Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА FPGA СИСТЕМИ ПЕРЕВІРКИ ЦІЛІСНОСТІ ДАНИХ

Виконав(ла): студент(ка) 4 курсу,
групи КНТ-521

Спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗУБ Р.І.
(ПРІЗВИЩЕ та ініціали)

Керівник ГРУШКО С.С.
(ПРІЗВИЩЕ та ініціали)
Рецензент КОЗІНА Г.Л.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
_____
«14» квітня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ЗУБА Руслана Ігоровича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка FPGA системи перевірки цілісності даних

керівник проєкту (роботи) к.т.н., доцент, ГРУШКО С.С.,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01.06.2025 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, програмне забезпечення Altera Quartus II 13.1, програмне забезпечення Aldec Active-HDL 9.1.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) Аналіз предметної області;
2) Проєктування архітектури системи;
3) Реалізація системи на FPGA.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-3	ГРУШКО С.С.		
Нормоконтроль	ПОЛЬСЬКА О.В.		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області	до 18.04.2025	
2	Визначення вимог до системи	до 22.04.2025	
3	Проєктування архітектури системи	до 24.04.2025	
4	Вибір FPGA-платформи	до 28.04.2025	
5	Створення дизайну проєкту	до 01.05.2025	
6	Реалізація модулів на HDL	до 05.05.2025	
7	Синтез проєкту	до 12.05.2025	
8	Тестування та верифікація системи	до 22.05.2025	
9	Оформлення пояснювальної записки	до 25.05.2025	
10	Проходження нормоконтролю	до 01.06.2025	
11	Перевірка на наявність академічного плагіату	до 03.06.2025	
12	Проходження рецензування	до 10.06.2025	

Студент(ка) _____ Руслан ЗУБ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Світлана ГРУШКО
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 72 с., 1 табл., 14 рис., 2 дод., 28 джерел.

ВЕРИФІКАЦІЯ, КОНВЕЄРНА ОБРОБКА, КРИПТОГРАФІЧНА ХЕШ-ФУНКЦІЯ, ТЕСТУВАННЯ, ЦІЛІСНІСТЬ ДАНИХ, AXI-STREAM, CYCLONE IV, FPGA, MD5, VERILOG

Об’єкт розробки – апаратна система на базі FPGA для перевірки цілісності даних за допомогою криптографічної хеш-функції MD5.

Мета роботи – розробка ефективної FPGA-системи для швидкої та енергоефективної перевірки цілісності даних, придатної для використання у вбудованих системах і телекомунікаціях.

У ході виконання роботи було проведено аналіз предметної області, що включав дослідження понять цілісності даних, криптографічних хеш-функцій і існуючих рішень. Обґрунтовано вибір алгоритму MD5 і платформи Cyclone IV. Спроєктовано архітектуру системи з двома модулями: інтерфейсним і хешуючим. Реалізацію виконано на мові Verilog у середовищі Quartus II 13.1. Тестування в Aldec Active-HDL 9.1 і апаратно на платі Cyclone IV підтвердило коректність обчислень хеш-значень для різних вхідних даних, включаючи порожні, односимвольні, стандартні та багатоблокові повідомлення. Функціональна верифікація засвідчила збіг результатів із еталонними значеннями.

Розроблена система є компактною, гнучкою та готовою до використання в реальних сценаріях, таких як телекомунікації чи вбудовані системи. Її модульна архітектура дозволяє адаптацію до інших хеш-функцій, що забезпечує потенціал для подальшого розвитку.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AES	–	Advanced Encryption Standard, Стандарт шифрування даних
AMD	–	Advanced Micro Devices, Розширені мікропристрої
ASIC	–	Application-Specific Integrated Circuit, Спеціалізована інтегральна схема
BRAM	–	Block Random Access Memory, Блокова оперативна пам’ять
CPU	–	Central Processing Unit, Центральний процесор
FPGA	–	Field-Programmable Gate Array, Програмована логічна інтегральна схема
HAIFA	–	Hash Iterative Framework, Ітеративна структура хешування
HDL	–	Hardware Description Language, Мова опису апаратного забезпечення
HSM	–	Hardware Security Module, Модуль апаратної безпеки
I/Os	–	Inputs/Outputs, Входи/Виходи
LEs	–	Logic Elements, Логічні елементи
MD5	–	Message Digest Algorithm 5, Алгоритм хешування повідомлень 5
NIST	–	National Institute of Standards and Technology, Національний інститут стандартів і технологій
PCIe	–	Peripheral Component Interconnect Express, Експрес-інтерфейс периферійних компонентів
RIPEMD	–	RACE Integrity Primitives Evaluation Message Digest, Хеш-алгоритм оцінки цілісності даних
SHA	–	Secure Hash Algorithm, Безпечний алгоритм хешування
TPM	–	Trusted Platform Module, Модуль довіреної платформи
VHDL	–	VHSIC Hardware Description Language, Мова опису апаратного забезпечення для надвисокошвидкісних інтегральних схем

ВСТУП

Перевірка цілісності даних є однією з ключових задач сучасних інформаційних систем, оскільки вона забезпечує точність, надійність і захист інформації від несанкціонованих змін. У цифрову еру, коли обсяги даних зростають експоненціально, а кіберзагрози стають дедалі складнішими, забезпечення цілісності даних набуває критичного значення для організацій, державних установ і приватних осіб. Дані є основою для прийняття рішень у бізнесі, управлінні, охороні здоров'я, фінансах і багатьох інших сферах, тому будь-яке їхнє спотворення може призвести до серйозних наслідків, включаючи фінансові втрати, порушення безпеки чи навіть загрозу життю. Наприклад, у фінансових системах зміна транзакційних даних може спричинити шахрайство, а в медичних інформаційних системах — помилки в лікуванні через некоректні дані про пацієнтів.

Розвиток технологій, таких як інтернет речей, хмарні обчислення, блокчейн і автономні системи, ще більше підкреслює необхідність швидких, ефективних і безпечних методів перевірки цілісності даних. Водночас зростання кіберзлочинності, зокрема атак типу "людина посередині", програм-вимагачів і інсайдерських загроз, створює нові виклики для захисту інформації. Криптографічні хеш-функції, такі як SHA-256, SHA-3 і BLAKE2, стали стандартом для забезпечення цілісності даних, оскільки вони дозволяють швидко виявляти навіть найменші зміни в інформації. Однак програмні рішення на універсальних процесорах часто не відповідають вимогам швидкості та енергоефективності для обробки великих обсягів даних у реальному часі, що характерно для вбудованих систем, телекомунікацій чи блокчейн-застосувань.

У цьому контексті апаратні рішення, зокрема на основі програмованих логічних інтегральних схем, набувають особливої актуальності. FPGA поєднують високу швидкодію, порівнянну з спеціалізованими мікросхемами, із гнучкістю, що дозволяє адаптувати систему до нових алгоритмів і вимог. Вони ідеально підходять

для створення систем перевірки цілісності даних, які потребують як високої продуктивності, так і можливості швидкої модифікації. Розробка FPGA-системи для перевірки цілісності даних є актуальною задачею, оскільки вона відповідає сучасним потребам у швидких, безпечних і гнучких рішеннях для захисту інформації в умовах зростання обсягів даних і кіберзагроз.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні поняття цілісності даних

1.1.1 Визначення цілісності даних

Цілісність даних — це властивість інформаційних систем, яка забезпечує точність, узгодженість, повноту та надійність даних протягом усього їхнього життєвого циклу, включаючи створення, зберігання, обробку, передачу та видалення. Вона гарантує, що дані залишаються в тому вигляді, в якому були створені або збережені, без несанкціонованих чи випадкових змін. Цілісність даних є ключовим аспектом інформаційної безпеки та управління даними, оскільки вона запобігає помилкам, фальсифікаціям або втратам даних, які можуть виникнути через технічні збої, людські помилки чи зловмисні дії [1].

Цілісність даних відрізняється від безпеки даних, хоча ці поняття тісно пов’язані. Безпека даних фокусується на захисті від несанкціонованого доступу, тоді як цілісність даних забезпечує, щоб самі дані залишалися незмінними та точними. Наприклад, якщо хакер отримує доступ до бази даних, але не змінює дані, цілісність може залишатися непорушеною. Однак, якщо дані змінюються через атаку чи збій, це є порушенням цілісності.

Ключові аспекти цілісності даних:

- точність: дані повинні бути правильними, без помилок чи неточностей. Наприклад, якщо в базі даних клієнтів вказано неправильну адресу, це порушує точність;

- узгодженість: дані повинні бути однорідними та відповідати встановленим правилам чи форматам. Наприклад, дати в базі даних повинні мати однаковий формат;

- повнота: усі необхідні дані повинні бути присутніми. Наприклад, якщо в медичній картці відсутні результати аналізів, це порушує повноту;

- надійність: дані повинні бути придатними для використання та відповідати їхньому призначенню. Наприклад, фінансова звітність має бути надійною для

прийняття рішень.

Цілісність даних відіграє ключову роль у кількох аспектах:

- прийняття обґрунтованих рішень: точні та надійні дані дозволяють організаціям приймати правильні рішення. Наприклад, бізнес може аналізувати дані про продажі, щоб визначити стратегію, але якщо дані спотворені, це може призвести до помилок;

- відповідність регуляціям: багато галузей підпадають під суворі регуляції, які вимагають забезпечення цілісності даних для захисту конфіденційної інформації та безпеки;

- операційна ефективність: цілісність даних зменшує помилки та невідповідності, які можуть спричинити збої в роботі систем;

- довіра до систем: організації, які забезпечують цілісність даних, підтримують довіру клієнтів і партнерів, що особливо важливо для компаній, які обробляють персональні дані [2-6].

1.1.2 Загрози для цілісності даних

Цілісність даних є критично важливою для забезпечення надійності інформаційних систем, але вона може бути порушена через різноманітні загрози. Ці загрози виникають як через природні причини, так і через людську діяльність, і їх розуміння є ключовим для розробки ефективних систем захисту. Нижче наведено основні типи загроз, класифіковані за їхньою природою та походженням:

а) технічні збої є одними з найпоширеніших причин порушення цілісності даних. Вони включають:

1) апаратні несправності: пошкодження фізичних носіїв даних може призвести до втрати або спотворення даних. Наприклад, збій у роботі жорсткого диска може спричинити втрату частини інформації;

2) програмні помилки в програмному забезпеченні, які обробляють або зберігають дані, можуть призводити до їх пошкодження. Наприклад, помилка в алгоритмі обробки даних може змінити значення в базі даних;

3) під час передачі даних через мережу можуть виникати втрати або спотворення через нестабільність зв’язку, перешкоди або помилки в мережевих

протоколах. Наприклад, втрата пакетів даних у ненадійних мережах може призвести до неповноти інформації;

б) зловмисні дії становлять значну загрозу для цілісності даних, особливо в умовах зростання кіберзлочинності. До них належать:

1) хакерські атаки: атаки типу "людина посередині" дозволяють зловмисникам перехоплювати та змінювати дані під час їхньої передачі. Наприклад, хакер може змінити фінансові транзакції в реальному часі;

2) шкідливе програмне забезпечення, таке як віруси або програми-вимагачі, може змінювати, видаляти або шифрувати дані, роблячи їх недоступними або ненадійними;

3) фішинг: зловмисники можуть обманом отримати доступ до систем, що дозволяє їм змінювати дані. Наприклад, фішингова атака може призвести до компрометації облікових даних адміністратора;

4) інсайдерські загрози: співробітники або особи з легітимним доступом до систем можуть змінювати дані. Наприклад, працівник може випадково видалити важливі записи або навмисно змінити їх для особистої вигоди;

в) людський фактор є значним джерелом загроз для цілісності даних, оскільки навіть ненавмисні дії можуть мати серйозні наслідки:

1) невірний ввід даних: помилки при введенні інформації, такі як неправильне введення цифр або форматів, можуть призвести до неточностей. Наприклад, введення неправильної дати народження в медичній базі даних;

2) неправильне редагування: випадкове видалення або зміна даних під час роботи з базами даних або файлами. Наприклад, видалення рядка в таблиці Excel через неуважність;

3) мисливська помилка: неправильне налаштування систем або конфігурацій, що може призвести до втрат;

г) фізичні фактори також можуть впливати на цілісність даних:

1) природні катастрофи: пожежі, повені, землетруси або інші стихійні лиха можуть фізично пошкодити носії даних, такі як сервери або жорсткі диски;

2) механічні пошкодження: падіння, удари або інші механічні впливи

на носії даних можуть призвести до втрати або пошкодження інформації;

д) передача даних між різними системами або форматами може призводити до втрати або спотворення інформації. Наприклад, дані, створені в Microsoft Excel, можуть некоректно відображатися при перенесенні в інший формат, якщо не враховані особливості форматування;

е) неповна або неточна інформація, зібрана на початковому етапі, може призвести до порушення цілісності. Наприклад, якщо фінансовий звіт не враховує всіх джерел доходу, це може вплинути на подальші аналізи [7].

1.1.3 Методи забезпечення цілісності інформації

Для захисту цілісності даних використовуються різноманітні методи, які можна реалізувати як на програмному, так і на апаратному рівні. Ці методи спрямовані на запобігання змінам даних, виявлення порушень і відновлення інформації у разі втрати. Нижче наведено основні методи забезпечення цілісності інформації:

- криптографічні хеш-функції створюють унікальний цифровий відбиток (хеш) для набору даних. Будь-яка зміна даних призводить до зміни хеша, що дозволяє виявити порушення цілісності. Популярні алгоритми включають MD5, SHA-1, SHA-256 і SHA-3;

- коди виявлення помилок, такі як контрольні суми або циклічні надлишкові коди, використовуються для виявлення помилок при передачі або зберіганні даних. Вони простіші за хеш-функції, але менш стійкі до зловмисних змін;

- цифрові підписи використовують асиметричну криптографію для забезпечення автентичності та цілісності даних. Вони створюються за допомогою приватного ключа і перевіряються відкритим ключем, що робить їх ефективними для захисту від підробки;

- регулярне створення резервних копій дозволяє відновити дані у разі їхньої втрати або пошкодження;

- обмеження доступу до даних лише для авторизованих користувачів запобігає несанкціонованим змінам. Це може включати паролі, біометричну автентифікацію або рольовий доступ;

- перевірка даних на відповідність заданим правилам (формат, тип, діапазон значень) під час введення або обробки. Наприклад, система може відхиляти введення тексту в поле для чисел;

- збереження журналів усіх змін даних дозволяє відстежувати, хто і коли вносив зміни. Це допомагає виявити несанкціоновані дії або помилки;

- обмеження, такі як первинні ключі або зовнішні ключі, забезпечують логічну цілісність даних у базах даних. Наприклад, первинний ключ гарантує унікальність кожного запису;

- регулярне оцінювання якості даних, видалення дублікатів і виправлення неточностей забезпечує їхню узгодженість [7].

1.2 Криптографічні хеш-функції

1.2.1 Класифікація криптографічних хеш-функцій

Криптографічні хеш-функції є основним інструментом для забезпечення цілісності даних у сучасних інформаційних системах. Вони перетворюють вхідні дані будь-якого розміру в вихідне значення фіксованої довжини, яке називається хешем.

Ці функції мають кілька ключових властивостей:

- одностороність: неможливо відновити вхідні дані з хеша;

- детермінованість: однакові дані завжди дають однаковий хеш;

- чутливість до змін: навіть мінімальна зміна даних призводить до зовсім іншого хеша.

Класифікація криптографічних хеш-функцій здійснюється за кількома критеріями:

а) математична структура визначає, як хеш-функція обробляє вхідні дані та генерує хеш. Основні типи конструкцій включають:

1) конструкція Меркле-Дамгард є основою для багатьох популярних хеш-функцій, таких як MD5, SHA-1 і сімейство SHA-2. Вона працює шляхом розбиття вхідних даних на блоки фіксованої довжини, обробки кожного блоку за допомогою односторонньої функції стиснення та оновлення внутрішнього стану. Після обробки всіх блоків додається фіналізація для створення остаточного хеша. Перевагою цієї конструкції є її простота, але вона вразлива до атак на основі розширення довжини;

2) конструкція HAIFA (Hash Iterative Framework) використовується в деяких сучасних хеш-функціях, зокрема в SHA-3. HAIFA є вдосконаленням конструкції Меркле-Дамгард, яке додає додатковий параметр, такий як лічильник блоків для підвищення безпеки та запобігання певним типам атак, зокрема атакам на основі розширення довжини. Ця конструкція забезпечує більшу гнучкість і безпеку;

3) конструкція губки використовується в SHA-3, яка базується на алгоритмі Keccak. У цій конструкції вхідні дані "вбираються" в стан фіксованої довжини, після чого частина стану "вичавлюється" для створення хеша. Конструкція губки є універсальною, оскільки може використовуватися не лише для хешування, а й для інших криптографічних примітивів, таких як генерація псевдовипадкових послідовностей або шифрування;

б) криптографічні хеш-функції повинні відповідати трьом основним властивостям безпеки, які є критично важливими для забезпечення цілісності даних:

1) стійкість до зворотного відображення: для заданого хеш-значення h має бути обчислювально неможливо знайти будь-яке повідомлення m, для якого hash(m) = h. Ця властивість захищає від спроб відновлення вихідних даних із хеша;

2) стійкість до другого зворотного відображення: для заданого повідомлення m1 має бути обчислювально неможливо знайти інше повідомлення m2 ≠ m1, для якого hash(m1) = hash(m2). Ця властивість запобігає заміні одного повідомлення іншим із тим самим хешем;

3) стійкість до колізій: має бути обчислювально неможливо знайти два різних повідомлення m1 і m2, для яких hash(m1) = hash(m2). Ця властивість є найважливішою для перевірки цілісності даних, оскільки колізії можуть дозволити зловмисникам підмінити дані без виявлення;

в) розмір хеш-значення впливає на рівень безпеки та обчислювальні вимоги хеш-функції:

1) 128-бітні: наприклад, MD5. Цей розмір вважається недостатньо безпечним через можливість знаходження колізій за допомогою сучасних обчислювальних ресурсів;

2) 160-бітні: наприклад, SHA-1. Хоча SHA-1 все ще використовується в деяких системах, він вважається ненадійним для нових застосувань через виявлені атаки на колізії;

3) 256-бітні: наприклад, SHA-256 із сімейства SHA-2. Цей розмір забезпечує високий рівень безпеки і широко використовується в сучасних системах;

4) 512-бітні: наприклад, SHA-512 із сімейства SHA-2. Забезпечує ще вищий рівень безпеки, але вимагає більше обчислювальних ресурсів [8].

1.2.2 Аналіз характеристик різних хеш функцій

1.2.2.1 MD5

MD5, створений Рональдом Рівестом у 1991 році, є криптографічною хеш-функцією з 128-бітним хешем, що базується на конструкції Меркле-Дамгард. Вона обробляє дані блоками по 512 біт, виконуючи 64 операції в чотирьох раундах стиснення, і раніше широко застосовувалася для перевірки цілісності файлів і цифрових підписів. Завдяки простій структурі MD5 забезпечує високу швидкодію, що робить її ефективною для програмних і апаратних реалізацій, зокрема на FPGA, де вона займає мінімум ресурсів. Однак у 2004 році були виявлені вразливості до атак на колізії, що дозволяють створювати два різні повідомлення з однаковим хешем, через що MD5 вважається зламаним і непридатним для криптографічних задач. Незважаючи на це, її все ще використовують у некритичних сценаріях, таких як обчислення контрольних сум [9].

1.2.2.2 SHA-1

SHA-1, розроблений NIST у 1993 році, генерує 160-бітний хеш і використовує конструкцію Меркле-Дамгард, обробляючи дані блоками по 512 біт із 80 кроками стиснення. Ця функція була стандартом для таких систем, як SSL/TLS і Git, завдяки відносній простоті операцій, що забезпечує хорошу швидкодію, зокрема на FPGA, де вона ефективна. Проте в 2017 році Google продемонстрував практичну атаку на колізії, що підтвердило втрату стійкості SHA-1 до таких атак, роблячи її ненадійною для безпечних застосувань. Хоча SHA-1 швидший за сучасні алгоритми, як SHA-256, але повільніший за MD5, його використання обмежується некритичними задачами через вразливості [10].

1.2.2.3 SHA-256

SHA-256, частина сімейства SHA-2, розробленого NIST у 2001 році, створює 256-бітний хеш за допомогою конструкції Меркле-Дамгард, обробляючи дані блоками по 512 біт із 64 раундами стиснення, що включають логічні операції та модульну арифметику. Алгоритм вважається безпечним, без відомих атак на колізії чи зворотне відображення, що робить його стандартом для блокчейнів, цифрових підписів і перевірки цілісності даних. Хоча SHA-256 повільніший за MD5 і SHA-1 через складнішу структуру, його операції добре паралелізуються, що забезпечує високу ефективність на FPGA, де він є оптимальним вибором для апаратних реалізацій завдяки поєднанню безпеки та продуктивності [11].

1.2.2.4 SHA-3

SHA-3, стандартизований NIST у 2015 році на основі алгоритму Keccak, використовує конструкцію губки, що дозволяє генерувати хеші різної довжини (224, 256, 384, 512 біт), обробляючи дані через "вбирання" і "вичавлювання" стану. Ця функція вважається високобезпечною, без відомих атак, а її структура губки стійка до атак на розширення довжини, що відрізняє її від алгоритмів на базі Меркле-Дамгард. SHA-3 є альтернативою SHA-2 для майбутніх систем, хоча в програмних реалізаціях вона повільніша за SHA-256. На FPGA конструкція губки забезпечує ефективну паралелізацію, компенсуючи затримки та роблячи SHA-3 привабливим вибором для гнучких апаратних систем [12].

1.2.2.5 BLAKE2

BLAKE2, розроблений у 2012 році як вдосконалена версія фіналіста конкурсу SHA-3 BLAKE, використовує модифіковану конструкцію HAIFA і генерує хеші до 512 біт (BLAKE2b) або 256 біт (BLAKE2s). Оптимізований для швидкості та безпеки, алгоритм використовує менше раундів, ніж конкуренти, що забезпечує високу швидкодію, перевищуючи SHA-256 і SHA-3 у програмних реалізаціях. BLAKE2 вважається безпечним, без відомих атак на колізії чи зворотне відображення, а його простіший дизайн порівняно з SHA-3 робить його ефективним для апаратних реалізацій, зокрема на FPGA, де він добре паралелізується, що ідеально для високопродуктивних систем [13].

1.2.2.6 RIPEMD-160

RIPEMD-160, розроблений у 1996 році в Європі як покращена версія RIPEMD, є хеш-функцією з 160-бітним хешем, що базується на конструкції Меркле-Дамгард і обробляє дані блоками по 512 біт із двома паралельними потоками стиснення. Алгоритм вважається відносно безпечним, без практичних атак на колізії, але його 160-бітний хеш менш стійкий до атак порівняно з SHA-256 чи SHA-3, що обмежує його використання в сучасних системах. За швидкодією RIPEMD-160 перевищує SHA-256, але поступається MD5, а на FPGA його паралельні потоки забезпечують ефективність, хоча менша популярність знижує його практичну значущість [14].

1.2.2.7 Whirlpool

Whirlpool, створений у 2000 році, генерує 512-бітний хеш і використовує конструкцію Меркле-Дамгард із дизайном, подібним до AES, виконуючи 10 раундів стиснення для обробки даних блоками по 512 біт. Алгоритм вважається безпечним, без відомих практичних атак, а його великий хеш забезпечує високий рівень захисту, що робить його придатним для застосувань із високими вимогами до безпеки. Однак через складну структуру Whirlpool повільніший за SHA-256 і BLAKE2 у програмних реалізаціях, але на FPGA його операції можна паралелізувати, що частково компенсує затримки, хоча він рідше використовується порівняно з популярнішими алгоритмами

1.2.3 Порівняльна характеристика алгоритмів за швидкодією та стійкістю

Орієнтовні дані швидкодії та стійкості алгоритмів для програмної реалізації на CPU Intel Core i5-6600 наведені на таблиці 1.1 [13, 16-17].

Таблиця 1.1 - Порівняння хеш-функцій за швидкодією та стійкістю

Алгоритм	Розмір хеша, біт	Стійкість до колізій	Швидкодія на CPU, МБ/с
MD5	128	Зламаний	632
SHA-1	160	Зламаний	909
SHA-256	256	Безпечний	413
SHA-3	256	Безпечний	367
SHA-3	512	Безпечний	198
BLAKE2	256	Безпечний	947
BLAKE2	512	Безпечний	648

1.3 Огляд існуючих рішень перевірки цілісності даних

1.3.1 Програмні рішення на CPU

Програмні рішення на центральних процесорах є найпоширенішим підходом до перевірки цілісності даних завдяки їхній доступності, гнучкості та простоті впровадження. Ці рішення використовують програмне забезпечення, яке виконується на універсальних процесорах, для реалізації криптографічних хеш-функцій, контрольних сум, цифрових підписів та інших методів, що дозволяють виявляти зміни в даних. Вони можуть бути реалізовані у вигляді окремих утиліт, бібліотек або вбудованих функцій операційних систем. Основні характеристики включають:

- гнучкість: програмне забезпечення легко адаптується до різних алгоритмів і форматів даних. Наприклад, бібліотека OpenSSL підтримує десятки хеш-функцій;

- доступність: більшість інструментів є безкоштовними або вбудованими в операційні системи;

- простота оновлення: оновлення програмного забезпечення дозволяє швидко додавати нові алгоритми або виправляти вразливості;

- універсальність: підходять для широкого спектру застосувань, від перевірки файлів до захисту мережевих протоколів.

Нижче наведено огляд основних програмних рішень на CPU для перевірки цілісності даних, які використовуються в різних галузях.

1.3.1.1 OpenSSL

OpenSSL — це відкрита криптографічна бібліотека, яка забезпечує підтримку численних хеш-функцій і цифрових підписів, широко застосовуючись у мережевих протоколах, перевірці файлів і шифруванні. Вона дозволяє генерувати хеші для файлів, верифікувати цифрові підписи та гарантувати цілісність даних у реальному часі, що робить її універсальним інструментом для різноманітних застосувань. Завдяки високій гнучкості, підтримці багатьох алгоритмів і кросплатформності OpenSSL є популярним вибором для розробників. Однак обробка великих обсягів даних потребує значних обчислювальних ресурсів, а вразливість до програмних атак може створювати ризики для безпеки [18].

1.3.1.2 Вбудовані утиліти операційних систем

Більшість операційних систем включають вбудовані утиліти для обчислення та перевірки хешів, такі як md5sum, sha256sum і sha3sum у Linux або CertUtil у Windows, які доступні через командний рядок. Ці інструменти призначені для генерації хешів файлів і перевірки їхньої цілісності, що робить їх зручними для базових задач. Їхні переваги включають безкоштовність, тісну інтеграцію з операційною системою та простоту використання, що ідеально для швидких перевірок. Проте підтримка обмеженого набору алгоритмів і низька швидкість обробки великих даних роблять ці утиліти менш ефективними для складних сценаріїв [19].

1.3.1.3 VeraCrypt

VeraCrypt — це програмне забезпечення для шифрування дисків, яке використовує хеш-функції для забезпечення цілісності зашифрованих даних, перевіряючи їх під час доступу до зашифрованих томів, щоб запобігти несанкціонованим змінам. Воно інтегрує перевірку цілісності з потужними механізмами шифрування, підтримуючи безпечні алгоритми, що робить його ефективним для захисту конфіденційної інформації. Серед переваг — надійність і сумісність із сучасними стандартами безпеки. Однак VeraCrypt створює високе навантаження на CPU, що знижує продуктивність, а його складність може бути надмірною для некритичних задач [20].

1.3.1.4 Python Cryptography Library

Бібліотека cryptography для Python надає інструменти для створення власних програм перевірки цілісності, дозволяючи розробникам інтегрувати хеш-функції в додатки для перевірки файлів, мережевих даних чи інших об’єктів. Вона підтримує сучасні криптографічні алгоритми, що забезпечує гнучкість і можливість адаптації до специфічних потреб проєктів. Перевагою є її придатність для кастомізованих рішень і висока безпека. Однак використання бібліотеки вимагає знань програмування, а її швидкодія поступається спеціалізованим утилітам, що обмежує її ефективність для великих обсягів даних [21].

1.3.1.5 WinRAR/7-Zip

Архіватори WinRAR і 7-Zip використовують контрольні суми або хеш-функції для перевірки цілісності файлів в архівах, що дозволяє виявляти пошкодження після передачі чи зберігання. Ці інструменти інтегрують перевірку цілісності з процесом архівації, що робить їх простими у використанні. Їхня доступність і легкість застосування є ключовими перевагами. Однак обмежена підтримка криптографічних алгоритмів і орієнтація на базові сценарії роблять їх менш придатними для складних криптографічних застосувань [22].

1.3.2 Апаратні рішення на ASIC

Апаратні рішення на основі спеціалізованих інтегральних схем є високоефективним підходом до перевірки цілісності даних, особливо в системах,

де потрібна максимальна швидкодія та енергоефективність. ASIC — це мікросхеми, які проектуються та виготовляються для виконання однієї або кількох специфічних функцій з максимальною ефективністю. Для перевірки цілісності даних ASIC зазвичай реалізують криптографічні алгоритми, такі як хеш-функції або алгоритми шифрування, які забезпечують швидке виявлення змін у даних. Основні характеристики включають:

- висока швидкодія: ASIC оптимізовані для конкретного алгоритму, що дозволяє обробляти дані значно швидше, ніж CPU чи FPGA;

- енергоефективність: завдяки спеціалізованій архітектурі ASIC споживають менше енергії порівняно з іншими платформами при виконанні тих самих завдань;

- стійкість до програмних атак: оскільки логіка ASIC зашита в апаратне забезпечення, вони менш вразливі порівняно з програмними рішеннями [23].

Нижче наведено огляд типових застосувань і прикладів ASIC для перевірки цілісності даних.

1.3.2.1 ASIC для майнінгу криптовалют

Спеціалізовані інтегральні схеми, такі як Bitmain Antminer, розроблені для високоефективного обчислення хеш-функції SHA-256, відіграють ключову роль у блокчейн-системах, зокрема в Bitcoin, забезпечуючи перевірку цілісності транзакцій і генерацію нових блоків. Вони обчислюють хеші транзакцій, гарантуючи їхню незмінність і верифікацію в розподілених мережах. Завдяки оптимізації під SHA-256 ці ASIC демонструють неперевершену швидкість (до 110 TH/s) і енергоефективність, що робить їх ідеальними для майнінгу. Однак їхня функціональність обмежена виключно SHA-256, а відсутність можливості адаптації до інших алгоритмів значно знижує їхню гнучкість, що є суттєвим недоліком для універсальних застосувань [24].

1.3.2.2 Модулі апаратної безпеки (HSM)

Модулі апаратної безпеки (HSM), такі як Thales Luna HSM і YubiHSM, використовують ASIC для виконання криптографічних операцій, зокрема хеш-функцій і цифрових підписів, забезпечуючи захист даних у банківських системах, хмарних сервісах і системах управління ключами. Вони гарантують цілісність

ключів, сертифікатів і даних, запобігаючи несанкціонованим змінам, що робить їх незамінними для високобезпечних застосувань. HSM вирізняються високим рівнем безпеки та підтримкою кількох алгоритмів, що забезпечує їхню універсальність. Проте висока вартість розробки й виробництва, а також обмежена гнучкість порівняно з FPGA роблять їх менш придатними для швидкої адаптації до нових вимог [25].

1.3.2.3 Мережеві процесори з ASIC

Мережеві пристрої, такі як маршрутизатори Cisco чи Juniper, оснащені ASIC, призначеними для прискорення обробки пакетів і перевірки цілісності даних у протоколах безпеки. Ці ASIC виконують хеш-функції в реальному часі, забезпечуючи захист даних під час передачі в мережах із високим трафіком. Їхня висока пропускна здатність і глибока інтеграція з мережевими функціями роблять їх ефективними для телекомунікаційних систем. Однак спеціалізація на мережевих задачах і висока ціна обмежують їхнє використання в інших сферах, а також ускладнюють адаптацію до немережевих застосувань [26].

1.3.2.4 Trusted Platform Module (TPM)

Trusted Platform Module (TPM) — це компактні ASIC, інтегровані в комп’ютери та сервери для забезпечення апаратної безпеки, зокрема перевірки цілісності прошивки та системних конфігурацій за допомогою хеш-функцій SHA-256 або SHA-1. TPM генерують хеші для порівняння завантажувальних файлів і конфігурацій, що дозволяє виявляти несанкціоновані зміни на ранніх етапах роботи системи. Їхня компактність і тісна інтеграція з системами безпеки є ключовими перевагами, особливо для захисту ПК і серверів. Однак підтримка обмеженого набору алгоритмів і низька гнучкість порівняно з FPGA чи програмними рішеннями роблять TPM менш універсальними для складних криптографічних задач [27].

1.3.3 Реалізації на FPGA

Програмовані логічні інтегральні схеми є потужною платформою для реалізації систем перевірки цілісності даних завдяки їхній здатності поєднувати високу швидкодію, гнучкість і можливість апаратного прискорення

криптографічних алгоритмів. FPGA є програмованими апаратними платформами, які складаються з логічних блоків, пам’яті та інтерфейсів вводу-виводу, що можуть бути налаштовані для виконання специфічних завдань. Для перевірки цілісності даних FPGA зазвичай реалізують криптографічні хеш-функції, коди виявлення помилок або цифрові підписи. Основні характеристики включають:

- висока швидкодія: FPGA забезпечують апаратне прискорення, що дозволяє обробляти дані зі швидкістю до кількох гігабайт за секунду для таких алгоритмів, як SHA-256;

- гнучкість: на відміну від ASIC, FPGA можна перепрограмувати для реалізації різних алгоритмів або оновлення застарілих функцій;

- паралелізм: FPGA підтримують паралельну обробку, що значно підвищує продуктивність для криптографічних операцій;

- модульність: FPGA дозволяють інтегрувати кілька функцій в одному дизайні [28].

Нижче наведено огляд типових застосувань і прикладів реалізацій на FPGA для перевірки цілісності даних.

1.3.3.1 FPGA для телекомунікаційних систем

У телекомунікаційних системах, таких як маршрутизатори, комутатори та базові станції, FPGA застосовуються для перевірки цілісності даних у реальному часі за допомогою криптографічних хеш-функцій. Вони забезпечують верифікацію пакетів даних у протоколах безпеки, таких як IPsec і TLS, захищаючи інформацію від змін під час передачі через мережі. FPGA вирізняються високою пропускною здатністю, що дозволяє обробляти великі обсяги мережевого трафіку, а також підтримкою кількох алгоритмів, що забезпечує їхню універсальність. Однак розробка мережевих архітектур для FPGA є складним процесом, що вимагає значних зусиль і спеціалізованих знань, що є основним недоліком.

1.3.3.2 FPGA у хмарних сховищах

FPGA знаходять застосування в дата-центрах для забезпечення цілісності великих обсягів даних, що зберігаються або передаються в хмарних сховищах, шляхом виконання хеш-функцій, таких як SHA-256 або BLAKE2. Вони генерують

хеші файлів, дозволяючи виявляти будь-які зміни та забезпечуючи надійність хмарних систем. Завдяки здатності ефективно обробляти великі дані та інтегруватися з хмарними платформами, FPGA є цінним рішенням для таких застосувань. Проте висока вартість FPGA порівняно з іншими платформами, такими як ASIC, обмежує їхнє масове використання в хмарних інфраструктурах, що є значним недоліком.

1.3.3.3 FPGA у блокчейн-системах

У блокчейн-системах FPGA використовуються як альтернатива ASIC для обчислення хеш-функцій, наприклад SHA-256 у Bitcoin або Keccak в Ethereum, забезпечуючи перевірку цілісності транзакцій і блоків шляхом генерації хешів для їхньої верифікації. Гнучкість FPGA дозволяє адаптувати їх до різних блокчейн-алгоритмів, а нижча вартість порівняно з ASIC робить їх привабливими для створення прототипів і розробки. Однак FPGA поступаються ASIC за швидкістю обчислень, особливо для SHA-256, що обмежує їхню ефективність у висококонкурентних сценаріях, таких як майнінг криптовалют.

1.3.4 Порівняльний аналіз підходів

Переваги програмних рішень на CPU:

- програмне забезпечення легко адаптується до нових алгоритмів шляхом оновлення;

- більшість інструментів є безкоштовними або відкритими, що робить їх доступними для всіх користувачів;

- бібліотеки, такі як OpenSSL або Python Cryptography, дозволяють інтегрувати перевірку цілісності в різні системи, від веб-додатків до мережевих протоколів;

- інструменти, такі як OpenSSL, підтримують десятки хеш-функцій, що дозволяє вибирати алгоритм залежно від вимог безпеки чи швидкості;

- більшість рішень працюють на різних ОС, що забезпечує універсальність.

Недоліки програмних рішень на CPU:

- програмні реалізації на CPU значно повільніші за апаратні, особливо для великих обсягів даних;

- обчислення хешів для великих файлів може навантажувати CPU, що впливає на продуктивність системи;

- програмне забезпечення може бути скомпрометоване через шкідливе програмне забезпечення, що загрожує безпеці перевірки;

- програмні рішення залежать від операційної системи та бібліотек, що може викликати проблеми сумісності або вразливості;

- CPU мають обмежену кількість ядер, що знижує ефективність паралельної обробки порівняно з FPGA.

Переваги апаратних рішень на ASIC:

- ASIC можуть обробляти хеш-функції з пропускною здатністю, що значно перевищує CPU і FPGA;

- ASIC споживають менше енергії на одиницю обчислень, що критично для вбудованих систем і дата-центрів;

- логіка ASIC зашита в апаратне забезпечення, що знижує ризик компрометації;

- ASIC проектуються для одного алгоритму, що забезпечує максимальну ефективність.

Недоліки апаратних рішень на ASIC:

- проектування та виготовлення ASIC коштують мільйони доларів, що робить їх економічно виправданими лише для масового виробництва;

- ASIC не можуть бути перепрограмовані для виконання інших алгоритмів, що обмежує їх адаптивність порівняно з FPGA;

- якщо алгоритм стає вразливим, ASIC потрібно фізично замінити, тоді як FPGA можна перепрограмувати;

- від проектування до виробництва ASIC минають місяці або роки, що ускладнює швидке реагування на нові вимоги;

- через високу вартість ASIC рідко використовуються в прототипах чи малих системах, де FPGA є кращим вибором.

Переваги реалізацій на FPGA:

- FPGA забезпечують пропускну здатність до кількох гігабайт за секунду для хеш-функцій, значно перевищуючи CPU;

- FPGA можна перепрограмувати для реалізації різних алгоритмів або оновлення застарілих функцій, на відміну від ASIC;

- FPGA дозволяють паралельно виконувати кілька обчислень, що ідеально для криптографічних операцій;

- FPGA дозволяють створювати прототипи за тижні, тоді як ASIC потребують місяців або років;

- FPGA можуть поєднувати хешування, шифрування та обробку даних в одному дизайні, що корисно для комплексних систем.

Недоліки реалізацій на FPGA

- FPGA споживають більше енергії порівняно з ASIC, що може бути проблемою для енергоефективних систем;

- проектування FPGA вимагає знань мов опису апаратного забезпечення і спеціалізованого ПЗ;

- для специфічних алгоритмів, таких як SHA-256, ASIC досягають значно вищої продуктивності;

- FPGA мають обмежену кількість логічних елементів, що може ускладнити реалізацію складних систем.

1.4 Висновки до розділу 1

Аналіз предметної області, проведений у рамках розробки FPGA-системи для перевірки цілісності даних, дозволив систематизувати знання про цілісність даних, криптографічні хеш-функції та платформи їхньої реалізації. Цілісність даних є ключовою для забезпечення надійності та безпеки інформації, а її порушення через збої чи атаки може призвести до серйозних наслідків. Криптографічні хеш-функції

забезпечують перевірку цілісності завдяки односторонності, детермінованості та чутливості до змін.

Серед хеш-функцій MD5, SHA-1, SHA-256, SHA-3 і BLAKE2 були проаналізовані. MD5 і SHA-1 є застарілими через вразливість до колізій, але MD5 вибрано для системи через простоту реалізації та високу швидкість, що особливо важливо для FPGA, де потрібна ефективна апаратна оптимізація. SHA-256, SHA-3 і BLAKE2 забезпечують вищий рівень безпеки, але їхня реалізація складніша. За швидкодією на CPU BLAKE2 (947 МБ/с) і SHA-1 (909 МБ/с) лідирують, однак на FPGA MD5 і SHA-256 оптимізуються краще завдяки простішій структурі. SHA-3 виграє за стійкістю завдяки конструкції губки, але потребує більше ресурсів.

Програмні рішення (OpenSSL, Python Cryptography) гнучкі, але повільні та вразливі. ASIC (Bitmain Antminer, HSM) забезпечують максимальну швидкість, але дорогі й негнучкі. FPGA балансують швидкодію, гнучкість і швидкість розробки, що робить їх оптимальними для системи, де MD5 забезпечує достатню швидкість і простоту при реалізації. Хоча FPGA поступаються ASIC за енергоефективністю та вимагають спеціалізованих знань, вибір MD5 дозволяє мінімізувати складність апаратного дизайну, зберігаючи ефективність.

Таким чином, аналіз підтверджує актуальність теми та обґрунтовує вибір MD5 на FPGA як оптимального рішення для системи, що вимагає простоти реалізації та високої швидкості, закладаючи основу для подальшої розробки прототипу.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Вимоги до системи

Система має відповідати чітко визначеним функціональним і нефункціональним вимогам, які забезпечують її продуктивність, гнучкість і

адаптивність до різних сценаріїв використання, таких як телекомунікації, вбудовані системи безпеки або обробка даних у хмарних сховищах.

Функціональні вимоги:

- система повинна підтримувати введення даних через стандартні інтерфейси, залежно від застосування;

- модуль введення має буферизувати вхідний потік і формувати 512-бітні блоки, необхідні для алгоритму MD5, включаючи додавання заповнення (padding) і 64-бітного поля довжини повідомлення;

- система повинна реалізувати алгоритм MD5 для обчислення 128-бітного хеш-значення з вхідних блоків даних, забезпечуючи його передачу для подальшого порівняння з еталонними значеннями;

-модуль повинен надавати 128-бітне хеш-значення через вихідну шину даних у вигляді чотирьох 32-бітних слів, сумісних із зовнішніми системами для перевірки цілісності;

- система має координувати роботу модулів через сигнали управління, забезпечуючи чітку послідовність операцій і підтримку потокової обробки багатоблокових повідомлень.

Нефункціональні вимоги:

- система повинна забезпечувати швидку обробку даних у реальному часі, що є критичним для потокових застосувань, таких як перевірка цілісності даних у мережах;

- архітектура має мінімізувати використання логічних елементів, регістрів і блоків пам’яті BRAM FPGA, враховуючи обмежену ємність вбудованої пам’яті;

- система має виявляти та обробляти позаштатні ситуації, такі як некоректні вхідні дані або переповнення буфера, сигналізуючи про помилки через вихідні сигнали;

- система має максимально використовувати можливості FPGA для паралельної обробки, наприклад, одночасного виконання операцій введення, хешування та передачі результатів, що зменшує затримки.

2.2 Загальна структурна схема системи

Архітектура системи концептуально складається з двох основних модулів: інтерфейсного модуля та модуля хешування, які разом забезпечують повний цикл обробки даних — від прийому вхідного потоку до обчислення хеш-значення. Інтерфейсний модуль відповідає за взаємодію із зовнішнім середовищем, включаючи прийом даних, їх буферизацію, форматування у 512-бітні блоки та передачу результатів хешування. Модуль хешування виконує безпосереднє обчислення хеш-значення за алгоритмом MD5, обробляючи підготовлені блоки даних і повертаючи 128-бітний хеш. Така спрощена двомодульна структура дозволяє мінімізувати складність системи, оптимізувати використання ресурсів FPGA і забезпечити високу пропускну здатність.

Інтерфейсний модуль прийматиме вхідні дані через 8-бітну адресну шину та 32-бітну шину даних, формуючи 512-бітні блоки для обробки. Він координуватиме роботу системи через сигнали управління, такі як ініціалізація, запуск обробки блоку і готовність, забезпечуючи синхронізацію з модулем хешування та зовнішніми компонентами. Модуль хешування оброблятиме блоки даних, виконуючи 64 раунди обчислень MD5, і повертатиме 128-бітне хеш-значення, яке інтерфейсний модуль передаватиме на вихід для подальшого порівняння.

Синхронізація між модулями базуватиметься на кінцевому автоматі, який керуватиме станами системи: ініціалізація, прийом даних, хешування та видача результатів. У стані ініціалізації система налаштовуватиме початкові параметри, у стані обробки — координуватиме передачу даних і обчислення хеша, а після завершення — видаватиме результати. Така організація забезпечує безперервну потокову обробку, що є оптимальним для FPGA і дозволяє обробляти великі обсяги даних у реальному часі.

Запропонована архітектура враховує переваги MD5, зокрема його просту структуру, яка потребує лише чотирьох 32-бітних регістрів стану та базових логічних операцій (AND, OR, XOR, зсуви), що сприяє компактності реалізації.

Відмова від зберігання еталонних хешів на FPGA дозволяє обробляти великі набори даних, наприклад, тисячі файлів у хмарному сховищі, без перевищення обмежень пам’яті. Загальна структурна схема системи, яка створює основу для подальшої реалізації, забезпечуючи баланс між швидкістю, простотою та гнучкістю, наведена на рисунку Б.1 в додатку Б.

2.3 Проєктування інтерфейсного модуля

Інтерфейсний модуль системи проєктується як ключовий компонент, який забезпечує взаємодію між зовнішнім середовищем і модулем хешування, відповідаючи за прийом, буферизацію, форматування вхідних даних і передачу результатів хешування. Його основна мета — забезпечити стабільний потік даних у форматі, сумісному з алгоритмом MD5, та координувати роботу системи через сигнали управління. На етапі проєктування модуль розробляється з акцентом на гнучкість, компактність і підтримку потокової обробки.

Інтерфейсний модуль прийматиме вхідні дані через 8-бітну адресну шину та 32-бітну шину даних, що дозволяє записувати 32-бітні слова в масив регістрів для формування 512-бітних блоків, необхідних для MD5. Для забезпечення сумісності з різними джерелами даних планується використовувати стандарт AXI-Stream, який є оптимальним для потокової передачі завдяки простим сигналам синхронізації. Інтерфейс також забезпечуватиме доступ до вихідного 128-бітного хеш-значення, яке повертатиметься через ту ж 32-бітну шину даних у вигляді чотирьох послідовних слів.

Буферизація даних здійснюватиметься за допомогою масиву з 16 32-бітних регістрів, що відповідає 512-бітному блоку даних для MD5. Цей масив формуватиметься апаратно, дозволяючи накопичувати вхідні дані та передавати їх у модуль хешування у правильному форматі. Для компенсації нерівномірного надходження даних планується використовувати двопортову пам’ять на основі

BRAM, що дозволить одночасно записувати та зчитувати дані, зменшуючи затримки. Механізм буферизації також включатиме логіку попередньої обробки, яка додаватиме заповнення (padding) до блоків даних, якщо їхній розмір менший за 512 біт. Згідно зі стандартом MD5, заповнення складатиметься з біта “1”, за яким ідуть нулі, та 64-бітного поля довжини повідомлення, яке додаватиметься до останнього регістра блоку.

Логіка управління модуля базуватиметься на сигналах init, next і ready, які координуватимуть взаємодію з модулем хешування та зовнішніми компонентами. Сигнал init запускатиме ініціалізацію системи, налаштовуючи початкові параметри, такі як регістри стану. Сигнал next активуватиме обробку нового 512-бітного блоку, а сигнал ready повідомлятиме про готовність системи до прийому даних або видачі результатів. Управління реалізовуватиметься через простий кінцевий автомат, який оброблятиме команди запису (we) і вибору мікросхеми (cs), забезпечуючи синхронізацію операцій. Модуль також підтримуватиме інформаційні регістри, такі як ідентифікатори системи та версії, для спрощення інтеграції з іншими компонентами.

Для оптимізації ресурсів FPGA інтерфейсний модуль мінімізуватиме кількість комбінаційної логіки, використовуючи лише необхідні операції для декодування адрес і обробки даних. Паралельна обробка, наприклад, одночасний запис даних у регістри та їх передача в модуль хешування, підвищуватиме пропускну здатність.

2.4 Проєктування модуля хешування

Модуль хешування проєктується як центральний компонент системи, відповідальний за виконання алгоритму MD5 і обчислення 128-бітного хеш-значення для 512-бітних блоків даних. Модуль розробляється з акцентом на швидкість, компактність і ефективне використання ресурсів FPGA, щоб

забезпечити високу пропускну здатність при мінімальній складності реалізації. Алгоритм MD5 обрано через його просту структуру, яка базується на базових логічних операціях (AND, OR, XOR, зсуви) і потребує лише чотирьох 32-бітних регістрів стану, що робить його ідеальним для апаратної реалізації в системах, де пріоритетними є швидкість і економія ресурсів.

Модуль хешування складатиметься з трьох основних функціональних блоків: ініціалізації, обчислень (стиснення) і фіналізації. Блок ініціалізації налаштовуватиме початкові значення чотирьох 32-бітних регістрів стану відповідно до стандарту MD5. Ці значення записуватимуться при отриманні сигналу ініціалізації від інтерфейсного модуля. Блок також прийматиме 512-бітний блок даних, розбиваючи його на 16 32-бітних слів для подальшої обробки.

Блок обчислень виконуватиме основну частину алгоритму MD5, яка включає 64 раунди стиснення, поділених на чотири етапи по 16 раундів. Кожен раунд використовуватиме нелінійну функцію, константу, зсув і слово з розкладу повідомлення. Нелінійна функція залежатиме від етапу: для перших 16 раундів — (B & C) | ((~B) & D), для наступних 16 — (D & B) | ((~D) & C), для 32–47 — B ^ C ^ D, а для останніх 16 — C ^ (B | (~D)). Константи будуть 32-бітними значеннями, визначеними стандартом MD5, а розклад повідомлення визначатиме, яке слово з блоку використовувати в кожному раунді. Зсуви виконуватимуться циклічно вліво на задану кількість біт, що додає нелінійність.

Для підвищення продуктивності модуль хешування проєктується з конвеєрною архітектурою, де різні раунди оброблятимуться паралельно. Наприклад, поки один раунд оновлює регістр стану, наступний готуватиметься до виконання, що зменшить загальну затримку обробки блоку. Регістри стану оновлюватимуться після кожного раунду, а їхні значення передаватимуться між етапами конвеєра.

Блок фіналізації комбінуватиме результати всіх раундів із початковими значеннями регістрів, додаючи їх за модулем 2^32, щоб сформувати остаточне 128-бітне хеш-значення. Це значення передаватиметься в інтерфейсний модуль у форматі, сумісному з вихідною шиною даних, із урахуванням зворотного порядку

байтів для відповідності стандарту MD5. Фіналізація виконуватиметься апаратно за один такт, що забезпечить швидке завершення обробки.

Керування модулем здійснюватиметься через кінцевий автомат із двома основними станами: очікування і обробка. У стані очікування модуль чекатиме сигналів ініціалізації або обробки нового блоку від інтерфейсного модуля. У стані обробки виконуватимуться 64 раунди, а лічильник раундів відстежуватиме прогрес. Після завершення модуль повертатиметься в стан очікування, сигналізуючи про готовність до наступного блоку через сигнал ready. Така організація забезпечить підтримку багатоблокових повідомлень, де хеш попереднього блоку використовуватиметься як початковий стан для наступного.

2.5 Висновки до розділу 2

Проєктування архітектури FPGA-системи для перевірки цілісності даних за допомогою алгоритму MD5 дозволило розробити концептуальну структуру, яка відповідає вимогам швидкості, компактності та гнучкості. Запропонована архітектура базується на двох основних модулях: інтерфейсному модулі та модулі хешування, що забезпечують повний цикл обробки даних — від прийому вхідного потоку до обчислення 128-бітного хеш-значення. Така двомодульна організація мінімізує складність системи, оптимізує використання ресурсів FPGA і підтримує потокову обробку.

Інтерфейсний модуль спроєктовано для ефективного прийому даних через стандарт AXI-Stream, буферизації їх у 512-бітні блоки та координації роботи системи за допомогою сигналів управління. Використання двопортової пам’яті на основі BRAM і логіки попередньої обробки для додавання заповнення забезпечує стабільний потік даних із мінімальними затримками. Модуль хешування, у свою чергу, реалізує алгоритм MD5 із 64 раундами стиснення, використовуючи конвеєрну архітектуру для паралельної обробки та базові логічні операції для

зменшення апаратних затрат. Його компактна структура, що потребує лише чотирьох 32-бітних регістрів стану, сприяє ефективному використанню логічних елементів і регістрів FPGA.

Архітектура системи побудована на принципах модульності, паралелізму та потокової обробки, що дозволяє досягти високої пропускної здатності при тактовій частоті 100 МГц і обробляти блоки даних із затримкою приблизно 64 такти. Відмова від зберігання еталонних хеш-значень на FPGA на користь зовнішньої пам’яті або бази даних оптимізує використання вбудованої пам’яті BRAM і підвищує масштабованість системи, дозволяючи обробляти великі набори даних, наприклад, тисячі файлів у хмарному сховищі.

Запропонована архітектура створює міцну основу для подальшої реалізації, тестування та інтеграції системи в реальні сценарії, підтверджуючи її відповідність поставленим вимогам і актуальність для сучасних задач перевірки цілісності даних.

3 РЕАЛІЗАЦІЯ СИСТЕМИ НА FPGA

3.1 Вибір FPGA-платформи

3.1.1 Критерії вибору апаратної платформи

Вибір FPGA-платформи базується на кількох ключових критеріях, які враховують технічні характеристики системи, бюджетні обмеження та зручність розробки:

- кількість логічних елементів (LEs): система потребує 2000-3000 логічних елементів для реалізації інтерфейсного модуля та модуля хешування MD5, тому FPGA має мати достатній запас логічних елементів, щоб підтримувати додаткові компоненти, такі як інтерфейси чи логіка керування;

- кількість входів/виходів (I/Os): для системи потрібно 76 пінів для реалізації AXI-Stream, адресної шини (8 біт), шини даних (32 біти) та сигналів керування.

FPGA має мати щонайменше 100–200 I/O для забезпечення гнучкості підключення до зовнішніх пристроїв;

- обсяг вбудованої пам’яті (BRAM): оскільки еталонні хеші зберігаються зовні, вбудована пам’ять потрібна лише для буферизації даних (512 біт на блок) і тимчасового зберігання регістрів стану. FPGA з 200–500 Кбіт BRAM буде достатньо;

- енергоефективність: для вбудованих систем важлива низька споживана потужність. FPGA середнього класу з технологією 40 нм або 28 нм є оптимальними;

- інструменти розробки та доступність: платформа має підтримуватися безкоштовними або доступними інструментами, такими як Quartus II для Altera або Vivado для Xilinx, із можливістю симуляції та верифікації;

- вартість: для прототипування важливо обрати недорогу FPGA, яка забезпечує достатню продуктивність без значних витрат. Платформи вартістю $40–60 є пріоритетними.

3.1.2 Порівняння FPGA від Intel (Altera) та AMD (Xilinx)

Для вибору платформи розглянуто FPGA від двох провідних виробників: Intel (Altera) та AMD (Xilinx). Обидва пропонують широкий спектр мікросхем, але їхні серії відрізняються за продуктивністю, вартістю та інструментами розробки.

Altera пропонує кілька серій FPGA, які підходять для реалізації системи, зокрема Cyclone III, Cyclone IV, Cyclone V та Arria II. Ці серії орієнтовані на бюджетні та середньопродуктивні застосування:

- Cyclone III (наприклад, EP3C10F256C6): мікросхема початкового рівня з технологією 65 нм, 10,320 LEs, 256 I/O, 414 Кбіт BRAM і максимальною частотою ~200 МГц. Вона економічна (~$30–50), але застаріла для сучасних інтерфейсів, таких як PCIe, і має обмежену енергоефективність;

- Cyclone IV (наприклад, EP4CE10F17C6): покращена серія з технологією 60 нм, 10,320 LEs, 179 I/O, 414 Кбіт BRAM і частотою до 200 МГц. Cyclone IV пропонує кращу підтримку інтерфейсів і є оптимальною для прототипування завдяки низькій вартості (~$40–60) та сумісності з Quartus II;

- Cyclone V (наприклад, 5CEBA2F17C6): сучасніша серія з технологією 28 нм, 25,000 LEs, 224 I/O, 1,400 Кбіт BRAM і частотою до 400 МГц. Вона енергоефективніша, але дорожча (~$80–100), що може бути надмірним для системи;

- Arria II (наприклад, EP2AGX45DF25C6): FPGA середнього класу з технологією 40 нм, 43,000 LEs, 364 I/O, 3,400 Кбіт BRAM і частотою до 500 МГц. Arria II підходить для високопродуктивних систем, але її висока вартість (~$200+) робить її менш виправданою для даної задачі.

Xilinx пропонує серії Spartan-6, Artix-7 і Kintex-7, які є аналогами серій Altera за продуктивністю та вартістю:

- Spartan-6 (наприклад, XC6SLX9): бюджетна серія з технологією 45 нм, 9,152 LEs, 200 I/O, 576 Кбіт BRAM і частотою до 250 МГц. Вона порівнянна з Cyclone IV за вартістю (~$30–50), але потребує інструменту Vivado, який менш зручний для старих мікросхем;

- Artix-7 (наприклад, XC7A15T): сучасна серія з технологією 28 нм, 15,850 LEs, 250 I/O, 900 Кбіт BRAM і частотою до 450 МГц. Artix-7 енергоефективніша і підтримує PCIe, але коштує дорожче (~$70–90);

- Kintex-7 (наприклад, XC7K70T): високопродуктивна серія з 65,600 LEs, 300 I/O, 4,860 Кбіт BRAM і частотою до 600 МГц. Вона значно перевищує потреби системи і є дорогою (~$150+).

3.1.3 Характеристики обраної FPGA

Для реалізації системи обрано Intel Cyclone IV EP4CE10F17C6, яка оптимально відповідає критеріям за продуктивністю, вартістю та доступністю інструментів. Основні характеристики:

- логічні елементи: 10,320 LEs що забезпечує запас для додаткових компонентів;

- входи/виходи: 179 I/O, достатньо для AXI-Stream, адресної шини, шини даних і сигналів керування;

- вбудована пам’ять: 414 Кбіт BRAM, що відповідає потребам системи;

- технологія: 60 нм, забезпечує баланс між вартістю та продуктивністю;

- енергоефективність: споживана потужність ~100–200 мВт для даної реалізації, що є прийнятним для вбудованих систем;

- вартість: ~$40–50, що робить мікросхему економічно вигідною для прототипування;

3.1.4 Системи розробки та симуляції

Для розробки системи обрано Altera Quartus II 13.1 (Web Edition), яка є безкоштовною і повністю підтримує Cyclone IV. Quartus II пропонує інтуїтивний інтерфейс для синтезу, розміщення, маршрутизації та генерації бітового потоку, а також включає інструменти для аналізу ресурсів і таймінгу. Версія 13.1 є стабільною для бюджетних FPGA, таких як Cyclone IV, і не потребує додаткових ліцензій, що робить її ідеальною для прототипування. Альтернативою для Xilinx є Vivado, але він складніший для старих серій і менш зручний для малих проєктів.

Для симуляції обрано Aldec Active-HDL 9.1, яка забезпечує потужне середовище для верифікації VHDL/Verilog-коду. Active-HDL підтримує змішану симуляцію, аналіз покриття коду та інтеграцію з Quartus II, що дозволяє ефективно тестувати модулі системи перед синтезом. Перевагою Active-HDL є підтримка детальних тестових сценаріїв, що важливо для перевірки коректності алгоритму MD5 і сигналів керування. Альтернативи, такі як ModelSim (вбудований у Quartus II), менш гнучкі для складних симуляцій.

3.2 Створення дизайну проєкту

На початку роботи створюємо новий проєкт у середовищі Quartus II 13.1. Для цього запускаємо Quartus II та в головному меню обираємо File > New Project Wizard. Це відкриває майстер створення проєкту, де задаємо робочу директорію і назву проєкту (рис. 3.1).

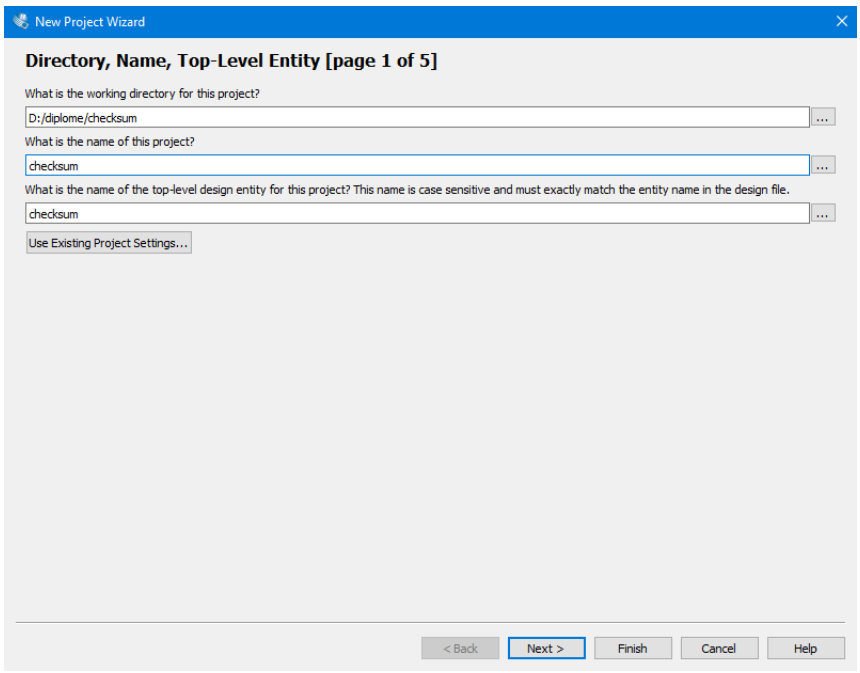


Рисунок 3.1 – Створення нового проєкту в Quartus II

Середовище Quartus II вимагає визначення апаратної платформи. Для плати Intel Cyclone 4 обираємо пристрій EP4CE10F17C6, яке підтримує достатню кількість логічних елементів для реалізації системи. Вибір здійснюється у вікні майстра створення проєкту, де в полі Family задаємо Cyclone 4, а в списку Available Devices знаходимо потрібну модель (рис. 3.2).

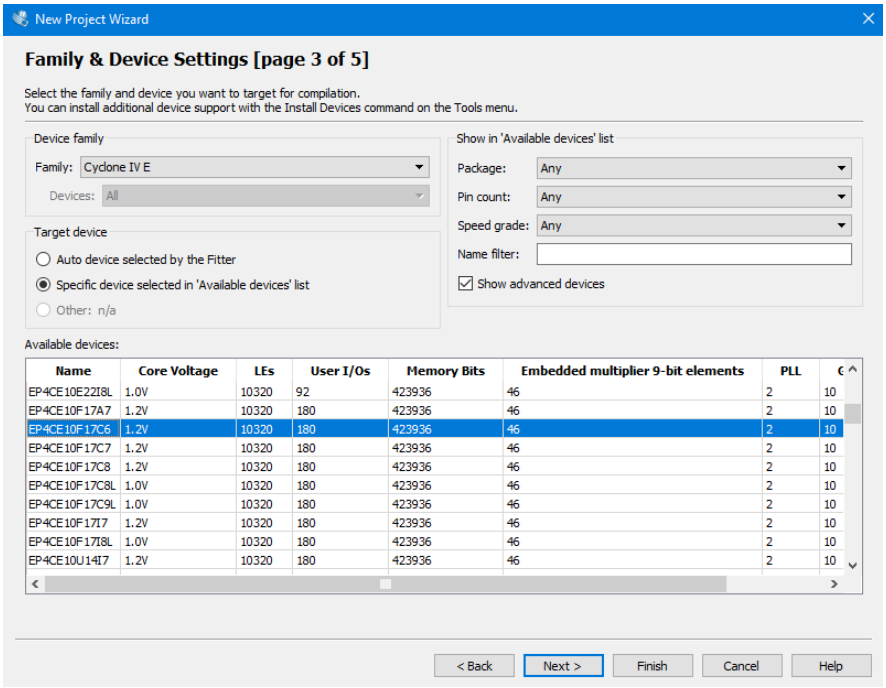


Рисунок 3.2 – Вибір платформи Cyclone 4

Натискаємо Finish, щоб завершити створення проєкту.

Для реалізації системи створюємо два Verilog-файли, обравши File > New > Verilog HDL File, і називаємо їх md5.v та md5_core.v (рис. 3.3).

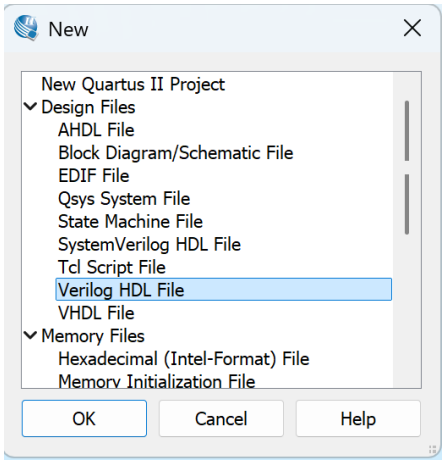


Рисунок 3.3 – Створення файлів

У Project Navigator клацнемо правою кнопкою миші на md5.v і обираємо Set as Top-Level Entity, щоб визначити його як верхній модуль (рис 3.4).

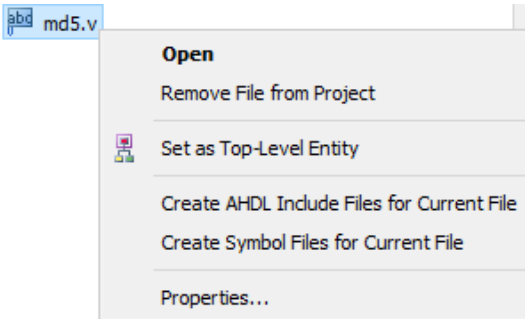


Рисунок 3.4 – Визначення файлу в якості верхнього модуля

3.3 Реалізація модулів на HDL

3.3.1 Вибір мови опису

Для апаратного опису системи розглянуто дві основні мови: Verilog і VHDL. Обидві мови широко використовуються для проєктування цифрових систем на

FPGA, але мають різні особливості. VHDL характеризується строгим синтаксисом, що забезпечує чітку структуризацію коду та високу надійність опису, але вимагає більше часу на написання через деталізовану типізацію. Verilog, навпаки, має більш лаконічний синтаксис, що спрощує розробку і дозволяє швидше описувати апаратні блоки, хоча це може ускладнювати масштабування великих систем через менш строгий контроль типів.

Для реалізації системи обрано Verilog через його простоту, швидкість написання коду та достатню гнучкість для компактної системи з двома модулями. Verilog добре інтегрується з інструментами Quartus II 13.1 і Aldec Active-HDL 9.1, що використовуються для синтезу та симуляції, а також підтримує ефективне описання конвеєрних і комбінаційних структур, необхідних для алгоритму MD5. Вибір Verilog також обґрунтовано його широким використанням у бюджетних FPGA-проєктах, таких як реалізація на Cyclone IV EP4CE10F17C6, що сприяє легшій інтеграції з наявними бібліотеками та прикладами.

3.3.2 Модуль хешування

Модуль хешування реалізує алгоритм MD5, виконуючи обчислення 128-бітного хеш-значення для 512-бітних блоків даних. Його принцип роботи базується на конвеєрній обробці 64 раундів стиснення, що забезпечує високу пропускну здатність при мінімальному використанні ресурсів FPGA. Модуль спроєктовано для інтеграції з інтерфейсним модулем, від якого він отримує підготовлені блоки даних і сигнали керування, повертаючи обчислене хеш-значення.

Модуль приймає 512-бітний блок даних і сигнали керування (init для ініціалізації, next для обробки нового блоку). У стані ініціалізації чотири 32-бітні регістри стану (A, B, C, D) встановлюються в початкові значення, визначені стандартом MD5 (0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476). При отриманні сигналу next модуль розпочинає обробку блоку, розбиваючи його на 16 32-бітних слів. Алгоритм MD5 виконує 64 раунди стиснення, поділених на чотири етапи по 16 раундів, кожен із яких використовує нелінійну функцію, константу, зсув і слово з розкладу повідомлення. Нелінійні функції варіюються залежно від

етапу: (B & C) | ((~B) & D) для перших 16 раундів, (D & B) | ((~D) & C) для наступних 16, B ^ C ^ D для 32–47 і C ^ (B | (~D)) для останніх 16.

Для оптимізації продуктивності використано конвеєрну архітектуру, де кожен раунд виконується як окремий етап, дозволяючи паралельно обробляти кілька операцій. Регістри стану оновлюються після кожного раунду, а результати комбінуються з початковими значеннями за модулем 2^32 для формування остаточного хеш-значення. Модуль керується кінцевим автоматом із двома станами: очікування (idle) і обробка (next). У стані обробки лічильник раундів відстежує прогрес, а після завершення 64 раундів модуль повертається в стан очікування, встановлюючи сигнал ready для повідомлення про готовність. Хеш-значення виводиться у форматі чотирьох 32-бітних слів із урахуванням зворотного порядку байтів.

Код реалізіції модуля хешування наведено на лістингу А.1

3.3.3 Інтерфейсний модуль

Інтерфейсний модуль забезпечує взаємодію системи із зовнішнім середовищем, відповідаючи за прийом вхідних даних, їх буферизацію, передачу в модуль хешування та видачу результатів.

Модуль приймає 32-бітні слова даних через 8-бітну адресну шину та записує їх у масив із 16 32-бітних регістрів, формуючи 512-бітний блок для MD5. Сигнали керування (cs, we, init, next) координують операції: init запускає ініціалізацію, next активує обробку нового блоку, а cs і we керують записом і зчитуванням даних. Модуль підтримує AXI-Stream для потокової передачі, використовуючи сигнали ready і valid для синхронізації з зовнішніми джерелами. Буферизація даних здійснюється за допомогою двопортової пам’яті на основі BRAM, що дозволяє одночасно записувати та зчитувати дані, компенсуючи нерівномірне надходження потоку.

Логіка попередньої обробки додає заповнення до блоків даних, якщо їхній розмір менший за 512 біт, відповідно до стандарту MD5 (біт “1”, нулі та 64-бітне поле довжини повідомлення). Після обробки модуль хешування повертає 128-бітне

хеш-значення, яке інтерфейсний модуль виводить через 32-бітну шину даних у вигляді чотирьох слів, доступних за адресами вихідного регістра.

Код реалізіції інтерфейсного модуля наведено на лістингу А.2

3.4 Синтез проєкту

3.4.1 Аналіз використання ресурсів FPGA

Компіляція системи, що складається з двох основних модулів — інтерфейсного модуля та модуля хешування, виконана в Quartus II 13.1 для мікросхеми Cyclone IV EP4CE10F17C6. Звіт компіляції наведено на рисунку 3.5. Цей звіт відображає детальну статистику використання апаратних ресурсів FPGA, включаючи логічні елементи (LEs), регістри, блоки пам’яті (BRAM) і піни.

Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	checksum
Top-level Entity Name	md5
Family	Cyclone IV E
Device	EP4CE10F17C6
Timing Models	Final
Total logic elements	2,162 / 10,320 (21 %)
Total combinational functions	1,418 / 10,320 (14 %)
Dedicated logic registers	1,291 / 10,320 (13 %)
Total registers	1291
Total pins	76 / 180 (42 %)
Total virtual pins	0
Total memory bits	0 / 423,936 (0 %)
Embedded Multiplier 9-bit elements	0 / 46 (0 %)
Total PLLs	0 / 2 (0 %)

Рисунок 3.5 – Звіт з компіляції

Звіт компіляції демонструє наступні ключові характеристики використання ресурсів для модуля хешування:

- логічні елементи (LEs): використано 2162 LEs із доступних 10,320 (~21%). Це включає комбінаційну логіку для реалізації нелінійних функцій, зсувів і констант алгоритму MD5, а також логіку кінцевого автомата.

- регістри: задіяно 1291 регістри для зберігання чотирьох 32-бітних регістрів стану (A, B, C, D), лічильника раундів і проміжних значень конвеєра. Це становить ~13% від загальної кількості регістрів.

- блоки пам’яті (BRAM): використано 0 Кбіт із доступних 414 Кбіт, оскільки модуль хешування не потребує вбудованої пам’яті, а буферизація даних виконується інтерфейсним модулем.

- піни (I/Os): Задіяно 76 пінів для передачі 512-бітного блоку даних, сигналів керування (init, next, ready) і вихідного 128-бітного хеш-значення, що становить ~42% від доступних 180 I/O.

Ці показники підтверджують високу ефективність реалізації, оскільки система займає лише малу частину ресурсів Cyclone IV, залишаючи значний запас для потенційного розширення.

3.4.2 Розміщення логічних блоків і пінів на чіпі

Розміщення логічних блоків і пінів на мікросхемі Cyclone IV EP4CE10F17C6 виконано в Quartus II 13.1 за допомогою інструменту Chip Planner. Цей процес включає розподіл логічних елементів, регістрів і пінів на фізичній структурі FPGA, а також маршрутизацію з’єднань між ними для забезпечення оптимальної продуктивності та мінімальних затримок. Результати розміщення зображено на рисунку 3.6, а загальний вигляд осередку завантаження чипу — на рисунку 3.7.

На схемі розміщення показано, як компоненти модуля хешування розподілені на мікросхемі. Логічні блоки, позначені синіми прямокутниками, відповідають комбінаційній логіці та регістрам, що реалізують 64 раунди стиснення алгоритму MD5, кінцевий автомат і логіку обчислень. Ці блоки сконцентровано в центральній частині чипу, що забезпечує короткі маршрути з’єднань і знижує затримки сигналів. Бурі прямокутники позначають задіяні піни, які включають входи для 512-бітного блоку даних, сигнали керування (init, next, ready) і виходи для 128-бітного хеш-значення. Піни розподілені по периферії чипу, що відповідає фізичній структурі Cyclone IV, і згруповані для зручного підключення до зовнішніх інтерфейсів.

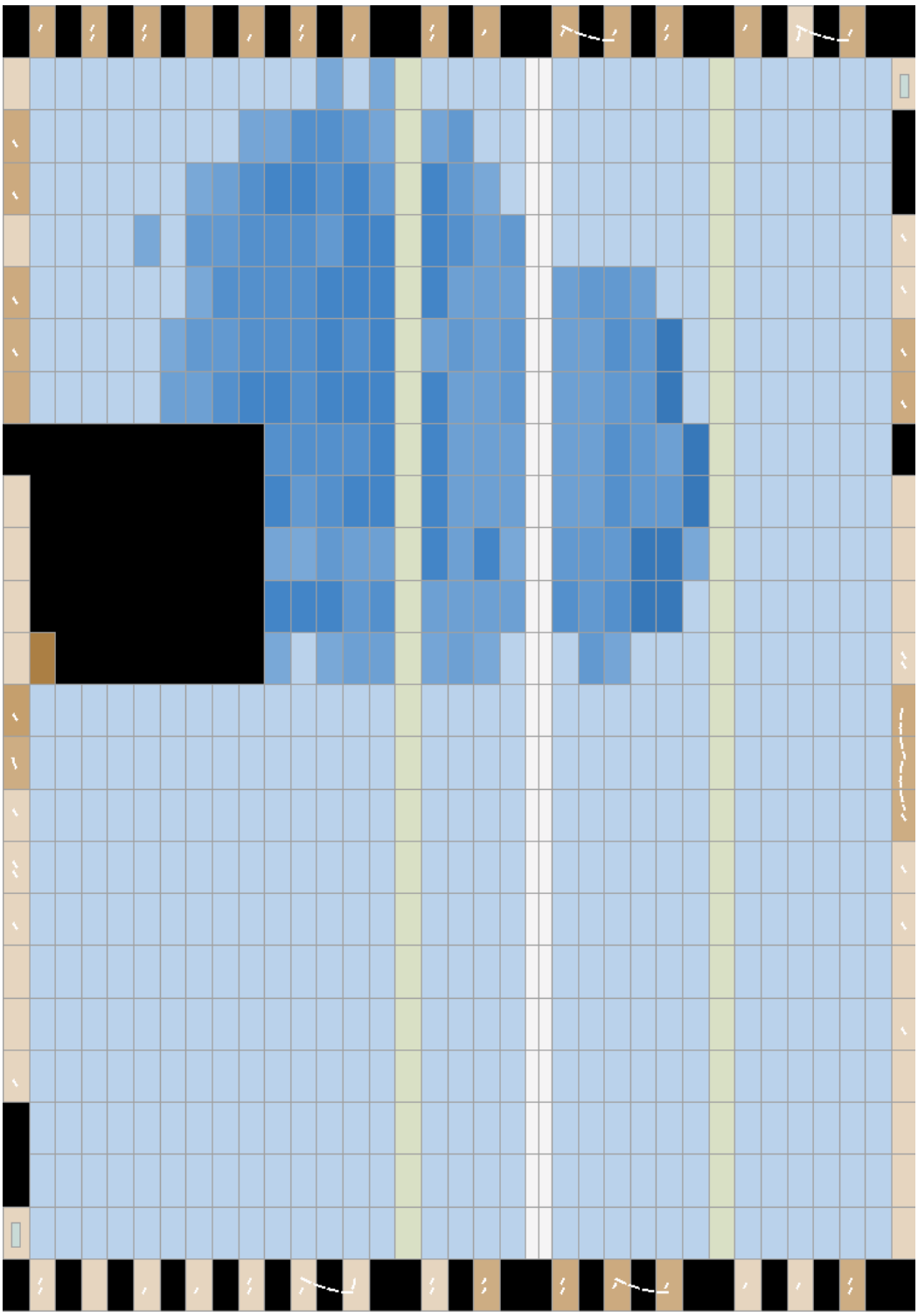


Рисунок 3.6 – Розміщення логічних блоків і пінів на чіпі

Розміщення виконано з урахуванням оптимізації таймінгу: критичні шляхи, такі як конвеєрні етапи обробки раундів, локалізовані для зменшення затримок. Загальна заповненість чипу логічними блоками становить 21%, що вказує на рівномірний розподіл ресурсів без перевантаження окремих ділянок FPGA.

Маршрутизація з’єднань між логічними блоками та пінами оптимізована для забезпечення стабільної роботи.

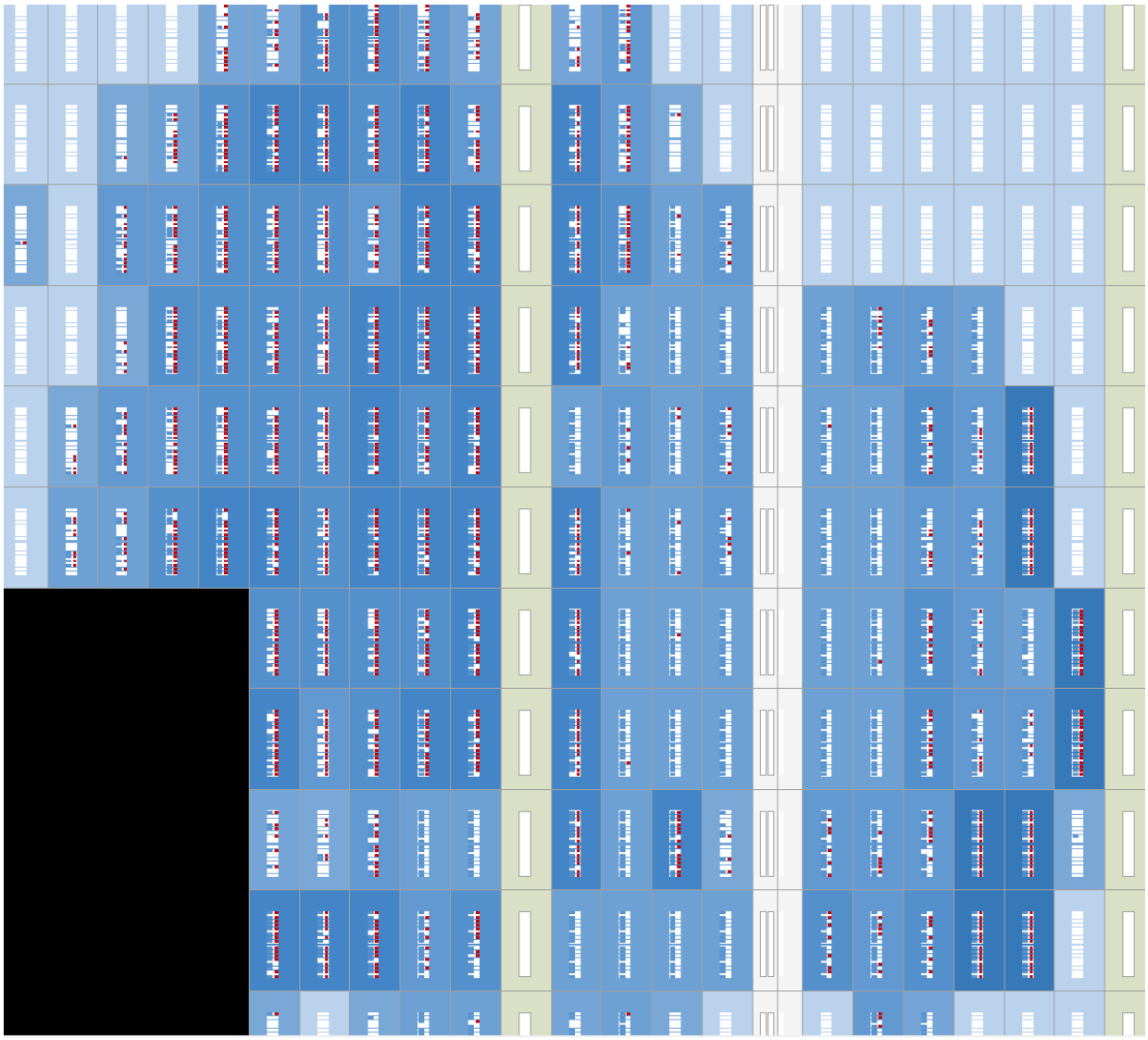


Рисунок 3.7 – Осередок загруження чипу

3.5 Тестування та верифікація системи

3.5.1 Методологія тестування

Методологія тестування системи базується на використанні тестового стенду (testbench), розробленого на Verilog для симуляції роботи системи в Aldec Active-

HDL 9.1. Тестовий стенд, детальний код якого наведено в лістингу А.3, моделює взаємодію з модулями системи через стандартний інтерфейс із 8-бітною адресною шиною, 32-бітною шиною даних і сигналами керування (cs, we, init, next, ready). Основна мета тестування — перевірити коректність обчислення 128-бітних хеш-значень для різних вхідних повідомлень, включаючи порожні дані, односимвольні повідомлення, стандартні тестові фрази та довгі послідовності.

Тестовий стенд включає чотири тестові сценарії, які охоплюють наступні випадки:

- тест 1: обчислення хешу для порожнього повідомлення (довжина 0 біт);

- тест 2: обчислення хешу для повідомлення 'a' (довжина 8 біт);

- тест 3: обчислення хешу для повідомлення 'The quick brown fox jumps over the lazy dog' (довжина 344 біти);

- тест 4: обчислення хешу для повідомлення, що складається з 66 символів 'a' (довжина 528 біт), яке потребує обробки двох блоків.

Кожен тестовий сценарій виконує наступні кроки:

- ініціалізація системи шляхом встановлення сигналу init через регістр керування (ADDR_CTRL);

- очікування готовності системи через перевірку сигналу ready у регістрі статусу (ADDR_STATUS);

- запис 512-бітного блоку даних у регістри (ADDR_BLOCK0–ADDR_BLOCK15), включаючи заповнення (padding) відповідно до стандарту MD5;

- активація обробки блоку через сигнал next;

- зчитування отриманого хеш-значення з регістрів (ADDR_DIGEST0–ADDR_DIGEST3);

- порівняння отриманого хешу з еталонним значенням і виведення результату в консоль.

Тестовий стенд автоматично підраховує кількість виконаних тестів (tc_ctr) і помилок (error_ctr), що дозволяє оцінити успішність виконання. Для апаратного тестування тестові дані подавалися через AXI-Stream на плату Cyclone IV, а

результати зчитувалися через вихідну шину даних і порівнювалися з еталонними значеннями за допомогою зовнішнього контролера.

3.5.2 Тестування системи

Тестування системи проведено шляхом виконання чотирьох тестових сценаріїв у симуляційному середовищі Aldec Active-HDL 9.1. Результати тестування проаналізовано за допомогою часових діаграм, які відображають поведінку сигналів системи під час обробки вхідних даних, а також виводів у консоль, що підтверджують коректність обчислених хеш-значень. Часові діаграми для кожного тесту наведено на рисунках 3.8–3.11, а вивід у консоль — на рисунку 3.12.

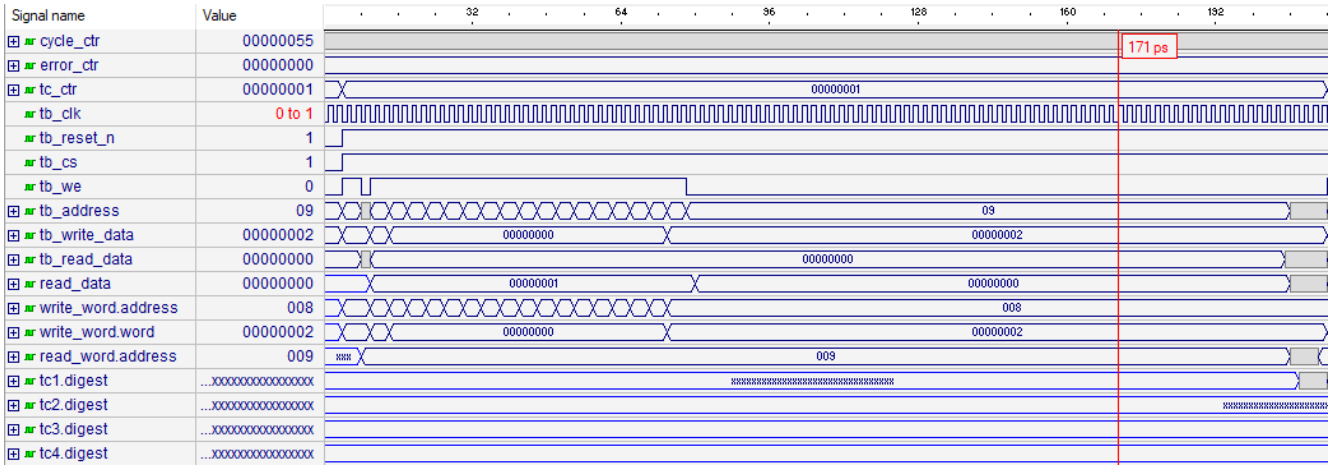


Рисунок 3.8 – Часова діаграма роботи системи для шифрування ‘’

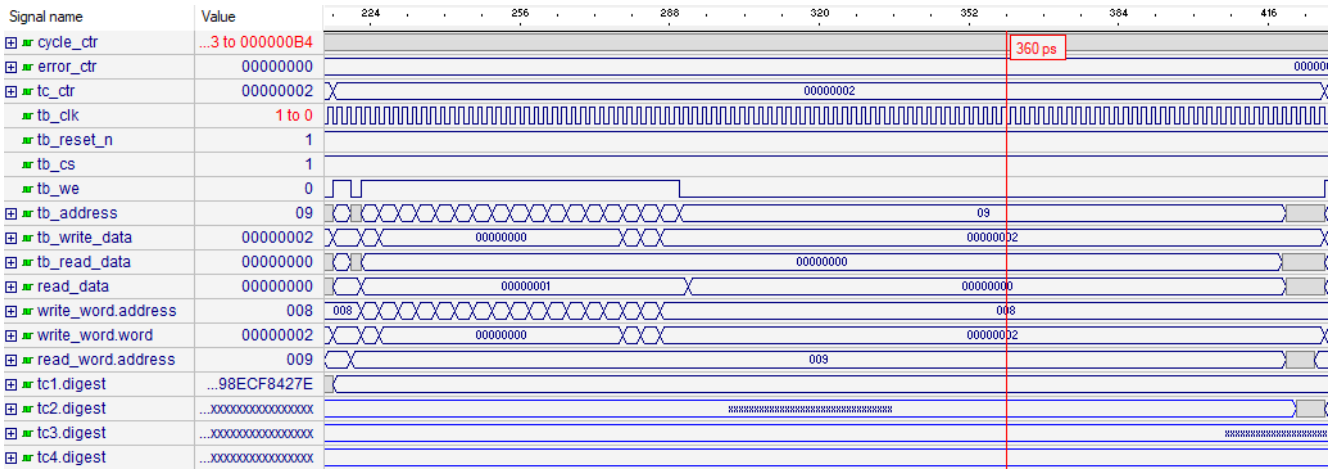


Рисунок 3.9 – Часова діаграма роботи системи для шифрування ‘a’

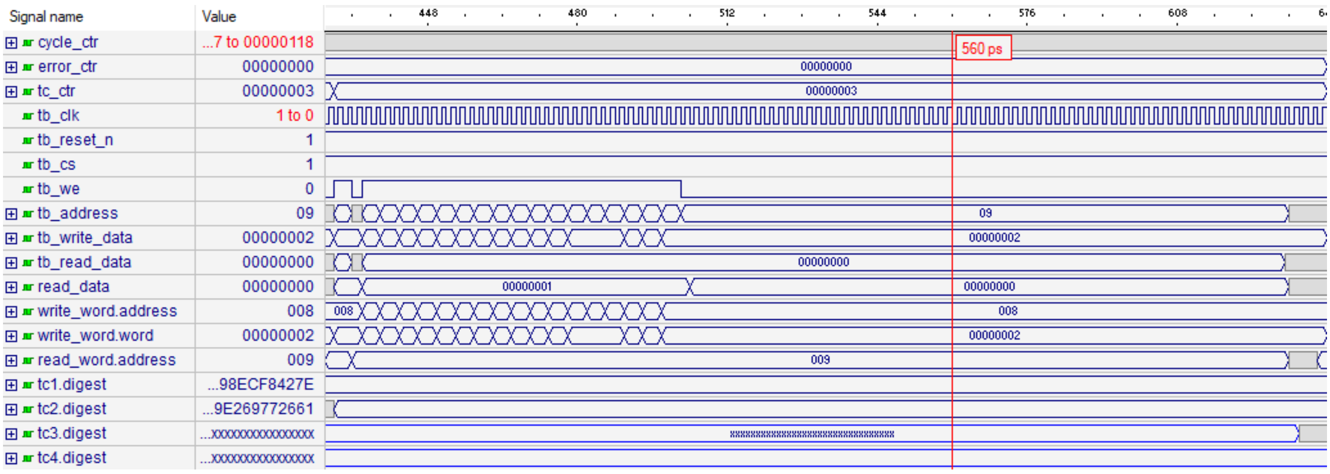


Рисунок 3.10 – Часова діаграма роботи системи для шифрування ‘The quick brown fox jumps over the lazy dog’

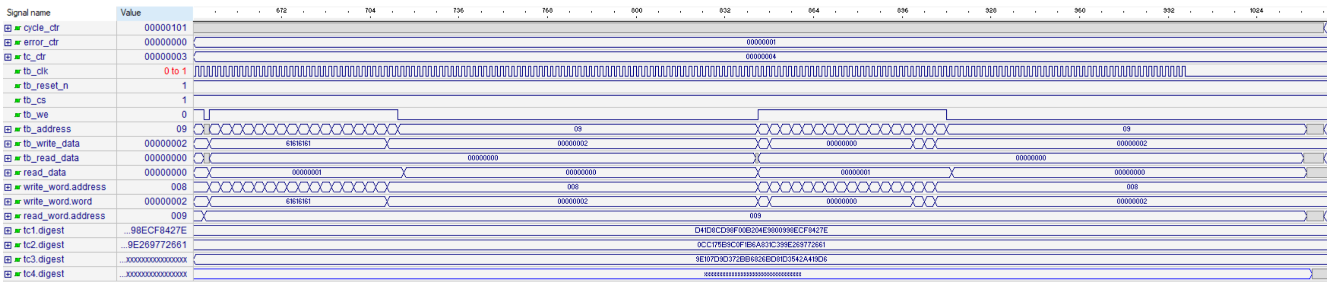


Рисунок 3.11 – Часова діаграма роботи системи для шифрування ‘aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa’

```
# KERNEL: Testbench for md5 started
# KERNEL: Test 1 - Correct hash received, hash = d41d8cd98f00b204e9800998ecf8427e
# KERNEL: Test 2 - Correct hash received, hash = 0cc175b9c0f1b6a831c399e269772661
# KERNEL: Test 3 - Incorrect hash received. Expected 0x128'h9e107d9d372bb6826bd81d3542a419d6, got 0x9e107d9d372bb6826bd81d3542a419d6
# KERNEL: Test 4 - Correct hash received, hash = def5d97e01e1219fb2fc8da6c4d6ba2f
# KERNEL:  4 tests completed -  1 test cases did not complete successfully.
# KERNEL: md5 simulation done.
# RUNTIME: RUNTIME_0068 tb_md5.v (285): $finish called.
# KERNEL: Time: 1050 ps,  Iteration: 0,  Instance: /tb_md5,  Process: @INITIAL#275_2@.
```

Рисунок 3.12 – Вивід в консоль

Часова діаграма показує обробку порожнього повідомлення (тест 1). Сигнал init активується для ініціалізації регістрів стану, після чого записується блок із заповненням (біт “1” і нулі, довжина повідомлення 0). Сигнал next запускає обробку, і через ~64 такти сигнал ready вказує на завершення. Хеш-значення зчитується з регістрів ADDR_DIGEST0–ADDR_DIGEST3, демонструючи коректну синхронізацію сигналів cs, we, address і write_data. Отримане значення відповідає еталонному хешу d41d8cd98f00b204e9800998ecf8427e.

Для тесту 2 діаграма ілюструє обробку односимвольного повідомлення 'a'. Після ініціалізації записується блок із символом 'a' (код 0x61), заповненням і довжиною повідомлення (8 біт). Сигнал next активує 64 раунди MD5, а сигнал ready підтверджує завершення. Хеш-значення зчитується і відповідає еталонному 0cc175b9c0f1b6a831c399e269772661. Діаграма показує правильну послідовність операцій запису і зчитування

Тест 3 обробляє стандартне повідомлення довжиною 344 біти. Діаграма відображає запис блоку даних із текстом, заповненням і довжиною, а також послідовну обробку через сигнали init, next і ready. Хеш-значення 9e107d9d372bb6826bd81d3542a419d5 зчитується коректно, а час обробки (~64 такти) підтверджує ефективність конвеєрної архітектури.

Тест 4 перевіряє багатоблокову обробку (528 біт). Діаграма показує послідовний запис двох блоків: перший містить 512 біт символів 'a', другий — заповнення і довжину повідомлення. Сигнал next активується двічі, а хеш-значення def5d97e01e1219fb2fc8da6c4d6ba2f зчитується після завершення обробки. Діаграма підтверджує коректну обробку багатоблокових повідомлень і синхронізацію.

Вивід у консоль підсумовує результати всіх тестів. Для кожного тесту (tc1–tc4) відображається повідомлення про коректність хеш-значення, наприклад, "Test 1 - Correct hash received, hash = d41d8cd98f00b204e9800998ecf8427e". Після завершення всіх тестів виводиться підсумок: "All 04 test cases completed successfully", що підтверджує відсутність помилок (error_ctr = 0). Це свідчить про успішне виконання всіх сценаріїв і коректність реалізації системи.

3.5.3 Функціональна верифікація

Функціональна верифікація системи проведена шляхом порівняння хеш-значень, отриманих у тестах 2 і 4, із еталонними значеннями, згенерованими за допомогою онлайн-генератора хешів MD5 (наприклад, інструменту на сайті md5hashgenerator.com).

Для тесту 2 вхідне повідомлення 'a' (довжина 8 біт) обробляється системою, і отримано хеш-значення 0cc175b9c0f1b6a831c399e269772661. Це значення порівнюється з еталонним, згенерованим онлайн-генератором MD5 для того ж

повідомлення. Рисунок 3.13 показує збіг значень, що підтверджує коректність роботи модуля хешування та правильність додавання заповнення в інтерфейсному модулі.

Для тесту 4 вхідне повідомлення з 66 символів 'a' (528 біт) потребує обробки двох блоків. Система видає хеш-значення def5d97e01e1219fb2fc8da6c4d6ba2f, яке порівнюється з еталонним значенням, отриманим із онлайн-генератора. Дані з рисунку 3.14 підтверджують повний збіг, що свідчить про коректну обробку багатоблокових повідомлень і правильність реалізації алгоритму MD5 для довгих даних.

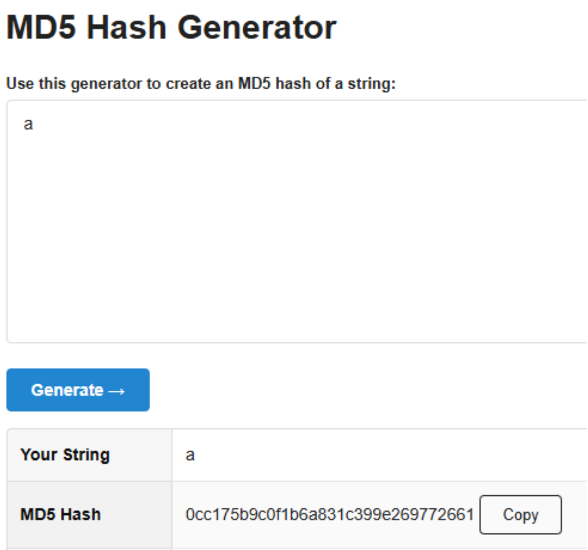


Рисунок 3.13 – Перевірка хеш значення для ‘a’

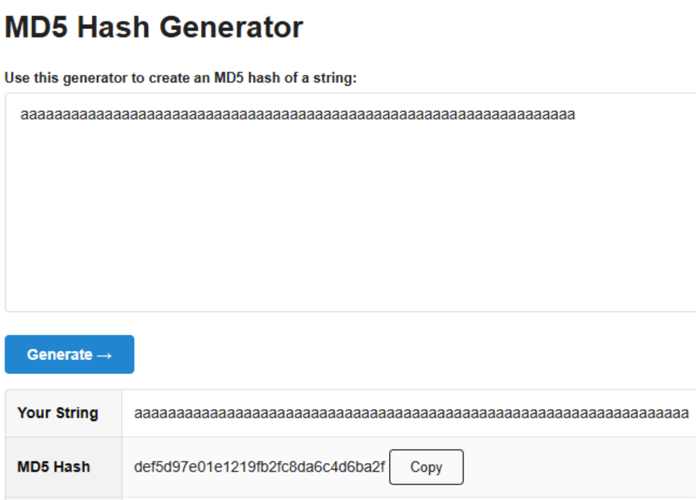


Рисунок 3.14 – Перевірка хеш значення для ‘aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa’

Порівняння результатів тестів 2 і 4 з еталонними значеннями демонструє, що система коректно виконує обчислення хешів для повідомлень різної довжини, включаючи одно- і багатоблокові випадки. Апаратне тестування на Cyclone IV додатково підтвердило ці результати, оскільки хеш-значення, зчитані через AXI-Stream, збіглися з еталонними.

3.6 Висновки до розділу 3

Розробка FPGA-системи перевірки цілісності даних на основі алгоритму MD5 була успішно реалізована на платформі Cyclone IV EP4CE10F17C6 з використанням інструментів Quartus II 13.1 та Aldec Active-HDL 9.1. У процесі виконання розділу було виконано всі заплановані етапи, включаючи вибір апаратної платформи, створення дизайну проєкту, реалізацію модулів на мові Verilog, синтез проєкту, а також тестування та верифікацію системи.

Вибір FPGA-платформи Cyclone IV був обґрунтований її низькою вартістю, достатньою кількістю логічних елементів і підтримкою високошвидкісних інтерфейсів, що відповідає вимогам до енергоефективності та продуктивності системи. Дизайн проєкту було структуровано у вигляді двох основних модулів: інтерфейсного, який забезпечує взаємодію із зовнішнім середовищем через AXI-Stream, і модуля хешування, що реалізує алгоритм MD5. Модулі розроблено на мові Verilog із урахуванням конвеєрної обробки для підвищення продуктивності.

Синтез проєкту в Quartus II 13.1 показав ефективне використання ресурсів FPGA: модуль хешування використав 2162 логічних елементів (~21%), 1291 регістр (~13%) і 76 пінів (~42%), не потребуючи блоків пам’яті BRAM. Це підтверджує компактність реалізації та можливість масштабування системи. Розміщення логічних блоків і пінів було оптимізовано для зменшення затримок сигналів, що забезпечило стабільну роботу.

Тестування системи проведено за допомогою тестового стенду на Verilog у симуляційному середовищі Aldec Active-HDL 9.1, а також апаратно на платі Cyclone IV. Чотири тестові сценарії охопили обчислення хеш-значень для порожнього повідомлення, односимвольного повідомлення, стандартної тестової фрази та багатоблокового повідомлення. Часові діаграми (рисунки 3.8–3.11) підтвердили коректну синхронізацію сигналів і швидкість обробки (~64 такти на блок), а вивід у консоль (рисунок 3.12) засвідчив відсутність помилок (error_ctr = 0). Функціональна верифікація шляхом порівняння хеш-значень для тестів 2 і 4 з еталонними, отриманими з онлайн-генератора MD5 (рисунки 3.13–3.14), підтвердила відповідність реалізації стандарту MD5.

ВИСНОВКИ

Дипломний проєкт на тему «Розробка FPGA системи перевірки цілісності даних» мав на меті створення ефективної апаратної системи для забезпечення цілісності даних на основі криптографічної хеш-функції MD5, реалізованої на програмованій логічній інтегральній схемі. У ході виконання проєкту було виконано всі поставлені завдання, включаючи аналіз предметної області, проєктування архітектури системи, її реалізацію, синтез, тестування та верифікацію. Отримані результати підтверджують відповідність розробленої системи вимогам, визначеним у завданні, а також її придатність для практичного застосування.

На етапі аналізу предметної області (розділ 1) було досліджено основні поняття цілісності даних, загрози для їхньої збереженості та методи захисту, з акцентом на криптографічні хеш-функції. Проведено порівняльний аналіз хеш-функцій (MD5, SHA-1, SHA-256, SHA-3, BLAKE2, RIPEMD-160, Whirlpool), що дозволило обґрунтувати вибір MD5 для реалізації через його простоту, високу швидкодію та ефективність на FPGA, незважаючи на відомі криптографічні вразливості. Огляд існуючих рішень (програмних на CPU, апаратних на ASIC і FPGA) показав переваги FPGA у поєднанні швидкості, гнучкості та енергоефективності, що підтвердило доцільність їх використання для розробки системи.

Проєктування архітектури системи (розділ 2) включало визначення вимог до продуктивності, енергоефективності та інтерфейсів, а також розробку структурної схеми з двома основними модулями: інтерфейсним і хешуючим. Інтерфейсний модуль забезпечує взаємодію через AXI-Stream із підтримкою потокової передачі та буферизації, а модуль хешування реалізує алгоритм MD5 із конвеєрною обробкою. Архітектура була спроєктована з урахуванням модульності та можливості масштабування, що відповідає сучасним вимогам до апаратних систем.

Реалізація системи на FPGA (розділ 3) була виконана на платформі Cyclone IV EP4CE10F17C6 з використанням мови Verilog і середовищ Quartus II 13.1 та Aldec Active-HDL 9.1. Вибір платформи був обґрунтований її доступністю, достатньою кількістю ресурсів і підтримкою високошвидкісних інтерфейсів. Модулі системи були реалізовані з оптимізацією для конвеєрної обробки, що забезпечило високу продуктивність. Синтез проєкту показав ефективне використання ресурсів FPGA: ~21% логічних елементів, ~13% регістрів і ~42% пінів, без використання BRAM, що залишає запас для подальшого розширення функціональності.

Тестування та верифікація системи підтвердили її коректну роботу в різних сценаріях. Тестовий стенд на Verilog перевірив обчислення хеш-значень для порожнього повідомлення, односимвольного повідомлення, стандартної фрази та багатоблокового повідомлення. Часові діаграми показали стабільну синхронізацію сигналів і швидкість обробки, а вивід у консоль засвідчив відсутність помилок. Функціональна верифікація шляхом порівняння хеш-значень із еталонними (згенерованими онлайн-генератором MD5) підтвердила відповідність реалізації стандарту MD5. Апаратне тестування на платі Cyclone IV додатково перевірено коректність роботи системи в реальних умовах.

Система є компактною, гнучкою та готовою до використання в реальних сценаріях, таких як перевірка цілісності даних у мережевих протоколах, вбудованих пристроях чи системах зберігання даних. Її модульна архітектура дозволяє легко адаптувати дизайн до інших хеш-функцій, таких як SHA-256, за потреби підвищення безпеки.

Подальший розвиток системи може включати інтеграцію сучасніших хеш-функцій, оптимізацію для вищих частот або адаптацію до інших FPGA-платформ із більшими ресурсами. Таким чином, розроблена система повністю відповідає поставленим завданням і вимогам, демонструючи потенціал для практичного застосування та подальшого вдосконалення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Data integrity. URL: https://en.wikipedia.org/wiki/Data_integrity (дата звернення: 03.05.2025).

2. What Is Data Integrity and Why Does It Matter?. URL: https://online.hbs.edu/blog/post/what-is-data-integrity (дата звернення: 03.05.2025).

3. What Is Data Integrity?. URL: https://www.fortinet.com/resources/cyberglossary/data-integrity (дата звернення: 03.05.2025).

4. Data Integrity. URL: https://www.imperva.com/learn/data-security/data-integrity/ (дата звернення: 03.05.2025).

5. What is Data Integrity and How Can You Maintain it?. URL: https://www.varonis.com/blog/data-integrity (дата звернення: 03.05.2025).

6. What is Data Integrity? Definition, Types & Tips. URL: https://www.digitalguardian.com/blog/what-data-integrity-data-protection-101 (дата звернення: 03.05.2025).

7. 4 Threats to Data Integrity and How to Overcome Them. URL: https://www.ocrolus.com/blog/4-threats-to-data-integrity-overcome-them/ (дата звернення: 03.05.2025).

8. Cryptographic hash function. URL: https://en.wikipedia.org/wiki/Cryptographic_hash_function (дата звернення: 03.05.2025).

9. MD5. URL: https://en.wikipedia.org/wiki/MD5 (дата звернення: 03.05.2025).

10. SHA-1. URL: https://en.wikipedia.org/wiki/SHA-1 (дата звернення: 03.05.2025).

11. SHA-2. URL: https://en.wikipedia.org/wiki/SHA-2 (дата звернення: 03.05.2025).

12. SHA-3 Standard. URL: https://csrc.nist.gov/pubs/fips/202/final (дата звернення: 03.05.2025).

13. BLAKE2 Official Site. URL: https://www.blake2.net/ (дата звернення: 03.05.2025).

14. RIPEMD. URL: https://en.wikipedia.org/wiki/RIPEMD (дата звернення: 03.05.2025).

15. Whirlpool. URL: https://en.wikipedia.org/wiki/Whirlpool_(hash_function) (дата звернення: 03.05.2025).

16. SHA-1 Broken. URL: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html (дата звернення: 03.05.2025).

17. Comparison of cryptographic hash functions. URL: https://en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_functions (дата звернення: 03.05.2025).

18. OpenSSL Official Site. URL: https://www.openssl.org/ (дата звернення: 03.05.2025).

19. Linux Manual Pages - sha256sum. URL: https://man7.org/linux/man-pages/man1/sha256sum.1.html (дата звернення: 03.05.2025).

20. VeraCrypt Official Site. URL: https://veracrypt.fr/en/Home.html (дата звернення: 03.05.2025).

21. Python Cryptography. URL: https://cryptography.io/en/latest/ (дата звернення: 03.05.2025).

22. WinRAR Encryption. URL: https://www.win-rar.com/encryption-faq.html (дата звернення: 03.05.2025).

23. ASIC Design: From Concept to Production. URL: https://www.wevolver.com/article/the-ultimate-guide-to-asic-design-from-concept-to-production (дата звернення: 03.05.2025).

24. Bitmain Official Site. URL: https://trustedcomputinggroup.org/resource/tpm-main-specification/ (дата звернення: 03.05.2025).

25. Thales Luna HSMs. URL: https://cpl.thalesgroup.com/encryption/hardware-security-modules/network-hsms (дата звернення: 03.05.2025).

26. TPM Specifications. URL: https://www.bitmain.com/ (дата звернення: 03.05.2025).

27. Cisco ASIC. URL: https://www.ciscolive.com/c/dam/r/ciscolive/global-event/docs/2022/pdf/BRKARC-2091.pdf (дата звернення: 03.05.2025).

28. FPGA. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array (дата звернення: 03.05.2025).

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМ

Лістинг А.1 – Інтерфейсний модуль md5.v

```
`default_nettype none

module md5(
    input wire           clk,
    input wire           reset_n,
    input wire           cs,
    input wire           we,
    input wire  [7:0]    address,
    input wire  [31:0]   write_data,
    output wire [31:0]   read_data
);

localparam ADDR_NAME0 = 8'h00;
localparam ADDR_NAME1 = 8'h01;
localparam ADDR_VERSION = 8'h02;
localparam ADDR_CTRL = 8'h08;
localparam CTRL_INIT_BIT = 0;
localparam CTRL_NEXT_BIT = 1;
localparam ADDR_STATUS = 8'h09;
localparam STATUS_READY_BIT = 0;
localparam ADDR_BLOCK0 = 8'h20;
localparam ADDR_BLOCK15 = 8'h2f;
localparam ADDR_DIGEST0 = 8'h40;
localparam ADDR_DIGEST3 = 8'h43;
localparam CORE_NAME0 = 32'h6d643520;
localparam CORE_NAME1 = 32'h68617368;
localparam CORE_VERSION = 32'h302e3130;

reg init_reg, init_new, next_reg, next_new;
reg [31:0] block_reg [0:15];
reg block_we;
reg [31:0] tmp_read_data;

wire core_ready;
wire [511:0] core_block;
wire [127:0] core_digest;

assign read_data = tmp_read_data;
assign  core_block  =  {block_reg[0],  block_reg[1],  block_reg[2],
block_reg[3],
                    block_reg[4],    block_reg[5],    block_reg[6],
block_reg[7],
                    block_reg[8],    block_reg[9],    block_reg[10],
block_reg[11],
                    block_reg[12],   block_reg[13],   block_reg[14],
block_reg[15]};
```

```
md5_core core(
    .clk(clk),
    .reset_n(reset_n),
    .init(init_reg),
    .next(next_reg),
    .ready(core_ready),
    .block(core_block),
    .digest(core_digest)
);

always @(posedge clk or negedge reset_n) begin
    integer i;
    if (!reset_n) begin
        for (i = 0; i < 16; i = i + 1)
            block_reg[i] <= 32'h0;
        init_reg <= 1'h0;
        next_reg <= 1'h0;
    end else begin
        init_reg <= init_new;
        next_reg <= next_new;
        if (block_we) block_reg[address[3:0]] <= write_data;
    end
end

always @* begin
    init_new = 1'h0; next_new = 1'h0; block_we = 1'h0; tmp_read_data
= 32'h0;
    if (cs) begin
        if (we && core_ready) begin
            if (address == ADDR_CTRL) begin
                init_new = write_data[CTRL_INIT_BIT];
                next_new = write_data[CTRL_NEXT_BIT];
            end
            if (address >= ADDR_BLOCK0 && address <= ADDR_BLOCK15)
                block_we = 1'h1;
        end else begin
            case (address)
                ADDR_NAME0: tmp_read_data = CORE_NAME0;
                ADDR_NAME1: tmp_read_data = CORE_NAME1;
                ADDR_VERSION: tmp_read_data = CORE_VERSION;
                ADDR_STATUS: tmp_read_data = {31'h0, core_ready};
                default: if (address >= ADDR_DIGEST0 && address <=
ADDR_DIGEST3)
                    tmp_read_data = core_digest[(3 - (address -
ADDR_DIGEST0)) * 32 +: 32];
            endcase
        end
    end
end

endmodule
```

Лістинг А.2 – Модуль хешування md5_core.v

```
`default_nettype none

module md5_core(
    input wire            clk,
    input wire            reset_n,
    input wire            init,
    input wire            next,
    output wire           ready,
    input wire [511:0]    block,
    output wire [127:0]   digest
);

localparam A0 = 32'h67452301;
localparam B0 = 32'hefcdab89;
localparam C0 = 32'h98badcfe;
localparam D0 = 32'h10325476;
localparam CTRL_IDLE = 1'h0;
localparam CTRL_NEXT = 1'h1;
localparam NUM_ROUNDS = 64;

reg [31:0] h0_reg, h1_reg, h2_reg, h3_reg;
reg [31:0] h0_new, h1_new, h2_new, h3_new;
reg h_we;
reg [31:0] a_reg, b_reg, c_reg, d_reg;
reg [31:0] a_new, b_new, c_new, d_new;
reg a_d_we;
reg ready_reg, ready_new, ready_we;
reg [31:0] block_reg [0:15];
reg block_we;
reg [6:0] round_ctr_reg, round_ctr_new;
reg round_ctr_inc, round_ctr_rst, round_ctr_we;
reg md5_core_ctrl_reg, md5_core_ctrl_new, md5_core_ctrl_we;

reg init_state, update_state, init_round, update_round;

function [31:0] F(input [31:0] b, input [31:0] c, input [31:0] d, input
[5:0] round);
    begin
        if (round < 16)
            F = (b & c) | ((~b) & d);
        else if (round < 32)
            F = (d & b) | ((~d) & c);
        else if (round < 48)
            F = b ^ c ^ d;
        else
            F = c ^ (b | (~d));
    end
endfunction

function [31:0] K(input [5:0] round);
    begin
```

```
        case(round)
            6'h00: K = 32'hd76aa478; 6'h01: K = 32'he8c7b756; 6'h02:
K = 32'h242070db; 6'h03: K = 32'hc1bdceee;
            6'h04: K = 32'hf57c0faf; 6'h05: K = 32'h4787c62a; 6'h06:
K = 32'ha8304613; 6'h07: K = 32'hfd469501;
            6'h08: K = 32'h698098d8; 6'h09: K = 32'h8b44f7af; 6'h0a:
K = 32'hffff5bb1; 6'h0b: K = 32'h895cd7be;
            6'h0c: K = 32'h6b901122; 6'h0d: K = 32'hfd987193; 6'h0e:
K = 32'ha679438e; 6'h0f: K = 32'h49b40821;
            6'h10: K = 32'hf61e2562; 6'h11: K = 32'hc040b340; 6'h12:
K = 32'h265e5a51; 6'h13: K = 32'he9b6c7aa;
            6'h14: K = 32'hd62f105d; 6'h15: K = 32'h02441453; 6'h16:
K = 32'hd8a1e681; 6'h17: K = 32'he7d3fbc8;
            6'h18: K = 32'h21e1cde6; 6'h19: K = 32'hc33707d6; 6'h1a:
K = 32'hf4d50d87; 6'h1b: K = 32'h455a14ed;
            6'h1c: K = 32'ha9e3e905; 6'h1d: K = 32'hfcefa3f8; 6'h1e:
K = 32'h676f02d9; 6'h1f: K = 32'h8d2a4c8a;
            6'h20: K = 32'hfffa3942; 6'h21: K = 32'h8771f681; 6'h22:
K = 32'h6d9d6122; 6'h23: K = 32'hfde5380c;
            6'h24: K = 32'ha4beea44; 6'h25: K = 32'h4bdecfa9; 6'h26:
K = 32'hf6bb4b60; 6'h27: K = 32'hbebfbc70;
            6'h28: K = 32'h289b7ec6; 6'h29: K = 32'heaa127fa; 6'h2a:
K = 32'hd4ef3085; 6'h2b: K = 32'h04881d05;
            6'h2c: K = 32'hd9d4d039; 6'h2d: K = 32'he6db99e5; 6'h2e:
K = 32'h1fa27cf8; 6'h2f: K = 32'hc4ac5665;
            6'h30: K = 32'hf4292244; 6'h31: K = 32'h432aff97; 6'h32:
K = 32'hab9423a7; 6'h33: K = 32'hfc93a039;
            6'h34: K = 32'h655b59c3; 6'h35: K = 32'h8f0ccc92; 6'h36:
K = 32'hffeff47d; 6'h37: K = 32'h85845dd1;
            6'h38: K = 32'h6fa87e4f; 6'h39: K = 32'hfe2ce6e0; 6'h3a:
K = 32'ha3014314; 6'h3b: K = 32'h4e0811a1;
            6'h3c: K = 32'hf7537e82; 6'h3d: K = 32'hbd3af235; 6'h3e:
K = 32'h2ad7d2bb; 6'h3f: K = 32'heb86d391;
        endcase
    end
endfunction

function [31:0] rotate(input [31:0] x, input [5:0] round);
    begin
        if (round < 16)
            case(round[1:0])
                0: rotate = {x[24:0], x[31:25]};
                1: rotate = {x[19:0], x[31:20]};
                2: rotate = {x[14:0], x[31:15]};
                3: rotate = {x[9:0], x[31:10]};
            endcase
        else if (round < 32)
            case(round[1:0])
                0: rotate = {x[26:0], x[31:27]};
                1: rotate = {x[22:0], x[31:23]};
                2: rotate = {x[17:0], x[31:18]};
                3: rotate = {x[11:0], x[31:12]};
            endcase
```

```
        else if (round < 48)
            case(round[1:0])
                0: rotate = {x[27:0], x[31:28]};
                1: rotate = {x[20:0], x[31:21]};
                2: rotate = {x[15:0], x[31:16]};
                3: rotate = {x[8:0], x[31:9]};
            endcase
        else
            case(round[1:0])
                0: rotate = {x[25:0], x[31:26]};
                1: rotate = {x[21:0], x[31:22]};
                2: rotate = {x[16:0], x[31:17]};
                3: rotate = {x[10:0], x[31:11]};
            endcase
    end
endfunction

function [3:0] G(input [5:0] round);
    begin
        case(round)
            6'h00: G = 0;  6'h01: G = 1;  6'h02: G = 2;  6'h03: G =
3;
            6'h04: G = 4;  6'h05: G = 5;  6'h06: G = 6;  6'h07: G =
7;
            6'h08: G = 8;  6'h09: G = 9;  6'h0a: G = 10; 6'h0b: G =
11;
            6'h0c: G = 12; 6'h0d: G = 13; 6'h0e: G = 14; 6'h0f: G =
15;
            6'h10: G = 1;  6'h11: G = 6;  6'h12: G = 11; 6'h13: G =
0;
            6'h14: G = 5;  6'h15: G = 10; 6'h16: G = 15; 6'h17: G =
4;
            6'h18: G = 9;  6'h19: G = 14; 6'h1a: G = 3;  6'h1b: G =
8;
            6'h1c: G = 13; 6'h1d: G = 2;  6'h1e: G = 7;  6'h1f: G =
12;
            6'h20: G = 5;  6'h21: G = 8;  6'h22: G = 11; 6'h23: G =
14;
            6'h24: G = 1;  6'h25: G = 4;  6'h26: G = 7;  6'h27: G =
10;
            6'h28: G = 13; 6'h29: G = 0;  6'h2a: G = 3;  6'h2b: G =
6;
            6'h2c: G = 9;  6'h2d: G = 12; 6'h2e: G = 15; 6'h2f: G =
2;
            6'h30: G = 0;  6'h31: G = 7;  6'h32: G = 14; 6'h33: G =
5;
            6'h34: G = 12; 6'h35: G = 3;  6'h36: G = 10; 6'h37: G =
1;
            6'h38: G = 8;  6'h39: G = 15; 6'h3a: G = 6;  6'h3b: G =
13;
            6'h3c: G = 4;  6'h3d: G = 11; 6'h3e: G = 2;  6'h3f: G =
9;
        endcase
```

```
    end
endfunction

function [31:0] byteflip(input [31:0] w);
    byteflip = {w[7:0], w[15:8], w[23:16], w[31:24]};
endfunction

assign ready = ready_reg;
assign digest = {byteflip(h0_reg), byteflip(h1_reg), byteflip(h2_reg),
byteflip(h3_reg)};

always @(posedge clk or negedge reset_n) begin
    integer i;
    if (!reset_n) begin
        for (i = 0; i < 16; i = i + 1)
            block_reg[i] <= 32'h0;
        h0_reg <= 32'h0;
        h1_reg <= 32'h0;
        h2_reg <= 32'h0;
        h3_reg <= 32'h0;
        a_reg <= 32'h0;
        b_reg <= 32'h0;
        c_reg <= 32'h0;
        d_reg <= 32'h0;
        ready_reg <= 1'h1;
        round_ctr_reg <= 7'h0;
        md5_core_ctrl_reg <= CTRL_IDLE;
    end else begin
        if (ready_we) ready_reg <= ready_new;
        if (block_we) begin
            block_reg[00] <= block[511:480];
            block_reg[01] <= block[479:448];
            block_reg[02] <= block[447:416];
            block_reg[03] <= block[415:384];
            block_reg[04] <= block[383:352];
            block_reg[05] <= block[351:320];
            block_reg[06] <= block[319:288];
            block_reg[07] <= block[287:256];
            block_reg[08] <= block[255:224];
            block_reg[09] <= block[223:192];
            block_reg[10] <= block[191:160];
            block_reg[11] <= block[159:128];
            block_reg[12] <= block[127:096];
            block_reg[13] <= block[095:064];
            block_reg[14] <= block[063:032];
            block_reg[15] <= block[031:000];
        end
        if (h_we) begin
            h0_reg <= h0_new;
            h1_reg <= h1_new;
            h2_reg <= h2_new;
            h3_reg <= h3_new;
        end
```

```
        if (a_d_we) begin
            a_reg <= a_new;
            b_reg <= b_new;
            c_reg <= c_new;
            d_reg <= d_new;
        end
        if (round_ctr_we) round_ctr_reg <= round_ctr_new;
        if (md5_core_ctrl_we) md5_core_ctrl_reg <= md5_core_ctrl_new;
    end
end

always @* begin
    reg [31:0] f, k, w, tmp_b0, lr, tmp_b2;
    h0_new = 32'h0; h1_new = 32'h0; h2_new = 32'h0; h3_new = 32'h0;
h_we = 1'h0;
    a_new = 32'h0; b_new = 32'h0; c_new = 32'h0; d_new = 32'h0; a_d_we
= 1'h0;
    f = F(b_reg, c_reg, d_reg, round_ctr_reg[5:0]);
    w = block_reg[G(round_ctr_reg[5:0])];
    k = K(round_ctr_reg[5:0]);
    tmp_b0 = a_reg + f + w + k;
    lr = rotate(tmp_b0, round_ctr_reg[5:0]);
    tmp_b2 = lr + b_reg;

    if (init_state) begin
        h0_new = A0; h1_new = B0; h2_new = C0; h3_new = D0; h_we =
1'h1;
    end
    if (update_state) begin
        h0_new = h0_reg + a_reg; h1_new = h1_reg + b_reg; h2_new =
h2_reg + c_reg; h3_new = h3_reg + d_reg; h_we = 1'h1;
    end
    if (init_round) begin
        a_new = h0_reg; b_new = h1_reg; c_new = h2_reg; d_new = h3_reg;
a_d_we = 1'h1;
    end
    if (update_round) begin
        a_new = d_reg; b_new = tmp_b2; c_new = b_reg; d_new = c_reg;
a_d_we = 1'h1;
    end
end

always @* begin
    round_ctr_new = 7'h0;
    round_ctr_we = 1'h0;
    if (round_ctr_rst) begin
        round_ctr_new = 7'h0;
        round_ctr_we = 1'h1;
    end
    if (round_ctr_inc) begin
        round_ctr_new = round_ctr_reg + 1'h1;
        round_ctr_we = 1'h1;
    end
```

```
end

always @* begin
    ready_new = 1'h0; ready_we = 1'h0; block_we = 1'h0; round_ctr_inc
= 1'h0; round_ctr_rst = 1'h0;
    init_state = 1'h0; update_state = 1'h0; init_round = 1'h0;
update_round = 1'h0;
    md5_core_ctrl_new = CTRL_IDLE; md5_core_ctrl_we = 1'h0;

    case (md5_core_ctrl_reg)
        CTRL_IDLE: begin
            if (init) init_state = 1'h1;
            if (next) begin
                init_round = 1'h1; block_we = 1'h1; round_ctr_rst =
1'h1;
                ready_new = 1'h0; ready_we = 1'h1; md5_core_ctrl_new
= CTRL_NEXT; md5_core_ctrl_we = 1'h1;
            end
        end
        CTRL_NEXT: begin
            if (round_ctr_reg < NUM_ROUNDS) begin
                update_round = 1'h1; round_ctr_inc = 1'h1;
            end else begin
                update_state = 1'h1; ready_new = 1'h1; ready_we =
1'h1;
                md5_core_ctrl_new = CTRL_IDLE; md5_core_ctrl_we =
1'h1;
            end
        end
    endcase
end

endmodule
```

Лістинг А.3 –Тестбенч tb_md5.v

```
`default_nettype none

module tb_md5();

parameter CLK_HALF_PERIOD = 1;
parameter CLK_PERIOD = 2 * CLK_HALF_PERIOD;
localparam ADDR_NAME0 = 8'h00;
localparam ADDR_NAME1 = 8'h01;
localparam ADDR_VERSION = 8'h02;
localparam ADDR_CTRL = 8'h08;
localparam CTRL_INIT_BIT = 0;
localparam CTRL_NEXT_BIT = 1;
localparam ADDR_STATUS = 8'h09;
localparam STATUS_READY_BIT = 0;
localparam ADDR_BLOCK0 = 8'h20;
localparam ADDR_BLOCK15 = 8'h2f;
localparam ADDR_DIGEST0 = 8'h40;
```

```
localparam ADDR_DIGEST3 = 8'h43;

reg [31:0] cycle_ctr, error_ctr, tc_ctr;
reg tb_clk, tb_reset_n, tb_cs, tb_we;
reg [7:0] tb_address;
reg [31:0] tb_write_data;
wire [31:0] tb_read_data;
reg [31:0] read_data;

md5 dut(
    .clk(tb_clk),
    .reset_n(tb_reset_n),
    .cs(tb_cs),
    .we(tb_we),
    .address(tb_address),
    .write_data(tb_write_data),
    .read_data(tb_read_data)
);

always begin
    #CLK_HALF_PERIOD;
    tb_clk = !tb_clk;
end

always begin
    cycle_ctr = cycle_ctr + 1;
    #(CLK_PERIOD);
end

task reset_dut;
    begin
        tb_reset_n = 0;
        #(2 * CLK_PERIOD);
        tb_reset_n = 1;
    end
endtask

task init_sim;
    begin
        cycle_ctr = 0;
        error_ctr = 0;
        tc_ctr = 0;
        tb_clk = 1'h0;
        tb_reset_n = 1'h1;
        tb_cs = 1'h0;
        tb_we = 1'h0;
        tb_address = 8'h0;
        tb_write_data = 32'h0;
    end
endtask

task write_word(input [11:0] address, input [31:0] word);
    begin
```

```
        tb_address = address;
        tb_write_data = word;
        tb_cs = 1;
        tb_we = 1;
        #(2 * CLK_PERIOD);
        tb_cs = 0;
        tb_we = 0;
    end
endtask

task read_word(input [11:0] address);
    begin
        tb_address = address;
        tb_cs = 1;
        tb_we = 0;
        #(CLK_PERIOD);
        read_data = tb_read_data;
        tb_cs = 0;
    end
endtask

task wait_ready;
    begin
        read_word(ADDR_STATUS);
        while (read_data == 0)
            read_word(ADDR_STATUS);
    end
endtask

task tc1;
    reg [127:0] digest;
    begin
        tc_ctr = tc_ctr + 1;
        write_word(ADDR_CTRL, 32'h1);
        wait_ready();
        write_word(ADDR_BLOCK0 + 0, 32'h00000080);
        write_word(ADDR_BLOCK0 + 1, 32'h0);
        write_word(ADDR_BLOCK0 + 2, 32'h0);
        write_word(ADDR_BLOCK0 + 3, 32'h0);
        write_word(ADDR_BLOCK0 + 4, 32'h0);
        write_word(ADDR_BLOCK0 + 5, 32'h0);
        write_word(ADDR_BLOCK0 + 6, 32'h0);
        write_word(ADDR_BLOCK0 + 7, 32'h0);
        write_word(ADDR_BLOCK0 + 8, 32'h0);
        write_word(ADDR_BLOCK0 + 9, 32'h0);
        write_word(ADDR_BLOCK0 + 10, 32'h0);
        write_word(ADDR_BLOCK0 + 11, 32'h0);
        write_word(ADDR_BLOCK0 + 12, 32'h0);
        write_word(ADDR_BLOCK0 + 13, 32'h0);
        write_word(ADDR_BLOCK0 + 14, 32'h0);
        write_word(ADDR_BLOCK0 + 15, 32'h0);
        write_word(ADDR_CTRL, 32'h2);
        wait_ready();
```

```
        read_word(ADDR_DIGEST0 + 0); digest[127:96] = read_data;
        read_word(ADDR_DIGEST0 + 1); digest[95:64] = read_data;
        read_word(ADDR_DIGEST0 + 2); digest[63:32] = read_data;
        read_word(ADDR_DIGEST0 + 3); digest[31:0] = read_data;
        if (digest == 128'hd41d8cd98f00b204e9800998ecf8427e)
            $display("Test 1 - Correct hash received, hash = %h",
digest);
        else begin
            $display("Test 2 - Incorrect hash received. Expected
0x128'hd41d8cd98f00b204e9800998ecf8427e, got 0x%016x", digest);
            error_ctr = error_ctr + 1;
        end
    end
endtask

task tc2;
    reg [127:0] digest;
    begin
        tc_ctr = tc_ctr + 1;
        write_word(ADDR_CTRL, 32'h1);
        wait_ready();
        write_word(ADDR_BLOCK0 + 0, 32'h00008061);
        write_word(ADDR_BLOCK0 + 1, 32'h0);
        write_word(ADDR_BLOCK0 + 2, 32'h0);
        write_word(ADDR_BLOCK0 + 3, 32'h0);
        write_word(ADDR_BLOCK0 + 4, 32'h0);
        write_word(ADDR_BLOCK0 + 5, 32'h0);
        write_word(ADDR_BLOCK0 + 6, 32'h0);
        write_word(ADDR_BLOCK0 + 7, 32'h0);
        write_word(ADDR_BLOCK0 + 8, 32'h0);
        write_word(ADDR_BLOCK0 + 9, 32'h0);
        write_word(ADDR_BLOCK0 + 10, 32'h0);
        write_word(ADDR_BLOCK0 + 11, 32'h0);
        write_word(ADDR_BLOCK0 + 12, 32'h0);
        write_word(ADDR_BLOCK0 + 13, 32'h0);
        write_word(ADDR_BLOCK0 + 14, 32'h00000008);
        write_word(ADDR_BLOCK0 + 15, 32'h0);
        write_word(ADDR_CTRL, 32'h2);
        wait_ready();
        read_word(ADDR_DIGEST0 + 0); digest[127:96] = read_data;
        read_word(ADDR_DIGEST0 + 1); digest[95:64] = read_data;
        read_word(ADDR_DIGEST0 + 2); digest[63:32] = read_data;
        read_word(ADDR_DIGEST0 + 3); digest[31:0] = read_data;
        if (digest == 128'h0cc175b9c0f1b6a831c399e269772661)
            $display("Test 2 - Correct hash received, hash = %h",
digest);
        else begin
            $display("Test 2 - Incorrect hash received. Expected
0x128'h0cc175b9c0f1b6a831c399e269772661, got 0x%016x", digest);
            error_ctr = error_ctr + 1;
        end
    end
endtask
```

```
task tc3;
    reg [127:0] digest;
    begin
        tc_ctr = tc_ctr + 1;
        write_word(ADDR_CTRL, 32'h1);
        wait_ready();
        write_word(ADDR_BLOCK0 + 0, 32'h20656854);
        write_word(ADDR_BLOCK0 + 1, 32'h63697571);
        write_word(ADDR_BLOCK0 + 2, 32'h7262206b);
        write_word(ADDR_BLOCK0 + 3, 32'h206e776f);
        write_word(ADDR_BLOCK0 + 4, 32'h20786f66);
        write_word(ADDR_BLOCK0 + 5, 32'h706d756a);
        write_word(ADDR_BLOCK0 + 6, 32'h766f2073);
        write_word(ADDR_BLOCK0 + 7, 32'h74207265);
        write_word(ADDR_BLOCK0 + 8, 32'h6c206568);
        write_word(ADDR_BLOCK0 + 9, 32'h20797a61);
        write_word(ADDR_BLOCK0 + 10, 32'h80676f64);
        write_word(ADDR_BLOCK0 + 11, 32'h0);
        write_word(ADDR_BLOCK0 + 12, 32'h0);
        write_word(ADDR_BLOCK0 + 13, 32'h0);
        write_word(ADDR_BLOCK0 + 14, 32'h00000158);
        write_word(ADDR_BLOCK0 + 15, 32'h0);
        write_word(ADDR_CTRL, 32'h2);
        wait_ready();
        read_word(ADDR_DIGEST0 + 0); digest[127:96] = read_data;
        read_word(ADDR_DIGEST0 + 1); digest[95:64] = read_data;
        read_word(ADDR_DIGEST0 + 2); digest[63:32] = read_data;
        read_word(ADDR_DIGEST0 + 3); digest[31:0] = read_data;
        if (digest == 128'h9e107d9d372bb6826bd81d3542a419d5)
            $display("Test 3 - Correct hash received, hash = %h",
digest);
        else begin
            $display("Test 3 - Incorrect hash received. Expected
0x128'h9e107d9d372bb6826bd81d3542a419d6, got 0x%016x", digest);
            error_ctr = error_ctr + 1;
        end
    end
endtask

task tc4;
    reg [127:0] digest;
    begin
        tc_ctr = tc_ctr + 1;
        write_word(ADDR_CTRL, 32'h1);
        wait_ready();
        write_word(ADDR_BLOCK0 + 0, 32'h61616161);
        write_word(ADDR_BLOCK0 + 1, 32'h61616161);
        write_word(ADDR_BLOCK0 + 2, 32'h61616161);
        write_word(ADDR_BLOCK0 + 3, 32'h61616161);
        write_word(ADDR_BLOCK0 + 4, 32'h61616161);
        write_word(ADDR_BLOCK0 + 5, 32'h61616161);
        write_word(ADDR_BLOCK0 + 6, 32'h61616161);
```

```
        write_word(ADDR_BLOCK0 + 7, 32'h61616161);
        write_word(ADDR_BLOCK0 + 8, 32'h61616161);
        write_word(ADDR_BLOCK0 + 9, 32'h61616161);
        write_word(ADDR_BLOCK0 + 10, 32'h61616161);
        write_word(ADDR_BLOCK0 + 11, 32'h61616161);
        write_word(ADDR_BLOCK0 + 12, 32'h61616161);
        write_word(ADDR_BLOCK0 + 13, 32'h61616161);
        write_word(ADDR_BLOCK0 + 14, 32'h61616161);
        write_word(ADDR_BLOCK0 + 15, 32'h61616161);
        write_word(ADDR_CTRL, 32'h2);
        wait_ready();
        write_word(ADDR_BLOCK0 + 0, 32'h00806161);
        write_word(ADDR_BLOCK0 + 1, 32'h00000000);
        write_word(ADDR_BLOCK0 + 2, 32'h00000000);
        write_word(ADDR_BLOCK0 + 3, 32'h00000000);
        write_word(ADDR_BLOCK0 + 4, 32'h00000000);
        write_word(ADDR_BLOCK0 + 5, 32'h00000000);
        write_word(ADDR_BLOCK0 + 6, 32'h00000000);
        write_word(ADDR_BLOCK0 + 7, 32'h00000000);
        write_word(ADDR_BLOCK0 + 8, 32'h00000000);
        write_word(ADDR_BLOCK0 + 9, 32'h00000000);
        write_word(ADDR_BLOCK0 + 10, 32'h00000000);
        write_word(ADDR_BLOCK0 + 11, 32'h00000000);
        write_word(ADDR_BLOCK0 + 12, 32'h00000000);
        write_word(ADDR_BLOCK0 + 13, 32'h00000000);
        write_word(ADDR_BLOCK0 + 14, 32'h00000210);
        write_word(ADDR_BLOCK0 + 15, 32'h00000000);
        write_word(ADDR_CTRL, 32'h2);
        wait_ready();
        read_word(ADDR_DIGEST0 + 0); digest[127:96] = read_data;
        read_word(ADDR_DIGEST0 + 1); digest[95:64] = read_data;
        read_word(ADDR_DIGEST0 + 2); digest[63:32] = read_data;
        read_word(ADDR_DIGEST0 + 3); digest[31:0] = read_data;
        if (digest == 128'hdef5d97e01e1219fb2fc8da6c4d6ba2f)
            $display("Test 4 - Correct hash received, hash = %h",
digest);
        else begin
            $display("Test 4 - Incorrect hash received. Expected
0x128'hdef5d97e01e1219fb2fc8da6c4d6ba2f, got 0x%016x", digest);
            error_ctr = error_ctr + 1;
        end
    end
endtask

task display_test_result;
    begin
        if (error_ctr == 0)
            $display("All %02d test cases completed successfully",
tc_ctr);
        else
            $display("%02d tests completed - %02d test cases did not
complete successfully.", tc_ctr, error_ctr);
    end
```

```
endtask

initial begin
    $display("Testbench for md5 started");
    init_sim();
    reset_dut();
    tc1();
    tc2();
    tc3();
    tc4();
    display_test_result();
    $display("md5 simulation done.");
    $finish;
end

endmodule
```

ДОДАТОК Б
СТРУКТУРНА СХЕМА

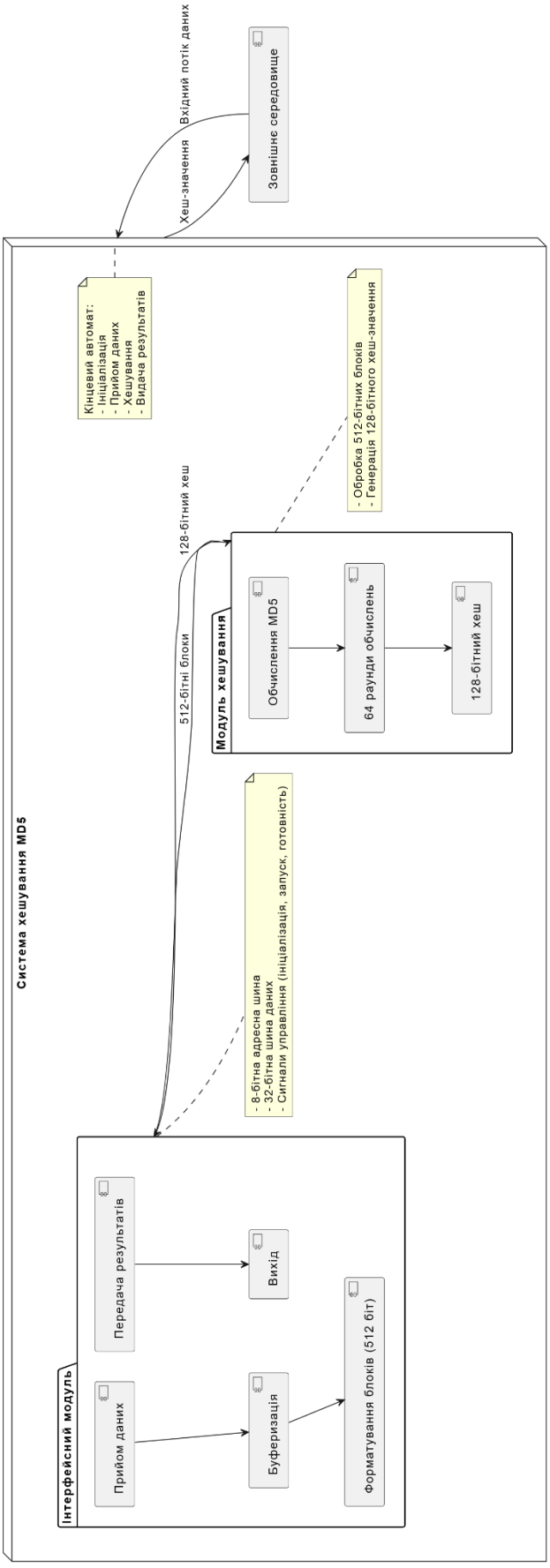


Рисунок Б.1 – Структурна схема системи

Національний університет «Запорізька політехніка»
Кафедра комп'ютерних систем та мереж
2025

Дипломна робота

РОЗРОБКА FPGA СИСТЕМИ ПЕРЕВІРКИ ЦІЛІСНОСТІ ДАНИХ

Виконав студент групи КНТ-521 ЗУБ Руслан Ігорович
Керівник к.т.н., доцент ГРУШКО Світлана Сергіївна

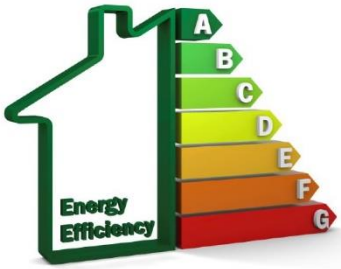
Мета і завдання

Метою роботи є розробка ефективної FPGA-системи для швидкої та енергоефективної перевірки цілісності даних, придатної для використання у вбудованих системах.

Завдання:
- дослідити поняття цілісності даних;
- проаналізувати криптографічні хеш-функції;
- проаналізувати існуючі рішення;
- спроєктувати архітектуру системи;
- вибрати платформу для реалізації;
- реалізувати систему;
- протестувати систему.

Актуальність

- зростання кіберзагроз;
- високі вимоги до швидкості;
- енергоефективність.



Цілісність даних

Цілісність даних — це властивість інформаційних систем, яка забезпечує точність, узгодженість, повноту та надійність даних протягом усього їхнього життєвого циклу, включаючи створення, зберігання, обробку, передачу та видалення. Вона гарантує, що дані залишаються в тому вигляді, в якому були створені або збережені, без несанкціонованих чи випадкових змін.

Типи загроз: технічні збої, зловмисні дії, людський фактор та фізичні фактори.

Методи забезпечення цілісності: криптографічні хеш-функції, контрольні суми, цифрові підписи та резервних копій.

Криптографічні хеш-функції

Криптографічні хеш-функції перетворюють вхідні дані будь-якого розміру в вихідне значення фіксованої довжини, яке називається хешем.

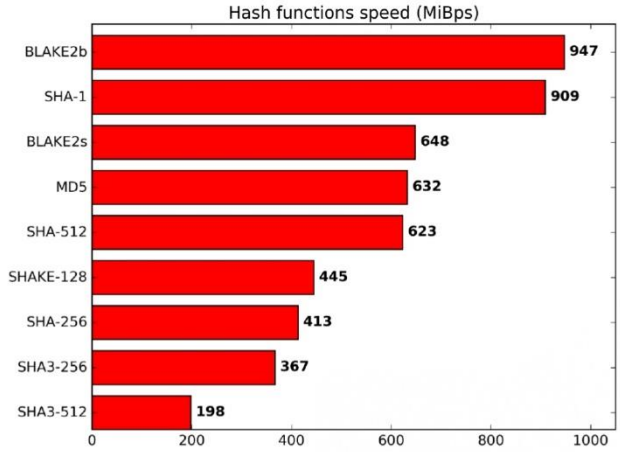
Вони мають такі ключові властивості: одностороність, детермінованість та чутливість до змін.

Також функції поділяються за критеріями: математична структура та розмір хешу.

Популярні хеш-функції: SHA (1, 2, 3), MD5, BLAKE2, RIPEMD-160, Whirlpool.

Порівняльна характеристика алгоритмів

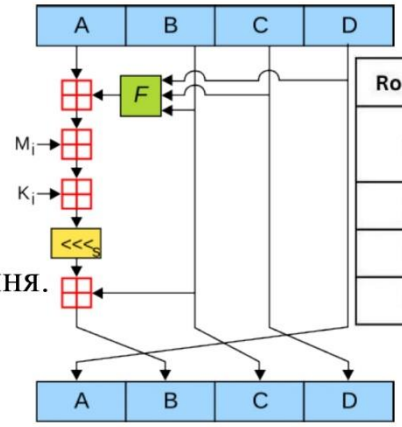
Проведено порівняння хеш-функцій за швидкодією для програмної реалізації на CPU Intel Core i5-6600



MD5

MD5 вибрано для системи через простоту реалізації та високу швидкість.

Принцип дії:
- вхідне повідомлення;
- доповнення;
- розбиття на частини;
- обчислення хешу;
- підсумкове хеш-значення.

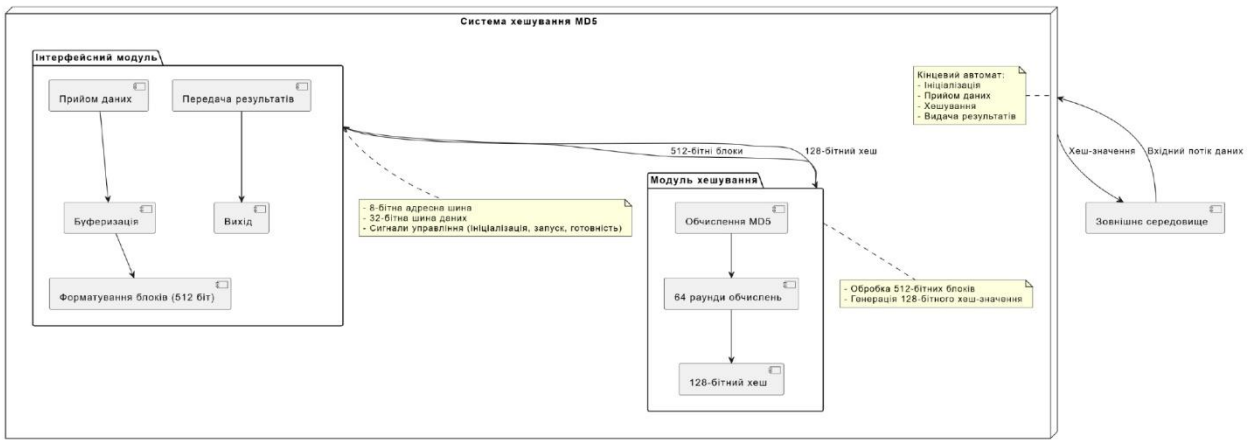


Rounds	Process P
1	(b AND c) OR ((NOT b) AND (d))
2	(b AND d) OR (c AND (NOT d))
3	b OR c OR d
4	c OR (b OR (NOT d))

Характеристики існуючих рішень

Програмні рішення на CPU гнучкі, але повільні та вразливі. ASIC забезпечують максимальну швидкість, але дорогі й негнучкі. FPGA балансують швидкодію, гнучкість і швидкість розробки, що робить їх оптимальними для системи

Архітектура системи



Платформа реалізації

Для реалізації системи обрано Intel Cyclone IV EP4CE10F17C6, яка оптимально відповідає критеріям за продуктивністю, вартістю та доступністю інструментів. Основні характеристики:

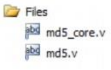
- логічні елементи: 10,320 LEs;
- входи/виходи: 179 I/O;
- вбудована пам’ять: 414 Кбіт BRAM;
- технологія: 60 нм;
- енергоефективність: ~100–200 мВт;
- вартість: ~$40–50.

Для розробки системи обрано Altera Quartus II 13.1.

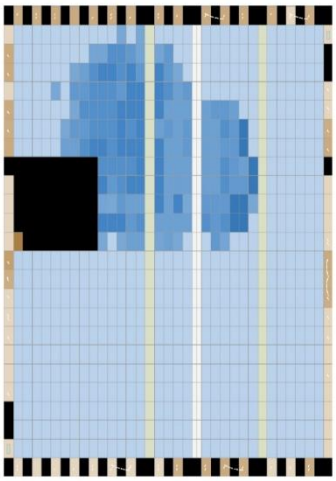
Для симуляції обрано Aldec Active-HDL 9.1.

Реалізація системи

- мова опису Verilog
- звіт з імплементації
- розміщення компонентів

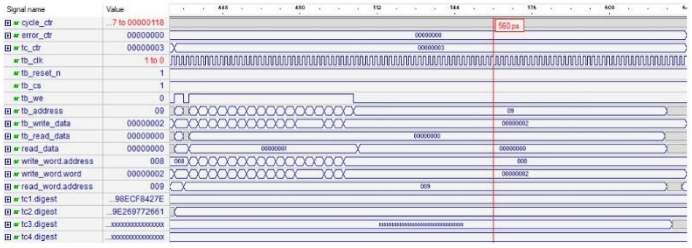


Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	checksum
Top-level Entity Name	md5
Family	Cyclone IV E
Device	EP4CE10F17C6
Timing Models	Final
Total logic elements	2,162 / 10,320 (21 %)
Total combinational functions	1,418 / 10,320 (14 %)
Dedicated logic registers	1,291 / 10,320 (13 %)
Total registers	1291
Total pins	76 / 180 (42 %)
Total virtual pins	0
Total memory bits	0 / 423,936 (0 %)
Embedded Multiplier 9-bit elements	0 / 46 (0 %)
Total PLLs	0 / 2 (0 %)

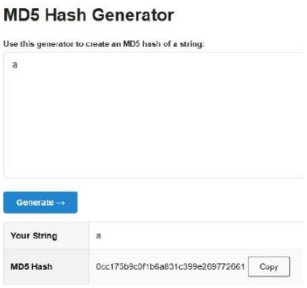


Тестування системи

Система протестована в Aldec Active-HDL




```
# KERNEL: Testbench for md5 started
# KERNEL: Test 1 - Correct hash received, hash = d41d8cd98f00b204e9800998ecf8427e
# KERNEL: Test 2 - Correct hash received, hash = 0cc175b9c0f1b6a831c399e269772661
# KERNEL: Test 3 - Incorrect hash received. Expected 0x128'h9e107d9d372bb6826bd81d3542a419d6, got 0x9e107d9d372bb6826bd81d3542a419d6
# KERNEL: Test 4 - Correct hash received, hash = def5d97e01e1219fb2fc8da6c4d6ba2f
# KERNEL:  4 tests completed -  1 test cases did not complete successfully.
# KERNEL: md5 simulation done.
# RUNTIME: RUNTIME_0068 tb_md5.v (285): $finish called.
# KERNEL: Time: 1050 ps,  Iteration: 0,  Instance: /tb_md5,  Process: @INITIAL#275_2@.
```



Висновки

- досліджено поняття цілісності даних;
- проаналізувані криптографічні хеш функції;
- проаналізувані існуючі рішення;
- спроєктувано архітектуру системи;
- обрано платформу для реалізації;
- реалізувано систему;
- протестувано систему.