятор, що перетворює код, написаний на мові асемблера в машинний код. В свою чергу, мова асемблера – це здебільшого система так званих «аліасів» (від лат. alias – інакше), тобто псевдонімів які використовуються для заміни машинних кодів на звичайні, зрозумілі людині слова. Окрім цього, асемблери зачасту підтримують додаткові розширення (зазвичай, у вигляді директив), що дозволяють ще ефективніше та простіше писати код. Так, приміром, Kick Assembler підтримує пряму вставку окремих байтів та цілих бінарних файлів, макроси, перелічуваний тип (enum), систему обробки помилок, різні кодування для текстових літералів, цикли та розгалуження, функції, структури, константи та ще безліч так званого «синтаксичного цукру» [3].

Тут зробимо невеличкий відступ та відмітимо, що під «синтаксичним цукром» зазвичай розуміють певні синтаксичні доповнення та розширення для мов програмування, використання яких не впливає на поведінку програми, проте дозволяє зробити використання даної мови більш зручним для програміста.

Повернемося до асемблерів. На лістингу 1.2 представлена вище описана програма «Hello, World!», але вже на мові асемблера. Відповідні команди також виділені через одну жирним, а повідомлення «Hello, World!» – похилим [2]. Цифри на початку рядків використовуються на даному лістингу для позначення умовних адресів пам’яті, по яких лежать дані кожного рядка програми та приведені тут для зручності. В реальних програмах вони не вказуються.

Лістинг 1.2 – «Hello, World!» на мові асемблера

```
XXXX:0100      mov   bx, 0111h
XXXX:0103      mov   cx, 000Dh
XXXX:0106      mov   ah, 0Eh
XXXX:0108      mov   al, [bx]
XXXX:010A      inc   bx
XXXX:010B      int   10h
XXXX:010D      loop  0108
XXXX:010F      int   20h
XXXX:0111 HW   db    'Hello, World!'
```

Кінець лістингу 1.2

Звичайно, як видно із лістингу, коду стало більше, проте читати і писати його набагато легше. Відразу помітні зрозумілі програмісту команди: «mov» – «move», тобто «перемістити», або, в даному контексті «помістити»; «inc» – «increment», тобто «інкрементувати»; «loop» – «цикл», і т.д.

Для повного розуміння, приведемо табличку відповідності асемблерного коду до машинного (табл. 1.1).

Як бачимо, відповідність прослідковується. Так, приміром, перша команда «mov bx, …» транслюється у «BB …», а «int 20h» у «CD 20». Необхідність запам’ятовувати опкод зникає.

Таблиця 1.1 – Відповідність асемблерного та машинного кодів

Асемблерний код	Машинний код
mov bx, 0111h	BB 11 01
mov cx, 000Dh	B9 0D 00
mov ah, 0Eh	B4 0E
mov al, [bx]	8A 07
inc bx	43
int 10h	CD 10
loop 0108	E2 F9
int 20h	CD 20
HW db 'Hello, World!'	*48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21*

Джерело: складено автором

У контексті асемблерів важливо відмітити, що не існує єдиного стандарту для мови асемблера, як то для високорівневих мов програмування. Різна апаратна архітектура (x86, ARM, PIC, та ін.) відрізняється своїм набором інструкцій, принципами роботи пам'яті, функціонуванням стеку, тощо. Окрім цього, на фінальну програму накладує відбиток і операційна система, під яку така програма розробляється.

Із вище сказаного слідує, що різні апаратні архітектури можуть мати абсолютно різну мову асемблера. При цьому, навіть, якщо використовується та ж мова, залежність від ОС не зникає, і програма, написана під Windows буде мати помітні відмінності від написаної під Linux. А, враховуючи, що єдиного стандарту не існує, була розроблена ще й величезна кількість асемблерів, що підтримують розробку під ту ж платформу та операційну систему, проте працюють з різним синтаксисом. Наприклад, NASM, MASM та GAS дозволяють писати програмне забезпечення під Windows, проте мають кардинально відмінний синтаксис, а також, підтримують різні розширення.

Тим не менше, варто відмітити, що у сучасному світі сформувалися два найчастіше використовуваних типи синтаксису для коду на мові асемблера:

– Intel Syntax;

– AT&T Syntax.

Розглянемо цей пункт детальніше. У таблиці 1.2 наведене порівняння деяких команд у обох синтаксисах.

Таблиця 1.2 – Порівняння Intel та AT&T синтаксису мови асемблера

Intel syntax	AT&T syntax
mov eax, 1	movl $1, %eax
mov ebx, 0FFh	movl $0xFF, %ebx
mov eax, [ecx]	movl (%ecx), %eax
mov eax, [ebx+3]	movl 3(%ebx), %eax
sub eax, [ebx+ecx*4h-20h]	subl -0x20(%ebx, %ecx, 0x4), %eax
mov al, bl	movb %bl, %al
; This is a comment	# This is a comment

Джерело: складено автором

З таблички вище бачимо основні відмінності:

1) суфікси, що описують розмір операндів. Так, Intel синтаксис не використовує спеціальних суфіксів, щоб позначити, над даними якого розміру буде здійснюватись операція, а опкод обирається асемблером, виходячи з вказаних операндів. В свою чергу, AT&T синтаксис передбачає явне вказання розміру. Так, приміром, суфікс «b» позначає «byte» (байт), «w» – «word» (слово, 2 байти), «l» – «long» (подвійне слово, 4 байти), а «q» – «quad word» («чотирикратне» слово, тобто 8 байтів);

2) порядок операндів. Intel синтаксис використовує порядок «destination, source», коли AT&T – «source, destination»;

3) префікси. У синтаксисі AT&T для назв регістрів використовується префікс «%», коли в Intel – ні. Щодо числових констант, то для immediate адресації перед числами в AT&T вказується «$», коли в Intel пишеться просто число. В той же час, для шістнадцяткових значень синтаксис Intel передбачає використання суфіксу «h», коли AT&T – префіксу «0x»;

4) для непрямої адресації у Intel використовуються квадратні дужки «[eax]», коли в AT&T – круглі «(eax)»;

5) непряма адресація зі зміщенням у синтаксисі Intel має вигляд: «[base+index*scale+disp]», а у AT&T: «disp(base, index, scale)»;

6) коментарі у синтаксисі Intel зазвичай починаються з крапки з комою « ; », а у AT&T – з решітки « # » [4, 5].

Звичайно, це не всі відмінності між двома варіантами синтаксису. Проте, по уже проаналізованому можна зробити висновок: синтаксис Intel є «чистішим» та «зручнішим» для читання, коли AT&T синтаксис – більш конкретизованим та точним.

Мова асемблера належить до так званої парадигми «неструктурного програмування». Це означає, що, на відміну від більшості звичних нам мов програмування, у мові асемблера відсутні синтаксичні конструкції, що дозволяють розбити програму на певні логічні «блоки», такі, як розгалуження, чи цикли. Весь функціонал даних алгоритмічних конструкцій реалізовується через «стрибки» по коду за допомогою команди безумовного переходу «jmp» («jump») та її умовних «сестер», таких як «je», «jne», «jb», і безлічі інших. У високорівневих мовах програмування аналогом цього може слугувати оператор goto. Звичайно, там він не рекомендується до використання, оскільки сильно ускладнює читання та підтримку коду, перетворюючи його у так званий «спагеті-код», тобто неструктурований та повний неявних і циклічних залежностей код [6]. Тим не менше, саме в такому ключі працює комп'ютер: послідовно виконуючи команди та «стрибаючи» з місця на місце по необхідності. А, оскільки, мова асемблера близька до апаратної частини, то і уникнути роботи в «термінах комп'ютера» не вийде.

Варто відзначити, що багато сучасних асемблерів підтримують певні «структурні» та, навіть, «процедурні» конструкції для мови асемблера. Наприклад, опис умов не за допомогою машинних інструкцій, а з використанням звичних нам «if … else …», такі ж високорівневі конструкції для циклів, тощо. MASM навіть має особливий синтаксис для процедур, що злегка нагадує процедурне програмування. Втім, такі розширення не є конструкціями самої мови асемблера, а представляють собою лише директиви асемблера, котрі той «під капотом» уже перетворює на машинні команди. Тим не менше, такий «синтаксичний цукор» робить досить складне програмування на мові асемблера трохи простішим.

Оскільки асемблери є підвидом компіляторів, то і розробка програм на мові асемблера тягне за собою як плюси, так і мінуси компільованих мов. Втім, через низькорівневість, вона має і свої унікальні переваги та недоліки.

До «плюсів» використання мови асемблера у процесі розробки програмного забезпечення можна віднести:

– можливість безпосередньої роботи на рівні апаратної частини, що дозволяє використовувати всі можливості «заліза», а також, користуватися деякими функціями, що на високому рівні не доступні;

– програми, написані на мові асемблера зазвичай мають набагато менший розмір, ніж розроблені на мовах високого рівня;

– можливість роботи напряму з «залізом» відкриває широкі можливості оптимізації ПЗ як по використанню ресурсів процесора, так і по пам’яті;

– код, написаний на мові асемблера може відносно просто викликатися із інших високорівневих мов. Відповідно, можна використовувати його для написання найбільш складних та ресурсоємних частин програми.

Тим не менше, розробка на мові асемблера має і свої, вкрай значні «мінуси»:

– низька переносимість зібраної програми. Як вже було сказано, різні апаратні та програмні платформи відрізняються за логікою своєї роботи. Тому, мова асемблера, як і інші компільовані мови, після збірки програми, не буде забезпечувати роботи на всіх процесорах та операційних системах;

– низька переносимість вихідного коду. Якщо у високорівневих мовах на рівні самої мови та стандартної бібліотеки передбачена повна, або ж, майже повна сумісність з усіма пристроями, під які існує компілятор, тобто, написана один раз програма може бути скомпільованою під різні платформи, то у випадку з мовою асемблера, значну частину коду доведеться переписати;

– низькорівневість мови асемблера передбачає високий рівень підготовки інженера. Це означає, що якщо для написання найпростіших програм, наприклад, на Python, розробнику необхідно знати лише саму мову та базові логічні принципи роботи імперативних мов програмування, то для розробки на

мові асемблера, девелоперу необхідні широкі знання в архітектурі комп’ютера, математиці, роботі з операційними системами, тощо;

– швидкість розробки на мові асемблера вкрай низька. Це зв’язано із її низькорівневістю. Програму на мові асемблера важко писати через велику кількість необхідних знань та багатослівність (вихідний код в 10, а то й 15 раз більший, ніж на високорівневих мовах). Окрім цього, код на мові асемблера важко відлагоджувати;

– програми, вручну написані на мові асемблера не завжди є більш оптимізованими. Зачасту, такий код виявляється не швидшим, а то й повільнішим, ніж згенерований автоматично високорівневим компілятором. В той же час, програма, написана на високорівневій мові є значно більш простою у написанні та підтримці, ніж розроблювана на мові асемблера.

В цілому, мова асемблера зайняла свою нішу, де вона є оптимальною та використовується здебільшого для:

– написання частин програми, що потребують високого рівня оптимізації;

– розробки програмного забезпечення під мікроконтролери (embedded-пристрої). При цьому, мова асемблера використовується зазвичай як допоміжна;

– отримання доступу до певних специфічних функцій процесора;

– написання низькорівневої «обв’язки» над машинними командами для hardware і подальшого використання її в якості API для викликів з високорівневих мов програмування;

– написання шкідливого програмного забезпечення. Цьому сприяють широкі можливості доступу до «заліза» та функцій операційної системи;

– так званої «ретро-розробки», тобто, розробки ПЗ під старі апаратні платформи;

– злому (хакінгу) програмного забезпечення з метою як отримання несанкціонованого доступу, так і для виявлення вразливостей системи.

1.2 Огляд базової архітектури асемблерів як підвиду компіляторів

Виділяють безліч видів машинних трансляторів. При цьому, усі вони засновані на двох базових принципах: компіляції та/або інтерпретації.

Компіляція працює за принципом перетворення коду, написаного на одній мові програмування у код на іншій. При цьому, зазвичай, код на високорівневій мові (C, C++, Rust, Pascal, і т.д.) транслюється компілятором у код на мові асемблера. В свою чергу, асемблер, як підвид компілятора, уже перетворює код, написаний на мові асемблера у машинні команди.

Тут важливо відмітити одну річ: результатом повної компіляції є уже абсолютно готовий до виконання машинний код. Звідси слідує, що трансляція компільованих мов програмування здійснюється зарання, у так званому, «compile-time», тобто, ще до запуску програми.

В свою чергу, інтерпретація передбачає, що вихідний код програми перетворюється у машинні команди, або ж виконується у середовищі інтерпретатора «на льоту», тобто, у так званому «runtime», або ж, іншими словами, прямо під час виконання програми.

Зазначимо наступне: інтерпретаторів у «чистому» вигляді майже не залишилося. Головна причина цього – низька швидкість роботи (розбору, аналізу та виконання) коду прямо у реальному часі. Тому, досить популярною сьогодні є концепція JIT-компіляції. Так, програма компілюється у проміжне представлення, після чого запускається на інтерпретацію, а тоді, якщо виконання натикається на надто повільну ділянку коду, відбувається докомпіляція даного сегменту уже напряму в машинний код.

Асемблери, як було сказано раніше, відносяться до підвиду компіляторів, а тому, будемо розглядати їх архітектуру саме на прикладі компіляторів в цілому.

Будь-який компілятор складається з де-кількох основних частин: лексера, парсера та генератора вихідного коду. Окрім цього, компілятори

високорівневих мов можуть містити і інші модулі. Зазвичай, в будові транслятора виділяють два «рівні»: front-end та back-end.

Front-end'ом називають частину компілятора, що займається конвертацією лексичного представлення програми (вихідного коду) в логічне. Ця частина дозволяє усунути всі синтаксичні особливості конкретної мови програмування та згенерувати її логічне представлення, зазвичай, у вигляді абстрактного синтаксичного дерева. Для високорівневих мов характерний конвеєр, що включає в себе:

– лексер. Це – компонент, що займається розбиттям тексту програми на синтаксичні токени. Ними можуть бути «числовий літерал», «строковий літерал», «коментар», «знак додавання», «ідентифікатор», «кома», тощо;

– парсер. Сюди потрапляють токени з лексера, після чого, відбувається відкидання непотрібних (наприклад, коментарів), а з тих, що мають семантичне значення будується абстрактне синтаксичне дерево. У випадку асемблера замість побудови дерева здійснюється аналіз та маппінг (відображення) асемблерних команд на машинні коди;

– семантичний аналізатор, що займається перевіркою правильності всіх операцій. Приміром, тут співставляються типи змінних із типами присвоюваних їм значень;

– IR-генератор. Це – генератор так званого «Intermediate Representation», тобто «проміжного представлення». Він конвертує AST в структуру, де прибрані всі не важливі для подальшої трансляції дані. Приміром, він може замінити назви локальних змінних на їх номери, тощо [7].

Наступний рівень інколи виділяють окремо, а інколи – включають до складу бек-енду. Це – етап, на якому здійснюється обробка та оптимізація побудованої моделі програми. Тут використовуються різного роду оптимізатори. Так, приміром, можуть «вирізатися» не використовувані змінні, а ті змінні, що не міняються перетворюватись у константи; прості функції вбудовуються прямо в код, щоб уникнути навантажень, пов'язаних з їх викликом, тощо. Окрім цього, оптимізація може включати в себе «розгортання»

циклів, коли за одну ітерацію будуть виконуватися дії, що раніше потребували де-кілька ітерацій. Також, якщо апаратна платформа дозволяє, використовується «векторизація», тобто обробка однією командою де-кількох значень за раз (SIMD) [8].

Далі, оптимізоване представлення програми потрапляє до бек-енду. Він займається тим, що перетворює логічне представлення в машинне, тобто транслює його в машинні коди. При цьому, у випадку з високорівневими мовами програмування спочатку здійснюється «переклад» на мову асемблера, а далі, уже викликається асемблер, що і займається фінальною «бінаризацією». В сучасних реаліях, у компіляторах тут виділяють два етапи:

– генерація коду. На цьому етапі у випадку високорівневих мов відбувається трансляція у мову асемблера, після чого викликається сам асемблер. В свою чергу, у асемблера цей етап пов’язаний з перетворенням асемблерних команд у машинний код;

– компоновка, або ж як її ще називають, лінковка (від англ. «linking» – «посилання», чи, в даному контексті, «зв’язування») – це процес перетворення двійкових об’єктних файлів у виконуваний файл. Справа в тім, що рідко коли реальна програма займає всього один файл вихідного коду. Значно частіше, це – десятки, чи, навіть, сотні файлів. Кожен з них компілюється окремо у так званий «об’єктний файл», а вже потім вони збираються у один виконуваний (або, приміром, у бібліотеку, якщо ми таку розробляємо). Окрім цього, сучасні комп’ютери (зачасту, навіть вбудовувані системи) мають між рівнем «заліза» та програмним забезпеченням прошарок у вигляді операційної системи. Вона, в свою чергу, також накладує свої вимоги на кожну програму, а тому – для роботи стороннього ПЗ необхідно «подружити», або ж, в нашому випадку, «зв’язати» програму із API операційної системи. Лінковка (як і бінаризація) також виконується окремою програмою, яка називається лінкер. Результатом її роботи є готовий до запуску виконуваний файл [7].

Така складна система використовується у, здебільшого, трансляторах високорівневих мов програмування. Для асемблерів же пайплайн трансляції

куди простіше – лексичний аналіз, семантичний аналіз та конвертація в машинні коди.

1.3 Постановка завдання на кваліфікаційну роботу бакалавра

Завданням на кваліфікаційну роботу є розробка власного асемблера. Проте, звичайно, проведення розробки ПЗ не можливо в один етап, без деталізації конкретного плану. У цьому розділі вже була проведена підготовча робота – досліджені принципи будови та функціонування асемблерів. В пункті 1.2 були проаналізовані основні архітектурні та, зокрема, компонентні рішення для розробки асемблерів. Тепер, визначимо роботи, котрі необхідно провести:

- вибір архітектури, під набір команд якої розроблятиметься асемблер;
- вибір інструментів та підходів до конструювання продукту;
- проектування архітектури майбутнього асемблера;
- програмна реалізація асемблера згідно обраного підходу;
- розробка тестів та тестування роботи програми;

Висновки до розділу 1

Сьогодні асемблери представляють собою складне та комплексне ПЗ, що включає безліч як основних функцій, так і так званого «синтаксичного цукру».

Асемблери є окремим підвидом компіляторів. Незважаючи на те, що їх функції та структура значно простіші, ніж у трансляторів високорівневих мов програмування, все ж, в плані архітектури вони не сильно відрізняються від компіляторів в цілому. Зачасту, асемблери все ж досить комплексні та включають в себе багато різноманітних модулів, компонентів, і т.д., що дозволяє забезпечити підтримку великої кількості мов асемблера та генерацію коду для безлічі апаратних платформ.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз вимог до розроблюваної системи

Після попереднього визначення теми кваліфікаційної роботи постало питання щодо вибору типу транслятора, котрий можна буде реалізувати. По проведенню аналізу було прийнято рішення, що найбільш оптимальною буде розробка асемблера. У першу чергу, це зв’язано з тим, що асемблери є найбільш простим типом трансляторів. До такого вибору мене підштовхнули:

– відсутність попереднього досвіду розробки трансляторів;

– простота внутрішньої структури асемблера, що складається всього лише з лексера, парсера та генератора коду. Завдяки цьому, на відміну від трансляторів високорівневих мов, не буде необхідності вибудовувати певну логічно-семантичну структуру у вигляді абстрактного синтаксичного дерева;

– наявність схожих асемблерів, що дозволяє звірити правильність результату роботи продукту, не запускаючи згенерований код на реальному процесорі.

Після того, як був визначений тип розроблюваного транслятора, прийшла черга обирати набір команд і архітектуру, під яку даний асемблер буде збирати програму. Окрім цього, необхідно було обрати синтаксис для мови асемблера.

Відразу було прийнято рішення щодо розробки асемблера під набір команд восьмибітного процесора. У першу чергу, це було пов’язано з тим, що більш складні архітектури мають набагато комплекснішу структуру машинних команд. Так, більшість сучасних процесорів, що використовуються у персональних комп’ютерах, базуються на архітектурі x86. Дана архітектура має вкрай неоднорідну структуру та розмір інструкцій. Приміром, різні інструкції, окрім опкоду (що може займати від 1 до 3 байтів) можуть містити ще й префікс, опис режиму адресації, зміщення, безпосереднє (immediate) значення, тощо. На рисунку 2.1 зображено схему, за якою будуються інструкції даної апаратної архітектури.

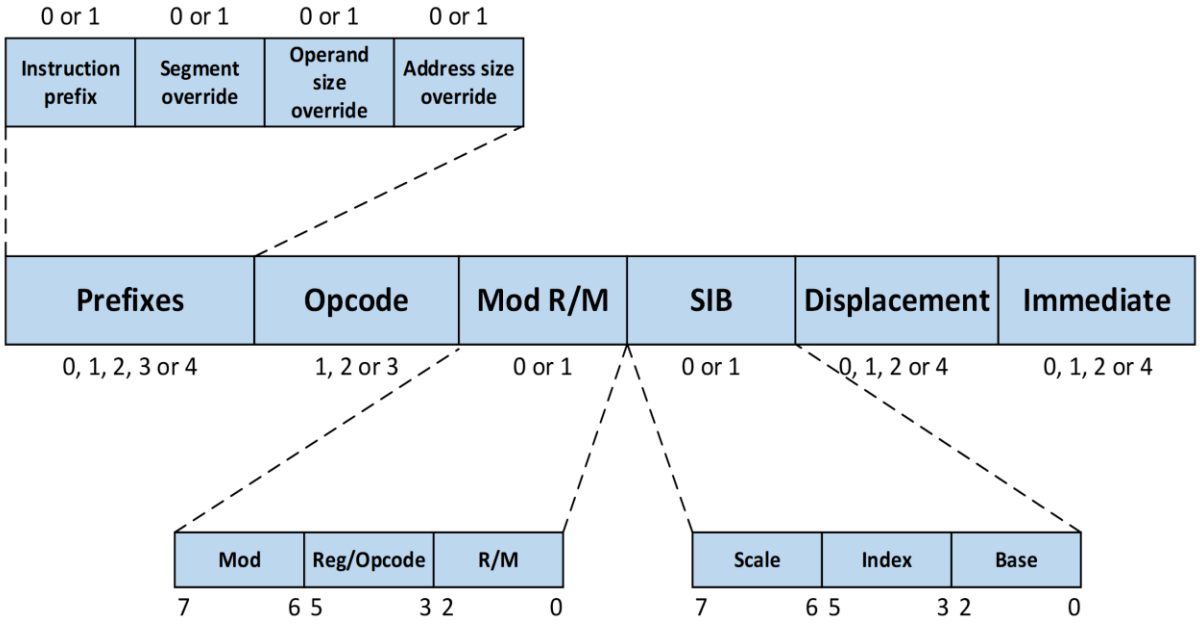

Рисунок 2.1 – Структура інструкцій архітектури x86

Таким чином, розмір інструкції може становити від 1 до 15 байтів [9]. В цей же час, інструкції на восьмибітній архітектурі зазвичай займають один, або два (рідше – три) байти, і містять опкод та один, максимум, два операнди.

Для більшості восьмибітних процесорів синтаксис мови асемблера визначений виробником, тому реалізовуватимемо ми його у відповідності до офіційних специфікацій. Саме тому, вибір типу синтаксису між Intel та AT&T не має ніякого сенсу.

В результаті аналізу наявної інформації про різні восьмибітні процесори було прийнято рішення щодо реалізації асемблера для набору команд процесора MOS Technology 6502. Цьому посприяли наступні причини:

– висока популярність процесора в минулому;

– велика кількість наявної документації;

– відносна простота архітектури процесора;

– даний процесор та його модифікації широко використовувались у «прикладних» пристроях, як от: Atari 2600, NES, Commodore 64, Apple II, тощо. Це, в свою чергу, дозволить застосовувати розроблюваний асемблер для ретро-розробки, тобто розробки під старі пристрої.

Після конкретизації об’єкту розробки визначимо основні вимоги до фінального продукту, котрі необхідно буде реалізувати. Для початку, визначимо функціональні вимоги:

– робота з набором інструкцій MOS Technology 6502. Асемблер повинен розпізнавати синтаксис мови асемблера, а також, мати можливість збирати вихідний код у бінарні виконувані файли з відповідним набором команд;

– розпізнавання файлів вихідного коду з розширенням «.asm»;

– результатом асемблювання повинен бути вихідний файл у бінарному форматі з розширенням «.bin»;

– наявність «синтаксичного цукру». У даному випадку, наявність певних директив асемблера та спрощень, що дозволять зручніше писати код. Наприклад, команди для вставки «сирих» даних та підтримка синтаксису визначення рядків у форматі ASCII.

Наступними, необхідно визначити нефункціональні вимоги до розроблюваного асемблера:

– здатність продукту до розширення підтримуваного набору команд. Наприклад, у подальшому, асемблер може бути розширеним підтримкою інструкцій процесора Western Design Center 65C02, та інших модифікацій звичайного MOS Technology 6502. Окрім цього, необхідна можливість додавати підтримку інших процесорів;

– здатність до розширення функціоналу. Наприклад, додання нових директив, синтаксичних спрощень, елементів структурного програмування, тощо.

2.2 Огляд архітектури процесора MOS Technology 6502

Перед проектуванням власного асемблера, необхідно розглянути основи архітектури процесора MOS Technology 6502, оскільки написання програм на низькорівневій мові, такій, як мова асемблера напряму пов’язане із маніпуляціями з апаратною частиною «заліза». В даному контексті, нас

цікавлять лише ті характеристики, котрі напряму впливають на написання програмного коду.

Основними особливостями процесора MOS Technology 6502, що важливі саме для програміста, є:

– паралельна обробка восьми біт даних;
– 56 інструкцій;
– 13 режимів адресації;
– десяткова та двійкова арифметика;
– підтримка реальної індексації;
– підтримка прямого доступу до пам’яті;
– підтримка немаскованих переривань;
– програмований Stack Pointer та стек змінної довжини;
– можливість адресації до 65K пам’яті [10].

Далі, для нашої роботи необхідно розглянути регістри процесора та набір інструкцій. Почнемо з регістрів (рис. 2.2).

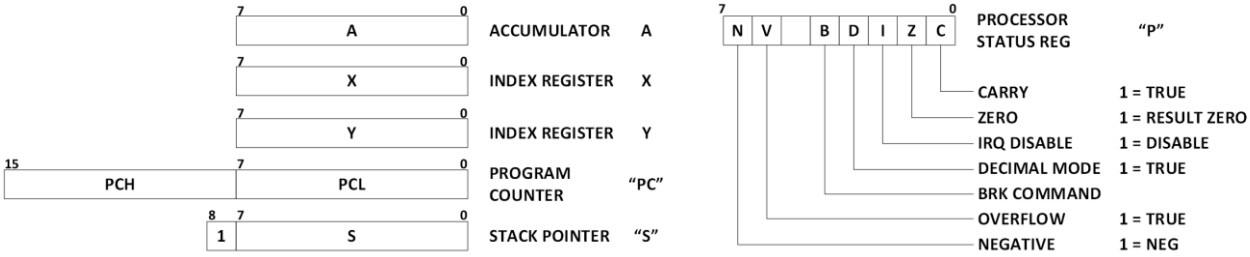

Рисунок 2.2 – Регістри процесора MOS Technology 6502

Розберемо детальніше кожен з вище означених регістрів, так як вони напряму використовуються при написанні програм на мові асемблера:

– регістр A. Accumulator. Восьмибітний регістр, що використовується як основний для більшості інструкцій. Де-які з арифметичних операцій доступні лише для використання з цим регістром;

– регістри X та Y. Index Registers. Два восьмибітних регістри індексації, що зазвичай використовуються для адресації та вказання зміщення в пам’яті;

– регістр PC. Program Counter. Єдиний шістнадцятибітний регістр процесора. Використовується для вказання повної адреси поточної команди. Складається з двох восьмибітних частин: PCH (Program Counter High) та PCL (Program Counter Low). Вони зберігають, як зрозуміло з назви, нижчу та вищу частини адреси. Завдяки 16-бітній розрядності цього регістра, процесор може адресувати 2^{16}, тобто, 65.536 байт, або ж, 65 кібібайт;

– регістр S. Stack Pointer. Восьмибітний регістр-вказівник стеку. Як зрозуміло з його розміру, апаратний стек на даному процесорі займає 2^8, тобто 256 байтів. При цьому, як видно зі схеми на рисунку 2.2, ще один, восьмий біт, автоматично встановлений в одиницю, що означає, що стек «лежить» на першій сторінці пам’яті. Іншими словами, він займає адресний простір $0100 – $01FF;

– регістр P. Processor Status Register. Це – так званий «статусний» регістр, що не використовується для маніпулювання даними, а відображає статус процесора та результати виконання певних операцій. Наприклад, якщо результатом якоїсь операції буде нуль, то відповідний прапорець «Z», тобто, «Zero» стане True. Даний регістр складається із «прапорців», або «флажків» (від англ. «flags»), кожен з яких займає один біт та, відповідно, відображає певний результат роботи процесора, чи виконання операції [11].

Наведемо детальний опис вище згаданих прапорців регістра Processor Status (за порядком в регістрі):

1) C. Carry. Прапорець переносу. Встановлюється в True, якщо результат операції перевищує місткість регістра (256 та більше);

2) Z. Zero. Прапорець «нуль». Встановлюється в True, якщо результатом операції є нуль;

3) I. IRQ Disable. Прапорець заборони зовнішніх переривань. Якщо встановлений в True, то зовнішні переривання заборонені;

4) D. Decimal Mode. Режим десяткової арифметики. Використовується досить рідко, проте дозволяє вказувати числа як десяткові та працювати з ними в такому ж ключі. Підтримує числа лише від 0 до 99 ($0 – $99);

5) B. BRK Command. Прапорець обробки переривань. Стає True, коли виконується команда BRK;

6) Цей прапорець не використовується та завжди дорівнює True;

7) V. Overflow. Прапорець переповнення. Стає True, якщо при операції відбувається арифметичне переповнення (overflow чи underflow);

8) N. Negative. Прапорець «від'ємності». Стає True, якщо результат виконання операції імовірно є від'ємним числом (тобто, коли старший біт результату дорівнює одиниці) [11].

Розглянути дані регістри було вкрай важливо, оскільки мова асемблера процесора MOS Technology 6502 дозволяє працювати як з основними регістрами, так і з окремими прапорцями.

Далі, скажемо де-кілька слів про роботу даного процесора із пам'яттю. По-перше, весь адресний простір процесора розбитий на так звані «сторінки» по 256 байт. При цьому, PCH адресує номер сторінки, а PCL – номер байту на сторінці. Це дуже важливо через дві причини:

– першу сторінку (тобто ту, де старша половина адреси є одиницею), як вже було сказано, займає стек, тому писати туди щось вручну не варто;

– нульова сторінка (так звана Zero Page), де старша частина кожної адреси дорівнює нулю має свої особливості. Так, по-перше, вона значно швидша для доступу, ніж всі інші сторінки пам'яті. Відповідно, тут можна зберігати дані, до яких потрібен частий доступ. По-друге, Zero Page має свої власні режими адресації, що дозволяють вказувати лише молодшу частину адреси.

Окрім цього, важливо зазначити, що процесор MOS Technology 6502 побудований на так званій Little Endian архітектурі. Це означає, що значення, які займають два та більше байтів (наприклад, абсолютна адреса) у пам'яті зберігаються в оберненому стані. Так, наприклад, якщо при прямому (Big Endian) порядку, число 0x12AB34CD зберігається в пам'яті як «12 AB 34 CD» (в порядку зростання адрес), то при Little Endian – як «CD 34 AB 12» [12].

Далі розглянемо систему інструкцій процесора MOS Technology 6502, оскільки наш асемблер і буде займатися перетворенням коду на мові асемблера у машинні команди.

Всього, набір інструкцій (Instruction Set) даного процесора містить 151 інструкцію. З них, «оригінальних» 56, а інші – їх різновиди з різними режимами адресації. Перелік усіх інструкцій з їх назвами, режимами адресації та опкодами знаходиться у додатку А, таблиці А.1. Кожна з таких інструкцій може мати розмір в 1, 2, чи 3 байти та складатися з обов'язкового однобайтного опкоду та опціонального одно- чи дво-байтного аргументу (рис. 2.3). Детальний опис кожної асемблерної мнемоніки знаходиться у додатку Б, таблиці Б.1.

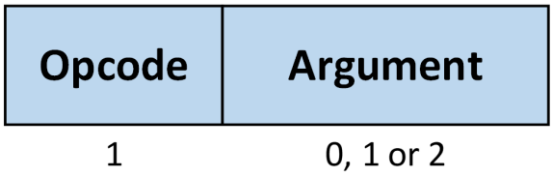

Рисунок 2.3 – Структура інструкції архітектури 6502

У кінці, розглянемо режими адресації процесора MOS Technology 6502. Це особливо важливо в контексті розробки асемблера, оскільки формат асемблерної інструкції напряму залежить від того, який режим адресації має та чи інша команда.

Як вже було сказано, процесор MOS Technology 6502 підтримує 13 режимів адресації [13]. У додатку В, таблиці В.1 наведений список цих режимів, їх сигнатури та детальний опис.

2.3 Проектування архітектури асемблера

Тепер, коли ми визначили вимоги до розроблюваного асемблера та розглянули архітектурні особливості процесора MOS Technology 6502, які, безперечно, накладають відбиток і на мову асемблера, котру ми збираємось

транслювати, пришла пора провести базове проектування майбутнього продукту.

Оскільки асемблер займається перетворенням текстової інформації (програмного коду на мові асемблера) у машинний код, для початку, визначимо основні синтаксичні конструкції, котрі повинен підтримувати наш асемблер. Виходячи з визначених вимог, поділимо усі конструкції на 3 групи:

– базові інструкції процесора. Ця група включає команди, котрі містять опкоди та аргументи, що напряму перетворюються у інструкції архітектури 6502. Тут міститься 151 інструкція, що описана у додатку А, таблиці А.1;

– макроси. Вони будуть представлені лише одною директивою «.define». Макроси дозволятимуть підставляти в певні місця коду одно-, чи дво- байтні числа. Це зручно для маніпуляцій з адресами, чи константами задля уникнення так званих «магічних чисел»;

– директиви асемблера. Це – спеціальні команди, що не відображаються напряму на інструкції процесора, а використовуються для налаштувань, чи додання сторонніх функцій, що виконуватимуться самим асемблером.

Директиви можна віднести до «синтаксичного цукру», який у нашому асемблері буде представлений:

– директивою «.byte», що дозволятиме вставити в код на поточну позицію один байт будь-якої інформації;

– директивою «.word», яка працюватиме так же, як і byte, проте вставлятиме уже машинне слово (два байти);

– директивою « * », котра позначатиме code position, тобто адресу, за якою буде вставлятися весь наступний код;

– спеціальним синтаксисом для рядків, щоб не вказувати їх за номерами символів. Ми використовуватимемо стандартне кодування ASCII. Для вставки рядка достатньо буде вказати команду byte, після чого у подвійних лапках вписати рядок.

Окрім вище сказаного, наш асемблер підтримуватиме синтаксис для коментарів, що розпочинатимуться з символу « ; ».

Далі, розглянемо пайплайн роботи асемблера. Спочатку, файл з розширенням «.asm» передаватиметься в якості вхідного параметра до асемблера. Там він відкриватиметься, з нього зчитуватиметься весь текст, котрий далі потраплятиме до лексера.

У лексичному аналізаторі відбуватиметься розбиття даного тексту на лексичні токени, визначені синтаксисом мови асемблера.

Після цього, дані токени потраплятимуть до парсера, що аналізуватиме їх, перевірятиме семантичну коректність та перетворюватиме ці токени у «serializable» токени. Їх особливість – вони вже описуватимуть реальні команди процесора, чи асемблера, але в об’єктному вигляді. Так, приміром, лексичний токен інструкції «lda» об’єднуватиметься зі своїм значенням «$4», в окремий об’єкт, котрий потім можна буде автоматично конвертувати в машинну інструкцію. Окрім цього, на початку даного етапу видалятимуться коментарі та підставлятимуться макроси.

Далі «серіалізовувані» токени потраплятимуть до генератора коду, котрий вже буде транслювати їх в бінарний вигляд.

У кінці, програма, що вже представлена машинними кодами буде зберігатися у файл з розширенням «.bin».

На рисунку 2.4 зображена діаграма комунікації розроблюваного асемблера. На ній вказані основні структурні модулі нашого продукту та вхідні і вихідні файли. Окрім цього, на даній діаграмі також відображені пересилки повідомлень між вище згаданими модулями, що мають вигляд токенів чи вихідного коду.

Після того, як ми визначили основні компоненти та те, як вони будуть взаємодіяти (тобто, поведінку системи), прийшла черга провести структурне проектування продукту. Тут, ми за допомогою діаграми класів (оскільки розробляємо програмне забезпечення в ООП-стилі) відобразимо структуру та ієрархію різних модулів асемблера.

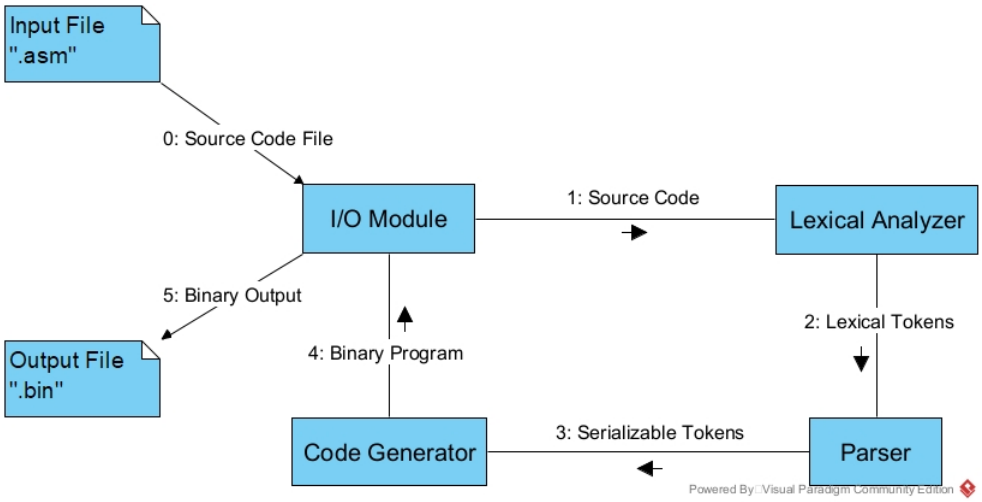

Рисунок 2.4 – Communication Diagram розроблюваного асемблера

Необхідно відзначити, що, звичайно, ми проведемо лише базове структурне моделювання асемблера. Ми відмітимо лише найнеобхідніші для розуміння структури продукту частини, оскільки до початку роботи ми аж ніяк не можемо знати точно, які додаткові компоненти нам знадобляться та які архітектурні рішення доведеться прийняти. Саме тому, розробка UML діаграми класів перед початком роботи над ПЗ завжди здійснюється відносно поверхнево. Звідси – важливо розуміти, що така схема не фінальна, і може, а скоріше, навіть, буде неодноразово змінюватися в процесі розробки. Результат ж нашого проектування зобразимо на рисунку 2.5.

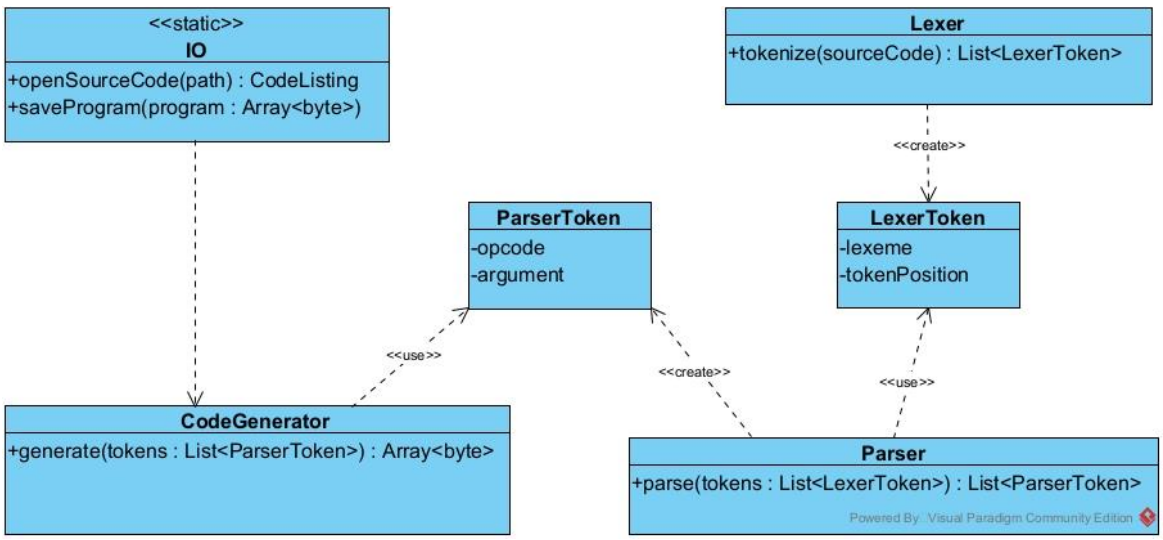

Рисунок 2.5 – Діаграма класів розроблюваного асемблера

Як видно з рисунку 2.5, основними компонентами нашої системи будуть:

– модуль вводу-виводу, що відповідатиме за відкриття коду на мові асемблера та збереження вже засембльованої програми;

– лексер, котрий прийматиме код програми та перетворюватиме його на лексичні токени;

– парсер, що перетворюватиме лексичні токени на токени, придатні до бінаризації;

– генератор коду, котрий генеруватиме машинний код із інформації, отриманої від парсера;

– токени лексера та парсера, що являтимуть собою, свого роду, DTO (data transfer object), тобто об’єкти, ціль яких – передача даних між модулями системи. При цьому, такі об’єкти зазвичай не мають власної поведінки, а виконують лише функцію «трансферу».

З іншими, додатковими компонентами системи та подробицями реалізації її окремих частин ми вже розберемося по ходу виконання практичної частини конструювання асемблера.

2.4 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Після проектування базової структури програмного забезпечення необхідно провести аналіз та вибір технологій та методів, що будуть використовуватись для реалізації поставленого завдання.

У процесі розробки ПЗ використовується безліч різних засобів та інструментів. У нашому випадку, для розробки асемблера нам знадобляться:

– мова програмування та транслятор до неї;

– інтегроване середовище розробки (IDE);

– система зборки проекту;

– шістнадцятковий редактор для перевірки результату роботи продукту;

– інструменти для тестування (приміром, тестовий фреймворк);

– сторонній асемблер, що підтримує набір інструкцій 6502 для звірки коректності роботи нашого продукту.

Сьогодні, де-факто стандартом для написання трансляторів мов програмування є мови C та C++. Вони використовуються найчастіше, коли справа доходить до написання системного програмного забезпечення, чи ПЗ, що потребує високого рівня оптимізації. Ці дві мови досить близькі: мова C++ була розроблена Б’ярном Страуструпом як об’єктно-орієнтована надбудова над C, та спочатку іменувалася просто як «C з класами». І хоча в подальшому шляхи розвитку цих мов досить помітно розійшлися, все ж, в ядрі своєму вони дуже схожі. Обидві вони є чисто компільованими, що накладає на них як всі плюси, так і мінуси компільованих мов. Головна їх перевага – можливість бути вкрай оптимізованими, що, в свою чергу, дозволяє максимально зменшити розмір виконуваного файлу та прискорити виконання програми. Саме тому, C та C++ є майже єдиними мовами для написання програмного забезпечення для embedded-пристроїв. Хоча, я б виділив іншу, суб’єктивно, дуже важливу перевагу – «явність» цих мов. Не зважаючи на те, що в C++ з’явилася підтримка ООП, namespace’ів, і т.д., обидві мови зберігають вкрай прямолінійний синтаксис. Так, у більшості сучасних мов програмування примітивні типи (int, char, bool, і т.д.) передаються за-замовчуванням по значенню, а складні (приміром, класи) – через посилання, хоча синтаксис ніяк цього не підкреслює. В свою чергу, C та C++ за замовчуванням передають все по значенню. А передача за посиланням здійснюється через явне вказання типу «pointer», чи «reference».

Для використання у нашому проекті ми оберемо мову C++. У першу чергу, це зумовлено описаними вище перевагами, а також, можливістю нативно працювати на низькому рівні (те, що називають «робота з байтами»). Ми віддамо перевагу цій мові, а не чистій C, оскільки C++ має підтримку ООП, що значно полегшить нам роботу.

У якості транслятора оберемо компілятор gcc. Він є одним з найбільш популярних сьогодні, наряду з clang та msvc. Вибір впав саме на нього,

оскільки цей компілятор підтримує найновіші стандарти та найбільшу кількість додаткових розширень [14].

У якості інтегрованого середовища розробки використаємо CLion. Це – одна з найкращих IDE для C++, а також, що досить важливо, вона нативно працює з обраним нами компілятором gcc. В якості системи збірки проекту тут за замовчуванням використовується CMake, тому для нашого проекту ми будемо використовувати саме його.

В якості шістнадцяткового редактора нам підійде будь-який hex-редактор, головне, щоб він мав можливість не тільки переглядати код програми, а й редагувати його.

Для тестування вихідного результату використаємо мову Python, що дозволить нам досить швидко написати прості скрипти для звірки отримуваного та очікуваного результатів роботи нашого асемблера.

Окрім цього, нам знадобиться сторонній асемблер, котрий ми зможемо використати для перевірки правильності роботи нашої розробки. Після пошуку аналогів, досить простим та ефективним для даної задачі виявився асемблер із симулятора восьмибітного комп’ютера з набором інструкцій 6502 «6502js». Використовуватимемо його.

Висновки до розділу 2

Перед початком розробки власного асемблера для симулятора восьмибітного процесора був проведений детальний аналіз вимог до розроблюваної системи. Там було прийнято рішення щодо розробки асемблера для процесора з набором інструкцій MOS Technology 6502. Після цього, були визначені конкретні вимоги до цього програмного забезпечення.

Наступним етапом було проведене дослідження архітектури процесора MOS Technology 6502 з перспективи програміста. Були розглянуті регістри, особливості роботи процесора з пам’яттю та режими адресації.

Після цього, було проведене проектування базової структури розроблюваного асемблера. Були визначені основні структурні компоненти ПЗ, а також, зв’язки між ними. Окрім цього, були описані де-які семантичні особливості мови асемблера, як-от види логічних конструкцій.

У кінці був проведений аналіз засобів та інструментів, що можуть бути використані для виконання поставленого завдання. Було прийнято рішення щодо використання для розробки асемблера мови програмування C++, а в якості інструменту для тестування – мови Python.

РОЗДІЛ 3
РОЗРОБКА АСЕМБЛЕРА ДЛЯ СИМУЛЯТОРА ВОСЬМИБІТНОГО ПРОЦЕСОРА

3.1 Практична реалізація об’єкта проектування

Практичну реалізацію асемблера прочнемо із аналізу синтаксису мови асемблера, котру збираємось транслювати. Для цього, розберемо основні синтаксичні конструкції.

Коментарі. Вони починаються із символу « ; » та являють собою текст, що використовується для коментування функціоналу чи структури певних частин коду. Коментарі не впливають на роботу програми, оскільки «вирізаються» ще до початку семантичного розбору інструкцій. При цьому, якщо в коді зустрічається символ « ; », то він робить коментарем все, що слідує за ним аж до кінця рядка.

Директиви асемблера. Це – інструкції для компілятора, що дозволяють маніпулювати даними, не пов’язаними напряму із машинними командами. Як вже було сказано, у нас буде де-кілька таких директив. Їх особливість – всі вони, за виключенням code position, починаються з крапки.

Директива code position має наступну структуру: спочатку йде символ asterisk « * », після нього – знак «дорівнює» « = », а далі – число, що позначає адресу в пам’яті, куди будуть записані наступні команди («*=$2000»).

Інші директиви, в свою чергу, мають іншу форму: крапка, за якою відразу йде назва директиви, приміром «.byte», після чого йдуть параметри. У випадку з директивою «.define», далі йде ідентифікатор, а після нього – число, що позначає значення, що буде підставленим скрізь, де «викликається» даний макрос (.define value $4). Директиви «.byte» та «.word» після себе приймають число, котре буде записане в код програми якраз в те місце, де ця директива знаходиться. При цьому, перша працює з однобайтними числами, а друга – з двохбайтними. Окрім цього, дані директиви можуть приймати список чисел, розділених комою, що дозволить вставити відразу де-кілька значень. Також,

вони можуть приймати рядок, що виділений символами « “ ». В такому випадку, у програмі кожен символ буде представлений своїм кодом в форматі ASCII, і розміром, відповідно – один, чи два байти.

Наступними йдуть безпосередньо інструкції для процесора. Вони, в залежності від режиму адресації, можуть мати різний вигляд (див. дод. В, табл. В.1). В цілому, тут використовуються:

– латинські букви для назв інструкцій та міток. При цьому, наш асемблер є регістронезалежним, тобто, для нього велика та маленька букви означають те ж саме. Окрім цього, в назвах міток можуть використовуватися цифри та нижні підкреслення « _ ». Цифра не може бути першим символом в назві мітки;

– числа можуть вказуватись у двійковій, вісімковій, десятковій та шістнадцятковій системах числення. При цьому, якщо число починається з «натуральної» цифри, то це – запис в десятковій системі. Якщо число починається з нуля, то це – вісімкова. В свою чергу, частіше всього, числа записуються у шістнадцятковій системи, де вони починаються зі знаку « $ ». Для вказання числа у двійковому вигляді використовується знак відсотка « % »;

– параметри операції розділяються за допомогою коми. При цьому, параметрами (в залежності від операції) можуть бути числа, мітки, макроси, чи назви регістрів;

– для режимів з непрямою адресацією використовуються круглі дужки « ( ) »;

– для immediate адресації перед значенням ставиться знак hash « # ». При цьому, для immediate завантаження в регістр старшої чи молодшої частини адреси використовуються оператори « #< » та « #> »;

– для позначення міток використовується ідентифікатор, після якого йде знак двокрапка « : ».

Вище означені аспекти було вкрай важливо розібрати, оскільки саме вони визначатимуть правила, за якими працюватиме лексер, тобто, лексичний аналізатор. Тепер визначимо основні токени, на які лексер розбиратиме вихідний код програм на мові асемблера:

– коментар (все що починається з « ; »);

– директива (те, що починається з крапки, або зірочка);

– текстовий рядок (все, що розміщено між скобками « “ »);

– чисельні константи. При цьому, тут ми розділимо їх на 4 види: двійкові, вісімкові, десяткові та шістнадцяткові. Визначатися тип константи буде за раніше описаними правилами;

– ідентифікатор. Сюди входитимуть як назви інструкцій, так і макроси та «виклики» міток. Іншими словами, все, що складається з букв, цифр, та нижнього підкреслення « _ »;

– атомарні константи. У цю групу ми включимо де-кілька типів токенів: кома, решітка, ліва дужка, права дужка, знак «менше», знак «більше», зірочка та знак «дорівнює».

Почнемо програмну реалізацію лексичного аналізатора. За основу візьмемо простий лексер, написаний для C++ та перепишемо його логіку згідно наших правил [15]. Він буде приймати на вхід програмний код, розбитий по рядках, після чого, оброблятиме його та повертатиме набір токенів, кожен з яких буде містити свій тип, лексему, а також, позицію в коді (рядок та стовпчик) для відображення повідомлень про помилки. Алгоритм роботи даного лексера буде наступним:

1) зчитати символ;

2) перевірити, чи є символ «skippable», якщо так, то пропустити його та перевірити наступний;

3) якщо символ розпізнається як значущий, викликати відповідну функцію (залежно від потенційного типу токена), котра сама зчитає наступні дані, допоки не визначить його лексему;

4) якщо ж дані не валідні, що виявилося до, чи після виклику функції, повернути тип токена «unexpected»;

5) продовжити зчитування, аж допоки не закінчиться вихідний код;

6) якщо всі дані були перетворені в валідні токени, то повернути список токенів, інакше – викинути помилку.

Далі токени з лексера потрапляють до парсера. Оскільки вони переміщуються між компонентами системи у вигляді простого списку, то у парсері, задля проведення семантичного аналізу, їх варто «упакувати» у початкові рядки – так, щоб кожна інструкція та її параметри знаходилися разом. Також, на цьому етапі вирізаються коментарі та відбувається підстановка макросів. Окрім цього, тут проводиться і додаткова обробка, як-от заміна міток на їх номери.

Після цього, відбувається семантичний аналіз коду. Здійснюється парсинг команд та аргументів, після чого, проводиться перевірка на коректність (відповідність інструкцій та їх аргументів), а далі – об’єднання їх в окремі об’єкти, що пізніше потраплять до генератора коду. Також, на цьому етапі окремо виділяються інструкції, що потребують «відносну адресу», або ж, зміщення (команди типу branch). Для таких операцій фінальне значення параметру буде розраховуватися уже в генераторі коду.

Після того, як парсер завершує роботу та повертає список своїх токенів, вони потрапляють у модуль генерації коду. Тут відбувається «бінаризація» команд у масив байтів. При цьому, враховуються як режими адресації, так і параметри кожної інструкції. Це відбувається у перший «прохід». Особливі інструкції, що оперують з відносною адресою (вираженою мітками), а також, ті, для роботи яких необхідно знати зміщення відносно даної інструкції, заносяться у окремий список. При цьому, опкод вноситься до лістингу програми, а, оскільки, ми знаємо режим адресації, то можемо залишити вільне місце під аргумент. Під час наступних «проходів» відбувається заповнення заздалегідь пропущених місць розрахованими адресами.

У кінці, кодогенератор повертає масив байтів, що являє собою вже готову програму. Далі, вона зберігається у форматі «.bin».

Тут важливо відмітити одну особливість: обробка помилок у нашому асемблері здійснюється не шляхом «fail fast», тобто раптовим викиданням виключення, а збиранням всіх помилок на рівні модуля (лексера, парсера, чи генератора коду). Потім, з цих помилок формується великий error list, котрий

тільки тепер викидається в якості exception’а. Це дозволяє побачити відразу цілий пласт помилок, та полагодити їх «пачкою», а не запускати асемблювання після виправлення кожної помилки, щоб знову отримати нову помилку.

Після завершення реалізації цих трьох основних модулів, а також, модулю вводу/виводу (що відповідає за відкриття вихідного коду та збереження фінального результату) можна сказати, що практична реалізація асемблера для набору інструкцій 6502 завершена. Наступним етапом стане розробка напів-автоматизованих тестів, проведення тестування та дебаґґінґ програмного забезпечення.

3.2 Розробка тестів, проведення тестування та налагодження інформаційно-комп’ютерної системи

Тестування програмного забезпечення проводиться на усіх етапах його життєвого циклу. Саме регулярне проведення контролю якості (Quality Assurance) є запорукою стабільної розробки та уникнення казусів, що можуть спричинити необхідність переробки частини, а то й усього ПЗ. Відповідно, відсутність регулярних перевірок якості може призвести до значного збільшення часових та грошових витрат на розробку.

У нашому випадку, ми також проводили тестування, хоч і не масштабне, на кожному етапі. Під час визначення типу розроблюваного транслятора ми розглянули доводи «за» і «проти» розробки компілятора, інтерпретатора та асемблера. Здорово оцінивши свої можливості та ресурси, ми зупинилися на ідеї реалізації останнього як найбільш простого у розробці.

На етапі аналізу вимог були визначені основні функції, котрі нам необхідно буде реалізувати, а також, був обраний набір команд, під який ми і розроблятимемо наш асемблер. На даному етапі тестувалася як потенційна складність майбутньої розробки (в залежності від вимог), так і доцільність вибору саме таких характеристик продукту.

На етапі практичної реалізації ми постійно проводили комплексну перевірку роботи вручну. Так, ми звіряли результати роботи нашого асемблера та того, котрий обрали за приклад. Окрім цього, ми перевіряли результати за допомогою відкриття виконуваного файлу в hex-редакторі та звірки отриманих кодів з матрицею, приведеною у додатку А, таблиці А.1. Зачасту, на цьому етапі розробники пишуть так звані юніт-тести (unit-tests), що призначені для перевірки роботи конкретних модулів. Але, у нашому випадку, через специфіку роботи та інструментів, проведення такого виду тестування було не зовсім доцільним.

У кінці, необхідно було розробити наглядний фінальний тест. Для цього, ми написали просту програму на мові Python, що звіряє результат роботи нашого асемблера з результатом роботи асемблера з симулятора «6502js». При цьому, спочатку була задумка щодо реалізації повністю автоматизованого тесту. Однак, через певні відмінності в синтаксисі (приміром, наш асемблер використовує «.define», замість «define», «.byte» замість «dcb», тощо), було прийнято рішення розробити лише часткову автоматизацію.

Використовувати даний тест необхідно за наступним принципом:

1) зберегти програму, написану на мові асемблера 6502 з використанням правил синтаксису нашого асемблера у файл з розширенням «.asm»;

2) скопіювати збережений код у новий файл, після чого, замінити де-які синтаксичні особливості на підтримувані симулятором «6502js». Як мінімум, замінити команди «.byte» на «dcb», «.define» на «define», а також (якщо необхідно) перетворити строки (strings) у вигляд байтових рядків. При цьому, все, окрім останнього, робиться автозаміною у будь-якому текстовому редакторі;

3) змінений вихідний код засемблювати у симуляторі, після чого, отриманий шляхом натиснення на кнопку «hexdump» результат повністю скопіювати у текстовий файл, який необхідно зберегти у папку з оригінальним кодом (з кроку 1) та назвати його так само, але з розширенням «.txt»;

4) запустити інтерпретатор Python, куди першим аргументом передати файл тесту (__init__.py), а другим – шлях до оригінального вихідного коду, збереженого на етапі 1. Якщо результатом роботи буде True, то асемблер працює нормально, якщо ж False – є помилка.

На цьому тестування розробленого асемблера під набір інструкцій процесора MOS Technology 6502 можна вважати завершеним.

Висновки до розділу 3

Розробка власного асемблера під набір інструкцій процесора MOS Technology 6502 дозволила закріпити знання та вміння щодо повного циклу розробки програмного забезпечення.

На початку практичної реалізації були визначені основні синтаксичні конструкції мови асемблера, котру ми збиралися транслювати. На цьому етапі ми розбили всі конструкції на де-кілька груп: синтаксис для машинних інструкцій, «синтаксичний цукор» та, окремо, коментарі.

Далі, була проведене розробка лексичного аналізатора, що, за заздалегідь визначеними правилами розбирав вихідний код програми на мові асемблера на набір лексичних токенів.

Після цього, був реалізований парсер, що збирав з вище згаданих токенів логічні команди (тобто, інструкції разом з їх аргументами), та в такому вигляді, віддавав їх до генератора коду.

Останнім в програмній реалізації був генератор коду, що перетворював відправлені уже парсером токени у машинні команди у вигляді двійкового коду.

Далі, по завершенню програмування асемблера, був проведений етап написання тестів на мові Python. Тут був розроблений головний тест, що звіряє дані, отримані від нашого асемблера з даними, обробленими за допомогою іншого асемблера з таким же набором інструкцій.

У подальшому, розроблений асемблер, що отримав назву MXASM може розвиватися у де-кількох напрямках. По-перше, це можливість підтримки більшої кількості «синтаксичного цукру». Наприклад, є можливість додати високорівневу «обгортку» у вигляді інструкцій «if … else …». По-друге, асемблер може бути розширений для роботи з наборами інструкцій інших процесорів.

ВИСНОВКИ

Розробка машинних трансляторів є перспективним напрямком у інженерії програмного забезпечення. Вона вимагає глибоких та фундаментальних знань щодо будови комп'ютера, теорії трансляції та принципів оптимізації роботи ПЗ. Саме тому, темою даної кваліфікаційної роботи була обрана розробка одного з видів трансляторів – асемблера.

Проведення теоретичного дослідження та закріплення отриманих знань на практиці під час виконання даної кваліфікаційної роботи дозволили навчитися розробляти прості транслятори мов програмування, у першу чергу – асемблери. А, оскільки асемблери працюють на стикові, власне, мов програмування та машинних команд, така тематика дала можливість отримати широкий спектр знань і навичок ще й відносно архітектури комп'ютера.

Перед початком роботи був проведений теоретичний аналіз предметної області об'єкта проектування та сучасного стану проблеми. Були проаналізовані як природа асемблерів в загальному, так і аналогічні рішення, що присутні на ринку.

По тому, було проведене дослідження архітектури, основних підходів та методів конструювання машинних трансляторів (здебільшого, компіляторів), з акцентом на асемблерах.

Далі, були здійснені аналіз вимог до фінального ПЗ та огляд архітектури процесора, для набору команд якого розроблявся асемблер, а також, було проведене проектування продукту та вибір технологій і алгоритмів його реалізації.

Після цього, програмне забезпечення було сконструйовано, за чим послідувало фінальне тестування.

Виконання кваліфікаційної роботи дозволило навчитися проектувати та конструювати машинні транслятори. Так, під час вивчення теоретичної частини була освоєна їх базова архітектура, досліджені принципи побудови їх основних

структурних компонентів, як-от лексерів та парсерів, а також, способи конвертації високорівневих команд в машинний код.

Під час же практичної реалізації асемблера були наочно закріплені отримані раніше знання та навички, а також, покращений досвід розробки програмного забезпечення в цілому.

Окрім цього, оскільки розробка велася під незнайому мені архітектуру, це дозволило ознайомитися з новими технічними рішеннями, як-от memory-mapped I/O, а також, краще закріпити знання будови комп’ютера.

Не менш важливо, що, як сам асемблер, так і досвід, отриманий під час його розробки можна використовувати для наглядного навчання інших інженерів проектуванню та конструюванню трансляторів.

В цілому, як сама розробка машинних трансляторів, так і навички, що використовуються в її процесі є хоч і нішевими, але досить перспективними. Особливо, враховуючи сучасні тренди на спрощення розробки та появу no-code та low-code рішень, котрі, без сумнівів, потребуватимуть спеціалістів для своєї реалізації. Окрім цього, не варто забувати і про дослідження в області квантових комп’ютерів. Хоча ця технологія поки що лише в стані зародження, але, у перспективі, вона також потребуватиме різних інструментів для розробників, котрі будуть її використовувати, в тому числі, і спеціальних трансляторів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Translators. *Tech Computer Science*. URL: https://teachcomputerscience.com/translators/ (дата звернення: 13.02.2023).

2. Машинний код. *Wikipedia*. URL: https://w.wiki/Da6d (дата звернення: 13.02.2023).

3. Assembler Directives. *Kick Assembler*. URL: http://www.theweb.dk/KickAssembler/webhelp/content/apas04.html (дата звернення: 14.02.2023).

4. Intel and AT&T Syntax. *CSCI 223 Computer Organisation and Assembly Language*. URL: https://imada.sdu.dk/u/kslarsen/dm546/Material/IntelnATT.htm (дата звернення: 14.02.2023).

5. Differences in Intel (NASM) vs AT&T (GAS) Syntax. URL: https://sdasgup3.github.io/Intel_Vs_Att_format/ (дата звернення: 14.02.2023).

6. Watts S. What is Spaghetti Code (And Why You Should Avoid It). *BMC*. URL: https://www.bmc.com/blogs/spaghetti-code/ (дата звернення: 15.02.2023).

7. Lefebvre P. Compilers Series' Articles. *Dev.to*. URL: https://dev.to/lefebvre/series/21363 (дата звернення: 17.02.2023).

8. Can You Trust a Compiler to Optimize Your Code? *matklad*. URL: https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html (дата звернення: 17.02.2023).

9. X86-64 Instruction Encoding. URL: https://www-user.tu-chemnitz.de/~heha/hsn/chm/x86.chm/x64.htm (дата звернення: 20.02.2023).

10. MOS Technology. MCS6500 Microprocessors. Preliminary Data Sheet. URL: http://archive.6502.org/datasheets/mos_6500_mpu_preliminary_may_1976.pdf (дата звернення: 20.02.2023).

11. 6502 Registers. *Codebase64*. URL: https://codebase64.org/doku.php?id=base:6502_registers (дата звернення: 20.02.2023).

12. Zack Jorquera. What is Little-Endian And Big-Endian Byte Ordering? *Section*. URL: https://www.section.io/engineering-education/what-is-little-endian-and-big-endian/ (дата звернення: 21.02.2023).

13. Instruction Set. Address Modes. *Masswerk*. URL: https://www.masswerk.at/6502/6502_instruction_set.html (дата звернення: 21.02.2023).

14. Extensions to the C Language Family. *gnu.org*. URL: https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html (дата звернення: 22.02.2023).

15. Juan Arrieta. Simple C++ Lexer. *GitHub Gist*. URL: https://gist.github.com/arrieta/1a309138689e09375b90b3b1aa768e20 (дата звернення: 23.02.2023).

ДОДАТКИ

Додаток А

Таблиця А.1 – Instruction Set процесора MOS Technology 6502

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK stk	ORA (zp,X)				ORA zp	ASL zp		PHP stk	ORA #	ASL A			ORA abs	ASL abs	
1-	BPL rel	ORA (zp),Y				ORA zp,X	ASL zp,X		CLC imp	ORA abs,Y				ORA abs,X	ASL abs,X	
2-	JSR abs	AND (zp,X)			BIT zpg	AND zp	ROL zp		PLP stk	AND #	ROL A		BIT abs	AND abs	ROL abs	
3-	BMI rel	AND (zp),Y				AND zp,X	ROL zp,X		SEC imp	AND abs,Y				AND abs,X	ROL abs,X	
4-	RTI stk	EOR (zp,X)				EOR zp	LSR zp		PHA stk	EOR #	LSR A		JMP abs	EOR abs	LSR abs	
5-	BVC rel	EOR (zp),Y				EOR zp,X	LSR zp,X		CLI imp	EOR abs,Y				EOR abs,X	LSR abs,X	
6-	RTS stk	ADC (zp,X)				ADC zp	ROR zp		PLA stk	ADC #	ROR A		JMP (ind)	ADC abs	ROR abs	
7-	BVS rel	ADC (zp),Y				ADC zp,X	ROR zp,X		SEI imp	ADC abs,Y				ADC abs,X	ROR abs,X	
8-		STA (zp,X)			STY zp	STA zp	STX zp		DEY imp		TXA imp		STY abs	STA abs	STX abs	
9-	BCC rel	STA (zp),Y			STY zp,X	STA zp,X	STX zp,Y		TYA imp	STA abs,Y	TXS imp			STA abs,X		
A-	LDY #	LDA (zp,X)	LDX #		LDY zp	LDA zp	LDX zp		TAY imp	LDA #	TAX imp		LDY abs	LDA abs	LDX abs	
B-	BCS rel	LDA (zp),Y			LDY zp,X	LDA zp,X	LDX zp,Y		CLV imp	LDA abs,Y	TSX imp		LDY abs,X	LDA abs,X	LDX abs,Y	
C-	CPY #	CMP (zp,X)			CPY zp	CMP zp	DEC zp		INY imp	CMP #	DEX imp		CPY abs	CMP abs	DEC abs	
D-	BNE rel	CMP (zp),Y				CMP zp,X	DEC zp,X		CLD imp	CMP abs,Y				CMP abs,X	DEC abs,X	

Продовження таблиці А.1

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
E-	CPX #	SBC (zp,X)			CPX zp	SBC zp	INC zp		INX imp	SBC #	NOP imp		CPX abs	SBC abs	INC abs	
F-	BEQ rel	SBC (zp),Y				SBC zp,X	INC zp,X		SED imp	SBC abs,Y				SBC abs,X	INC abs,X	

Додаток Б

Таблиця Б.1 – Список інструкцій процесора MOS Technology 6502

№ з/п	Інструкція	Опис	№ з/п	Інструкція	Опис
1	ADC	Add with carry	29	JSR	Jump subroutine
2	AND	And (with accumulator)	30	LDA	Load accumulator
3	ASL	Arithmetic shift left	31	LDX	Load X
4	BCC	Branch on carry clear	32	LDY	Load Y
5	BCS	Branch on carry set	33	LSR	Logical shift right
6	BEQ	Branch on equal (zero set)	34	NOP	No operation
7	BIT	Bit test	35	ORA	Or with accumulator
8	BMI	Branch on minus (negative set)	36	PHA	Push accumulator
9	BNE	Branch on not equal (zero clear)	37	PHP	Push processor status (P)
10	BPL	Branch on plus (negative clear)	38	PLA	Pull accumulator
11	BRK	Break / Interrupt	39	PLP	Pull processor status (P)
12	BVC	Branch on overflow clear	40	ROL	Rotate left
13	BVS	Branch on overflow set	41	ROR	Rotate right
14	CLC	Clear carry	42	RTI	Return from interrupt
15	CLD	Clear decimal	43	RTS	Return from subroutine
16	CLI	Clear interrupt disabled	44	SBC	Subtract with carry
17	CLV	Clear overflow	45	SEC	Set carry
18	CMP	Compare (with accumulator)	46	SED	Set decimal
19	CPX	Compare with X	47	SEI	Set interrupt disables
20	CPY	Compare with Y	48	STA	Store accumulator
21	DEC	Decrement	49	STX	Store X
22	DEX	Decrement X	50	STY	Store Y
23	DEY	Decrement Y	51	TAX	Transfer accumulator to X
24	EOR	Exclusive or (with accumulator)	52	TAY	Transfer accumulator to Y
25	INC	Increment	53	TSX	Transfer stack pointer to X
26	INX	Increment X	54	TXA	Transfer X to accumulator
27	INY	Increment Y	55	TXS	Transfer X to stack pointer
28	JMP	Jump	56	TYA	Transfer Y to accumulator

Додаток В

Таблиця В.1 – Режими адресації процесора MOS Technology 6502

Режим	Назва	Сигнатура	Код	Опис
A	Accumulator	OPC A	OP	Операнд – регістр A
abs	Absolute	OPC $HHLL	OP LL HH	Операнд – абсолютна адреса $HHLL
abs,X	Absolute, X-indexed	OPC $HHLL,X	OP LL HH	Операнд – адреса; ефективна адреса – адреса з додаванням X із переносом
abs,Y	Absolute, Y-indexed	OPC $HHLL,Y	OP LL HH	Операнд – адреса; ефективна адреса – адреса з додаванням Y із переносом
#	Immediate	OPC #$BB	OP BB	Операнд – байт $BB
impl	Implied	OPC	OP	Операнд неявний
ind	Indirect	OPC ($HHLL)	OP LL HH	Операнд – адреса; ефективна адреса – вміст word за адресою ($HHLL)
X,ind	X-indexed, indirect	OPC ($LL,X)	OP LL	Операнд – zeropage адреса; ефективна адреса – word на (LL + X, LL + X + 1) з додаванням без переносу ($00LL + X)
ind,Y	Indirect, Y-indexed	OPC ($LL),Y	OP LL	Операнд – zeropage адреса; ефективна адреса – word на (LL, LL + 1) з додаванням Y без переносу ($00LL) + Y
rel	Relative	OPC $BB	OP BB	Операнд – мітка (зміщення); цільова адреса – PC + знакове зміщення $BB
zpg	Zeropage	OPC $LL	OP LL	Операнд – zeropage адреса ($00LL)
zpg,X	Zeropage, X-indexed	OPC $LL,X	OP LL	Операнд – zeropage адреса; ефективна адреса – адреса з додаванням X без переносу
zpg,Y	Zeropage, Y-indexed	OPC $LL,Y	OP LL	Операнд – zeropage адреса; ефективна адреса – адреса з додаванням Y без переносу