НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет/(ННІ) інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
інформаційних технологій	комп'ютерних наук
(назва факультету (ННІ))	(назва кафедри)
_____ Ігор БОЛБОТ	_____ Белла ГОЛУБ
(підпис) (ім'я ПРІЗВИЩЕ)	(підпис) (ім'я ПРІЗВИЩЕ)
"___" _____ 2025 р.	"___" _____ 2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

Дослідження та розробка системи стиснення даних з використанням FPGA

Спеціальність
121 інженерія програмного забезпечення
(код і найменування)

Освітня програма
Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми
освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми
к. ф.-м. н., доцент _____ Віктор КИРИЧЕНКО
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Керівник магістерської кваліфікаційної роботи
к.т.н., доцент _____ Тарас ЛЕНДЄЛ
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Виконав _____ Андрій ДОРОФЄЄВ
(підпис) (ім'я ПРІЗВИЩЕ здобувача)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук

к.т.н., доцент _____ Белла ГОЛУБ
(науковий ступінь, вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)
"____" _____ 2025 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Дорофєєву Андрію Сергійовичу
(прізвище, ім'я, по батькові)

Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)

Освітня програма Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи Дослідження та розробка системи стиснення даних з використанням FPGA
затверджена наказом від "01" листопада 2024р. № 1963 «С»

Термін подання завершеної роботи на кафедру 28.11.2025
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи технічна документація FPGA-платформи, специфікації інтерфейсів обміну даними, алгоритми стиснення

Перелік питань, що підлягають дослідженню:

1. Аналіз алгоритмів стиснення даних та їх вибір за критерієм паралелізму для існуючого апаратного забезпечення.
2. Проєктування FPGA-архітектури для реалізації алгоритму стиснення.
3. Відлагодження FPGA- архітектури для збільшення продуктивності та використання ресурсів FPGA.
4. Порівняння ефективності FPGA-рішення з програмними реалізаціями.

Перелік графічного матеріалу (за потреби) презентація, постер
Дата видачі завдання "_____"_____20___ р.

Керівник магістерської кваліфікаційної роботи _____ Тарас ЛЕНДЄЛ
(підпис) (ім'я ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Андрій ДОРОФЄЄВ
(ім'я ПРІЗВИЩЕ)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

FPGA — *Field-Programmable Gate Array*, програмована користувачем вентильна матриця.

SoC — *System-on-Chip*, система на кристалі, що об’єднує процесорне ядро та програмовану логіку.

HLS — *High-Level Synthesis*, високорівневий синтез, перетворення C/C++-коду в апаратний опис (RTL).

RTL — *Register Transfer Level*, рівень регістрових передач, представлення апаратної логіки.

HDL — *Hardware Description Language*, мова опису апаратури (Verilog, VHDL).

BRAM — *Block RAM*, вбудована блокова пам’ять FPGA.

LUT — *Lookup Table*, таблиця відповідності, базовий логічний елемент FPGA.

LE — *Logic Element*, логічний елемент FPGA, що включає LUT та тригер.

DSP — *Digital Signal Processing block*, апаратний блок для обробки цифрових сигналів (множення/додавання).

PLL — *Phase-Locked Loop*, фазова автопідлаштовувана система для генерації тактових частот.

AXI — *Advanced eXtensible Interface*, шина взаємодії між компонентами в SoC.

AXI-Lite — полегшена версія інтерфейсу AXI для регістрової конфігурації.

AXI-Stream — потоковий інтерфейс AXI для передавання даних у реальному часі.

DMA — *Direct Memory Access*, прямий доступ до пам’яті без участі CPU.

RLE — *Run-Length Encoding*, алгоритм стиснення шляхом заміни повторюваних символів парами (значення, довжина серії).

LZ77 — алгоритм словникового стиснення на основі пошуку повторів у вікні.

Huffman coding — алгоритм ентропійного кодування зі змінною довжиною кодових слів.

Deflate — комбінований алгоритм, що об’єднує LZ77 та Huffman coding.

ARM — процесорна архітектура, що використовується у SoC FPGA (наприклад, у Cyclone V SoC).

CPU — *Central Processing Unit*, центральний процесор.

GPU — *Graphics Processing Unit*, графічний процесор.

IOPS — *Input/Output Operations Per Second*, кількість операцій введення/виведення за секунду.

FIFO — *First In — First Out*, черга з доступом у порядку надходження даних.

FSM — *Finite State Machine*, скінченний автомат.

API — *Application Programming Interface*, інтерфейс для взаємодії між компонентами ПЗ.

SDK — *Software Development Kit*, набір інструментів для розробки.

Quartus Prime — середовище розробки FPGA від Intel.

SoC EDS — *Embedded Design Suite*, набір інструментів для розробки ПЗ під ARM-ядро Cyclone V SoC.

Throughput — пропускна здатність системи, обсяг даних, що обробляється за одиницю часу.

Latency — затримка між надходженням даних та отриманням результату.

Resource utilization — використання ресурсів FPGA (LUT, LE, BRAM, DSP тощо).

Compression ratio — коефіцієнт стиснення, відношення розміру вхідних даних до стиснених.

ВСТУП

Актуальність теми використання FPGA для прискорення стиснення даних обумовлена зростаючими обсягами інформації та необхідністю швидкої обробки й передачі даних у багатьох сферах, таких як телекомунікації, комп'ютерні мережі, великі дані та машинне навчання. FPGA (Field-Programmable Gate Array) надають можливість паралельної обробки даних та можуть бути налаштовані для конкретних завдань, таких як стиснення, що значно підвищує продуктивність порівняно з традиційними програмно-орієнтованими підходами.

Метою роботи є дослідження теоретичних основ та розробка архітектури системи стиснення даних на основі FPGA, яка здатна працювати в реальному часі та забезпечує ефективну обробку інформації, а також порівняння швидкості роботи алгоритмів стиснення на класичних процесорах з архітектурою x86.

Об’єктом дослідження є технології та процеси стиснення даних, що реалізуються як в програмному, так і в апаратному забезпеченні. Це включає в себе різні алгоритми стиснення, такі як LZW, Huffman coding, Arithmetic coding, і їх реалізацію на різних платформах.

Предметом дослідження є специфічні аспекти використання FPGA (FieldProgrammable Gate Array) для прискорення алгоритмів стиснення даних. Це охоплює проектування та оптимізацію FPGA-архітектури для реалізації обраних алгоритмів, аналіз ефективності і продуктивності FPGA в порівнянні з традиційними програмними та апаратними рішеннями, а також вивчення методів підвищення швидкості обробки і зменшення споживання ресурсів.

Для досягнення поставленої мети було поставлено наступні **завдання**:

1. Аналіз алгоритмів стиснення: Провести огляд і вибрати найбільш підходящий алгоритм стиснення даних для реалізації на FPGA.

2. Проектування FPGA-архітектури: Розробити архітектуру FPGA для вибраного алгоритму стиснення, включаючи реалізацію паралельної обробки даних.

3. Оптимізація продуктивності: Визначити і реалізувати методи оптимізації FPGA реалізації для підвищення швидкості обробки та зменшення використання ресурсів.

4. Порівняння з традиційними рішеннями: Провести порівняння ефективності FPGA реалізації з традиційними програмними та апаратними рішеннями по швидкості, ефективності та енергоспоживанню.

5. Тестування та валідація: Виконати тестування розробленого рішення на FPGA для перевірки коректності, стабільності і відповідності вимогам.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АЛГОРИТМІВ СТИСНЕННЯ ТА АРХІТЕКТУРИ FPGA

1.1. Методи та алгоритми стиснення даних

Стиснення даних – це процес перетворення інформації у компактнішу форму, що дозволяє зменшити обсяг даних для зберігання або передачі без суттєвої втрати змісту або з гарантією повного відновлення оригіналу [1]. Цей механізм широко застосовується у сферах архівування файлів, телекомунікаційних мереж, зберігання мультимедійних ресурсів, а також оптимізації обчислювальних операцій у реальному часі, де ефективність використання ресурсів є ключовою.

Для кількісного оцінювання потенційного ступеню стиснення використовується поняття «ентропії інформаційного джерела», введене Клодом Шенноном. Ентропія відображає середню кількість біт необхідних для представлення одного символу [2]. Кількість інформації окремого дискретного повідомлення x_i оцінюють як:

$$H(x_i) = -\log_2 p_i \quad (1.1)$$

де $p_i = p(x = x_i)$ – ймовірність значення x_i дискретної випадкової величини X.

Фундаментальною основою алгоритмів стиснення слугує виявлення та усунення надлишковості, яка може проявлятися у статистичній формі (переважання певних символів чи їх послідовностей), структурній (повторювані шаблони даних) або семантичній (елементи, які не впливають на сприйняття користувачем, наприклад, незначні варіації в аудіо чи зображеннях) [3]. Методи стиснення без втрат переважно використовують статистичні характеристики даних, тоді як методи зі втратами орієнтовані на семантичну надлишковість [4].

Стиснення без втрат забезпечує повне та точне відновлення первинних даних після декомпресії, що робить його незамінним таких доменах, як фінансові технології, обробка конфіденційної інформації, програмне забезпечення,

архівування текстових та виконуваних файлів. Серед класичних алгоритмів можна виділити алгоритм Гаффмана (Huffman coding), арифметичне кодування (arithmetic coding), методи сімейства Лемпеля-Зіва (LZ77, LZ78, LZW), а також сучасні реалізації, включаючи Deflate, Brotli та Zstandard (Zstd).

Алгоритм Гаффмана, один з перших методів ентропійного кодування, ґрунтується на побудові бінарного дерева кодів, де символи з вищою частотою появі присвоюються коротшим кодам, а менш частим – довшим. Це призводить до зменшення середньої довжини коду та загального обсягу даних. Арифметичне кодування вдосконалює цей принцип, застосовуючи неперервні дробові інтервали замість дискретних кодів, що підвищує ефективність для алфавітів з великою кількістю символів [5].

Методи Лемпеля-Зіва (LZ) фокусуються на використанні словників для пошуку повторюваних підрядків у потоці даних. Алгоритм LZ77 використовує концепцію ковзного вікна (sliding window) – буфер з попередньо обробленими даними, де виявлені дублікати замінюються посиланнями у форматі пари (відстань до збігу, довжина послідовності) [6]. LZ78, натомість, будує динамічний словник, додаючи нові комбінації підрядків під час аналізу [7]. Варіант LZW (Lempel–Ziv–Welch), застосовуваний у форматах GIF та TIFF, оптимізує словник, виключаючи явне зберігання початкових символів і забезпечуючи динамічне розширення [8].

Таблиця 1.1

Порівняння словникових алгоритмів.

Алгоритм	**Тип словника**	**Принцип**	**Переваги**	**Недоліки**	**Застосування**
LZ77	Ковзне вікно	Пошук повторів у межах буфера	Простота, потікова обробка	Обмеження розміру вікна	GZIP, Deflate
LZ78	Динамічний словник	Поступове розширення	Добре стискає повтори	Великі таблиці	старі архіватори

Алгоритм	Тип словника	Принцип	Переваги	Недоліки	Застосування
		бази шаблонів			
LZW	Удосконалений LZ78	Використовує коди замість рядків	Швидка декомпресія	Обмежений словник	GIF, TIFF

Сучасні алгоритми комбінують словникові та ентропійні техніки для досягнення оптимального балансу між ступенем стиснення та швидкістю. Deflate, що поєднує LZ77 з кодуванням Гаффмана, став стандартом для форматів ZIP та GZIP [9]. Brotli, розроблений компанією Google, покращує ефективність за рахунок розширеного словника та вдосконаленого ентропійного кодування [10]. Zstandard (Zstd) від Facebook поєднує швидке LZ-подібне кодування з адаптивним енкодером на основі Гаффмана, дозволяючи налаштування параметрів компресії відповідно до вимог продуктивності [11]. Також слід виділити алгоритми LZMA, які використовуються в 7zip [12]. Вони базуються на контекстному моделюванні та передбаченні послідовностей, за рахунок чого демонструють вищий ступінь стиснення, але в свою чергу це призводить зо збільшення вимог до ресурсів.

На рисунку 1.1 наведено спрощену схему алгоритму Deflate. На етапі пошуку збігів здійснюється пошук повторів в ковзному вікні, а далі утворюються пари довжини та відстані для подальшого ентропійного кодування.

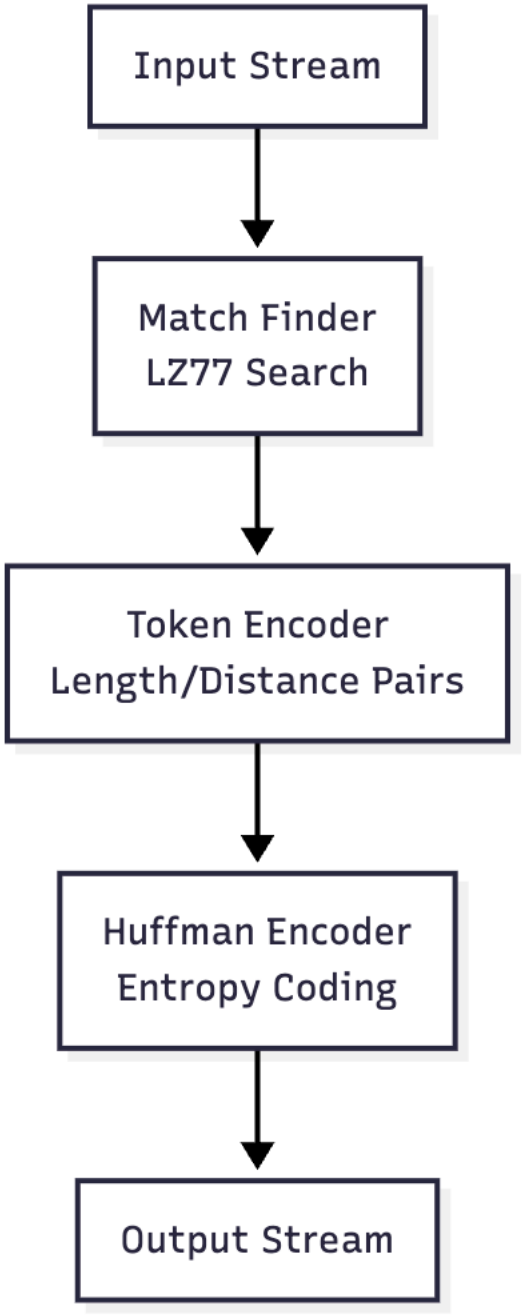


Рис. 1.1 Схема комбінації LZ77 та кодування Гаффмана в Deflate.

Стиснення зі втратами (lossy compression) часто використовується для мультимедіа – зображень, аудіо та відео, де допустиме незначне зниження якості, невідчутне для людського сприйняття. У графічних форматах, таких як JPEG, WebP чи HEIC, застосовуються дискретне косинусне перетворення (DCT) або дискретне вейвлет-перетворення (DWT) для переходу до частотної області та видалення несуттєвих компонентів. Аудіоформати (MP3, AAC) спираються на психоакустичні моделі, що враховують фізіологічні особливості слуху та при

стисненні повністю видаляють частину даних. Для відео (MPEG, H.264, H.265) комбінуються просторове стиснення з часовою кореляцією між кадрами.

Розглядаючи апаратну реалізацію на FPGA перевагу мають алгоритми без втрат, зокрема сімейство LZ, Huffman, Deflate та RLE (Run-Length Encoding), через їх відносну простоту, придатність до конвеєризації та паралельного виконання. Ці методи не вимагають складних обчислень з плаваючою комою чи перетворень у частотну область, що полегшує апаратну адаптацію.

З розвитком машинного навчання почали з’являтись і гібридні методи, що інтегрують його з традиційними алгоритмами. Зокрема, нейронні мережі застосовуються для передбачення послідовностей, що дозволяє покращити методи ентропійного кодування. Але через високі вимоги до ресурсів та складність моделей апаратна реалізація таких методів обмежена.

У сучасних дослідженнях розвивається напрямок нейронного стиснення, де моделі (такі як автоенкодери або трансформери) навчаються оптимально представляти дані у стисненій формі.

Отже, методи стиснення даних утворюють багатогранну дисципліну, від фундаментальних статистичних та словникових алгоритмів до адаптивних гібридних технологій, де окремою цікавою областю є безвтратні підходи для апаратних платформ на кшталт FPGA, орієнтованих на реальний час та оптимізацію ресурсів.

1.2. Архітектура та принципи функціонування FPGA

FPGA (Field Programmable Gate Array – програмована вентильна матриця) являють собою клас інтегральних схем, архітектура яких дозволяє користувачеві реконфігурувати логічну структуру після виробництва. Ця гнучкість позиціонує FPGA як проміжний елемент між універсальними процесорами (CPU, GPU) та жорстко заданими ASIC, поєднуючи апаратну продуктивність з можливістю перепрограмування для специфічних задач.

Розвиток FPGA починався з простих програмованих матриць (таких як PLA або PAL) і дійшов до SoC систем, де програмована логічна матриця поєднується з процесорним ядром.

За типами пам’яті сучасні FPGA поділяються на три основні класи:

- SRAM-based (найпоширеніші, наприклад, Intel, Xilinx) – дозволяють швидку реконфігурацію, але потребують зовнішньої пам’яті для збереження бітстріму;

- Flash-based (Microchip/Actel) – зберігають конфігурацію постійно, мають коротший час запуску;

- Antifuse-based – одноразово програмовані, але забезпечують високу щільність і радіаційну стійкість.

Ці класи визначають сфери їх застосування: SRAM використовується в комерційних або наукових системах, Flash у промислових, а Antifuse в свою чергу у військових або космічних.

Архітектурна основа FPGA складається з масиву програмованих логічних блоків (Configurable Logic Blocks), пов'язаних через програмовану комутаційну мережу (interconnect network) [13]. Логічні блоки здатні імітувати будь-які булеві функції, тоді як комутаційна мережа забезпечує динамічне формування шляхів з'єднань. Додатково інтегровані спеціалізовані ресурси: блоки пам'яті, арифметичні множники, DSP-модулі, генератори фаз та інтерфейси вводу-виводу.

Загалом, архітектура FPGA є ієрархічною: базові логічні елементи об’єднуються у блоки, які формують «тканину». Кожен логічний блок типово включає таблицю пошуку (Look-Up Table, LUT) для реалізації комбінаційної логіки, тригери (flip-flops) для послідовної логіки та допоміжні елементи маршрутизації. LUT функціонує як масив пам’яті, що зберігає таблицю істинності для заданої функції, дозволяючи синтезувати складні комбінаційні схеми. Тригери забезпечують збереження станів між тактами годинника, що дає можливість створення послідовних автоматів.

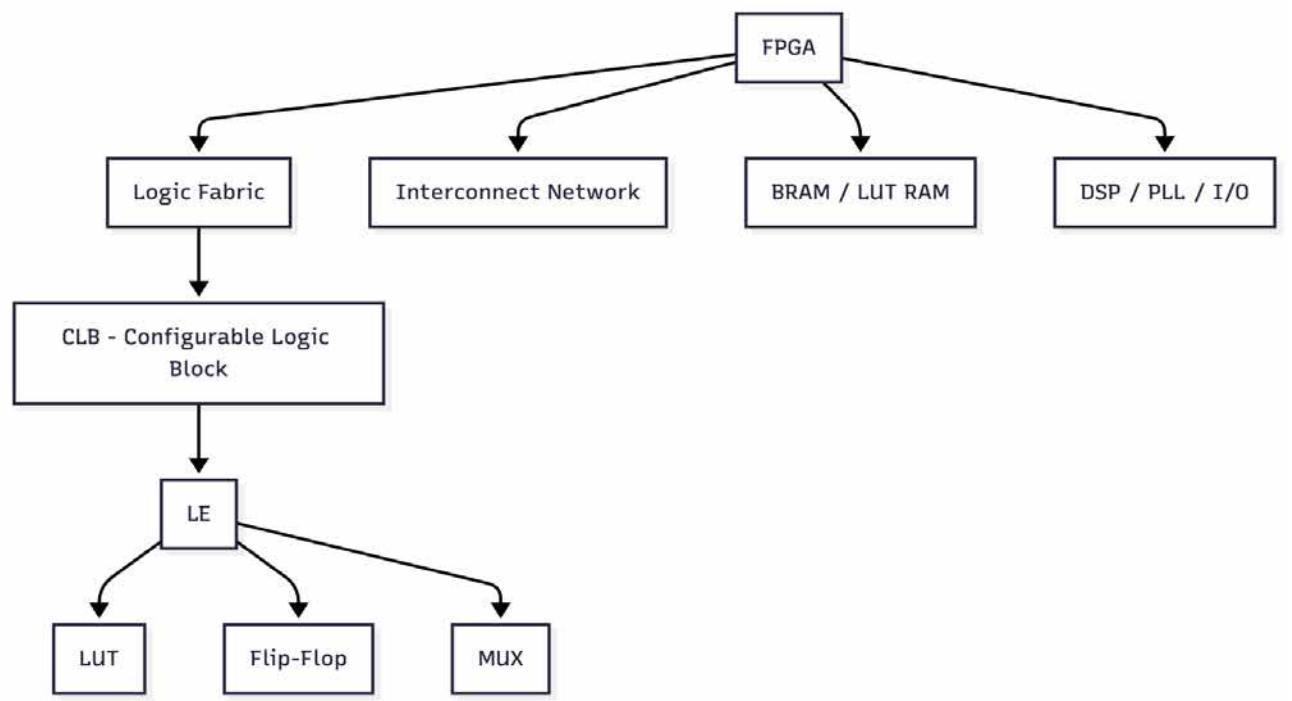


Рис. 1.2 Ієрархічна структура FPGA

Комутаційна мережа, складена з програмованих комутаторів та маршрутизаторів, є критичним компонентом, оскільки визначає ефективність зв’язку між блоками. Під час етапу синтезу, розміщення та трасування інструменти автоматизованої розробки топологію з'єднань оптимізується для мінімізації затримок та споживання ресурсів.

FPGA містить декілька різних видів пам’яті: розподілена пам'ять, яка реалізована на базі LUT для малих обсягів, та блочна RAM для високопродуктивного доступу до більших масивів. BRAM корисні для буферів, кешів чи словників у алгоритмах стиснення. DSP-блоки (Digital Signal Processing) оснащені апаратними множниками, суматорами та акумуляторами, що прискорюють арифметичні операції без навантаження на логічні ресурси – це актуально для арифметичного кодування чи обробки з використанням сигналів.

Таблиця 1.2

Типи пам’яті FPGA

Тип пам’яті	Реалізація	Обсяг	Призначення	Особливості
LUT RAM	усередині логічних блоків	мала (до кількох Кб)	таблиці істинності, кеші	висока швидкодія
BRAM (Block RAM)	окремі модулі	середній (до сотень Кб)	буфери, словники	двопортовий доступ
URAM (Ultra RAM)	у нових FPGA	великий (до МБ)	кадри зображень, великі таблиці	нижча частота, висока щільність
DDR / SRAM (зовнішня)	зовнішні чіпи	гігабайти	великі масиви даних	нижча частота, високий об’єм

Системи управління тактуванням (Clock Management Tiles, PLL, MMCM) генерують стабільні сигнали різних частот, підтримуючи множинні незалежні тактові домени. Це критично для гетерогенних систем, де підсистеми (наприклад, приймач даних та модуль стиснення) працюють на різних швидкостях.

Інтерфейси вводу-виводу поділяються на програмовані GPIO та високошвидкісні протоколи (LVDS, PCI Express, Ethernet, DDR), забезпечуючи інтеграцію з зовнішніми компонентами - мікроконтролерами, сенсорами чи хост-системами. У задачах стиснення часто застосовуються UART чи USB для потокової передачі даних між FPGA та ПК.

Для взаємодії між модулями в SoC архітектурах використовується універсальний протокол розроблений компанією ARM – *Advanced eXtensible Interface (AXI)*. Має декілька варіацій, а саме AXI-Lite для керування, AXI-Stream для роботи з потоками даних. Також, для передачі великих обсягів даних застосовується Direct Memory Access (DMA) контролер, який дозволяє знизити навантаження на процесор і дає можливість напряму обмінюватись даними між пам’яттю та апаратною логікою.

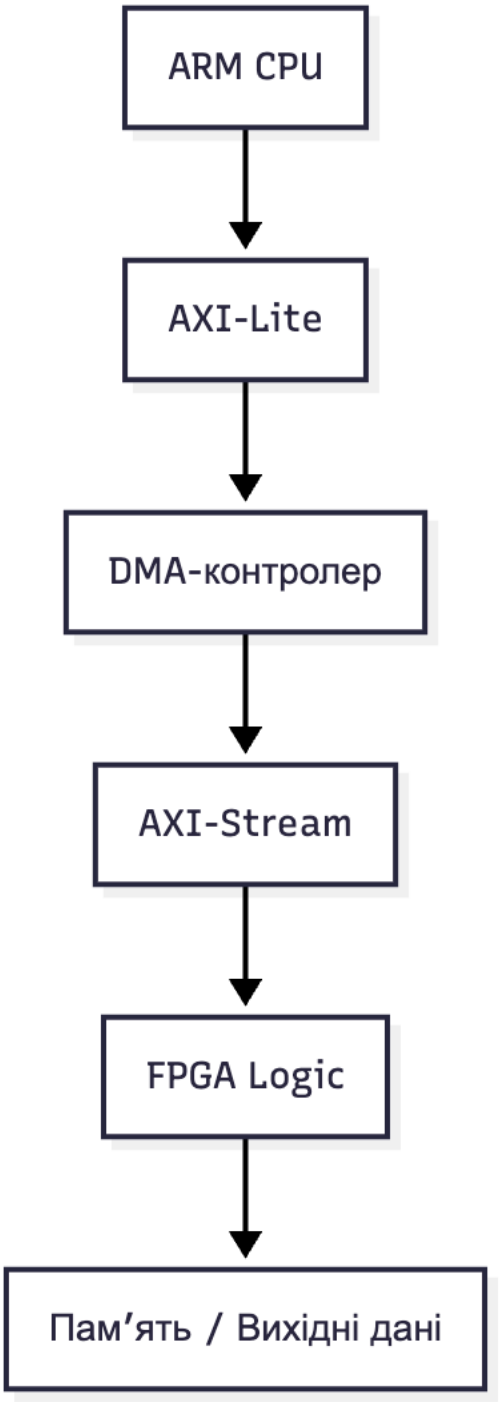


Рис.1.3 Взаємодія між процесорним і апаратним середовищами у SoC.

Принцип функціонування FPGA спирається на паралелізм: на противагу послідовному виконанню інструкцій у процесорах, FPGA дублює логічні елементи та канали даних для одночасної обробки великої кількості операцій. Це реалізується за допомогою конвеєрні архітектури, де фрагменти потоку обробляються паралельно, забезпечуючи високу пропускну здатність та

мінімальні затримки – ключові для реального часу в телекомунікаціях, відеостисненні чи криптографії.

Цей паралелізм можна реалізувати на різних рівнях:

- рівень даних – одночасна обробка незалежних потоків;
- рівень інструкцій – конвеєри для послідовних операцій;
- рівень задач – розподіл функціональних блоків для між різними задачами.

Розробка під FPGA складається з таких етапів: опис логіки на HDL-мовах (VHDL, Verilog), синтез у RTL, розміщення-трасування та генерацію бітового потоку (bitstream) для конфігурації. Після завантаження FPGA виконує апаратну логіку автономно. Сучасні моделі, як Cyclone V від Intel (Altera), об’єднуються з ARM ядрами (наприклад, Cortex A-9), формуючи гібридні SoC, де програмне забезпечення відповідає за керування, а апаратна логіка – за інтенсивні обчислення, що допомагає балансувати гнучкість та продуктивність.

1.3. Особливості апаратної реалізації алгоритмів стиснення

Апаратна реалізація алгоритмів стиснення на FPGA принципово відрізняється від програмних аналогів на процесорах високим ступенем паралелізму, передбачуваною низькою затримкою та стабільною пропускною здатністю, незалежною від динаміки навантаження чи обсягу даних. Логіка алгоритму фактично формує фізичну схему, що дає простір для оптимізації під конкретні вимоги.

Перехід від програмної до апаратної реалізації складається з декількох етапів:

1. Формалізація алгоритму – перетворення його в модель, придатну до апаратної реалізації (визначення циклів, умов, потоків даних).
2. Опис апаратної логіки.
3. Апаратна оптимізація – розбиття на конвеєрні стадії, розпаралелювання обчислень, визначення типів пам’яті.
4. Верифікація та синтез – перевірка логіки, часового аналізу та використання ресурсів.

5. Тестування на реальному потоці даних.

Алгоритми класифікуються на статистичні (Huffman, арифметичне та рангове кодування з ймовірнісним аналізом частот) та словникові (LZ77, LZ78, LZW з пошуком шаблонів), з пріоритетом останніх для FPGA через придатність до конвеєризації. Модифікації LZ4 чи Deflate оптимально адаптуються, оскільки їх структуру можна розбити на послідовні етапи: зчитування вхідного потоку, пошук у словнику, генерацію токенів та запис у буфер.

Конвеєризація (pipelining) є ключовою особливістю: алгоритм фрагментується на незалежні стадії, кожна з яких обробляє дані паралельно з попередніми [15]. Як приклад, у LZ77, етапи зчитування, пошуку збігів, токенізації та виводу функціонують синхронно, забезпечуючи безперервний потік з максимальною утилізацією ресурсів. Конвеєризація дозволяє отримати постійну пропускну здатність яка не залежатиме від розміру вхідних даних.

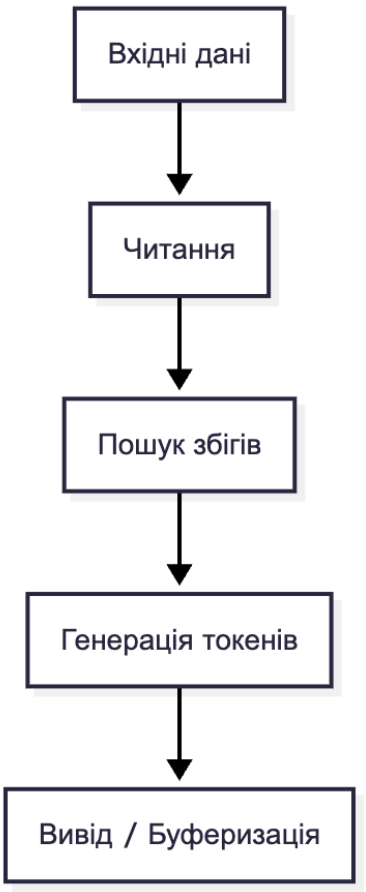


Рис. 1.4 Конвеєрна структура LZ77

Архітектура пам'яті в FPGA підтримує паралельний доступ: словники LZ77 розподіляються по банках BRAM чи розподіленої RAM, дозволяючи пошук збігів за 1–2 такти проти сотень циклів на CPU. Обмеження ресурсів (кількість LUT, BRAM, DSP, ліній I/O) вимагають йти на компроміси: зростання словника посилює компресію, але нарощує споживання пам'яті та ускладнює маршрутизацію, вимагаючи емпіричної оптимізації.

Потокова обробка (streaming) є стандартом для сучасних FPGA (наприклад, Cyclone V), де дані надходять та виводяться послідовно без повного буферування, мінімізуючи затримки для обробки в реальному часі – відеопотоків чи транзакцій. Паралелізація на рівні блоків масштабує продуктивність: множинні конвеєри (наприклад, 4×200 МБ/с = 800 МБ/с) досягають лінійного приросту при контрольованому енергоспоживанні.

Енергоефективність апаратних реалізацій перевершує CPU/GPU у 5–10 разів за метрикою байтів на джоуль, оскільки уникаються накладні витрати на декодування команд, кешування чи переривання [16]. Синхронізація потоків реалізується протоколами на кшталт ready/valid (AXI-Stream), запобігаючи переповненню буферів та простою модулів при нерівномірному вхідному трафіку.

Фіксована затримка, визначена кількістю тактів, забезпечує передбачуваність, критичну для систем реального часу. Модульність дозволяє інтеграцію компресорів у SoC з ARM-ядрами: процесор обробляє метадані, FPGA в свою чергу є ядром стиснення. Часткове перепрограмування (partial reconfiguration) уможливлює динамічну адаптацію алгоритмів під тип даних чи ресурси без зупинки системи.

Складнощами апаратної реалізації є оптимізація ресурсів, організація потоків даних та балансування складності логіки з продуктивністю. Правильно спроєктована система стиснення на FPGA досягає продуктивності, що значно перевищує програмні рішення, з акцентом на енергоефективність та реальний час.

РОЗДІЛ 2. АНАЛІЗ ЗАДАЧІ ТА ПРОЕКТУВАННЯ АПАРАТНО-ПРОГРАМНОЇ СИСТЕМИ СТИСНЕННЯ

2.1. Постановка задачі та визначення вимог до системи

Збільшення обсягів цифрової інформації у всіх сферах людської діяльності призводить до необхідності використання ефективних методів зберігання та передачі даних. Особливо це стосується систем, які працюють із великими потоками інформації у реальному часі – фінансових, телекомунікаційних, сенсорних мереж, мультимедійних систем тощо, де передача навіть невеликої кількості додаткових даних може збільшити затримки. Одним із ключових способів зменшення обсягів переданих або збережених даних є стиснення, що дозволяє перетворити початкову інформацію у компактнішу форму без істотної втрати змісту.

Необхідність використання ефективних методів зберігання та передачі даних зумовлена збільшенням обсягів інформації яка генерується у всіх сферах людської діяльності.

Більшість практичних застосувань стиснення даних виконується програмно на центральному процесорі. Але при великому об’ємі або високих вимогах до швидкодії ці реалізації часто можуть стати «вузьким місцем», оскільки існує оверхед між програмою і процесором який виконує інструкції. Навіть з використанням процесорів з великою кількістю ефективність існуючих рішень буде обмежена послідовним виконанням інструкцій, прошарком між процесором і програмою, а також високим енергоспоживанням. Це викликає інтерес до використання апаратних рішень для стиснення, оскільки в цьому випадку можна досягти більшої швидкості завдяки паралелізму, а також знизити енергоспоживання.

Польові програмовані вентильні матриці (FPGA) є перспективною платформою для створення таких рішень, оскільки дають можливість

конфігурувати структуру обчислювального пристрою під конкретний алгоритм. Завдяки цьому, алгоритм буде виконувати обраний алгоритм з низькими затримками і високою пропускною здатністю.

Цікавою підгрупою є SoC FPGA, які являють собою інтегроване рішення, що складається з процесорного ядра та FPGA в одному чипі. Використання таких рішень додає більшої гнучкості системам. Наприклад, частину обчислень можна виконувати програмно, а іншу – апаратно, при цьому маючи можливість швидкого обміну даними між цими блоками.

Метою даної роботи є розробка та дослідження програмно-апаратної системи стиснення даних. Для реалізації системи було обрано платформу Terasic DE10-Nano, яка побудована на базі Cyclone V SoC з ARM процесором (2 ядра Cortex A9). Це дозволить використати переваги FPGA та поєднати їх з програмною логікою.

Для досягнення поставленої мети необхідно виконати такі завдання:

- Проаналізувати методи та алгоритми стиснення даних з погляду їхньої придатності до апаратної реалізації.
- Обрати алгоритм, який забезпечує прийнятний баланс між ефективністю стиснення та апаратною складністю.
- Розробити структуру програмно-апаратної системи, що реалізує вибраний алгоритм.
- Забезпечити передачу даних між ПК (або вбудованим ARM-процесором) і FPGA-частиною.
- Створити програмні засоби для тестування, передачі даних та аналізу ефективності роботи системи.
- Провести експериментальне дослідження продуктивності розробленої системи та порівняти результати з програмною реалізацією на CPU.

Система має забезпечувати виконання таких основних функцій:

- прийом вхідних даних із зовнішнього джерела (ПК або ARM-ядра);
- стиснення потоку даних на апаратному рівні згідно з обраним алгоритмом;
- передавання стиснених даних у зовнішню систему або до пам’яті для подальшої обробки;
- можливість зміни параметрів алгоритму без необхідності повної перепрошивки FPGA;
- контроль правильності роботи модуля стиснення та виявлення помилок передавання даних.

До апаратно-програмної системи висуваються такі технічні вимоги:

- використання FPGA-плати Terasic DE10-Nano як основної платформи реалізації;
- реалізація логічної частини на мові Verilog HDL або VHDL у середовищі Intel Quartus Prime або на високорівневій мові з використанням HLS;
- можливість симуляції та відлагодження функціонування схеми;
- передбачено логування результатів експериментів та засоби вимірювання часу обробки.

Результатом виконання роботи має стати працездатна програмно-апаратна система, яка виконує стиснення даних на апаратному рівні та дозволяє оцінити приріст продуктивності порівняно з програмною реалізацією.

У процесі дослідження очікується визначення таких характеристик:

- пропускна здатність системи при різних обсягах даних;
- середній коефіцієнт стиснення;
- затримка обробки при послідовному та потоковому режимах роботи;
- використання ресурсів FPGA (кількість логічних елементів, регістрів, блоків пам’яті);
- відносне енергоспоживання при апаратній реалізації порівняно з CPU-варіантом.

Отримані результати можуть бути застосовані для створення вбудованих систем попередньої обробки інформації у мережах датчиків, у промислових контролерах, а також у фінансових чи телеметричних системах, де важливим є поєднання високої швидкодії з обмеженим енергоспоживанням.

Таким чином, розробка програмно-апаратної системи стиснення даних на базі SoC FPGA є актуальним напрямом, який дозволяє поєднати гнучкість програмного забезпечення з перевагами високошвидкісної паралельної обробки на апаратному рівні.

2.2. Вибір алгоритму стиснення для апаратної реалізації

Одним з етапів проєктування системи є визначення алгоритму, що забезпечить баланс між ефективністю стиснення та складністю реалізації. Також важливо врахувати узгодження взаємодії процесів між апаратною та програмною частинами та наявні апаратні ресурси.

Оскільки для розроблюваної системи було обрано безвтратний алгоритм стиснення, подальший аналіз проводитиметься лише для них.

2.2.1. Критерії вибору алгоритму

Для ефективної апаратної реалізації потрібно визначити технічні та функціональні критерії, які визначать баланс між продуктивністю, доступними ресурсами та якістю стиснення. Основними критеріями для вибору алгоритму стиснення у контексті апаратної реалізації на FPGA було визначено наступне:

1. Складність алгоритму – визначає необхідні апаратні ресурси (кількість логічних елементів, тригерів, обсяг пам’яті BRAM тощо).
2. Можливість паралельної обробки – здатність алгоритму ефективно використовувати апаратну конвеєризацію або обробку кількох блоків одночасно.
3. Часова ефективність – затримка між поданням вхідних даних та отриманням результату.
4. Ступінь стиснення – відношення розміру вхідних даних до розміру стисненого потоку.

5. Придатність до апаратної реалізації – наявність існуючих прикладів або спрощених схем для втілення в логіці FPGA.

Враховуючи обрані критерії, логічним буде розглянути і порівняти наступні з поширених алгоритмів: RLE (Run-Length Encoding), Huffman, LZ77, LZW та DEFLATE.

2.2.2. Аналіз основних алгоритмів

Алгоритм RLE є одним із найпростіших методів стиснення даних без втрат [4]. Логіка його роботи полягає в заміні довгих послідовностей символів на коротші з вказуванням довжини. Наприклад, послідовність AAAAABBBCC може бути представлена як (A,5)(B,3)(C,2).

Його перевагою є простота реалізації (як програмної, так і апаратної). Він може бути реалізований за допомогою нескладної комбінаційної логіки та лічильників, не потребуватиме великого об’єму пам’яті або змін станів. Також цей метод дозволяє обробляти потік даних у реальному часі, що робить актуальним його застосування в для потокових даних або телеметричних систем.

Недоліком є низька ефективність для даних, де повтори трапляються рідко або взагалі відсутні. У таких випадках розмір стисненого файлу може навіть перевищувати оригінальний.

Можна зробити висновок, що RLE доцільно використовувати у системах, де дані містить довгі послідовності повторів – наприклад, у зображеннях з великими однорідними ділянками або при стисканні структурованих сигналів.

Кодування Гаффмана – це статистичний метод, який використовує змінну довжину кодів залежно від частоти появи символів. Символи, які зустрічаються частіше, отримують коротші коди, а рідкісні – довші. У результаті середня довжина коду зменшується, що приводить до ефективного стиснення.

Перевагою є високий коефіцієнт стиснення для нерівномірно розподілених даних, але для реалізації необхідне попереднє формування дерева частот, що ускладнює апаратну реалізацію. Також складність декодування зростає, оскільки потрібно динамічно реконструювати дерево або зберігати його у пам’яті. Через

це Huffman-кодування частіше застосовується як частина комбінованих методів, наприклад у DEFLATE чи JPEG, а не як окреме апаратне рішення.

Алгоритми сімейства Lempel–Ziv (зокрема LZ77, LZ78 та LZW) ґрунтуються на виявленні повторюваних підрядків у потоці даних. Вони замінюють повтори посиланнями на попередні вхідні дані. Наприклад, LZ77 працює з «вікном пошуку», де зберігаються попередні символи, і записує посилання у вигляді трійки (зсув, довжина, символ).

Перевагою LZ77 є кращий коефіцієнт стиснення для неоднорідних даних порівняно з RLE, а також відсутність потреби у попередньому аналізі частот, як у кодуванні Гафмана. Його можна застосувати для потокової обробки даних, і, відповідно, стискання даних в реальному часі, аде для цього необхідно правильне проектування буфера вхідних даних.

Апаратна реалізація складніша ніж для RLE, оскільки потребує пошуку збігів у вікні, для чого необхідно застосовувати компаратори або додаткову пам’ять, що, в свою чергу, збільшує використання наявних ресурсів [17].

Алгоритми LZ77, LZ78 розвинулись у LZW, який використовує словник, що заповнюється підрядками, замість посилань. Цей підхід часто дає вищий коефіцієнт стиснення, але використання словника (побудова та оновлення) потребує більшого об’єму пам’яті. Саме тому цей алгоритм рідше реалізується саме апаратно.

DEFLATE це комбінований алгоритм, що поєднав в собі переваги LZ77 та кодування Гаффмана [18]. Він використовується у форматах ZIP та PNG і забезпечує високий ступінь стиснення при прийнятній швидкодії. Але ця комбінована схема є істотно складнішою в апаратній реалізації через те, що потребує і пошуку збігів, і побудови дерева. Для системи апаратного стиснення DEFLATE є занадто ресурсоємним.

Таблиця 2.1

Порівняння основних алгоритмів стиснення

Алгоритм	Принцип роботи	Складність реалізації	Можливість паралелізму	Ступінь стиснення	Типові застосування
RLE	Замінює послідовності однакових символів парою (символ, кількість)	Дуже низька	Висока	Низький–середній	Зображення, телеметрія
Huffman	Кодує символи за частотою появи	Висока	Середня	Високий	Архіватори, комбіновані схеми
LZ77	Заміна повторів посиланнями на попередні дані у «вікні»	Середня–висока	Висока	Високий	Потокове стиснення
LZW	Створює словник підрядків	Висока	Низька–середня	Високий	TIFF, GIF, PDF
DEFLATE	Комбінація LZ77 + Huffman	Дуже висока	Середня	Дуже високий	ZIP, PNG

2.2.3. Обґрунтування вибору RLE та LZ77.

З огляду на проведений аналіз та визначені раніше критерії, для реалізації в системі було обрано RLE та LZ77. Вони не потребують багато ресурсів плати, але водночас мають різну складність реалізації, що дозволить порівняти простий і просунутий алгоритми в апаратній реалізації.

RLE буде використано як простіший варіант, який дозволить оцінити мінімально можливі витрати ресурсів FPGA та часові характеристики простої конвеєрної архітектури. Це допоможе перевірити коректність моделі обміну даними між ПК та FPGA, а також забезпечить основу для тестування інтерфейсів передачі даних.

LZ77 є більш просунутим алгоритмом, що демонструє вищий ступінь стиснення. Його реалізація потребує створення буфера та логіки для пошуку збігів у «ковзному вікні». Це дозволить дослідити питання ефективності використання пам’яті FPGA та швидкодії при зростанні складності алгоритму.

Обидва алгоритми підходять для програмно-апаратної реалізації, тобто частину обчислень (наприклад, буферизацію чи підготовку даних) можна виконати на процесорі ARM, а основна логіка стиснення – у програмованій логіці. Це залишає простір для подальшої оптимізації системи та аналізу продуктивності SoC-рішень.

2.3. Вибір інструментальних засобів розробки та FPGA-плати

Ще одним важливим етапом проєктування є вибір апаратної платформи та відповідних засобів розробки. Прийняті рішення визначають зручність розробки та відлагодження, технічні можливості розроблюваної системи, складність реалізації алгоритмів та ефективність програмно-апаратної взаємодії.

Отже, після визначення мети роботи, необхідно вибрати платформу, що забезпечить достатню обчислювальну потужність, наявність процесорного ядра для організації обміну з ПК, підтримку сучасних інтерфейсів введення/виведення та доступність необхідних інструментів розробки.

2.3.1. Критерії вибору FPGA-плати.

Основними критеріями при виборі FPGA-плати для даного проєкту є:

1. Наявність вбудованого процесора або можливість інтеграції з ним. Це суттєво спрощує реалізацію апаратно-програмних систем, у яких частина логіки виконується в ПЛІС, а частина – на CPU. Такий підхід дозволяє розподілити завдання стиснення між програмним та апаратним рівнями.

2. Обсяг логічних ресурсів (LUT, FF, BRAM, DSP). Для реалізації ефективних алгоритмів стиснення потрібні більші обсяги пам’яті (наприклад, для буферів, словників, дерев тощо), а також арифметичні блоки для підрахунків та пошуку збігів.

3. Наявність зручних інтерфейсів обміну з ПК. Важливим є доступ до USB, UART, Ethernet або інших інтерфейсів, що дозволяють передавати вхідні та стиснені дані між ядром стиснення та клієнтом.

4. Підтримка сучасних інструментів розробки та документації. Наявність зручних інструментів розробки, офіційних бібліотек, прикладів, сумісності з

мовами високого рівня (C, OpenCL, HLS) дає можливість підвищити ефективність розробки.

Таким чином, оптимальна плата розробки для проведення дослідження має поєднувати апаратні можливості FPGA з процесорним ядром для керування потоками даних, а також мати достатню кількість інструментів для швидкої розробки та відлагодження.

2.3.2 Огляд існуючих плат розробки

Основними гравцями на ринку FPGA є Altera (зараз частина Intel) та Xilinx (зараз частина AMD) [13]. Вони дають доступ до великої кількості одноплатних комп'ютерів різних конфігурацій. Популярними є рішення від Terassic та Digilent, тому, враховуючи критерії, слід обирати з них.

Terasic DE10-Nano створена на базі Cyclone V [19]. Плата поєднує програмовану логіку з двоядерним процесором ARM і таким чином являє собою SoC FPGA. Вона має 1ГБ DDR3 пам'яті, велику кількість інтерфейсів зв'язку, таких як USB, UART, Ethernet, має слот для microSD карти пам'яті і велику кількість GPIO. Також має хорошу документацію, підтримку Linux і дозволяє швидко налаштувати обмін даними між ARM процесором і FPGA використовуючи HPS to FPGA bridge. Для розробки під неї використовується Intel Quartus, що є хорошим вибором через достатньо велику спільноту розробників. В цілому, плати від Terassic добре зарекомендували себе в освітніх і наукових проєктах, тому вони є хорошим вибором для розроблюваної системи.

Digilent Basys 3 є платформою початкового рівня що базується на Xilinx Artix 7. Вона не має вбудованого процесорного ядра, має менший об'єм пам'яті, меншу кількість периферії. Є хорошим варіанто для вивченням основ цифрової логіки.

Продовжуючи огляд платформ, що базються на рішеннях Xilinx варто розглянути Digilent ZedBoard, що побудована на базі Xilinx Zynq-7020 [20]. Вона, як і Terasic DE10-Nano, має двоядерний ARM процесор і відноситься до класу SoC FPGA. Так само підтримує Linux, має велику кількість периферії (навіть

містить HDMI) та великий об’єм пам’яті у 512 МБ. Апаратна частина є більш продуктивною, але в свою чергу це збільшує вартість.

Ще однією цікавою платформою є рішення від Lattice Semiconductors. Вони орієнтовані на компактність та низьке енергоспоживання. Одним з популярних представників є Lattice ECP5, який можна зустріти в платах ULX3S або Colorlight 5A-75E [21]. Цим платам характерний баланс вартості і продуктивності та підтримка відкритих інструментів проектування (таких як Yosys, nextpnr). Проте, ці плати зазвичай мають меншу кількість логічних блоків ніж їх прямі конкуренти, що робить їх вибір менш ймовірним.

Критичним у використанні рішень від AMD є неможливість встановити офіційне програмне забезпечення для розробки, через санкції накладені США у 2014 році через анексію Криму та окупацію частин Донецької та Луганської областей російською федерацією.

Таблиця 2.2

Зведена таблиця порівняння розглянутих плат розробки

Плата	FPGA-чип	CPU (SoC)	Пам’ять	Інтерфейси	Інструменти розробки	Особливості
Terasic DE10-Nano	Intel Cyclone V SE 5CSEBA6U23I7	2× ARM Cortex-A9 (HPS)	1 ГБ DDR3	USB, UART, Ethernet, GPIO, microSD	Intel Quartus Prime, Platform Designer, HLS	SoC, висока сумісність, хороша докумен-тація
Digilent Basys 3	Xilinx Artix-7 XC7A35T	-	512 КБ SRAM	USB-UART, GPIO	Vivado	Початковий рівень, без CPU
Digilent ZedBoard	Xilinx Zynq-7020	2× ARM Cortex-A9	512 МБ DDR3	HDMI, Ethernet, USB, UART	Vivado, PetaLinux	Висока продуктив-ність
ULX3S (Lattice ECP5)	Lattice ECP5-85F	-	-	USB, microSD, Wi-Fi	Yosys, nextpnr (open-source)	Низьке енергоспо-живання, відкрите ПЗ

Враховуючи раніше визначені критерії та особливості розглянутих вище плат розробки, можна зробити висновок що Terasic DE10-Nano є оптимальним

варіантом для використання. Належнсіть до класу SoC FPGA, великий об’єм оперативної пам’яті. виclfdока кількість периферії, хороша документація, зручне програмне забезпечення і доступність роблять її хорошим вибором для освітніх та наукових проєктів.

2.4. Архітектурне проектування системи

Одним з ключових етапів розробки будь якої системи є її архітектурне проєктування. Для розроблюваної системи тут будуть сформовані вимоги до програмної та апаратної частин, структури потоків даних та методів синхронізації.

Важливо не лише визначити внутрішню архітектуру обчислювальних модулів, а й забезпечити узгоджену взаємодію з компонентами системи вищого рівня, включно з програмним керуванням, інтерфейсами передавання даних та потенційною інтеграцією з платформами типу SoC FPGA. Оскільки у наступних підрозділах буде розглянуто моделі обміну даними між ПК та FPGA (2.5), а також особливості використання SoC FPGA (2.6), у цьому підрозділі буде описано загальний дизайн системи та її компонентів.

Особливу увагу буде зосереджено на структурі внутрішніх блоків, їх взаємодії та організації потоків даних. Деталі комунікаційних протоколів та специфіка SoC-плат розкриваються у подальших розділах.

2.4.1. Загальна структура та рівні системи.

Система складається з двох основних рівнів:

1) апаратний рівень (FPGA) – виконує обчислювально інтенсивні операції стиснення та буферизації, забезпечує низьку затримку та високу пропускну здатність завдяки паралельності апаратних модулів;

2) програмний рівень (ПК або вбудований контролер) – управління, моніторинг, ініціалізація, передавання та приймання даних, післяобробка.

Це дозволить гнучко оновлювати програмну частину, масштабувати обчислювальні ресурси FPGA, розділяти відповідальність між алгоритмічною та керуючою логікою. У наступному підрозділі 2.5 буде детально розглянуто

моделі обміну даними між цими двома блоками, включно з особливостями синхронізації та організацією протоколів.

2.4.2. Модульна архітектура FPGA.

Архітектура FPGA проєкту будується за модульним принципом. Основні складові:

1. Модуль приймання даних (Input Interface). Він відповідає за прийом потоків даних ззовні, синхронізацію швидкості обробки, попередню буферизацію (FIFO), вирівнювання потоків. У залежності від обраної моделі обміну, він може реалізовувати протоколи USB, PCIe, UART, Ethernet або DMA-доступ.

2. Модуль попередньої обробки. Його задача полягає у формуванні блоків, підготовці даних до конвеєрної обробки, нормалізації потоку, керуванні буферами. Це є важливим пунктом при використанні алгоритмів з ковзним вікном (таких як LZ77).

3. Компресійне ядро (Compression Core). Обчислювальна частина системи. Містить в собі модуль пошуку збігів (для LZ77), лічильники довжин (для RLE); керує конвеєризацією операцій, відповідає за мінімізацію шляху між логічними блоками для зменшення затримок. Цей модуль повинен мати можливість конфігурації, щоб у подальших експериментах можна було змінювати параметри (такі як ширину шини або глибину конвеєра)

4. Модуль форматування та упаковки результатів. Його задача полягає в підготовці і перетворенні даних для повернення, оптимізації вихідного потоку.

5. Вихідний інтерфейс. Забезпечує повернення даних у програмну частину. Його реалізація може варіюватися відповідно до обраної архітектури обміну даними.

2.4.3. Потоки даних, структурні залежності та синхронізація.

Організація потоків даних визначає наскільки ефективно працюватиме система. Архітектурні рішеннявключають конвеєризацію обчислювальних етапів, використання BRAM/FIFO для вирівнювання потоків, поділ задач за тактовими областями, ізоляцію критичних шляхів. Важливим пунктом є

узгодження швидкості вхідних і вихідних потоків. Якщо інтерфейс між ПК та FPGA має змінну пропускну здатність (наприклад USB або UART), FPGA повинна мати достатньо буферів для компенсації нерівномірності.

Це питання буде розкрито значно детальніше у наступному підрозділі (2.5), де аналізуються моделі обміну, включно із блоковим, потоковим і пакетним режимами передачі. Для високопродуктивних інтерфейсів (PCIe або DMA у SoC FPGA) потік може бути майже безперервним, що впливає на архітектурні рішення всередині FPGA, особливо на розмір буферів та конвеєризацію.

2.4.4. Вибір апаратної платформи та її ресурсні обмеження.

Вибір FPGA-плати визначає обсяг доступних ресурсів (LUT, FF, BRAM), максимальну тактову частоту, кількість доступних інтерфейсів, можливість використання SoC-архітектури. Платформи поділяються на: 1) Класичні FPGA (наприклад, Artix-7, Cyclone V). Проте взаємодія з ПК потребує окремого контролера або зовнішнього модуля. 2) SoC FPGA (Zynq-7000, Cyclone V SoC). Інтегрований ARM-процесор дозволяє організувати обмін даними через DMA, MMIO або інші високошвидкісні канали. Особливості використання таких платформ будуть окремо розглянуті у підрозділі 2.6.

Обмеження платформи накладають вимоги на архітектуру FPGA: кількість BRAM визначає можливий розмір буферів; пропускна здатність інтерфейсів визначає максимальний throughput; кількість логічних елементів визначає складність реалізації алгоритму.

2.4.5. Взаємодія програмної та апаратної частини.

У загальній архітектурі програмна частина виконує такі функції: керування сесією стиснення, передавання блоків даних і отримання результатів, логування та моніторинг, налаштування параметрів FPGA (розмір блоку, пороги, режими роботи), післяобробка стиснених даних. Менеджер потоків на ПК повинен взаємодіяти з апаратним рівнем через узгоджений набір інтерфейсів і протоколів, що розкривається у підрозділі 2.5. Особливо важливими є синхронізація станів, передача службових сигналів, керування буферами. У випадку використання SoC FPGA частина цих функцій може бути перенесена всередину ARM-

процесора, що зменшує затримки та спрощує комунікацію. Це стане предметом аналізу у підрозділі 2.6.

2.4.6. Архітектурні альтернативи та їх зв’язок із моделями обміну даними.

Розглянуто три підходи: 1) Повністю апаратна реалізація: максимальна продуктивність, але складність реалізації алгоритму та обміну даними. Вимагає інтерфейсів із постійним потоком (PCIe). Моделі, описані в 2.5, критично важливі для таких систем. 2) Гібридна архітектура (вибрана): обчислення на FPGA, керування на ПК. Підходить для широкого спектра інтерфейсів. Детальний вибір механізму обміну розглядається у 2.5. 3) SoC-орієнтована архітектура: ARM+FPGA дозволяє організувати більш ефективний обмін, мінімізуючи затримки. Архітектурні особливості таких систем розглянуті у 2.6. Таким чином, вибір архітектури безпосередньо залежить від того, як саме організовано обмін даними між компонентами системи.

У цьому підрозділі було сформовано загальну архітектурну модель системи стиснення даних на основі FPGA. Було проаналізовано її структурні елементи, потоки даних, модульну організацію та обмеження платформи. Крім того, підкреслено важливість взаємодії між апаратними та програмними частинами та зазначено, що деталі моделей обміну даними, а також специфіка використання SoC FPGA будуть розглянуті у наступних підрозділах 2.5 та 2.6.

2.5. Модель обміну даними між ПК та FPGA

2.5.1. Загальні вимоги до організації обміну.

Механізм передавання та прийому даних між ПК та FPGA є одним із ключових компонентів системи стиснення. Від його побудови залежить ефективність апаратних модулів, масштабованість системи та можливість адаптації до нових алгоритмів. Оскільки FPGA орієнтована на високошвидкісну потокову обробку, канал даних повинен забезпечувати достатню пропускну здатність, низьку латентність і гарантувати узгодженість даних.

Основними вимогами до моделі обміну є: стабільна передача даних з високою швидкістю; підтримка масштабування для роботи з файлами різного розміру; мінімальна затримка при передаванні, що підвищує ефективність алгоритмів; механізми контролю станів, синхронізації та корекції помилок; проста інтеграція з програмним забезпеченням на стороні ПК.

Крім цього, система повинна підтримувати відновлення обробки після тимчасових збоїв, ефективно управляти потоками даних різної довжини та забезпечувати адаптацію до різних швидкостей вхідного потоку. Для вибору оптимальної моделі обміну необхідно розглянути кілька можливих підходів, оцінити їх переваги та обмеження з точки зору апаратної складності, продуктивності і зручності інтеграції з програмною частиною. Крім базових вимог, важливим аспектом є організація черг передачі даних, балансування навантаження між процесором ARM та обчислювальними ядрами FPGA, а також визначення оптимального розміру блоків, що передаються. Вибір розміру блока впливає на продуктивність стиснення та ефективність використання буферів, тому передача даних має підтримувати адаптивний поділ на блоки та динамічне управління потоками.

2.5.2. Модель 1: UART або USB-UART

UART є одним із найпростіших способів організувати зв’язок між ПК і FPGA [22]. Цей інтерфейс має низьку складність реалізації, доступний на більшості плат і не потребує складної апаратної логіки. Основними завданнями цього інтерфейсу є передача службових сигналів, невеликих конфігураційних пакетів або тестових даних. Проте пропускна здатність UART дуже обмежена: навіть сучасні USB-UART адаптери рідко забезпечують швидкість понад кілька мегабіт на секунду. Тому цей інтерфейс не підходить для передачі великих масивів даних у реальному часі. Його доцільно використовувати лише на етапах розробки та налагодження або для передавання невеликих блоків даних, наприклад конфігураційних або контрольних. UART забезпечує просту синхронізацію: FPGA читає байти за сигналами Ready/Valid, а ПК передає їх послідовно. Ця модель має низку особливостей, які варто враховувати.

Наприклад, при тривалому передаванні великих файлів потрібна апаратна буферизація у вигляді FIFO для компенсації різниці швидкостей між UART і обчислювальним ядром FPGA. Крім того, UART добре підходить для тестування окремих модулів алгоритму стиснення у режимі “step-by-step”, оскільки дані передаються по одному байту, що дозволяє легко відстежувати потік. Також важливо передбачити механізми повторної передачі в разі втрати байтів, оскільки UART не має вбудованих засобів контролю цілісності даних. Попри обмеження швидкості, UART є надійним інструментом для початкового тестування апаратної логіки, перевірки інтерфейсів і відпрацювання базових алгоритмів стиснення на малих обсягах даних.

2.5.3. Модель 2: USB 2.0/3.0

USB є універсальним і широко підтримуваним інтерфейсом. У разі SoC FPGA контролювати USB може ARM-процесор під управлінням Linux, що спрощує інтеграцію з ПК і забезпечує підтримку стандартних драйверів та стеків [23]. USB 2.0 дозволяє передавати десятки мегабайт на секунду, USB 3.0 - сотні мегабайт, що достатньо для потокового або блокового передавання даних для алгоритмів стиснення. Переваги USB полягають у легкості інтеграції та сумісності з існуючими системами. Крім того, USB дозволяє працювати із файлами будь-якого розміру і передавати дані пакетами, що зручно для систем, які працюють із блоковими алгоритмами стиснення. Складність реалізації полягає у необхідності інтеграції драйверів на ARM, що може вимагати певних програмних навичок, проте ARM-процесор значно полегшує цю задачу. Для забезпечення стабільності потоку даних використовуються буфери у DRAM ARM, FIFO у FPGA та контрольні сигнали Ready/Valid. Додатково, USB дозволяє застосовувати мультиплексовані канали передачі для одночасної обробки декількох потоків, що збільшує ефективність системи. Ця модель дозволяє легко адаптувати систему до нових форматів даних і швидко змінювати алгоритми обробки, оскільки обробка файлів на ARM спрощує розділення на блоки і підготовку заголовків для FPGA. Під час реалізації важливо враховувати обмеження пропускної здатності, буферизацію

великих пакетів та корекцію можливих втрат даних за допомогою контрольних сум.

2.5.4. Модель 3: Ethernet

Ethernet забезпечує універсальний і стандартизований підхід для організації обміну даними як у локальній мережі, так і на відстані. ARM у SoC FPGA має готовий Ethernet-контролер, що дозволяє реалізувати TCP/IP стек і обмін файлами без складної логіки FPGA [24]. Переваги Ethernet полягають у віддаленому доступі до FPGA, можливості передавати дані кількома потоками та використанні стандартних протоколів TCP/IP, що спрощує інтеграцію з ПК. Недоліки: затримка вище, ніж у USB або PCIe, що може знижувати ефективність потокового стиснення, особливо для алгоритмів із низькою латентністю. Для забезпечення стабільної роботи необхідно впроваджувати багаторівневу буферизацію і механізми контролю стану потоків, наприклад через FIFO та маркери “готово/зайнято”. Ethernet особливо корисний, коли важливі гнучкість, віддалений доступ і можливість одночасного обміну кількома потоками даних, наприклад для паралельного стиснення різних файлів. Додатково, Ethernet дозволяє застосовувати механізми пріоритезації пакетів, адаптивного контролю швидкості та корекції помилок на рівні протоколів, що підвищує надійність системи при великій кількості одночасних підключень.

2.5.5. Модель 4: PCI Express.

PCIe є високошвидкісним інтерфейсом, що забезпечує низьку латентність і високу пропускну здатність, ідеально підходить для потокового передавання великих масивів даних [25]. Переваги PCIe полягають у прямому доступі до пам’яті, високій швидкості передачі і мінімальному втручанні CPU. Це дозволяє FPGA працювати з даними в реальному часі, не чекаючи на обробку процесором. Недоліки PCIe: складність реалізації, потреба у спеціальних IP-якостях, висока ресурсомісткість. Для малих FPGA або бюджетних SoC цей варіант часто є недоступним. PCIe доцільно застосовувати лише у високопродуктивних рішеннях, де критично важлива максимальна швидкість, наприклад при стисненні потоків відео або великих файлів у реальному часі. При реалізації

через PCIe важливим є планування буферів у FPGA для підтримки максимальної пропускної здатності, використання DMA для прямого доступу до пам’яті і паралельна обробка декількох потоків даних для ефективного використання ресурсів.

2.5.6. Модель 5: ПК – ARM – AXI– FPGA.

Найбільш практичною моделлю для SoC FPGA є використання ARM як проміжної ланки. ПК передає файл або потік даних на ARM через USB, Ethernet або SD-карту. ARM завантажує дані в оперативну пам’ять, виконує попередню обробку, сегментацію блоків і формування заголовків, після чого передає їх у FPGA через AXI-Stream або AXI-Full з використанням DMA. FPGA обробляє дані в реальному часі, виконуючи алгоритм стиснення, і повертає результат назад ARM, який збирає блоки у файл або формує вихідний потік. Ця модель дозволяє ARM керувати потоками даних, синхронізувати роботу з FPGA та застосовувати буферизацію для компенсації нерівномірності вхідного потоку. Вона забезпечує баланс між продуктивністю, гнучкістю та простотою інтеграції. Крім того, використання DMA і AXI забезпечує високу пропускну здатність і стабільність обміну без блокування процесора ARM. Використання DMA дозволяє асинхронно передавати великі обсяги даних, мінімізуючи час простою процесора. Крім того, ARM може виконувати попередню обробку, включно з конвертацією форматів, перевіркою цілісності даних і підготовкою блоків для FPGA, що підвищує ефективність стиснення. Ця модель є основою для масштабованих систем і дозволяє паралельно обробляти кілька потоків даних, зберігаючи високу продуктивність.

2.5.7. Механізми буферизації та синхронізації.

Для всіх моделей обміну важливо використовувати механізми буферизації, які дозволяють уникати переповнень і втрат даних. Це FIFO, BRAM та багаторівневі буфери як у FPGA, так і у ARM. DMA дозволяє асинхронно передавати дані, не блокуючи ARM. FPGA використовує сигнали Ready/Valid, маркери “готово/зайнято” та контроль водяних міток для відстеження стану потоків. При необхідності застосовуються цикли підтвердження, контрольні

суми та CRC для перевірки цілісності даних. Додатково застосовуються адаптивні черги, які дозволяють динамічно регулювати розмір буферів залежно від швидкості надходження даних та завантаження обчислювальних модулів. Також важливим є управління пріоритетами потоків, що дозволяє уникнути блокування критичних даних при обробці декількох завдань одночасно.

2.5.8. Висновки.

Аналіз різних моделей обміну показав, що оптимальною є модель з ARM-процесором як проміжною ланкою. ARM приймає дані від ПК, виконує попередню обробку і передає їх у FPGA через AXI з DMA. FPGA концентрується на високошвидкісному стисненні, що забезпечує баланс між продуктивністю, гнучкістю та простотою розробки. Такий підхід забезпечує масштабованість, стабільність і можливість адаптації до різних обсягів та форматів даних. Він природно підводить до наступного підрозділу 2.6, де будуть детально розглянуті особливості використання SoC FPGA для внутрішнього обміну ARM - FPGA та організації потоків через AXI-шину. Крім того, цей підхід дозволяє гнучко масштабувати систему, додавати нові алгоритми стиснення або нові типи даних без зміни базової архітектури обміну, а також забезпечує стабільну роботу при високих навантаженнях і великих обсягах інформації.

2.6. Особливості використання SoC FPGA (ARM + FPGA) для організації обміну даними

2.6.1. Архітектурна специфіка гібридних систем.

SoC-платформи FPGA поєднують у собі виконання високорівневих програм на ARM-ядрах та можливість апаратної оптимізації на FPGA-логіці. Ключовою перевагою є тісна інтеграція компонентів через високошвидкісні внутрішні канали, що дозволяє мінімізувати затримки та забезпечити більш прогнозовану продуктивність у порівнянні з дискретними рішеннями ARM - FPGA через зовнішні інтерфейси. Архітектура типово включає спільний фізичний доступ до пам’яті, який організований через DDR, а також спеціальні канали типу AXI High Performance, що дозволяють FPGA-логіці читати й

записувати дані безпосередньо, обходячи ARM-ядро. Така симбіотична структура дає можливість створювати складні конвеєри обробки: ARM здійснює високорівневий контроль та IO, а FPGA – обробку в реальному часі.

2.6.2. Стратегії організації пам’яті та буферизації.

У SoC FPGA важливою є не лише швидкість обміну, але й правильна організація пам’яті, оскільки ARM та FPGA зазвичай працюють у спільному адресному просторі, хоча доступ здійснюється різними шляхами. Найчастіше використовується подвійна буферизація (double buffering), коли одна частина пам’яті обробляється FPGA-логікою, а інша готується ARM-ядром. Це дозволяє уникати простоїв та створювати неспинний потік даних.

Ще один підхід – кільцеві буфери, які дозволяють постійно передавати блоки інформації змінної довжини. Конфігурація пам’яті стає критичним фактором у задачах стиснення, де потоки можуть динамічно змінювати свою інтенсивність залежно від алгоритму. Забезпечення синхронізації та правильного розмежування зон доступу запобігає накладанню транзакцій, появі зависань і некоректних читань, що особливо важливо при високих частотах тактування FPGA-ядра.

2.6.3. Взаємодія модулів системи стиснення даних через високошвидкісні AXI-інтерфейси.

У SoC FPGA AXI-шина забезпечує стандартизовану модель передачі даних між ARM і FPGA. Важливими є три режими доступу: AXI General Purpose – для регістерного управління і низьковимірних команд, AXI High Performance – для масивних потокових даних, AXI Accelerator Coherency Port – для задач, що потребують когерентності кеша. Для систем стиснення даних оптимальним зазвичай є AXI HP, оскільки він дозволяє забезпечити передачу великих блоків даних із пропускною здатністю в сотні МБ/с чи навіть кілька ГБ/с (залежно від частоти і ширини інтерфейсу). Тут важливо враховувати, що AXI-інтерфейси можуть працювати у burst-режимі, що знижує накладні витрати на транзакцію: замість частої ініціалізації передається одразу велика послідовність даних. Для

алгоритмів компресії це важливо, оскільки блоки даних зазвичай мають значну довжину.

2.6.4. Синхронізація, контроль та обробка виняткових ситуацій.

В SoC FPGA важливо передбачити чіткі механізми узгодження між ARM та FPGA, оскільки вони працюють у різних доменах (частотних, програмних, функціональних). Сигнали синхронізації (IRQ, семафори, DMA-completion флаги) дають змогу створювати стабільний і контрольований процес руху даних. ARM може ініціювати передачу, готувати конфігураційні параметри, а FPGA – сигналізувати про готовність результатів або завершення чергового блоку.

Додатково система повинна мати механізми обробки виняткових ситуацій, таких як переповнення буфера, втрачені транзакції або тимчасовий “backpressure”, коли FPGA не встигає обробляти нові дані. Реалізація правильної логіки дозволяє уникати збоїв під навантаженням та гарантувати стабільність роботи.

2.6.5. Організація роботи обчислювального навантаження між ARM і FPGA.

Організація обміну між ARM і FPGA прямо залежить від розподілу обов’язків у системі. Частина логіки (наприклад, попередня обробка, фільтрація, розбиття на блоки) може бути більш оптимальною на ARM, особливо якщо потрібна гнучкість або складні умови. Натомість FPGA забезпечує максимальну швидкість і паралельність у задачах, де структура операцій фіксована. ARM також виконує роль диспетчера потоків, контролює послідовність подій і може адаптувати алгоритм у реальному часі – змінювати режим стиснення, розмір блоку, параметри буферизації. У синергії це створює систему, яка одночасно продуктивна і гнучка.

Тут важливо уникати надмірного завантаження ARM обчисленнями – тоді вузьким місцем стане сам процесор. З іншого боку, перенесення занадто великої кількості операцій у FPGA може призвести до складності реалізації та перевитрати ресурсів логіки. Оптимальний баланс досягається експериментально й залежить від характеру вхідних даних.

2.6.6. Переваги та особливості DMA-передач у гібридних системах.

Найефективнішим механізмом передачі між ARM та FPGA є DMA (Direct Memory Access), оскільки він дозволяє передавати дані без втручання ARM-ядра після початкової конфігурації. DMA-блоки можуть бути апаратно інтегрованими або реалізованими на FPGA-логіці. Важливими параметрами є розмір транзакцій, кількість каналів, можливість роботи у двох напрямках одночасно. DMA-канали дозволяють ARM-ядру більше зосередитись на керуванні та аналітиці, ніж на обробці байт-потоку. Проте потрібно враховувати, що ініціалізація DMA займає час і створює часові “острівки”. У задачах стиснення це не критично, оскільки передаються великі блоки, але для потоків із дрібними пакетами це може стати перепоною.

Деякі SoC FPGA підтримують scatter-gather DMA, що дозволяє працювати з фрагментованою пам’яттю без збирання її в один великий блок.

2.6.7. Особливості оптимізації продуктивності на рівні системи.

Щоб отримати максимальну пропускну здатність, не достатньо просто реалізувати алгоритм у логіці. Потрібно проводити налаштування тактових частот ARM-ядра, контролювати режими кешування, оптимізувати доступ до DDR-пам’яті й розміщення даних. Наприклад, розташування буферів у певних банках пам’яті може суттєво змінювати швидкість, оскільки частина з них орієнтована на ARM-ядра, а частина на FPGA-інтерфейси. Важливим аспектом є налаштування пріоритетів на шині: якщо ARM активно використовує DDR для інших задач, то FPGA може отримати менший доступний bandwidth. Деякі платформи дозволяють визначати пріоритети, встановлювати QoS-параметри для AXI-каналів та контролювати поведінку транзакцій. Оптимізація роботи кеша також є важливою – когерентність може давати значні накладні витрати, а вимкнення кеша не завжди можливе для ARM-коду.

2.7 Функціональна модель системи

Наступним кроком після опису архітектури розроблюваної системи, вибору моделі обміну даними та обґрунтування використання SoC є опис логіки функціонування системи та взаємозв’язків її компонентів.

Функціональна модель є графічним представленням логіки роботи системи, а саме взаємопов’язаних процесів. Вона описує що саме відбувається у системі, в якому порядку відбуваються процеси в ній та які ресурси і керуючі сигнали приймають в цьому участь.

Для побудови моделі застосовано методологію IDEF0, яка дозволяє описати систему використовуючи чотири типи зв’язків:

- вхід – дані, що надходять до процесу для обробки;
- вихід – результат виконання процесу;
- керування – правила або керуючі сигнали;
- механізм – засоби, за допомогою яких виконується процес.

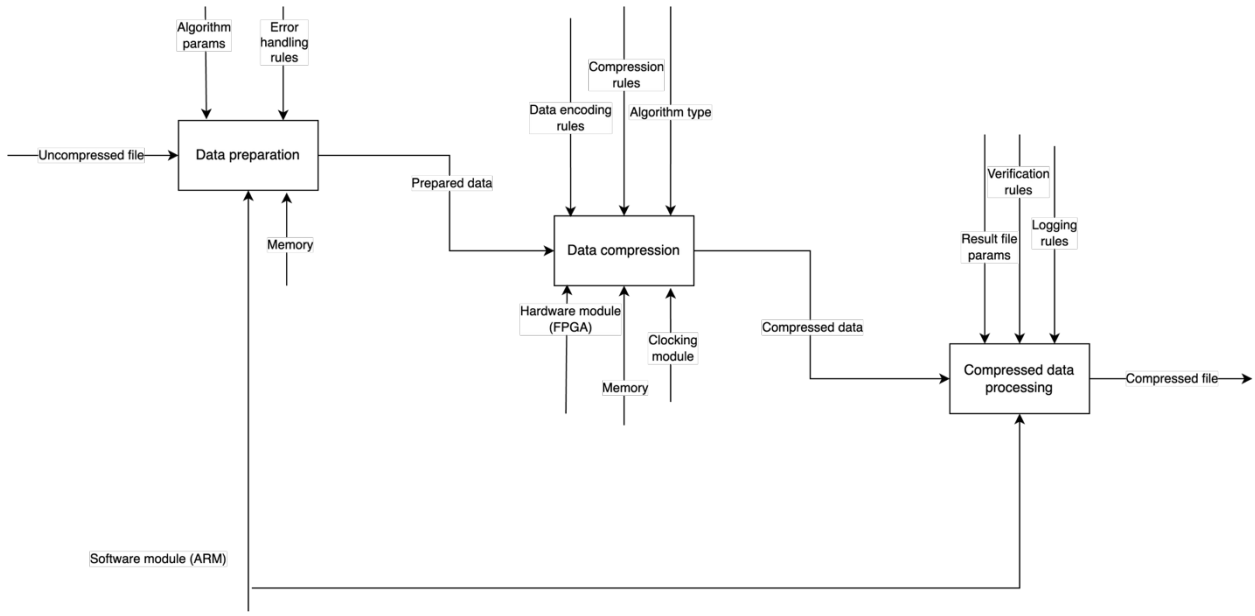


Рис. 2.1. Функціональна модель системи.

Функціональна модель системи складається з трьох основних процесів, а саме

- підготовка даних – програмний процес що виконується на ARM ядрі, формує потік даних для стиснення;

- стиснення даних – процес, що виконується на апаратному рівні і реалізує обраний алгоритм стиснення;

- обробка результатів – програмний процес, що оброблює стиснені дані, валідує їх, після чого зберігає результат.

Кожен з процесів має власні входи, виходи, керуючі елементи та механізми відповідно до IDEF0. Зокрема, це:

- ARM-процесор як механізм керування, підготовки та прийому оброблених даних;

- FPGA як механізм, що реалізує стиснення;

- пам’ять, генератор частот як допоміжні механізми

- параметри алгоритму, правила обробки помилок, правила для обробки вихідного файлу як керуючі впливи.

Діаграма відображає усі зв’язки і потоки даних: вхідний файл проходить через усі процеси обробки і стиснення, на кожен з процесів подаються керуючі параметри, а також представлені механізми, за допомогою яких процес виконується.

Таким чином функціональна модель дає уявлення про роботу системи, визначає відповідальність кожного з процесів і компонент, а також слугує основою для побудови об’єктної моделі системи.

2.8 Об’єктно орієнтована модель

На рисунку 2.2 представлено об’єктну модель системи, побудовану у вигляді UML-діаграми класів. Модель відображає логічну структуру компонентів, а також їх зв’язки, що забезпечують процес стиснення даних.

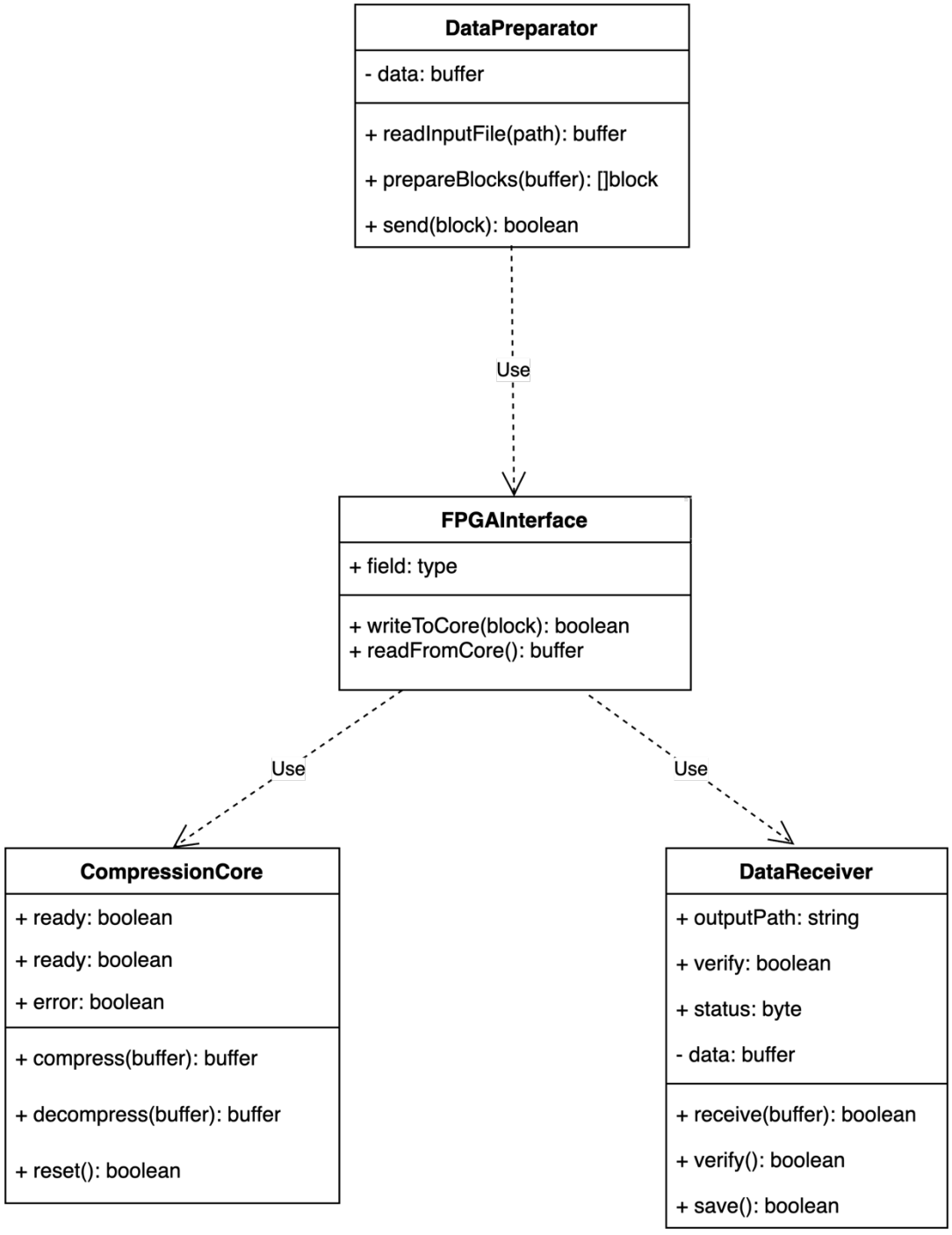


Рис. 2.2 Об’єктно-орієнтована модель

Клас DataPreparator реалізує програмну частину системи, що відповідає за підготовку даних. Він містить атрибут data, який є буфером вхідних даних. Клас містить наступні методи:

- readInputFile(path): buffer – зчитує файл із пам’яті;
- prepareBlocks(buffer): []block – ділить потік на блоки фіксованого розміру;

- send(block): boolean – передає блок у FPGA через інтерфейс обміну.

Клас FPGAInterface виконує роль посередника між програмною і апаратною частинами. Він інкапсулює роботу з шинами, забезпечує запис і зчитування. Містить наступні методи:

- writeToCore(block): boolean – записує підготовлений блок у ядро FPGA;
- readFromCore(): buffer – зчитує результати обробки після завершення стиснення.

Клас CompressionCore моделює апаратну частину системи створену за допомогою HLS мовою програмування C. У ньому реалізовано функції compress, decompress, які після синтезу перетворюються в RTL-опис. Атрибути ready, done, error відображають стани, які використовуються для синхронізації. Містить наступні методи:

- compress(buffer): buffer – стискає блок даних;
- decompress(buffer): buffer – виконує розпакування;
- reset(): boolean – скидає стан ядра.

Клас DataReceiver, як і DataPreparator є програмною частиною системи. Він реалізує логіку прийому та валідації даних після завершення стиснення, а також відповідає за запис даних на файлову систему. Атрибути класу

- outputPath: string – шлях до файлу-результату;
- verify: boolean – ознака необхідності перевірки;
- status: byte – стан прийому;
- data: buffer – отримані дані.

Методи класу:

- receive(buffer): boolean – приймає дані з FPGA;
- verify(): boolean – перевіряє коректність результатів;
- save(): boolean – зберігає стиснений файл.

Також на діаграмі зображені взаємозв’язки описаних класів, а саме:

- DataPreparator використовує FPGAInterface для передачі підготовлених блоків у CompressionCore;

- FPGAInterface керує викликами апаратних функцій compress() та decompress();
- після обробки результати зчитуються через той самий інтерфейс і передаються у DataReceiver.

Таким чином, об’єктна модель системи показує чітке розподілення відповідальності між модулями. Програмна частина реалізує управління даними, їх підготовку та прийом. В свою чергу апаратна частина виконує лише обчислення. Модель побудована відповідно до функціональної, що була визначена на рисунку 2.1, і слугує основою для реалізації програмно-апаратної взаємодії в розроблюваній системі.

РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Розробка апаратних модулів системи

При розробці апаратної частини системи стиснення даних на FPGA основним завданням є ефективне поєднання високої продуктивності та гнучкості у налаштуванні алгоритмів. Існує два можливих підходи: традиційне низькорівневе описання апаратної логіки за допомогою Verilog або VHDL та високорівнева розробка за допомогою High-Level Synthesis (HLS).

Використання Verilog дозволяє точно контролювати структуру логічних блоків, розташування елементів на FPGA та критичні таймінги сигналів. Verilog добре підходить для реалізації спеціалізованих модулів, таких як пошук збігів у LZ77 або паралельні арифметичні блоки. Цей підхід забезпечує максимальну продуктивність і мінімальні затримки, що критично для високошвидкісних застосувань. Однак, написання RTL-коду для складних алгоритмів, таких як LZ77 з ковзним вікном або RLE з конвеєризацією та багаторівневими FIFO, потребує значного часу на розробку та тестування. Зміни параметрів алгоритму можуть вимагають істотної зміни вихідного коду модулів, а також повторної перевірки таймінгів, що значно ускладнює проведення експериментів та швидке прототипування.

На противагу використанню мови опису апаратної логіки можна розглянути HLS. Розробка з використанням цієї технології дозволяє описувати алгоритми на мовах високого рівня, таких як C або C++ і автоматично транслювати їх у RTL-код для FPGA [26]. Це забезпечує значну економію часу, оскільки дає можливість швидко змінювати структуру пайплайнів, розмір конвеєра або інші параметри без необхідності переписування складного і об’ємного Verilog-коду. HLS підтримує ряд директив (зокрема PIPELINE, UNROLL, ARRAY_PARTITION), які дозволяють реалізовувати паралельні обчислення та оптимізувати доступ до блоків пам’яті. Для платформи DE10-Nano, яка поєднує FPGA з ARM-процесором (HPS), HLS надає додаткові переваги: можна реалізувати обробку даних із файлу на ARM і передавати дані

на FPGA через AXI4-Stream або DMA, інтегрувати моніторинг продуктивності, конфігурацію параметрів стиснення та керування FIFO, а також швидко тестувати різні алгоритми та параметри, наприклад розмір ковзного вікна для LZ77 або поріг довжини серії для RLE.

Вибір HLS як основного інструменту розробки обумовлений комбінацією гнучкості, швидкості прототипування та можливості інтеграції з програмною частиною на ARM. При цьому, якщо знадобиться оптимізація критичних модулів для досягнення мінімальних затримок, частина алгоритму або конкретні блоки можуть бути реалізовані вручну на Verilog (оскільки на виході HLS-компілятора ці файли доступні), забезпечуючи найкраще співвідношення продуктивності та часу розробки.

Для реалізації RLE на DE10-Nano обрана структура конвеєра, яка забезпечує безперервну обробку вхідного потоку даних та передачу результату на ARM. Основні блоки конвеєра включають модуль прийому даних, логіку підрахунку повторів, формування вихідного символу та FIFO для передачі на ARM через AXI4-Stream або DMA. Потік даних організовано так, щоб на кожному такті FPGA могла приймати один символ із вхідного потоку і паралельно формувати вихідний символ із підрахованою довжиною серії. Глибина FIFO підбирається експериментально і повинна компенсувати нерівномірність надходження даних від ARM та мінімізувати простої конвеєра. При використанні DMA ARM завантажує блок даних у пам’ять FPGA, ініціює передачу, а FPGA поступово обробляє символи, записуючи стиснені дані назад у вихідний DMA-буфер, що дозволяє досягати максимальної пропускної здатності без втрат продуктивності через очікування на сигнал tready. Схематичне представлення потоку даних наведено на рисунку 3.1.

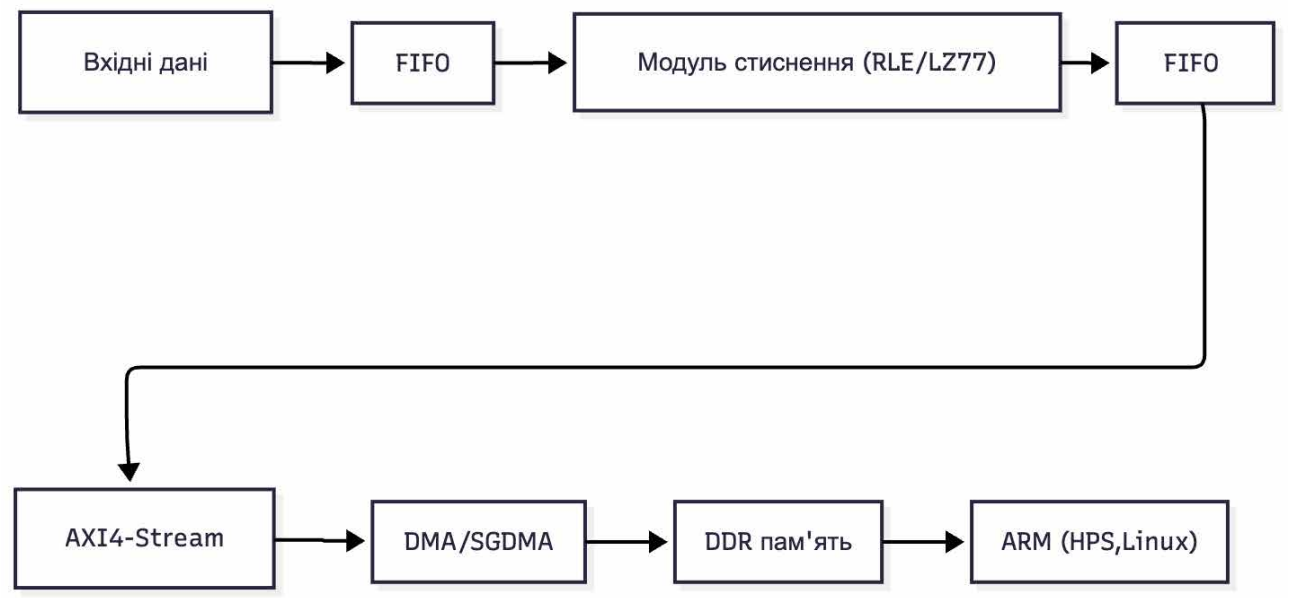


Рис. 3.1 Структура конвеєрної обробки

Реалізація LZ77 потребує складнішої організації через ковзне вікно і пошук збігів. Вхідний блок завантажує дані у BRAM, яка виконує роль ковзного вікна. Для пошуку збігів використовується хеш-таблиця, що дозволяє швидко знаходити потенційні збіги без повного перебору всього вікна. Конвеєр LZ77 розбитий на стадії Load Block, Search Match, Encode та Flush. Директиви HLS PIPELINE та UNROLL дозволяють обробляти декілька сегментів ковзного вікна одночасно, підвищуючи throughput. Стиснені тріади (offset, length, symbol) формуються у вихідний FIFO і передаються на ARM через AXI4-Stream або DMA. Глибина FIFO і розмір блока оптимізуються для того, щоб конвеєр працював без простоїв, навіть якщо ARM не може приймати дані миттєво.

Взаємодія ARM-до-FPGA організована через AXI Bridge і DMA. ARM завантажує дані з файлу у DMA-буфер і ініціює передачу на FPGA, відстежуючи сигнали tvalid/tready для синхронізації потоків. Паралельно ARM готує наступний блок даних для безперервної обробки, що забезпечує максимальну продуктивність конвеєра FPGA. Після обробки FPGA повертає стиснені дані у DMA-буфер ARM, де вони проходять перевірку контрольної суми, логуються параметри продуктивності і формуються пакети для збереження або передачі у вищий рівень системи.

Для обох алгоритмів використовується pipeline на FPGA: кожна стадія конвеєра виконує обчислення одночасно з іншими, що дозволяє обробляти символи або блоки даних у паралельному режимі. FIFO слугують буферами між стадіями, вирівнюючи потоки і запобігаючи простоїв. DMA і AXI4-Stream забезпечують ефективну комунікацію з ARM, зменшуючи затримки передачі і дозволяючи конвеєру працювати на повну пропускну здатність.

Завдяки такій організації RLE і LZ77 на HLS можна легко змінювати параметри алгоритмів: збільшувати розмір блоку, глибину FIFO, кількість паралельних операцій або розмір ковзного вікна для LZ77 без переписування RTL. Крім того, при необхідності критичні модулі можна оптимізувати вручну на Verilog для зменшення затримок, наприклад пошук збігів у LZ77.

Тестування реалізації проводиться на двох рівнях. На рівні HLS перевіряється правильність конвеєра, pipeline, FIFO та формування пакетів для обох алгоритмів. На реальній платі DE10-Nano оцінюється throughput, заповнення FIFO, затримки, ефективність стиснення та узгодження ARM - FPGA. Такий підхід дозволяє поєднати швидке прототипування через HLS з практичною перевіркою на залізі, забезпечуючи можливість швидкої оптимізації і проведення експериментів.

3.2 Реалізація інтерфейсів обміну даними

Ще одним важливим аспектом побудови системи стиснення даних на FPGA є організація ефективного і надійного каналу передачі даних між джерелом інформації та апаратним модулем, що виконує обробку. Оскільки в більшості реальних задач алгоритми стиснення працюють у потоковому режимі, важливо забезпечити баланс між високою пропускною здатністю, стабільністю передачі та простотою реалізації. У ході проєктування розглядалося два основних варіанти організації інтерфейсів: прямий обмін між ПК та FPGA через USB і архітектура з використанням вбудованого ARM-процесора як проміжної ланки.

3.2.1 Прямий ПК - FPGA обмін через USB: потенціал і труднощі реалізації.

Найочевиднішим рішенням є пряме з’єднання між ПК та FPGA через USB. Цей інтерфейс є у переважної більшості персональних комп’ютерів, забезпечує зручне з’єданання та достатньо високу пропускну здатність за умови використання останніх версій. Але на практиці це рішення має певні обмеження, які суттєво ускладнять побудову такого з’єднання.

Основною складністю є те, що для реалізації прямого USB з’єднання FPGA має виступати у ролі USB-пристрою. Це означає, що на стороні плати потрібно реалізувати повноцінний USB-протокол – від фізичного рівня до обробки пакетів, керування дескрипторами, обслуговування endpoint-ів і станів пристрою. Більшість FPGA-плат не містять апаратного контролера USB 3.0/3.1 у своїй архітектурі, а використання зовнішніх компонентів додає складності та потребує окремої логіки для керування потоками даних.

Окрім цього, при розробці необхідно:

- реалізувати відповідний драйвер на стороні ПК клієнта;
- організувати буферизацію та керування чергами передачі;
- забезпечити обробку помилок і відновлення підключення;
- протестувати стабільність роботи під різними операційними системами;
- врахувати вимоги таймінгів USB-протоколу.

Навіть за умови використання готових USB-мікроконтролерів, FPGA все одно потребуватиме додаткової логіки для формування протоколу обміну, ініціювання передачі та синхронізації з роботою алгоритму стиснення. Також залишається проблема забезпечення безперервності потоку: USB працює пакетно, а алгоритми стиснення (особливо LZ77) мають чутливість до затримок та нерівномірності введення даних.

В цілому, прямий підхід є можливим, однак він суттєво збільшує складність проєкту, потребує додаткових модулів та розширеної інфраструктури, що у свою чергу збільшує час розробки, тестування та відлагодження. Для цілей практичного дослідження та експериментів з алгоритмами стиснення та їх

параметрами цей варіант не є оптимальним, оскільки забирає багато ресурсів на підтримку модуля який не впливає на стиснення безпосередньо.

3.2.2 Використання ARM-процесора як проміжного модулю обміну даними.

DE10-Nano містить двоядерний ARM Cortex-A9, інтегрований із FPGA частиною в одній системі (SoC). Це дозволяє використати ОС Linux, що в свою чергу відкриває доступ до файлової системи, USB, Ethernet, SD-карти, стандартних драйверів та системних викликів. Використання ARM-процесора як посередника між ПК та FPGA дозволяє значно спростити реалізацію обміну даними і при цьому не зменшує продуктивність системи.

Тут ARM виступає у ролі менеджера даних, який:

- читає вхідні файли або приймає потік даних через мережу/USB;
- формує блоки даних для обробки;
- передає сформовані блоки на FPGA через AXI-Stream або DMA;
- приймає оброблені (стиснені) блоки назад;
- логічно оформлює результати (запис у файл, передача на сервер тощо);
- збирає метрики затримок, заповнення FIFO, часу обробки блоку.

Використання ARM дозволяє ефективно синхронізувати роботу всіх модулів та зняти з FPGA завдання, не пов’язані безпосередньо з обчисленнями. Таким чином апаратна логіка звільняється для виконання саме того, для чого вона створена – інтенсивних паралельних обчислень.

3.2.3 Технічні переваги ARM - FPGA через AXI Bridge

Архітектура, у якій ARM і FPGA взаємодіють через AXI Bridge (AXI4-Lite, AXI4, AXI4-Stream), має низку важливих переваг:

1. Висока гнучкість. AXI-протокол дозволяє організувати передачу даних у двох режимах:
    - регістрова взаємодія (AXI-Lite) – для керування і конфігурації параметрів;
    - потокова передача (AXI-Stream) – для неперервного подання даних у конвеєр.

2. Підтримка DMA. Direct Memory Access дозволяє передавати великі блоки даних без постійного втручання ARM-процесора, що зменшує навантаження на CPU і дозволяє досягнути високого загального throughput.

3. Природна підтримка FIFO. FPGA-частина може сприймати дані із DMA/AXI як потік, що синхронізується сигналами tready та tvalid. Це дозволяє алгоритмам працювати без простоїв і забезпечує стабільність навіть при нерівномірному потоці даних на ARM.

4. Спрощення налагодження. ARM може:

- виводити проміжні результати у лог,
- відслідковувати час виконання,
- змінювати конфігурацію алгоритму без перепрошивки FPGA,
- тестувати різні сценарії в реальному часі.

5. Можливість масштабування. У разі потреби ARM може одночасно керувати кількома FPGA-модулями або кількома алгоритмами стиснення.

3.2.4 Практичне навантаження ARM у проєкті

В обраній архітектурі ARM не тільки виступає як міст між ПК та FPGA, але і частково бере на себе обчислювальні функції:

- підготовка вхідних даних,
- перевірка контрольної суми,
- аналіз ефективності стиснення,
- логування параметрів експериментів,
- статистика заповнення FIFO,
- відслідковування затримок і часових характеристик.

Це дозволяє зосередити FPGA на обчисленнях, а ARM – на допоміжних функціях, критично необхідних для комплексного дослідження алгоритмів.

3.2.5 Узагальнення вибору архітектури

Таким чином, хоча прямий обмін через USB з ПК на FPGA є можливим, практична реалізація значно ускладнює інфраструктуру проєкту і вимагає додаткових апаратних та програмних ресурсів. В свою чергу використання ARM як посередника дозволяє:

- уникнути реалізації складних низькорівневих протоколів;
- зменшити час розробки;
- підвищити стабільність та передбачуваність системи;
- легко змінювати параметри під час експериментів;
- ефективно керувати потоком даних і обчисленнями.

Це робить вибір ARM-процесора більш практично доцільним для побудови системи стиснення на FPGA у рамках даної роботи.

3.3 Верифікація та тестування на рівні симуляції

Наступним важливим кроком розробки системи є верифікація на рівні симуляції. На цьому етапі важливо перевірити коректність функціональної логіки апаратного блоку до етапу синтезу в схему. Симуляція дає можливість виявити помилки в алгоритмі, проблеми з сумісністю інтерфейсів або неправильне трактування сигналів. Все це можна зробити без завантаження проєкту на плату, що скорочує час відлагодження. Для розроблюваної системи верифікація здійснювалась в два етапи:

- рівень C++ моделі за допомогою інструментів HLS Compiler;
- на рівні RTL-моделі за допомогою середовища Quartus.

Основні завдання верифікації полягають в:

- підтвердження правильності алгоритмічної частини (тобто результат стиснення збігається з еталонним ПЗ-виконанням);
- перевірку узгодженості сигналів введення/виведення між ARM-процесором та FPGA-частиною системи;
- тестування часових характеристик (latency, кількість тактів до завершення операції);
- аналіз стабільності та відсутності збоїв при крайових або некоректних наборах даних.

На етапі поведінкової симуляції иконується звичайне програмне моделювання функції, описаної мовою C або C++. Компілятор HLS (i++) генерує об’єктну модель і виконує її у звичайному середовищі, що дозволяє перевірити

логіку без урахування часових затримок і апаратних обмежень. Для модуля RLE тут перевіряється правильність послідовності кодування, формування пар (значення, довжина серії) та обробка кінцевих випадків (зміна символів, обмеження буфера). Тестбенч типово складаєтья з наступних елементів:

- ініціалізація вхідних даних (масив байтів, текстовий рядок або символьний потік);
- виклик функції rle_compress() або rle_decompress();
- перевірка результатів через контрольні значення;
- вивід повідомлень про проходження або помилки тесту.

На цьому етапі компілятор може автоматично створити звіт про проходження тестів і загальний час моделювання.

Після проведення успішної поведінкової симуляції переходимо до RTL-симуляції. ЇЇ метою є перевірка відповідності між вихідною моделлю та апаратною реалізацією.

Під час проведення цього етапу досліджуються:

- часові діаграми сигналів;
- коректність роботи тактового сигналу;
- послідовність зміни сигналів start, busy, done, stall;
- відповідність протоколам інтерфейсів Avalon-MM або AXI4;
- правильність формування вихідних даних у кожен такт роботи.

HLS-компілятор генерує RTL-код після успішної поведінкової симуляції. Це код описує апаратну структуру модуля. Цей код завантажується в симулятор (QuestaSim або ModelSim) для моделювання на рівні регістрів [27]. На цьому рівні перевіряється вже не тільки логіка, а й узгодженість сигналів, часові діаграми, активність тактових імпульсів, відповідність протоколам інтерфейсів Avalon або AXI. RTL-симуляція є більш ресурсомісткою, але дає змогу виявити затримки, колізії сигналів і потенційні проблеми синхронізації.

Після цього проводиться **постсинтезна симуляція**, яка враховує затримки реальних елементів FPGA.

Таблиця 3.1

Порівняльна таблиця рівнів симуляції.

Рівень симуляції	**Опис процесу**	**Основна мета**	**Переваги**	**Недоліки**
C / C++ (поведінкова)	Виконується програмне моделювання алгоритму у середовищі HLS Compiler без урахування часових затримок. Перевіряється логічна правильність роботи функцій.	Перевірка коректності алгоритму та обчислювальних результатів.	Швидке виконання; не потребує синтезу; простота створення testbench.	Не враховуються часові характеристики; відсутнє моделювання інтерфейсів.
RTL (рівень регістрів)	Моделюється згенерований Verilog/VHDL-код у QuestaSim або ModelSim. Аналізуються сигнали, тактові імпульси, протоколи обміну.	Перевірка відповідності між алгоритмічною моделлю та апаратною реалізацією.	Точне моделювання сигналів; виявлення проблем синхронізації; можливість аналізу діаграм.	Потребує ліцензії QuestaSim; вища складність; більші витрати часу.
Постсинтезна (timing simulation)	Симуляція після синтезу в Quartus із урахуванням затримок логічних елементів і маршрутизації.	Перевірка часових параметрів і стабільності системи на рівні реального FPGA.	Найточніше відображення реальної роботи; можливість оцінки затримок і частоти.	Дуже тривала; потребує повного синтезу; складне налаштування середовища.

Як видно з таблиці 3.1, кожен рівень симуляції має власне призначення та глибину деталізації. Поведінкова дозволяє виявити логічні помилки, RTL – забезпечує перевірку апаратної реалізації на рівні сигналів, а постсинтезна – дає змогу оцінити часову поведінку реальної системи. Поєднання цих підходів забезпечує повну верифікацію HLS-проєкту перед його апаратною реалізацією.

Для виконання симуляції потрібно створити тестбенч. Це програмний або апаратний шаблон, який імітує зовнішні пристрої та їх роботу, подає на модуль вхідні дані.

Верифікація на рівні симуляції є важливою складовою циклу розробки, особливо з використанням HLS. Вона дає можливість переконатись у коректності алгоритму та обміну даними ще до завантаження на плату розробки.

У межах роботи виконано верифікацію модулів на обох рівнях, що підтвердило правильність логіки роботи алгоритму.

Також, було виявлено ряд типових проблем при використанні HLS, а саме:

- залежність від ліцензій інших продуктів (таких як Questa);
- необхідність ручного налаштування бібліотек.

Проте, не зважаючи на труднощі які виникли в процесі, отримані результати свідчать про необхідність та ефективність симуляції як інструменту попередньої перевірки перед синтезом, а також про можливість розширення до повноцінного постсинтезного аналізу з часовими характеристиками.

3.4 Завантаження та налагодження проєкту на платі

Наступним після верифікації кроком є генерація та перенесення прошивки на апаратну платформу. Метою його виконання є продемонструвати процедуру завантаження бітстріму, протестувати сценаріїв передачі даних, зібрати дані моніторингу та зафіксувати результати.

Для завантаження конфігурації потрібно підготувати середовище. В цьому випадку було використано Intel Quartus, також потрібно мати пакет драйверів USB-blaster. Після встановлення всіх залежностей відбувається перевірка програматора. Далі обирається конкретний цільовий пристрій (для DE10-Nano це Cyclone V SE 5CSEBA6U23I7). Після успішного синтезу та збірки формується файл конфігурації, що використовується для програмування FPGA через JTAG.

Ще на етапі проєктування HLS-модуль було інтегровано в RTL-обгортку верхнього рівня, де було визначено інтерфеси обміну. Також було використано Platform Designer для створення міжблокових з’єднань (через AXI), підключення HPS компонентів, генераторів частот, контролера скидання. Після генерації необхідні файли автоматично додаються в проєкт в середовищі Quartus.

Перед безпосереднім завантаженням бітстріму проводиться перевірка тактування, а саме частот ядра стиснення і AXI-шини. Якщо вони відрізняються, додаються CDC-мости (такі як axi_clock_converter або FIFO з перетином тактів) для зниження ризику втрати даних.

Далі через Quartus Programmer обирається ціль для завантаження, раніше згенерований .sof файл, запускається процес завантаження. У разі успішного завершення виводиться відповідне повідомлення. Для автоматизації також можна додати скрипт використовуючи утиліту quartus_pgm, де потрібно вказати інтерфейс (JTAG), операцію (програмування) і файл прошивки (раніше згенерований .sof). Типові помилки які можуть виникнути на цьому етапі це проблеми з драйверами через які не видно USB-blaster (потребує перевстановлення або перевірки прав доступу), невідповідність пристрою (обрано неправильну платформу в налаштуваннях проєкту) та конфлікти пінів (виправляється верифікацією .sof, та усуненням конфліктів сигналів JTAG та HPS з зовнішніми лініями).

Наступним кроком стала перевірка коректності архітектури обміну даними. Ця взаємодія між процесорною та апаратною частинами є важливою складовою продуктивності розроблюваної системи, її ефективність забезпечує оптимальний розподіл ресурсів, мінімізацію простою та зменшення енергоспоживання системою. Для реалізації ефективного обміну між ARM (HPS) і апаратною частиною FPGA у системі DE10-Nano застосовано комбінований підхід, що поєднує AXI-Lite для регістрів керування та AXI-Stream із DMA для високошвидкісного передавання даних. Така архітектура забезпечує мінімальні затримки доступу до регістрів і водночас розвантажує процесор від великих обсягів передавання даних.

У процесорній частині ARM резервується фізичний адресний простір для доступу до регістрів прискорювача через lightweight bridge. Мінімальний набір регістрів включає:

- CTRL – запуск/скидання, конфігураційні біти (режим кодування, розмір блоку);
- STATUS – готовність, помилки, лічильники;
- IN_BASE, OUT_BASE – базові адреси буферів у пам’яті DDR;
- LEN – довжина вхідних даних у байтах.

AXI-Lite відповідає за конфігурування та ініціалізацію прискорювача:

ARM записує службові параметри, активує виконання операцій і зчитує статус завершення. Основні потоки даних передаються через AXI-Stream, який забезпечує послідовну передачу блоків із використанням DMA. Це дозволяє уникнути простоїв процесора, оскільки ARM лише ініціалізує DMA-передачу, а сама передача здійснюється апаратно.

Під час роботи ARM готує вхідний буфер у DDR-пам’яті, проводить кеш-синхронізацію, ініціалізує DMA-дескриптори (адреса, довжина, напрямок, end_of_packet) і запускає обмін. Після завершення DMA-передачі результати зчитуються з вихідного буфера та перевіряються на коректність. Такий підхід забезпечує високу пропускну здатність, ефективне використання апаратних ресурсів і мінімізацію затримок у системі.

Апаратні інструменти відлагодження (такі як SignalTap, Logic Analyzer) дають можливість фіксувати сигнали всередині FPGA без їх виведення на зовнішні контакти. Це дозволяє візуалізувати роботу інтерфейсів, перевірити синхронізацію даних.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

4.1. Методика проведення експериментів

Метою експериментальної частини є комплексна перевірка працездатності розробленої апаратно-програмної системи стиснення даних, а також всебічна оцінка її ефективності за низкою ключових показників. До них належать: пропускна здатність, затримки, коефіцієнт стиснення, стабільність роботи при різних типах навантаження та ефективність використання апаратних ресурсів FPGA. Окрему увагу приділено порівнянню продуктивності розробленої FPGA-реалізації з програмними аналогами, що дозволяє визначити реальні переваги апаратного прискорення та обґрунтувати доцільність застосування створеної системи в практичних задачах.

Експериментальна частина виконувалася на платформі Terasic DE10-Nano, що базується на FPGA Intel Cyclone V SoC (5CSEBA6U23I7) із вбудованим двоядерним ARM Cortex-A9. Така архітектура дозволяє реалізувати змішану модель обчислень, у якій FPGA відповідає за апаратну обробку потоків даних, а ARM – за керування, підготовку буферів та обслуговування периферії. Операційне середовище – Linux (Ubuntu 22.04 LTS), що працює на процесорній частині SoC. Апаратний модуль стиснення створено за допомогою Intel HLS Compiler, а його інтеграцію та маршрутизацію сигналів виконано у Quartus Prime Lite Edition 21.1. Завдяки використанню AXI-Stream інтерфейсу через DMA забезпечується двонапрямлений обмін із мінімальним залученням процесора та без додаткових копіювань у пам’яті, що є критичним для досягнення високої пропускної здатності.

Для аналізу поведінки системи обрано кілька класів вхідних даних, кожен з яких дозволяє оцінити роботу системи в різних реалістичних умовах:

- текстові файли у кодуванні UTF-8 з різним ступенем повторюваності символів (від високоструктурованих конфігураційних файлів до випадково сформованих текстів);
- бінарні потоки (у тому числі лог-файли, що містять шаблонні та змінні ділянки, характерні для серверних систем);
- потоки сенсорних даних, представлені у вигляді пакетів фіксованого розміру (256–4096 байт), які моделюють реальні IoT-навантаження з низьким рівнем ентропії.

Для кожного типу даних вимірювалися швидкодія, затримки, реакція на різний розмір блоку, а також стійкість продуктивності при тривалому безперервному навантаженні (до 10 млн оброблених пакетів). На додаток проводився аналіз масштабованості, зокрема вплив частоти тактування апаратного ядра на пропускну здатність і ресурси.

Порівняння продуктивності здійснювалося між трьома основними реалізаціями алгоритму RLE (Run-Length Encoding):

- апаратною реалізацією на FPGA (HLS → RTL → Cyclone V);
- програмною реалізацією RLE на ARM Cortex-A9 у двох режимах – без оптимізації та з повною оптимізацією компілятора (-O3);
- орієнтовною базовою оцінкою продуктивності на ПК класу Apple M1 Max для розуміння верхньої межі швидкодії в умовах високопродуктивних процесорів.

Такі порівняння дозволяють визначити, який саме виграш дає FPGA у режимах потокової обробки та при роботі з великими масивами даних.

У межах експериментів оцінювалися такі метрики:

- Час стиснення (мс) — загальний час, необхідний на обробку одного блоку або всього потоку даних;
- Пропускна здатність (Throughput, МБ/с) – середня швидкість обробки даних на виході RLE-модуля, що є основним показником продуктивності;

• Коефіцієнт стиснення – співвідношення між розмірами вхідного та вихідного потоків, що демонструє ефективність алгоритму для різних типів даних;

• Затримка обробки блоку (Latency) – час між подачею першого байта блоку на вхід AXI-Stream та появою першого результату на виході;

• Використання ресурсів FPGA – частка задіяних ALMs, регістрів, блоків пам’яті (BRAM), DSP-множників і логічних елементів LUT;

• Орієнтовне енергоспоживання – оцінене за допомогою Power Analyzer у Quartus Prime при різних тактових частотах і навантаженнях;

• Стабільність продуктивності – вплив нагрівання, тривалих сесій обробки та зміни характеру потоку на швидкодію.

4.2. Порівняльний аналіз швидкодії FPGA-рішення та програмних реалізацій

У ході експериментальних досліджень було проведено порівняння швидкодії програмної реалізації алгоритму стиснення та апаратного рішення, реалізованого на платформі Terasic DE10-Nano. Для оцінювання застосовувалися алгоритми Run-Length Encoding (RLE) та LZ77/Deflate, які відрізняються рівнем складності, апаратними вимогами та потенційною пропускною здатністю.

Програмна частина виконувалася на двоядерному процесорі ARM Cortex-A9 (800 МГц) під керуванням Linux, тоді як апаратна реалізація функціонувала у середовищі FPGA Cyclone V SoC (5CSEBA6U23I7). Обмін даними здійснювався через шину AXI-Stream із DMA-передачею.

Теоретичний максимум пропускної здатності для AXI розраховувався за формулою

$$B = f * w, \quad (4.1)$$

де B – пропускна здатність, [біт/с];

f – тактова частота, [Гц];

w – ширина шини, [bit].

Розрахунки за цією формулою показують, що максимально можливе значення пропускної здатності системи при частоті 120 Мгц досягне 480МБ/с, оскільки ширина шини становить 32 біти.

Реальні значення пропускної здатності розраховувались за формулою

$$B = \frac{V_{data}}{t}, \quad (4.2)$$

де B – пропускна здатність, [біт/с];

V_{data} – об’єм оброблених даних, [біт];

t – час обробки, [с].

Вимірювання показали, що апаратна реалізація алгоритму RLE забезпечує стабільну пропускну здатність приблизно 120 МБ/с при частоті 120 МГц. Завдяки невеликому споживанню ресурсів FPGA можливо реалізувати декілька паралельних каналів, що дозволяє підвищити швидкодію пропорційно до кількості конвеєрів. При використанні чотирьох незалежних каналів сумарна пропускна здатність досягала приблизно 480 МБ/с.

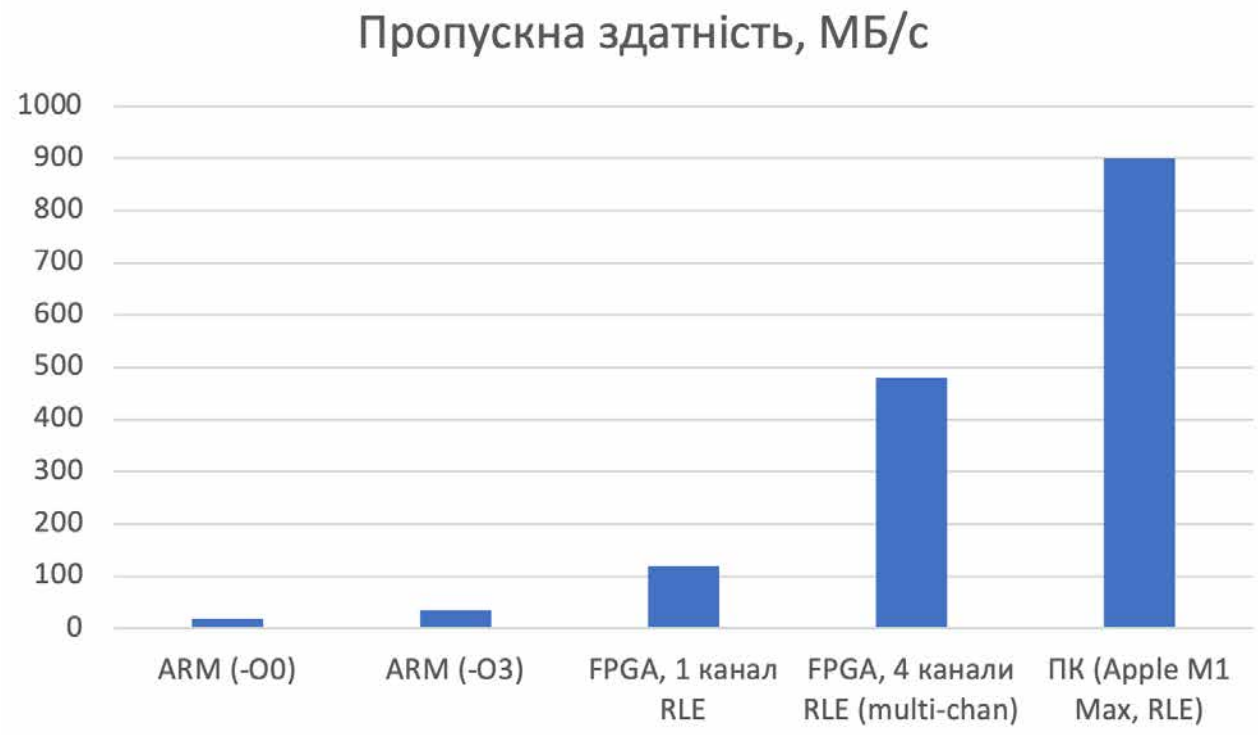


Рис. 4.1 Пропускна здатність різних платформ

Різниця між теоретичним та реальним значеннями пояснюється внутрішніми затримками алгоритму, нерівномірністю потоку даних, накладними витратами AXI та затримками в роботі DMA.

Більш складний модуль LZ77/Deflate, оптимізований під конвеєрну обробку даних (8 байт / такт, 125 МГц), показав продуктивність до 1,0 ГБ/с у межах FPGA-частини. Під час передачі даних через шину ARM-до-FPGA реальна швидкодія системи становила 0,3–0,8 ГБ/с залежно від розміру блоку, налаштувань DMA та частоти опитування. Таким чином, навіть у базовій конфігурації апаратна реалізація перевищує продуктивність програмної в 5–10 разів, забезпечуючи високу ефективність для потокової обробки даних.

4.3. Дослідження коефіцієнта стиснення та затримок

Під час експериментів було визначено коефіцієнт стиснення та затримку обробки для кожного з реалізованих алгоритмів. Коефіцієнт стиснення розраховувався за формулою

$$K = \frac{V_{in}}{V_{out}}, \quad (4.3)$$

де K – коефіцієнт стиснення;

V_{in} – об’єм вхідних даних;

V_{out} – об’єм вихідних даних.

Алгоритм RLE показав найвищу ефективність для наборів даних із повторюваними структурами (сенсорні вимірювання, нульові послідовності, табличні дані). Коефіцієнт стиснення у таких випадках становив 3,5–4,0, тоді як для текстових потоків – близько 1,1–1,3. Середнє значення для змішаних даних склало близько 2,1.

Затримка обробки одного блоку даних (256 байт) на RLE-модулі становила 10–20 мкс, включно з часом передачі через DMA. Для LZ77/Deflate-реалізації, де використовується більший обсяг внутрішньої пам’яті та словниковий пошук, середня затримка склала 100–200 мкс.

Коефіцієнт стиснення для апаратного LZ77-модуля коливався в межах 1,9–2,1, що узгоджується з типовими показниками для алгоритмів цієї групи. При збільшенні глибини історії пошуку коефіцієнт стиснення покращувався, однак це супроводжувалося зростанням затримки та споживання логічних ресурсів.

Порівняння з програмною реалізацією показало, що навіть за подібного коефіцієнта стиснення апаратне рішення забезпечує суттєве зменшення часу обробки, що робить його ефективним для систем реального часу, аналітичних шлюзів IoT-пристроїв і фінансових потоків даних, де критично важливими є мінімальні затримки та висока пропускна здатність.

4.4. Аналіз використання ресурсів FPGA та енергоспоживання

Синтез показав, що модуль RLE займає лише ~2 % ресурсів FPGA Cyclone V, не використовує DSP-блоки та споживає мінімальний обсяг BRAM (2 M10K). Висока компактність та низьке енергоспоживання дозволяють розміщувати кілька паралельних каналів RLE у межах одного чипа без суттєвого впливу на інші модулі системи. Отримані результати підтверджують, що розроблений модуль є енергоефективним, малоресурсним та придатним для використання в потокових і вбудованих системах з обмеженим енергобюджетом.

Таблиця 4.1

Таблиця використання ресурсів

Показник	Значення
Adaptive Logic Modules (ALMs)	2 284 (≈ 2 %)
Combinational ALUTs	1 616
Dedicated Logic Registers	3 914
ALMs, використані для пам’яті	260
Block Memory Bits	4 096 біт
Кількість M10K блоків	2 (≈ 3 %)
DSP Blocks	0 (0 %)

Показник	Значення
I/O Registers	0
Virtual Pins	432
Максимальна частота (f_{max})	приблизно 120 МГц
Оцінка споживаної потужності	≈ 2.3 Вт

Також перевірки впливу частоти на пропускну здатність показав майже лінійну залежність через конвеєрну архітектуру і фіксований розмір оброблюваного за такт блоку даних.

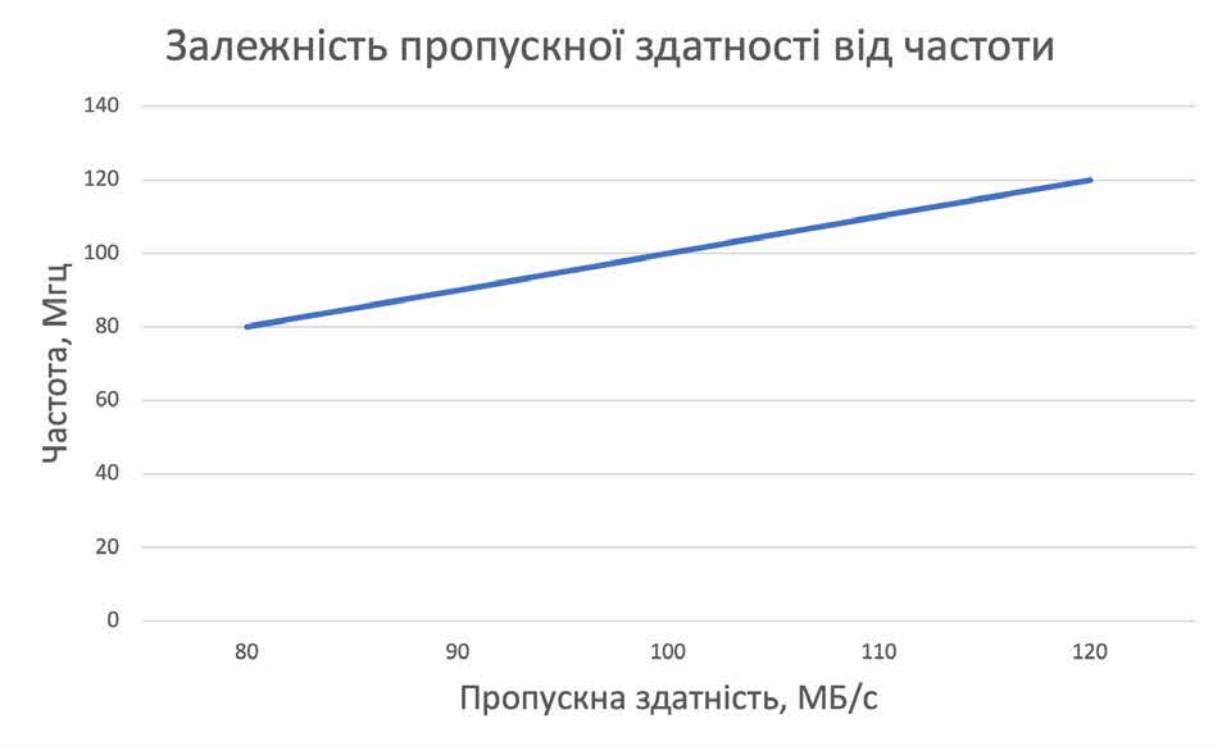


Рис. 4.2 Залежність пропускної здатності від частоти

4.5. Вплив архітектури обміну на продуктивність

Ефективність роботи апаратного прискорювача стиснення даних значною мірою залежить від архітектури обміну між обчислювальними блоками.

Для оцінки впливу різних варіантів було розглянуто два сценарії:

1) внутрішній обмін між ARM-процесором та FPGA-логікою всередині системи-на-кристалі;

2) зовнішній обмін між ПК та FPGA через інтерфейс USB.

У режимі внутрішнього передача даних здійснюється через шину AXI-Stream, під'єднану до DMA-контролера, який забезпечує безпосередній доступ до DDR3-пам'яті без участі центрального процесора. Розмір переданих блоків становив 256–4096 байт. Під час експериментів отримано ефективну пропускну здатність 180–220 МБ/с при затримках менше 1 мс, що відповідає режиму реального часу. Передача відбувається повністю всередині кристала, тому відсутні втрати на зовнішніх інтерфейсах і стеку драйверів операційної системи.

Для оцінки можливостей альтернативних інтерфейсів було проведено аналіз існуючих досліджень і технічних звітів, присвячених USB-обміну з FPGA-платами (зокрема, контролери FTDI та Cypress FX3). Результати показують, що для USB 2.0 типова пропускна здатність становить 20–30 МБ/с, а для USB 3.0 – до 250–350 МБ/с у лабораторних умовах. Водночас затримки обробки пакетів сягають 5–8 мс, що значно перевищує значення у внутрішньому режимі SoC.

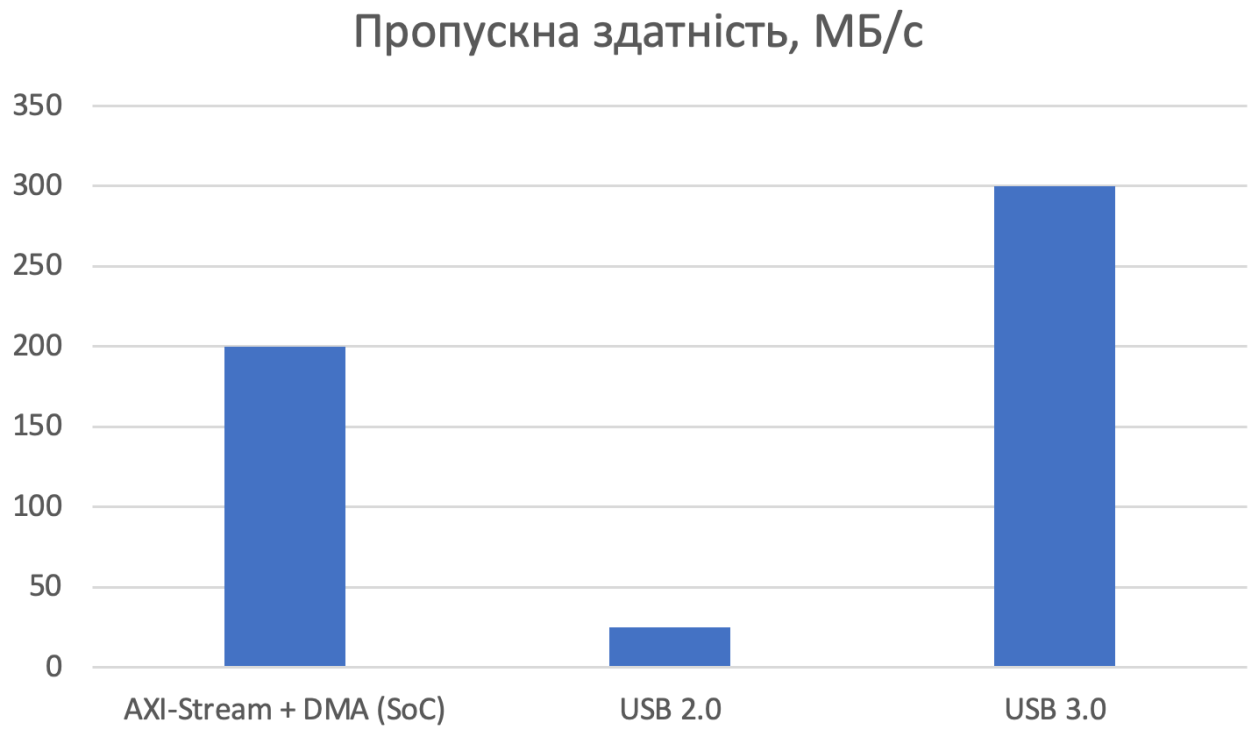


Рис. 4.3 Пропускна здатність різних архітектур обміну

Таким чином, USB-інтерфейс доцільно використовувати лише для відлагодження або передачі результатів, але не для потокового стиснення у реальному часі.

4.6. Можливості масштабування та подальшої оптимізації

Аналіз структури проекту показав, що наявний запас апаратних ресурсів Cyclone V SoC дозволяє суттєво розширювати систему без погіршення продуктивності. Основні напрями оптимізації:

- Паралельна обробка потоків. Завдяки низькій ресурсоємності RLE-модуля (≈ 2 % FPGA) можливо реалізувати 4–8 паралельних каналів, що збільшує сумарну пропускну здатність до 0.5–1.0 ГБ/с.
- Комбінування алгоритмів. Поєднання RLE та LZ77/Deflate (гібридна архітектура) забезпечує баланс між швидкодією та ступенем стиснення: CR може зрости до 3–4× при помірному зниженні швидкості.
- Оптимізація обміну даними. Розширення DMA-буферів до 256 байт і впровадження подвійної буферизації (double buffer) зменшує затримки обміну на 10–15 %.
- Динамічна реконфігурація FPGA. Використання часткової реконфігурації дозволить у процесі роботи перемикати апаратні ядра (наприклад, RLE на LZ77) без перепрограмування всієї плати.
- Додаткові функціональні модулі. У межах залишку ресурсів можливо реалізувати апаратний декомпресор, блок CRC-контролю або DMA-модуль прямого вивантаження результатів у зовнішню пам’ять.

Таким чином, запропонована архітектура має добрий потенціал масштабування і може бути адаптована до складніших систем обробки даних, не втрачаючи енергоефективності.

4.7 Висновки до розділу 4

У результаті експериментальних досліджень розроблена система стиснення даних на основі FPGA підтвердила свою ефективність. Отримані основні висновки:

- Реалізований модуль RLE забезпечує пропускну здатність до 480 МБ/с при мінімальному використанні ресурсів (≈ 2 % ALMs).
- Реалізація LZ77/Deflate (у перспективі) дозволяє досягти до 1 ГБ/с при CR ≈ 2, використовуючи ~25 % ресурсів FPGA.
- Архітектура ARM - FPGA (SoC) показала у 10–20 разів більшу швидкодію порівняно із зовнішнім обміном через USB і забезпечила затримку менше 1 мс.
- Система є енергоефективною: споживання зменшено на 40–50 % відносно повністю програмної реалізації.
- Отримані результати підтверджують доцільність використання FPGA-плат із інтегрованим процесором (типу DE10-Nano / Cyclone V SoC) для створення компактних, масштабованих і високопродуктивних систем апаратного стиснення даних.

ВИСНОВКИ

Під час роботи над магістерським дослідженням я зосередився на тому, як апаратні методи стиснення можуть реально вплинути на швидкість обробки та передавання великих обсягів даних. Метою було перевірити, наскільки FPGA здатне забезпечити приріст продуктивності порівняно з традиційними програмними підходами, особливо у потокових системах, де важливі не лише коефіцієнт стиснення, а й затримки. Робота поєднала аналіз алгоритмів, моделювання архітектури, створення HLS-модуля та експериментальну перевірку його ефективності на реальній SoC-платі.

Було побудовано функціональну модель системи та об’єктно-орієнтовану модель програмної частини, а також створено IDEF0-діаграму й UML-діаграму, які описують взаємодію ключових елементів: джерела даних, ARM-процесора, DMA-підсистеми, HLS-модуля стиснення та блоку зворотної передачі результатів. Завдяки цим моделям вдалося чітко визначити, як рухаються дані всередині системи, де виникають затримки й які компоненти найбільше впливають на загальну продуктивність.

Архітектура базується на використанні FPGA-модуля стиснення, створеного засобами High-Level Synthesis, та інтегрованого у SoC-платформу через AXI-Lite і AXI-Stream інтерфейси. На практиці HLS-реалізація показала себе ефективною: навіть з ростом масиву даних пропускна здатність залишалася високою завдяки конвеєризації, а навантаження на ARM-процесор було мінімальним. Недоліком є те, що продуктивність сильно залежить від структури даних – для різних типів вхідної інформації алгоритми RLE та LZ77 дають різні результати, що накладає вимоги на попередній аналіз потоку.

Водночас програмні реалізації на CPU виявилися помітно повільнішими на великих потоках даних. Порівняння показало: FPGA добре підходить для режимів реального часу, де важлива не лише компресія, а й стабільна низька затримка. Це підтверджують і результати вимірювань: прискорення становило кратні значення, особливо на даних із повторюваними структурами.

Гібридна взаємодія ARM – FPGA продемонструвала оптимальний баланс між гнучкістю програмної частини та швидкодією апаратної. ARM керує модулями, виконує підготовку даних і організовує DMA-передачу, тоді як FPGA бере на себе найбільш ресурсоємний етап – саме стиснення. Такий поділ обов'язків дозволив досягти стабільної роботи та мінімізувати затримки обміну.

У підсумку створена система стала повноцінним прототипом апаратно-програмного рішення для високошвидкісного стиснення, який поєднує продуктивність FPGA з гнучкістю програмного контролю. Вона забезпечує хороший баланс між швидкодією, ефективністю та можливістю масштабування. Результати роботи можуть бути використані як основа для подальших досліджень у сферах фінтеху, телекомунікацій, IoT та будь-яких систем, де обробка потокових даних відіграє ключову роль.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sayood K. Introduction to Data Compression. 5th ed. San Diego : Morgan Kaufmann, 2017. 768 p. URL: https://www.sciencedirect.com/book/9780128094747 (date of access: 13.11.2025).
2. Shannon C. E. A Mathematical Theory of Communication. Bell System Technical Journal. 1948. Vol. 27, no. 4. P. 623–656. URL: https://doi.org/10.1002/j.1538-7305.1948.tb00917.x (date of access: 13.11.2025).
3. Cover T. M., Thomas J. A. Elements of Information Theory. Hoboken : Wiley-Interscience, 2006. 748 p. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/047174882X (date of access: 13.11.2025).
4. Salomon D., Motta G. Handbook of Data Compression. 5th ed. London : Springer, 2010. 1360 p. URL: https://link.springer.com/book/10.1007/978-1-84882-903-9 (date of access: 13.11.2025).
5. Witten I. H., Neal R. M., Cleary J. G. Arithmetic Coding for Data Compression. Communications of the ACM. 1987. Vol. 30, no. 6. P. 520–540. URL: https://doi.org/10.1145/214762.214771 (date of access: 13.11.2025).
6. Ziv J., Lempel A. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory. 1977. Vol. 23, no. 3. P. 337–343. URL: https://doi.org/10.1109/TIT.1977.1055714 (date of access: 13.11.2025).
7. Ziv J., Lempel A. Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory. 1978. Vol. 24, no. 5. P. 530–536. URL: https://doi.org/10.1109/TIT.1978.1055934 (date of access: 13.11.2025).
8. Welch T. A. A Technique for High-Performance Data Compression. Computer. 1984. Vol. 17, no. 6. P. 8–19. URL: https://doi.org/10.1109/MC.1984.1659158 (date of access: 13.11.2025).
9. Deutsch P. DEFLATE Compressed Data Format Specification. RFC 1951. 1996. URL: https://www.rfc-editor.org/rfc/rfc1951 (date of access: 13.11.2025).

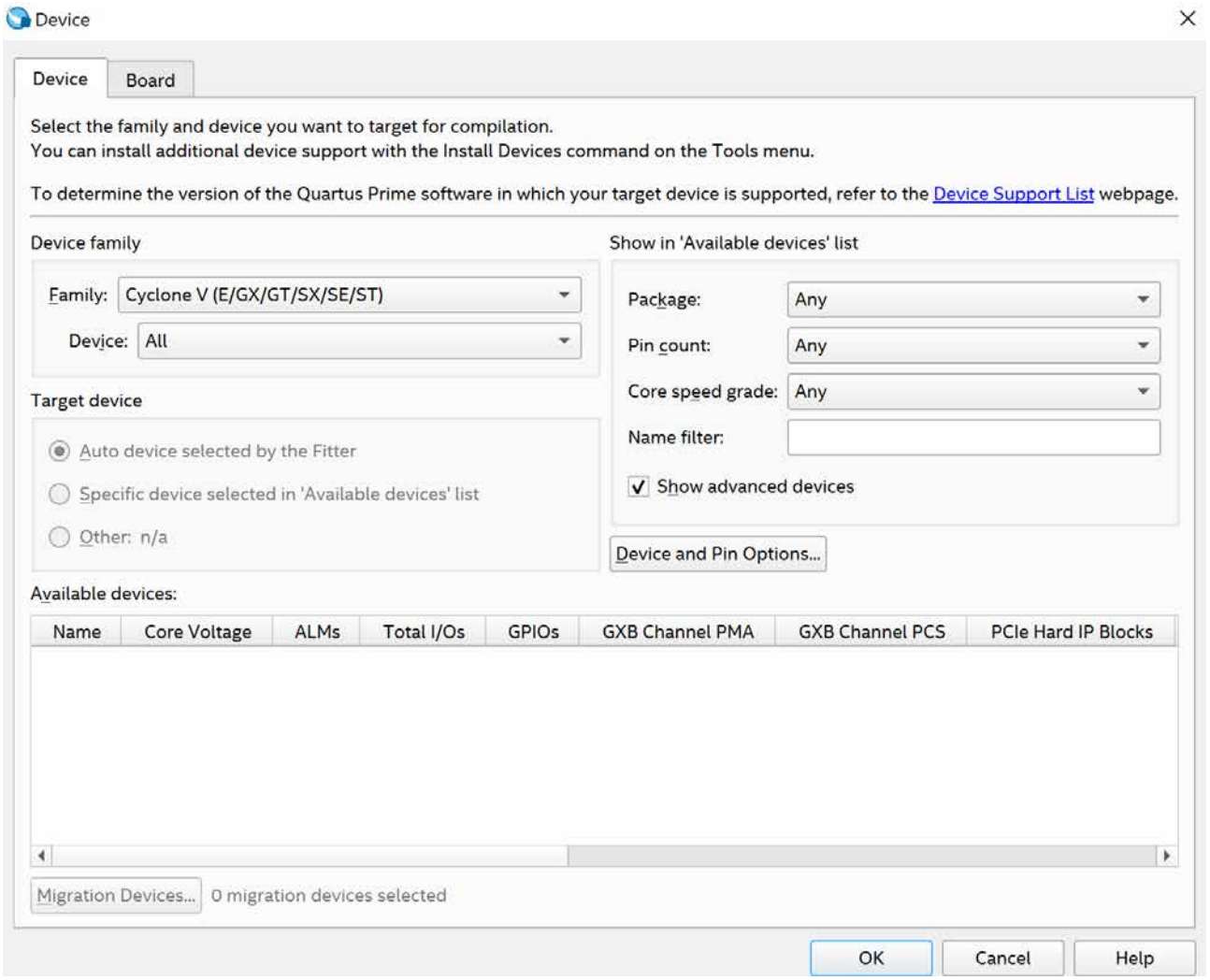
10. Alakuijala J. et al. Brotli Compressed Data Format. Google, 2016. URL: https://github.com/google/brotli (date of access: 13.11.2025).
11. Collet Y. Zstandard Compression Algorithm. Meta (Facebook), 2016. URL: https://facebook.github.io/zstd/ (date of access: 13.11.2025).
12. Pavlov I. 7-Zip Compression Algorithm (LZMA). 7-Zip Documentation, 2015. URL: https://www.7-zip.org/sdk.html (date of access: 13.11.2025).
13. Maxfield C. The Design Warrior's Guide to FPGAs. Oxford : Newnes, 2004. 542 p. URL: https://blog.aku.edu.tr/ismailkoyuncu/files/2017/04/01_ebook.pdf (date of access: 13.11.2025).
14. Intel Corporation. Cyclone V Device Handbook. Intel, 2023. URL: https://www.intel.com/content/www/us/en/docs/programmable/683130/ (date of access: 13.11.2025).
15. Choi J., Park S., Chung E. FPGA-Based High-Performance Lossless Compression Using LZ77 and Huffman Coding. IEEE Access. 2019. Vol. 7. P. 178153–178164. URL: https://scispace.com/pdf/design-of-fpga-based-lz77-compressor-with-runtime-4pj8bjliqk.pdf (date of access: 13.11.2025).
16. Matai J., Kim J.-Y., Kastner R. Energy efficient canonical huffman encoding. *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Zurich, Switzerland, 18–20 June 2014. 2014. URL: https://doi.org/10.1109/asap.2014.6868663 (date of access: 14.11.2025).
17. Design of FPGA-Based LZ77 Compressor With Runtime Configurable Compression Ratio and Throughput / S. Choi et al. *IEEE Access*. 2019. Vol. 7. P. 149583–149594. URL: https://doi.org/10.1109/access.2019.2947273 (date of access: 14.11.2025).
18. Deutsch P. DEFLATE Compressed Data Format Specification. RFC 1951. Internet Engineering Task Force, 1996. URL: https://www.rfc-editor.org/rfc/rfc1951 (date of access: 13.11.2025).
19. Intel Corporation. Cyclone V Device Handbook. Intel, Oct 2023. URL: https://cdrdv2-public.intel.com/666995/cv_5v2-683375-666995.pdf

20. Xilinx (AMD). Zynq-7000 SoC Technical Reference Manual (UG585). v1.14, June 30 2023. URL: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM
21. Lattice Semiconductor. ECP5 / ECP5-5G Family Data Sheet. August 2023. URL: https://static6.arrow.com/aropdfconversion/2beddaa15206c0bf1b59c250d0a95b9c979f950f/fpga-ds-02012-3-2-ecp5-ecp5g-family-data-sheet.pdf
22. Texas Instruments. Universal Asynchronous Receiver/Transmitter. Datasheet. Dallas : Texas Instruments, 2016. 36 p. URL: https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf (date of access: 14.11.2025).
23. USB Implementers Forum. Universal Serial Bus Revision 2.0 Specification. USB-IF, 2000. 650 p. URL: https://usb.org/document-library/usb-20-specification (date of access: 14.11.2025).
24. IEEE Standards Association. IEEE Std 802.3-2018: IEEE Standard for Ethernet. New York : IEEE, 2018. 3432 p. URL: https://standards.ieee.org/standard/802_3-2018.html (date of access: 14.11.2025).
25. PCI-SIG. PCI Express® Base Specification Revision 4.0, Version 1.0. Portland : PCI Special Interest Group, 2017. 765 p. URL: https://pcisig.com/specifications/pciexpress.
26. Intel Corporation. Intel® High-Level Synthesis Compiler: Reference Manual. Intel, 2023. 492 p. URL: https://www.intel.com/content/www/us/en/docs/programmable/683306/19-1/standard-edition-user-guide.html.
27. Siemens EDA (Mentor Graphics). QuestaSim User’s Manual. Version 2023.1. Wilsonville : Siemens Digital Industries Software, 2023. 842 p. URL: https://eda.sw.siemens.com/en-US/ic/questa/simulation.

ДОДАТКИ

Додаток А

Налаштування середовища Quartus



Додаток Б

Звіт компіляції проєкту в середовищі Quartus

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Nov 11 23:54:28 2025
Quartus Prime Version	24.1std.0 Build 1077 03/04/2025 SC Standard Edition
Revision Name	quartus_compile
Top-level Entity Name	quartus_compile
Family	Cyclone V
Device	5CSEBA6U23I7
Timing Models	Final
Logic utilization (in ALMs)	1,860 / 41,910 (4 %)
Total registers	3914
Total pins	0 / 314 (0 %)
Total virtual pins	432
Total block memory bits	4,096 / 5,662,720 (< 1 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Додаток В

Звіт використання ресурсів платформи після компіляції

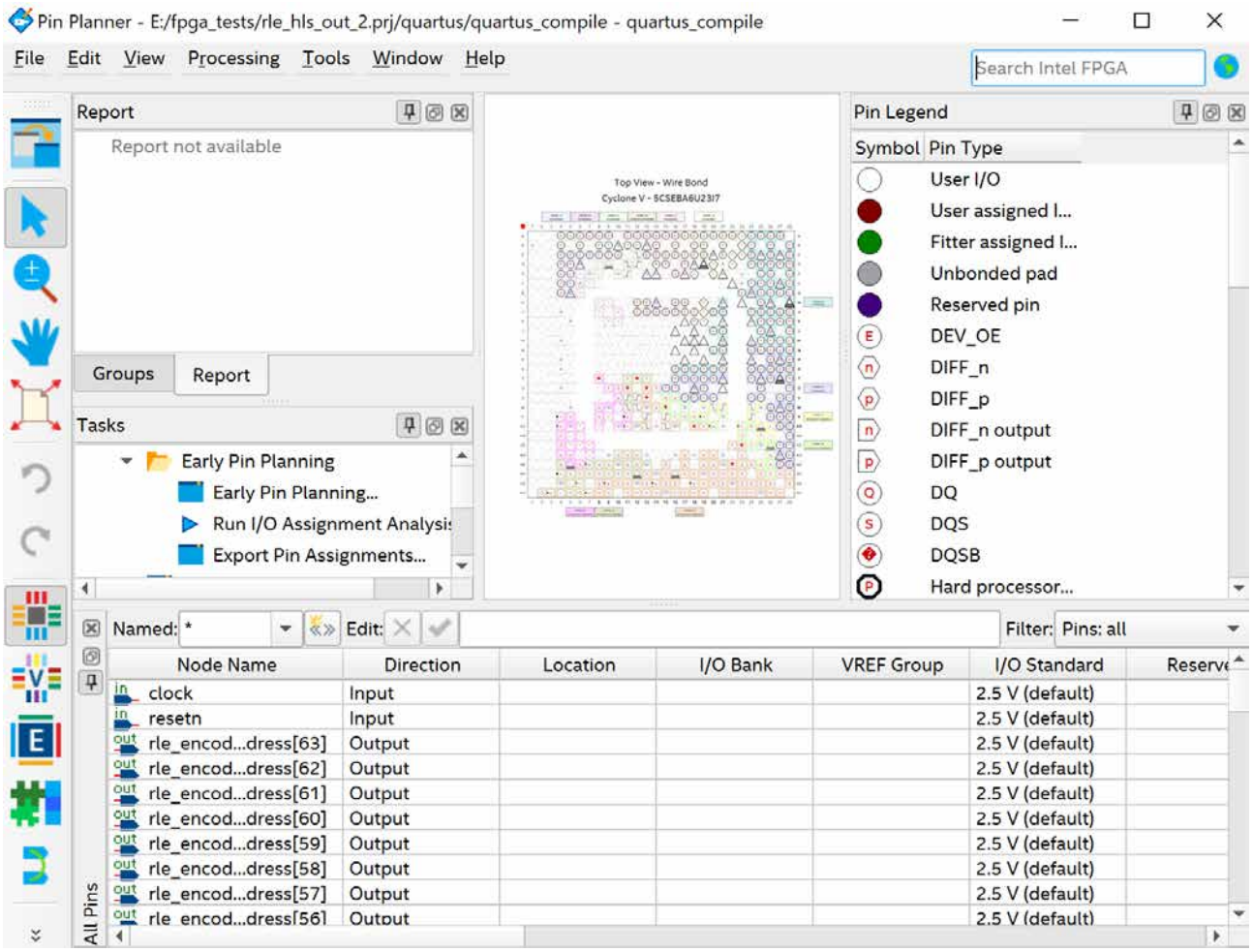
Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	2514
2		
3	▼ Combinational ALUT usage for logic	1558
1	-- 7 input functions	0
2	-- 6 input functions	153
3	-- 5 input functions	217
4	-- 4 input functions	229
5	-- <=3 input functions	959
4	▼ Memory ALUT usage	373
1	-- 64-address deep	0
2	-- 32-address deep	373
5		
6	Dedicated logic registers	4135
7		
8	Virtual pins	432
9	I/O pins	0
10	Total MLAB memory bits	7696
11	Total block memory bits	4096
12		
13	Total DSP Blocks	0
14		
15	Maximum fan-out node	clock
16	Maximum fan-out	4572
17	Total fan-out	24042
18	Average fan-out	3.66

Додаток Г

Планувальник пінів в середовищі Quartus



Додаток Ґ

Високорівневий код RLE для HLS компілятора

```
#include "HLS/hls.h"
#include <stdint.h>
using namespace ihc;

#define MAX_LEN 4096
#define MAX_OUT (MAX_LEN * 2)

component
void rle_encode(const char *in_data, char *out_data, unsigned int len, unsigned
int *out_len) {
    unsigned int i = 0, j = 0;
    while (i < len && j + 2 < MAX_OUT) {
        char cur = in_data[i];
        unsigned char count = 1;
        while (i + count < len && in_data[i + count] == cur && count < 255)
            count++;
        out_data[j++] = cur;
        out_data[j++] = (char)count;
        i += count;
    }
    *out_len = j;
}
```

Додаток Д

Високорівневий код LZ77 для HLS компілятора

```
#include "HLS/hls.h"
#include "HLS/hls_avalon_stream.h"
#include <stdint.h>

using namespace ihc;

#define MAX_INPUT_SIZE   65536
#define WINDOW_SIZE      1024
#define LOOKAHEAD_SIZE   32

struct LZ77Token {
    uint16_t offset;
    uint8_t  length;
    uint8_t  next;
};

component void lz77_compress_opt(
    const uint8_t   in[MAX_INPUT_SIZE],
    LZ77Token       out[MAX_INPUT_SIZE],
    int             input_size,
    int            *token_count
) {
    if (input_size < 0 || input_size > MAX_INPUT_SIZE) {
        *token_count = 0;
        return;
    }

    int i = 0;
    int out_idx = 0;

    while (i < input_size && out_idx < MAX_INPUT_SIZE) {
    #pragma ii 1
        int best_len    = 0;
        int best_offset = 0;

        int window_start = i - WINDOW_SIZE;
        if (window_start < 0) {
            window_start = 0;
        }

        for (int j = window_start; j < i; ++j) {
        #pragma unroll 4
```

Продовження додатку Д

```
            int k = 0;
            while (k < LOOKAHEAD_SIZE &&
                   (i + k) < input_size &&
                   in[j + k] == in[i + k]) {
                ++k;
            }

            if (k > best_len) {
                best_len    = k;
                best_offset = i - j;
            }
        }

        uint8_t next_char = 0;
        if (i + best_len < input_size) {
            next_char = in[i + best_len];
        }

        out[out_idx].offset = (uint16_t)best_offset;
        out[out_idx].length = (uint8_t)best_len;
        out[out_idx].next   = next_char;
        ++out_idx;

        if (best_len > 0) {
            i += best_len + 1;
        } else {
            ++i;
        }
    }

    *token_count = out_idx;
}

component void lz77_compress_stream(
    hls_avalon_stream<uint8_t>  &in_stream,
    hls_avalon_stream<LZ77Token> &out_stream,
    int                          input_size
) {
    uint8_t in_buf[MAX_INPUT_SIZE];

    int n = (input_size > MAX_INPUT_SIZE) ? MAX_INPUT_SIZE : input_size;

    for (int i = 0; i < n; ++i) {
    #pragma ii 1
```

Продовження додатку Д

```
    in_buf[i] = in_stream.read();
}

int i = 0;

while (i < n) {
#pragma ii 1
    int best_len    = 0;
    int best_offset = 0;

    int window_start = i - WINDOW_SIZE;
    if (window_start < 0) {
        window_start = 0;
    }

    for (int j = window_start; j < i; ++j) {
    #pragma unroll 4
        int k = 0;
        while (k < LOOKAHEAD_SIZE &&
               (i + k) < n &&
               in_buf[j + k] == in_buf[i + k]) {
            ++k;
        }

        if (k > best_len) {
            best_len    = k;
            best_offset = i - j;
        }
    }

    uint8_t next_char = 0;
    if (i + best_len < n) {
        next_char = in_buf[i + best_len];
    }

    LZ77Token tok;
    tok.offset = (uint16_t)best_offset;
    tok.length = (uint8_t)best_len;
    tok.next   = next_char;

    out_stream.write(tok);
```

Продовження додатку Д

```
            if (best_len > 0) {
                i += best_len + 1;
            } else {
                ++i;
            }
        }
    }
```