МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ДІДЕНКО МАРИНА МИХАЙЛІВНА

Допускається до захисту:
В.о. завідувача кафедри
прикладної математики та
кібербезпеки,
_____Луценко А. В.
«__»_____20__р.

РОЗРОБКА СХЕМИ ПЕРЕВІРКИ ДОСТУПУ ДО ПАМ'ЯТІ, ЯКА ПРАЦЮЄ З ІНФОРМАЦІЄЮ ПРО ХИБНІ ДАНІ В КЕШІ

Спеціальність 105 Прикладна фізика та наноматеріали

Кваліфікаційна (магістерська) робота

Науковий керівник:
Загоруйко Л. В.,
к.т.н., доцент,
доцент кафедри прикладної
математики та кібербезпеки
_____
(підпис)

Оцінка : ___/ ___ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця 2024

АНОТАЦІЯ

Діденко М.М. розробка схеми перевірки доступу до пам'яті, яка працює з інформацією про хибні дані в кеші. Спеціальність 105 «Прикладна фізика та наноматеріали», Освітня програма «Технології інтернету речей». Донецький національний університет імені Василя Стуса, Вінниця, 2024.

У роботі досліджено архітектуру комп’ютера, кеш-пам’ять та методи забезпечення цілісності даних у сучасних обчислювальних системах. Розглянуто типи архітектур, зокрема Гарвардську та фон Нейманівську, а також методи перевірки даних, такі як контрольна сума, ECC пам'ять і механізми кеш-когерентності. Проаналізовано вплив схем доступу до пам’яті на продуктивність системи та особливості процесорів на основі фон Нейманівської архітектури. Запропоновано та вдосконалено схему перевірки доступу до пам’яті, спрямовану на виявлення й виправлення помилок у кеш-пам’яті, що підвищує швидкість і ефективність роботи систем.

Клочові слова : схема доступу до пам’яті, кеш-пам’ять, архітектура фон Неймана, продуктивність системи, процесори Pentium.

80 с., 5 табл., 43 рис., 5 дод., 37 джерел.

Didenko M.M. Development of a memory access verification scheme that works with information about false data in the cache. Specialty 105 “Applied Physics and Nanomaterials”, Educational program “Internet of Things Technologies”. Vasyl' Stus Donetsk National University, Vinnytsia, 2024.

The paper investigates computer architecture, cache memory and methods of ensuring data integrity in modern computing systems. The types of architectures, in particular Harvard and von Neumann, as well as data verification methods, such as checksums, ECC memory, and cache coherence mechanisms, are considered. The influence of memory access schemes on system performance and features of processors based on the von Neumann architecture is analyzed. A memory access verification scheme aimed at detecting and correcting errors in the cache memory is proposed and improved, which increases the speed and efficiency of systems.

Key words: memory access scheme, cache memory, von Neumann architecture, system performance, Pentium processors.

80 p., 5 tables, 43 figures, 5 appendix, 37 sources.

ЗМІСТ

ВСТУП

Розвиток комп’ютерних технологій є головним аспектом прогресу у багатьох сферах людської діяльності, починаючи від наукових досліджень та інженерних розробок і закінчуючи комунікацією, розвагами та управлінням складними системами. У міру того, як технології стають більш потужними та доступними, зростають і вимоги до їхньої продуктивності, надійності та стійкості до збоїв. Одним із найважливіших аспектів сучасних обчислювальних систем є ефективне управління пам’яттю, яка виступає критичним компонентом у забезпеченні швидкодії, стабільності роботи та безпеки даних.

Сучасні обчислювальні завдання стають дедалі складнішими та масштабнішими, що зумовлює потребу в обробці великих обсягів даних у найшвидші терміни часу. У цьому контексті пам’ять є ключовим елементом, що забезпечує взаємодію між процесором і даними. Питання цілісності та достовірності даних, які зберігаються, обробляються та передаються в пам’яті, також дедалі стають важливішими і набувають особливого значення. Це зумовлено не лише прагненням уникнути витоку важливої інформації, але й потребою гарантувати коректність обчислень і стабільність роботи комп’ютерної системи в цілому.

Кеш-пам’ять відіграє центральну роль у сучасних обчислювальних системах. Завдяки своїй швидкодії вона дозволяє значно скоротити час доступу до часто використовуваних даних, що суттєво підвищує продуктивність процесора. Зокрема, використання кешу забезпечує оптимізацію роботи системи, зменшуючи затримки, які виникають під час звернення до основної пам’яті. Проте кеш-пам’ять має свої технічні виклики, особливо щодо забезпечення цілісності даних. Через особливості апаратної організації, енергетичні впливи, електромагнітні перешкоди або помилки при записі та зчитуванні можуть виникати збої різного характеру, які призводять до спотворення інформації.

Це завдання стає ще більш актуальним в умовах багатоядерних процесорів і багаторівневих ієрархій пам’яті, де взаємодія між ядрами та пам’яттю вимагає синхронізації й гарантії точності даних. Крім того, поширення паралельних

обчислень та обробки даних у реальному часі вимагає, щоб такі механізми були не лише надійними, а й швидкими, аби не впливати на загальну продуктивність системи.

Актуальність роботи. Сучасна обчислювальна техніка постійно вдосконалюється, що зумовлює необхідність оптимізації взаємодії між різними рівнями пам’яті комп’ютерів. Архітектура комп'ютерів та ефективність роботи кеш-пам'яті є важливими аспектами для забезпечення високої продуктивності системи. Надійність збереження та обробки даних стає критичним фактором, особливо в умовах стрімкого зростання обсягів інформації.

Мета роботи – розробити схему перевірку доступу до пам’яті, яка працює з інформацією про хибні дані в частинах кеш

Для досягнення поставленої мети потрібно виконати наступні **завдання:**

- Аналіз відомих архітектур комп'ютерів та методів перевірки даних;
- Порівняння схем доступу до пам’яті та процесорів з фон Нейманівською архітектурою;
- Розробка схеми перевірки доступу до пам’яті, що працює з інформацією про хибні дані в частинах кеш та коду, що симулює роботу схеми

Об’єкт дослідження – є архітектура комп'ютера, системи управління пам’яттю та механізми перевірки даних у комп’ютерних системах.

Предмет дослідження – є методи перевірки цілісності даних та оптимізації доступу до пам’яті, включаючи технології роботи з кеш-пам’яттю та забезпечення її когерентності, а також вплив цих методів на продуктивність системи.

Практичне значення дослідження полягає в можливості застосування отриманих результатів для підвищення продуктивності та надійності сучасних комп’ютерних систем. Розроблені схеми перевірки доступу до пам’яті, що враховують роботу з хибними даними в кеш-пам’яті, можуть бути інтегровані у проектування процесорів і систем з високими вимогами до швидкодії.

РОЗДІЛ 1. АРХІТЕКТУРА КОМП'ЮТЕРА ТА КЕШ-ПАМ'ЯТЬ: ОГЛЯД МЕТОДІВ ПЕРЕВІРКИ ДАНИХ

1.1. Архітектура комп'ютера та типи

Архітектура комп'ютера — це структура і поведінка комп'ютера, яку бачить програміст на рівні мови асемблера. Вона включає формати даних, команд, методи адресації, систему команд, а також організацію процесора, пам'яті та пристроїв введення-виведення. Дані зазвичай представлені у двійковій системі числення, хоча для спрощення можуть використовуватись інші форми, як-от вісімкова, шістнадцяткова або символьна. Команди складаються з коду операції та адресної частини, що вказує адреси операндів. Система команд має бути повною, щоб забезпечувати виконання всіх необхідних завдань, і ортогональною, без зайвих команд. Існують комп'ютери з комплексною (CISC), спрощеною (RISC), доповненою та спеціалізованою системою команд.

Архітектура також визначає методи адресації, місце розміщення даних і типи пам'яті, як-от стекова, акумуляторна чи з регістрами загального користування, а пам'ять може бути ієрархічною (з використанням кешування) або лінійною. Команди можуть мати різні формати, бути одноадресними, двоадресними чи триадресними, з різною розрядністю та структурою.

Організація компонентів також різниться: архітектура може підтримувати паралельну або конвеєрну обробку даних, різні способи організації пам'яті та введення-виведення. Архітектура комп'ютера визначає ключові характеристики системи, впливаючи на продуктивність, обсяг пам'яті, вартість, надійність, типи задач, які можна виконувати, та інші споживчі параметри. Загалом, архітектура комп'ютера — це інтерфейс між апаратним і програмним забезпеченням, який визначає параметри та принципи взаємодії різних компонентів системи [1].

Можна виділити основні типи архітектур по використанню пам’яті[2]:

- Гарвардська архітектура : було розглянуто у роботі Aurélien Francillon та Claude Castelluccia та Richard Pawson у своїй статі;

- Архітектура фон Неймана: Rudolf Eigenmann та David J. Lilja детально розглядають її у своїй роботі; також чудово продемонтровано аспекти даної архітектури у роботі Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, і Yoav Etsion

1.1.1. Гарвардської архітектура

Термін "гарвардська архітектура" не був введений до 1970-х років і спочатку не використовувався для опису машин, розроблених у Гарвардській обчислювальній лабораторії (HCL) під керівництвом Говарда Айкена. Згодом цей термін був застосований до цих машин, які мали окремі пам'яті для інструкцій і даних [3].

Гарвардська архітектура – архітектура, головною відмінністю в якої є те, що дані та оператори (алгоритм) зберігаються окремо, тобто фізично розділена пам’ять для програм і даних, до яких процесор звертається через різні шини. Через це підвищується продуктивність системи через паралельну обробку даних та команд [4].

Рис. 1.1 – Гарвардська архітектура

Ядро комп'ютера за архітектурою Джона фон Неймана складається з процесора та оперативної пам'яті. Бажано, щоб ці дві частини ядра працювали синхронно, не сповільнюючи одна одну, тобто мали однакову швидкодію. Однак на практиці пам'ять зазвичай працює значно повільніше за процесор, і з розвитком інтегральних технологій цей розрив лише збільшується. Щоб зменшити цей розрив, застосовують структурні підходи, наприклад, збільшують розрядність слова пам'яті. Такий підхід реалізовано в гарвардській архітектурі, яка використовує два окремі запам'ятовувальні пристрої. Це дозволяє паралельно виконувати операції вибору команд програми та операції читання і запису даних і результатів обчислень [1].

Основними особливостями даної архітертури є:

- Окремі шини – ця архітектура має дві різні шини (одна для передачі даних, інша – інструкцій). Саме ці дві шини дозволяють мати одночасний доступ;
- Розділена пам’ять – дані та інструкції зберігаються в окремих пам’ятях і завдяки цьому можна забезпечити відсутність конфліктів при доступі до них;
- Паралельний доступ до даних та виконання інструкції – через розділені шини, процесор може паралельно отримувати дані та виконувати інструкції.

Головним недоліком даної архітектури є важкість реалізації, оскільки для кожного з запам’ятовуючих пристроїв необхідний свій контролер та шина. Це призводить до великої кількості з’єднань в системі і негативно впливає на складність реалізації та проектування. Якщо ж говорити і про інші недоліки даної архітектури, то до них можна віднести : збільшення енергоживлення (через те що шини та пам’яті є розділеними, то вони й відповідно потребують більше енергії і це може бути недоліком для систем з обмеженими ресурсами), менша гнучкість (це мається на увазі те, що оскільки пам’яті розділені, то для пам’яті даних і пам’яті інструкцій розмір жорстко визначений і якщо для однієї з цих пам’ятей потрібен більший розмір, то його досить складно перерозподілити) [5].

У сучасному контексті термін "гарвардська архітектура" часто застосовується до мікропроцесорів RISC, які кешують інструкції та дані окремо, хоча це не є справжньою гарвардською архітектурою. Richard Pawson стверджує, що це більше схоже на модифіковану архітектуру фон Неймана, оскільки сучасні системи потребують інтеграції з операційними системами. Робота Pawson-а також підкреслює, що існує міф про те, що гарвардська архітектура є кращою за архітектуру фон Неймана. Це сприйняття може вводити в оману студентів і фахівців, оскільки сучасні комп'ютери не можна просто класифікувати як "фон Неймана" або "гарвардські". Pawson зазначає, що хоча гарвардська архітектура має свої переваги, її застосування в історії комп'ютерних наук не було настільки значним, як це часто вважається. Він підкреслює, що багато сучасних комп'ютерних систем фактично реалізують принципи, які можна вважати модифікацією архітектури фон Неймана [3].

Таким чином, стаття ставить під сумнів традиційні уявлення про гарвардську архітектуру, підкреслюючи її історичний контекст, еволюцію терміна та його сучасні інтерпретації.

Важливо відмітити, що гарвардська архітектура широко використовується в спеціалізованих системах, де потрібні мінімальні затримки та висока швидкодія. До прикладу таких систем можуть бути : мікроконтролери, цифрові сигнальні процесори, системи реального часу, графічні процесори, космічні технології, модулі з обмеженим енергоспоживанням і т.д.

Стосовно безпеки даної архітектури, вона вважається більш безпечної ніж архітектура фон Неймана (буде розглянуто нижче). Ключовими факторами переваги гарвардської архітектури в безпеці є : розділена пам’ять (оскільки дані та інструкції зберігаються в різних пам’ятях, то зловмисний код не може легко змінити інструкції безпосередньо через маніпуляції з даними, як це можливо в фон Нейманівській архітектурі. Це знижує ймовірність атак типу code injection (впровадження коду), захист від переповнення буфера (У гарвардській архітектурі переповнення буфера не може так легко призвести до виконання довільного коду, оскільки дані та інструкції фізично розділені. Це може ускладнити проведення атак,

які використовують переповнення буфера для маніпуляції інструкціями.), більший контроль над доступом (Завдяки окремим шинам і розділеній пам'яті гарвардська архітектура може забезпечити кращий контроль доступу до інструкцій і даних, і саме це забезпечує захист від несанкціонованого доступу).

Але якою б безпечною не здавалась дана архітектура, вона теж має свої недоліки, про які розповідають у своїй статті Aurélien Francillon та Claude Castelluccia. У розділі 2.1 "The AVR architecture" автори зазначають, що в мікроконтролерах гарвардської архітектури інструкції і дані зберігаються в фізично відокремлених пам'ятях. Це означає, що дані не можуть бути виконані, оскільки процесор може завантажувати інструкції лише з пам'яті програми. Це створює враження, що модифікація програми віддалено є неможливою, оскільки для цього потрібен фізичний доступ до пам'яті . У розділі 4.2 "Exploitation of sensor nodes" автори обговорюють, як традиційні атаки, такі як переповнення буфера, не можуть бути застосовані до сенсорів на основі гарвардської архітектури, таких як Micaz. Вони підкреслюють, що стандартні атаки, які виконують код, ін’єктований у стек, є неможливими через архітектурні обмеження . У розділі 5.3 "Incremental attack description" автори описують, як їх атака намагається обійти ці обмеження, використовуючи мета-гаджети та фейковий стек для ін'єкції коду в пам'ять програми. Це свідчить про те, що, незважаючи на те, що гарвардська архітектура ускладнює ін'єкцію коду, існують способи обійти ці обмеження [6].

Таким чином, автори підкреслюють, що хоча гарвардська архітектура забезпечує певний рівень безпеки, вона не є абсолютно непроникною для атак, і існують способи, якими зловмисники можуть скористатися вразливостями системи.

1.1.2. Архітектура фон Неймана

Архітектура фон Неймана (модель фон Неймана або Принстонська архітектура) - це комп'ютерна архітектура, заснована на Першому проекті звіту про EDVAC,[1] написаному Джоном фон Нейманом у 1945 році. Документ описує

архітектуру електронного цифрового комп'ютера з такими компонентами: процесорний блок з арифметико-логічним блоком і регістрами процесора, блок керування, що включає регістр інструкцій і лічильник програм, пам'ять, що зберігає дані та інструкції, зовнішній запам'ятовуючий пристрій, механізми вводу і виводу [7].

Процесор (або центральний обчислювальний блок) виконує всі обчислення та керує обробкою даних. Він складається з арифметико-логічного блоку (ALU) для виконання арифметичних і логічних операцій, а також блоку керування, який управляє послідовністю виконання команд. Крім того, процесор містить регістри для зберігання проміжних даних і результатів [8].

Пам'ять у цій архітектурі одночасно зберігає як інструкції (програмний код), так і дані. Вона розділена на комірки, кожна з яких має унікальну адресу, за якою зберігаються або інструкції, або дані. Одна спільна шина використовується для передавання інструкцій і даних між процесором, пам'яттю та іншими пристроями. Ця загальна шина є основою для відомого обмеження, що називається "вузьким місцем фон Неймана" (англ. Von Neumann bottleneck) [9] . Це означає, що, оскільки одна шина використовується і для інструкцій, і для даних, процесор не може одночасно завантажувати інструкцію та виконувати операцію з даними, що створює затримки.

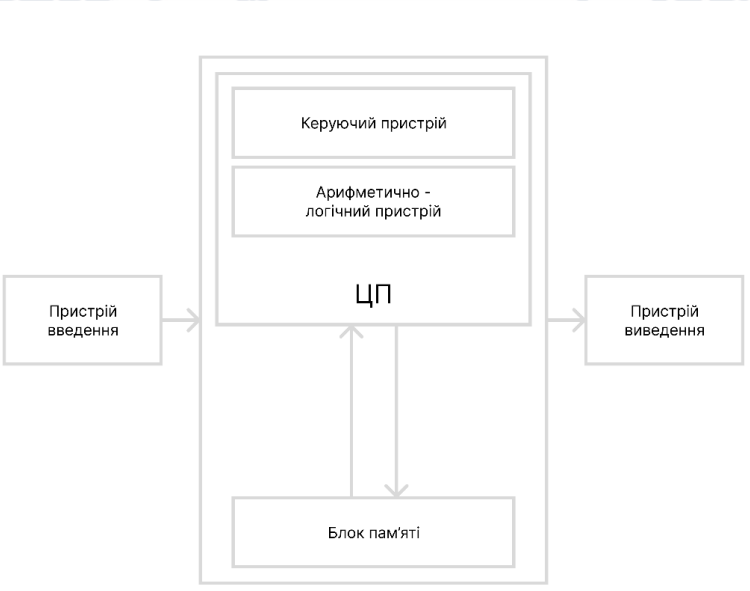

Рис. 1.2. – Архітектура фон Неймана

Серед переваг архітектури фон Неймана — простота реалізації, адже зберігання інструкцій і даних в одній пам'яті спрощує структуру комп'ютера та робить його дешевшим у виробництві. Універсальність дозволяє створювати програми, які змінюють самі себе під час виконання, що є важливою властивістю для сучасних обчислень. Також таку архітектуру легко програмувати, оскільки всі інструкції програми зберігаються в пам'яті, і комп'ютер може послідовно їх виконувати, що полегшує програмування.

Водночас, у архітектури фон Неймана є і недоліки. Найголовніший з них — це "вузьке місце", яке виникає через використання однієї шини для інструкцій і даних, що обмежує швидкість обробки та викликає затримки. Крім того, спільна пам'ять для інструкцій і даних може призводити до конфліктів та затримок, особливо в багатозадачних системах. Через лінійне виконання команд та відсутність високого рівня паралелізму фон-нейманівська архітектура обмежує швидкодію, що є критичним у сучасних обчисленнях [8].

Rudolf Eigenmann та David J. Lilja підкреслює, що архітектура фон Неймана стала основою для більшості сучасних комп'ютерів, які виконують послідовність інструкцій, що оперують на єдиному потоці даних. Він також згадує, що концепції, пов'язані з архітектурою фон Неймана, були розроблені не лише самим фон Нейманом, а й багатьма іншими особами, що викликало певні суперечки серед істориків технологій щодо справжніх джерел цих ідей. Автори зазначають, що для подальшого покращення продуктивності комп'ютерних систем необхідно впроваджувати нові інноваційні техніки, такі як паралельне виконання та спекулятивне виконання. У сучасних комп'ютерних системах використовується комбінація архітектур Гарварда та фон Неймана в їхній пам'яті. Архітектура Гарварда використовується на чіпі для кешу, тоді як основна пам'ять поза чіпом використовує архітектуру фон Неймана. Це дозволяє одночасний доступ до інструкцій та даних, але також викликає проблему несумісності кешу, коли одні й ті ж адреси можуть мати різні значення в кешах та основній пам'яті. Автори також акцентують увагу на проблемі кешу, яка виникає в системах з окремими кешами для даних та інструкцій. Це може призвести до несумісності значень, що

зберігаються в різних кешах та основній пам'яті, що є критичним для коректного виконання програм [10].

Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, та Yoav Etsion зазначають, що фон Неймановська архітектура має певні переваги, зокрема можливість експлуатації обмеженого рівня паралелізму інструкцій (ILP), паралелізму на рівні даних (DLP) та паралелізму на рівні потоків (TLP). Однак, для досягнення DLP і TLP програміст або компілятор повинен явно виражати ці аспекти. Основною перевагою фон Неймановської моделі є її здатність до обробки традиційних контрольних потоків, що робить її популярною для багатьох програмних застосувань, незважаючи на її обмеження в паралелізмі. Автори підкреслюють, що фон Неймановська архітектура має дві основні характеристики, які обмежують її паралелізм: наявність програмного лічильника та глобальної оновлювальної пам'яті. Ці особливості призводять до двох фундаментальних обмежень: затримок пам'яті та синхронізації потоків.

Фон Неймановські машини виконують інструкції послідовно, що обмежує їх здатність до паралельного виконання. Хоча архітектура може використовувати деякі обмежені форми паралелізму, такі як ILP, DLP і TLP, вона все ще залишається в основному послідовною через свою залежність від програмного лічильника.

Таким чином, основні особливості фон Неймановської архітектури включають:

1. Виконання інструкцій у послідовному порядку;
2. Використання програмного лічильника для управління виконанням;
3. Глобальна оновлювальна пам'ять, що може призводити до затримок і проблем з синхронізацією.

1.2. Методи перевірки даних

У кеш-пам'яті існують різні методи перевірки даних, щоб забезпечити їхню правильність та унеможливити помилки:

- Метод контрольної суми: William досить детально проаналізував цей метод у своїй роботі;
- Метод перевірки тегів: завдяки роботі Alan Jay ми маємо змогу з цим детально ознайомитись.
- Метод перевірки парності : у роботу було описано Ihsen, Smail, Fadi та Mohamed та роботі William;
- ECC (Error-Correcting Code) Memory : Daniele, Nicola, Michael та Cecilia розглянули цю технологію у своїй роботі;
- Кеш-когерентність (Cache Coherence) : у своїй роботі це питання дослідили Singh, Shriraman, Fung, O'Connor, Aamodt.

1.2.1. Метод контрольної суми

У кеш-пам'яті процесора використовується метод контрольних сум для забезпечення цілісності даних. Цей метод полягає в тому, що для кожного блоку даних у кеш-пам'яті обчислюється контрольна сума. Контрольна сума - це число, яке обчислюється за допомогою певного алгоритму на основі даних у блоці. Якщо дані в блоці змінюються, то контрольна сума також зміниться.

Коли процесор звертається до даних у кеш-пам'яті, він також отримує контрольну суму для цього блоку даних. Потім процесор обчислює контрольну суму даних, які він отримав з кеш-пам'яті. Якщо обчислена контрольна сума збігається з контрольною сумою, яка була отримана з кеш-пам'яті, то дані вважаються цілісними. Якщо контрольні суми не збігаються, то дані вважаються пошкодженими[11].

William звертають увагу на ефективність контрольних сум у своїй роботі. Зокрема, вони порівнюють контрольні суми з циклічними надлишковими кодами (CRC) за допомогою метрик ймовірності невиявленої помилки (Pud) та Хеммінгової відстані (HD) при різних довжинах кодових слів. Вони надають кількісні порівняння альтернативних схем виявлення помилок. Деякі з їх висновків можна узагальнити наступним чином:

«Додавання за допомогою доповнення до одиниці завжди переважає додавання за допомогою доповнення до двох, оскільки воно захоплює ефекти виконання операції додавання. Будь-який метод додавання значно переважає схеми парності, такі як поздовжня контрольна сума (LRC)».

Цей аналіз показує, що автори вважають контрольні суми важливими для забезпечення надійності даних, а також проводять порівняння їх ефективності з CRC за допомогою конкретних метрик.

Автор також зазначає, що контрольні суми можуть бути ефективним методом виявлення помилок з такими перевагами :

1. Вони можуть бути ефективними у виявленні помилок з використанням певних метрик ефективності, які порівнюються з CRC;
2. Контрольні суми можуть бути ефективними для виявлення помилок для певної кількості бітів у контрольній послідовності;
3. Вони можуть бути простими у реалізації як у програмному, так і у апаратному забезпеченні .

Ці переваги можуть робити метод контрольних сум корисним та безпечнішим для використання в різних системах забезпечення надійності даних[12].

1.2.2 Модель перевірки тегів

Метод тегів - це метод перевірки на валідність в частинах кеш, який використовується для визначення того, чи є дані в кеш-пам'яті актуальними. Актуальні дані - це дані, які не були змінені з моменту їх завантаження в кеш-пам'ять. Неактуальні дані - це дані, які були змінені після їх завантаження в кеш-пам'ять.

Метод тегів заснований на тому, що для кожного блоку даних у кеш-пам'яті зберігається його адреса та тег. Адреса використовується для визначення того, де знаходиться блок даних у кеш-пам'яті. Тег використовується для визначення того, чи є блок даних актуальним.

Коли процесор звертається до даних у кеш-пам'яті, він надає адресу блоку даних. Процесор також отримує тег блоку даних з кеш-пам'яті.

Alan Jay у своїй роботі описує, що теги елементів всіх вибраних наборів порівнюються з реальною адресою з швидкого кеш-буферу (TLB). Якщо збіг відбувається, то рядок, що містить цільові місця, зчитується в зсувний регістр, а статус заміни елементів в кеші оновлюється. Якщо відбувається промах (тобто теги адрес в кеші не збігаються), тоді реальна адреса потрібного рядка передається в основну пам'ять. Інформація про статус заміни використовується для визначення, який рядок видалити з кешу, щоб зробити місце для цільового рядка.

Автор вказує, що цей метод дозволяє значно зменшити час доступу до кешу, оскільки він дозволяє прямо адресувати кеш за віртуальною адресою, уникнувши етап перекладу. Це може значно підвищити швидкість доступу до кеш-пам'яті [13].

1.2.3 Метод перевірки парності

Перевірка парності (parity checking) - це метод, який використовується для виявлення помилок в передачі або зберіганні даних. Він полягає в додаванні додаткового біту до двійкового коду, щоб забезпечити, що загальна кількість одиниць у даних, включаючи біт парності, буде парною (парна парність) або непарною (непарна парність). Це дозволяє виявляти помилки в одному біті [14].

William зазначає, що обчислення парності може бути використане як приклад для обговорення конструкції та оцінки кодів помилок без використання складної математики. При обчисленні парності, бітова послідовність розбивається на групи бітів, і для кожної групи обчислюється парність (кількість одиниць в групі). Якщо кількість одиниць в групі парна, то до групи додається біт зі значенням 0, якщо непарна - зі значенням 1. При отриманні даних, отримувач також розбиває бітову послідовність на групи та обчислює парність для кожної групи. Якщо кількість одиниць в групі не відповідає парності, то відбулася помилка.

Також він звертає увагу на те, що хоча обчислення парності може бути простим методом виявлення помилок, воно не є ефективним для виявлення всіх

типів помилок. Крім того, використання парності не дозволяє виправляти помилки, а тільки виявляти їх. Тому, для більш надійного виявлення та виправлення помилок, можуть бути використані більш складні методи, такі як контрольні суми[15].

Якщо помилка виявляється, оригінальний блок стає застарілим, і замість нього використовується копія. Вони використовують факт, що розмір робочого набору кеш-пам'яті в багатьох вбудованих застосуваннях менше 4-8 КБ, що дозволяє використовувати додаткову площу кеш-пам'яті для виявлення та виправлення можливих помилок, зазначають автори описано Ihsen, Smail, Fadi та Mohamed. У своїй роботі автори представляють архітектуру PmC², яка є новаторською технологією для стійкої кеш-пам'яті і. Вони досліджують можливості виявлення та виправлення помилок на основі метода перевірки парності, що дозволяє працювати під агресивним зниженням напруги без високої ймовірності відмови в кеш-пам'яті. Автори також проводять експерименти, щоб продемонструвати зменшення споживання енергії кеш-пам'яті порівняно з традиційними методами, при цьому зазначають, що втрата продуктивності є незначною.

Автори вважають, що їхня архітектура може бути покращена шляхом розширення досліджень для інших видів застосувань, розробки за допомогою двовимірної перевірки паритету та поєднання з іншими методами виявлення та виправлення помилок [14].

1.2.4 ECC (Error-Correcting Code) Memory

Ця технологія використовується в деяких типах кеш-пам'яті, особливо у високонадійних системах. ECC-пам'ять може виявляти і виправляти помилки, які виникають під час зберігання або передачі даних. Техніка ECC включає додавання додаткових бітів до пам'яті для зберігання зайвої інформації, що дозволяє пам'яті виявляти та виправляти певні типи помилок, які можуть виникати через електричні

або магнітні перешкоди, радіацію або інші фактори навколишнього середовища. Це допомагає забезпечити точність та надійність збережених даних.

ECC-пам'ять зазвичай використовує різні коди корекції помилок, такі як SEC (одноразова корекція помилок) або DED (виявлення подвійних помилок), для забезпечення різних рівнів виявлення та корекції помилок.

Автори статті "Error Correcting Code Analysis for Cache Memory High Reliability and Performance" висловлюють думку про те, що важливо аналізувати та оцінювати різні типи ECC для пам'яті кеша з метою покращення їхньої здатності до виявлення та корекції помилок поза стандартними SEC/DED Hsiao кодами. Вони розглядають різні компроміси, які можна досягти в області накладення, продуктивності та здатності до корекції, що надає проектувальникам уявлення про вибір оптимального ECC та організації коду/сегменту для конкретного застосування.

Автори статті виділяють наступні переваги ECC для пам'яті кеша:

- Забезпечення надійності даних: ECC дозволяє виявляти та коригувати помилки, що можуть виникнути під час зберігання або витягування даних, що забезпечує точність та надійність збережених даних.
- Зменшення впливу помилок на продуктивність: ECC дозволяє зменшити вплив помилок на продуктивність системи, оскільки він дозволяє виявляти та коригувати помилки без необхідності повторного виконання операцій.
- Забезпечення безпеки: ECC дозволяє забезпечити безпеку даних, оскільки він дозволяє виявляти та коригувати помилки, що можуть бути спричинені зовнішніми факторами, такими як радіація або електричні перешкоди.

Підвищення продуктивності: ECC дозволяє підвищити продуктивність системи, оскільки він дозволяє виявляти та коригувати помилки без необхідності повторного виконання операцій, що зменшує час, необхідний для виконання операцій[16].

1.2.5 Кеш-когерентність

Кеш-когерентність - це властивість багатопроцесорних систем, яка забезпечує, що кожен кеш пам'яті в системі містить правильні та актуальні дані. Це означає, що якщо два процесори намагаються змінити одну і ту ж локацію пам'яті одночасно, то всі інші кеші в системі будуть оновлені з правильними даними[17].

Автори зазначають, що кеш-когерентність є важливою для багатопроцесорних систем, де кожен процесор має свій власний кеш пам'яті. Вони також зазначають, що кеш-когерентність може бути реалізована за допомогою різних протоколів, таких як MESI, MSI, MOESI тощо.

В своїй статі автори розглядають і аналізують кеш-когерентність на основі графічного процесора (GPU) і вони кажуть : "Графічні процесорні одиниці (GPU) стали загальнодоступними в обчисленнях високої продуктивності загального призначення. Традиційно обмежені регулярним паралелізмом, останні дослідження показали, що навіть високо неправильні алгоритми можуть отримати значні прискорення на GPU. Крім того, включення багаторівневої ієрархії кешу в останніх GPU звільняє програміста від обтяження управління кешами програмним засобом та подальше збільшує привабливість GPU як платформи для прискорення додатків з неправильними шаблонами доступу до пам'яті. GPU не мають кеш-когерентності та вимагають вимкнення приватних кешів, якщо додаток вимагає, щоб операції з пам'яттю були видимими для всіх ядер." [18].

Згідно з цитатою, автори статті вказують на відсутність кеш-когерентності в GPU та необхідність вимкнення приватних кешів, якщо додаток вимагає, щоб операції з пам'яттю були видимими для всіх ядер. Це може свідчити про те, що автори вважають кеш-когерентність менш ефективним методом для досягнення потрібної видимості операцій з пам'яттю для всіх ядер у GPU.

Автори виявили кілька проблем, пов'язаних з впровадженням традиційних протоколів кеш-когерентності в архітектури GPU:

1. Накладні витрати на трафік кеш-когерентності: традиційні протоколи кеш-когерентності призводять до зайвого трафіку кеш-когерентності для додатків GPU, що містять дані, які не потребують когерентності.

2. Складність управління тимчасовими станами: управління тимчасовими станами для тисяч запитів до пам'яті ускладнює апаратне забезпечення та збільшує складність системи.
3. Потреба у великому обсязі пам'яті для відстеження активних запитів: вимоги до обсягу пам'яті для відстеження тисяч активних запитів до пам'яті можуть бути непрактичними для реалізації.
4. Збільшення складності протоколів та повідомлень: традиційні протоколи кеш-когерентності вводять складність у вигляді тимчасових станів та додаткових класів повідомлень, що може призвести до збільшення споживання енергії.

1.3. Висновок до розділу 1

В даному розділі було оглянуто дві основні архітектури ЕОМ – Гарвардську та фон Неймана. Щоб підсумувати наведене вище, підведемо підсумок через таблицю (таб. 1.1). Переважно архітектура фон Неймана використовується для загальноцільових комп’ютерів (для яких важлива гнучкість та простота реалізації), в той час як гарвардська архітектура використовується для вбудованих систем (в яких важлива швидкодія й паралельна обробка інструкцій та даних).

Також в даному розділі було оглянуто методів перевірку даних що забезпечують цілісність та надійність системи. Щоб підсумувати розглянуті вище методи, пропонується переглянути таблицю 1.2 (див. ДОДАТОК А)

Таблиця 1.1 - Узагальнена різниця Гарвардської та фон Нейманівської архітектури

Характеристика	**Фон Неймана**	**Гарвардська**
Шини	Одна	Дві
Зберігання інструкцій і даних	В одній пам’яті	В окремих пам’ятях
Пропускна здатність	Нижча (через одну шину для інструкцій і даних)	Вища (через різні шини)

Продовження таблиці 1.1

Продуктивність	Обмежена "вузьким місцем фон Неймана"	Більш висока(через паралельну передачу інструкцій і даних)
Складність апаратного забезпечення	Простіша	Складніша (через потребу у двох пам'ятях і двох шинах)
Використання в системах	Загальноцільові комп'ютери, персональні ПК	Вбудовані системи, мікроконтролери, DSP
Затримки при доступі	Затримки через одночасний доступ до інструкцій і даних	Менші затримки, оскільки інструкції і дані доступні паралельно
Енергоефективність	Вища (менше компонентів)	Менш енергоефективна (більше компонентів)
Застосування в багатозадачності	Легше реалізувати завдяки єдиній пам'яті	Ускладнене через розділення інструкцій і даних

На основі даної порівняльної характеристики, за прототип буде обрано архітектуру фон Неймана за прототип, оскільки вона є більш універсальної в роботі та використовується нами у побуті щоденно. У порівняні з Гарвардською архітектурою, архітектура фон Неймана дозволяє ефективно виконувати широкий спектр завдань, за допомогою своєї лінійної структурі та підходу до зберігання й обробки даних.

РОЗДІЛ 2. АНАЛІЗ АРХІТЕКТУР ДОСТУПУ ДО ПАМ’ЯТІ

2.1 Аналіз схем доступу до пам’яті

В даному пункті роботи відбудеться детальному аналізу основних методів доступу до пам'яті на основі огляду, що був проведений у розділі першому. Кожен метод оцінюється за низкою параметрів, включаючи технічні характеристики, переваги, недоліки, і практичне застосування в реальних умовах.

Аналіз прямого доступу до пам'яті (DMA): прямий доступ до пам'яті дозволяє пристроям обмінюватися даними з пам'яттю без прямого втручання центрального процесора. Це знижує навантаження на процесор і підвищує швидкість передачі даних[19].

Технічні аспекти:

1. Швидкість передачі даних: Зазвичай вища, ніж при інших методах доступу через відсутність проміжних операцій з CPU;
2. Пропускна здатність: Залежить від швидкості шини та архітектури системи.

Переваги:

1. Зниження навантаження на CPU: Процесор може виконувати інші завдання під час передачі даних;
2. Підвищення продуктивності: Швидша передача даних і зменшення затримок.

Недоліки:

1. Складність інтеграції: Вимагає підтримки на апаратному рівні та правильної настройки системи;
2. Вразливість до помилок: Неправильна реалізація може призводити до помилок передачі даних або збоїв системи.

Аналіз асоціативного методу доступу (AAM): асоціативний доступ до пам'яті дозволяє системі звертатися до даних, базуючись на вмісті, а не на специфічній адресі пам'яті [20].

Технічні аспекти:

1. Ефективність кешування: Часто використовується у кеш-пам'ятях для швидкого знаходження даних;
2. Час доступу: Може варіюватися в залежності від алгоритмів імплементації та розміру даних.

Переваги:

1. Швидкий пошук даних: Висока швидкість знаходження потрібних даних без потреби знати точну адресу;
2. Гнучкість: Підходить для застосувань, де дані часто змінюються або доповнюються.

Недоліки:

1. Складність імплементації: Вимагає складних алгоритмів та може бути вартісним у реалізації;
2. Обмеження щодо обсягу даних: Може бути неефективним для дуже великих обсягів даних через обмеження кеш-пам'яті.

Аналіз послідовного (SAM) та випадкового (RAM) методів доступу: дані читаються або записуються послідовно. Часто застосовується у стрімінгових медіа та бекап-системах [21, 22].

Випадковий доступ (RAM): Доступ до даних відбувається без будь-якої попередньої послідовності, що дозволяє швидке звертання до будь-якої області пам'яті.

Технічні аспекти:

1. Час доступу: SAM має більшу затримку порівняно з RAM, де доступ майже миттєвий;
2. Пропускна здатність: RAM забезпечує високу пропускну здатність, в той час як SAM може бути обмеженим швидкістю медіа.

Переваги та недоліки:

1. SAM: Вище зазначені технічні аспекти є перевагами при застосуванні в певних сценаріях і недоліками в інших;

2. RAM: Висока гнучкість та швидкість доступу є ключовими перевагами, але вартість і потреба в енергії для підтримки можуть бути визначені як недоліки.

Зробивши цей аналіз схем доступу до пам’яті, можна занести результати до таблиці 2.1 для кращого розуміння та візуалізації результатів.

Таблиця 2.1 - Результати аналізу схем доступу до пам’яті

Метод доступу	Технічні аспекти	Переваги	Недоліки
Прямий доступ (DMA)	Висока швидкість передачі даних; Залежить від швидкості шини та архітектури	Зниження навантаження на CPU; Підвищення продуктивності	Складність інтеграції; Вразливість до помилок
Асоціативний доступ (AAM)	Ефективність кешування; Час доступу може варіюватися	Швидкий пошук даних; Гнучкість	Складність імплементації; Обмеження щодо обсягу даних
Послідовний доступ (SAM)	Час доступу може бути вищим; Залежний від швидкості медіа	Ефективний для стрімінгових медіа та бекапів	Низька гнучкість у випадковому доступі до даних
Випадковий доступ (RAM)	Миттєвий доступ до будь-якої частини пам'яті; Висока пропускна здатність	Висока гнучкість; Швидкий доступ до даних	Висока вартість; Велика потреба в енергії

2.2 Аналіз порівняльної ефективності схем доступу

В даному підрозділі порівнянню ефективності різних методів доступу до пам'яті, щоб зрозуміти, як вони впливають на продуктивність системи в різних умовах використання. Аналіз включає порівняння часу доступу, швидкості

передачі даних, і впливу на загальну пропускну здатність системи. Кожен метод оцінюється за критеріями, такими як ефективність, вартість і підходящість для конкретних операційних завдань.

Порівняння часу доступу:

1. DMA: Час доступу мінімальний, оскільки CPU не залучений безпосередньо, що робить цей метод ідеальним для високошвидкісних операцій;
2. AAM: Час доступу змінний, але може бути оптимізований за допомогою ефективного кешування;
3. SAM: Час доступу зазвичай більший через послідовну природу операцій;
4. RAM: Миттєвий доступ до даних незалежно від їх місцезнаходження в пам'яті.

Аналіз швидкості передачі даних:

1. DMA: Висока швидкість передачі, яка може використовуватися для масивних даних, таких як відео або аудіопотоки.
2. AAM: Швидкість залежить від алгоритмів кешування; може бути дуже швидкою для невеликих наборів даних;
3. SAM: Швидкість передачі даних може бути обмежена медіа, але це підходить для додатків, які не вимагають частого випадкового доступу;
4. RAM: Висока швидкість передачі, що робить його ідеальним для додатків, які вимагають швидкого доступу до великої кількості даних, як в базах даних.

Оцінка впливу на пропускну здатність системи:

1. DMA: Може збільшити пропускну здатність системи, оскільки звільняє CPU для інших завдань;
2. AAM: Вплив на систему залежить від ефективності імплементації кешу; ідеальний для систем, де швидкий пошук даних є критичним;

3. SAM: Може обмежувати загальну пропускну здатність системи, якщо дані не оптимізовані для послідовного доступу;
4. RAM: Забезпечує високу пропускну здатність завдяки миттєвому доступу до будь-якої частини пам'яті.

Порівняння енергоефективності:

1. DMA: Покращує загальну енергоефективність, мінімізуючи навантаження на CPU, що знижує загальне споживання енергії системою;
2. AAM: Може вимагати додаткових ресурсів для управління кешем, що потенційно підвищує споживання енергії, але оптимізація кешу може зменшити цей вплив;
3. SAM: Зазвичай має високу енергоефективність, оскільки механізми прості і обмежені в енергоспоживанні;
4. RAM: Енергоефективність залежить від технології пам'яті; деякі сучасні типи RAM дуже ефективні, але старіші моделі можуть бути менш ефективними.

Порівняння загальної пропускної здатності системи:

1. DMA: Значно підвищує пропускну здатність системи, оскільки CPU може виконувати інші задачі під час передачі даних [23];
2. AAM: Пропускна здатність може бути високою, особливо коли кеш-пам'ять ефективно використовується для часто використовуваних даних;
3. SAM: Пропускна здатність обмежена механічними обмеженнями та послідовністю доступу до даних;
4. RAM: Забезпечує високу пропускну здатність завдяки миттєвому доступу до будь-якої частини пам'яті.

Порівняння вартості:

1. DMA: Вимагає інвестицій у спеціалізоване обладнання, що може підвищити початкові витрати.

2. AAM: Інвестиції у розробку ефективних кешуючих алгоритмів та потужного обладнання для їх реалізації можуть бути значними.
3. SAM: Зазвичай має низьку вартість через простоту технологій та широку доступність.
4. RAM: Вартість може бути високою для передових технологій швидкого доступу, але знижується з часом у міру розвитку технологій.

Порівняння надійності:

1. DMA: Висока надійність при адекватному управлінні ресурсами та правильному технічному обслуговуванні.
2. AAM: Надійність залежить від алгоритмів кешування та якості кеш-пам'яті.
3. SAM: Надійність може варіюватися залежно від якості носіїв даних і правильності їх використання.
4. RAM: Висока надійність з сучасними технологіями захисту та відновлення даних [24].

Зробивши цей аналіз порівняльної ефективності схем доступу, можна занести результати до таблиці 2.2 для кращого розуміння та візуалізації результатів.

Таблиця 2.2 - Результати аналізу порівняльної ефективності схем доступу

Метод доступу	**Прямий доступ (DMA)**	**Асоціативний доступ (AAM)**	**Послідовний доступ (SAM)**	**Випадковий доступ (RAM)**
Час доступу	Мінімальний	Змінний, залежить від кешування	Вищий, послідовний	Миттєвий
Швидкість передачі даних	Висока для масивних даних	Варіюється; швидка для невеликих наборів	Обмежена медіа, але підходить для специфічних застосувань	Висока для швидкого доступу

Продовження таблиці 2.2

Вплив на пропускну здатність системи	Збільшує пропускну здатність, звільняючи CPU	Залежить від ефективності кешування	Може обмежувати загальну пропускну здатність	Забезпечує високу пропускну здатність за рахунок миттєвого доступу
Енергоефект.	Покращує загальну ефективність, знижуючи навантаження на CPU	Може збільшити споживання через потребу управління кешем	Висока, оскільки механізми обмежені в енергоспоживанні	Залежить від технології; сучасні типи дуже ефективні
Пропускна здатність системи	Значно підвищує, оскільки CPU може виконувати інші задачі	Висока для додатків з інтенсивним використанням пам'яті	Обмежена механічними обмеженнями та послідовністю доступу	Висока, ідеально для операцій з великою кількістю даних
Вартість	Вимагає інвестицій у спеціалізоване обладнання	Інвестиції у розробку ефективних кешуючих алгоритмів	Зазвичай низька через простоту технологій	Висока для передових технологій, але знижується з часом
Надійність	Вимагає інвестицій у спеціалізоване обладнання	Залежить від алгоритмів кешування	Залежить від якості носіїв даних	Висока з сучасними технологіями захисту даних

2.3 Аналіз впливу на продуктивність системи

Цей розділ аналізує, як різні методи доступу до пам'яті впливають на продуктивність обчислювальних систем. Аналіз зосереджується на визначенні часу відгуку системи, ефективності ресурсів, а також здатності методів доступу підтримувати високу пропускну спроможність та мінімізувати затримки.

Аналіз часу відгуку системи:

1. DMA: Забезпечує мінімальний час відгуку, оскільки периферійні пристрої можуть напряму обмінюватися даними з пам'яттю, не завантажуючи CPU;
2. AAM: Час відгуку може варіюватися, але оптимально налаштовані системи кешування значно покращують швидкість доступу до часто використовуваних даних;
3. SAM: Час відгуку збільшується через необхідність доступу до даних у фіксованій послідовності, що може уповільнювати операції;
4. RAM: Практично миттєвий час відгуку, оскільки можливий доступ до будь-якої частини пам'яті на вимогу.

Аналіз ефективності ресурсів:

1. DMA: Забезпечує високу ефективність використання ресурсів, знижуючи навантаження на процесор і звільняючи його для інших завдань;
2. AAM: Ефективність залежить від розміру і конфігурації кеш-пам'яті, що вимагає додаткових ресурсів для управління;
3. SAM: Енергійно ефективний, але менш гнучкий у використанні, що може зменшувати загальну ефективність системи в залежності від типу завдань;
4. RAM: Використовується в основному для операцій, що вимагають високої швидкості обробки, високо ефективний для різноманітних додатків.

Аналіз пропускної спроможності:

1. DMA: Підвищує загальну пропускну здатність системи, дозволяючи одночасне виконання декількох операцій з даними;
2. AAM: Оптимізація кешу може значно покращити пропускну спроможність, особливо для інтенсивних даних;

3. SAM: Має обмеження пропускної спроможності через послідовний характер доступу, що не ідеально для задач з великим обсягом даних;
4. RAM: Відмінна пропускна здатність для великої кількості даних, забезпечуючи швидкий і надійний доступ до пам'яті.

Аналіз системних затримок:

1. DMA: Зменшує затримки завдяки безпосередній передачі даних між пам'яттю та периферійними пристроями;
2. AAM: Ефективне кешування може значно знизити затримки при частих запитах;
3. SAM: Затримки можуть збільшуватись з ростом обсягу послідовно процесованих даних [25];
4. RAM: Мінімізує затримки завдяки миттєвому доступу до будь-якої частини пам'яті.

Зробивши цей аналіз впливу на продуктивність системи, можна занести результати до таблиці 2.3 для кращого розуміння та візуалізації результатів.

Таблиця 2.3 - Результати аналізу впливу на продуктивність системи схем доступу

Метод доступу	**Час відгуку системи**	**Ефективність ресурсів**	**Пропускна спроможність**	**Системні затримки**
Прямий доступ (DMA)	Мінімальна	Висока	Підвищує	Зменшує
Асоціативний доступ (AAM)	Змінна	Залежить від кешу	Може покращувати	Знижує при ефектному кешуванні
Послідовний доступ (SAM)	Збільшується	Досить висока	Обмежена	Можуть збільшуватись
Випадковий доступ (RAM)	Миттєва	Висока для різноманітних додатків	Відмінна	Мінімізує

2.4. Аналіз процесорів з фон Нейманівською архітектурою : Pentium, Pentium MMX, Pentium II, Pentium III, Pentium 4

Pentium (1993): Перший процесор Intel із суперскалярною архітектурою, що дозволив виконувати кілька інструкцій за такт. Зовнішній кеш-пам'яті L2 збільшувала продуктивність, але впливала на швидкість через зовнішнє розташування. Використовувався переважно в домашніх ПК для базових обчислень. Суперскалярний наступник 486-го процесора від Intel. Має наступні особливості: Дві 32-розрядні цілочисельні конвеєри 486-го типу з перевіркою залежностей; може виконувати максимум дві інструкції за цикл; конвеєрна система числення з плаваючою комою; 16 кілобайт кеш-пам'яті на кристалі; передбачення розгалужень; 64-розрядний інтерфейс пам'яті; 3,1 млн. транзисторів на 262-міліметровому кристалі. 4 мм2; ~2,3 мільйона транзисторів в логіці ядра; тактова частота 66 МГц; тепловиділення 16 Вт; продуктивність цілочисельних операцій 64,5 SPECint92; продуктивність операцій з плаваючою комою 56,9 SPECfp92; 8 32-бітних регістрів загального призначення; 8 80-бітних регістрів з плаваючою комою. Він називається «Pentium», тому що є п'ятим у лінійці 80x86. Він міг би називатися 80586, якби американський суд не постановив, що ви не можете зареєструвати номер як торгову марку [26].

Pentium MMX (1996): Удосконалений процесор Intel, створений на основі архітектури Pentium, отримав підтримку технології MMX (MultiMedia eXtension), яка значно підвищувала продуктивність у мультимедійних і графічних обчисленнях. Цей процесор був орієнтований на персональні комп’ютери, які використовувалися для домашніх і офісних завдань, із наголосом на обробку графіки, відео та аудіо.

Процесор зберіг дві 32-розрядні суперскалярні цілочисельні конвеєрні лінії, здатні виконувати до двох інструкцій за цикл, а також підтримував конвеєрну обробку чисел із плаваючою комою. Внутрішній кеш було збільшено до 32 кілобайт (по 16 КБ для даних і інструкцій), що підвищувало швидкодію. Завдяки технології MMX процесор отримав 57 нових інструкцій для роботи з

мультимедійними потоками, а також додаткові регістри для обробки цілочисельних операцій, що прискорило виконання графічних і звукових завдань. При цьому регістри MMX дублювали вже наявні 80-бітні регістри з плаваючою комою (FPU), що дозволяло зменшити апаратну складність [27, 28].

Pentium MMX виготовлявся за 0,35-мікронним техпроцесом і мав 4,5 мільйона транзисторів на кристалі площею 140 мм². Частота процесора досягала 233 МГц, а тепловиділення коливалося в межах 15-20 Вт залежно від моделі. Завдяки 64-розрядному інтерфейсу пам'яті та оптимізації роботи з кешем, продуктивність підвищувалася як у звичайних задачах, так і в нових мультимедійних застосунках. Цей процесор став важливим кроком в еволюції лінійки Pentium, пропонуючи користувачам не лише базову обчислювальну потужність, але й ефективність для нових технологічних потреб кінця 1990-х [29].

Pentium II (1997): Наступний процесор у лінійці Intel, побудований на вдосконаленій архітектурі P6, яка вперше з'явилася у Pentium Pro. Pentium II поєднав переваги попередника з новими можливостями, спрямованими на підвищення продуктивності та підтримку мультимедійних завдань. Процесор був орієнтований на широкий спектр користувачів – від офісних працівників до геймерів, завдяки своїй універсальності та значному приросту потужності.

Pentium II був виготовлений за 0,35-мікронним, а пізніше 0,25-мікронним техпроцесом, і мав до 7,5 мільйонів транзисторів. Нововведенням стала інтеграція кеш-пам'яті другого рівня (L2), яка працювала на половинній частоті ядра. Хоча це й уповільнювало доступ до кешу в порівнянні з внутрішнім кешем, як у Pentium Pro, таке рішення дозволило знизити витрати та збільшити ємність кешу, яка сягала 512 КБ.

Процесор підтримував технологію MMX, вперше впроваджену в Pentium MMX, що забезпечувало прискорення обробки мультимедійних даних. Архітектура P6 забезпечувала динамічне виконання інструкцій, включаючи передбачення розгалужень, позачергове виконання та динамічне повторне впорядкування інструкцій, що значно підвищувало ефективність роботи.

Процесор мав 32 КБ внутрішньої кеш-пам'яті першого рівня (по 16 КБ для даних та інструкцій). Він підтримував 64-розрядний інтерфейс пам’яті та досягав тактових частот від 233 до 450 МГц. Підтримка нових інструкцій та оптимізована архітектура дозволяли Pentium II демонструвати високу продуктивність у числових і мультимедійних задачах, а також забезпечувати кращу сумісність із сучасним програмним забезпеченням. Завдяки інноваційній конструкції у вигляді SECC (Single Edge Contact Cartridge), процесор виглядав як плата в пластиковому картриджі, що полегшувало встановлення та захищало чіп і кеш [26,30].

Pentium III (1999): Удосконалений процесор Intel, який базувався на архітектурі P6, що використовувалася у Pentium II, але був доповнений новими можливостями та розширеннями для підвищення продуктивності, зокрема у сфері мультимедіа, 3D-графіки та мережевих обчислень. Процесор був розроблений для використання у персональних комп’ютерах високого рівня, робочих станціях і серверах.

Pentium III впровадив технологію Streaming SIMD Extensions (SSE), яка додала 70 нових інструкцій для прискорення обчислень із плаваючою комою, обробки мультимедійних даних та складної 3D-графіки. SSE доповнила технологію MMX, яку процесор також підтримував. Це зробило Pentium III популярним вибором для роботи з вимогливими програмами, включаючи CAD-системи, ігри та мультимедійне програмне забезпечення.

Процесор мав 32 КБ кеш-пам'яті першого рівня (16 КБ для даних і 16 КБ для інструкцій) і кеш другого рівня (L2), розташований зовні або інтегрований на кристал у пізніших версіях. Розмір кешу L2 становив 256 КБ, і він працював на повній частоті ядра в інтегрованих варіантах. Це забезпечувало швидший доступ до даних і підвищувало загальну продуктивність.

Процесор був виготовлений за 0,25-мікронним техпроцесом, а пізніше – за 0,18-мікронним, що дозволило досягти більш високих тактових частот від 450 МГц до 1,13 ГГц. Pentium III мав до 9,5 мільйонів транзисторів на кристалі, підтримував 64-розрядний інтерфейс пам'яті та оснащувався динамічним виконанням інструкцій із передбаченням розгалужень і позачерговим виконанням.

Однією з інновацій стало впровадження унікального ідентифікатора (Processor Serial Number), який викликав суперечки через занепокоєння щодо конфіденційності, оскільки дозволяв ідентифікувати кожен процесор у мережі. Пізніше Intel дозволила користувачам вимикати цю функцію.

Pentium III випускався у форм-факторах SECC2 (картридж) і FC-PGA (пакет із роз'ємом), що забезпечувало широку сумісність із різними системами. Завдяки високій продуктивності та інноваціям, Pentium III став основним вибором для комп'ютерів кінця 1990-х і початку 2000-х, зокрема для вимогливих задач у бізнесі, освіті та домашньому використанні [31, 32].

Pentium 4 (2000): Процесор Intel, що став великим кроком вперед завдяки впровадженню нової архітектури NetBurst, яка була спеціально розроблена для досягнення високих тактових частот і продуктивності. Pentium 4 орієнтувався на домашніх і корпоративних користувачів, пропонуючи значний приріст потужності для роботи з мультимедіа, графікою та мережевими обчисленнями.

Архітектура NetBurst запровадила гіпертривалий 20-ступеневий конвеєр, що дозволило підвищити частоти до рекордних на той час значень. Однак це спричиняло деяке зниження ефективності на інструкцію через тривалий цикл виконання команд. Процесор отримав підтримку технології Streaming SIMD Extensions 2 (SSE2), яка додала 144 нові інструкції для прискорення обчислень із плаваючою комою, 3D-графіки та мультимедійних задач.

Pentium 4 мав 256 КБ кеш-пам'яті другого рівня (L2) у ранніх версіях, яка була інтегрована на кристал і працювала на повній частоті ядра. Пізніші моделі збільшили обсяг кешу до 512 КБ і більше, що значно підвищувало продуктивність у ресурсоємних завданнях. Інтерфейс пам'яті був 64-розрядним, із підтримкою сучасних технологій, таких як Rambus DRAM (RDRAM) для підвищеної пропускної здатності.

Процесор виготовлявся за 0,18-мікронним техпроцесом, містив 42 мільйони транзисторів і підтримував частоти від 1,3 ГГц до 2 ГГц у початкових моделях. Завдяки підвищенню тактових частот, Pentium 4 встановив нові стандарти для

обчислювальної потужності, хоча високе тепловиділення і споживання енергії залишалися викликом.

Інновацією стала шина Quad-Pumped Bus, яка досягала швидкості передачі даних до 3,2 ГБ/с, забезпечуючи швидкий обмін інформацією між процесором і пам'яттю. Також впроваджено технології передбачення розгалужень, позачергового виконання інструкцій і динамічного управління живленням, що оптимізувало енергоефективність у багатьох сценаріях.

Pentium 4 випускався у форм-факторах PGA423 і пізніше PGA478, а також у версіях для серверів і робочих станцій. Цей процесор був популярним для роботи з вимогливим програмним забезпеченням, мультимедіа та іграми, ставши основним вибором у багатьох ПК початку 2000-х [33, 34].

На основі цього короткого опису про дані процесори, можна підвести короткий висновок, про різні покоління процесорів на основі структури фон Неймана.

Таблиця 2.4 - Основні різниці в різних поколіннях процесорів Pentium

Процесор	**Pentium (1993)**	**Pentium MMX (1996)**	**Pentium II (1997)**	**Pentium III (1999)**	**Pentium 4 (2000)**
Техпроцес	0,8 мкм	0,35 мкм	0,35 мкм	0,25 мкм	0,18 мкм
Тактова частота	60-300 МГц	166-233 МГц	233-450 МГц	450-1400 МГц	1,3-3,8 ГГц
Кеш L1	16 КБ (8+8)	32 КБ (16+16)	32 КБ (16+16)	32 КБ (16+16)	8 КБ + 12К (мікроопер.)

Продовження таблиці 2.4

Кеш L2	Відсутній (зовнішній)	Зовнішній (на мат. платі)	512 КБ (на картриджі, ½ швидк.)	256-512 КБ (інтегр., повна швид.)	256 КБ-2 МБ (інтегр., повна швид.)
Особливості	Суперскалярна архітектура	MMX-інструкції для мультимедіа	Картридж SECC (Slot 1)	SSE-інструкції, інтеграція L2	NetBurst, Hyper-Threading

До ключових змін в кеш-пам’яті можна віднести наступні аспекти:

- Pentium MMX збільшив L1 до 32 КБ, що було першим значним апгрейдом кешу;
- Pentium II вперше отримав L2-кеш у 512 КБ, хоч і повільніший за ядро;
- Pentium III інтегрував L2-кеш у ядро, зробивши його швидшим;
- Pentium 4 мав найшвидший і найбільший кеш L2, а також новий формат L1-кешу.

РОЗДІЛ 3. РОЗРОБКА СХЕМИ ПЕРЕВІРКИ ДОСТУПУ ДО ПАМ'ЯТІ, ЩО ПРАЦЮЄ З ІНФОРМАЦІЄЮ ПРО ХИБНІ ДАНІ В ЧАСТИНАХ КЕШ

3.1. Прототип схеми перевірки доступу до пам'яті, що працює з інформацією про хибні дані в частинах кеш

За основу схеми перевірки доступу до пам'яті, що працює з інформацією про хибні дані в частинах кеш буде взято схему з роботи Dongkyun, A., Gyungho, L., яка має назву A Memory Access Validation Scheme against Payload Injection Attacks (рис.3.1) [35].

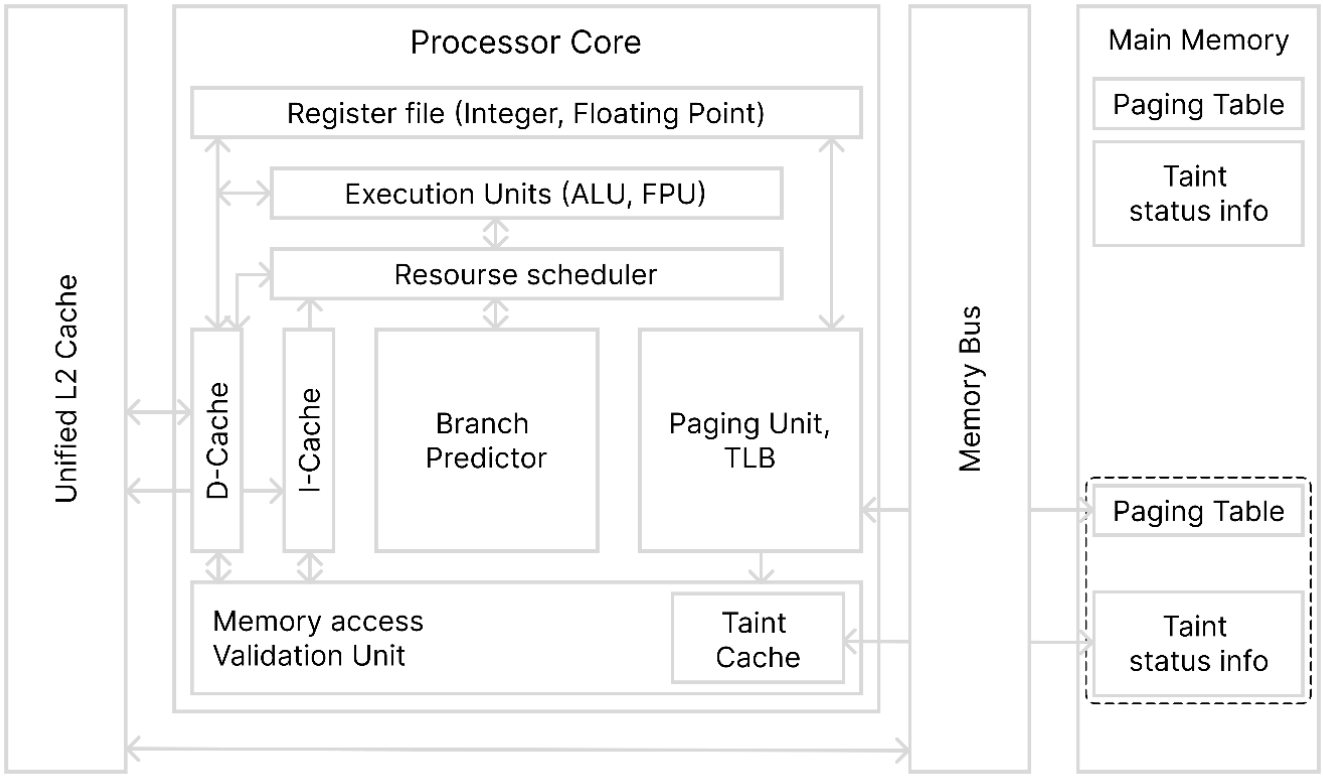

Рис. 3.1 – Прототип схеми перевірки доступу до пам'яті, що працює з інформацією про хибні дані в частинах кеш

Дана схема складається з наступних елементів:

- Register file (Файл регістрів) – це надзвичайно швидка пам'ять всередині центрального процесора, де зберігаються дані, з якими процесор працює в даний момент. Існує кілька типів регістрів: для цілих чисел

(використовуються для простих обчислень) та для чисел з плаваючою крапкою (для складніших математичних операцій). Регістровий файл є одним з ключових компонентів процесора за рахунок швидкості своєї роботи (доступ до даних у регістрах набагато швидший, ніж до основної пам’яті), а також того, що дає можливість процесору використувувати різні типи регістрів. Проте, через обмежене місце на чіпі процесора, кількість регістрів обмежена;

- Execution Units (Виконавчі пристрої) – це частини процесора, які безпосередньо виконують арифметичні та логічні операції, тобто обчислення. Основні типи виконавчих пристроїв: ALU (арифметико-логічний пристрій): виконує базові математичні операції (додавання, віднімання тощо) та логічні операції (І, АБО, НЕ) з цілими числами. Це основа для більшості обчислень. FPU (пристрій для операцій з плаваючою крапкою): спеціалізується на роботі з дробовими числами та складнішими математичними операціями, такими як тригонометричні функції, логарифми тощо. Використовується для наукових обчислень, графіки та інших задач, що вимагають високої точності;
- Resource scheduler (Планувальник ресурсів) – це система, яка розподіляє різні ресурси (процесорний час, пам'ять, дисковий простір тощо) між різними програмами, які працюють одночасно. Основними його завданнями є планування ресурсів, управління процесами та пріоритизація. Тобто, планувальник, в залежності від рівня важливості, вирішує якому процесу дати більше ресурсів та які процеси повинні працювати в даний момент, а які будуть тимчасово призупинені. Використовуються різні стратегії, такі як Round Robin, Priority Scheduling, Multilevel Queue Scheduling. Ефективний планувальник забезпечує високу продуктивність, стабільність, справедливий розподіл;
- D-Cache, I-Cache - D-кеш — це надшвидка пам'ять всередині процесора, призначена для тимчасового зберігання даних, до яких процесор часто звертається. І-кеш – це надшвидка пам'ять всередині процесора, призначена

для тимчасового зберігання інструкцій (команд), які процесор виконує. Роль L1 Cache у багаторівневій системі кешування: L1 Cache має дуже малий розмір (наприклад, 32-128 КБ для кожного з типів I-Cache та D-Cache), але дуже високу швидкість; він завжди розташовується безпосередньо на самому процесорі (CPU) або максимально близько до нього; завдяки такому розташуванню знижується час доступу до даних та інструкцій, що дозволяє мінімізувати простої процесора під час очікування потрібної інформації;

- Unified L2 Cache (Об’єднаний кеш L2) – це тип кеш-пам’яті другого рівня, який використовується в сучасних багатоядерних процесорах для підвищення швидкодії. На відміну від традиційних процесорів, де кожне ядро має свій окремий кеш L2, в системах з об’єднаним кешем він є спільним для всіх ядер. Це дозволяє ядрам ефективніше використовувати пам’ять, зменшує затримки доступу до даних і покращує загальну продуктивність системи. Такий підхід особливо корисний в багатозадачних системах, де кілька ядер працюють одночасно. Крім того, він допомагає зменшити енергоспоживання системи. Ключові переваги: економія пам’яті, збільшення швидкодії, краще управління даними;
- Branch Predictor (Передбачувач розгалуджень) – це спеціальний модуль у процесорі, який намагається заздалегідь визначити, яким шляхом виконання програма піде після зустрічі умовного переходу (наприклад, "якщо", "цикл"). Це дозволяє процесору не зупинятися і продовжувати роботу, навіть не дочекавшись результату перевірки умови. Він аналізує попередню історію виконання програми і на її основі робить прогноз щодо результату наступного переходу. Існує кілька методів передбачення, від простих (наприклад, завжди передбачати один і той же результат) до складних (використання таблиць історії та кореляцій). Ефективний Branch Predictor дозволяє значно підвищити продуктивність процесора, особливо при виконанні програм з великою кількістю умовних переходів;
- Paging Unit (Блок сторінкової організації пам'яті) та TLB (Translation Lookaside Buffer) – це важливі компоненти, які дозволяють комп'ютеру

ефективно використовувати пам'ять. Працюють наступним чином: програми працюють з віртуальною пам'яттю, яка логічно розділена на невеликі блоки – сторінки. Реальна фізична пам'ять також розділена на блоки – фрейми. Paging Unit перетворює віртуальні адреси, які використовує програма, на фізичні адреси, за якими дані зберігаються в пам'яті. TLB це кеш, який зберігає нещодавно використані перетворення віртуальних адрес у фізичні, щоб прискорити цей процес. Це дозволяє ефективніше використовувати пам’ять, оскільки сторінки віртуальної пам'яті можуть бути більшими за розмір фізичної пам'яті;

- Memory Access Validation Unit (Одиниця перевірки доступу до пам'яті) – модуль, який стежить за тим, щоб кожна програма в комп'ютері мала доступ тільки до "своєї" частини пам'яті. Вона перевіряє, чи має програма право читати, писати або виконувати код в конкретній ділянці пам'яті. Це забезпечує захист даних (запобігає випадковому або навмисному пошкодженню даних однієї програми іншою), стабільність системи (перешкоджає програмам "залазити" в чужі області пам'яті, що може призвести до збоїв і зависань системи), безпеку (захищає систему від зловмисних програм, які можуть намагатися отримати доступ до чужих даних або виконувати власний код);
- Taint Cache – це технологія, яка використовується для відстеження та контролю "заражених" даних у процесорі. "Заражені" дані – це дані, які можуть бути потенційно небезпечними, наприклад, отримані ззовні (від користувача, з мережі). Є важливим компонентом, оскільки забезпечує безпека та продуктивність. Запобігає використанню неперевірених даних у критичних частинах програм, що може призвести до вразливостей. Швидко виявляє потенційно небезпечні дані, не сповільнюючи роботу системи. Якщо дані "заражені", система може обмежити їх використання або повністю заборонити;
- Memory Bus (Шина пам’яті) – один з ключових компонентів системи, по якому дані передаються між процесором та оперативною пам’яттю. Чим

швидше дані передаються між процесором і пам'яттю, тим швидше працює комп'ютер. Швидка шина дозволяє обробляти великі обсяги даних. Існують різні типи шин пам'яті, кожна з яких має свої особливості і використовується в різних типах комп'ютерів;

- Paging table in main memory – це спеціальна структура даних, що зберігається в оперативній пам'яті комп'ютера. Вона використовується для перетворення віртуальних адрес (тих, які бачить програма) на фізичні адреси (ті, за якими дані знаходяться насправді в пам'яті). Це дозволяє операційній системі ефективніше керувати пам'яттю, ізолювати процеси один від одного та забезпечувати віртуальну пам'ять. Коли програма звертається за адресою, процесор спочатку шукає відповідність у TLB. Якщо не знаходить, то звертається до таблиці сторінок. Знайшовши відповідний запис, процесор отримує фізичну адресу і звертається до пам'яті;
- Taint status info in main memory – це спеціальна мітка, яка вказує на те, чи є дані потенційно небезпечними. Ця мітка зберігається в оперативній пам'яті разом з самими даними. Вона дозволяє відстежувати шлях даних від їхнього джерела (наприклад, користувацький ввід) і запобігати зловмисним атакам, таким як SQL-ін'єкції. Дані, отримані з неперевірених джерел, автоматично позначаються як "заражені". Якщо такі дані використовуються для створення нових даних, нова інформація також отримує мітку "заражена". Перед виконанням критичних операцій система перевіряє мітку даних. Якщо дані "заражені", операція може бути заблокована або виконана з додатковими перевірками.

3.2. Симулятивний код для схема перевірки доступу до пам’яті, що працює з інформацією про хибні дані в частинах кеш

Код, який буде симулювати роботу схеми буде створено за допомогою програми Matlab – платформою для програмування та числових обчислень [36].

Весь програмний код можна переглянути в ДОДАТОК Б та код для побудови графіків до нього, які демонструють наявну роботу програми можна буде переглянути в ДОДАТОК В. Сам код складається з наступним важливих блоків: ініціалізація системних компонентів, лічильники продуктивності кешу, позначення однієї адреси як зараженої (це зроблено для того, щоб показати, як схема буде працювати, коли в неї попадають заражені дані; а так як ввести зараз заражені дані в неї неможливо, то заразимо одну адресу), попередньо задаємо значення для кешу та пам'яті (рис.3.2), введення операндів для ALU/FPU (користувач сам може ввести значення з якими хоче працювати), вибір операції для введених операндів (також користувачеві дається змога самотужки обрати яку операцію він хоче виконати над своїми операндами), введення віртуальної адреси (користувач сам обирає в яку віртуальну адресу), лічильник часу (для того, щоб можна було відслідкувати швидкодію роботи) (рис.3.3), виконання ALU/FPU операцій та зберігання результату в перший регістр (рис.3.4), перевірка доступу до пам'яті (рис.3.5), перевірка кешу (для якої також працює таймер) (рис.3.6), перевірка на Taint Cache і виведення результатів (рис.3.7). Також було зроблено виведення певних результатів на графіки, а саме графік продуктивності кешу, стан Taint Cache та графік часу виконання (рис.3.8).

```
%% CPU Simulation with Memory Access Validation

% Ініціалізація системних компонентів
register_file = zeros(1, 8); % Реєстровий файл
L1_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L1 кеш
L2_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L2 кеш
paging_table = containers.Map({30, 40, 50, 70}, {100, 200, 300, 400}); % Таблиця сторінок
main_memory = containers.Map({100, 200, 300}, {256, 512, 1024}); % Основна пам'ять
taint_cache = containers.Map('KeyType', 'double', 'ValueType', 'logical'); % Taint Cache

% Лічильники продуктивності кешу
cache_hits_L1 = 0; % Хіти L1
cache_hits_L2 = 0; % Хіти L2
memory_accesses = 0; % Доступи до пам'яті

% Попередньо додані значення для кешу і пам'яті
L1_cache(50) = 128; % Додано значення в L1 Cache
L2_cache(40) = 256; % Додано значення в L2 Cache
main_memory(300) = 512; % Додано значення в основну пам'ять
paging_table(60) = 300; % Віртуальна адреса 60 мапиться на фізичну адресу 300

% Позначаємо одну адресу як "tainted"
tainted_address = 40; % Вибрана адреса для позначення
taint_cache(tainted_address) = true; % Позначаємо адресу 40 як "tainted"
```

Рис. 3.2 – Блоки коду, що відповідають за ініціалізація системних компонентів, лічильники продуктивності кешу, попередньо додані значення для кешу і пам'яті та позначення однієї адреси як "tainted" у прототипі

```
%% Введення операндів для ALU/FPU
operand1 = input('Enter the first operand: '); % Користувач вводить перший операнд
operand2 = input('Enter the second operand: '); % Користувач вводить другий операнд

% Вибір операції для ALU/FPU
disp('Select operation:');
disp('1 - Addition');
disp('2 - Subtraction');
disp('3 - Multiplication');
disp('4 - Division');
operation_code = input('Enter the operation code (1-4): ');

% Введення віртуальної адреси
virtual_address = input('Enter the virtual memory address: ');
```

Рис. 3.3 – Блоки коду, що відповідають за введення операндів, вибір операції, введення віртуальної адреси у прототипі

```
%% Виконання ALU операцій
tic; % Початок загального таймера
alu_start_time = tic; % Початок таймера ALU
switch operation_code
    case 1 % Додавання
        alu_result = operand1 + operand2;
        operation_name = 'Addition';
    case 2 % Віднімання
        alu_result = operand1 - operand2;
        operation_name = 'Subtraction';
    case 3 % Множення
        alu_result = operand1 * operand2;
        operation_name = 'Multiplication';
    case 4 % Ділення
        if operand2 ~= 0
            alu_result = operand1 / operand2;
            operation_name = 'Division';
        else
            alu_result = NaN; % Якщо ділення на нуль
            operation_name = 'Error: Division by Zero';
        end
    otherwise
        alu_result = NaN; % Якщо код операції невідомий
        operation_name = 'Unknown Operation';
end
timing_ALU = toc(alu_start_time); % Час виконання ALU

% Зберігаємо результат у перший регістр
register_file(1) = alu_result;
```

Рис. 3.4 – Блоки коду, що відповідають за виконання операцій та збереження результату у прототипі

```
%% Перевірка доступу до пам'яті (Memory Access Validation Unit)
validation_start_time = tic;
if isKey(paging_table, virtual_address)
    physical_address = paging_table(virtual_address); % Отримуємо фізичну адресу
    if isKey(main_memory, physical_address)
        validation_status = 'Memory Access Valid';
    else
        validation_status = 'Memory Access Invalid (Address Not Found in Main Memory)';
    end
else
    validation_status = 'Memory Access Invalid (Address Not Found in Paging Table)';
end
timing_memory_validation = toc(validation_start_time);
```

Рис. 3.5 – Блок коду, що відповідає за перевірку доступу до пам’яті у прототипі

```
%% Перевірка кешу
cache_start_time = tic;
if isKey(L1_cache, virtual_address)
    data = L1_cache(virtual_address); % Дані знайдено в L1 кеші
    cache_status = 'L1 Cache Hit';
    cache_hits_L1 = cache_hits_L1 + 1; % Оновлюємо хіти L1
elseif isKey(L2_cache, virtual_address)
    data = L2_cache(virtual_address); % Дані знайдено в L2 кеші
    cache_status = 'L2 Cache Hit';
    cache_hits_L2 = cache_hits_L2 + 1; % Оновлюємо хіти L2
    L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
elseif isKey(paging_table, virtual_address)
    if strcmp(validation_status, 'Memory Access Valid')
        data = main_memory(physical_address); % Дані знайдено в пам'яті
        cache_status = 'Memory Access';
        memory_accesses = memory_accesses + 1; % Оновлюємо доступи до пам'яті
        L2_cache(virtual_address) = data; % Завантажуємо в L2 кеш
        L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
    else
        data = NaN; % Адреса не знайдена
        cache_status = 'Memory Access Invalid';
    end
else
    data = NaN; % Віртуальна адреса не знайдена
    cache_status = 'Address Not Found in Paging Table';
end
timing_cache = toc(cache_start_time); % Час доступу до кешу

% Завершення загального часу
timing_total = toc; % Загальний час
```

Рис. 3.6 – Блоки коду, що відповідають за перевірку кешу і збереження загального часу у прототипі

```
%% Перевірка Taint Cache
if isKey(taint_cache, virtual_address) && taint_cache(virtual_address)
    taint_status = 'Data is Tainted';
else
    taint_status = 'Data is Clean';
    taint_cache(virtual_address) = false; % Позначаємо адресу як "clean"
end
```

```
%% Вивід результатів
disp('=== CPU Simulation Results ===');
disp(['Operation: ', operation_name]);
disp(['ALU Result: ', num2str(alu_result)]);
disp(['Validation Status: ', validation_status]);
disp(['Cache Status: ', cache_status]);
disp(['Taint Cache Status: ', taint_status]);
disp(['ALU Execution Time: ', num2str(timing_ALU), ' seconds']);
disp(['Memory Validation Time: ', num2str(timing_memory_validation), ' seconds']);
disp(['Cache Access Time: ', num2str(timing_cache), ' seconds']);
disp(['Total Simulation Time: ', num2str(timing_total), ' seconds']);
```

Рис. 3.7 – Блоки коду, що відповідають на заражені дані та вивід даних у прототипі

```
%% Побудова графіків

% 1. Продуктивність кешу
figure;
cache_data = [cache_hits_L1, cache_hits_L2, memory_accesses];
bar(cache_data);
set(gca, 'XTickLabel', {'L1 Hits', 'L2 Hits', 'Memory Accesses'});
title('Cache Performance');
xlabel('Cache Categories');
ylabel('Counts');
grid on;

% 2. Графік часу виконання
figure;
timing_data = [timing_ALU, timing_memory_validation, timing_cache, timing_total];
bar(timing_data);
set(gca, 'XTickLabel', {'ALU Time', 'Validation Time', 'Cache Time', 'Total Time'});
title('Execution Times');
xlabel('Execution Stages');
ylabel('Time (seconds)');
grid on;

% 3. Стан Taint Cache
figure;
taint_addresses = keys(taint_cache);
taint_states = cell2mat(values(taint_cache));
bar(cell2mat(taint_addresses), double(taint_states), 'r');
title('Taint Cache Status');
xlabel('Memory Addresses');
ylabel('Taint State (1 = Tainted, 0 = Clean)');
grid on;
```

Рис. 3.8 – Вивід графіка продуктивності кешу, часу виконання завдань та стан зараженості

Вводимо дані для першого контр прикладу (рис.3.9), та отримуємо результат (рис.3.10) та також три графіки, які демонструють отримані результати (рис.3.11).

```
Enter the first operand: 10
Enter the second operand: 5
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 1
Enter the virtual memory address: 100
```

Рис. 3.9 – Вхідні дані першого контр. прикладу

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 15
Validation Status: Memory Access Invalid (Address Not Found in Paging Table)
Cache Status: Address Not Found in Paging Table
Taint Cache Status: Data is Clean
ALU Execution Time: 0.0001406 seconds
Memory Validation Time: 0.0005939 seconds
Cache Access Time: 0.0005324 seconds
Total Simulation Time: 0.0015517 seconds
```

Рис.3.10 – Результати перших вхідних даних

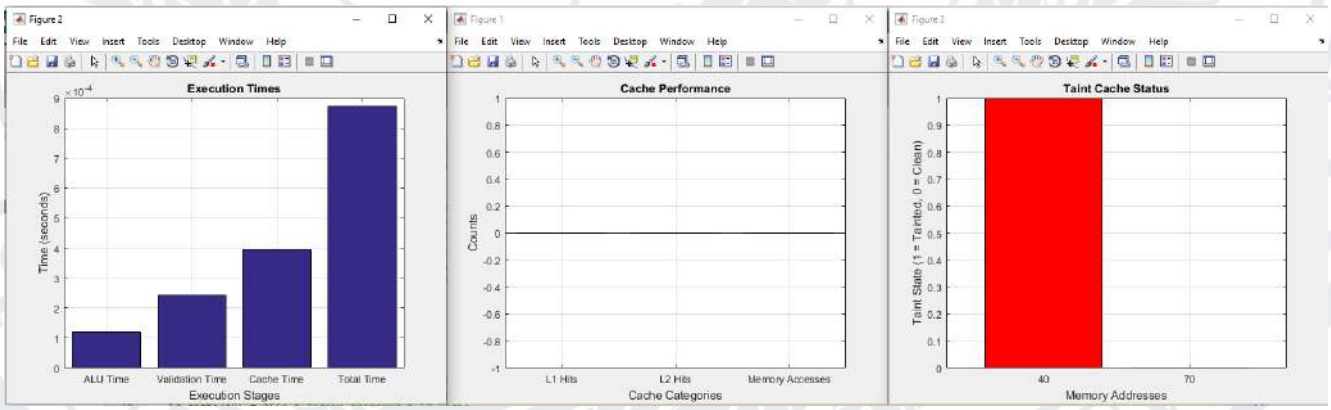

Рис.3.11 – Результуючі графіки до першого вхідного прикладу

Цей сценарій перевіряє, як система обробляє запит до віртуальної адреси, яка не мапиться на жодну фізичну адресу. Така ситуація може виникнути через помилку у програмному забезпеченні або неправильно налаштовану таблицю сторінок.

Вводимо дані для другого контр/ прикладу (рис.3.12), та отримуємо результат (рис.3.13) та також три графіки, які демонструють отримані результати (рис.3.14).

```
Enter the first operand: 12
Enter the second operand: 6
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 1
Enter the virtual memory address: 70
```

Рис. 3.12 – Вхідні дані другого контр. приклад

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 18
Validation Status: Memory Access Invalid (Address Not Found in Main Memory)
Cache Status: Memory Access Invalid
Taint Cache Status: Data is Clean
ALU Execution Time: 0.000119 seconds
Memory Validation Time: 0.0002417 seconds
Cache Access Time: 0.0003939 seconds
Total Simulation Time: 0.0008728 seconds
```

Рис. 3.13 – Результати других вхідних даних

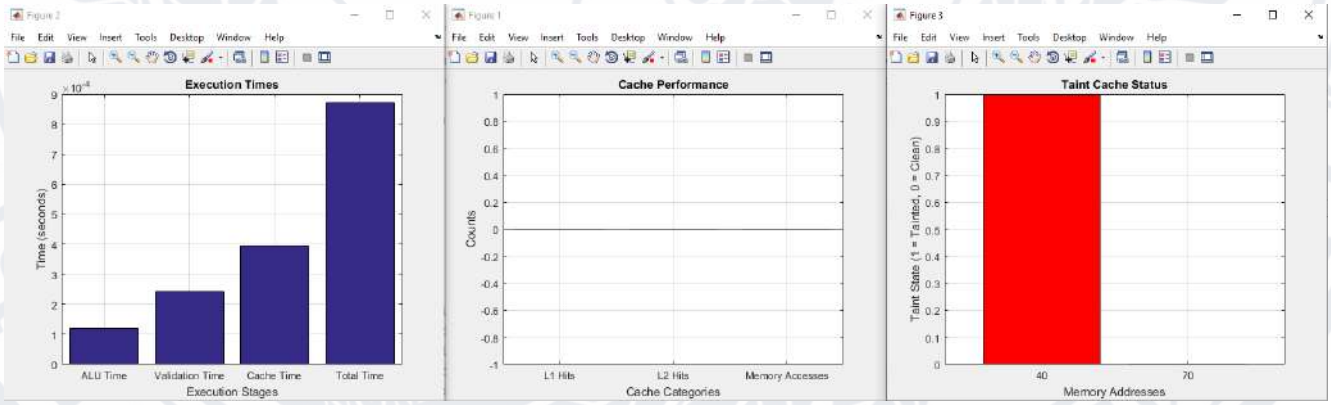

Рис.3.14 – Результуючі графіки до другого вхідного прикладу

Віртуальна адреса існує в paging_table, але фізична адреса, на яку вона мапиться, не має відповідного значення в main_memory. Наприклад, віртуальна адреса 70 мапиться на фізичну адресу 400, яка відсутня в main_memory. Це імітує помилку завантаження даних у пам’ять.

Вводимо дані для третього контр. прикладу (рис.3.15), та отримуємо результат (рис.3.16) та також три графіки, які демонструють отримані результати (рис.3.17).

```
Enter the first operand: 20
Enter the second operand: 10
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 3
Enter the virtual memory address: 40
```

Рис. 3.15 – Вхідні дані третього контр. приклад

```
=== CPU Simulation Results ===
Operation: Multiplication
ALU Result: 200
Validation Status: Memory Access Valid
Cache Status: L2 Cache Hit
Taint Cache Status: Data is Tainted
ALU Execution Time: 0.001371 seconds
Memory Validation Time: 0.0016767 seconds
Cache Access Time: 0.0016848 seconds
Total Simulation Time: 0.0060291 seconds
```

Рис. 3.16 – Результати третіх вхідних даних

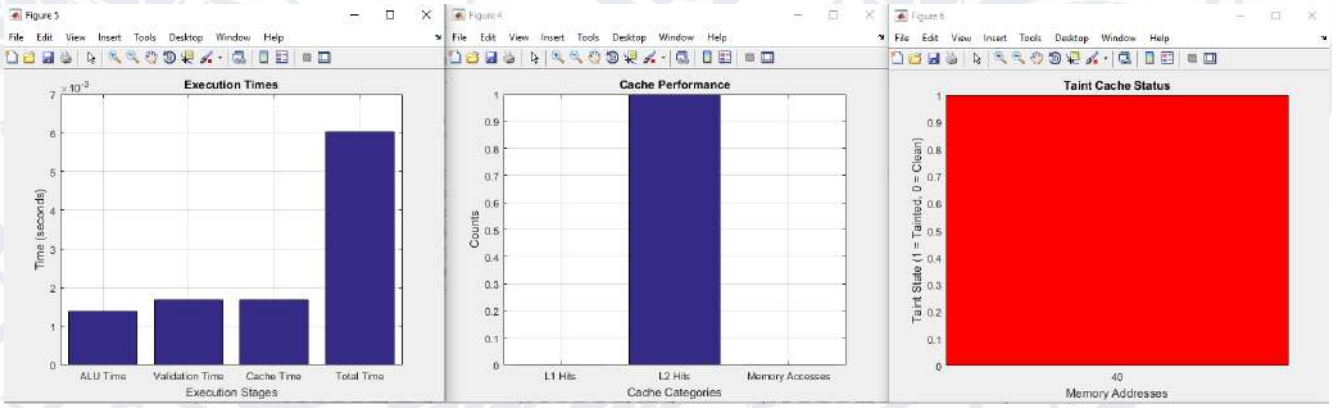

Рис.3.17 – Результуючі графіки до третього вхідного прикладу

Цей контр-приклад перевіряє роботу taint_cache. Якщо віртуальна адреса позначена як "tainted" (позначає небезпечні або скомпрометовані дані), система повинна це розпізнати і відобразити відповідний статус.

Вводимо дані для четвертого контр. Прикладу (рис.3.18), та отримуємо результат (рис.3.19) та також три графіки, які демонструють отримані результати (рис.3.20).

```
Enter the first operand: 25
Enter the second operand: 5
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 4
Enter the virtual memory address: 50
```

Рис. 3.18 – Вхідні дані четвертого контр. приклад

```
=== CPU Simulation Results ===
Operation: Division
ALU Result: 5
Validation Status: Memory Access Valid
Cache Status: L1 Cache Hit
Taint Cache Status: Data is Clean
ALU Execution Time: 0.000621 seconds
Memory Validation Time: 0.0001159 seconds
Cache Access Time: 0.0003626 seconds
Total Simulation Time: 0.0011332 seconds
```

Рис. 3.19 – Результати четвертих вхідних даних

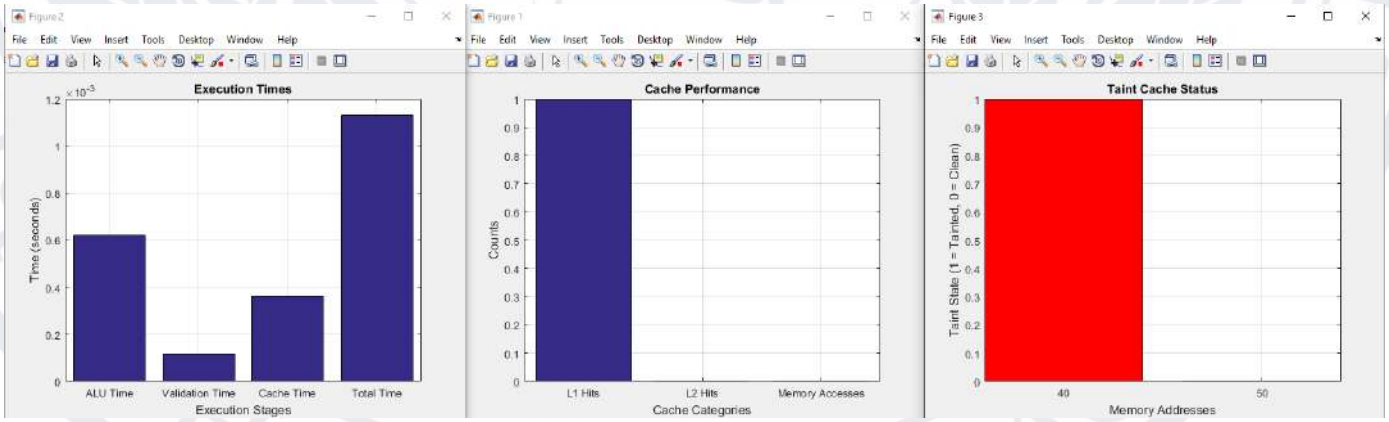

Рис.3.20 – Результуючі графіки до четвертих вхідного прикладу

Цей контр-приклад демонструє найшвидший сценарій доступу до даних, коли вони вже є в L1-cache. Це важливо для перевірки ефективності роботи кешу в сприятливих умовах.

Вводимо дані для п’ятого контр. прикладу (рис.3.21), та отримуємо результат (рис.3.22) та також три графіки, які демонструють отримані результати (рис.3.23).

```
Enter the first operand: 18
Enter the second operand: 6
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 1
Enter the virtual memory address: 40
```

Рис. 3.21 – Вхідні дані п’ятого контр. приклад

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 24
Validation Status: Memory Access Valid
Cache Status: L2 Cache Hit
Taint Cache Status: Data is Tainted
ALU Execution Time: 1.28e-05 seconds
Memory Validation Time: 0.0001454 seconds
Cache Access Time: 0.0001047 seconds
Total Simulation Time: 0.0002928 seconds
```

Рис. 3.22 – Результати п’ятих вхідних даних

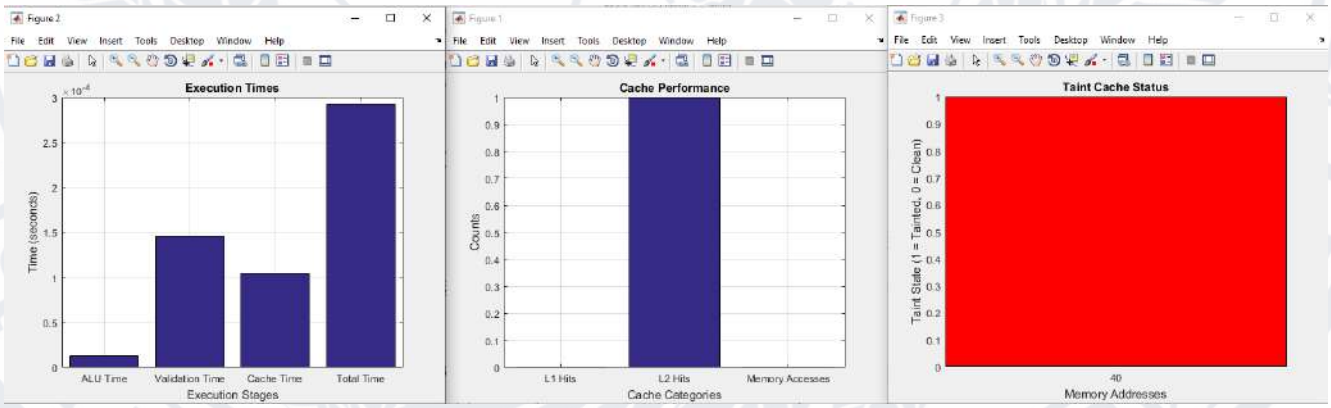

Рис.3.23 – Результуючі графіки до п’ятого вхідного прикладу

Цей контр-приклад демонструє сценарій, коли дані відсутні в L1-cache, але знайдені в L2-cache.

Вводимо дані для шостого контр. прикладу (рис.3.24), та отримуємо результат (рис.3.25) та також три графіки, які демонструють отримані результати (рис.3.26).

```
Enter the first operand: 15
Enter the second operand: 3
Select operation:
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
Enter the operation code (1-4): 3
Enter the virtual memory address: 60
```

Рис. 3.21 – Вхідні дані шостого контр. приклад

```
=== CPU Simulation Results ===
Operation: Multiplication
ALU Result: 45
Validation Status: Memory Access Valid
Cache Status: Memory Access
Taint Cache Status: Data is Clean
ALU Execution Time: 1.82e-05 seconds
Memory Validation Time: 0.0001204 seconds
Cache Access Time: 0.0009831 seconds
Total Simulation Time: 0.0011418 seconds
```

Рис. 3.22 – Результати шостих вхідних даних

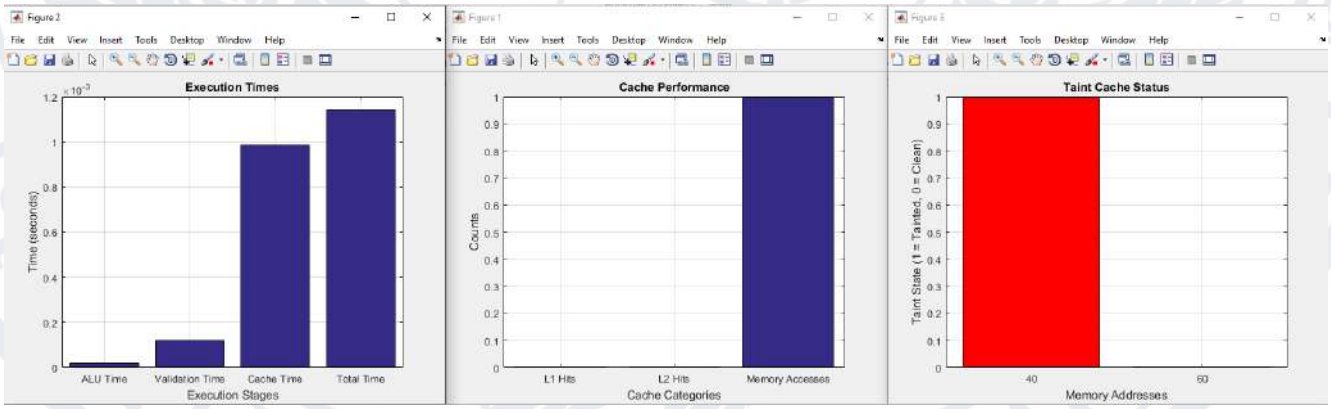

Рис.3.23 – Результуючі графіки до шостого вхідного прикладу

Цей контр-приклад демонструє сценарій, коли дані не знайдені в кешах (L1-cache, L2-cache), але доступні в основній пам'яті (main_memory). Система повинна завантажити ці дані в кеші.

3.3. Вдосконалена схема перевірки доступу до пам’яті, що працює з інформацією про хибні дані в частинах кеш

Щоб вдосконалити прототип схеми перевірки доступу до пам’яті, що працює з інформацією про хибні дані в частинах кеш, буде додано до неї наступні елементи.

Prefetcher (модуль попередньої вибірки даних) є важливим компонентом для підвищення продуктивності процесора. Його мета — передбачити та завантажити наперед дані або інструкції, які можуть знадобитися процесору, зменшуючи затримки, пов'язані з доступом до пам'яті [37].

Він працює у двох напрямках:

- Data Prefetcher: Попередньо завантажує дані у кеш-пам'ять, передбачаючи, що ці дані знадобляться процесору для обчислень;
- Instruction Prefetcher: Завантажує інструкції наперед у кеш інструкцій, щоб зменшити затримки під час виконання програмного коду.

Розміщення та зв’язки у схемі:

- Зв'язок з L1 (D-Cache та I-Cache): Data Prefetcher тісно працює з D-Cache, оскільки його завдання — заздалегідь завантажувати дані, необхідні для виконання майбутніх обчислень. Instruction Prefetcher інтегрується з I-Cache для попереднього завантаження інструкцій, необхідних для виконання програм, щоб уникнути затримок під час обробки коду;
- Розміщення між L1 та L2 Cache: Префетчери повинні бути розташовані з L2 Cache та L1 Cache, оскільки це дозволяє ефективно керувати потоком даних між рівнями кешу. Така конфігурація допоможе збалансувати завантаження кешу і уникнути простоїв при очікуванні даних або інструкцій з пам'яті;
- Зв'язок з підсистемою пам'яті: Префетчер може також безпосередньо передбачати запити до пам'яті та заздалегідь завантажувати необхідні блоки в кеш L1 або L2, зменшуючи затримки під час доступу до даних.

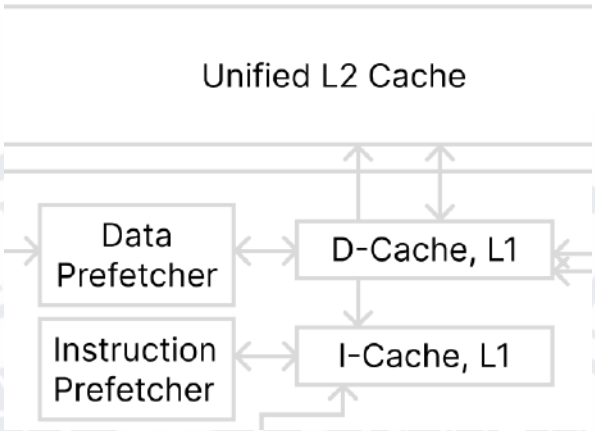

Рис. 3.24 – Розташування Data та Instruction Prefetcher

3.4. Фінальна схема доступу до пам’яті

На основі аргументів щодо необхідності та розміщення і попередніх зазначений підпунктах, в даному підпункті буде наведена покращена схема доступу до пам’яті що працює з хибними даними в частинах кеш.

Дана схема не тільки забезпечує швидкодію та обмін між різними частинами схеми, але й враховує елементи покращення продуктивності та зниження затримок.

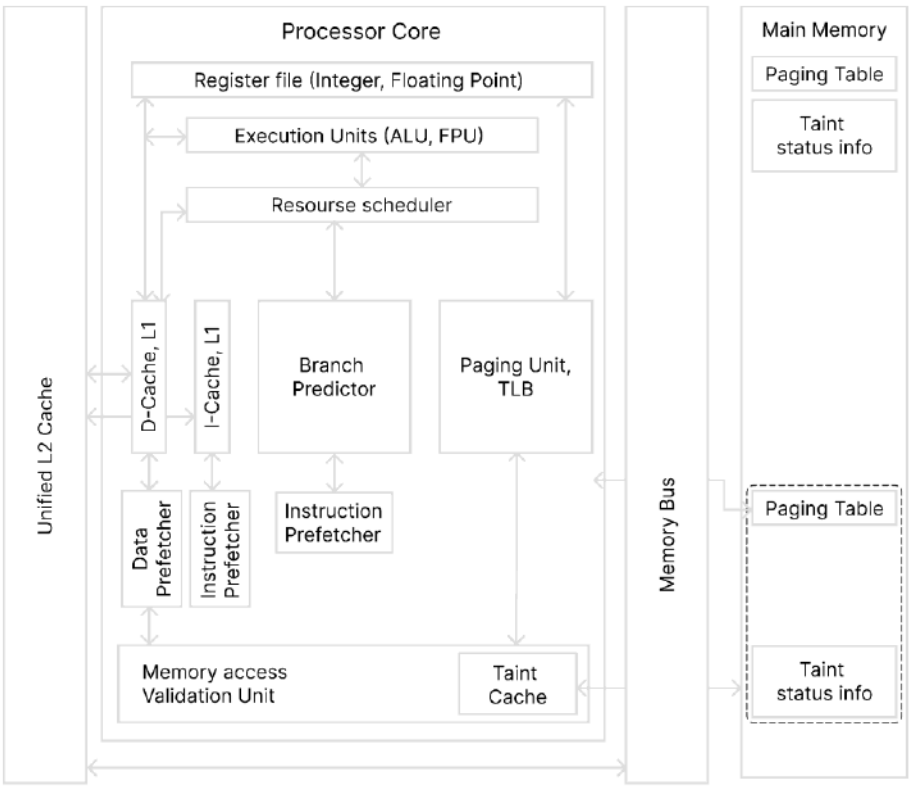

Рис. 3.26 - Схема доступу до пам’яті що працює з хибними даними в частинах кеш

3.5. Симуляція схеми перевірки доступу до пам’яті, що працює з інформацією про хибні дані в частинах кеш

Код, який буде наведено в даному підпункті теж буде написано в Matlab [36]. За своєю структурою він буде аналогічним, але до нього буде додано декілька нових блоків, які будуть відповідати за роботу, Data prefetcher та Instruction Prefetcher (рис.3.27) . В ДОДАТОК Д наведено повну версію цього коду.

```
%% Instruction Prefetching
prefetch_start_time = tic;
disp('Instruction Prefetching: Loading next instruction...');
% Емуляція роботи Instruction Prefetcher
pause(0.001); % Затримка для імітації
timing_instruction_prefetch = toc(prefetch_start_time);
```

```
%% Data Prefetching
data_prefetch_start_time = tic;
disp('Data Prefetching: Loading data to L1 Cache...');
if isKey(L2_cache, virtual_address)
    L1_cache(virtual_address) = L2_cache(virtual_address); % Завантаження даних у L1 Cache
end
timing_data_prefetch = toc(data_prefetch_start_time);
```

Рис. 3.27 – Блоки коду, що відповідають за ініціалізація системних компонентів, лічильники продуктивності кешу, позначення однієї адреси як "tainted" та попередньо додані значення для кешу і пам'яті

Для того щоб подивитись як на швидкодію схеми вплинули додані блоки, введемо всі ті ж самі контрольні приклади. Вони будуть перевіряти схему на ті ж самі елементи. Після того, як буде отримано результати з урахуванням нових блокі, порівняємо їх.

Вводимо ті самі вхідні дані, що і в минулому першому контрольному прикладі, та отримуємо результат (рис.3.28) та також три графіки, які демонструють отримані результати (рис.3.29).

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 15
Validation Status: Memory Access Invalid (Address Not Found in Paging Table)
Cache Status: Address Not Found in Paging Table
Taint Cache Status: Data is Clean
Validation Status: Memory Access Invalid (Address Not Found in Paging Table)
Instruction Prefetch Time: 0.0012011 seconds
Data Prefetch Time: 0.0001827 seconds
ALU Execution Time: 0.0001394 seconds
Memory Validation Time: 0.0001757 seconds
Cache Access Time: 0.0005401 seconds
Total Simulation Time: 0.0025375 seconds
```

Рис. 3.28 – Результати перших вхідних даних

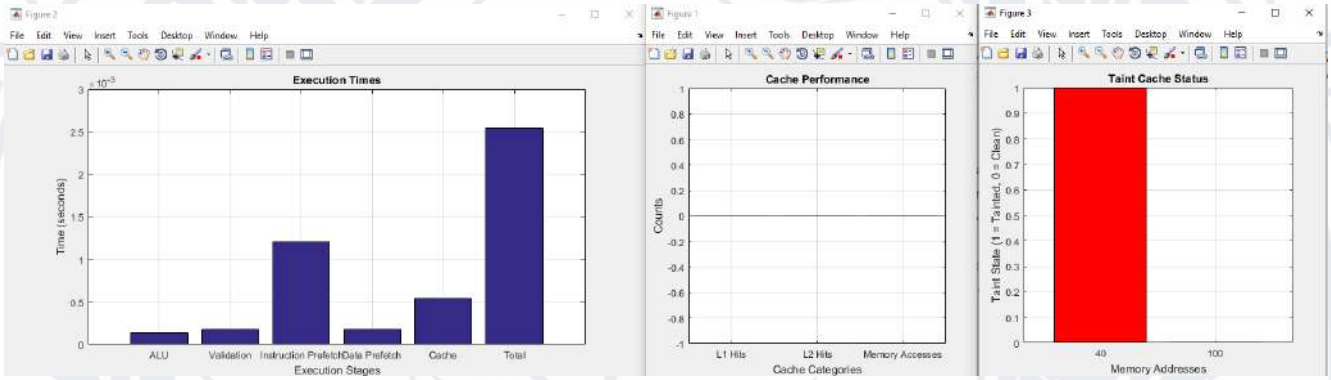

Рис.3.29 – Результуючі графіки до першого вхідного прикладу

Вводимо ті самі вхідні дані, що і в минулому другому контрольному прикладі, та отримуємо результат (рис.3.30) та також три графіки, які демонструють отримані результати (рис.3.31).

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 18
Validation Status: Memory Access Invalid (Address Not Found in Main Memory)
Cache Status: Memory Access Invalid
Taint Cache Status: Data is Clean
Validation Status: Memory Access Invalid (Address Not Found in Main Memory)
Instruction Prefetch Time: 0.0011973 seconds
Data Prefetch Time: 0.0001329 seconds
ALU Execution Time: 0.0001219 seconds
Memory Validation Time: 0.0002507 seconds
Cache Access Time: 0.0002617 seconds
Total Simulation Time: 0.0021139 seconds
```

Рис. 3.30 – Результати других вхідних даних

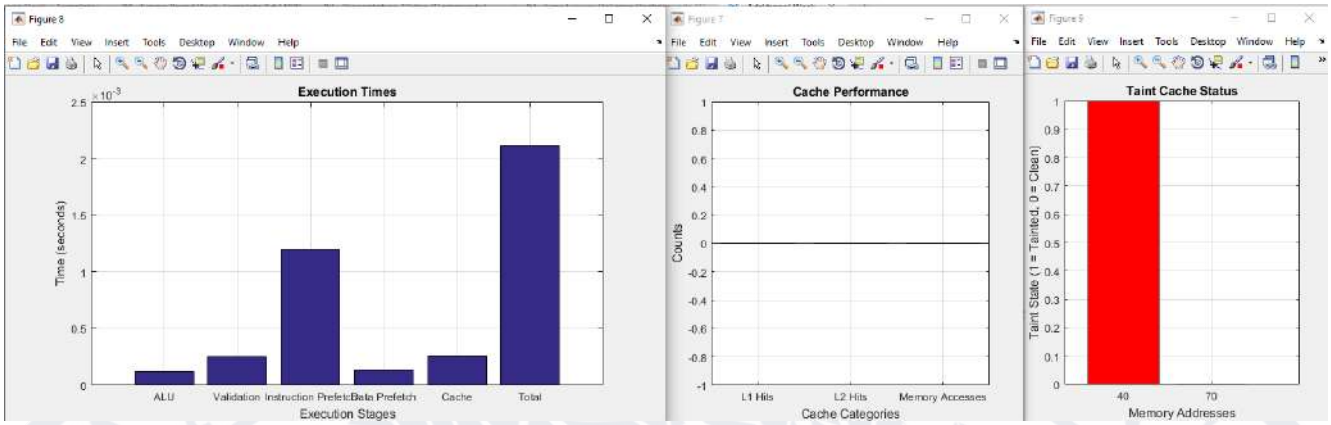

Рис.3.31 – Результуючі графіки до другого вхідного прикладу

Вводимо ті самі вхідні дані, що і в минулому третьому контрольному прикладі, та отримуємо результат (рис.3.32) та також три графіки, які демонструють отримані результати (рис.3.33).

```
=== CPU Simulation Results ===
Operation: Multiplication
ALU Result: 200
Validation Status: Memory Access Valid
Cache Status: L1 Cache Hit
Taint Cache Status: Data is Tainted
Validation Status: Memory Access Valid
Instruction Prefetch Time: 0.0012827 seconds
Data Prefetch Time: 0.0003705 seconds
ALU Execution Time: 0.0001323 seconds
Memory Validation Time: 0.0002261 seconds
Cache Access Time: 0.0001763 seconds
Total Simulation Time: 0.0023557 seconds
```

Рис. 3.32 – Результати третіх вхідних даних

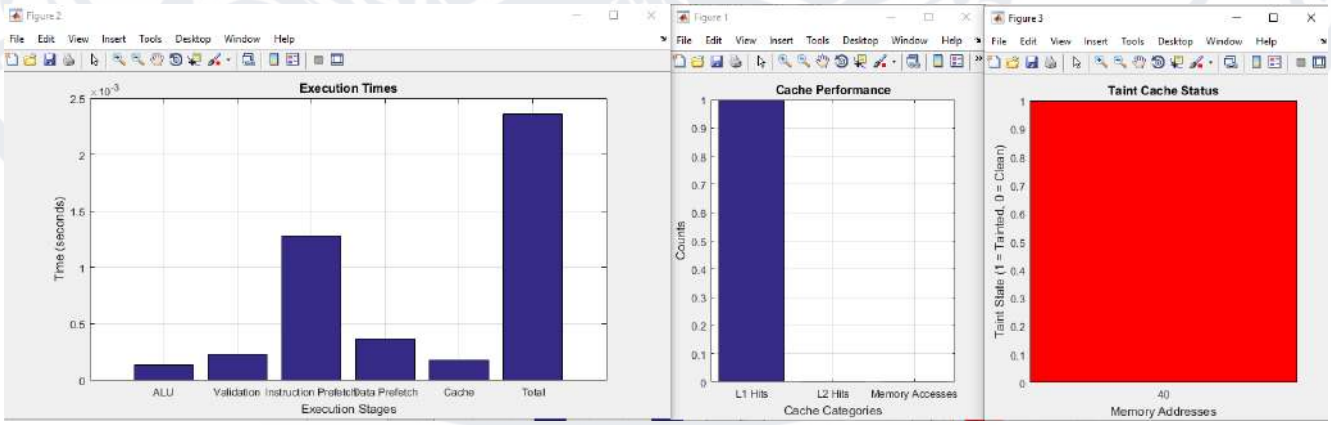

Рис.3.33 – Результуючі графіки до третього вхідного прикладу

Вводимо ті самі вхідні дані, що і в минулому четвертому контрольному прикладі, та отримуємо результат (рис.3.34) та також три графіки, які демонструють отримані результати (рис.3.35).

```
Operation: Division
ALU Result: 5
Validation Status: Memory Access Valid
Cache Status: L1 Cache Hit
Taint Cache Status: Data is Clean
Validation Status: Memory Access Valid
Instruction Prefetch Time: 0.0015236 seconds
Data Prefetch Time: 0.0004104 seconds
ALU Execution Time: 0.0006342 seconds
Memory Validation Time: 0.0009378 seconds
Cache Access Time: 0.000658 seconds
Total Simulation Time: 0.0048989 seconds
```

Рис. 3.34 – Результати четвертих вхідних даних

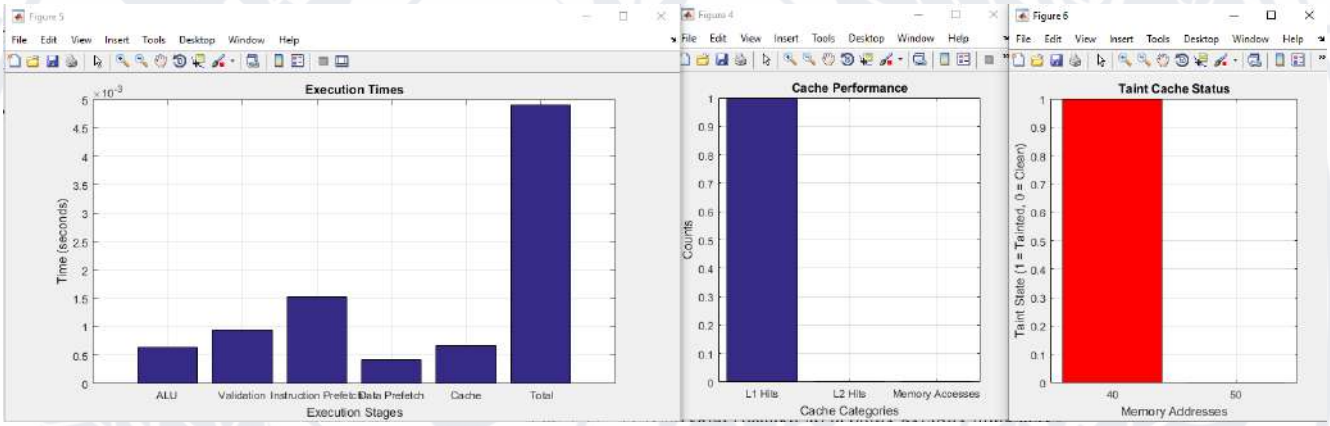

Рис.3.33 – Результуючі графіки до четвертого вхідного прикладу

Вводимо ті самі вхідні дані, що і в минулому п’ятому контрольному прикладі, та отримуємо результат (рис.3.34) та також три графіки, які демонструють отримані результати (рис.3.35).

```
=== CPU Simulation Results ===
Operation: Addition
ALU Result: 24
Validation Status: Memory Access Valid
Cache Status: L1 Cache Hit
Taint Cache Status: Data is Tainted
Validation Status: Memory Access Valid
Instruction Prefetch Time: 0.0011517 seconds
Data Prefetch Time: 0.0011162 seconds
ALU Execution Time: 0.000443 seconds
Memory Validation Time: 0.0001693 seconds
Cache Access Time: 9.27e-05 seconds
Total Simulation Time: 0.0030399 seconds
```

Рис.3.34 – Результати п’ятих вхідних даних

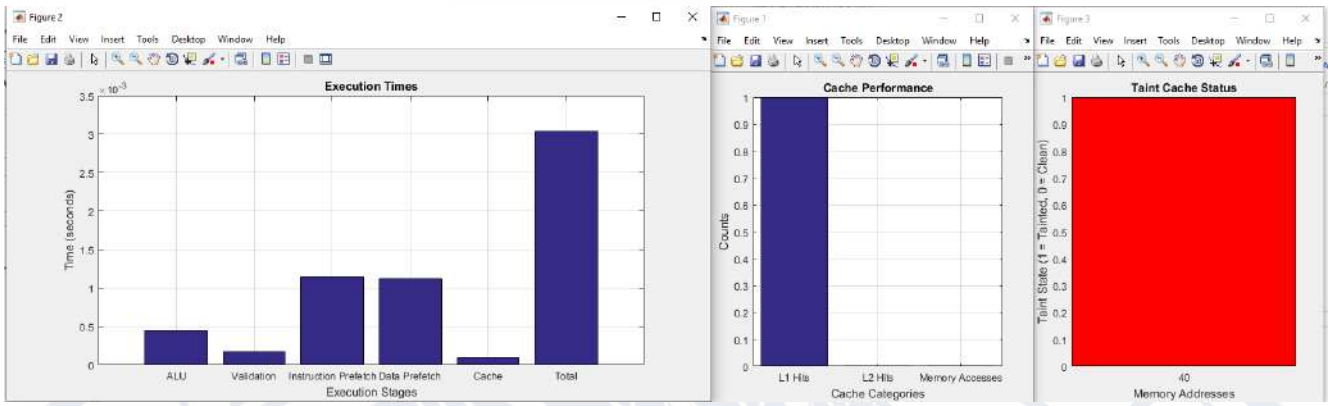

Рис.3.34 – Результуючі графіки до п’ятого вхідного прикладу

Вводимо ті самі вхідні дані, що і в минулому шостому контрольному прикладі, та отримуємо результат (рис.3.36) та також три графіки, які демонструють отримані результати (рис.3.37).

```
Operation: Multiplication
ALU Result: 45
Validation Status: Memory Access Valid
Cache Status: Memory Access
Taint Cache Status: Data is Clean
Validation Status: Memory Access Valid
Instruction Prefetch Time: 0.0011014 seconds
Data Prefetch Time: 0.0001117 seconds
ALU Execution Time: 2.2e-05 seconds
Memory Validation Time: 0.0001107 seconds
Cache Access Time: 0.0003691 seconds
Total Simulation Time: 0.0017496 seconds
```

Рис.3.35 – Результати шостих вхідних даних

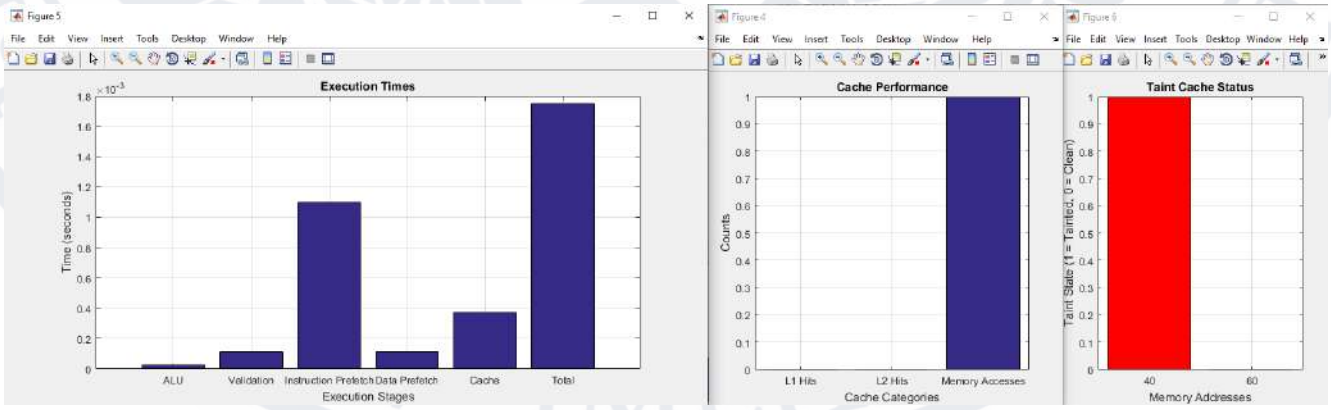

Рис.3.36 – Результуючі графіки до шостого вхідного прикладу

3.6. Висновок до розділу 3

На основі першого контрольного прикладу у двох схемах, була промодельовано чи система коректно реагує на ситуацію, коли віртуальна адреса введена користувачем, не існує в paging_table (тобто таблиці сторінок). В цьому випадку система не може знайти відповідність віртуальної адреси фізичній адресі, що блокує доступ до даних. Для того щоб все перевірити все і зробити розрахунки, прототип схеми витратив 0,0015517 секунд, в той час як покращена схема витратила 0,0025375 секунди, тобто на 0,0009858 секунди більше. Але слід врахувати той факт, що друга схема витратила більше часу, через те, що ще виконувала додаткову операції елементів *Prefetcher*-ів. Якщо брати сема час виконання математичної операції, то друга система виконала його на 0,0000012 секунд швидше; доступ до пам'яті відбувся на 0,0004182 секунд швидше; перевірка кешу відбулась на 0,0000077 секунд довше.

Другий контрольний приклад демонструє ситуацію, коли віртуальна адреса існує в paging_table, але фізична адреса, на яку вона мапиться, відсутня в основній пам'яті (main_memory). Це симулює помилку завантаження або видалення даних. Друга схема виконувала всі операції та перевірки на 0,0012411 секунди довше; математична операція також виконувалась довше, на 0,0000029 секунди, перевірка доступу до пам'яті відбувалась на 0,000009 секунди довше. Проте перевіка кешів відбувалась у другій схемі на 0,0001322 секунди швидше.

В третьому прикладі перевіряється робота механізму taint_cache, який ідентифікує "tainted" дані (небезпечні або скомпрометовані). Цей статус важливий для захисту системи від обробки потенційно шкідливих даних. Друга схема виконала свої дії на 0,0036734 секунди швидше, що є досить високою різницею в часі, враховуючи, що до схеми додались елементи; кеш перевірявся швидше на 0,0015085 секунди; доступ до пам'яті відбувся на 0,0014506 секунди швидше. Проте розрахунок математичних операцій відбувся на 0,0012387 секунди довше.

У четвертому випадку демонструєть найшвидший доступ до даних, коли вони вже знаходяться в L1_cache. В даному випадку, друга схема працювала на

0,0037657 секунди довше. Проте доступ до кешу, який перевірявся в даному випадку відбувся у другій схемі на 0,0002954 секунди швидше. Перевірка доступу до пам’яті у другій схемі зайняла на 0,0008219 секунди довше; виконання операцій відбулось на 0,0000132 секунди довше.

У п’ятому випадку було перевірено сценарій, коли дані відсутні в L1_cache, але знайдені в L2_cache. Після цього вони будуть завантажені в L1_cache. У цьому прикладі, по загальному часі виконання, друга схема «програла» на 0,0027471 секунди; на 0,000012 у роботі кешу. Проте у доступі до пам’яті, система «виграла» на 0,0000239 секунди; та у математичному розрахунку на 0,0004302 секунди.

В останньому прикладі демонструється сценарій, коли дані не знайдені в кешах, але доступні в основній пам'яті (main_memory). У даному випадку, друга схема виконує всі операції на 0,0006078 секунди швидше; доступ до кешу відбувся на 0,000614 секунди швидше; доступ до пам’яті відбувся на 0,0000097 секунди швидше та сама математична операція відбувалась на 0,0000038 секунди швидше.

ВИСНОВОК

У даній роботі проведено огляд відомих архітектур комп’ютерів, включаючи класичні та сучасні підходи до побудови апаратного забезпечення, а також методів перевірки даних, що використовуються для забезпечення надійності та точності роботи системи. Особливу увагу приділено аналізу структурних компонентів комп’ютерних систем і механізмів взаємодії між ними.

Окрім цього, було детально проаналізовано різні архітектури доступу до пам’яті, враховуючи такі аспекти, як: аналіз схем доступу до пам’яті, в тому числі пряму, асоціативну та каскадну організацію, а також методи оптимізації доступу; аналіз порівняльної ефективності різних схем доступу до пам’яті, включаючи час затримки та пропускну здатність; а також аналіз впливу цих схем на загальну продуктивність системи. Було досліджено вплив архітектури пам’яті на швидкодію процесора та взаємодію з іншими елементами системи.

Крім цього, проведено аналіз еволюції процесорів сімейства Pentium, розроблених на основі фон Неймановської архітектури. Розглянуто ключові технічні вдосконалення в кожному поколінні, такі як збільшення продуктивності, покращення систем кешування, оптимізація паралельної обробки команд та зменшення енергоспоживання. Окремо досліджено, як зміни в архітектурі вплинули на обчислювальну потужність і адаптацію до вимог сучасного програмного забезпечення.

У роботі також представлено розроблену схему доступу до пам’яті, що працює з хибними даними в частинах кешу. Було створено симулятивний код для перевірки ефективності запропонованої схеми. Симуляція дозволила детально вивчити, як добре працює ця схема, включаючи обробку помилкових записів, час доступу до пам’яті, обробку конфліктів у кеші та загальний вплив на продуктивність системи. Результати симуляції підтвердили, що запропонована схема має потенціал для впровадження у сучасних обчислювальних системах, особливо в умовах інтенсивної роботи з великими обсягами даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Мельник А. Архітектура комп’ютера. 2008, 469 с.
2. Архітектура фон Неймана і гарвардська архітектура. URL : http://surl.li/cfhajd
3. Pawson, R. The myth of the harvard architecture. *IEEE Annals of the History of Computing*, 2022, 44.3: 59-69 pp.
4. Салапатов В. І. Особливості підвищення якості програм у сигнальних процесорів. Вісник національного технічного університету України «КПІ», Інформатика, управління та обчислювальна техніка, 2009, 51: 74-77 с.
5. Гарвардська архітектура. URL: http://surl.li/ftxxrw
6. Francillon, A., Castelluccia, C. Code injection attacks on harvard-architecture devices. In: Proceedings of the 15th ACM conference on Computer and communications security. 2008, 15-26 pp.
7. Von Neumann, J. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 1993, 15.4: 27-75 pp.
8. Von Neumann architecture. URL: http://surl.li/sazgvr
9. За межами Quantum: MemComputing ASIC може зруйнувати 2048-бітне шифрування RSA. URL: http://surl.li/zwfhxv
10. Eigenmann, R., Lilja, D. J. Von neumann computers. Wiley Encyclopedia of Electrical and Electronics Engineering. 1998, 23: 387-400 pp.
11. Checksum URL: http://surl.li/onimx
12. Koopman, P., Driscoll, K., Hall, B. Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity. New Jersey, 2015. 111 pp.
13. Smith, A. J. Cache memories. *ACM Computing Surveys (CSUR)*, *14*(3), 1982. 473-530 pp.
14. Parity check. URL: http://surl.li/oniwv
15. Alouani, I., Niar, S., Kurdahi, F., Abid, M. Parity-based mono-copy cache for low power consumption and high reliability. In 2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP). IEEE, 2012. p. 44-48

16. Rossi, D., Timoncini, N., Spica, M., Metra, C. Error correcting code analysis for cache memory high reliability and performance. In: 2011 Design, Automation & Test in Europe. IEEE, 2011. p. 1-6.
17. Cache Coherence. URL: http://surl.li/onjct
18. Singh, I., Shriraman, A., Fung, W. W., O'Connor, M., Aamodt, T. M. Cache coherence for GPU architectures. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2013. p. 578-590.
19. Chartoff, R. P., Menczel, J. D., Dillman, S. H. Dynamic mechanical analysis (DMA). Thermal analysis of polymers: fundamentals and applications, 2009, 387-495.
20. Охримчук, Д. Д. Метод побудови моделі адаптивного асоціативного кешу в інформаційних системах. 2021.
21. Random-access memory. URL: http://surl.li/zqsoyr
22. Static random-access memory. URL: http://surl.li/pgealb
23. Recio, R., Metzler, B., Culley, P., Hilland, J., Garcia, D. A remote direct memory access protocol specification. 2007.
24. Chang, T. C., Chang, K. C., Tsai, T. M., Chu, T. J., Sze, S. M. Resistance random access memory. Materials Today. 2016, 19.5: 254-264 pp.
25. Dekker, R., Beenker, F., Thijssen, L. A realistic fault model and test algorithms for static random access memories. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1990, 9.6: 567-572 pp.
26. Ko, S. B. Pentium vs Pentium II. 1999 12p.
27. Ahn, J. W., Sung, W. Pentium-MMX-based implementation of a digital copier. In: 1998 IEEE Workshop on Signal Processing Systems. SIPS 98. Design and Implementation (Cat. No. 98TH8374). IEEE, 1998. 142-151 pp.
28. Peleg, A., Wilkie, S., Weiser, U. Intel MMX for multimedia PCs. Communications of the ACM. 1997, 40.1: 24-38 pp.
29. Intel Pentium MMX 166 MHz specifications, URL: http://surl.li/shxhye
30. Shanley, T. Pentium Pro and Pentium II system architecture. Addison-Wesley Professional, 1998.

31. Diefendorff, K. Pentium iii= pentium ii+ sse. Microprocessor Report. 1999, 13.3: 1-6 pp.
32. Ram, A. Pentium III. *Personal Computer World*, 1999, 22.4: 122-4.
33. Boggs, D., Baktha, A., Hawkins, J., Marr, D. T., Miller, J. A., Roussel, P., Venkatraman, K. S. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. Intel Technology Journal, 2004, 8.1.
34. Sprunt, B. Brinkley. Pentium 4 performance-monitoring features. *Ieee Micro*, 2002, 22.04: 72-82 pp.
35. Ahn, D., Lee, G. A memory-access validation scheme against payload injection attacks. IEEE Transactions on Dependable and Secure Computing, 2014, 12.4: 387-399 pp.
36. MATLAB. URL: http://surl.li/wbjlig
37. Prefetcher. UTL: http://surl.li/nwlxbo

ДОДАТКИ

ДОДАТОК А Порівняльна таблиця методів перевірки даних

ДОДАТОК Б Симулятивний код схеми доступу до пам’яті, що працює з хибними даними в частинах кеш для прототипу

ДОДАТОК В Код для побудови графіків для прототипу

ДОДАТОК Г Симулятивний код для покращеної схеми доступу до пам’яті, що працює з хибними даними в частинах кеш для прототипу.

ДОДАТОК Д Код для побудови графіків для покращеної схеми

ДОДАТОК А

Порівняльна таблиця методів перевірки даних

Метод	Призначення	Механізм роботи	Переваги	Недоліки
Метод контрольної суми	Перевірка цілісності даних	Обчислює контрольну суму (наприклад, CRC або MD5), яка додається до переданих даних; приймач повторно обчислює контрольну суму для перевірки точності	Простий у використанні, невелика обчислювальна складність	Не може виявляти всі помилки, особливо якщо дані змінюються аналогічним чином
Метод перевірки тегів	Ідентифікація помилок у пам'яті або даних	Присвоює унікальний тег або ідентифікатор кожному фрагменту даних або блоку пам'яті для контролю за змінами	Забезпечує швидку перевірку точності в складних системах	Обмежений в застосуванні через необхідність додаткової пам’яті або ресурсів

Метод перевірки парності	Виявлення одиничних помилок у даних	Додає один біт парності до кожного байта або блоку даних; перевіряє кількість 1-бітів (парність/непарність)	Простий і швидкий спосіб виявлення помилок	Виявляє лише одну помилку, не може виправляти або виявляти декілька помилок
ECC	Виявлення та виправлення помилок у пам'яті	Використовує код Хеммінга або інші алгоритми для виявлення та виправлення помилок у пам’яті	Може виявляти та виправляти окремі помилки, виявляти кілька помилок одночасно	Вимагає додаткової пам'яті та впливає на продуктивність за рахунок обчислень
Кеш-когерентність	Синхронізація даних у кешах багатоядерних систем	Використовує протоколи (наприклад, MESI) для збереження синхронізації кешів на різних ядрах	Зменшує кількість помилок при доступі до даних, підвищує продуктивність в багатоядерних системах	Збільшує складність системи, потребує додаткових обчислювальних ресурсів

ДОДАТОК Б

Симулятивний код схеми доступу до пам’яті, що працює з хибними даними в частинах кеш для прототипу.

```
%% CPU Simulation with Memory Access Validation
% Ініціалізація системних компонентів
register_file = zeros(1, 8); % Реєстровий файл
L1_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L1 кеш
L2_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L2 кеш
paging_table = containers.Map({30, 40, 50, 70}, {100, 200, 300, 400}); % Таблиця сторінок
main_memory = containers.Map({100, 200, 300}, {256, 512, 1024}); % Основна пам'ять
taint_cache = containers.Map('KeyType', 'double', 'ValueType', 'logical'); % Taint Cache

% Лічильники продуктивності кешу
cache_hits_L1 = 0; % Хіти L1
cache_hits_L2 = 0; % Хіти L2
memory_accesses = 0; % Доступи до пам'яті

% Попередньо додані значення для кешу і пам'яті
L1_cache(50) = 128; % Додано значення в L1 Cache
L2_cache(40) = 256; % Додано значення в L2 Cache
main_memory(300) = 512; % Додано значення в основну пам'ять
paging_table(60) = 300; % Віртуальна адреса 60 мапиться на фізичну адресу 300

% Позначаємо одну адресу як "tainted"
tainted_address = 40; % Вибрана адреса для позначення
taint_cache(tainted_address) = true; % Позначаємо адресу 40 як "tainted"

%% Введення операндів для ALU
operand1 = input('Enter the first operand: '); % Користувач вводить перший операнд
operand2 = input('Enter the second operand: '); % Користувач вводить другий операнд
```

```
% Вибір операції для ALU
disp('Select operation:');
disp('1 - Addition');
disp('2 - Subtraction');
disp('3 - Multiplication');
disp('4 - Division');
operation_code = input('Enter the operation code (1-4): ');

% Введення віртуальної адреси
virtual_address = input('Enter the virtual memory address: ');

%% Лічильники часу
timing_ALU = 0; % Час виконання ALU
timing_cache = 0; % Час доступу до кешу
timing_memory_validation = 0; % Час перевірки доступу до пам’яті
timing_total = 0; % Загальний час

%% Виконання ALU операцій
tic; % Початок загального таймера
alu_start_time = tic; % Початок таймера ALU
switch operation_code
    case 1 % Додавання
        alu_result = operand1 + operand2;
        operation_name = 'Addition';
    case 2 % Віднімання
        alu_result = operand1 - operand2;
        operation_name = 'Subtraction';
    case 3 % Множення
        alu_result = operand1 * operand2;
        operation_name = 'Multiplication';
    case 4 % Ділення
        if operand2 ~= 0
            alu_result = operand1 / operand2;
            operation_name = 'Division';
        else
            alu_result = NaN; % Якщо ділення на нуль
            operation_name = 'Error: Division by Zero';
        end
    otherwise
```

```
        alu_result = NaN; % Якщо код операції невідомий
        operation_name = 'Unknown Operation';
end
timing_ALU = toc(alu_start_time); % Час виконання ALU

% Зберігаємо результат у перший регістр
register_file(1) = alu_result;

%% Перевірка доступу до пам'яті (Memory Access Validation Unit)
validation_start_time = tic;
if isKey(paging_table, virtual_address)
    physical_address = paging_table(virtual_address); % Отримуємо фізичну адресу
    if isKey(main_memory, physical_address)
        validation_status = 'Memory Access Valid';
    else
        validation_status = 'Memory Access Invalid (Address Not Found in Main
Memory)';
    end
else
    validation_status = 'Memory Access Invalid (Address Not Found in Paging Table)';
end
timing_memory_validation = toc(validation_start_time);

%% Перевірка кешу
cache_start_time = tic;
if isKey(L1_cache, virtual_address)
    data = L1_cache(virtual_address); % Дані знайдено в L1 кеші
    cache_status = 'L1 Cache Hit';
    cache_hits_L1 = cache_hits_L1 + 1; % Оновлюємо хіти L1
elseif isKey(L2_cache, virtual_address)
    data = L2_cache(virtual_address); % Дані знайдено в L2 кеші
    cache_status = 'L2 Cache Hit';
    cache_hits_L2 = cache_hits_L2 + 1; % Оновлюємо хіти L2
    L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
elseif isKey(paging_table, virtual_address)
    if strcmp(validation_status, 'Memory Access Valid')
        data = main_memory(physical_address); % Дані знайдено в пам'яті
        cache_status = 'Memory Access';
        memory_accesses = memory_accesses + 1; % Оновлюємо доступи до пам'яті
```

```
        L2_cache(virtual_address) = data; % Завантажуємо в L2 кеш
        L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
    else
        data = NaN; % Адреса не знайдена
        cache_status = 'Memory Access Invalid';
    end
else
    data = NaN; % Віртуальна адреса не знайдена
    cache_status = 'Address Not Found in Paging Table';
end
timing_cache = toc(cache_start_time); % Час доступу до кешу

% Завершення загального часу
timing_total = toc; % Загальний час

%% Перевірка Taint Cache
if isKey(taint_cache, virtual_address) && taint_cache(virtual_address)
    taint_status = 'Data is Tainted';
else
    taint_status = 'Data is Clean';
    taint_cache(virtual_address) = false; % Позначаємо адресу як "clean"
end

%% Вивід результатів
disp('=== CPU Simulation Results ===');
disp(['Operation: ', operation_name]);
disp(['ALU Result: ', num2str(alu_result)]);
disp(['Validation Status: ', validation_status]);
disp(['Cache Status: ', cache_status]);
disp(['Taint Cache Status: ', taint_status]);
disp(['ALU Execution Time: ', num2str(timing_ALU), ' seconds']);
disp(['Memory Validation Time: ', num2str(timing_memory_validation), ' seconds']);
disp(['Cache Access Time: ', num2str(timing_cache), ' seconds']);
disp(['Total Simulation Time: ', num2str(timing_total), ' seconds']);
```

ДОДАТОК В

Код для побудови графіків для прототипу

```
%% Побудова графіків
% 1. Продуктивність кешу
figure;
cache_data = [cache_hits_L1, cache_hits_L2, memory_accesses];
bar(cache_data);
set(gca, 'XTickLabel', {'L1 Hits', 'L2 Hits', 'Memory Accesses'});
title('Cache Performance');
xlabel('Cache Categories');
ylabel('Counts');
grid on;

% 2. Графік часу виконання
figure;
timing_data = [timing_ALU, timing_memory_validation, timing_cache, timing_total];
bar(timing_data);
set(gca, 'XTickLabel', {'ALU Time', 'Validation Time', 'Cache Time', 'Total Time'});
title('Execution Times');
xlabel('Execution Stages');
ylabel('Time (seconds)');
grid on;

% 3. Стан Taint Cache
figure;
taint_addresses = keys(taint_cache);
taint_states = cell2mat(values(taint_cache));
bar(cell2mat(taint_addresses), double(taint_states), 'r');
title('Taint Cache Status');
xlabel('Memory Addresses');
ylabel('Taint State (1 = Tainted, 0 = Clean)');
grid on;
```

ДОДАТОК Г

Симулятивний код для покращеної схеми доступу до пам’яті, що працює з хибними даними в частинах кеш для прототипу.

```
%% CPU Simulation with Memory Access Validation
% Ініціалізація системних компонентів
register_file = zeros(1, 8); % Реєстровий файл
L1_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L1 кеш
L2_cache = containers.Map('KeyType', 'double', 'ValueType', 'double'); % L2 кеш
paging_table = containers.Map({30, 40, 50, 70}, {100, 200, 300, 400}); % Таблиця сторінок
main_memory = containers.Map({100, 200, 300}, {256, 512, 1024}); % Основна пам'ять
taint_cache = containers.Map('KeyType', 'double', 'ValueType', 'logical'); % Taint Cache

% Лічильники продуктивності кешу
cache_hits_L1 = 0; % Хіти L1
cache_hits_L2 = 0; % Хіти L2
memory_accesses = 0; % Доступи до пам'яті

% Попередньо додані значення для кешу і пам'яті
L1_cache(50) = 128; % Додано значення в L1 Cache
L2_cache(40) = 256; % Додано значення в L2 Cache
main_memory(300) = 512; % Додано значення в основну пам'ять
paging_table(60) = 300; % Віртуальна адреса 60 мапиться на фізичну адресу 300

% Позначаємо одну адресу як "tainted"
tainted_address = 40; % Вибрана адреса для позначення
taint_cache(tainted_address) = true; % Позначаємо адресу 40 як "tainted"

%% Введення операндів для ALU
operand1 = input('Enter the first operand: '); % Користувач вводить перший операнд
operand2 = input('Enter the second operand: '); % Користувач вводить другий операнд

% Вибір операції для ALU
disp('Select operation:');
```

```
disp('1 - Addition');
disp('2 - Subtraction');
disp('3 - Multiplication');
disp('4 - Division');
operation_code = input('Enter the operation code (1-4): ');

% Введення віртуальної адреси
virtual_address = input('Enter the virtual memory address: ');

%% Лічильники часу
timing_ALU = 0; % Час виконання ALU
timing_cache = 0; % Час доступу до кешу
timing_memory_validation = 0; % Час перевірки доступу до пам’яті
timing_total = 0; % Загальний час

%% Виконання ALU операцій
tic; % Початок загального таймера
alu_start_time = tic; % Початок таймера ALU
switch operation_code
    case 1 % Додавання
        alu_result = operand1 + operand2;
        operation_name = 'Addition';
    case 2 % Віднімання
        alu_result = operand1 - operand2;
        operation_name = 'Subtraction';
    case 3 % Множення
        alu_result = operand1 * operand2;
        operation_name = 'Multiplication';
    case 4 % Ділення
        if operand2 ~= 0
            alu_result = operand1 / operand2;
            operation_name = 'Division';
        else
            alu_result = NaN; % Якщо ділення на нуль
            operation_name = 'Error: Division by Zero';
        end
    otherwise
        alu_result = NaN; % Якщо код операції невідомий
        operation_name = 'Unknown Operation';
```

```
end
timing_ALU = toc(alu_start_time); % Час виконання ALU

% Зберігаємо результат у перший регістр
register_file(1) = alu_result;

%% Перевірка доступу до пам'яті (Memory Access Validation Unit)
validation_start_time = tic;
if isKey(paging_table, virtual_address)
   physical_address = paging_table(virtual_address); % Отримуємо фізичну адресу
   if isKey(main_memory, physical_address)
      validation_status = 'Memory Access Valid';
   else
      validation_status = 'Memory Access Invalid (Address Not Found in Main
Memory)';
   end
else
   validation_status = 'Memory Access Invalid (Address Not Found in Paging Table)';
end
timing_memory_validation = toc(validation_start_time);

%% Instruction Prefetching
prefetch_start_time = tic;
disp('Instruction Prefetching: Loading next instruction...');
% Емуляція роботи Instruction Prefetcher
pause(0.001); % Затримка для імітації
timing_instruction_prefetch = toc(prefetch_start_time);

%% Data Prefetching
data_prefetch_start_time = tic;
disp('Data Prefetching: Loading data to L1 Cache...');
if isKey(L2_cache, virtual_address)
   L1_cache(virtual_address) = L2_cache(virtual_address); % Завантаження даних у
L1 Cache
end
timing_data_prefetch = toc(data_prefetch_start_time);

%% Перевірка кешу
cache_start_time = tic;
```

```
if isKey(L1_cache, virtual_address)
    data = L1_cache(virtual_address); % Дані знайдено в L1 кеші
    cache_status = 'L1 Cache Hit';
    cache_hits_L1 = cache_hits_L1 + 1; % Оновлюємо хіти L1
elseif isKey(L2_cache, virtual_address)
    data = L2_cache(virtual_address); % Дані знайдено в L2 кеші
    cache_status = 'L2 Cache Hit';
    cache_hits_L2 = cache_hits_L2 + 1; % Оновлюємо хіти L2
    L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
elseif isKey(paging_table, virtual_address)
    if strcmp(validation_status, 'Memory Access Valid')
        data = main_memory(physical_address); % Дані знайдено в пам'яті
        cache_status = 'Memory Access';
        memory_accesses = memory_accesses + 1; % Оновлюємо доступи до пам'яті
        L2_cache(virtual_address) = data; % Завантажуємо в L2 кеш
        L1_cache(virtual_address) = data; % Завантажуємо в L1 кеш
    else
        data = NaN; % Адреса не знайдена
        cache_status = 'Memory Access Invalid';
    end
else
    data = NaN; % Віртуальна адреса не знайдена
    cache_status = 'Address Not Found in Paging Table';
end
timing_cache = toc(cache_start_time); % Час доступу до кешу

% Завершення загального часу
timing_total = toc; % Загальний час

%% Перевірка Taint Cache
if isKey(taint_cache, virtual_address) && taint_cache(virtual_address)
    taint_status = 'Data is Tainted';
else
    taint_status = 'Data is Clean';
    taint_cache(virtual_address) = false; % Позначаємо адресу як "clean"
end

%% Вивід результатів
disp('=== CPU Simulation Results ===');
```

```
disp(['Operation: ', operation_name]);
disp(['ALU Result: ', num2str(alu_result)]);
disp(['Validation Status: ', validation_status]);
disp(['Cache Status: ', cache_status]);
disp(['Taint Cache Status: ', taint_status]);
disp(['Validation Status: ', validation_status]);
disp(['Instruction Prefetch Time: ', num2str(timing_instruction_prefetch), ' seconds']);
disp(['Data Prefetch Time: ', num2str(timing_data_prefetch), ' seconds']);
disp(['ALU Execution Time: ', num2str(timing_ALU), ' seconds']);
disp(['Memory Validation Time: ', num2str(timing_memory_validation), ' seconds']);
disp(['Cache Access Time: ', num2str(timing_cache), ' seconds']);
disp(['Total Simulation Time: ', num2str(timing_total), ' seconds']);
```

ДОДАТОК Д

Код для побудови графіків для покращеної схеми

```
%% Побудова графіків

% 1. Продуктивність кешу
figure;
cache_data = [cache_hits_L1, cache_hits_L2, memory_accesses];
bar(cache_data);
set(gca, 'XTickLabel', {'L1 Hits', 'L2 Hits', 'Memory Accesses'});
title('Cache Performance');
xlabel('Cache Categories');
ylabel('Counts');
grid on;

% 2. Графік часу виконання
figure;
timing_data = [timing_ALU, timing_memory_validation, timing_instruction_prefetch, timing_data_prefetch, timing_cache, timing_total];
bar(timing_data);
set(gca, 'XTickLabel', {'ALU', 'Validation', 'Instruction Prefetch', 'Data Prefetch', 'Cache', 'Total'});
title('Execution Times');
xlabel('Execution Stages');
ylabel('Time (seconds)');
grid on;

% 3. Стан Taint Cache
figure;
taint_addresses = keys(taint_cache);
taint_states = cell2mat(values(taint_cache));
bar(cell2mat(taint_addresses), double(taint_states), 'r');
title('Taint Cache Status');
xlabel('Memory Addresses');
ylabel('Taint State (1 = Tainted, 0 = Clean)');
grid on;
```