Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Засоби паралельних обчислень для прискорення емуляції процесора на прикладі Nintendo GameBoy

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Михайло ОРДИНЦЕВ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Олександр МАМЧИЧ
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ординцеву Михайлу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Засоби паралельних обчислень для прискорення емуляції процесора на прикладі Nintendo GameBoy

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) мова програмування: Rust; 2) бібліотека: SDL2

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз проблеми та огляд існуючих рішень;
2) розробка емулятору системи Nintendo Game Boy;
3) впровадження технологій та методів паралельного обчислення у емулятор;
4) аналіз зміни продуктивності;
5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Вибір технології розробки та інструменталь-	31.05.25-02.06.25	
	них засобів		
3	Розробка алгоритмічного забезпечення	03.06.25-05.06.25	
4	Розробка та відлагодження програмного	06.06.25-09.06.25	
	забезпечення		
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25-11.06.25	
6	Подання кваліфікаційної роботи керівникові та	12.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ Ас. Олександр МАМЧИЧ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 65 с., 4 рис., 4 табл., 1 дод., 14 джерел.

ЕМУЛЯЦІЯ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ПРИСКОРЕННЯ ЕМУЛЯЦІЇ , ІГРОВА КОНСОЛЬ, SDL2, RUST.

Метою кваліфікаційної роботи є дослідження можливостей застосування паралельних обчислень для підвищення продуктивності емуляції процесора, зокрема на прикладі класичної ігрової консолі Nintendo Game Boy. Робота спрямована на виявлення ефективних підходів до розпаралелювання обчислень у рамках обмеженого апаратного функціоналу та архітектурних особливостей оригінальної платформи.

У ході виконання кваліфікаційної роботи я вивчив наявні підходи до розпаралелювання емуляторів, зосередившись на проектах з відкритим кодом. Особливу увагу приділено тому, як інші розробники намагаються пришвидшити емуляцію без втрати точності. Після аналізу я вирішив створити власний емулятор Game Boy, щоб мати повний контроль над архітектурою та експериментами. У процесі розробки я реалізував базову однопоточну версію, а потім поступово почав переносити деякі частини в окремі потоки. Зокрема, рендеринг відеобуфера було винесено в окремий потік, що дозволило зменшити затримки під час малювання кадрів. Досвід показав, що не всі частини можна ефективно розділити без ризику втрати точності. Попри це, реалізований підхід підтвердив потенціал використання паралельних обчислень у сфері емуляції.

ABSTRACT

Bachelor’s thesis: 65 pages, 4 figures, 4 tables, 1 appendices, 14 sources.

EMULATION, PARALLEL COMPUTING, EMULATION SPEEDUP, GAME CONSOLE, SDL2, RUST.

The major goal of this thesis is to explore the potential of parallel computing to improve the performance of processor emulation, using the classic Nintendo Game Boy console as a case study. The work focuses on identifying effective approaches to parallelizing computations within the constraints of limited hardware capabilities and the architectural specifics of the original platform.

In order to achieve this goal, existing approaches to parallelizing emulators – especially open-source projects – were analyzed. Particular attention was paid to how other developers attempt to accelerate emulation without compromising accuracy. Based on this analysis, a custom Game Boy emulator was developed to allow full control over its architecture and implementation details. A single-threaded version was implemented first, followed by gradual separation of some components into parallel threads. In particular, the video buffer rendering was offloaded to a separate thread, which helped reduce frame rendering latency. The experience revealed that not all parts can be effectively parallelized without risking loss of accuracy. Nevertheless, the implemented solution demonstrated that parallel computation has promising potential in the context of emulation.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

APU – звуковий процесор (англ. Audio Processing Unit)

ARC – атомарний лічильник посилань (англ. Atomic Reference Counter)

CPU – центральний процесор (англ. Central Processing Unit)

DIV – регістр-лічильник тактів (англ. Divider Register)

DMA – прямий доступ до пам’яті (англ. Direct Memory Access)

FIFO – принцип черги "першим прийшов – першим обслужений" (англ. First In, First Out)

IME – головний дозвіл переривань (англ. Interrupt Master Enable)

LCD – рідкокристалічний дисплей (англ. Liquid Crystal Display)

PC – лічильник команд (англ. Program Counter)

PPU – процесор обробки зображень (англ. Picture Processing Unit)

RAM – оперативна пам’ять (англ. Random Access Memory)

ROM – постійна пам’ять (англ. Read Only Memory)

SDL2 – проста бібліотека для роботи з мультимедіа (англ. Simple DirectMedia Layer 2)

TAC – регістр керування таймером (англ. Timer Control)

TIMA – таймерний лічильник (англ. Timer Counter)

TMA – регістр початкового значення таймера (англ. Timer Modulo)

ВСТУП

Сучасне програмування стикається з наступною проблемою: потужність багатоядерних систем часто спрямовується на емуляцію пристроїв, які історично були принципово одноядерними. Саме розв'язання цього фундаментального виклику – створення ефективного паралельного емулятора для системи з послідовним виконанням інструкцій – формує ядро даного дослідження.

Історично шлях емуляторів ігрових консолей нерозривно пов'язаний з еволюцією обчислювальної техніки та програмування. Перші спроби емуляції датуються 1990-ми роками, коли обчислювальні можливості персональних комп’ютерів досягли рівня, що дозволяв відтворювати системи попереднього покоління. Відтоді технології емуляції зробили значний крок уперед, трансформувавшись від елементарних інтерпретаторів до комплексних систем, що активно використовують динамічну рекомпіляцію та інші передові підходи. У сучасному ландшафті розробки програмного забезпечення емулятори посідають важливе місце, дозволяючи реконструювати роботу різноманітних систем на актуальних комп'ютерних платформах. Емуляція апаратного забезпечення залишається одним із найскладніших напрямків програмування, що вимагає глибокого проникнення в архітектуру процесорів та специфіку інших компонентів обчислювальних систем. Особливий інтерес у цьому контексті становить емуляція ігрових консолей, які вирізняються своєю унікальною архітектурою та жорсткими ресурсними обмеженнями. Сучасні методики емуляції відкривають можливості не лише для відтворення ігор, але й для проведення автоматизованого аналізу ігрового середовища [1].

Nintendo Game Boy – одна з найбільш знакових консолей в історії – і досі слугує яскравим прикладом компактної обчислювальної системи з прозорою та чітко окресленою архітектурою. Емуляція Game Boy є цінним

дослідницьким полігоном для вивчення принципів роботи процесорів, пам'яті та систем введення-виведення, особливо в контексті виклику паралельної емуляції одноядерної платформи.

Основна складність полягає в необхідності детермінованого відтворення поведінки апаратної частини, що висуває підвищені вимоги до точності синхронізації та обчислювальних ресурсів. Ці вимоги особливо критичні при спробах масштабування або оптимізації швидкодії. У зв’язку з цим виникає потреба у виявленні тих компонентів емуляції, які можуть бути ефективно розпаралелені без порушення загальної послідовності обчислень і хронологічної достовірності емуляції.

Центральний виклик – знайти ті частини роботи емулятора, які можна розпаралелити без шкоди для коректної поведінки системи. Адже хоч паралельні обчислення сьогодні стали нормою, сама логіка Game Boy залишається послідовною: процесор виконує інструкції одну за одною, без відгалужень. Але не вся система працює синхронно з CPU – і саме в цьому полягає можливість.

У рамках цього дослідження було створено gb-emulator – паралельний емулятор Game Boy, у якому основна увага приділяється точному відтворенню роботи процесора Sharp LR35902 (модифікований Z80), а також ефективній обробці відеовиводу. Щоб використати багатоядерні процесори сучасних ПК, рендеринг зображення було винесено в окремі потоки: кожен із них відповідає за свою частину кадру. Такий підхід дозволяє значно прискорити роботу емулятора. Експерименти показали, що завдяки цьому підходу вдалося майже вдвічі збільшити кількість кадрів за секунду (FPS) без втрати точності емуляції. Це підтверджує ефективність розробленої архітектури. У процесі розробки виникло чимало цікавих технічних задач – зокрема, пов’язаних із синхронізацією потоків, підтримкою детермінованої поведінки, і забезпеченням стабільної взаємодії між компонентами системи.

Спроба знайти відповідь на це запитання стала центральною темою дослідження. Щоб краще зрозуміти потенціал паралельної обробки, було

проаналізовано структуру типової емуляційної системи та визначено, які її частини найменш залежні від жорсткої послідовності виконання. Особливу увагу було приділено балансуванню між продуктивністю та точністю, адже будь-яке втручання в архітектуру класичної системи може призвести до втрати автентичності емуляції.

У цьому контексті особливої ваги набуває практичний експеримент – спроба винести окремі функціональні підсистеми за межі основного потоку виконання. Саме така архітектурна трансформація дозволяє краще використати переваги сучасного багатоядерного середовища без порушення внутрішньої логіки емулятора.

У роботі детально проаналізовано, як саме було реалізовано паралельну обробку, які переваги вона дала та які виклики поставила. Також обговорюються шляхи подальшої оптимізації та перспективи розширення паралелізації на інші частини емулятора – можливо, навіть на сам процесор, якщо підійти до цього дуже обережно. ця робота не тільки демонструє конкретний спосіб підвищення продуктивності класичних емуляторів, а й пропонує загальну методику – як виявляти приховані можливості для розпаралелення в системах, які спершу виглядають повністю послідовними

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

В ході виконання кваліфікаційної роботи слід вирішити наступні задачі:

- проаналізувати вже готові рішення емуляції ігрової консолі Nintendo Game Boy;
- проаналізувати можливі шляхи паралелізації;
- проаналізувати можливості для паралелізації емуляції даної системи;
- розробити структуру емулятору із використанням паралельних обчислень;
- зробити аналіз досягнутих результатів.

1.1 Аналіз існуючих рішень

У галузі емуляції Nintendo Game Boy накопичено значну кількість програмних рішень, які охоплюють як закриті комерційні продукти, так і відкриті проєкти з активною спільнотою розробників. Кожен з емуляторів має власну архітектуру, набір функціональних можливостей, рівень точності та продуктивності, а також по-різному використовує або ігнорує сучасні підходи до паралельної обробки даних. Аналіз існуючих рішень є важливим етапом у створенні власного емулятора, оскільки дозволяє оцінити поточний стан індустрії, визначити кращі практики, а також виявити типові помилки та обмеження. Зокрема, прицільна увага приділяється тому, чи реалізовано в цих проєктах підтримку багатопоточності та в який спосіб це зроблено.

1.1.1 Аналіз емулятору BGB

BGB (рисунок 1.1) - є одним із найпопулярніших і найточніших емуляторів Nintendo Game Boy, розроблений з акцентом на максимальну

достовірність емуляції. Цей емулятор надзвичайно точно відтворює роботу оригінального обладнання, що дозволяє запускати навіть демосцени та ігри, які використовують нестандартні особливості апаратного забезпечення [2].

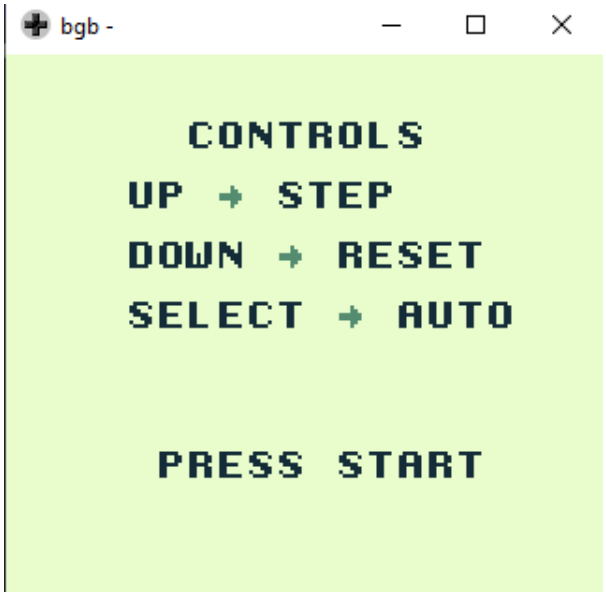

Рисунок 1.1 – Робота емулятору BGB

Особливої уваги заслуговують вбудовані потужні інструменти для відлагодження, включаючи асемблерний дизайнер, переглядач пам'яті та точні профілювальники, що робить його неоціненним інструментом для розробників ігор та дослідників архітектури Game Boy. Однак BGB реалізований як однопотоковий додаток, що обмежує його продуктивність на сучасних багатоядерних системах. Відсутність паралелізації призводить до неповного використання доступних обчислювальних ресурсів.

Хоча для емуляції Game Boy однопотокової продуктивності сучасних процесорів у більшості випадків достатньо, це обмежує можливості масштабування під час виконання додаткових ресурсоємних операцій.

1.1.2 Аналіз емулятору VisualBoyAdvance

VisualBoyAdvance (рисунок 1.2) - емулятор із відкритим кодом для Game Boy та Game Boy Advance [3]. Цей емулятор здобув популярність

завдяки своїй доступності на різних операційних системах, таких як Windows, Linux та macOS, а також завдяки відносно низьким вимогам до апаратного забезпечення, що робило його привабливим для широкого кола користувачів, включаючи власників старих або малопотужних комп’ютерів.

Рисунок 1.2 – Робота емулятору VisualBoyAdvance

Однією з головних переваг VisualBoyAdvance був широкий набір функцій, які значно покращували зручність користування: можливість зберігати та завантажувати стан гри в будь-який момент, підтримка функції прискореної емуляції (speed-up), опції зміни масштабу та фільтрів зображення, а також підтримка геймпадів. Однак, незважаючи на масове використання, VisualBoyAdvance має певні обмеження в точності емуляції, особливо в контексті роботи з тайм-критичними аспектами системи. У порівнянні з новішими емуляторами оригінальна реалізація VBA часто не забезпечує cycle-accurate поведінку, що може призводити до незначних або навіть критичних відхилень у роботі деяких ігор. Важливо зазначити, що оригінальний код VBA не використовував жодної форми паралелізації – всі основні компоненти, включаючи CPU, графіку та звук, працювали у одному потоці.

1.1.3 Аналіз емулятору mGBA

mGBA – сучасний високоточний емулятор (рисунок 1.3), створений з урахуванням помилок попередніх проєктів [4]. Розробка mGBA була розпочата з акцентом на детермінованість, продуктивність та портативність.

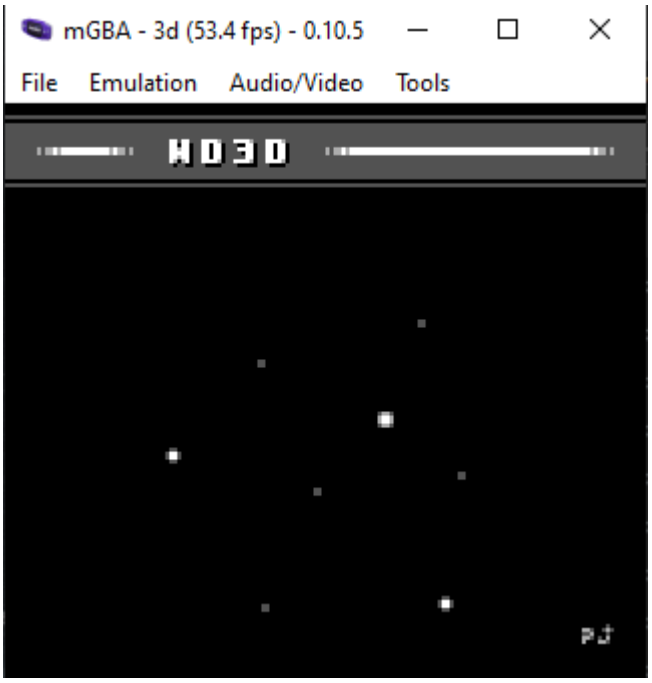

Рисунок 1.3 – Робота емулятору mGBA

Емулятор реалізує точну емуляцію всіх компонентів системи, включаючи точні тайминги процесора та відеоконтролера. Це забезпечує сумісність з широким спектром ігор, включаючи ті, що використовують специфічні особливості апаратного забезпечення. Архітектура mGBA розроблена з думкою про модульність, що полегшує перенесення на різні платформи та інтеграцію з іншими проєктами.

У контексті паралелізації mGBA реалізує деякі форми паралельної обробки для неемуляційних аспектів, таких як рендеринг графіки та обробка звуку. Основна логіка емуляції залишається переважно послідовною для забезпечення точності.

1.1.4 Аналіз емулятору SameBoy

SameBoy (рисунок 1.4) – відносно новий емулятор, який приділяє особливу увагу точності емуляції та підтримці всіх моделей Game Boy (включаючи оригінальний Game Boy, Game Boy Color та Super Game Boy). Емулятор точно відтворює особливості різних моделей Game Boy, включаючи відмінності в кольорових палітрах та часових характеристиках . SameBoy також відзначається вражаючими можливостями відтворення звуку, з точною емуляцією звукового чіпа Game Boy, що особливо важливо для музичних демосцен та ігор з акцентом на звуковий супровід [5].

Рисунок 1.4 – Робота емулятору SameBoy

В архітектурі SameBoy основна логіка емуляції реалізована послідовно, але деякі аспекти, такі як обробка звуку та рендеринг, можуть виконуватися в окремих потоках. Це дозволяє досягти балансу між точністю та продуктивністю.

Проєкт має відкритий код та активну спільноту розробників, що забезпечує постійне вдосконалення та адаптацію нових технологій. Однак, як і інші емулятори Game Boy, SameBoy не реалізує глибоку паралелізацію

основного процесу емуляції через значні складнощі з синхронізацією компонентів.

1.2 Висновки із аналізу вже існуючих рішень

Аналізуючи існуючі емулятори Nintendo Game Boy, можна помітити загальну тенденцію: основна логіка емуляції процесора та інших компонентів системи зазвичай реалізується послідовно для забезпечення точності та простоти синхронізації.

Паралельні обчислення переважно застосовуються для допоміжних операцій, наприклад, деякі емулятори намагаються винести рендеринг графіки в окремий потік, інші – обробку звуку або навіть синхронізацію периферії. У більшості випадків архітектура все ще залишається переважно однопотоковою, що пов’язано як з історичними обмеженнями, так і з складністю реалізації точного таймінгу.

Така ситуація зумовлена специфікою архітектури Game Boy, де всі компоненти тісно взаємодіють і синхронізуються за тактами процесора. Спроби повної паралелізації емуляції різних компонентів стикаються зі значними складнощами в забезпеченні коректної синхронізації та обміну даними між потоками, що часто нівелює потенційний виграш у продуктивності.

Більшість проєктів дотримуються послідовного підходу до основної логіки емуляції, застосовуючи паралельні обчислення лише для неемуляційних аспектів.

1.3 Підходи до паралелізації їх плюси та мінуси

Підходи до паралелізації в контексті емуляції можна розділити на кілька основних категорій. Кожна з них має свої особливості, переваги та обмеження. Вибір конкретного підходу залежить від поставлених цілей –

максимальної продуктивності, точності або гнучкості – а також від архітектурних особливостей самої системи, яку емулюють:

- паралелізація на рівні емульованої системи;
- паралелізація на рівні інструкцій;
- паралелізація допоміжних операцій;
- спекулятивна паралелізація;
- гібридні стратегії.

1.3.1 Паралелізація на рівні емульованої системи

Паралелізація на рівні емульованої системи передбачає розділення емуляції різних компонентів системи на окремі потоки. Цей підхід був успішно реалізований у проєкті PQEMU, що демонструє значне підвищення продуктивності емуляції [6]. В архітектурі Nintendo Game Boy такими компонентами є процесор, відеоконтролер (PPU), звуковий процесор (APU) та контролери пам'яті.

Така стратегія паралелізації є найбільш інтуїтивною, оскільки відображає фізичну структуру емульованої системи. Проте цей підхід зіштовхується з суттєвими проблемами синхронізації, адже компоненти реальної системи функціонують у тісній взаємодії з точною часовою синхронізацією. Складність полягає в забезпеченні коректної послідовності подій та обміну даними між компонентами, що працюють у різних потоках.

Дослідження COREMU показало, що масштабовані та портативні рішення для паралельної емуляції дозволяють ефективно розподіляти навантаження між ядрами процесора [7]. Спроби реалізації цього підходу часто призводять до складних механізмів синхронізації, які можуть негативно вплинути на загальну продуктивність через накладні витрати на взаємодію між потоками. Крім того, необхідність частого обміну даними між компонентами може створювати вузькі місця в продуктивності, особливо коли швидкість міжпотокової комунікації стає обмежувальним фактором.

Основні переваги цього підходу:

- природне відображення архітектури оригінальної системи;
- потенційно висока продуктивність при правильній реалізації;
- можливість масштабування на системи з великою кількістю ядер.

З основних недоліків можна виділити:

- складні механізми синхронізації;
- високі накладні витрати на міжпотокову взаємодію;
- ризик виникнення станів гонки та інших проблем багатопотокового програмування.

1.3.2 Паралелізація на рівні інструкцій

Паралелізація на рівні інструкцій намагається емулювати кілька інструкцій процесора одночасно, що є особливо складним завданням через послідовну природу виконання програми та залежності між окремими інструкціями. Цей підхід вимагає глибокого аналізу коду, що виконується, для виявлення незалежних інструкцій, які можуть бути виконані паралельно без порушення логіки програми.

В контексті емуляції Game Boy з процесором Sharp LR35902 ця задача ускладнюється наявністю побічних ефектів від виконання інструкцій, таких як зміна прапорців стану процесора, які можуть впливати на подальше виконання програми. Реалізація паралелізації на рівні інструкцій вимагає створення складних механізмів аналізу залежностей та розподілу завдань, що може бути невиправданим з точки зору співвідношення складності реалізації до отриманого приросту продуктивності.

Цей підхід більш характерний для сучасних процесорів із суперскалярною архітектурою, ніж для емуляції відносно простих 8-бітних систем. Однак деякі експериментальні емулятори використовують елементи такого підходу, аналізуючи блоки інструкцій і виявляючи можливості для паралельного виконання незалежних блоків.

Основні виклики цього підходу:

- складність аналізу залежностей між інструкціями;
- необхідність розробки спеціалізованих алгоритмів для виявлення незалежних інструкцій;
- потреба в додатковому проміжному представленні коду для аналізу.

1.3.3 Паралелізація допоміжних операцій

Паралелізація допоміжних операцій, підхід, який був успішно реалізований у даному проєкті, залишає основний процес емуляції послідовним, що забезпечує детермінованість відтворення логіки роботи емульованої системи, але переносить ресурсоємні операції, такі як рендеринг графіки, в окремі паралельні потоки.

Такий підхід дозволяє ефективно використовувати багатоядерні процесори сучасних комп'ютерів без необхідності впровадження складних механізмів синхронізації основної логіки емуляції. У реалізованому емуляторі Nintendo Game Boy процес рендерингу було розділено на кілька потоків, кожен з яких відповідає за обробку певної частини зображення. Завдяки цьому, вдалося досягти майже двократного підвищення продуктивності (кількості кадрів за секунду) порівняно з однопотоковою реалізацією. Цей підхід особливо ефективний для операцій, які є обчислювально інтенсивними і при цьому можуть бути природно розділені на незалежні частини для паралельної обробки. Він не стосується критичних компонентів, таких як CPU чи PPU, але дозволяє розвантажити основний потік і підвищити загальну продуктивність емулятора.

Окрім рендерингу графіки, такими операціями можуть бути:

- обробка та генерація звуку;
- фільтрація та покращення зображення;
- запис та відтворення стану емуляції;
- мережева взаємодія у багатокористувацьких емуляторах.

Перевагами такого підходу є:

- збереження точності емуляції основної логіки системи;
- відносна простота реалізації порівняно з іншими підходами;
- хороший баланс між продуктивністю та складністю;
- мінімальні ризики порушення коректності емуляції.

1.3.4 Спекулятивна паралелізація

Спекулятивна паралелізація представляє собою більш експериментальний підхід, при якому система намагається передбачити можливі результати виконання. Цей метод базується на ідеї, що в деяких випадках можна передбачити найімовірніші шляхи виконання програми і почати їх обчислення заздалегідь.

Такий підхід застосовується в сучасних процесорах для підвищення продуктивності через спекулятивне виконання інструкцій, але в контексті емуляції він ще не набув широкого поширення через складність реалізації та потенційно високі накладні витрати.

Основні виклики спекулятивної паралелізації:

- складність передбачення можливих шляхів виконання програми;
- високі накладні витрати на обчислення альтернативних варіантів;
- необхідність швидкого відкату до коректного стану при невірному передбаченні;
- обмежена ефективність для застарілих архітектур з простим потоком виконання.

Потенційні переваги цього підходу:

- можливість значного прискорення емуляції для програм з передбачуваною поведінкою;
- використання простоїв процесора для обчислення потенційних майбутніх станів;
- можливість оптимізації виконання умовних переходів та циклів.

1.3.5 Гібридні стратегії

Гібридні стратегії виникають як відповідь на обмеження окремих підходів до паралелізації. Вони дозволяють гнучко комбінувати різні методи залежно від контексту виконання та особливостей цільової системи. Наприклад, можлива комбінація паралелізації на рівні системи для компонентів із меншою залежністю один від одного, як звуковий процесор, із послідовною емуляцією компонентів, що тісно взаємодіють між собою – процесор та відеоконтролер). Проект Graphite демонструє ефективність розподіленої паралельної симуляції для багатоядерних архітектур [8].

Деякі сучасні емулятори застосовують динамічний підхід до паралелізації, автоматично визначаючи оптимальну стратегію в залежності від:

- поточного навантаження на систему;
- характеру емульованої програми;
- доступних апаратних ресурсів;
- бажаного балансу між точністю та швидкістю емуляції.

Важливими аспектами гібридних стратегій є:

- адаптивність до різних сценаріїв використання;
- можливість динамічного перерозподілу обчислювальних ресурсів;
- поєднання переваг різних підходів для досягнення оптимального результату;
- застосування спеціалізованих стратегій для різних підсистем емульованого пристрою.

Використання гібридних підходів дозволяє досягти найкращого балансу між продуктивністю та точністю емуляції, адаптуючись до конкретних умов та вимог користувача. Це відкриває можливості для гнучкого масштабування, оптимізації під різні апаратні конфігурації та забезпечення стабільної роботи в широкому спектрі сценаріїв.

1.4 Архітектурні виклики для паралелізації

Попри візуальну простоту консолі, точна емуляція Nintendo Game Boy вимагає глибокого розуміння його внутрішньої архітектури та значного інженерного зусилля для відтворення автентичної поведінки. Ключовим чинником складності є жорстка взаємозалежність між усіма підсистемами, які працюють у тісному часовому зв'язку. Центральний процесор (CPU), графічний процесор (PPU), звукова система (APU), таймери, пам'ять, а також ввід/вивід – усі ці компоненти функціонують у суворій синхронізації на рівні тактів, і навіть невелике відхилення від очікуваного таймінгу може призвести до появи графічних артефактів, неправильного відтворення звуку або збоїв у ігровій логіці. Це створює серйозні виклики при побудові програмної моделі: кожна інструкція CPU може мати точний фіксований або змінний час виконання, який повинен бути врахований під час емуляції, особливо в контексті доступу до спільних ресурсів, таких як пам'ять чи регістри. Наприклад, інструкція запису в пам'ять під час певного стану відеоконтролера (наприклад, у період сканування OAM або передачі DMA) повинна або бути відхилена, або викликати зміну стану системи, що має бути правильно змодельовано в коді.

Складність зростає ще більше, коли постає питання продуктивності. На перший погляд, Game Boy має незначну обчислювальну потужність, і може здатися, що емуляція такої платформи не вимагає серйозних ресурсів. Проте у разі реалізації точної емуляції, де необхідно зберігати хронологічну послідовність змін стану системи з точністю до одного такту, звичайна однопотокова реалізація швидко стає вузьким місцем.

І тут постає ключовий архітектурний виклик: паралелізація компонентів. На відміну від сучасних архітектур, де підсистеми працюють асинхронно і можуть бути оброблені у окремих потоках, Game Boy спроєктований як суворо послідовна система, де будь-який компонент може впливати на інший у будь-який момент часу. Наприклад, PPU змінює стан

екрану у строго визначених фазах сканування, які збігаються з виконанням інструкцій CPU, а звук формується в залежності від точної тривалості імпульсів. Спроба винести обробку будь-якої з цих систем у паралельний потік одразу створює проблему актуальності даних: як забезпечити, щоб потік, який емулює PPU, точно знав, в якому стані перебуває CPU на кожному такті? А якщо врахувати ще й таймери або доступ до спільної пам’яті, проблема стає експоненційно складнішою.

Класичні підходи до паралелізації (наприклад, використання черг повідомлень між потоками, або блокуючих структур синхронізації) не підходять, бо сам по собі оверхед від синхронізації може знівелювати будь-яку продуктивність. Крім того, паралельне виконання може призвести до порушення порядку подій, що в свою чергу – до непередбачуваних поведінок, які важко виявити або діагностувати. У Game Boy навіть глюки або особливі побічні ефекти, пов'язані з «race conditions» на апаратному рівні, іноді використовувались у іграх як очікувана поведінка – і цю поведінку також треба повторити в рамках емулятора.

Таким чином, для підвищення продуктивності залишається дуже вузьке поле для маневру. У реальних реалізаціях паралелізація можлива лише для допоміжних компонентів, таких як рендеринг графіки на стороні хоста, логування, обробка виводу або масштабування зображення. Але основний цикл емульованої системи – CPU, PPU, APU, таймери – повинен залишатися синхронним і працювати у суворій часовій послідовності. Навіть відносно прості спроби «рознести» ці частини по потоках призводять до втрати точності або непередбачуваної поведінки.

1.5 Обраний підхід до паралелізації

У розробленому емуляторі Nintendo Game Boy винесення допоміжних операцій у паралельні потоки стало оптимальним рішенням, яке забезпечило баланс між складністю реалізації та зростанням продуктивності. Такий підхід

дозволив зберегти детермінованість роботи основних компонентів системи, водночас істотно прискоривши її загальне функціонування завдяки оптимізації найбільш ресурсоємного етапу – рендерингу графіки.

Застосована стратегія відповідає сучасним галузевим тенденціям і раціонально використовує потенціал багатоядерних процесорів, що дало змогу досягти суттєвого приросту швидкодії без компромісів щодо точності – одного з ключових критеріїв якісної емуляції. Зафіксоване майже дворазове підвищення продуктивності підтвердило дієвість цього підходу й заклало надійну основу для подальшого вдосконалення.

Надалі реалізацію можна розширити за рахунок впровадження інших методів паралелізації та переходу до гібридних стратегій. Це дозволить ще ефективніше розподіляти обчислювальні ресурси між підсистемами емулятора, підтримуючи високу детермінованість і стабільність. Поточна архітектура вже показала добру масштабованість, тому її розвиток може включати глибшу оптимізацію потоків, мінімізацію блокувань і поліпшення синхронізації. З огляду на актуальні тенденції у сфері багатоядерних обчислень, такі кроки відкривають шлях до ще більшого приросту продуктивності, що є критично важливим для забезпечення плавної роботи в реальному часі. Таким чином, реалізований підхід є не лише ефективним у нинішньому вигляді, а й має вагомий потенціал для майбутньої еволюції та адаптації до зростаючих вимог.

2 АНАЛІЗ ТЕХНОЛОГІЙ, ЩО ВИКОРИСТОВУЮТЬСЯ

Успішна реалізація програмного забезпечення залежить не лише від правильно сформульованої ідеї, а й від грамотного вибору інструментів. У цьому розділі розглянуто набір технологій, що були використані під час розробки: мова програмування, бібліотека для графічного інтерфейсу, середовище розробки та система контролю версій. Кожна з них була обрана на основі відповідності цілям проєкту, критеріям ефективності, зручності та надійності.

2.1 Обрані технології для розробки

Для реалізації цього проєкту було обрано набір інструментів, які забезпечують баланс між високою продуктивністю, зручністю розробки та безпекою. Основною мовою програмування став Rust, що вирізняється сучасною системою типів і підтримкою безпечної багатопоточності. Для реалізації графічного виводу й обробки взаємодії з користувачем використовувалася бібліотека SDL2, яка є стандартом де-факто в розробці емуляторів і ігор. Розробка здійснювалася у середовищі Visual Studio Code, що забезпечує зручну інтеграцію з екосистемою Rust та іншими інструментами.

2.1.1 Мова програмування Rust

Rust – це системна мова програмування, яка поєднує низькорівневу ефективність із високим рівнем безпеки. Вона була створена з метою забезпечити продуктивність, порівняну з C та C++, але без властивих цим мовам проблем – зокрема, з управлінням пам’яттю, race conditions і буферними переповненнями [9]. Rust досягає цього завдяки унікальній

системі ownership, яка дозволяє компілятору гарантувати відсутність помилок, пов’язаних із використанням звільненої пам’яті чи даних з кількох потоків без синхронізації.

Окремо варто відзначити, що Rust чудово масштабується – від невеликих інструментів командного рядку до великих систем. Його строгий компілятор часто називають «найкращим другом програміста», адже він допомагає уникати сотень потенційних помилок ще до запуску програми. Крім того, Rust має багату екосистему бібліотек (crates), серед яких, наприклад, serde для серіалізації, tokio для асинхронного програмування та sdl2 для мультимедійної роботи. Rust має зручний менеджер пакетів Cargo, який автоматизує збірку, встановлення залежностей та тестування. Його екосистема активно розвивається, а спільнота орієнтована на високу якість коду та документації.

Мова активно використовується у сфері розробки системного ПЗ, вбудованих систем, веб-серверів (наприклад, через фреймворк Actix або Rocket), а також у сфері криптовалют, ігрової індустрії та навіть при розробці ядра Linux.

Rust підтримує безпечне багатопотокове програмування «із коробки», змушуючи розробника явно вказувати права власності на ресурси, що значно знижує ймовірність помилок синхронізації.

Переваги Rust роблять його чудовим вибором для проєктів, де важлива як продуктивність, так і надійність, особливо там, де помилки можуть мати критичні наслідки. Таким чином, Rust став фундаментом проєкту, забезпечивши стабільність, продуктивність і сучасні засоби безпеки пам’яті.

2.1.2 Мультимедійна бібліотека SDL2

SDL2 – це кросплатформна бібліотека, яка забезпечує прямий доступ до низькорівневих мультимедійних можливостей комп’ютера, таких як графіка, аудіо, обробка введення з клавіатури, миші або геймпада. Вона

широко використовується у розробці ігор, емуляторів, візуалізаційних систем, інтерактивних додатків, а також у навчальних цілях, де потрібно швидко створити просте графічне середовище [10].

SDL2 написана мовою C, але має обгортки для багатьох інших мов, включно з C++, Rust, Python тощо. Вона підтримує роботу з 2D-графікою, роботу з таймерами, обробку подій та навіть керування вікнами. Крім того, SDL2 дозволяє легко інтегруватися з іншими графічними API, такими як OpenGL, Vulkan або Direct3D. SDL2 також підтримує апаратне прискорення через використання SDL_Renderer, що дозволяє ефективно оновлювати графіку в режимі реального часу. Бібліотека добре масштабується – від простих 2D-демо до складних графічних рушіїв.

Однією з головних переваг SDL2 є її простота у використанні. Вона не нав’язує складних патернів, що робить її зручною для швидкого прототипування та навчання. Також важливо, що вона є повністю відкритою і має активну спільноту, яка підтримує розвиток бібліотеки. Завдяки простоті інтеграції та широким можливостям, SDL2 стала ідеальним вибором для реалізації візуальної частини застосунку.

2.1.3 Середовище розробки Visual Studio Code

Visual Studio Code (VS Code) – це потужний і водночас легкий редактор коду з відкритим вихідним кодом, розроблений компанією Microsoft. Він призначений для швидкої та зручної розробки програмного забезпечення різної складності: від невеликих скриптів до масштабних проєктів. VS Code підтримує понад сотню мов програмування завдяки вбудованим механізмам автодоповнення, підсвічуванню синтаксису, відлагодженню та широкому набору розширень [11].

Однією з ключових переваг редактора є його розширюваність. Через внутрішній маркетплейс можна встановити плагіни для мов, фреймворків, інструментів аналізу коду, систем контролю версій (Git), роботи з Docker,

базами даних тощо. З таких хотілося б відмітити rust-analyzer, який додає підтримку мови програмування Rust і CodeLLDB, який додає можливість зручної відладки для більшості з існуючих компільованих мов програмування. Особливо зручною є інтеграція з терміналом, що дозволяє не залишаючи редактору виконувати збірку, запуск тестів або інші команди.

Важливо також, що VS Code підтримує інтелектуальні підказки (IntelliSense), рефакторинг і статичний аналіз коду, що підвищує продуктивність розробника. Середовище має глибоку інтеграцію з Git і дозволяє керувати змінами у коді без використання командного рядка.

Завдяки підтримці відлагоджувачів, інтеграції з Git та зручному інтерфейсу, VS Code є чудовим інструментом як для новачків, так і для професійних розробників. Він підтримує роботу в різних операційних системах (Windows, macOS, Linux) та має активну спільноту, яка регулярно публікує нові теми, розширення та оновлення.

Таким чином, Visual Studio Code забезпечує зручне, інтуїтивне середовище для ефективної роботи з кодом, особливо у поєднанні з інструментами екосистеми Rust.

2.1.4 Технологія контролю версій Git

Git – це розподілена система контролю версій, яка дозволяє ефективно керувати історією змін у програмному коді [12]. Вона стала стандартом у сучасній розробці програмного забезпечення завдяки своїй гнучкості, швидкості та надійності. Git дозволяє створювати гілки (branches), зливати зміни (merge), повертатися до попередніх версій, а також ефективно працювати над проєктами як індивідуально, так і в команді. Його розподілена природа означає, що кожен розробник має повну копію репозиторію, що дає змогу працювати офлайн і знижує ризик втрати даних.

GitHub – це вебплатформа для хостингу репозиторіїв Git, яка надає зручний інтерфейс для співпраці, перегляду змін, обговорення коду та

керування задачами (issues). GitHub також підтримує систему pull request – механізм перевірки й узгодження змін у коді, що є стандартом в open source та корпоративній розробці. Крім того, платформа інтегрується з CI/CD системами, що дозволяє автоматизувати збірку та тестування коду.

У сукупності Git і GitHub формують сучасний стандарт управління розробкою, забезпечуючи контроль, історію, співпрацю та автоматизацію, що є критично важливим для командних або довготривалих проєктів.

2.2 Архітектурні підходи

Під час розробки емулятора було застосовано низку архітектурних підходів, які забезпечують баланс між продуктивністю, структурною ясністю та точністю відтворення поведінки емульованої системи. До таких підходів належать: станова машина (State Machine), data-driven design, shared state synchronization та імперативний стиль програмування. Кожен із них виконує свою роль в архітектурі проєкту та дозволяє ефективно вирішувати специфічні завдання.

2.2.1 Станова машина

Станова машина – це архітектурний підхід, у якому поведінка системи залежить від поточного стану. Для кожного стану визначається окремий набір дій та умови переходу до інших станів. Цей підхід широко застосовується в побудові процесорів, контролерів вводу/виводу, графічних процесорів, парсерів та інтерфейсів користувача [13].

Використання State Machine дозволяє чітко структурувати логіку компонентів із передбачуваним життєвим циклом. Наприклад, у графічних підсистемах етапи вибірки пікселів або сканування спрайтів можна формалізувати як окремі стани з переходами між ними.

2.2.2 Data-Driven дизайн

Підхід Data-Driven Design (DDD) полягає в тому, що поведінка програми визначається не жорстко закодованою логікою, а даними, які можна змінювати, не змінюючи сам код.

Цей підхід особливо ефективний при реалізації інтерпретаторів, компіляторів, емуляторів, а також у системах із великою кількістю однотипних структур, таких як інструкції процесора чи елементи GUI. У DDD замість численних умов (if, match, switch) використовується таблиця структур або конфігурацій, у якій кожен елемент описує дію, режим адресації, параметри тощо. Це полегшує розширення функціональності та спрощує підтримку.

2.2.3 Синхронізація спільного стану

При використанні багатопоточності виникає потреба у координації доступу до спільних ресурсів. Підхід Shared State Synchronization забезпечує коректну взаємодію між потоками за допомогою атомарних операцій, м'ютексів, умовних змінних тощо. Зокрема, часто використовуються такі інструменти, як Mutex, Atomic, або Barrier для координації виконання в багатопоточному середовищі [14].

Цей підхід застосовується, наприклад, у паралельному рендерингу графіки, коли кілька потоків обробляють різні частини зображення, а головний потік чекає завершення за допомогою лічильника або іншого синхронізатора.

2.2.4 Імперативний стиль програмування

Імперативний стиль – це один із найстаріших і найпоширеніших підходів до програмування, який базується на послідовному виконанні

інструкцій. У цьому стилі розробник явно описує, як досягти результату, керуючи потоком виконання програми за допомогою змінних, умовних операторів, циклів і викликів функцій. Це забезпечує високий рівень контролю над усіма аспектами роботи програми.

Такий підхід особливо підходить для задач, де важлива точна послідовність дій, контроль за ресурсами та ефективне використання обчислювальних можливостей. Завдяки передбачуваності та прямолінійності імперативний стиль є базовим у мовах системного рівня, таких як C, C++ чи Rust, де важливо мати прямий вплив на пам’ять, потоки, пристрої введення-виведення.

Імперативний стиль також полегшує реалізацію складних алгоритмів, в яких потрібно враховувати проміжні стани та взаємодію численних компонентів. У поєднанні з іншими підходами – такими як state machine або data-driven – він дозволяє будувати чітку, продуктивну і контрольовану архітектуру. Його застосування особливо виправдане в умовах обмежених ресурсів або в системах, де потрібна детермінованість відтворення та мінімізація накладних витрат.

3 АРХІТЕКТУРА ПРОГРАМИ

У цьому розділі детально описано архітектурну структуру розробленого емулятора Game Boy та алгоритми, які реалізують обчислення, з особливим акцентом на методах паралельної обробки.

Основна мета – продемонструвати, як грамотне розділення відповідальностей між модулями та продумане використання багатопоточності дозволяють досягнути істотного покращення продуктивності без втрати точності емуляції.

Але під час детального аналізу стало зрозуміло, що емуляція одноядерного синхронізованого процесору із тісно зв’язаними елементами, які працюють у чіткій послідовності – нетривіальна задача, яку не так і легко виконати. Було обране рішення реалізувати основний цикл емулятору і звести до мінімуму оверхед від системи та зовнішніх пристроїв за допомогою паралелізації її окремих компонентів, які можуть призвести до падіння

3.1 Модуль CPU та обробка переривань

Модуль CPU відповідає за повну емуляцію центрального процесора консолі Game Boy – Sharp LR35902. Цей процесор є гібридом інструкційного набору Intel 8080 та Zilog Z80, із частковими змінами, які зробили його простішим і водночас більш ефективним для вбудованих систем.

Емуляція CPU включає реалізацію повного набору інструкцій, включаючи як базові арифметичні та логічні операції, так і операції керування потоком, роботу з пам’яттю, стосом та регістрами. Процесор підтримує понад 500 інструкцій, і кожна з них виконується за фіксовану кількість тактів, що є критично важливим для точного відтворення поведінки оригінального заліза. У реальній консолі багато елементів, які тісно прив’язані до точного ходу машинного часу, тому навіть невеликі

розбіжності в циклічному виконанні інструкцій можуть призвести до некоректної поведінки.

3.1.1 Архітектура та функціональність CPU

Центральний процесор реалізовано у вигляді структури Cpu (лістинг 3.1), яка інкапсулює всі необхідні компоненти для емуляції роботи реального процесора Game Boy.

Лістинг 3.1 – Реалізація структури Cpu

```
pub struct Cpu {
    pub regs: Registers,
    // current fetch
    fetched_data: u16,
    mem_dest: u16,
    dest_is_mem: bool,
    cur_opcode: u8,
    pub cur_inst: Instruction,
    pub is_halted: bool,
    pub interrupt_master_enabled: bool,
    enabling_ime: bool,
}
```

Процесор Game Boy має стандартний для архітектури Z80 набір 8-бітних регістрів (A, B, C, D, E, F, H, L), які часто використовуються попарно для формування 16-бітних значень (AF, BC, DE, HL). Особливу роль відіграє регістр F, який містить набір прапорців стану:

- Z (Zero) - встановлюється, якщо результат операції дорівнює нулю;
- N (Subtract) - встановлюється при виконанні операції віднімання;
- H (Half Carry) - встановлюється при переносі з нижньої половини байта;
- C (Carry) - встановлюється при переносі/заємі.

Структура процесора Game Boy містить не лише регістри та прапорці, але й важливі допоміжні поля для обробки інструкцій. Поле fetched_data зберігає дані, отримані під час виконання інструкції, mem_dest вказує адресу

пам'яті призначення, коли операція працює з пам'яттю, а dest_is_mem визначає, чи є призначення операції регістром або областю пам'яті. Поля cur_opcode та cur_inst відстежують поточну виконувану інструкцію. Особливу роль відіграють прапорці стану процесора: is_halted вказує, що процесор перебуває в режимі очікування (інструкція HALT), а interrupt_master_enabled та enabling_ime керують обробкою переривань.

3.1.2 Цикл виконання інструкцій

Ключовим аспектом роботи модуля CPU є цикл виконання інструкцій, реалізований у методі step (лістинг 3.2). Цей метод послідовно виконує наступні послідовні дії:

- зчитування наступної інструкції;
- зчитування необхідних данних для виконання інструкції;
- виконання інструкції;
- обробка переривань.

Лістинг 3.2 – Метод step, що реалізує основний цикл CPU

```
pub fn step(&mut self, bus: &mut Bus) {
    if !self.is_halted {
        self.fetch_instruction(bus);
        bus.cycle(1);
        self.fetch_data(bus);
        self.execute(bus);
    } else {
        bus.cycle(1);
        if bus.interrupts.flags != 0 {
            self.is_halted = false;
        }
    }
    if self.interrupt_master_enabled {
        self.handle_interrupts(bus);
        self.enabling_ime = false;
    }
    if self.enabling_ime {
        self.interrupt_master_enabled = true;
    }}
```

У методі fetch_instruction операційний код зчитується з пам'яті за адресою, що знаходиться в програмному лічильнику (PC), після чого PC інкрементується. Далі метод fetch_data відповідає за отримання даних відповідно до режиму адресації поточної інструкції, заповнюючи поля fetched_data та mem_dest необхідними значеннями.

3.1.3 Режими адресації

Процесор Game Boy підтримує різноманітні режими адресації, реалізовані через перелік AddressMode (лістинг 3.3). Кожен режим адресації визначає, які дані потрібно завантажити та як вони будуть використані під час виконання інструкції.

Лістинг 3.3 – Перелік усіх можливих режимів роботи із пам’ятю

```
pub enum AddressMode {
    Imp,            // Імпліцитна адресація
    RegD16,         // Регістр + 16-бітні дані
    RegReg,         // Регістр у регістр
    MemReg,         // З пам'яті у регістр
    Reg,            // Один регістр
    RegD8,          // Регістр + 8-бітні дані
    RegMem,         // З регістра у пам'ять
    RegHLI,         // Регістр + інкремент HL
    RegHLD,         // Регістр + декремент HL
    HLIReg,         // Інкремент HL + регістр
    HLDReg,         // Декремент HL + регістр
    RegA8,          // Регістр + 8-бітна адреса
    A8Reg,          // 8-бітна адреса + регістр
    HLRegSPReg,     // HL + регістр стекового покажчика
    D16,            // 16-бітні дані
    D8,             // 8-бітні дані
    D16Reg,         // 16-бітні дані + регістр
    MemD8,          // Пам'ять + 8-бітні дані
    Mem,            // Доступ до пам'яті
    A16Reg,         // 16-бітна адреса + регістр
    RegA16,         // Регістр + 16-бітна адреса
}
```

3.1.4 Виконання інструкцій

Після отримання всіх необхідних даних, інструкція виконується за допомогою методу execute (лістинг 3.4), який викликає відповідну функцію залежно від типу інструкції.

Лістинг 3.4 – Виконання інструкції

```
pub fn execute(&mut self, bus: &mut Bus) {
       match self.cur_inst.in_type {
           IT::None => panic!(),
           IT::Nop => self.nop_in(),
           IT::Ld => self.ld_in(bus),
           IT::Inc => self.inc_in(bus),
           // Інші інструкції}}
```

3.1.5 Обробка переривань

Після виконання інструкції перевіряється наявність переривань і, якщо це необхідно, вони обробляються. CPU Game Boy використовує стек для зберігання адрес повернення при виклику підпрограм та обробці переривань.

Обробка переривань виконується після кожної інструкції, якщо дозволено глобальний прапорець переривань (IME). Game Boy підтримує п'ять типів переривань, які можуть бути дозволені або заборонені як індивідуально, так і глобально:

- VBlank (0x01) – сигналізує про завершення виведення кадру (вертикальна синхронізація);
- LcdStat (0x02) – змінює стан LCD-контролера;
- Timer (0x04) – викликається при спрацюванні таймера;
- Serial (0x08) – відповідає за події послідовного порту;
- Joypad (0x10) – активується при натисканні кнопок керування.

Кожному перериванню відповідає власна адреса в таблиці векторів переривань, на яку процесор переходить при його обробці. Стан системи

переривань відображається структурою InterruptState (лістинг 3.5), яка містить два регістри: прапорці переривань (flags) та маску дозволених переривань (enabled).

Лістинг 3.5 – Структура стану переривань

```
pub struct InterruptState {
    pub flags: u8,
    pub enabled: u8
}
```

Переривання вважається активним лише у випадку, якщо відповідний біт встановлено як у регістрі прапорців, так і в масці дозволених переривань. Це забезпечує гнучкий механізм пріоритезації та фільтрації переривань.

Метод handle_interrupts (лістинг 3.6) перевіряє наявність активних переривань і, якщо такі є, зберігає адресу повернення в стек, вимикає IME і передає управління обробнику відповідного переривання.

Лістинг 3.6 – Процес обробки переривань

```
pub fn handle_interrupts(&mut self, bus: &mut Bus) {
    match () {
      _ if self.process_interrupt(bus, Interrupt::VBlank) => (),
      _ => (), // Обробка інших типів переривань
    }
}
fn process_interrupt(&mut self, bus: &mut Bus, interrupt:
Interrupt) -> bool {
    let address = interrupt.address();

    if !bus.interrupts.is_active(interrupt) {
        return false;
    }
    self.stack_push16(self.regs.pc, bus);
    self.regs.pc = address;
    bus.interrupts.remove_flag(interrupt);
    self.is_halted = false;
    self.interrupt_master_enabled = false;

    true
}
```

Коли переривання обробляється, відбуваються наступні дії:

- адреса повернення (поточний PC) зберігається у стеку;
- PC змінюється на адресу обробника переривання;
- прапорець переривання знімається в регістрі IF;
- процесор виходить зі стану HALT, якщо був у ньому;
- головний дозвіл переривань (IME) вимикається.

Однією з критичних вимог для точної емуляції є правильне відтворення тактування оригінального апаратного забезпечення. Процесор Game Boy працює на частоті 4,194304 МГц, і кожна інструкція займає певну кількість машинних циклів. В емуляторі це реалізовано шляхом виклику bus.cycle(n), що симулює проходження n циклів і синхронізує всі компоненти системи. Завдяки такому підходу досягається висока детермінованість емуляції, що дозволяє коректно відтворювати роботу оригінальних ігор, включаючи складні випадки синхронізації та тайминги.

3.2 Шина пам'яті

Шина пам'яті є ключовим компонентом архітектури емулятора Game Boy, що забезпечує комунікацію між центральним процесором та усіма периферійними пристроями системи. Вона абстрагує доступ до різних сегментів адресного простору та реалізує модель відображення пам'яті, яка відповідає принципам функціонування оригінального апаратного забезпечення.

3.2.1 Структура шини пам’яті

У розробленому емуляторі шина представлена структурою Bus (лістинг 3.7), що виступає центральним вузлом системи та містить посилання на всі основні компоненти:

- графічний процесор (PPU);

- контролер таймера;
- DMA-контролер для прямого доступу до пам'яті;
- ігровий контролер для обробки введення;
- модулі пам'яті (ROM, RAM);
- контролер серіального порту;
- систему обробки апаратних переривань;
- інтерфейс екрану.

Така архітектура дозволяє централізовано координувати функціонування всіх елементів під час емуляції, забезпечуючи коректну синхронізацію та уніфікований доступ до адресного простору. Централізація доступу до пам'яті є критично важливою, оскільки в оригінальній системі Game Boy кожен компонент потенційно може модифікувати дані в будь-якій доступній ділянці пам'яті, що становить суттєвий виклик при подальшій паралелізації обчислень.

Лістинг 3.7 – Реалізація структури Bus

```
#repr(C)
pub struct Bus<'a> {
    pub interrupts: InterruptState,
    rom: Rom,                       // Картридж з грою
    ram: Ram,                       // Внутрішня пам'ять
    pub ppu: Ppu,                   // Графічний процесор
    dma: Dma,    // Контролер прямого доступу до пам'яті
    pub emu: Emu,                   // Стан емулятора
    pub timer: Timer,               // Таймер
    pub gamepad: Gamepad,           // Ігровий контролер
    pub screen: &'a mut dyn GbWindow, // Інтерфейс екрану
    serial_data: [u8; 2],// Дані для серіального порту
}
```

3.2.2 Адресний простір та доступ до пам’яті

Адресний простір Game Boy має чітко визначену сегментацію, де кожний діапазон адрес призначений для певного типу пам'яті або периферійного пристрою (таблиця 3.1).

Таблиця 3.1 – Адресний простір Game Boy

Діапазон адрес	Призначення
0x0000 – 0x3FFF	Постійна частина ROM (банк 0)
0x4000 – 0x7FFF	Перемикальний ROM (банки 1+)
0x8000 – 0x97FF	Відеопам'ять (VRAM) для графічних елементів
0x9800 – 0x9BFF	Карта фону 1
0x9C00 – 0x9FFF	Карта фону 2
0xA000 – 0xBFFF	Зовнішня RAM картриджа
0xC000 – 0xCFFF	Внутрішня RAM (WRAM) банк 0
0xD000 – 0xDFFF	WRAM банк 1 (для Game Boy Color)
0xE000 – 0xFDFF	Ехо-пам'ять (дзеркало WRAM)
0xFE00 – 0xFE9F	Пам'ять OAM для спрайтів
0xFEA0 – 0xFEFF	Зарезервована область
0xFF00 – 0xFF7F	Регістри вводу/виводу
0xFF80 – 0xFFFE	Високошвидкісна RAM (HRAM)
0xFFFF	Регістр дозволу переривань

Доступ до цих сегментів адресного простору реалізовано через методи read та write (лістинг 3.8), які інкапсулюють логіку обробки кожного діапазону адрес та перенаправляють звернення до відповідних компонентів системи.

Лістинг 3.8 – Методи для взаємодії з адресним простором

```
pub fn read(&self, address: u16) -> u8 {
    match address {
        interrupts::INTERRUPT_ENABLE_ADDRESS =>
self.interrupts.enabled,
        0..0x8000 => self.rom.read(address),// ...інші сегменти}
}
pub fn write(&mut self, address: u16, value: u8) {
    match address {
        interrupts::INTERRUPT_ENABLE_ADDRESS =>
self.interrupts.enabled = value, // ...інші сегменти}}
```

3.2.3 Синхронізація компонентів системи

Окрім маршрутизації доступу до пам'яті, шина відповідає за синхронізацію всіх компонентів емулятора. Цей процес реалізовано через метод cycle (лістинг 3.9), який імітує проходження одного машинного циклу, послідовно оновлюючи стан таймерів, графічного процесора та інших пристроїв. Кожен цикл емуляції складається з чотирьох тактів, протягом яких відбувається покрокове оновлення внутрішніх станів всіх компонентів системи та обробка потенційних DMA-передач.

Лістинг 3.9 – Метод синхронізації компонентів системи

```
pub fn cycle(&mut self, cycles: i32) {
    let bus: &mut Bus = make_mut_ref!(self);
    for _ in 0..cycles {
        for _ in 0..4 {
            self.timer.ticks = self.timer.ticks.wrapping_add(1);
            self.timer.tick(bus);
            self.ppu.tick(bus);
        }
        self.dma.tick(bus);
    }
}
```

3.2.4 Виклики в реалізації шини пам'яті

Описана архітектура шини пам'яті створює специфічні виклики для розпаралелювання обчислень при емуляції. Наявність таких тісно синхронізованих процесів суттєво ускладнює реалізацію розпаралелювання емуляції, оскільки кожен компонент повинен залишатися узгодженим з іншими упродовж усього часу виконання програми. Навіть незначне відхилення в часі може призвести до порушення логіки гри або відображення графіки.

Крім того, кожен із компонентів системи може ініціювати переривання, які необхідно коректно обробляти. Система переривань Game Boy дозволяє

реагувати на асинхронні події, такі як завершення кадру, сигнали введення з джойстика, події таймера або серійного порту. Система переривань вимагає строгої послідовності дій, що може блокувати паралельне виконання. Правильна реалізація обробки переривань є критично важливою для сумісності з великою кількістю ігор, які активно використовують ці механізми.

3.3 Таймер та його функціонування в архітектурі Game Boy

Таймер є критичним компонентом архітектури Game Boy, що забезпечує точну часову синхронізацію системи та коректне функціонування всіх залежних від часу процесів. Він генерує переривання через визначені інтервали часу, надаючи можливість програмам виконувати часозалежні операції з високою точністю. Система таймера Game Boy представлена чотирма спеціалізованими регістрами, розташованими в адресному просторі вводу-виводу (таблиця 3.2).

Таблиця 3.2 – Адресний простір таймеру

Назва регістру	Адреса	Призначення
DIV	0xFF04	Дільник частоти, 16-бітний регістр, що автоматично інкрементується з фіксованою частотою незалежно від стану системи
TIMA	0xFF05	Лічильник таймера, 8-бітний регістр, що інкрементується з частотою, визначеною регістром
TMA	0xFF06	Модуль таймера, 8-бітний регістр, значення якого завантажується в TIMA при переповненні останнього
TAC	0xFF07	Контроль таймера, 8-бітний регістр, що визначає режим роботи та частоту інкрементування таймера

В розробленому емуляторі таймер реалізований як незалежний модуль, представлений структурою Timer, яка інкапсулює всі необхідні регістри та функціональність для прецизійної емуляції поведінки таймера Game Boy (лістинг 3.10).

Лістинг 3.10 – Модуль Timer

```
pub struct Timer {
    pub div: u16,      // Регістр дільника (DIV)
    pub tima: u8,     // Регістр лічильника таймера (TIMA)
    pub tma: u8,       // Регістр модуля таймера (TMA)
    pub tac: u8,       // Регістр контролю таймера (TAC)
    pub ticks: u32,    // Внутрішній лічильник тактів}
```

Регістр TAC використовується для конфігурації частоти та активації таймера. Біти 0-1 регістра визначають вибір робочої частоти таймера згідно схеми.

Таблиця 3.3 – Конфігурація таймеру

Порядок бітів	Частота роботи таймеру (Гц.)	Кількість тактів процессору (т.)
00	4096 Гц.	1024 т.
01	262144 Гц.	16 т.
10	65536 Гц.	64 т.
11	16384 Гц.	256 т.

Функціонування таймера в емуляторі реалізовано через метод tick() (лістинг 3.11), який викликається синхронно з кожним тактом процесора. При кожному виклику регістр DIV інкрементується. Цей регістр є 16-бітним лічильником, хоча при читанні через адресу 0xFF04 доступним є лише старший байт. Відповідно до обраної частоти, система відстежує спадання конкретного біту в регістрі DIV. Метод використовує механізм порівняння попереднього та поточного стану відстежуваного біту, що дозволяє точно

визначити момент переходу з 1 в 0. Такий підхід забезпечує високу детермінованість емуляції та відповідність оригінальному апаратному забезпеченню. Коли фіксується спадання відповідного біту і таймер активований (біт 2 регістра TAC встановлений), значення регістра TIMA збільшується на одиницю. Важливо відзначити, що запис будь-якого значення в регістр DIV призводить до його скидання в нуль, що може використовуватися програмами для точного вимірювання часу.

При переповненні регістра TIMA (досягнення значення 0xFF), відбуваються наступні дії:

- скидання TIMA до значення, вказаного в регістрі TMA;
- генерація переривання таймера;
- збереження процесором поточного програмного лічильника у стеку;
- перехід до обробника переривання за адресою 0x0050.

Лістинг 3.11 – Метод функціонування таймеру

```
pub fn tick(&mut self, bus: &mut Bus) {
    let prev_divider = self.div;
    self.div = self.div.wrapping_add(1);
    let timer_update = match self.tac & 0b11 {
    if timer_update && (self.tac & (1 << 2)) != 0 {
        if self.tima == 0xFF {
            self.tima = self.tma;
            bus.interrupts.enable_flag(Interrupt::Timer);
        }
        self.tima += 1;}
}
```

Механізм роботи таймера має ключове значення для функціонування Game Boy, оскільки забезпечує:

- створення часозалежних ефектів у іграх та програмах;
- синхронізацію аудіовізуальних компонентів;
- точне вимірювання інтервалів часу для ігрового процесу;
- генерацію псевдовипадкових чисел для багатьох ігор;
- керування швидкістю анімації та оновленням екрану.

Імплементація таймера в розробленому емуляторі повністю відтворює поведінку оригінального апаратного забезпечення з високою точністю. Це дозволяє емулювати часову логіку ігор відповідно до специфікацій оригінальної платформи, забезпечуючи автентичний ігровий досвід. Точна емуляція таймера є однією з ключових вимог для стабільної роботи більшості ігор Game Boy, особливо тих, що покладаються на точну часову синхронізацію для критичних аспектів своєї функціональності.

3.4 Модуль PPU та використання паралельних обчислень

Процесор обробки піксельної графіки (PPU) є критичним компонентом апаратної архітектури Game Boy, що відповідає за генерацію візуального зображення. Сучасна емуляція цього компоненту потребує особливого підходу, оскільки графічна підсистема є одним із найбільш ресурсомістких та складних елементів при емуляції. Використання паралельних обчислень дозволяє значно підвищити продуктивність та ефективність емуляції PPU Game Boy.

В оригінальному пристрої Game Boy, PPU функціонує як відокремлений модуль, що працює асинхронно відносно центрального процесора (CPU), взаємодіючи з ним через переривання та спільний доступ до пам'яті.

3.4.1 Структура PPU та принцип роботи

У представленій реалізації емулятора, PPU модуль реалізовано через структуру Ppu (лістинг 3.12), яка інкапсулює всі необхідні компоненти для точної емуляції роботи графічного процесора. Ключовим елементом в архітектурі є структура PixelFiFo, яка моделює конвеєрну обробку пікселів, притаманну оригінальному PPU ігрової консолі Game Boy.

Лістинг 3.12 – Модуль PPU

```
pub struct Ppu {
    pub oam_ram: [Oam; 40], // інформація про спрайти
    pub vram: [u8; 0x2000], // відеопам’ять
    line_sprites: VecDeque<Oam>, // інформація про спрайти на
поточній лінії
    fetched_entry_count: u8,
    fetched_entries: [Oam; 3],
    pub window_line: u8,
    pub pfc: PixelFiFo, // черга пікселів
    pub current_frame: u32,
    pub line_ticks: u32,
    pub video_buffer: [Color; FRAME_BUFFER_SIZE],
    pub lcd: Lcd,
}
```

Процес рендерингу проходить у методі tick та складається з наступних етапів:

- завантаження номерів тайлів з карти тайлів;
- зчитування даних тайлів з відеопам'яті;
- завантаження спрайтів та інформації про них;
- формування черги пікселів з урахуванням пріоритетів;
- виведення пікселів на екран.

3.4.2 Використання паралельних обчислень для відображення графіки

Важливою інновацією реалізованого емулятора є застосування паралельних обчислень для підвищення продуктивності процесу рендерингу. На відміну від оригінального PPU, який послідовно не обробляв кожен піксель, лише заповнюючи буфер, після чого контролер екрану займався їх відображенням, нами потрібно емулювати як процесор, так і мікроконтролер екрану, але сучасні багатоядерні процесори дозволяють розподілити навантаження між кількома потоками виконання. В розробленому емуляторі застосовано декомпозицію за даними, де кадр розбивається на горизонтальні смуги, які обробляються паралельно різними потоками (лістинг 3.13).

Такий підхід особливо ефективний у нашому контексті, оскільки:

- забезпечує рівномірний розподіл навантаження між потоками;
- мінімізує потребу в синхронізації під час обробки;
- ефективно використовує кеш процесора завдяки локальності даних.

Лістинг 3.13 – Реалізація паралельної обробки кадру

```
pub fn draw_frame(&mut self, buffer: &[GbColor]) {
    // [...початок методу, вимірювання часу...]
    let texture_creator = self.canvas.texture_creator();
    let mut texture = texture_creator
        .create_texture_streaming(
            sdl2::pixels::PixelFormatEnum::RGB24,
            X_RES as u32,Y_RES as u32,)
        .unwrap();
    // Спільні ресурси для всіх потоків
    let shared_pixel_data = Arc::new(Mutex::new(vec![0u8; (X_RES
* Y_RES * 3) as usize]));
    let shared_buffer: Arc<[GbColor]> =
Arc::from(buffer.to_vec().into_boxed_slice());
    let done_counter = Arc::new(AtomicUsize::new(0));
    let num_threads = self.thread_senders.len();
    let chunk_height = Y_RES as usize / num_threads;
    // Розподіл завдань між потоками
    for (i, tx) in self.thread_senders.iter().enumerate() {
        let start_y = i * chunk_height;
        let end_y = if i == num_threads - 1 {
            Y_RES as usize} else {
            (i + 1) * chunk_height};
        tx.send(FrameTask {
            start_y,
            end_y,
            buffer: shared_buffer.clone(),
            pixel_data: shared_pixel_data.clone(),
            done_counter: done_counter.clone(),
        })
        .unwrap();
    }// Очікування завершення всіх потоків
    while done_counter.load(Ordering::SeqCst) < num_threads {
        std::thread::yield_now();
    }// Відображення результату
    let pixel_data = shared_pixel_data.lock().unwrap();
    texture.update(None, &pixel_data, (X_RES * 3) as usize)
        .unwrap();
    self.canvas.copy(&texture, None, None).unwrap();
    self.canvas.present();
    // [...кінець методу, оновлення лічильників...]
}
```

Як можна побачити, процес рендерингу кадру з використанням паралельних обчислень включає наступні етапи:

1. підготовка даних та ресурсів:
    1) створення текстури SDL для фінального відображення;
    2) підготовка спільних буферів даних через Arc та Mutex;
    3) ініціалізація атомарного лічильника завершених задач;
2. декомпозиція та розподіл завдань:
    1) розбиття кадру на горизонтальні смуги для кожного потоку;
    2) формування структур FrameTask для кожної смуги ;
    3) відправлення завдань потокам через канали mpsc::Sender;
3. паралельна обробка даних:
    1) кожен потік обробляє свою смугу зображення;
    2) конвертація даних з формату кольорів Game Boy у RGB формат;
    3) запис результатів у спільний буфер;
    4) інкремент атомарного лічильника після завершення обробки;
4. синхронізація та відображення результатів:
    1) очікування завершення обробки всіх смуг (моніторинг атомарного лічильника);
    2) оновлення текстури SDL даними з обробленого буфера;
    3) відображення текстури на канвас SDL;
    4) оновлення статистики кадрів (FPS).

Для організації паралельної обробки використовується спеціальна структура FrameTask (лістинг 3.14), яка містить всю необхідну інформацію для обробки сегмента кадру окремим потоком. В реалізації використано декілька ключових механізмів для безпечної паралельної обробки даних:

- канали передачі повідомлень - для надсилання завдань потокам;
- атомарні лічильники - для відстеження завершених завдань;
- м'ютекси - для безпечного доступу до спільного буферу пікселів;
- атомарні посилання - для спільного володіння ресурсами між потоками.

Лістинг 3.14 – Структура завдання для потоку

```
struct FrameTask {
    start_y: usize,          // Початкова Y-координата смуги
    end_y: usize,            // Кінцева Y-координата смуги
    buffer: Arc<[GbColor]>,  // Спільний буфер з даними кольорів
Game Boy
    pixel_data: Arc<Mutex<Vec<u8>>>, // Спільний вихідний буфер
RGB даних
    done_counter: SharedCounter,    // Атомарний лічильник
завершених завдань
}
```

Важливо відзначити, що потоки створюються один раз при ініціалізації вікна та продовжують існувати протягом всього життєвого циклу емулятора, очікуючи нові завдання через канали (лістинг 3.15). Це дозволяє уникнути накладних витрат на постійне створення та знищення потоків.

Лістинг 3.15 – Створення потоків та їх логіки виконання

```
 for _ in 0..numthreads {
        let (tx, rx) = channel::<FrameTask>();
            thread_senders.push(tx);
            std::thread::spawn(move || {
              while let Ok(task) = rx.recv() {
                  let mut local = vec![0u8; X_RES as usize *
(end_y - start_y) * 3];
                    for y in start_y..end_y {
                        for x in 0..X_RES as usize {
                         let index = x + y * X_RES as usize;
                         let color = buffer[index].to_color();
                         let pixel_index = (y - start_y) * X_RES
as usize * 3 + x * 3;
                         local[pixel_index] = color.r;
                         local[pixel_index + 1] = color.g;
                         local[pixel_index + 2] = color.b;
                     }
                }
                    let mut pd = pixel_data.lock().unwrap();
                    let offset = start_y * X_RES as usize * 3;
                       pd[offset..offset +
local.len()].copy_from_slice(&local);
                done_counter.fetch_add(1, Ordering::SeqCst);
               }
            })}
```

4 ОЦІНКА ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ПАРАЛЕЛІЗАЦІЇ

Під час розробки програмного емулятора консолі Game Boy було виявлено, що, попри зовнішню простоту її апаратної архітектури, створення точної моделі роботи цієї системи вимагає високого рівня уваги до деталей. Особливо критичним аспектом виявилася потреба у суворому дотриманні внутрішньої послідовності виконання інструкцій. Консоль складається з кількох ключових компонентів – центрального процесора (CPU), відеопідсистеми (PPU), звукового модуля (APU), системних таймерів і пам’яті – які функціонують у постійній взаємодії. Кожен компонент синхронізується з іншими через спільну часову базу, що обумовлює жорстку вимогу до точності таймінгів і порядку доступу до ресурсів.

4.1 Стратегія часткової паралелізації

Було обрано стратегію часткової паралелізації – тобто збереження послідовної структури основного циклу емуляції (CPU, пам’ять, PPU, таймери) при виносі другорядних або незалежних за часом компонентів у окремі робочі потоки. Зокрема, найбільшим вузьким місцем у продуктивності став етап підготовки графічного буфера до відображення. У внутрішньому представленні зображення зберігається у форматі, зручному для PPU, але не придатному для прямого виводу. Для передачі даних у графічний API (в даному випадку SDL2) необхідна попередня трансформація піксельних даних – конвертація з палітри Game Boy у формат ARGB або RGBA, створення текстур, оновлення буфера тощо. Ця операція сама по собі не є складною, але включає обробку тисяч пікселів кожні 1/60 секунди, і є повністю незалежною від логіки CPU.

У зв’язку з цим було прийнято рішення реалізувати спеціальний механізм паралельної обробки відеобуфера. Головний потік формує

зображення, після чого передає його у чергу для обробки окремими потоками, кожен з яких відповідає за частину зображення. Така схема дозволила рівномірно розподілити навантаження між доступними ядрами CPU та уникнути блокування основного потоку. Для реалізації цього підходу використовувалися базові засоби паралелізації, доступні в мові програмування Rust: std::thread для створення робочих потоків, Arc та Mutex для організації доступу до спільних структур, а також AtomicUsize для лічильників і сигналізації завершення роботи.

4.2 Оцінка ефективності реалізованого підходу

Щоб об'єктивно оцінити ефективність реалізованої оптимізації, було проведено серію тестувань на відкритих демонстраційних ROM-файлах, які мають відкритий вихідний код і не порушують авторських прав.

4.2.1 Опис експериментального стенду

Для тестування використовувався персональний комп’ютер із наступними характеристиками:

- процесор – AMD Ryzen 7 4800H;
- відеокарта – AMD Radeon Vega 8;
- оперативна пам’ять – 16 Гб;
- операційна система – Windows 10 22H2.

4.2.2 Тестування додатку

Під час вимірювань використовувалася основна метрика – середня кількість кадрів в секунду (FPS), що дозволяє оцінити загальну швидкодію емулятора в умовах максимально близьких до реального використання. Тестування проводилося за ідентичних умов до та після впровадження

паралелізації (таблиця 3.1). Як видно з таблиці, приріст продуктивності у вибраних демо-програмах коливається в межах від 59.5% до 146.7%, залежно від складності сцени та навантаження на систему рендерингу. Найбільше покращення спостерігається в графічно насичених демо з великою кількістю елементів на екрані.

Таблиця 4.1 - Порівняння FPS у вибраних програмах до та після застосування паралелізації графічного рендерингу

Назва	Середній FPS до оптимізації	Середній FPS після оптимізації	Приріст (%)
BobtInvite	55.5	88.5	59.5%
Alt Too	60.0	148.0	146.7%
Wyrmhole	49.0	79.5	62.2%
Rex Run	56.5	128.5	127.4%
3D Demo	65.2	148.3	127.5%

4.3 Переваги часткової паралелізації

На момент написання роботи в емуляторі ще не реалізовано модуль звуку (APU) або можливість локального мережевого з'єднання для гри в кооперативному режимі (Link Cable multiplayer). Водночас навіть за відсутності цих двох підсистем вдалося досягти значного приросту продуктивності виключно завдяки виносу графічного рендерингу в окремі потоки.

Це свідчить про ефективність обраного підходу: звільнення основного потоку від обробки візуальних даних дало змогу значно підвищити частоту кадрів без зміни самої логіки емуляції. Такий підхід не лише покращує швидкодію, а й залишається сумісним із вимогами до точності синхронізації, що є критичним для моделювання оригінального апаратного забезпечення Game Boy.

Крім того, побудована архітектура є масштабованою – з’являється потенціал для виносу в окремі потоки інших підсистем, таких як обробка звуку, введення/виведення чи мережеві функції.

4.4 Недоліки часткової паралелізації

Попри досягнутий приріст продуктивності, основний цикл емуляції залишився послідовним і централізованим. Це означає, що всі критично важливі компоненти – CPU, обробка таймерів та доступ до пам’яті – досі виконуються в одному потоці. Відповідно, у довгостроковій перспективі саме цей цикл стає головним обмеженням масштабування.

Крім того, багатопоточна архітектура ускладнює структуру програми: з’являється потреба у синхронізації даних (черги, блокування, атомарні лічильники), що ускладнює налагодження та збільшує ризик гонок даних у разі помилок в реалізації.

ВИСНОВКИ

У процесі розробки емулятора Game Boy було доведено, що навіть у системах з високим ступенем залежності між підсистемами – таких як центральний процесор, графічний процесор, звуковий модуль та контролери вводу – можливо ефективно застосовувати паралелізм. Хоча на перший погляд повна паралелізація виглядає привабливою – адже вона обіцяє значне зростання продуктивності за рахунок кращого використання ресурсів багатоядерних процесорів – на практиці її застосування у випадку точного емулятора виявилося недоцільним. Причиною цьому є жорсткі вимоги до синхронізації та дотримання таймінгів, які є критично важливими для правильної роботи ігор та демонстрацій, особливо тих, що використовують нестандартні техніки відображення або акуратно вирахувані затримки.

Натомість була обрана стратегія часткової паралелізації – тобто винесення обчислювально затратних, але логічно незалежних підзадач у фонові потоки. До таких задач належать, зокрема, рендеринг відеобуфера, обробка вводу/виводу, а в подальшому – синтез звуку (APU) та емуляція мережевої взаємодії (Link Cable multiplayer). Такий підхід дозволяє досягти балансу між збереженням точності моделювання та покращенням загальної продуктивності, зменшуючи навантаження на головний емуляційний цикл.

Факт того, що вдалося досягти істотного приросту FPS без втручання в логіку емуляції, свідчить про правильність обраної точки прикладення паралелізму. Винесення рендерингу у фоновий потік дало змогу уникнути простоїв у головному потоці, не порушивши послідовності виконання інструкцій чи взаємодії компонентів. Це особливо важливо у випадках, коли моделювана система є детермінованою – тобто повинна реагувати однаково за однакових умов.

Зібраний у межах цього проєкту досвід є цінним не лише у вузькому контексті Game Boy – він демонструє загальні принципи, які можуть бути

застосовані до проєктування емуляторів інших платформ, складних симуляцій або систем реального часу. В усіх таких системах існує розмежування між критичними задачами та некритичними. Успішна ідентифікація та відокремлення таких задач – ключ до масштабованої та стабільної архітектури.

Обрана модель вже продемонструвала свій потенціал до розширення. У майбутньому вона може бути адаптована до складніших сценаріїв – таких як підтримка кількох платформ, впровадження багатопоточних аудіо та мережевих обробників, або навіть побудова гнучкого конфігураційного середовища, де користувач сам обирає, які частини системи виносити в окремі потоки. Такий рівень гнучкості відкриває двері до використання емулятора не лише на персональних комп’ютерах, але й на вбудованих системах, мобільних пристроях, чи навіть у хмарних середовищах.

Загалом, стратегія часткової паралелізації виявилася не лише технічно виправданою з точки зору продуктивності, але й методологічно обґрунтованою – вона дозволяє розвивати проєкт поступово, мінімізуючи ризики і не потребуючи кардинального переписування вже реалізованої логіки. Це дозволяє зберігати сталість, стабільність та передбачуваність поведінки, які є основою якісного емулятора.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Thibeault C., Hervé J.-Y. Object Detection in Emulated Console Games. Annual International Conference on Computer Games, Multimedia and Allied Technology. 2015. URL: https://doi.org/10.5176/2251-1679_cgat15.11 (дата звернення: 10.05.2025).

2. BGB Official Website. BGB. URL: https://bgb.bircd.org/ (дата звернення: 20.05.2025).

3. VisualBoyAdvance Official Git Repository. URL: https://github.com/visualboyadvance-m/visualboyadvance-m. (дата звернення: 20.05.2025).

4. mGBA Official Website. mGBA. URL: https://mgba.io/faq.html (дата звернення: 21.05.2025).

5. SameBoy Official Website. SameBoy. URL: https://sameboy.github.io/ (дата звернення: 21.05.2025).

6. PQEMU: A Parallel System Emulator Based on QEMU / J.-H. Ding et al. 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), Tainan, Taiwan, 7–9 December 2011. 2011. URL: https://doi.org/10.1109/icpads.2011.102 (дата звернення: 10.05.2025).

7. COREMU / Z. Wang et al. the 16th ACM symposium, San Antonio, TX, USA, 12–16 February 2011. New York, New York, USA, 2011. URL: https://doi.org/10.1145/1941553.1941583 (дата звернення: 11.05.2025).

8. Graphite: A distributed parallel simulator for multicores / J. E. Miller et al. 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), Bangalore, 9–14 January 2010. 2010. URL: https://doi.org/10.1109/hpca.2010.5416635 (date of access: 11.05.2025).

9. Rust Programing Language Official Website. Rust Lang. URL: https://www.rust-lang.org/ (дата звернення: 20.05.2025).

10. SDL2 Official Wiki. SDL2 Wiki. URL:

https://wiki.libsdl.org/SDL2/FrontPage (дата звернення: 20.05.2025).

11. Visual Studio Code Official Website. Visual Studio Code. URL: https://code.visualstudio.com/ (дата звернення: 20.05.2025).

12. Chacon S., Straub B. Pro Git Ebook. Git SCM. URL: https://git-scm.com/book/en/v2 (date of access: 20.05.2025).

13. Behavioral Patterns: State Machine. Refactoring Guru. URL: https://refactoring.guru/design-patterns/state (дата звернення: 20.05.2025).

14. Nichols C., Klabnik S. Rust Programming Language. 2nd ed. No Starch Press, Incorporated, 2019. 560 p.