

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет прикладної математики

**Кафедра системного програмування і спеціалізованих
комп'ютерних систем**

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___» _____ 2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Системне програмування та спеціалізовані комп'ютерні системи»

спеціальності 123 «Комп'ютерна інженерія»

**на тему: «Спеціалізована комп'ютерна система для емуляції і
налагодження програм»**

Виконав (-ла):

студент (-ка) IV курсу, групи KB-12

Чернишков Олександр Олександрович _____

Керівник:

Доцент, кандидат наук, старший науковий співробітник

Тесленко Олександр Кирилович _____

Консультант з нормоконтролю:

Доцент кафедри СПСКС, кандидат наук

Клятченко Ярослав Михайлович _____

Рецензент:

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент (-ка) _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування і
спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___» _____ 20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Чернишкову Олександр Олександровичу

1. Тема проєкту «Спеціалізована комп'ютерна система для емуляції і налагодження програм», керівник проєкту Тесленко Олександр Кирилович, кандидат наук, старший науковий співробітник, затверджені наказом по університету від «___» _____ 20__ р. № _____

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту _____

4. Зміст пояснювальної записки _____

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) _____

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	15.04.2025	
2.	Розроблення та узгодження технічного завдання	30.04.2025	
3.	Аналіз існуючих рішень	05.05.2025	
4.	Підготовка матеріалів першого розділу дипломного проєкту	10.05.2025	
5.	Розробка програмної реалізації дипломного проєкту	15.05.2025	
6.	Підготовка матеріалів другого розділу дипломного проєкту	18.05.2025	
7.	Підготовка графічної частини дипломного проєкту	20.05.2025	
8.	Оформлення документації дипломного проєкту	25.05.2025	
9.	Попередній огляд матеріалів диплому на кафедрі	30.05.2025	

Студент

Олександр ЧЕРНИШКОВ

Керівник

Олександр ТЕСЛЕНКО

АНОТАЦІЯ

Бакалаврський дипломний проєкт включає пояснювальну записку (61 стор., 8 рис., 5 табл., список використаної літератури з 8 найменувань, 2 додатків).

Метою розробки дипломного проєктування є спеціалізована комп'ютерна система для емуляції та налагодження програм, що виконуються на умовному мікропроцесорі. Основною ціллю роботи є створення навчального та практичного інструменту, який дозволяє в інтерактивному режимі досліджувати виконання інструкцій, роботу регістрів, пам'яті та логіку програмного коду, виконувати налагодження роботи програмного коду, не використовуючи фізичне апаратне забезпечення.

Програмне забезпечення спеціалізованої комп'ютерної системи реалізовано мовою C++ з використанням бібліотеки Qt для створення графічного інтерфейсу користувача. В основі системи – емулятор умовного процесора з підтримкою базового набору інструкцій (арифметичних, логічних, умовного та безумовного переходу, інструкцій взаємодії із пам'яттю), симуляція регістрів, оперативної пам'яті та покрокове виконання команд. Користувач має змогу завантажувати програму з текстового файлу, запускати її виконання повністю або покроково, аналізуючи в реальному часі зміну стану пам'яті та регістрів.

У процесі реалізації спроектовано модульну архітектуру, що включає окремі компоненти для інтерфейсу, обробки інструкцій та логіки виконання. У рамках роботи також проведено порівняльний аналіз існуючих інструментів емуляції та налагодження, зокрема таких як QEMU, DOSBox, GDB, Turbo Debugger та інших. Було виявлено, що більшість з них орієнтовані на широко функціональне або спеціалізоване використання, мають складні інтерфейси та високу вхідну складність для новачків. Це стало підставою для розробки власного компактного рішення, оптимізованого для навчального середовища та наочного дослідження, налагодження виконання програм на спрощеній архітектурі.

Розроблене рішення може бути використане у навчальних цілях для вивчення архітектури процесора, основ комп'ютерної логіки, принципів роботи налагоджувачів та операцій пам'яті.

Ключові слова:

Емуляція, налагодження, RISC-процесор, Qt, інтерфейс, регістр, пам'ять, C++, інструкції, симуляція, покрокове виконання, навчальна система.

ABSTRACT

The bachelor's thesis includes an explanatory report (61 pages, 8 figures, 5 tables, a list of 8 references, and 2 appendices).

The objective of the diploma project is the development of a specialized computer system for the emulation and debugging of programs executed on a hypothetical microprocessor. The main goal is to create an educational and practical tool that allows interactive exploration of instruction execution, register operations, memory usage, and program logic, as well as debugging functionality, without relying on physical hardware.

The software for the specialized computer system was implemented in C++ using the Qt library to build the graphical user interface. The core of the system is an emulator of a simplified processor that supports a basic instruction set (arithmetic, logical, conditional and unconditional branching, memory access instructions), register simulation, memory simulation, and step-by-step execution of instructions. The user can load a program from a text file and run it either continuously or step-by-step, observing memory and register state changes in real time.

During the development process, a modular architecture was designed, consisting of separate components for the interface, instruction processing, and execution logic. Additionally, a comparative analysis of existing emulation and debugging tools—such as QEMU, DOSBox, GDB, Turbo Debugger, and others—was conducted. It was found that most of these tools are targeted toward advanced or highly specialized usage and tend to have complex interfaces with a steep learning curve for beginners. This prompted the development of a compact and intuitive solution optimized for educational purposes and visual exploration of instruction flow on a simplified architecture.

The developed solution can be used in academic settings to study processor architecture, computer logic fundamentals, debugger operation principles, and memory management.

Keywords:

Emulation, debugging, RISC processor, Qt, interface, register, memory, C++, instructions, simulation, step execution, educational tool.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	A4	ІАЛЦ.045300.006 Д2	Організація тестування програми.	1		
			Схема алгоритму			
	A4	ІАЛЦ.045300.007 Д3	Організація класів та їх взаємодії.	1		
			Схема структурна			
	A4	ІАЛЦ.045300.008 Д4	Організація викликів функцій та методів.	1		
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної записки.	1		
			Лістинг розробленої програми.			
			Графічний матеріал.			
			Презентація бакалаврського проєкту.			

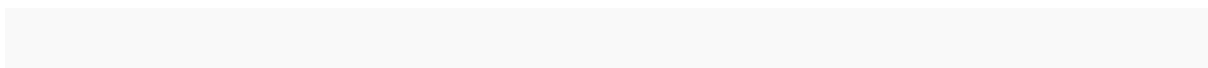
Змін.	Арк.	№ докум.	Підпис	Дата

ІАЛЦ.045300.001 ОА

Арк.
2

ЗМІСТ

<u>1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ</u>	2
<u>2. ПІДСТАВА ДЛЯ РОЗРОБКИ</u>	2
<u>3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ</u>	2
<u>4. ДЖЕРЕЛА РОЗРОБКИ</u>	2
<u>5. ТЕХНІЧНІ ВИМОГИ</u>	2
<u>5.1. Вимоги до програмного продукту, що розробляється</u>	2
<u>5.2. Вимоги до апаратного забезпечення</u>	3
<u>5.3. Вимоги до програмного та апаратного забезпечення користувача</u>	3
<u>6. ЕТАПИ РОЗРОБКИ</u>	4



					ІАЛЦ. 045300.002 ТЗ				
Зм	Лист	№ докум.	Підп.	Дата	Спеціалізована комп'ютерна система для емуляції і налагодження програм Технічне завдання				
Розроб.	Чернишков О.О.								
Перев.	Тесленко О. К.								
Н. контр.	Клятченко Я. М.								
Зав. каф.	Тарасенко В. П.				Лім. Лист 1 Листів 4 НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-12				

НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Спеціалізована комп'ютерна система для емуляції і налагодження програм».

Галузь застосування: освітні та дослідницькі задачі у сфері комп'ютерної інженерії, зокрема вивчення принципів роботи процесорів, відлагодження програмного коду на низькому рівні.

ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проєкту є створення спеціалізованої комп'ютерної системи, що дозволяє проводити емуляцію та покрокове налагодження програм, написаних під віртуальний процесор із обмеженим набором інструкцій.

ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації у періодичних виданнях та електронні статті у мережі Інтернет.

ТЕХНІЧНІ ВИМОГИ

Вимоги до програмного продукту, що розробляється

- підтримка завантаження вхідних програм з текстових файлів;

					ІАЛЦ.045300.002 ТЗ	Лист
Зм	Лист	№ докум.	Підп.	Дата		2

- відображення структури пам'яті: адреси, значення, регістри;
- підтримка покрокового виконання інструкцій;
- підтримка емуляції обмеженого набору інструкцій;
- підтримка візуального графічного інтерфейсу користувача;
- можливість зупинки, скидання та повторного запуску програми;
- візуальне відображення поточної інструкції та зміни у пам'яті.

Вимоги до апаратного забезпечення

- Процесор: Intel або AMD з частотою не нижче 2.0 ГГц;
- Оперативна пам'ять: від 4 ГБ;
- Відеоадаптер з підтримкою OpenGL 2.0;
- Мишка та клавіатура.

Вимоги до програмного та апаратного забезпечення користувача

- Операційна система: Windows 10/11 x64.

					ІАЛЦ.045300.002 ТЗ	Лист
						3
Зм	Лист	№ докум.	Підп.	Дата		

ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів
1.	Вивчення літератури за тематикою проєкту	15.04.2025
2.	Розроблення та узгодження технічного завдання	30.04.2025
3.	Аналіз існуючих рішень	05.05.2025
4.	Підготовка матеріалів першого розділу дипломного проєкту	10.05.2025
5.	Розробка програмної реалізації дипломного проєкту	15.05.2025
6.	Підготовка матеріалів другого розділу дипломного проєкту	18.05.2025
7.	Підготовка графічної частини дипломного проєкту	20.05.2025
8.	Оформлення документації дипломного проєкту	25.05.2025
9.	Попередній огляд матеріалів диплому на кафедрі	30.05.2025

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	A4	ІАЛЦ.045300.004 ПЗ	Спеціалізована комп'ютерна система для емуляції і налагодження програм	61		
			Пояснювальна записка			
	A4	ІАЛЦ.045300.005 Д1	Виконання роботи програми.	1		
			Схема алгоритму			
	A4	ІАЛЦ.045300.006 Д2	Організація тестування програми.	1		
			Схема алгоритму			
	A4	ІАЛЦ.045300.007 Д3	Організація класів та їх взаємодії.	1		
			Схема структурна			
	A4	ІАЛЦ.045300.008 Д4	Організація викликів функцій та методів.	1		
			Схема алгоритму			

[illegible]

ЗМІСТ

<u>ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ</u>	3
<u>ВСТУП</u>	4
<u>1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ</u>	6
<u>1.1. Огляд сучасних мікропроцесорів простої архітектури</u>	6
<u>1.2. Поняття емуляції та віртуалізації обчислювальних процесів</u>	15
<u>1.3. Призначення та функції програм-налагоджувачів</u>	17
<u>1.4. Порівняльний аналіз існуючих емуляторів та налагоджувачів</u>	19
<u>1.5. Оцінка можливості застосування існуючих рішень у навчальних цілях</u>	27
<u>1.6. Оцінка та обґрунтування теми дипломного проекту</u>	28
<u>2. АЛГОРИТМ І РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ</u>	30
<u>2.1. Загальна структура програми</u>	30
<u>2.2. Формат подання вхідної програми та інструкцій</u>	32
<u>2.3. Алгоритм виконання інструкцій емулятором</u>	33
<u>2.4. Опис регістрів та реалізованих інструкцій</u>	37
<u>2.5. Структура моделі пам'яті</u>	40
<u>2.6. Побудова графічного інтерфейсу користувача</u>	41
<u>2.7. Механізми покрокового виконання та оновлення пам'яті</u>	44
<u>2.8. Опис основних класів, структур та логіки взаємодії</u>	46
<u>3. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ</u>	51

					ІАЛЦ.045300.004 ПЗ									
З.м.м	Лист	№ докум.	Підп.	Дата	Спеціалізована комп'ютерна система для емуляції і налагодження програм Пояснювальна записка					Літ.	Лист	Листів		
Розроб.	Чернишков О.О.												1	61
Перев.	Тесленко О.К.													
Н. контр.	Клятченко Я.М.													
Зав. каф.	Тарасенко В.П.									НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-12				

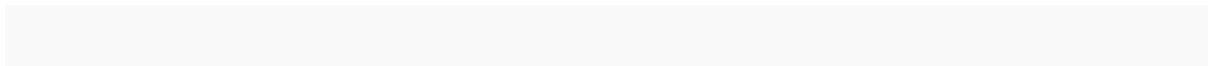
3.1. Верифікація роботи реалізованої програми	51
3.2. Оцінка зручності використання інтерфейсу	54
3.3. Порівняння результатів з існуючими аналогами	55
3.4. Висновки щодо ефективності реалізації	56
ВИСНОВОК	58
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	61

ДОДАТКИ

Додаток 1. Копії графічних матеріалів

- ІАЛЦ.045300.005 Д1. Виконання роботи програми. Схема алгоритму
- ІАЛЦ.045300.006 Д2. Організація тестування програми. Схема алгоритму
- ІАЛЦ.045300.007 Д3. Організація класів та їх взаємодії. Схема структурна
- ІАЛЦ.045300.008 Д4. Організація викликів функцій та методів. Схема алгоритму

Додаток 2. Фрагменти програмного коду



ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

АЛП – Арифметико-логічний пристрій.

RISC – Система зі скороченим набором інструкцій (Reduced Instruction Set Computer).

ARM – 32-бітна RISC архітектура процесорів (Advanced RISC Machine).

NVIC – Модуль контролю переривань (Nested Vectored Interrupt Controller).

IoT – Інтернет речей (Internet of Things).

PC – Лічильник програми (Program Counter).

UART – Універсальний асинхронний приймач/передавач.

SPI – Послідовний периферійний інтерфейс (Serial Peripheral Interface).

GDB – Налагоджувач GNU Debugger.

AFD – Налагоджувач Advanced Full-screen Debugger.

C++ – Мова програмування загального призначення.

Qt – Крос-платформовий інструментарій розробки програмного забезпечення.

MIPS – Архітектура мікропроцесорів (Microprocessor without Interlocked Pipeline Stages).

GUI – Графічний інтерфейс користувача (Graphical user interface).

ВСТУП

У сучасному світі розвиток комп'ютерної техніки та програмного забезпечення відбувається стрімкими темпами. Зростає складність мікропроцесорних систем, підвищуються вимоги до ефективності, надійності та оптимізації коду, особливо у вбудованих системах та ресурсно обмежених пристроях. У зв'язку з цим усе більшої актуальності набувають засоби, що дозволяють аналізувати, тестувати та налагоджувати поведінку програм без необхідності безпосереднього виконання на фізичному обладнанні. Одним із таких інструментів є програмні емулятори та налагоджувачі. Емуляція є важливою частиною процесу розробки програмного забезпечення для мікропроцесорних систем, особливо тоді, коли фізичний пристрій ще не створено або його використання є небажаним через ризики, вартість чи складність. Емулятори дозволяють створити віртуальне середовище, в якому модель процесора імітує поведінку справжнього пристрою, що відкриває широкі можливості для відлагодження, тестування та навчання. У свою чергу, налагоджувачі дають змогу детально дослідити хід виконання програми, відстежувати значення змінних, вміст регістрів, оперативної пам'яті та знаходити логічні помилки у коді. Однак попри наявність багатьох комерційних і відкритих програмних рішень, більшість з них є або надто складними, або надто орієнтованими на конкретне обладнання чи комерційні цілі. У навчальних цілях або для розробки простих систем потрібен доступний та інтуїтивно зрозумілий інструмент, який дозволяє дослідити основні принципи роботи мікропроцесорів, аналізувати команди, структуру пам'яті та логіку виконання інструкцій у максимально наочному вигляді. Саме на вирішення цієї проблеми спрямований даний дипломний проєкт, метою роботи якого є створення програмного інструменту, що дозволяє імітувати виконання програм на базі умовної простої мікропроцесорної архітектури, відображати хід виконання інструкцій, маніпулювати пам'яттю та мати зручний графічний

інтерфейс користувача, орієнтований на навчальне використання. Розроблений інструмент дозволить студентам, викладачам та інженерам-початківцям краще зрозуміти принципи роботи комп'ютерних систем, відлагоджувати навчальні програми, експериментувати з пам'яттю та інструкціями, не маючи під рукою фізичного пристрою. Крім того, даний проєкт стане корисною основою для подальшого розвитку – наприклад, для підтримки інших архітектур, реалізації трасування виконання, профілювання продуктивності тощо.

					ІАЛЦ.045300.004 ПЗ	Лист
						5
Зм	Лист	№ докум.	Підп.	Дата		

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ

Огляд сучасних мікропроцесорів простої архітектури

Мікропроцесор – це центральний елемент комп’ютерної або вбудованої обчислювальної системи, який виконує обробку даних відповідно до заданої послідовності інструкцій. З технічної точки зору, мікропроцесор є інтегральною схемою, яка об’єднує в собі кілька функціональних блоків: пристрій керування, арифметико-логічний пристрій (АЛП), набір регістрів, шину обміну даними та пристрій формування адреси.

Пристрій керування відповідає за декодування та послідовне виконання інструкцій, які зчитуються з оперативної пам’яті. АЛП виконує арифметичні та логічні операції над даними, що зберігаються у регістрах. Регістри – це швидкодійна пам’ять мікропроцесора, в якій зберігаються операнди і результати обчислень. Шина забезпечує взаємодію між усіма модулями системи, включно з пам’яттю та зовнішніми пристроями. Пристрій формування адреси виконує необхідні обчислення для адресного доступу до даних у пам’яті.

У сучасних процесорах також часто інтегрований математичний співпроцесор (FPU), що спеціалізується на операціях з рухомою комою. Завдяки можливості роботи паралельно із АЛП, він дозволяє суттєво прискорити виконання складних математичних обчислень, що мають широке застосування в наукових, графічних та інженерних обчисленнях. Крім того, багато мікропроцесорів містять вбудовані кеш-пам’ять, таймери, контролери переривань та інші периферійні модулі.

Мікропроцесори широко використовуються не тільки в персональних комп’ютерах, але й у вбудованих системах: мобільних телефонах, мережевому обладнанні, системах автоматизації, побутовій техніці, автомобільній електроніці. У таких застосуваннях вони часто є частиною мікроконтролера (MCU) або системи на кристалі (SoC).

Існує кілька основних типів архітектур мікропроцесорів. Найпоширенішими є:

- CISC (Complex Instruction Set Computer) – архітектура зі складним набором команд. Кожна інструкція може виконувати складні операції, що зменшує кількість команд у програмі, але часто вимагає більше тактів для виконання. Прикладом є архітектура x86.
- RISC (Reduced Instruction Set Computer) – архітектура з обмеженим, але оптимізованим набором простих інструкцій. Перевагами є зменшена складність мікропроцесора, можливість реалізації конвеєрного виконання команд і, відповідно, підвищення продуктивності при меншому енергоспоживанні. Найвідоміші представники – ARM, RISC-V.
- VLIW (Very Long Instruction Word) та EPIC (Explicitly Parallel Instruction Computing) – менш поширені архітектури, що застосовуються у високопродуктивних або спеціалізованих системах, де є потреба у високому рівні паралелізму.

У цьому дипломному проєкті основна увага буде приділена архітектурам типу RISC, оскільки вони є більш придатними для моделювання, аналізу та навчання основам функціонування процесорів. Серед ключових принципів RISC-архітектури слід відзначити:

- обмежений набір інструкцій, кожна з яких виконується за фіксований (часто один) такт;
- однакова довжина всіх інструкцій, що полегшує їх декодування;
- мінімальна кількість форматів інструкцій;
- відсутність інструкцій, які безпосередньо працюють з пам'яттю (окрім завантаження та збереження);
- велика кількість універсальних регістрів;
- фокус на спрощенні апаратної реалізації для забезпечення високої швидкодії та низького енергоспоживання.

Для глибшого розуміння архітектур RISC доцільно порівняти кілька реальних мікропроцесорів, які належать до цього класу. Зокрема, буде розглянуто три популярні представники:

1. Xtensa LX6 у складі мікроконтролера ESP32 – 32-бітний мікропроцесор, що реалізує RISC-подібну архітектуру, та має можливість конфігуруватися до вимог застосування у складі мікроконтролера;

2. ARM Cortex-M3 – 32-бітний мікропроцесор, що поєднує високу продуктивність та низьке енергоспоживання, та який використовується у великому різноманітті користувацьких пристроїв, а не лише у складі мікроконтролерів;

3. Atmel AVR5 у складі ATmega328P – 8-бітний мікропроцесор який реалізує систему команд, близьку до ідеології RISC, та став широко відомий завдяки платформі Arduino.

Розглянемо архітектуру мікропроцесора Xtensa LX6 у складі мікроконтролера ESP32. Мікроконтролер ESP32, розроблений компанією Espressif Systems, широко використовується в IoT-пристроях завдяки своїй низькій вартості, енергоефективності та високому ступеню інтеграції. В його основі лежить одне або два ядра Xtensa LX6, ліцензованих у компанії Cadence Design Systems, які реалізують RISC-подібну архітектуру з можливістю розширення набору інструкцій та налаштування апаратних блоків під потреби виробника [1]. Основні характеристики Xtensa LX6:

- 32-розрядна гарвардська архітектура з окремими шинами інструкцій і даних;
- Підтримка до 64 регістрів загального призначення;
- Можливість розширення інструкцій через Xtensa ISA customization;
- Підтримка векторних переривань;
- Підтримка конфігурованих кеш-пам'яті (до 64 КБ інструкцій та 64 КБ даних);
- Наявність MMU (memory management unit) у версії з RTOS;

- Робота на частоті до 240 МГц у ESP32.

У стандартній конфігурації ядро Xtensa LX6 використовує 5-стадійний конвеєр, що забезпечує виконання більшості інструкцій майже за один такт, за винятком доступу до пам'яті або операцій з плаваючою комою (за наявності відповідного співпроцесора). Проте у версії, реалізованій у мікроконтролері ESP32, компанія Espressif модифікувала архітектуру, збільшивши глибину конвеєра до 7 стадій. Це дозволило досягти вищих тактових частот – до 240 МГц – без зменшення продуктивності та з урахуванням енергоспоживання [2]. На схемі на рисунку 1 можна побачити приклад реалізації 5-стадійного конвеєра LX6, аналогічний якому був реалізований у багатьох мікропроцесорах RISC-архітектури.

У стандартній конфігурації ядро має 16 регістрів загального призначення (AR), однак у конфігурації для ESP32 активовано 64. Це було досягнуто шляхом збільшення фізичної кількості регістрів до 64, і організацією доступу до них за допомогою відображення віртуальних регістрів у фізичні, та керування цим процесом за допомогою спеціального реєстру. Регістр лічильника програми (PC) містить адресу наступної інструкції для виконання. Він оновлюється автоматично під час виконання інструкцій. Загалом використовується більше 200 регістрів спеціального призначення, а в залежності від конфігурації, можуть додаватися додаткові регістри.

Набір інструкцій Xtensa є результатом масштабних досліджень правильного балансу функцій для найкращого задоволення потреб ринку вбудованих процесорів. Він запозичує найкращі функції інших архітектур, а також пропонує власні інновації ISA.

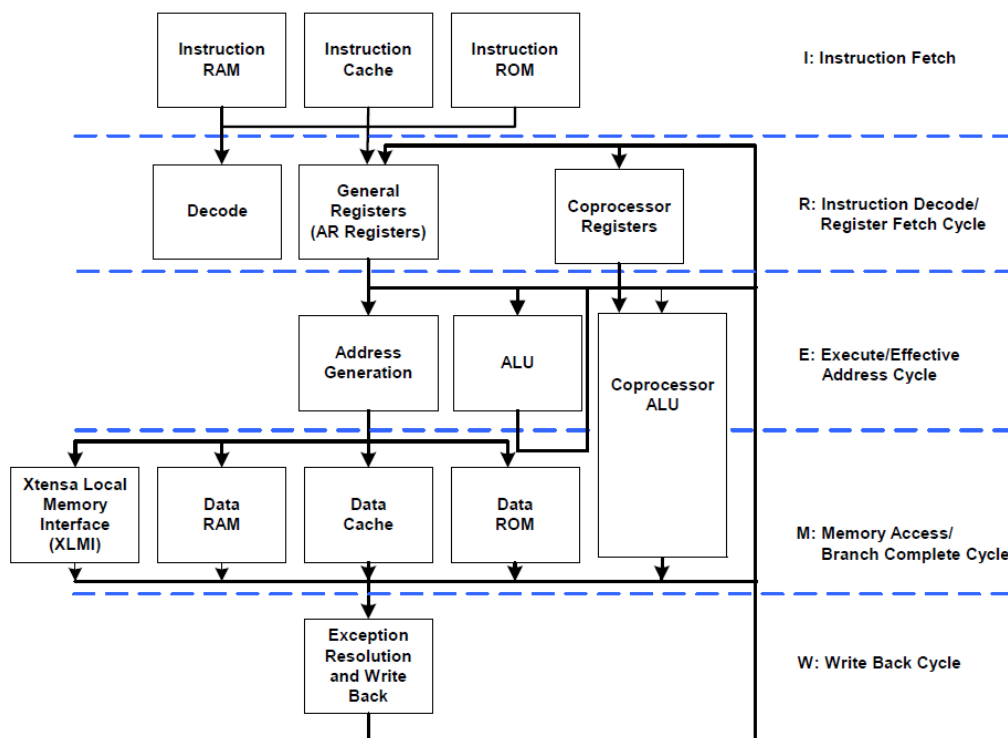


Рисунок 1 – Приклад впровадження 5-стадійного конвеєру Xtensa

Хоча Xtensa ISA більшість своїх функцій запозичує від RISC, вона зосереджена на областях, в яких старіші CISC-архітектури були найсильнішими, таких як компактний код. Ядро Xtensa ISA реалізовано як набір 24-бітних інструкцій, які виконують 32-бітні операції. Ширина інструкцій була обрана, головним чином, з урахуванням економії розміру коду. Самі інструкції були обрані за їхню корисність у широкому спектрі вбудованих програм [3]. У різних конфігураціях доступні також спеціальні інструкції для доступу до периферійних пристроїв через регістри вводу-виводу (IO register space), а також інструкції керування кешем, синхронізації та енергоспоживанням (наприклад, WAITI – очікування переривання). Завдяки розширенню TIE (Tensilica Instruction Extension) Espressif Systems змогли додати до ядра підтримку таких інструкцій, як AES, SHA, RSA, що значно пришвидшує криптографічні операції. У ESP32 ядро LX6 працює з інтегрованою кеш-пам'яттю, зовнішньою флеш-пам'яттю, та підтримує керування живленням з переходом у режим

глибокого сну (deep sleep). Також мікроконтролер має реалізацію апаратного сторожового таймера, RTC (Real-time clock), Wi-Fi та Bluetooth, що реалізовані як периферійні блоки з доступом до регістрів.

Ядро Xtensa LX6 демонструє поєднання ефективності RISC-підходу з можливістю адаптації під завдання конкретного пристрою, а його особливості дозволяють не лише вивчати принципи роботи процесора, але і виконувати різноманітні робочі задачі.

Розглянемо для порівняння архітектуру мікропроцесора ARM Cortex-M3. ARM Cortex-M3 – це 32-розрядне ядро, створене компанією ARM Holdings спеціально для використання у вбудованих системах з високими вимогами до енергоефективності, швидкодії та простоти програмування. Воно є представником сімейства ARMv7-M, що орієнтоване на застосування у мікроконтролерах середнього класу. Cortex-M3 був основою для популярних мікроконтролерів, таких як STM32F1, LPC1768, EFM32 та інших, і широко використовується в промисловій автоматизації, сенсорах, побутовій електроніці та медичних пристроях. Основні характеристики ARM Cortex-M3 згідно з [4]:

- 32-розрядна гарвардська архітектура з модифікованим доступом до пам'яті;
- Робота з 16 або 32-бітовими інструкціями завдяки Thumb-2;
- Наявність 13 регістрів загального призначення, регістрів стека, регістру лічильника програми і регістра стану;
- Час виконання більшості інструкцій – один такт (один цикл);
- Вбудований модуль керування перериваннями NVIC (Nested Vectored Interrupt Controller) з підтримкою до 240 зовнішніх переривань;
- Сумісність з інструментами ARM (Keil, IAR, CMSIS);
- Робоча частота – до 100 МГц у більшості реалізацій.

Cortex-M3 має чітко визначену регістрову модель:

- R0-R12 – регістри загального призначення;

- SP (R13) – реєстр стека, який може бути основним (MSP) або процесним (PSP);
- LR (R14) — реєстр повернення (link register);
- PC (R15) — лічильник програми;
- xPSR — реєстр стану, який поєднує APSR (статус арифметичних операцій), IPSR (стан переривань) та EPSR (стан виконання).

Cortex-M3 використовує набір інструкцій Thumb-2 – це розширення 16-бітного набору Thumb до підтримки повноцінних 32-бітових інструкцій. Основні групи інструкцій включають: арифметичні і логічні інструкції, інструкції завантаження і збереження даних, інструкції розгалуження та викликів функцій, а також інструкції керування станом та перериваннями. На відміну від мікропроцесорів серії Xtensa, ядра Cortex-M3 мають 3-стадійний конвеєр, схему роботи якого можна побачити на рисунку 2.

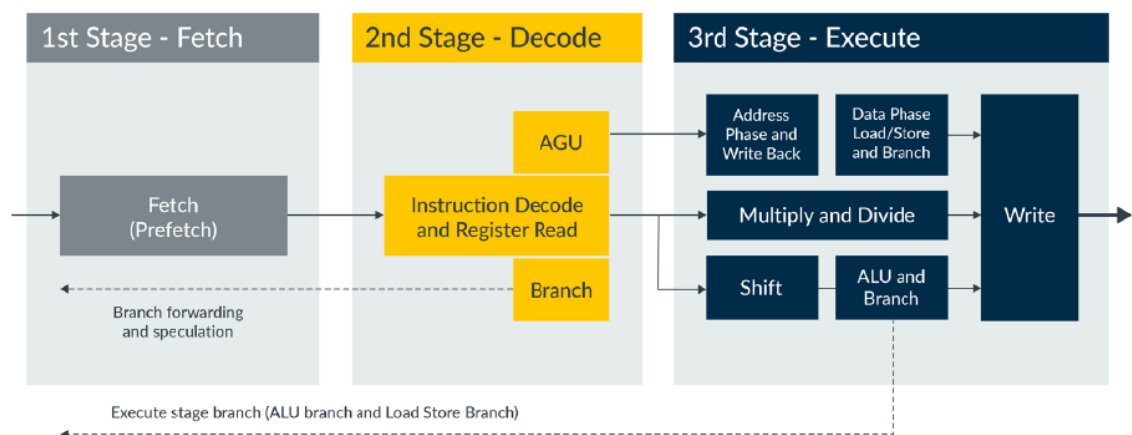


Рисунок 2 – Схема роботи 3-стадійного конвеєру Cortex-M3

Cortex-M3 має вбудований NVIC – апаратний контролер векторних переривань з підтримкою пріоритетів. При обробці переривань автоматично зберігається мінімальний контекст (R0-R3, R12, LR, PC, xPSR) у стек. Це дозволяє обробляти події з мінімальними затримками. Крім того, передбачена підтримка tail-chaining – автоматичного переходу до наступного переривання

без повернення до основного коду. Окрім цього, Cortex-M3 має низку унікальних апаратних можливостей:

- Підтримка бітових операцій без потреби у маскуванні (bit-banding – доступ до окремих бітів в SRAM/периферії через псевдоадреси);
- Підтримка управління живленням через інструкції WFI, WFE;
- Вбудований системний таймер SysTick для використання в RTOS;
- Визначене машинне виключення при спробах доступу до недоступної пам'яті, ділення на нуль тощо;
- Можливість використання альтернативного стеку (PSP) в RTOS.

Мікропроцесор ARM Cortex-M3 поєднує високу ефективність виконання інструкцій, просту регістрову модель та потужні засоби для роботи з перериваннями. Його набір інструкцій Thumb-2 дозволяє зберігати компактність коду, що робить його ідеальним для вбудованих систем.

Cortex-M3 – це приклад класичної архітектури з фокусом на зручність використання, низьке енергоспоживання та високу надійність, яка стала стандартом у сучасних мікроконтролерах.

І наостанок розглянемо мікропроцесори із архітектурою Atmel AVR, які є складовою частиною 8-бітного мікроконтролера ATmega328P. Мікроконтролер ATmega328P є одним із найпопулярніших представників 8-бітних мікроконтролерів на базі ядра AVR5, розроблених компанією Atmel (нині Microchip Technology). Він широко використовується в домашніх проєктах, освітніх платформах (зокрема Arduino Uno) та вбудованих системах із невисокими вимогами до обчислювальних потужностей. Ядро AVR5 реалізує спрощену RISC-архітектуру, оптимізовану для високої ефективності виконання інструкцій при малій тактовій частоті. Основні характеристики ядра AVR5 у ATmega328P:

- 8-розрядна гарвардська архітектура з окремими шинами для інструкцій та даних;
- 32 регістри загального призначення;

- Підтримка одноктактного виконання більшості інструкцій;
- Наявність 16-бітної шини адреси яка дозволяє адресувати до 64 КБ програмної пам'яті;
- Підтримка до 131 інструкції з фіксованою довжиною (16 або 32 біти);
- Програмована флеш-пам'ять обсягом 32 КБ;
- Частота до 20 МГц (в ATmega328P – до 16 МГц);
- Вбудований контролер переривань, таймери, UART, SPI, I2C, АЦП та ін.

Ядро має 32 універсальні регістри (R0-R31), які є частиною АЛП і доступні для всіх арифметичних, логічних та копіювальних операцій. Три пари регістрів (R26-R31) можуть утворювати 16-бітні адресні регістри, що використовуються для непрямої адресації пам'яті. Окрім цього присутній регістр стану SREG який містить сервісні прапорці станів, а також і інші регістри. Інструкційний набір AVR5 – це компактний RISC-набір з фіксованою довжиною інструкцій у 16 або 32 біти. Основні групи інструкцій складають: арифметичні та логічні інструкції, інструкції передачі даних, інструкції розгалуження, умовні інструкції, та інструкції керування перериваннями та станом. Більшість інструкцій виконуються за один такт, що дозволяє досягати високої ефективності навіть при низьких частотах.

У складі мікроконтролера ATmega328P ядро AVR5 поєднується з низкою периферій: 3 таймери, АЦП з 10-бітною роздільною здатністю, UART, SPI, I2C для обміну даними, вбудованим генератором, та пам'яттю SRAM розміром 2 КБ і постійною пам'яттю розміром 32 КБ. Подібна реалізація є прикладом класичної 8-бітної архітектури з ефективним RISC-дизайном. Незважаючи на обмеженість ресурсів, воно забезпечує швидке виконання коду, гнучку роботу з периферією та стабільну обробку подій у реальному часі. Простота архітектури та доступна ціна зробили його вибором №1 для навчання, розробки прототипів та вбудованих систем.

Проаналізовані мікропроцесори – Xtensa LX6, ARM Cortex-M3 та Atmel AVR5 – демонструють різні підходи до побудови обчислювальних систем, орієнтованих на вбудовані застосування. Архітектура Xtensa LX6 вирізняється високим рівнем налаштувань під конкретного замовника і під конкретну задачу, підтримкою розширюваного набору інструкцій та ефективною роботою з паралельними обчисленнями і периферією. Cortex-M3 від ARM демонструє універсальність, низьке енергоспоживання та високу інтеграцію з розвинутою інфраструктурою програмування, а також чудово підходить для задач які потребують виконання роботи у режимі реального часу. AVR-архітектура, попри свою простоту та 8-бітну розрядність, залишається популярною в освітньому середовищі та для невеликих проєктів через свою зрозумілість, надійність та велику спільноту. Зіставлення цих мікропроцесорів дозволяє краще зрозуміти еволюцію їх архітектури, компроміси між продуктивністю, енергоефективністю, складністю реалізації та гнучкістю. Кожен з розглянутих процесорів є яскравим представником свого класу і показує, як архітектурні рішення впливають на сферу застосування пристрою. Як наслідок, розуміння архітектурних особливостей різних мікропроцесорів дає змогу не лише краще адаптувати програмне забезпечення до апаратної платформи, а й свідомо обирати оптимальне рішення для конкретних завдань з огляду на ресурсні обмеження, швидкодію та функціональні можливості.

Поняття емуляції та віртуалізації обчислювальних процесів

У сучасних обчислювальних системах все частіше використовуються методи емуляції та віртуалізації, що дозволяють підвищити ефективність, гнучкість і універсальність програмно-апаратного забезпечення. Обидва поняття мають спільну ідею – створення абстрактного середовища, яке імітує роботу певного обладнання або програмної платформи, однак реалізуються різними способами і для різних цілей.

Емуляція – це процес, при якому одна система (емулятор) імітує поведінку іншої системи на рівні апаратного або програмного забезпечення. Основна мета емуляції – відтворити функціонування цільової архітектури або пристрою настільки точно, щоб програмне забезпечення для цієї цільової платформи працювало без змін. Типовими прикладами емуляції є:

- емулятори старих ігрових приставок (наприклад, NES, Sega) на сучасних комп'ютерах;
- емулятори мікроконтролерів або процесорів (наприклад, QEMU, Keil uVision, AVR Simulator);
- налагоджувальні середовища, які дозволяють перевіряти виконання коду для мікропроцесора без фізичного пристрою.

Емуляція часто вимагає значних обчислювальних ресурсів, оскільки кожна інструкція цільової архітектури інтерпретується або транслюється в реальному часі на хост-системі. Проте ця технологія має ряд переваг, а саме дозволяє налагодження та тестування без реального обладнання, а також дає можливість збереження застарілих або недоступних систем.

Окрім емуляції, існує поняття віртуалізації. Віртуалізація – це технологія, яка дозволяє створювати ізольовані обчислювальні середовища (віртуальні машини) на основі однієї фізичної апаратної платформи. У такому середовищі віртуальні машини можуть запускати власні операційні системи, працюючи паралельно одна з одною. Головна відмінність від емуляції полягає в тому, що віртуалізація зазвичай не імітує іншу апаратну архітектуру, а працює на тій же, що й хост-система (наприклад, x86 на x86). Віртуалізація передбачає частковий або повний розподіл ресурсів (CPU, RAM, HDD, мережа) між віртуальними середовищами. Віртуалізація поділяється на декілька типів:

- Апаратна (hardware-assisted) – використовує підтримку віртуалізації на рівні процесора (Intel VT-x, AMD-V);
- Паравіртуалізація – гостьова операційна система адаптована до віртуального середовища (наприклад, Xen);

- Повна віртуалізація – гостьова операційна система не знає, що працює у віртуальному середовищі (наприклад, VMware, VirtualBox).

Типовими прикладами застосування віртуалізації є:

- об'єднання декількох серверів в один фізичний пристрій;
- запуск декількох операційних систем на одному ПК для тестування;
- створення ізольованих середовищ для безпечного запуску програм;
- організація хмарних обчислень.

Таким чином, емуляція та віртуалізація – це дві важливі технології, що дозволяють абстрагуватися від фізичної платформи та забезпечувати гнучкість у розробці, тестуванні та використанні програмного забезпечення. Емуляція частіше застосовується у вбудованих системах, де потрібно повторити поведінку конкретного мікроконтролера або процесора, тоді як віртуалізація забезпечує масштабування та ефективне використання обчислювальних ресурсів у великих інформаційних системах. Розуміння їхніх переваг та обмежень є ключовим для вибору відповідного підходу в розробці складних цифрових систем.

Призначення та функції програм-налагоджувачів

У процесі розробки програмного забезпечення, особливо для низькорівневих або вбудованих систем, неминуче виникають помилки – як логічні, так і синтаксичні. Для виявлення, аналізу та виправлення таких помилок використовуються програми-налагоджувачі (debuggers). Це спеціальні інструменти, які дозволяють розробнику контролювати виконання програми на різних етапах, досліджуючи її внутрішній стан у реальному часі.

Налагоджувач (debugger) – це програмне забезпечення, яке дозволяє крок за кроком відстежувати виконання програми, переглядати та змінювати значення змінних, регістрів, пам'яті, керувати точками зупину (breakpoints) і виявляти причини помилок у коді. Налагоджувачі можуть бути:

- вбудованими в інтегровані середовища розробки (наприклад, Keil, Atmel Studio, Visual Studio, Eclipse);
- окремими програмами, такими як GNU Debugger, LLDB, OllyDbg;
- апаратними, які взаємодіють із фізичним пристроєм через інтерфейси JTAG/SWD/ISP (наприклад, Atmel-ICE, ST-Link).

Налагоджувачі забезпечують широкий набір інструментів, які дозволяють не тільки знаходити помилки, а й аналізувати логіку роботи програми. Серед найбільш важливих функцій які зазвичай присутні у налагоджувачах, виділяють наступні:

1. Крокове виконання програми (step-by-step execution) – це дозволяє виконувати програму поетапно – по одній інструкції або рядку коду, а також дає змогу детально прослідкувати, як змінюється стан програми на кожному кроці.
2. Установка та обробка точок зупинки (breakpoints) – надає можливість вказувати на зупинку виконання програми в конкретному місці, а також керувати такими зупинками за допомогою певних умов.
3. Перегляд змінних і регістрів – дає можливість виведення вмісту змінних, регістрів процесора, стеку тощо.
4. Вивчення пам'яті (memory inspection) – інструмент, який надає доступ до будь-якої області оперативної пам'яті і дозволяє перегляд або зміну її вмісту, що може бути особливо корисно при роботі з вказівниками, буферами, масивами тощо.
5. Контроль виконання – дає можливість контролювати процес виконання програми, запускати, призупиняти, перезапускати програму з будь-якого місця.
6. Відлагодження переривань і викликів функцій – дозволяє відстежувати обробку апаратних переривань, перехід між підпрограмами та повернення до викликача.

7. Аналіз стеку викликів – надає інструмент для виведення послідовності функцій, що були викликані до поточного моменту, а також дає змогу зрозуміти логіку виконання при глибокій рекурсії або помилках у вкладених викликах.

8. Виведення асемблерного коду – для випадків низькорівневої налагоджувальної роботи надає можливість перегляду дизасембльованого коду разом із прив'язкою до машинних інструкцій.

Програми-налагоджувачі є ключовим інструментом у роботі програміста, що дозволяє не тільки виявляти та усувати помилки, але й глибше розуміти логіку виконання програмного коду. Вони особливо важливі у розробці низькорівневого ПЗ, де критично важливо забезпечити стабільність та коректну взаємодію із апаратним забезпеченням. Знання принципів роботи налагоджувача та вміння його ефективно використовувати – одна з основ професійної підготовки розробника мікропрограм.

Порівняльний аналіз існуючих емуляторів та налагоджувачів

Для виконання порівняльного аналізу, розглянемо для початку декількох представників емуляторів та налагоджувачів. У даному розділі для порівняльного аналізу було обрано по три варіанти програмного забезпечення. А саме, серед емуляторів: QEMU (Quick EMUlator), DOSBox, та FCEUX. В свою чергу серед налагоджувачів буде виконано порівняльний аналіз серед: GDB (GNU Debugger), AFD (Advanced Full-screen Debugger), та Turbo Debugger.

QEMU (Quick EMUlator) – це універсальний емулятор та віртуалізатор машин з відкритим кодом [6]. Його активно використовують для запуску та тестування операційних систем, розробки прошивок для вбудованих пристроїв, досліджень безпеки та створення віртуальних середовищ. QEMU підтримує широкий спектр архітектур, таких як: x86, x86-64, ARM, RISC-V, PowerPC, MIPS, SPARC та інші. Це робить його універсальним інструментом для крос-

платформної розробки. Використання даного емулятора доступно у декількох режимах роботи, а саме:

- Системна емуляція (System Emulation) – емуляція повної машини, включаючи процесор, оперативну пам'ять та периферійні пристрої. У цьому режимі можливий запуск повноцінної гостьової операційної системи на віртуальній апаратній платформі.
- Прискорена віртуалізація – у разі наявності гіпервізора (наприклад, KVM, Xen або Hypervisor.Framework), QEMU може делегувати виконання інструкцій безпосередньо хост-процесору, що значно підвищує продуктивність порівняно з повною емуляцією.
- Емуляція лише процесора (User-mode Emulation) – дозволяє запускати окремі програми, скомпільовані для іншої архітектури, без потреби емуляції всієї операційної системи.

Серед особливостей даного емулятора можна виділити наступне:

- Підтримка знімків пам'яті (snapshots);
- Можливість підключення GDB для налагодження;
- Віртуальна мережа, серійні порти, USB;
- Вивід трасування інструкцій та логування подій.

DOSBox – це легкий та зручний емулятор MS-DOS, орієнтований передусім на запуск старих ігор і програм DOS-епохи (1980–1990-х років). Зараз його часто використовують для збереження доступу до програмного забезпечення, яке більше не працює на сучасних операційних системах. Серед його функцій та особливостей можна виділити наступне:

- Емуляція процесора x86 (до 486 включно);
- Підтримка VGA, VESA, EGA, CGA, Tandy-графіки;
- Емуляція звукових карт: Sound Blaster, Gravis Ultrasound тощо;
- Файлова система FAT через внутрішній доступ до директорій.
- Можливість монтованих віртуальних дисків;

- Налаштування швидкості процесора (кількість тактів/інструкцій в сек.);
- Проста система скриптів для автозавантаження програм;
- Підтримка геймпадів, мишки, MIDI, ігрових портів.
- Не потребує встановлення драйверів або DOS-ядра;
- Надзвичайно стабільна емуляція для ігор, що чутливі до таймінгів;
- Є режим трасування команд (але без повноцінного відлагоджувача).

Такий емулятор ідеально підходить для відтворення "ретро"-програм. Завдяки точній емуляції графіки та звуку, DOSBox став стандартом де-факто для старих ігор DOS.

FCEUX – один з найпопулярніших емуляторів класичної 8-бітної консолі Nintendo Entertainment System (NES). Він розроблений не тільки для гри, а й для дослідження ROM-файлів, зворотньої інженерії, відлагодження та розробки власних ігор. Він виконує емуляцію наступних апаратних пристроїв:

- CPU: MOS 6502 (спрощена версія без BCD);
- PPU: графічний процесор з підтримкою спрайтів, фонових плиток та ефектів прокрутки;
- APU: 5-канальний звук;
- Контролери: емуляція ігрових контролерів, можливість додати віртуальних гравців.

Серед його функціоналу можна виділити наступне:

- Вбудований дизасемблер, який дозволяє переглядати машинний код в реальному часі;
- Інтерфейс вікна пам'яті, який дає можливість спостереження за змінними та структурами;
- Аналіз виводу зображення покроково;
- Керування виконанням коду за допомогою точок зупинки, за адресою, записом або читанням;

- Вбудовані Lua-скрипти, які дають можливість автоматизації, створення тестів або ботів;
- Виконання повного логування виконання робочої програми з коментарями;
- Підтримка збережень, записів відео та кадрів;
- Тестові програми для перевірки точності емуляції;
- Зручний інтерфейс для новачків і професіоналів.

FCEUX поєднує в собі точну емуляцію NES з повноцінними інструментами налагодження, які дозволяють використовувати цей емулятор як навчальний та дослідницький засіб.

Далі наведемо порівняльну таблицю, для наглядного аналізу відмінностей між різними типами емуляторів.

Таблиця 1 – Порівняльна таблиця функціональності емуляторів.

<i>Критерій</i>	QEMU	DOSBox	FCEUX
<i>Підтримка архітектур</i>	x86, ARM, MIPS, RISC-V, та інші	x86 (MS-DOS)	8-бітний 6502
<i>Точність емуляції</i>	Висока	Висока для DOS	Висока для NES
<i>Наявність графічного інтерфейсу</i>	Обмежена	Так	Так
<i>Засоби налагодження</i>	Так (через GDB, логування)	Мінімальні	Розвинені вбудовані засоби
<i>Призначення</i>	Операційні системи,	Ігри/програми під DOS	Ретроігри, дослідження ROM

<i>Критерій</i>	QEMU	DOSBox	FCEUX
	прошивки, драйвери		
<i>Продуктивність</i>	Висока з KVM	Висока	Висока
<i>Ліцензія</i>	GNU GPL	GNU GPL	GNU GPL

Тепер виконаємо порівняльний аналіз програм-налагоджувачів, а саме: GDB (GNU Debugger), AFD (Advanced Full-screen Debugger), та Turbo Debugger.

GDB (GNU Debugger) – це потужний текстовий налагоджувач для програм, написаних мовами C, C++, Fortran, Ada та ін. Це один з найпопулярніших інструментів у середовищі Unix/Linux [5]. Його функціонал:

- Покрокове виконання (step-by-step) на рівні інструкцій або рядків коду;
- Breakpoints – точки зупинки за адресою, функцією, умовою;
- Перегляд і зміна змінних, регістрів, пам'яті;
- Відлагодження зворотного ходу (reverse debugging);
- Можливість запуску програми "від початку до точки зупину";
- Робота з вмістом пам'яті робочої програми після виникнення помилок у роботі;
- Підключення до віддалених систем (через GDB server);
- Підтримка багатопотокових і багатопроцесорних програм;
- Інтеграція з емуляторами (наприклад, QEMU) та IDE (Eclipse, VSCode тощо).

Із особливостей можна виділити наступне:

- Відсутній графічний інтерфейс користувача, а вся робота із програмою відбувається через термінал;

- Присутній функціонал написання та використання скриптів;
- Часто використовується в автоматизованому тестуванні.

Підтримка величезної кількості архітектур, можливість роботи з вбудованими пристроями, віддаленим налагодженням, детальний контроль над усім процесом виконання – все це робить GDB стандартом для низькорівневого налагодження.

AFD – це спеціалізований налагоджувач, створений для зручної роботи з асемблерним кодом, особливо кодом, зібраним компілятором FASM (Flat Assembler). Він орієнтований на x86 і x86-64 архітектури та має зручний графічний інтерфейс. Серед його функціоналу можна виділити наступне:

- Покрокове виконання асемблерних інструкцій;
- Вікно реєстрів процесора та їх модифікація;
- HEX-вікно для перегляду стану даних у пам'яті;
- Автоматичне розпізнавання інструкцій FPU/SIMD;
- Декодування інструкцій і кольоровий дизасемблер;
- Вікно стеку з автоматичною аналітикою;
- Підтримка позначок (labels), переходів, графів викликів;
- Статичний аналіз коду.

Серед особливостей даного налагоджувача можна виділити той факт, що він орієнтований на роботу з низькорівневим машинним кодом, та має високу точність відображення структури виконання заданої програми. Має мінімум зайвих функцій, через що добре підходить для використання у навчальних цілях.

Turbo Debugger – класичний налагоджувач для DOS-програм, розроблений Borland і орієнтований на мови Pascal і C. Він працює у текстовому інтерфейсі, з підтримкою віконного режиму DOS. Серед його функціоналу можна виділити наступне:

- Налagodження програм у реальному DOS-середовищі;
- Перегляд вихідного коду разом із дизасемблером;

- Точки зупинки за кодом, змінними, умовами;
- Перегляд пам'яті, стеку, регістрів;
- Профілювання часу виконання;
- Покрокове виконання функцій;
- Простий відлагодочний скриптовий інтерфейс;
- Повна інтеграція із такими IDE як Turbo Pascal, Turbo C.

Нажаль, даний налагоджувач працює тільки у середовищі операційної системи DOS, та підходить лише для 16-бітних програм. Але через свою простоту він був еталоном для студентів та початківців у 90-х. Його простота і швидкість зробили його популярним серед розробників DOS-утиліт.

Наведемо порівняльну таблицю, для наглядного порівняння відмінностей у функціоналі та можливостях оглянутих налагоджувачів.

У ході аналізу трьох популярних емуляторів – QEMU, DOSBox та FCEUX – було виявлено, що кожен із них орієнтований на специфічні задачі та має власну область застосування. QEMU є універсальним системним емулятором з підтримкою багатьох архітектур і здатністю до повної віртуалізації.

Таблиця 2 – Порівняльна таблиця функціональності налагоджувачів.

<i>Критерій</i>	GDB	AFD	Turbo Debugger
<i>Графічний інтерфейс</i>	Немає	Так	Так
<i>Асемблер / дизасемблер</i>	Так	Так	Так
<i>Робота з пам'яттю</i>	Так, дозволяє і читання і запис	Так	Так
<i>Робота з регістрами</i>	Так	Так	Так
<i>Підтримка умовних breakpoints</i>	Так	Так	Так
<i>Підтримка скриптів та автоматизації</i>	Так	Автоматичне розпізнавання деяких інструкцій	Немає
<i>Робота з багатопотоковими програмами</i>	Так	Немає	Немає
<i>Підтримка операційних систем</i>	Підтримується Linux, macOS, Windows	Підтримується Windows	Підтримується виключно DOS

Він активно використовується для емуляції сучасних ОС та налагодження embedded-систем. DOSBox ідеально підходить для запуску старих DOS-

додатків, що робить його незамінним інструментом для ретро-програмного забезпечення. Він має просту конфігурацію та високий рівень сумісності з класичними іграми та утилітами. FCEUX, у свою чергу, демонструє вузькоспеціалізовану емуляцію конкретної ігрової консолі (NES), пропонуючи широкий спектр інструментів для аналізу, модифікації ROM-ів і запису ігрового процесу.

Щодо налагоджувачів – GDB, AFD та Turbo Debugger – вони також відображають широкий спектр підходів до налагодження програм. GDB – потужний, універсальний інструмент, який забезпечує гнучке й глибоке налагодження як користувацьких, так і системних програм, включаючи підтримку віддаленої роботи та багатопотоковості. AFD більше підходить для низькорівневої роботи з асемблером, пропонуючи інтуїтивно зрозумілий графічний інтерфейс і візуальний аналіз машинного коду. Turbo Debugger, хоча вже є застарілим, усе ще виконує важливу роль у розумінні принципів налагодження класичних DOS-додатків, і часто використовується в освітніх цілях або під час зворотної розробки старого ПЗ.

Оцінка можливості застосування існуючих рішень у навчальних цілях

У процесі аналізу існуючих емуляторів та налагоджувачів було виявлено, що більшість сучасних інструментів створювались для професійного використання або мають вузьке спеціалізоване призначення. Такі засоби, як QEMU чи GDB, хоча й володіють потужними можливостями, орієнтовані на досвідчених користувачів і потребують глибоких знань у налаштуванні та роботі з ними. Інші рішення, як Turbo Debugger чи AFD, є застарілими, мають обмежену підтримку сучасних платформ або не забезпечують належного рівня гнучкості.

Особливу увагу заслуговує проблема застосування цих інструментів у навчальному процесі. Для студентів, які лише починають знайомитися з архітектурою мікропроцесорів і принципами виконання команд, більшість

доступних рішень виглядають надмірно складними. Вони або перевантажені професійними функціями, або взагалі не дозволяють користувачу контролювати процес виконання інструкцій покроково. Крім того, немає єдиного зручного інтерфейсу для візуального представлення регістрів, пам'яті, команд та результатів виконання.

У результаті можна зробити висновок, що наразі не існує простого, інтуїтивного та універсального налагоджувача, який дозволяв би студентам симулювати виконання інструкцій умовного процесора в навчальних цілях, із можливістю відслідковувати зміну вмісту регістрів, адресної пам'яті та переходи по інструкціях. Це створює потребу в створенні нового інструменту, який буде орієнтований саме на освітнє застосування.

Оцінка та обґрунтування теми дипломного проєкту

На підставі проведеного аналізу та оцінки поточних програмних засобів було визначено наявність важливого недоліку: відсутність простого, доступного та наочного інструменту для симуляції та налагодження програм під умовні процесори з RISC-архітектурою. Існуючі емулятори або є надто загальними, або перевантажені функціоналом, що ускладнює їх використання студентами та початківцями.

У зв'язку з цим було обрано тему дипломного проєкту "Спеціалізована комп'ютерна система для емуляції і налагодження програм", основною метою якої є розробка базового налагоджувача для умовного процесора з архітектурою типу RISC. Проєкт передбачає створення програмного середовища, що дозволяє в інтерактивному режимі відслідковувати виконання інструкцій, зміну значень регістрів, а також працювати зі спрощеною моделлю пам'яті.

Система буде реалізована з обмеженою, але достатньою функціональністю для ілюстрації базових понять машинного коду та принципів роботи процесора. Таким чином, створюється універсальний та зручний інструмент, призначений для використання у навчальному процесі, з

можливістю подальшого розширення функціональності відповідно до потреб користувачів.

АЛГОРИТМ І РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Загальна структура програми

Проектування програмного забезпечення вимагає чіткої організації коду, що дозволяє забезпечити його зручність у супроводі, тестуванні, масштабуванні та повторному використанні. Одним з ключових принципів, що застосовується при створенні складних програм, є модульність.

Модульність – це принцип поділу програми на відносно незалежні частини (модулі), кожна з яких виконує окрему функцію. Кожен модуль має чітко визначену відповідальність, мінімальну взаємозалежність із іншими частинами програми та відкритий інтерфейс взаємодії. Основними перевагами модульного підходу є:

- Зрозуміла структура коду, де кожен компонент відповідає лише за одну частину логіки.
- Можливість ізоляції та тестування модулів окремо.
- Спрощене внесення змін до окремих компонентів без необхідності переписувати всю програму.
- Розширюваність: до вже наявної програми легко додавати нові функції.

При побудові модульної структури зазвичай виділяють логіку обробки даних, інтерфейс користувача та окремо реалізують алгоритмічну частину. Така розділеність дозволяє реалізувати принцип інкапсуляції, коли деталі реалізації приховані від зовнішніх компонентів, і доступ до них відбувається лише через інтерфейси.

Розроблене програмне забезпечення реалізовано за принципами модульного об'єктно-орієнтованого підходу з використанням мови C++ та графічної бібліотеки Qt. Програма складається з трьох основних функціональних компонентів (модулів):

1. ProcessorEmulator – ядро логіки симуляції
Відповідає за управління регістрами, пам'яттю, виконання інструкцій, обробку

станів (завершення, помилки) та керування лічильником команд. Містить функціональність для завантаження програми з текстового файлу та покрокового її виконання.

2. `InstructionSet` – набір реалізацій інструкцій. Включає методи для обробки кожної окремої інструкції (`SET`, `ADD`, `LOAD`, `BEQ` тощо). Цей клас взаємодіє з `ProcessorEmulator` через вказівник, що передається в конструкторі, та безпосередньо змінює його внутрішній стан під час виконання команд.

3. `MainWindow` – графічний інтерфейс користувача (GUI). Реалізований за допомогою засобів бібліотеки Qt. Клас `MainWindow` забезпечує:

- Відображення регістрів та пам'яті;
- Завантаження програми з файлу;
- Керування процесом виконання (кнопки "Step", "Run", "Reset");
- Виведення статусу та помилок;
- Покрокове візуальне оновлення інтерфейсу відповідно до виконаних команд.

На представленій нижче діаграмі (рис. 3) зображено взаємозв'язок між основними модулями програми. Ієрархічна структура починається з головної функції `main()`, яка ініціалізує графічний інтерфейс `MainWindow`. У свою чергу, `MainWindow` створює екземпляр ядра програми — класу `ProcessorEmulator`, який безпосередньо керує виконанням інструкцій за допомогою модуля `InstructionSet`. Така структура забезпечує чіткий розподіл обов'язків між компонентами та полегшує супровід і розширення програмного забезпечення.

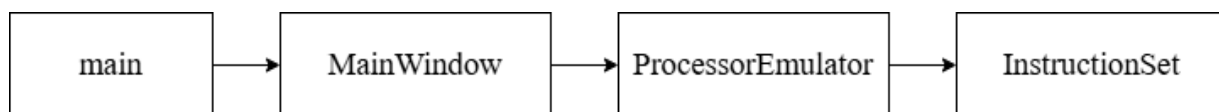


Рисунок 3 – Діаграма зв'язків модулів

Загальна структура програми чітко поділена на функціональні блоки згідно з принципами модульності. Взаємозв'язки між модулями реалізовані через об'єктно-орієнтовані механізми взаємодії, що забезпечує надійність, масштабованість та зручність у підтримці. Такий підхід дозволяє з легкістю розширити програму новими інструкціями, типами регістрів або навіть новими архітектурами процесорів у майбутньому.

Формат подання вхідної програми та інструкцій

Однією з ключових особливостей будь-якої системи емуляції або налагодження програмного коду є формат подання вхідної програми, яка підлягає виконанню. У сучасних мікропроцесорах програми зазвичай представлені у вигляді бінарного машинного коду, що завантажується в пам'ять пристрою та виконується процесором безпосередньо. Проте для потреб налагодження, тестування, а особливо – у навчальних цілях, частіше використовується зрозумілий людині текстовий формат, зручний для аналізу та вивчення.

У процесорах архітектури RISC програма зазвичай складається з набору простих інструкцій фіксованого або близького до фіксованого формату. Такі інструкції зазвичай мають коротку довжину (наприклад, 16, 24 або 32 біти) і включають код операції (opcode) та операнди – регістри, константи або адреси пам'яті.

З огляду на навчально-дослідницький характер розробленого ПЗ, було прийнято рішення використовувати простий текстовий формат з розширенням .txt, що дозволяє легко писати, читати, налагоджувати та редагувати програму вручну. Програма-емулятор працює з таким форматом, сприймаючи кожен рядок файлу як одну інструкцію умовної асемблерної мови. Це дозволяє студентам, викладачам або дослідникам безпосередньо спостерігати за виконанням кожної інструкції, а також краще розуміти послідовність дій у програмі.

Інструкції у вхідному файлі мають асемблероподібний синтаксис і подаються у вигляді:

ІНСТРУКЦІЯ ОПЕРАНД1, ОПЕРАНД2, ...

Порожні рядки і рядки, що починаються з «#», вважаються коментарями і ігноруються під час обробки, а операнди розділяються комами та пробілами. При зчитуванні файлу інструкції автоматично перетворюються у верхній регістр для уніфікації синтаксису.

Приклад вхідного файлу:

Простий приклад програми

SET R1, 10

SET R2, 20

ADD R3, R1, R2

STORE R3, 50

LOAD R4, 50

BEQ R3, R4, 9

NOP

JMP 2

У наведеному прикладі демонструється типова програма, яка виконує арифметичну операцію, працює з пам'яттю, а також містить умовний та безумовний перехід.

Алгоритм виконання інструкцій емулятором

Емулятори процесорів призначені для відтворення поведінки апаратного забезпечення на програмному рівні. Основною метою емуляції є відтворення результатів виконання інструкцій, а не обов'язково точне повторення всіх апаратних процесів. Це дозволяє спростити реалізацію емулятора, зосередившись на функціональній точності, що особливо важливо в навчальних та дослідницьких цілях.

Базовий цикл виконання інструкцій в емуляторах зазвичай включає три основні етапи:

1. Зчитування (Fetch): Отримання наступної інструкції з пам'яті за адресою, вказаною лічильником команд (Program Counter, PC).
2. Декодування (Decode): Визначення типу інструкції та її операндів.
3. Виконання (Execute): Здійснення операції, передбаченої інструкцією, з відповідним оновленням стану регістрів, пам'яті та лічильника команд.

Цей цикл, відомий як Fetch-Decode-Execute[7], є основою для багатьох емуляторів, включаючи ті, що моделюють прості процесори, такі як 6502 або MIPS.

У більшості емуляторів основа виконання інструкцій зводиться до класичної схеми Fetch-Decode-Execute, що реалізується у вигляді окремої функції або циклічної конструкції в коді. Наприклад, у емуляторі Game Boy функція `exec_next()` виконує чергову інструкцію, починаючи з ініціалізації лічильника команд, зчитування коду з пам'яті, його декодування та виконання [8].

Такий підхід відображає принцип циклічного інтерпретатора, що поетапно моделює поведінку процесора з високою точністю. Альтернативно, в окремих системах застосовують трасяцію у проміжний байткод або емуляцію на рівні мікрооперацій, однак такі методи частіше використовуються для продуктивних або складних систем. У загальному випадку, схема Fetch-Decode-Execute залишається базовою моделлю для більшості простих та навчальних емуляторів.

У розробленому емуляторі використовується спрощений варіант циклу Fetch-Decode-Execute, адаптований для умовного RISC-процесора з фіксованим набором інструкцій. Основні етапи виконання інструкцій реалізовані наступним чином:

1. Зчитування інструкції: Метод `step()` класу `ProcessorEmulator` отримує поточну інструкцію з пам'яті, використовуючи значення лічильника команд (PC).

2. Декодування та виконання: Зчитана інструкція передається до методу `execute()` класу `InstructionSet`, який визначає тип інструкції та викликає відповідну функцію для її виконання.

3. Оновлення стану: Після виконання інструкції оновлюються відповідні регістри, пам'ять та лічильник команд, підготовлюючи емулятор до наступного циклу.

Для кращого розуміння алгоритму роботи розробленої програми на рисунку 4 подано блок-схему виконання однієї інструкції. Дана схема відображає послідовність етапів: зчитування інструкції з пам'яті, її розбір, виконання та оновлення стану емулятора. У разі виникнення помилки виконання програма переходить у стан зупинки. Така блок-схема демонструє базову логіку, яка лежить в основі інтерпретації кожної інструкції в процесі симуляції. Такий підхід забезпечує чітку модульну структуру, де кожен компонент відповідає за окрему частину процесу емуляції, що сприяє легкості розширення та підтримки коду.

Обраний алгоритм виконання інструкцій був визначений з урахуванням наступних факторів:

- Простота реалізації: Спрощений цикл `Fetch-Decode-Execute` дозволяє зосередитися на функціональній точності емуляції без необхідності моделювання складних апаратних механізмів, таких як конвеєризація або динамічне планування інструкцій.
- Навчальна спрямованість: Такий підхід є ідеальним для освітніх цілей, оскільки дозволяє студентам краще зрозуміти базові принципи роботи процесора та взаємодії між його компонентами.
- Модульність: Розділення функціональності між класами `ProcessorEmulator` та `InstructionSet` сприяє гнучкості та масштабованості

програми, дозволяючи легко додавати нові інструкції або модифікувати існуючі.

- Зручність налагодження: Кожен етап виконання інструкції чітко визначений, що полегшує процес виявлення та виправлення помилок, а також аналізу поведінки емулятора.

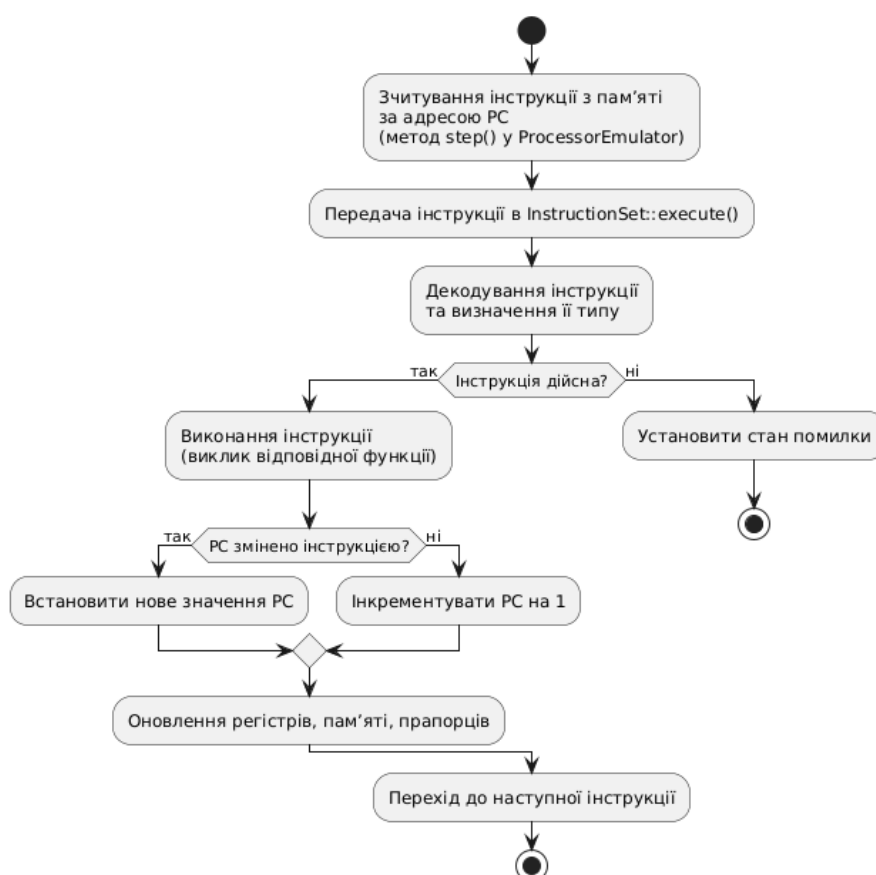


Рисунок 4 – Блок-схема алгоритму виконання інструкції

Таким чином, обраний алгоритм забезпечує баланс між простотою реалізації та функціональною повнотою, що робить його оптимальним для даного проекту.

Опис регістрів та реалізованих інструкцій

Процесори з архітектурою RISC (Reduced Instruction Set Computer) характеризуються простотою та симетричністю регістрів. Зазвичай вони мають від 16 до 32 (а іноді й більше) регістрів загального призначення (GPR – General Purpose Registers), які використовуються як для збереження тимчасових значень, так і для передавання операндів до інструкцій.

Окрім GPR, RISC-процесори можуть мати:

- PC (Program Counter) – регістр лічильника команд, що вказує на адресу наступної інструкції.
- SR/PSR (Status Register / Program Status Register) – регістр прапорців (Zero, Negative, Overflow, Carry тощо).
- SP (Stack Pointer) – регістр, що вказує на вершину стеку (не завжди присутній у спрощених реалізаціях).

Ці регістри використовуються переважно апаратно, але в більшості емуляторів частина з них моделюється програмно.

У RISC-процесорах формат інструкцій зазвичай має фіксовану довжину (наприклад, 32 біти) та чітку структуру. Такий підхід дозволяє легко декодувати інструкції та реалізувати конвеєризацию. У таблиці 3 перелічено типові інструкції в RISC-процесорах.

У розробленій програмі моделюється 8 регістрів загального призначення (R0–R7). Вони зберігаються у масиві `int registers[8]` у класі `ProcessorEmulator`. Доступ до них здійснюється за індексами за допомогою методів класу:

- `int getRegister(int index) const;`
- `void setRegister(int index, int value);`

Також реалізовано лічильник команд (PC) — змінна `programCounter`, яка оновлюється автоматично або вручну залежно від інструкції (наприклад, JMP чи BEQ).

Таблиця 3 – Типові інструкції у архітектурі RISC

Інструкція	Призначення
ADD	Додавання двох регістрів
SUB	Віднімання
AND	Побітове AND
OR	Побітове OR
MOV	Копіювання значення між регістрами
LOAD	Завантаження з пам'яті в регістр
STORE	Збереження з регістра в пам'ять
JMP	Безумовний перехід
BEQ	Умовний перехід (якщо значення рівні)
NOP	Порожня інструкція, яка нічого не виконує

Крім цього, підтримуються два прапорці стану:

- zeroFlag — встановлюється, якщо результат дорівнює нулю;
- negativeFlag — встановлюється, якщо результат від'ємний.

Ці прапорці оновлюються автоматично при записі у регістри.

Інструкції записуються у текстовому форматі, де кожен рядок містить одну інструкцію у вигляді:

ІНСТРУКЦІЯ ОПЕРАНД1, ОПЕРАНД2, ...

У програмі реалізовано 11 інструкцій, які відповідають базовому набору умовного RISC-процесора. Перелік всіх реалізованих інструкцій наведено у таблиці 4. Усі інструкції реалізовані як окремі методи у класі InstructionSet, з викликом через функцію execute() залежно від вхідного операційного коду.

Реалізована у програмі модель умовного RISC-процесора включає мінімально необхідний набір інструкцій та регістрів, достатній для демонстрації базових принципів виконання команд, роботи з пам'яттю та

умовного переходу. Наявність 8 регістрів загального призначення дозволяє ефективно виконувати арифметичні й логічні операції, а підтримка простих інструкцій з уніфікованим форматом значно полегшує аналіз і моделювання. Таким чином, реалізована система інструкцій і регістрів є достатньою та цілком придатною для розв'язання задач симуляції програмного коду в рамках навчального проєкту.

Таблиця 4 – Перелік всіх реалізованих інструкцій

Інструкція	Формат запису	Призначення
<i>LOAD</i>	LOAD REG, ADDR	Завантаження значення з пам'яті у регістр
<i>STORE</i>	STORE REG, ADDR	Запис значення з регістра у пам'ять
<i>SET</i>	SET REG, IMM	Запис константи у регістр
<i>MOV</i>	MOV REG1, REG2	Копіювання значення з одного регістра в інший
<i>ADD</i>	ADD REG1, REG2, REG3	Додавання значень двох регістрів, результат у третій регістр
<i>SUB</i>	SUB REG1, REG2, REG3	Віднімання значення REG3 від REG2, результат у REG1
<i>AND</i>	AND REG1, REG2, REG3	Побітова операція AND над REG2 і REG3, результат у REG1
<i>OR</i>	OR REG1, REG2, REG3	Побітова операція OR над REG2 і REG3, результат у REG1
<i>JMP</i>	JMP ADDR	Безумовний перехід на вказану адресу інструкції
<i>BEQ</i>	BEQ REG1, REG2, ADDR	Якщо REG1 дорівнює REG2 — перейти на адресу ADDR

Інструкція	Формат запису	Призначення
<i>NOP</i>	NOP	Інструкція, яка не виконує жодної дії (порожня операція)

Структура моделі пам'яті

У RISC-процесорах пам'ять традиційно організована як лінійний масив байтів або слів, адресований за допомогою цілих чисел. Характерною особливістю архітектури є розділення інструкцій та даних (так звана гарвардська архітектура), хоча в деяких випадках застосовується і уніфікований підхід (модель фон Неймана).

У класичних RISC-системах, таких як ARM, MIPS або RISC-V, робота з пам'яттю обмежена спеціальними інструкціями завантаження та збереження (LOAD, STORE), а всі інші операції виконуються лише над регістрами. Такий підхід спрощує конвеєризацію та пришвидшує виконання команд.

У більшості програмних емуляторів пам'ять моделюється як одновимірний масив (наприклад, `uint8_t memory[SIZE]` або `int memory[SIZE]`), де індекси відповідають адресам пам'яті, а значення – збереженим байтам або словам.

Існують різні підходи до моделювання пам'яті:

- Фіксована пам'ять – масив із заданим розміром. Перевага – простота реалізації. Недолік – обмежена гнучкість.
- Динамічна пам'ять – наприклад, `QMap<int, int>` або `std::unordered_map`, яка дозволяє адресувати великі області при економії пам'яті.
- Сторінкова модель – застосовується у складніших емуляторах (QEMU), що моделюють управління віртуальною пам'яттю.

У розробленому симуляторі реалізовано просту модель фіксованої пам'яті розміром 4000 байтів на 1000 елементів:

```
int memory[1000];
```

Кожна комірка пам'яті вважається словом (типу `int`), що дозволяє зберігати прості значення – адреси, дані, константи. Адресація відбувається за допомогою цілих чисел від 0 до 999. Доступ до пам'яті реалізований через методи класу `ProcessorEmulator`:

- `int getMemory(int address) const;`
- `void setMemory(int address, int value);`

Усі звернення до пам'яті перевіряються на коректність, тобто чи знаходиться адреса у межах допустимого діапазону ($0 \leq \text{address} < 1000$). Це дозволяє уникнути помилок доступу та забезпечити стабільність виконання.

Інструкції `LOAD` і `STORE` є єдиними засобами доступу до пам'яті, що відповідає принципам RISC-архітектури. Також можна зазначити, що обрана модель пам'яті повністю відповідає навчальному призначенню проєкту:

- Проста у реалізації та розумінні.
- Не потребує складних структур управління пам'яттю.
- Достатньо гнучка для демонстрації типових операцій збереження/завантаження.
- Підтримується візуальне відображення стану пам'яті у графічному інтерфейсі користувача (перші 50 комірок виводяться у таблиці).

У перспективі реалізацію пам'яті можна масштабувати до сторінкової або динамічної моделі, однак у поточному вигляді вона є оптимальною для початкового ознайомлення з принципами роботи RISC-процесора та системного програмного забезпечення.

Побудова графічного інтерфейсу користувача

Більшість сучасних емуляторів використовують графічний інтерфейс для полегшення взаємодії з користувачем, особливо у випадках, коли продукт орієнтовано на навчальні або дослідницькі цілі. Наприклад:

- QEMU здебільшого працює у консольному режимі, але має зовнішні GUI-оболонки, такі як AQEMU або QEMU Manager.

- DOSBox має мінімальний графічний інтерфейс і працює через вікно емуляції командного рядка.
- NES-емулятори (наприклад, FCEUX або Nestopia) зазвичай мають зручні вікна відлагодження, перегляду пам'яті та відеовиходу.
- Навчальні середовища, такі як MARS (для MIPS) чи Ripes (для RISC-V), мають візуальні панелі, які показують регістри, пам'ять і конвеєрну архітектуру. Такий підхід значно покращує засвоєння матеріалу.

У рамках реалізації даного програмного забезпечення для побудови графічного інтерфейсу було використано фреймворк Qt 6 – сучасну крос-платформову бібліотеку з відкритим вихідним кодом, яка дозволяє створювати графічні застосунки з використанням елементів управління, подій, сигналів і слотів. Qt підтримує:

- Побудову багатопанельних вікон.
- Підключення обробників подій.
- Створення таблиць, списків, діалогів.
- Розробку через Qt Designer або вручну на C++.

Qt добре підходить для швидкого створення освітніх та демонстраційних застосунків, особливо коли важлива візуалізація внутрішнього стану програми.

Графічна частина програми реалізована у класі MainWindow, який є головним вікном застосунку і успадковує базовий клас QMainWindow. У цьому класі створюються та керуються всі візуальні елементи інтерфейсу.

На рисунку 5 зображено графічний інтерфейс програми, що складається з панелі керування, таблиць регістрів і пам'яті, а також області для відображення інструкцій.

Інтерфейс організовано у вигляді двох панелей:

- Ліва – елементи керування (кнопки) та статуси.
- Права – таблиця регістрів, таблиця пам'яті та вікно інструкцій.

Інтерфейс розроблено з метою забезпечення зручного візуального контролю за виконанням програмного коду.

У таблиці 5 перераховані основні графічні елементи інтерфейсу користувача, реалізовані в межах головного вікна програми, разом із їхнім призначенням.

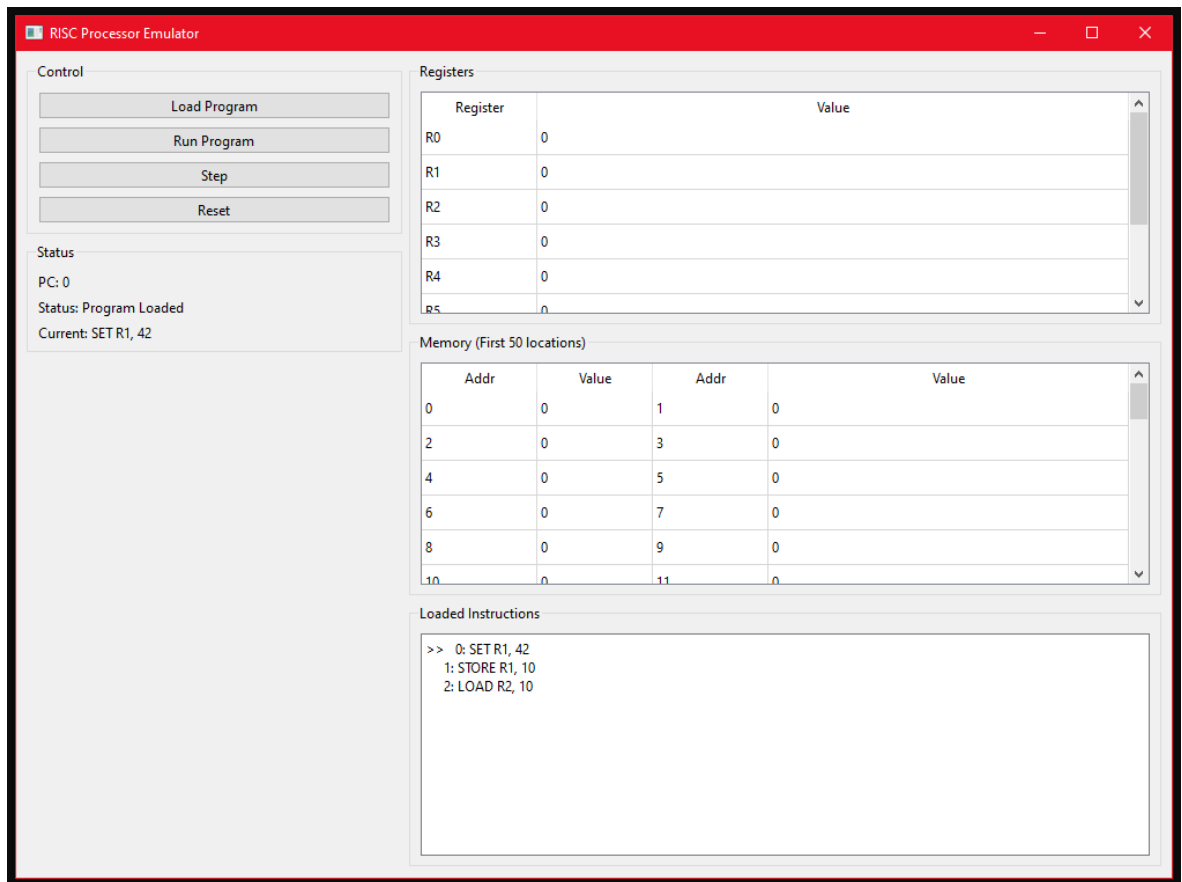


Рисунок 5 – Зображення реалізованого інтерфейсу

Таблиця 5 – Ключові елементи інтерфейсу

Елемент	Тип	Призначення
Кнопка "Load Program"	QPushButton	Завантаження програми з текстового файлу
Кнопка "Run Program"	QPushButton	Автоматичне виконання інструкцій до завершення
Кнопка "Step"	QPushButton	Покрокове виконання однієї інструкції

Елемент	Тип	Призначення
<i>Кнопка "Reset"</i>	QPushButton	Скидання стану процесора та пам'яті
<i>Таблиця регістрів</i>	QTableWidget	Відображає значення 8 регістрів загального призначення
<i>Таблиця пам'яті</i>	QTableWidget	Відображає перші 100 комірок пам'яті (по 4 на рядок)
<i>Вікно інструкцій</i>	QTextEdit	Виводить повний список інструкцій програми із зазначенням поточної
<i>Статусні мітки</i>	QLabel	Відображають статус виконання, поточну інструкцію, значення PC

Завдяки реалізованому підходу користувач може не тільки виконувати програму, але й спостерігати за змінами внутрішнього стану у реальному часі, що є особливо корисним у навчальному процесі.

Механізми покрокового виконання та оновлення пам'яті

Однією з ключових функцій реалізованого емулятора є можливість покрокового виконання інструкцій, що особливо важливо для аналізу та навчання. Це дозволяє користувачу поступово спостерігати за зміною стану регістрів і пам'яті після кожної виконаної інструкції. Механізм активується при натисканні кнопки "Step", яка прив'язана до слота MainWindow::stepProgram(). Цей слот викликає метод ProcessorEmulator::step(), що відповідає за обробку однієї інструкції програми. Логіка викликів виглядає так:

1. Користувач натискає кнопку Step.
2. Викликається метод MainWindow::stepProgram(), який:
 - перевіряє, чи завантажена програма та чи не завершено виконання,

- викликає emulator->step(),
- оновлює інтерфейс через updateDisplay() (вивід регістрів, пам'яті, інструкцій, статусу).

3. Метод ProcessorEmulator::step():

- зчитує поточну інструкцію з пам'яті програм (список інструкцій),
- виконує її через instructionSet.execute(),
- оновлює programCounter, якщо він не був модифікований вручну (наприклад, командою JMP або BEQ),
- встановлює відповідні прапорці (finished, error).

Цей механізм імітує цикл обробки команд процесором і дозволяє користувачу детально простежити, як виконуються інструкції на кожному кроці.

Модель пам'яті реалізована у вигляді масиву `int memory[1000]` усередині класу `ProcessorEmulator`. Всі звернення до неї відбуваються лише через відповідні інструкції, відповідно до принципів RISC-архітектури.

Оновлення пам'яті можливе лише через інструкції `LOAD` і `STORE`, які реалізовані в класі `InstructionSet`. Вони викликаються з методу `InstructionSet::execute()`, що обирає відповідну функцію на основі імені команди:

- Інструкція `LOAD REG, ADDR`:
 - зчитує значення з комірки пам'яті `memory[ADDR]`,
 - записує його у регістр `REG` через `emu->setRegister()`.
- Інструкція `STORE REG, ADDR`:
 - зчитує значення з регістра `REG` через `emu->getRegister()`,
 - записує його у пам'ять за адресою `ADDR` через `emu->setMemory()`.

Обидві функції (`getMemory`, `setMemory`) містять перевірку адреси на коректність, що запобігає виходу за межі пам'яті.

Після кожного кроку виконання пам'ять відображається у таблиці пам'яті графічного інтерфейсу, оновлення якої виконується через метод `MainWindow::updateMemory()`.

Таким чином, реалізований механізм покрокового виконання дозволяє послідовно виконувати інструкції, що значно полегшує процес аналізу. А доступ до пам'яті строго контролюється відповідними інструкціями, що забезпечує коректну поведінку моделі та відповідає принципам RISC-архітектури.

Опис основних класів, структур та логіки взаємодії

У розробленому програмному забезпеченні вся функціональність розподілена між трьома основними модулями (класами): `MainWindow`, `ProcessorEmulator` та `InstructionSet`. Кожен із них відповідає за окремий рівень логіки: інтерфейс користувача, логіка емулятора та виконання інструкцій відповідно.

Клас `MainWindow` є головним вікном програми, яке реалізує графічний інтерфейс користувача. Він відповідає за відображення стану регістрів, пам'яті, інструкцій, а також за обробку натискань на кнопки управління.

Основні поля:

- `QPushButton *loadButton, *runButton, *stepButton, *resetButton` – кнопки управління виконанням.
- `QTableWidget *registerTable, *memoryTable` – таблиці відображення регістрів і пам'яті.
- `QTextEdit *instructionDisplay` – вікно з інструкціями.
- `QLabel *statusLabel, *pcLabel, *currentInstructionLabel` – текстові поля для поточного стану.
- `ProcessorEmulator *emulator` – вказівник на об'єкт логіки емулятора.
- `bool programLoaded` – прапорець успішного завантаження програми.

Основні методи:

- `MainWindow(QWidget *parent = nullptr)` – конструктор інтерфейсу.
- `~MainWindow()` – деструктор, очищує ресурси.
- `void loadProgram()` – завантаження текстової програми з файлу.
- `void runProgram()` – запуск програми до завершення.
- `void stepProgram()` – покрокове виконання однієї інструкції.
- `void resetEmulator()` – скидання стану.
- `void updateDisplay()` – оновлення вмісту всіх вікон.
- `void updateRegisters(), updateMemory(), updateInstructions(), updateStatus()` – оновлення відповідних частин інтерфейсу.

Клас `ProcessorEmulator` реалізує логіку симульованого RISC-процесора. Він зберігає стан регістрів, пам'яті, лічильника команд та виконує інструкції через взаємодію з класом `InstructionSet`.

Основні поля:

- `int registers[8]` – масив із 8 регістрів загального призначення.
- `int memory[1000]` – масив пам'яті розміром 1000 комірок.
- `int programCounter` – лічильник команд.
- `bool zeroFlag, negativeFlag` – службові прапорці.
- `QStringList instructions` – список інструкцій програми.
- `bool finished, error` – прапорці стану.
- `QString errorMessage` – повідомлення про помилку.
- `InstructionSet instructionSet` – об'єкт, який відповідає за виконання інструкцій.

Основні методи:

- `bool loadProgram(const QString &filename)` – завантаження інструкцій з файлу.
- `void step()` – виконання однієї інструкції.
- `void reset()` – скидання стану процесора.
- `bool isFinished(), hasError()` – перевірка стану виконання.

- QString getErrorMessage() – повернення повідомлення про помилку.
- int getRegister(int), setRegister(int, int) – доступ до регістрів.
- int getMemory(int), setMemory(int, int) – доступ до пам'яті.
- int getProgramCounter(), setProgramCounter(int) – доступ до РС.
- bool getZeroFlag(), setZeroFlag(bool) – доступ до прапорця нуля.
- bool getNegativeFlag(), setNegativeFlag(bool) – доступ до прапорця від'ємності.

- const QStringList& getInstructions() – повертає інструкції.

Клас InstructionSet відповідає за декодування та виконання інструкцій. Він приймає розбиту інструкцію та аргументи, перевіряє їх, виконує дію і повідомляє про помилки.

Основні поля:

- ProcessorEmulator *emu – вказівник на об'єкт емулятора.
- QString lastError – останнє повідомлення про помилку.
- bool pcModified – прапорець модифікації РС (наприклад, при JMP).

Основні методи:

- bool execute(const QString& opcode, const QStringList& operands) – головна точка входу для виконання інструкції.
- QString getLastError() – повертає останню помилку.
- bool wasPCModified(), resetPCModified() – керування станом РС.
- int parseRegister(const QString&), parseImmediate(const QString&) – розбір операндів.
- bool isValidRegister(const QString&), isValidAddress(int) – перевірка правильності операндів.
- bool executeAdd(), executeLoad(), executeStore() тощо – реалізація кожної інструкції окремо.

Зв'язок між трьома класами побудовано за принципом ієрархії, що показано на схемі на рисунку 6.

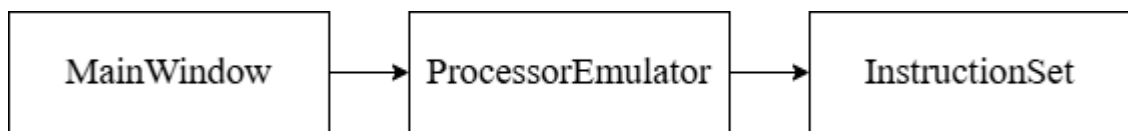


Рисунок 6 – Зв'язок між класами

Для реалізації такого зв'язку між класами, взаємодія між об'єктами класу відбувається наступним чином:

- MainWindow ініціалізує об'єкт ProcessorEmulator, викликає його методи step(), loadProgram(), getRegister() тощо.
- ProcessorEmulator, у свою чергу, має вбудований об'єкт InstructionSet, через який виконує інструкції.
- InstructionSet викликає методи emu->getRegister(), emu->setMemory() для зміни стану емулятора.
- Всі зміни стану відображаються в інтерфейсі через методи MainWindow::updateDisplay().

Розділення логіки на три окремі класи забезпечує чітку модульність, зручність підтримки та розширення функціоналу. Така структура є типовою для архітектурного підходу "MVC" (Model-View-Controller), де:

- MainWindow – View + Controller,
- ProcessorEmulator – Model,
- InstructionSet – Execution Unit.

Це забезпечує зрозумілий і логічний потік даних та команд у програмі.

У цьому розділі було детально розглянуто структуру та логіку роботи програмного забезпечення, розробленого для емуляції виконання інструкцій умовного RISC-процесора. Проведено аналіз основних модулів програми, зокрема класів MainWindow, ProcessorEmulator та InstructionSet, які реалізують відповідно графічний інтерфейс користувача, логіку ядра емулятора та набір інструкцій для виконання.

Також було проаналізовано формат представлення вхідної програми, принципи роботи з пам'яттю, реалізацію регістрів, алгоритм виконання інструкцій та механізм покрокового виконання. Розроблений графічний інтерфейс на базі фреймворку Qt забезпечує зручність взаємодії та наочне представлення стану системи.

Загалом, реалізоване рішення дозволяє ефективно виконувати емуляцію простих програм у навчальних цілях, забезпечує прозору структуру коду та модульну архітектуру, що спрощує розширення функціоналу в майбутньому.

					<i>ІАЛЦ.045300.004 ПЗ</i>	<i>Лист</i>
						50
<i>Зм</i>	<i>Лист</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Верифікація роботи реалізованої програми

Верифікація програмного забезпечення – це процес перевірки правильності реалізації функціоналу відповідно до вимог. Для реалізованої системи емуляції умовного RISC-процесора тестування включало створення тестових програм із заданим набором інструкцій, запуск їх у емуляторі та перевірку зміни стану регістрів і пам'яті після кожного кроку. Ціль верифікації – переконатися, що кожна інструкція виконується коректно, а також що реалізовано правильну логіку переходів, обробки умов, доступу до пам'яті та оновлення регістрів. Процес верифікації включав такі етапи:

1. Створення текстового файлу з інструкціями. Кожна програма містить 5–10 інструкцій, що перевіряють певні функціональні можливості.
2. Завантаження програми через інтерфейс. В програмі натискається кнопка "Load Program", після чого інструкції зчитуються та виводяться у вікні.
3. Покрокове виконання інструкцій. За допомогою кнопки "Step" відбувається послідовне виконання команд.
4. Моніторинг стану системи. У режимі реального часу оновлюються значення регістрів, пам'яті та поточної інструкції.
5. Порівняння фактичного результату з очікуваним. Якщо кінцеві значення відповідають очікуванню – інструкція вважається успішно верифікованою.

Такий підхід дозволив протестувати базовий набір інструкцій, а також перевірити загальний потік виконання.

Приклад 1. Тест арифметичних інструкцій:

Метою цього тесту є перевірка коректності роботи інструкцій SET, ADD, SUB, AND, OR.

Текст програми (Test1.txt):

SET R0, 10

SET R1, 5

ADD R2, R0, R1

SUB R3, R0, R1

AND R4, R0, R1

OR R5, R0, R1

Очікувані результати:

- R0 = 10
- R1 = 5
- R2 = 15 (10 + 5)
- R3 = 5 (10 - 5)
- R4 = 0 (10 & 5 = 1010 & 0101 = 0000)
- R5 = 15 (10 | 5 = 1010 | 0101 = 1111)

На рисунку 7 можна спостерігати результат виконання тесту (Test1.txt).

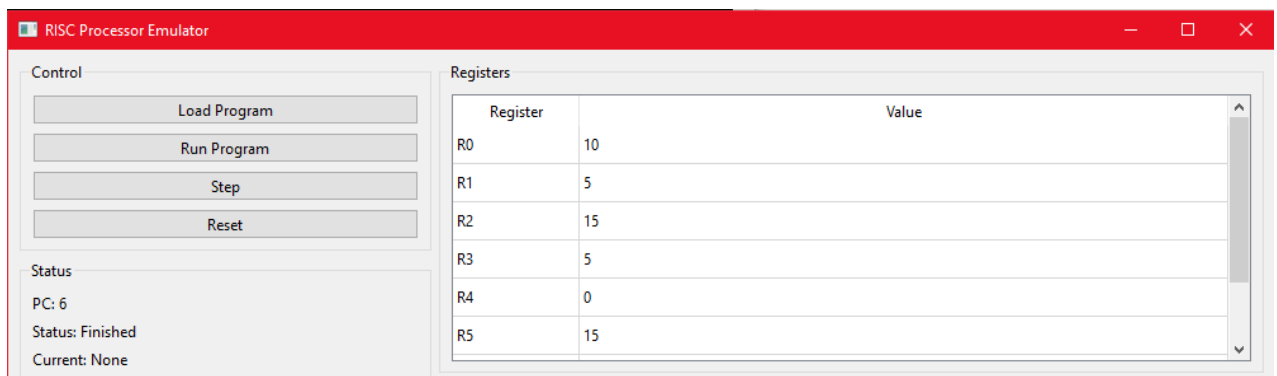


Рисунок 7 – Результат роботи емулятора після виконання (Test1.txt)

Цей тест показав правильність реалізації арифметичних і логічних операцій. Значення регістрів відповідають очікуваним, помилок не зафіксовано.

Приклад 2. Тест умовного переходу

Тест спрямований на перевірку інструкцій BEQ, JMP, а також роботи з пам'яттю (LOAD, STORE).

Текст програми (Test2.txt):

SET R0, 5

SET R1, 5

BEQ R0, R1, 4

SET R3, 10

JMP 6

SET R4, 10

STORE R0, 0

LOAD R2, 0

Очікувані результати:

- R0 = 5
- R1 = 5
- BEQ спрацює, оскільки R0 == R1, тому пропустимо інструкцію SET R3, 10
- JMP спрацює, тому пропустимо інструкцію SET R4, 10
- R0 зберігається в пам'ять за адресою 0
- R2 = 5 (значення з пам'яті)

На рисунку 8 можна спостерігати результат виконання тесту (Test2.txt).

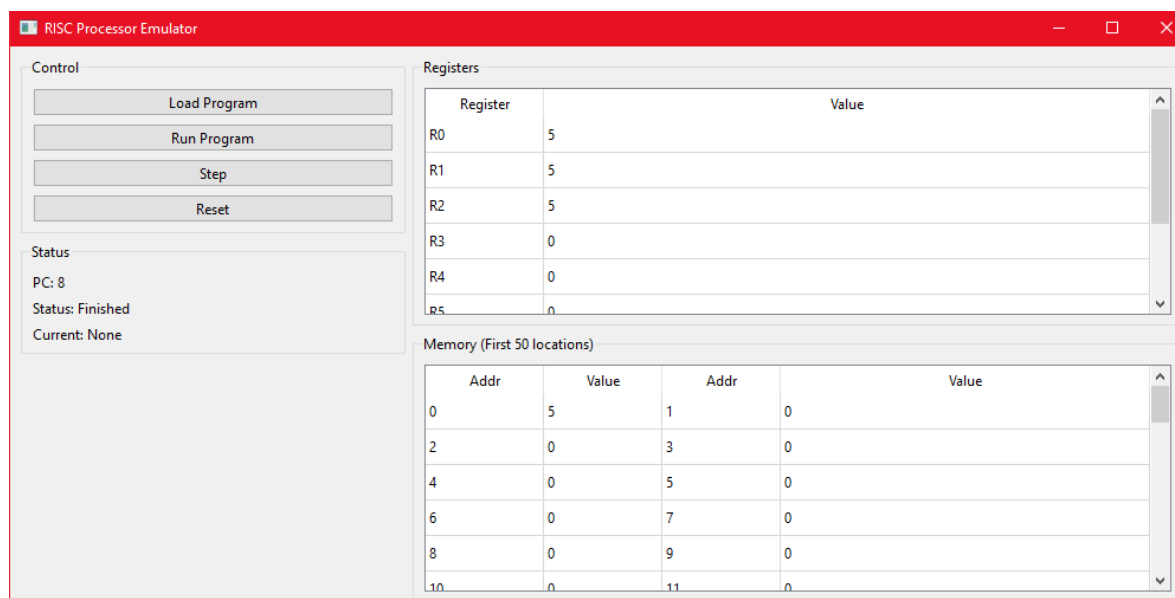


Рисунок 8 – Результат роботи емулятора після виконання (Test2.txt)

В результаті було підтверджено правильність умовного переходу та роботи з пам'яттю.

Результати верифікації підтверджують правильність роботи реалізованих інструкцій. Кожна команда виконується у відповідності до очікуваної поведінки. Графічний інтерфейс дозволяє ефективно спостерігати за зміною стану системи, що є надзвичайно зручним для навчальних цілей.

Оцінка зручності використання інтерфейсу

Одним із ключових завдань при створенні спеціалізованої комп'ютерної системи для навчальних цілей є забезпечення інтуїтивно зрозумілого, функціонального та візуально комфортного інтерфейсу користувача. Інтерфейс, розроблений у цьому дипломному проєкті, реалізований із використанням фреймворку Qt та призначений для максимальної наочності під час дослідження виконання інструкцій умовного мікропроцесора.

У програмі чітко відокремлені основні функціональні блоки:

- Панель керування з кнопками для завантаження програми, запуску, покрокового виконання та скидання.
- Таблиця регістрів, що виводить значення усіх доступних регістрів у режимі реального часу.
- Таблиця оперативної пам'яті, яка показує перші 50 комірок пам'яті в зручному табличному вигляді.
- Вікно з переліком завантажених інструкцій, де поточна команда виділяється для візуального відслідковування ходу виконання.
- Статусна панель, яка інформує користувача про поточний стан системи (запущено, зупинено, помилка, завершено).

Усі елементи згруповані логічно, що дозволяє новому користувачеві швидко зорієнтуватися у функціоналі програми без попередньої підготовки. Особлива увага приділена тому, щоб програма була зрозумілою для студентів технічних спеціальностей, які лише починають знайомство з принципами роботи процесора, пам'яті та налагодження програм.

Результати неформального тестування інтерфейсу серед кількох користувачів підтвердили, що інтерфейс є зручним для базової роботи, а також має потенціал до розширення функціональності в майбутньому. Використання Qt дозволило досягти високого рівня стабільності, кросплатформеності та адаптивності візуального оформлення.

Таким чином, інтерфейс програми повністю задовольняє поставлені цілі дипломного проєкту – забезпечення простого, наочного та функціонального середовища для дослідження та налагодження виконання програм в умовах емуляції процесора.

Порівняння результатів з існуючими аналогами

У ході виконання дипломного проєкту була реалізована спеціалізована комп'ютерна система для емуляції та налагодження програм, яка орієнтована на освітнє використання. Щоб оцінити доцільність створеного програмного забезпечення, проведено порівняльний аналіз із низкою популярних емуляторів і налагоджувачів, таких як QEMU, DOSBox, GDB та Turbo Debugger.

Більшість розглянутих інструментів орієнтовані на розробку та запуск реального програмного забезпечення під конкретні архітектури або операційні системи. Наприклад:

- QEMU підтримує повноцінну емуляцію десятків процесорних архітектур, включаючи ARM, MIPS, x86, RISC-V тощо. Він дозволяє запускати повноцінні операційні системи, але має складний інтерфейс і потребує попередньої підготовки з конфігурації.
- DOSBox більше орієнтований на запуск старих DOS-додатків, а не на гнучку симуляцію процесора.
- GDB та Turbo Debugger є потужними налагоджувачами, однак мають досить складний текстовий інтерфейс, що ускладнює сприйняття матеріалу для новачків.

У порівнянні з цими рішеннями, реалізоване ПЗ має декілька важливих переваг у навчальному контексті:

- Простота: інтерфейс програми інтуїтивно зрозумілий навіть для користувачів без досвіду роботи з низькорівневим програмуванням.
- Наглядність: усі зміни в пам'яті, регістрах, виконуваних інструкціях відображаються у реальному часі, що значно полегшує розуміння внутрішньої логіки процесора.
- Легкість у використанні: відсутня необхідність у встановленні складних середовищ чи залежностей.
- Гнучкість моделі: реалізована архітектура дозволяє розширення функціоналу шляхом додавання нових інструкцій або розширення кількості регістрів.

З іншого боку, варто відзначити, що реалізована система не претендує на універсальність і не підтримує повноцінну компіляцію/збірку бінарних файлів, віртуалізацію апаратного забезпечення чи роботу з реальними архітектурами – на відміну від QEMU або GDB. Основна її перевага полягає у наочності та зручності саме в освітньому середовищі, а не у професійній розробці або налагодженні складних систем.

Таким чином, у рамках поставленої задачі розроблене рішення вдало поєднує простоту, наочність та функціональність. У порівнянні з існуючими аналогами, програма забезпечує базові можливості налагодження, але з максимально зрозумілим та доступним для початківця інтерфейсом, що робить її доцільною саме для використання в навчальних курсах з архітектури комп'ютерів, мікропроцесорної техніки або низькорівневого програмування.

Висновки щодо ефективності реалізації

Реалізована система продемонструвала високу ефективність у межах своєї спеціалізації – освітнього інструменту для вивчення принципів роботи процесора та інструкційного потоку. Програма дозволяє зручно завантажувати

користувацький код, контролювати виконання кожної інструкції, відображати зміни у регістрах і пам'яті, що особливо корисно для студентів і викладачів дисциплін з комп'ютерної архітектури.

Ключовими факторами ефективності реалізації є:

- модульна структура коду, що спрощує підтримку та можливість розширення функціональності;
- графічний інтерфейс, побудований на базі Qt, що забезпечує зручність користування;
- простота взаємодії та доступність функцій без потреби в глибоких технічних знаннях.

Хоча система не підтримує виконання складних програм чи емуляцію реального апаратного середовища, вона ідеально підходить для візуального розуміння логіки виконання коду та роботи з пам'яттю, що підтверджено тестуванням та порівнянням з наявними аналогами.

Таким чином, реалізована програма є ефективним прикладом навчального симулятора, що забезпечує якісну підтримку процесу вивчення основ мікропроцесорної техніки та системного програмування.

ВИСНОВОК

У межах бакалаврського дипломного проєкту було виконано повний цикл дослідження, розробки, реалізації та аналізу спеціалізованої комп'ютерної системи для емуляції і налагодження програм, що імітують виконання інструкцій умовного RISC-процесора. Проєкт реалізовано як навчальний програмний інструмент, який дозволяє в інтерактивному режимі моделювати базові принципи функціонування мікропроцесорів, працювати з пам'яттю та регістрами, а також здійснювати покрокове налагодження програмного коду.

У першому розділі дипломної роботи було проведено ґрунтовний аналіз існуючих рішень у сфері емуляції та налагодження програм. Розглянуто основи архітектури мікропроцесорів, включаючи відмінності між CISC і RISC, а також особливості таких процесорів, як Xtensa LX6, ARM Cortex-M3 та Atmel AVR5. Вивчено базові поняття, пов'язані з віртуалізацією, моделюванням інструкцій, симуляцією регістрів та пам'яті. Додатково було розглянуто поширені емулятори (QEMU, DOSBox, NES емулятор) та налагоджувачі (GDB, AFD, Turbo Debugger), порівняно їх функціональність у вигляді таблиць, що дозволило зробити висновок про відсутність простих та універсальних рішень, зручних для використання в навчальному процесі. Це й стало мотивацією для розробки власного програмного рішення.

У другому розділі було розроблено архітектуру програми, що складається з трьох ключових модулів: MainWindow (інтерфейс користувача), ProcessorEmulator (логіка виконання), InstructionSet (обробка інструкцій). Програма реалізована мовою C++ з використанням бібліотеки Qt, що дозволило побудувати сучасний графічний інтерфейс. Користувач має змогу завантажити текстову програму, написану в умовному асемблері, яка містить інструкції базового набору (арифметичні, логічні, умовні переходи, взаємодія з пам'яттю тощо). Інтерфейс дозволяє в інтерактивній формі пояснювати студентам принципи роботи команди, показувати помилки програм, демонструвати

гілкування та умови переходу. Було описано механізм покрокового виконання команд, реалізацію реєстрів, формат інструкцій, модель пам'яті, логіку взаємодії між модулями. Архітектура системи розроблена згідно з принципами об'єктно-орієнтованого програмування, з чітким розподілом обов'язків між класами. Це забезпечує хорошу масштабованість коду, його тестованість та повторне використання. Завдяки модульному підходу, програму легко підтримувати та розширювати у майбутньому без ризику виникнення конфліктів між компонентами. У процесі реалізації проєкту було досягнуто усіх поставлених початкових цілей: реалізовано стабільне ядро емуляції, створено модуль управління інструкціями, побудовано інтуїтивно зрозумілий графічний інтерфейс, проведено тестування з використанням контрольних програм. Функціональність повністю відповідає навчальним потребам, а також може бути використана для демонстраційних цілей. Під час розробки дипломного проєкту було суттєво поглиблено знання з мов програмування C++ та бібліотеки Qt, опановано принципи побудови архітектури програмного забезпечення, а також відпрацьовано навички створення інтуїтивного інтерфейсу користувача. Здобутий досвід є цінним кроком у професійному становленні в сфері програмної інженерії та вбудованих систем.

У третьому розділі було проведено верифікацію програми, описано процес тестування інструкцій на прикладі простих програм із використанням реалізованих інструкцій. Представлено знімки екрану з результатами виконання, що підтверджують коректність реалізації. Додатково проаналізовано зручність інтерфейсу, описано переваги та особливості взаємодії з системою. Також виконано порівняльний аналіз із існуючими аналогами, який підтвердив, що реалізоване рішення має меншу складність, кращу інтуїтивність, а також адаптоване для початкового рівня користувачів. Зроблено висновки про ефективність реалізації, яка дозволила досягнути поставленої мети – створення функціонального навчального інструменту для вивчення принципів архітектури мікропроцесорів.

Таким чином, у результаті виконання дипломного проєкту:

- Розроблено спеціалізовану комп'ютерну систему, що виконує завдання емуляції та налагодження програм на умовному RISC-процесорі.
- Реалізовано покрокове виконання інструкцій, зручну візуалізацію стану регістрів та пам'яті.
- Забезпечено модульну архітектуру з розділенням на інтерфейсну частину, ядро емуляції та обробку інструкцій.
- Проведено аналіз аналогів, обґрунтовано потребу у створенні власного рішення.
- Підтверджено працездатність і ефективність реалізованого ПЗ шляхом тестування та верифікації.

Розроблена програма може бути використана у навчальному процесі, для демонстрації основ комп'ютерної логіки, структури процесора, принципів виконання та налагодження коду. У перспективі можливе подальше розширення функціоналу: підтримка нових інструкцій, розширення кількості регістрів, графічна візуалізація команд та збереження результатів виконання.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Espressif Systems. ESP32 Technical Reference Manual [Електронний ресурс]. – Режим доступу:
https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
2. Espressif Systems. ESP32 Datasheet [Електронний ресурс]. – Режим доступу:
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
3. Cadence Design Systems. Xtensa Instruction Set Architecture (ISA) Reference Manual [Електронний ресурс]. – Режим доступу:
https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/silicon-solutions/compute-ip/isa-summary.pdf
4. Yiu J. The Definitive Guide to the ARM Cortex-M3. – Newnes, 2008.
5. Free Software Foundation. GDB: The GNU Project Debugger Documentation [Електронний ресурс]. – Режим доступу:
<https://www.gnu.org/savannah-checkouts/gnu/gdb/documentation/>
6. QEMU Developers. QEMU Documentation [Електронний ресурс]. – Режим доступу: <https://www.qemu.org/documentation/>
7. Intel Corporation. The Journey Inside: Microprocessors [Електронний ресурс]. – Режим доступу:
<https://www.intel.com/content/dam/www/program/education/us/en/documents/the-journey-inside/microprocessor/tji-microprocessors-handout2.pdf>
8. RealBoy Emulator. Emulating the core [Електронний ресурс]. – Режим доступу: <https://realboyemulator.wordpress.com/2013/01/08/emulating-the-core-1/>