МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА

(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій

(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних систем та технологій

(повна назва кафедри (предметної, циклової комісії))

# Кваліфікаційна робота

на здобуття рівня вищої освіти «бакалавр»

(рівень вищої осівти)

**Проектування трійкової елементної бази для побудови обчислювальної системи**

**Designing a ternary element base for building a computer system**


Виконав: студент денної форми навчання

Спеціальність 123 Комп'ютерна інженерія

(шифр і назва напряму підготовки, спеціальності)

Окучинський Гавриїл Павлович

(прізвище, ім'я, по-батькові)

Керівник ст.викл. Мартинович Л.Я.

(науковий ступінь, вчене звання, прізвище та ініціали, підпис)

Рецензент проф. Гунченко Ю. О.

(науковий ступінь, вчене звання, прізвище та ініціали)


Рекомендовано до захисту:

Протокол засідання кафедри

№___ від «___»_____ 2025 р.

Завідувач кафедри

_____ Юрій ГУНЧЕНКО

(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___

протокол № __ від «___» ____ 2025 р.

Оцінка_____ /_____ /_____

(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ Світлана АНТОЩУК

(підпис) (ім'я, прізвище)

Одеса – 2025

# АНОТАЦІЯ

Дипломна робота присвячена дослідженню та реалізації трійкової (тритової) елементної бази для побудови обчислювальної системи. У роботі проаналізовано історію розвитку трійкової логіки та комп'ютерів, розглянуто математичні основи тризначної логіки та її порівняння з традиційною двійковою логікою. Наведено огляд відомих реалізацій трійкових обчислювальних пристроїв, зокрема комп'ютерів "Сетунь" та сучасних прототипів. Практична частина роботи містить опис розробки арифметико-логічного пристрою (АЛП) на трійковій логіці двома способами: програмною реалізацією на Python з графічним інтерфейсом користувача та апаратною мовою опису VHDL. Описано основні логічні та арифметичні операції в трійковій системі, особливості їх реалізації та використання. Розроблений трійковий АЛП дозволяє виконувати додавання, віднімання, множення, інкремент та базові логічні операції над трійковими числами. Графічний інтерфейс забезпечує зручність взаємодії з програмою та наочне відображення результатів. У висновках сформульовано основні результати роботи, показано переваги і перспективи трійкового підходу в обчислювальній техніці.

# ANNOTATION

The bachelor's thesis is devoted to the research and implementation of a ternary (three-valued) element base for building a computing system. The work analyzes the history of the development of ternary logic and computers, examines the mathematical foundations of three-valued logic and its comparison with traditional binary logic. A review of known implementations of ternary computing devices is provided, including the "Setun" computer and modern prototypes. The practical part of the work describes the design of an Arithmetic Logic Unit (ALU) based on ternary logic in two ways: a software implementation in Python with a graphical user interface, and a hardware implementation using the VHDL description language. The main logical and arithmetic operations in the ternary system are described, along with details of their implementation and usage. The developed ternary ALU can perform addition, subtraction, and basic logical operations on ternary numbers. The graphical interface provides convenient user interaction and visual display of results. In the conclusions, the main results of the work are summarized, and the advantages and prospects of the ternary approach in computing technology are demonstrated.

**ЗМІСТ**

Стор.

# ПЕРЕЛІК СКОРОЧЕНЬ

АЛП – арифметико-логічний пристрій (Arithmetic Logic Unit, ALU).

ЕОМ – електронна обчислювальна машина (комп’ютер).

VHDL – Very High Speed Integrated Circuit Hardware Description Language (мова опису апаратури).

GUI – Graphical User Interface (графічний інтерфейс користувача).

МС – мультизначна (многозначна) логіка (MVL, Multi-Valued Logic).

PDP – Power-Delay Product – добуток потужності й затримки сигналу; комплексний показник ефективності цифрової схеми, що відображає енергетичні витрати на виконання однієї логічної операції.

# ВСТУП

Сучасна комп’ютерна інженерія майже повністю базується на двійковій логіці, яка оперує двома станами – “0” та “1”. Двійкова система стала домінуючою завдяки простоті технічної реалізації – транзистори та інші електронні компоненти природно працюють у двох станах (відключено/включено). Водночас, ще на початку розвитку обчислювальної техніки дослідники розглядали можливість використання більш ніж двох логічних станів. Зокрема, трійкова логіка (тризначна логіка) з трьома станами привертає увагу вчених вже майже століття. Вона є найпростішим розширенням двійкової ідеї і теоретично може забезпечити більш високу ефективність представлення та обробки інформації.

Трійкова обчислювальна система потенційно здатна зменшити обсяг даних для представлення інформації та підвищити швидкодію окремих операцій. Згідно з теоретичними оцінками, система числення з основою 3 є найефективнішою серед цілих основ з точки зору “щільності” кодування інформації. Дійсно, для представлення числа заданого діапазону потрібно менше тритів (трійкових розрядів), ніж бітів, що обіцяє виграш у пам’яті та пропускній здатності. Крім того, тризначна логіка дозволяє в деяких випадках спростити логічні операції. Наприклад, порівняння двох чисел на більше/менше/рівність у трійковій логіці можна виконати однією операцією, отримуючи одразу три можливі результати, тоді як у двійковій системі для цього потрібна комбінація з двох бінарних перевірок. Трійкова симетрична система числення має ту перевагу, що додатні і від’ємні числа обробляються однаково, без потреби у спеціальному поданні від’ємних чисел (як, наприклад, додатковий код у двійковій системі. Це може спростити і прискорити апаратну реалізацію арифметичних операцій.

Метою даної роботи є проектування елементної бази та основних компонентів обчислювальної системи, що функціонує на принципах трійкової логіки. Зокрема, необхідно розробити арифметикологічний

пристрій (АЛП), здатний виконувати базові операції над трійковими числами, продемонструвати його роботу як програмними методами, так і на рівні опису апаратури. Це дозволить оцінити практичну реалізовність трійкового підходу, його переваги та недоліки.

Для досягнення поставленої мети в роботі вирішуються такі завдання:

- проаналізувати історичний розвиток трійкової (тризначної) логіки та існуючих трійкових обчислювальних машин;
- здійснити огляд реалізованих рішень – від перших трійкових комп’ютерів до сучасних прототипів та дослідницьких проектів;
- розробити арифметико-логічний пристрій на основі трійкової логіки засобами програмування (на мові Python), що включає модель виконання логічних і арифметичних операцій над трійковими числами;
- реалізувати VHDL-модель трійкового АЛП, яка описує роботу пристрою на рівні апаратури і може бути перевірена засобами симуляції;
- створити графічний інтерфейс користувача для демонстрації роботи трійкового АЛП, забезпечивши зручність введення даних і відображення результатів;
- проаналізувати результати реалізації, підтвердити правильність роботи трійкового АЛП та зробити висновки щодо доцільності використання трійкової логіки.

Об’єкт дослідження – процеси та пристрої обчислювальної техніки, що використовують багатозначну (тризначну) логіку.

Предметом дослідження є методи побудови арифметико-логічного пристрою на основі трійкової елементної бази та особливості реалізації трійкових операцій.

Методи дослідження обрано аналіз літературних джерел з теорії багатозначної логіки та історії комп’ютерів; математичне моделювання

операцій у трійковій системі; програмування мовою Python для створення програмної моделі АЛП; опис цифрової схеми мовою VHDL та її перевірка засобами моделювання.

Наукова новизна і практична значущість. Наукова новизна роботи полягає в узагальненні теоретичних і практичних підходів до проектування трійкових обчислювальних систем на сучасному етапі. Попри тривалу історію, трійкова логіка не набула широкого практичного застосування, тож актуальним є дослідження можливостей її реалізації із застосуванням сучасних засобів розробки. У практичному аспекті значущість роботи полягає у створенні діючого прототипу трійкового АЛП, який може бути використаний для навчальних цілей, демонстрації принципів багатозначної логіки, а також слугувати основою для подальших досліджень у галузі нестандартних обчислювальних архітектур.

Дипломна робота складається зі вступу, двох розділів основної частини, висновків і списку використаних джерел. У вступі обґрунтовано актуальність теми, сформульовано мету і завдання, визначено об’єкт, предмет і методи дослідження. Розділ 1 містить теоретичний матеріал: розглянуто історичний розвиток трійкової логіки та комп’ютерів, викладено математичні основи тризначної логіки, проведено порівняння з двійковою логікою, а також здійснено огляд відомих реалізацій трійкових обчислювальних систем. Розділ 2 присвячено практичній реалізації: описано розробку трійкового АЛП мовою Python (разом з графічним інтерфейсом користувача) та на VHDL, детально пояснено реалізацію основних логічних і арифметичних операцій, а також розглянуто особливості функціонування створеного програмно-апаратного забезпечення. У висновках підбито підсумки виконаної роботи, сформульовано основні результати та рекомендації щодо подальших досліджень.

# 1 ТЕОРЕТИЧНІ ОСНОВИ ТРІЙКОВОЇ ЛОГІКИ ТА АЛЬТЕРНАТИВНИХ ОБЧИСЛЕНЬ

## 1.1 Історія та розвиток трійкової логіки

Трійкова (тернарна) логіка є однією з форм багатозначної логіки, у якій логічні величини можуть приймати більше ніж два значення. Концепцію трьохзначної логіки вперше запропонував польський логік Ян Лукасевич у 1920-х роках [1]. Його підхід мав велике філософське значення, однак згодом трійкова логіка отримала інженерне втілення в цифрових системах. Вона дозволяє описати не лише істину та хибність, а й проміжні стани, такі як невизначеність або часткова істина. Завдяки цій ідеї з’явилась можливість розвитку багатозначних логічних систем для моделювання реальних процесів із нечіткими параметрами [2].

У 1958 році в СРСР було створено перший у світі трійковий комп’ютер «Сетунь» під керівництвом М.М. Брусєнцова. Обчислювальна машина використовувала симетричну трійкову систему числення та мала низку переваг: менше логічних елементів для реалізації операцій, менше споживання пам’яті, вища швидкодія. Однак через складність фізичної реалізації трирівневих сигналів та відсутність відповідних виробничих технологій подальший розвиток трійкової техніки було припинено. Незважаючи на це, у Японії та США впродовж 1990–2010-х років періодично повертались до ідеї трійкових обчислень, особливо в галузі квантових технологій та наноелектроніки [3].

## 1.2 Симетрична трійкова система числення

Симетрична трійкова система базується на множині значень {-1, 0, +1}. Вона дозволяє уникати окремого зберігання знаку числа, що особливо корисно при обробці чисел із плаваючою або фіксованою точкою. На

відміну від класичної (несиметричної) трійкової системи з основою 3 і цифрами {0,1,2}, симетрична система є математично зручнішою для реалізації логічних схем і дозволяє досягти більшого ступеня уніфікації в архітектурі обчислювальних пристроїв [4].

Наприклад:

- Десяткове 0 = 0 (трійкове)
- Десяткове 1 = +1
- Десяткове -1 = -1
- Десяткове 2 = +1 -1
- Десяткове -2 = -1 +1
- Десяткове 3 = +1 0

Усі арифметичні дії над числами у цій системі можна реалізовувати без необхідності додаткового переносу, використовуючи методи обчислень із фіксованою точкою. Реалізація трійкового АЛП (арифметико-логічного пристрою) вимагає створення ефективної логіки перетворення тритів у внутрішні сигнали, які можна апаратно обробляти [5].

## 1.3 Порівняння трійкової та двійкової логіки

Традиційно цифрові системи будуються на основі двійкової логіки, яка оперує двома станами: 0 і 1. Однак трійкова логіка, що використовує три стани (наприклад, –1, 0, +1 у симетричному варіанті), пропонує альтернативу з потенційно вищою інформаційною ефективністю та новими підходами до побудови обчислювальних структур.

Основні відмінності між двійковою та трійковою логікою наведено в таблиці 1.1.

Таблиця 1.1 — Порівняння двійкової та трійкової логіки

| Параметр | Двійкова логіка | Трійкова логіка |
|---|---|---|
| Кількість логічних станів | 2 (0, 1) | 3 (–1, 0, +1) |
| Щільність інформації | 1 біт | ≈ 1.584 біт/трит |
| Обчислювальна складність | Вища | Потенційно нижча |
| Фізична реалізація | Добре відпрацьована | Складніша (особливо в CMOS) |
| Стандартизація | Висока | Практично відсутня |
| Енергоспоживання | Відносно високе | Потенційно нижче (менше переходів) |

Завдяки трьом станам, трит містить більше інформації, ніж біт, що дозволяє зменшити середню розрядність чисел при збереженні того ж самого обсягу даних. Наприклад, для представлення одного байта (8 біт) у трійковій логіці достатньо лише 6 тритів.

Крім того, деякі логічні операції в трійковій логіці можуть бути реалізовані більш компактно, що потенційно зменшує кількість необхідних транзисторів. Проте основним бар’єром до широкого впровадження трійкової логіки залишається складність її фізичної реалізації, відсутність усталених стандартів і потреба у специфічному обладнанні для генерації та розпізнавання трьох рівнів сигналу.

Попри це, трійкова логіка викликає інтерес як у науковців, так і у розробників експериментальних систем, оскільки вона дозволяє переосмислити принципи побудови цифрових пристроїв з точки зору енергоефективності, щільності інформації та обчислювальної потужності.

## 1.4 Актуальність досліджень

Сучасна комп’ютерна інженерія стикається з обмеженнями класичної двійкової логіки: енергоспоживання, швидкість обробки, розміри чипів. Трійкова логіка дозволяє зменшити кількість міжз’єднань, скоротити площу схеми та підвищити обчислювальну потужність. Саме тому наукова спільнота проявляє підвищений інтерес до тернарних систем.

Зокрема, актуальним напрямом є використання трійкової логіки в:

- квантових комп’ютерах (логіка qutrit);
- нейромережевих прискорювачах;
- логіці із нечіткими станами (fuzzy logic);
- комп’ютерному моделюванні й обробці мовних сигналів [6].

У майбутньому трійкова логіка може стати базовою для нових стандартів представлення даних у мікро- та наноелектроніці.

## 1.5 Приклади реалізації трійкових обчислень

Крім історичного комп’ютера «Сетунь», у США в рамках навчальних програм у 2000-х створювались емулятори тернарних процесорів на основі FPGA. У Японії досліджувалась можливість використання трійкових вентилів у проектах контролерів для робототехніки. В окремих роботах реалізовано трійкову арифметику в середовищах ModelSim і Verilog, що дозволило дослідити поведінку нестандартних логік при великій кількості змінних. З використанням оптимізованих арифметичних алгоритмів у трійкових схемах спостерігається покращення показника PDP на 36,8 % порівняно з аналогами на двійковій логіці [7].

# 2 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТРІЙКОВОГО АРИФМЕТИКО-ЛОГІЧНОГО ПРИСТРОЮ

## 2.1 Постановка задачі

Метою даної роботи є розробка програмного та апаратного забезпечення моделі трійкового арифметико-логічного пристрою (АЛП), що функціонує на основі симетричної трійкової логіки. Іншими словами, необхідно спроєктувати та реалізувати арифметико-логічний пристрій – ключовий компонент обчислювальної системи, що виконує арифметичні та логічні операції над даними. На відміну від класичного двійкового АЛП, який оперує бітою логікою (0 і 1), у цьому проєкті використовується тризначна (трійкова) логіка**,** де кожен трит (трійковий розряд) може набувати одного з трьох значень: -1, 0 або +1 (симетрична система). Такий підхід ґрунтується на перевагах симетричної трійкової логіки, розглянутих у теоретичній частині: підвищення інформаційної ємності, ефективніше кодування від'ємних чисел та потенційне зменшення кількості логічних елементів в апаратурі за рахунок скорочення розрядності чисел [2].

Для досягнення поставленої мети визначено основні вимоги до системи:

- Підтримка 8 базових операцій над трійковими числами: арифметичних – додавання, віднімання, інкремент (збільшення на 1) та множення; логічних – AND (кон'юнкція), OR (диз'юнкція), NOT (заперечення) та IMPL (логічна імплікація).
- Послідовне введення операндів – система повинна приймати багатоцифрові трійкові операнди шляхом подання тритів послідовно, по одному за раз (аналогічно тому, як у реальних схемах цифри вводяться послідовно у регістр).

- Подвійна реалізація: розробка програмної моделі АЛП мовою Python (для демонстрації роботи та зручності тестування) та апаратурної моделі на мові опису апаратури VHDL. Обидві реалізації повинні виконувати однаковий набір операцій і давати тотожні результати.
- Графічний інтерфейс користувача (GUI) для програмної моделі – забезпечення інтерактивної взаємодії з користувачем, включно з зручним введенням трійкових чисел, вибором операції, відображенням результатів, а також додатковими функціями (довідка, історія обчислень, налаштування).
- Система довідки, історії та налаштування – програмна реалізація повинна містити допоміжні модулі: довідкову систему (для пояснення користувачеві правил роботи), журнал (історію) виконаних обчислень та функцію налаштування для таких об'єктів інтерфейсу як фон, кнопки та текст.
- Верифікація роботи моделі шляхом симуляції в середовищі Quartus Prime Lite – апаратна частина має бути перевірена у програмній моделі (без фактичного завантаження у фізичний пристрій), для підтвердження коректності виконання всіх трійкових операцій згідно з передбаченою логікою керування.

Таким чином, у цьому розділі буде описано структуру та реалізацію трійкового АЛП згідно з наведеними вимогами: спочатку розглянуто загальну архітектуру системи та використану станну машину, далі подано формат кодування операцій і представлення даних, після чого детально описано програмну (Python) та апаратну (VHDL) реалізації, включно з графічним інтерфейсом та допоміжними модулями. Для кожної із підтримуваних операцій наведено алгоритми реалізації та таблиці істинності, що ілюструють їх роботу.

## 2.2 Загальна архітектура системи

Проєктований трійковий АЛП має модульну структуру і розділяється на окремі логічні блоки. На верхньому рівні можна виділити такі основні компоненти:

- Блок прийому даних – здійснює послідовне введення тритів операндів від користувача або з тестового середовища. Цей блок відповідає за прийом кожної цифри трійкового числа і передачу її у відповідний буфер.
- Буфери збереження операндів A і B – регістри, що накопичують введені трити, формуючи повні операнди. Окремий буфер передбачено для першого операнда A та для другого B. Після завершення введення числа буфер зберігає його трійкове представлення до моменту виконання операції.
- Декодер коду операції – приймає введений код операції та активує відповідну операційну функцію АЛП. В залежності від двотритового коду декодер генерує сигнали керування для вибору потрібної операції (арифметичної чи логічної).
- Арифметико-логічний обчислювач (ядро) – основний функціональний блок, що безпосередньо виконує обчислення над операндами. Саме в цьому ядрі реалізовано алгоритми додавання, віднімання, інкременту, множення, а також логічних операцій AND, OR, NOT, IMPL для трійкових даних. Ядро отримує на вході два операнди (A, B) та код операції, а на виході формує результат R.
- Блок формування результату – здійснює обробку отриманого від ядра результату перед подачею його на вивід. Зокрема, може виконуватися перетворення формату (напрям, нормалізація

знакового представлення, усунення провідних нулів-тритів тощо) перед відображенням результату.

– Інтерфейс виведення результату – забезпечує відображення результату користувачеві (у випадку програмної моделі – це графічне поле або консоль, у випадку апаратної – умовний вихідний порт або регістр результату). В контексті GUI даний інтерфейс представлений елементами вікна програми, що показують результат обчислення у трійковому форматі.

Координація роботи всіх зазначених компонент здійснюється спеціальним принципом керування, який визначає порядок переходів між станами вводу, обчислення та виводу. Нижче розглянуто принцип дії цього керування.

## 2.3 Принцип керування виконанням операцій

У реалізованому трійковому арифметико-логічному пристрої (АЛП) управління виконанням обчислень здійснюється по-різному залежно від середовища: у програмній реалізації на Python використано подієво-орієнтовану модель, тоді як в апаратній реалізації мовою VHDL застосовано синхронний підхід на основі тактового сигналу без побудови станової машини.

У програмній частині проєкту, реалізованій мовою Python з використанням графічної бібліотеки Tkinter, керування обчислювальним процесом побудовано на взаємодії користувача з графічним інтерфейсом. Користувач вводить операнди у вигляді рядків, що містять символи "-", "0" та "+", які відповідають тритам зі значеннями -1, 0 та +1 відповідно. Обчислення ініціюються натисканням на одну з кнопок, що відповідають арифметичним або логічним операціям. У відповідь на подію натискання викликається відповідна функція обробки, яка зчитує введені рядки, перетворює їх у внутрішній формат (списки тритів), обчислює результат за

допомогою відповідного методу класу TernaryALU та виводить результат назад у рядковому вигляді. Усі дії виконуються послідовно, миттєво після натискання кнопки, без затримок, таймерів або керуючих сигналів. Весь цикл — від зчитування даних до відображення результату — відбувається синхронно в межах одного програмного виклику, що відповідає стандартній подієво-орієнтованій архітектурі GUI-додатків.

На відміну від цього, апаратна реалізація на VHDL використовує синхронну модель, в якій виконання операції запускається зовнішнім сигналом start під час активного фронту тактового сигналу clk. У модулі передбачено сигнали reset, start, op_code, а також входи для операндів a_in і b_in, кожен з яких має довжину 6 біт і представляє три трити по два біти. При активації сигналу reset усі регістри та вихідні сигнали скидаються до нульового стану. Якщо сигнал start встановлено в логічну 1, і настає фронт clk, пристрій аналізує значення op_code і визначає, яку з восьми підтримуваних операцій потрібно виконати.

Арифметичні операції (додавання, віднімання, інкремент, множення) реалізуються шляхом попереднього перетворення вхідних тритів у цілі десяткові значення. Далі виконується відповідна операція, після чого результат знову перетворюється у тритовий формат із двобітовим кодуванням кожного трита. Логічні операції (AND, OR, NOT, IMPL) не потребують переходу до десяткового представлення — кожен трит обробляється окремо відповідно до визначеної таблиці істинності, і результат формується безпосередньо на рівні логіки.

Після завершення обчислення результат подається на вихід trit_out у вигляді 6-бітового вектора, а сигнал done встановлюється у логічну 1, що свідчить про завершення виконання. Весь обчислювальний цикл в апаратній реалізації завершується протягом одного тактового циклу після подачі start, без будь-яких проміжних станів або багататактної логіки. Керування побудовано у вигляді оператора case, що дозволяє обрати реалізацію певної операції на основі коду op_code, а у разі некоректного коду

використовується гілка when others, яка встановлює виходи в нульовий стан або залишає їх без змін.

Таким чином, програмна реалізація орієнтована на взаємодію з користувачем у реальному часі і здійснює обчислення в момент натискання на елемент інтерфейсу, тоді як апаратна реалізація працює на базі тактового сигналу та реагує на зовнішні керуючі сигнали, виконуючи операцію за один цикл. Обидва підходи демонструють однакову функціональність, але реалізують принцип керування відповідно до особливостей своїх платформ.

## 2.4 Формат кодування операцій

Кожна з восьми підтримуваних операцій має закріплений двотритовий код (чотири біти), яким вона задається в системі. Вибір коду здійснено таким чином, щоб розрізняти арифметичні та логічні операції та спростити декодування. У таблиці 2.1 наведено відповідність між кодом і операцією:

Таблиця 2.1 – Відповідність кодів операціям

| Код (2 трити) | Операція |
|---|---|
| 00 00 | додавання (ADD) |
| 00 01 | віднімання (SUB) |
| 11 10 | інкремент (INC) |
| 00 11 | множення (MUL) |
| 01 00 | кон’юнкція (AND) |
| 01 01 | диз’юнкція (OR) |
| 11 11 | заперечення (NOT) |
| 01 11 | імплікація (IMPL) |

Декодер операції, отримавши двотритовий код op_code, генерує сигнал вибору відповідної гілки операції всередині ядра АЛП. Під час моделювання на VHDL цей код використовується в операторі case для активації відповідного фрагмента архітектури (див. лістинг А6). У програмній моделі Python код операції інтерпретується програмно: він може надходити безпосередньо від GUI (наприклад, натиснення на кнопку операції) або задаватися при автоматичному тестуванні; у будь-якому разі, відбувається його відповідність до потрібної функції класу АЛП.

## 2.5 Внутрішнє представлення тритів

Реалізація трійкової логіки в цифровому середовищі потребує вибору зручного представлення тритів. У симетричній трійковій системі значення одного трита фізично можна закодувати кількома бітами [2]. В даному проєкті використано уніфіковане подання, при якому кожен трит відображається двома бітами. Кодовані значення обрано наступним чином:

- 00 – представляє трійкове значення -1 (мінімальне значення, логічний “False”).
- 01 або 10 – представляє 0 (нейтральне значення). Допускається дві різні бітові комбінації для нульового трита – це зроблено для спрощення реалізації перенесень та деяких логічних функцій, забезпечуючи симетрію кодування.
- 11 – представляє +1 (максимальне значення, логічний “True”).

Таким чином, симетричні крайні значення -1 та +1 мають однозначні бітові коди, а нейтральний 0 може бути закодований двома еквівалентними способами (01 і 10). Подвійне кодування нуля використовується задля уніфікації алгоритмів: воно дозволяє реалізувати, наприклад, інкремент і додавання без спеціальних випадків, а також полегшує побудову переходів між станами -1, 0, +1 у схемі (за рахунок можливості зміни лише одного біта

при переході 0→+1 або 0→-1). У апаратній моделі VHDL трити оголошені як двобітові вектори типу std_logic_vector(1 downto 0) . У програмній моделі Python використовується ціле представлення триту (ціле число -1, 0 або 1) при обчисленнях, але при обміні даними з користувачем застосовується строковий формат із символами '-', '0' і '+' .

Варто зауважити, що симетричне кодування полегшує реалізацію арифметичних операцій. Зокрема, додавання у такому коді часто не потребує складного переносного механізму: сума тритів -1 і +1 дає 0 без переносу, а сума +1 і +1 еквівалентна -1 з переносом +1 у старший розряд, що простіше обробити, ніж у двійковому випадку [2]. Ці властивості було враховано при програмуванні алгоритмів додавання і інкременту.

## 2.6 Програмна реалізація трійкового АЛП (Python)

Python-модель АЛП реалізована у вигляді окремого модуля TernaryALU (див. лістинг А.2 ). Даний модуль містить клас TernaryALU, який інкапсулює всю логіку роботи з трійковими числами: від розбору рядкових вводів до виконання операцій та формування результатів. Програмна архітектура побудована таким чином, щоб підтримувати довільну кількість тритів у операндах (числа можуть бути будь-якої довжини) і забезпечувати коректність операцій для різних випадків (у тому числі граничних значень та помилкового вводу).

Основні компоненти класу TernaryALU:

- Функції перетворення і форматування: метод parse(str) перетворює рядок, що містить трійкове число (напр. “-0+”), у внутрішній список тритів [-1, 0, 1]. Навпаки, метод format(list) виконує форматування списку тритів назад у рядок символів “-0+” для зручного відображення результатів. Допоміжні методи char_to_trit(c) та trit_to_char(t) виконують перетворення окремого символу триту в числове значення (-1,0,1) і навпаки.

При першій згадці символів '-', '0' та '+' у тексті GUI або консолі користувача, вони пояснюються як від'ємний, нульовий та додатний трійкові розряди відповідно.

- Нормалізація довжин операндів: метод _normalize(a, b) приймає два списки тритів (операнди A і B) та доповнює їх нульовими тритами спереду до однакової довжини. Це необхідно для коректного виконання побітових логічних операцій над числами різної довжини та для вирівнювання розрядності перед арифметичним додаванням. Наприклад, якщо A має довжину 3 трити, а B – 2 трити, то до B буде додано один провідний нуль-трит.
- Арифметичні операції: методи add(a, b), sub(a, b), inc(a) та mul(a, b) реалізують додавання, віднімання, інкремент і множення відповідно. У програмній моделі прийнято рішення виконувати ці операції через проміжне десяткове представлення для спрощення алгоритмів: списки тритів операндів спочатку конвертуються у цілі числа (де -1, 0, +1 трактуються як цифри у базі 3), виконується відповідна операція над цими цілими (додавання, віднімання тощо), після чого результат переводиться назад у трійковий формат. Такий підхід є допустимим на програмному рівні і гарантує правильність обчислень, оскільки використовуються вбудовані цілочисельні операції Python. Конвертація реалізована методами ternary_to_decimal(trits) та decimal_to_ternary(n). Останній метод особливо цікавий тим, що враховує специфіку симетричного представлення: якщо при діленні остача дорівнює 2, він інтерпретує її як -1 з переносом +1 у старший розряд (тобто 2 в балансовій системі еквівалентно -1 з утворенням додаткового триту). Це дозволяє правильно перевести, скажімо, десяткове

число 5 у збалансовану трійкову форму (+1 -1, що відповідає +1 з переносом -1). Для прикладу, прототип реалізації функції додавання наведено у лістингу А.2 .

– Логічні операції: методи t_and(a, b), t_or(a, b), t_not(a) та t_impl(a, b) реалізують логічні функції AND, OR, NOT та IMPL відповідно над трійковими операндами. На відміну від арифметичних, ці операції реалізовані побітно, без переходу до десяткового. Перед виконанням логічних операцій операнди також нормалізуються до однакової довжини (використовується _normalize). Алгоритми реалізації логічних функцій базуються на визначеннях тризначної логіки:
– Кон’юнкція (AND) – реалізовано через вибір мінімуму між двома тритами на кожній позиції: t_and(a,b) = [min(x, y) for x,y in zip(a,b)]. Таким чином, якщо хоч один із відповідних тритів операндів дорівнює -1 (логічна “хиба”), результат на цій позиції буде -1; якщо жоден не є -1, але хоча б один 0 (невизначеність), результат буде 0; і лише коли обидва трити +1 (логічна “істина”), результат +1.
– Диз’юнкція (OR) – реалізовано через вибір максимуму: t_or(a,b) = [max(x, y) for x,y in zip(a,b)]. Таким чином, якщо хоча б один з тритів дорівнює +1 (“істина”), результат буде +1; якщо жоден не +1, але хоча б один 0, результат 0; і лише коли обидва трити -1 (“хиба”), результат -1.
– Заперечення (NOT) – реалізовано як поелементна зміна знаку: t_not(a) = [-x for x in a]. Кожен трит операнда інвертується: +1 перетворюється на -1, -1 на +1, а нейтральний 0 залишається 0 (невизначеність заперечується до невизначеності).
– Імплікація (IMPL) – реалізовано за логічною формулою імплікації: A → B еквівалентно ¬A ∨ B. У трійковій

інтерпретації це можна подати як t_impl(a,b) = [max(-x, y) for x,y in zip(a,b)]. Тобто для кожної позиції обчислюється максимум між запереченим значенням триту A та відповідним тритом B. Це відповідає правилу: якщо A = +1 (істина), тоді значення імплікації дорівнює значенню B; якщо A = -1 (хиба), то імплікація автоматично істинна (+1), незалежно від B; якщо ж A = 0 (невизначено), то значення імплікації збігається із значенням B у випадку істини (+1) або залишається невизначеним (0) і хибним не стає.

Для ілюстрації роботи наведених логічних функцій нижче подано відповідні таблиці істинності для кожної з них (табл. 2.2 – 2.5). Тут символи -1, 0 та +1 інтерпретуються, відповідно, як False, Unknown та True в термінах логіки.

Таблиця 2.2 – Таблиця істинності операції AND (A ∧ B)

| A \ B | -1 | 0 | +1 |
|---|---|---|---|
| -1 | -1 | -1 | -1 |
| 0 | -1 | 0 | 0 |
| +1 | -1 | 0 | +1 |

Таблиця 2.3 – Таблиця істинності операції OR (A ∨ B)

| A \ B | -1 | 0 | +1 |
|---|---|---|---|
| -1 | -1 | 0 | +1 |
| 0 | 0 | 0 | +1 |
| +1 | +1 | +1 | +1 |

Таблиця 2.4 – Таблиця істинності операції IMPL (A → B)

| A \ B | -1 | 0 | +1 |
|---|---|---|---|
| -1 | +1 | +1 | +1 |
| 0 | 0 | 0 | +1 |
| +1 | -1 | 0 | +1 |

Таблиця 2.5 – Таблиця істинності операції NOT (¬A)

| A | ¬A |
|---|---|
| -1 | +1 |
| 0 | 0 |
| +1 | -1 |

Як видно з наведених таблиць, реалізація логічних операцій узгоджується зі стандартними законами тризначної логіки (наприклад, для імплікації істинне значення передбачає, що при хибному A висновок завжди істинний, а при істинному A значення імплікації збігається зі значенням B). Усі функції класу TernaryALU повертають новий список тритів, що є результатом операції, не змінюючи вихідні операнди (таким чином, реалізація є функціонально чистою, що полегшує відлагодження). Перед обробкою усі трити операндів проходять нормалізацію і перевірку на допустимість, що гарантує коректність роботи навіть у випадку різної довжини чисел або невалідних символів вводу (при виявленні помилки вводу генерується виняток ValueError, який обробляється у вищому рівні програми, сигналізуючи користувачу про некоректні дані).

На рівні модулів програми, окрім класу TernaryALU, реалізовано декілька допоміжних компонентів, що забезпечують інтерактивність та зручність роботи користувача з АЛП.

## 2.7 Графічний інтерфейс та допоміжні модулі програми

Програмна реалізація трійкового АЛП оснащена графічним інтерфейсом користувача (GUI) на базі бібліотеки Tkinter (стандартний інструмент для побудови GUI в Python). Графічний інтерфейс значно полегшує взаємодію з системою, дозволяючи користувачу вводити трійкові числа та обирати операції у зручний спосіб, без необхідності ручного кодування у консолі.

На рисунку 2.1 зображено головне вікно програми-емулятора трійкового АЛП. Основні елементи інтерфейсу такі:

- Поля введення для двох операндів (Operand A і Operand B), куди користувач може ввести числа у трійковому форматі, використовуючи символи “-”, “0” і “+”. Валідація введення забезпечує, що жодні інші символи (наприклад, цифри 2, 3 або літери) не приймаються – у разі хибного символу система видає повідомлення про помилку.
- Набір кнопок для вибору операції. Операції представлені їхніми назвами (ADD, SUB, MUL тощо). При виборі операції її код автоматично формується і передається в логіку програми та починає обчислення.
- Поле виведення результату – відображає отриманий результат операції у трійковому форматі. Також відображае повідомлення у випадку некоректного вводу.
- Історія обчислень: демонструє список останніх виконаних операцій. Програма зберігає історію у модулі history.py, що керує додаванням нових записів (формула операції та результат) після кожного обчислення (рис. 2.2). Реалізацію модуля історії наведено у лістингу А.3.

– Налаштування теми: відкриває діалог (рис. 2.3) для вибору кольорової теми інтерфейсу. В модулі theme_manager.py (див. лістинг А.4) реалізовано механізм вибору кольору з палітри для тексту, фону та кнопок.
– Довідкова система: довідку реалізовано у модулі help_system.py (див. лістинг А.5). У довідці, зокрема, описано призначення кожної з 8 операцій, формат введення трійкових чисел, а також наведено приклади обчислень з таблицями істинності(рис. 2.4).

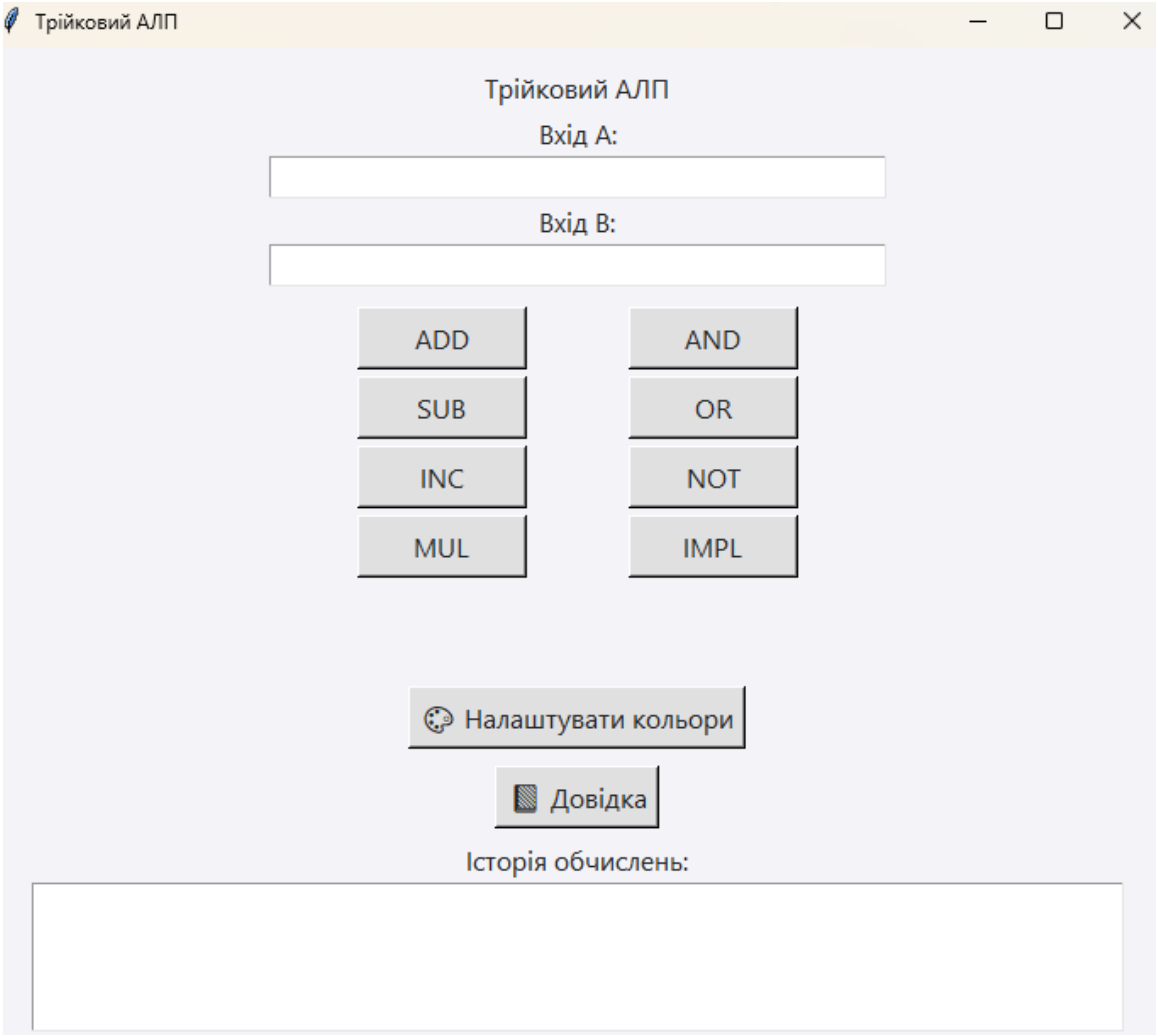



Рисунок 2.1 – Головне вікно графічного інтерфейсу програми трійкового АЛП

```
ADD: [0, 0, 1], [0, -1, 0] => [-1, 1]
INC: [0, 1, 1], None => [1, -1, -1]
AND: [1, -1, 0], [0, 0, 0] => [0, -1, 0]
IMPL: [1, -1, 0], [1, -1, 0] => [1, 1, 0]
```

Рисунок 2.2 – Вікно історії обчислень

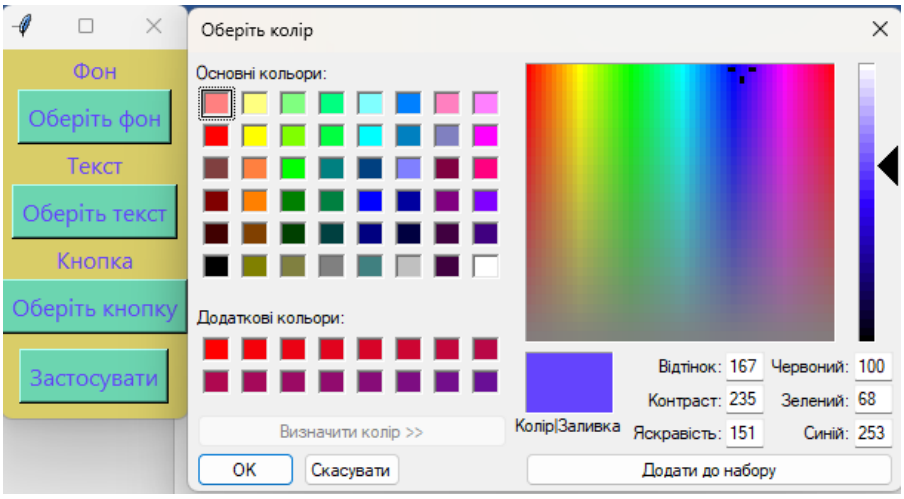



Рисунок 2.3 – Вікно налаштування теми інтерфейсу користувача

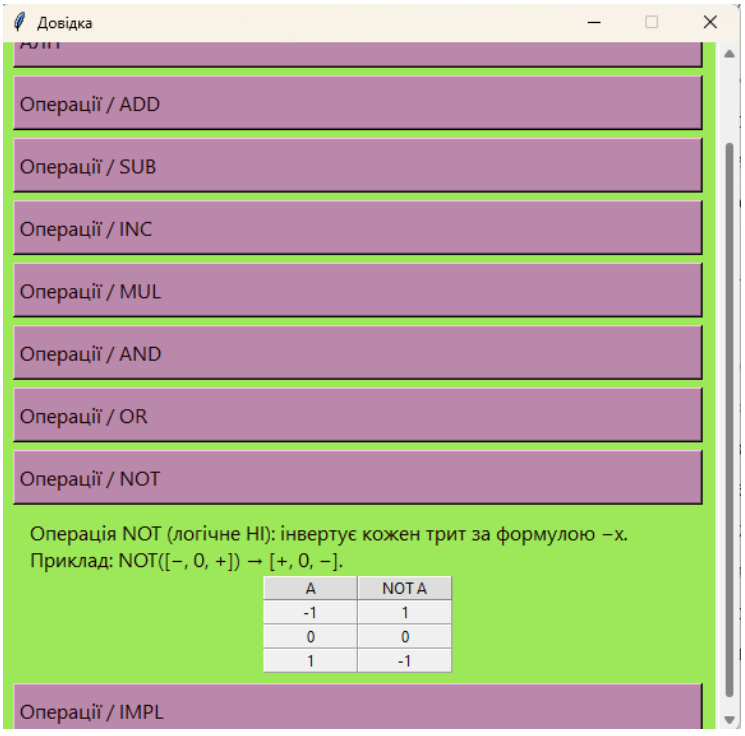



Рисунок 2.4 – Вікно довідки

Всі перераховані модулі (ядро alu.py та допоміжні history.py, theme_manager.py, help_system.py, а також головний скрипт запуску main.py) організовані у вигляді єдиного застосунку. Головна функція програми (див. лістинг А.1) ініціалізує клас TernaryALU, створює головне вікно Tkinter та зв’язує елементи GUI з викликами відповідних методів АЛП. Після цього програма переходить у цикл обробки подій інтерфейсу, очікуючи дії користувача.

Взаємодія між GUI та логікою наступна: коли користувач натискає на кнопку одної з восьми операцій, з полів вводу зчитуються рядки операндів; вони передаються до методу parse(), отримані списки тритів передаються в обраний метод (наприклад, TernaryALU.add() або TernaryALU.t_and() залежно від вибору операції), після чого результат у вигляді списку тритів через метод format() перетворюється на рядок і відображається. Одночасно запис про виконану операцію додається у модуль історії. Якщо введено некоректний символ або сталася інша помилка (наприклад, ділення на нуль, хоча в нашому наборі операцій такого немає), програма відобразить повідомлення з описом помилки (реалізовано через перехоплення винятків, що генеруються методами TernaryALU).

Графічний інтерфейс було протестовано на різних варіантах введення, у тому числі на максимально довгих числах та граничних комбінаціях. Всі функціональні можливості GUI підтвердили свою коректність: операції виконуються правильно, історія фіксується, зміна теми відбувається миттєво, довідкова інформація відображається відповідно до задуму. Таким чином, програмна частина АЛП повністю реалізована та готова до використання.

## 2.8 Апаратна модель трійкового АЛП (VHDL)

Апаратна частина розробки була реалізована за допомогою мови опису апаратури VHDL. Метою стало створення логічного модуля, здатного

виконувати трійкові арифметичні та логічні операції з дотриманням симетричної системи числення. У фокусі реалізації — компактність структури, функціональна завершеність та простота масштабування.

Основна частина коду розміщена у файлі ternary_alu.vhd (див. лістинг А.6). В модулі описано сутність (entity) з такими сигналами:

- a_in, b_in: вхідні вектори шириною 6 біт, які кодують по три трити кожен;
- op_code: 4-бітовий сигнал, який визначає обрану операцію;
- start, clk, reset: сигнали керування;
- trit_out: результат операції (6 біт);
- done: прапорець завершення обчислення.

Вся обчислювальна логіка описана в межах одного синхронного процесу, що реагує на фронт сигналу clk. Цей процес перевіряє наявність активного сигналу start та переходить до обробки відповідно до значення op_code.

Окремі частини коду розділені функціонально:

- Блок перетворення вхідних даних: за допомогою функції trits_to_int обидва вектори (a_in та b_in) перетворюються у десяткові значення на основі симетричної трійкової логіки (–1, 0, +1);
- Обчислювальний блок: реалізований за допомогою оператора case, кожен варіант якого відповідає одній із восьми операцій. Арифметичні операції виконуються над перетвореними числами, тоді як логічні — обробляють кожну пару тритів окремо;
- Формування результату: за допомогою int_to_trits результат знову кодується у вигляді тритів та подається на trit_out;
- Після завершення обчислення встановлюється сигнал done.

Завдяки використанню окремих допоміжних функцій код лишається зрозумілим, а логіка кожної операції — легко керованою. Наприклад, імплементація логічної імплікації (IMPL) базується на таблиці істинності для трійкової логіки Лукасевича, де обчислення проводиться як min(2, max(0, 3 – a + b)) для кожної пари тритів.

Особливістю апаратної моделі є повна синхронність: усі обчислення виконуються в один такт, одразу після подачі сигналу start. Для забезпечення цього в коді відсутня будь-яка станна автоматика — натомість реалізовано пряме керування через if rising_edge(clk) та умовні гілки.

Фрагмент логічної схеми модуля, синтезованої за допомогою RTL Viewer у середовищі Quartus Prime Lite, наведено на рисунку 2.5. Ця схема ілюструє структурне представлення модуля та взаємозв'язки між функціональними блоками. Повна схема наведена у додатку Б.

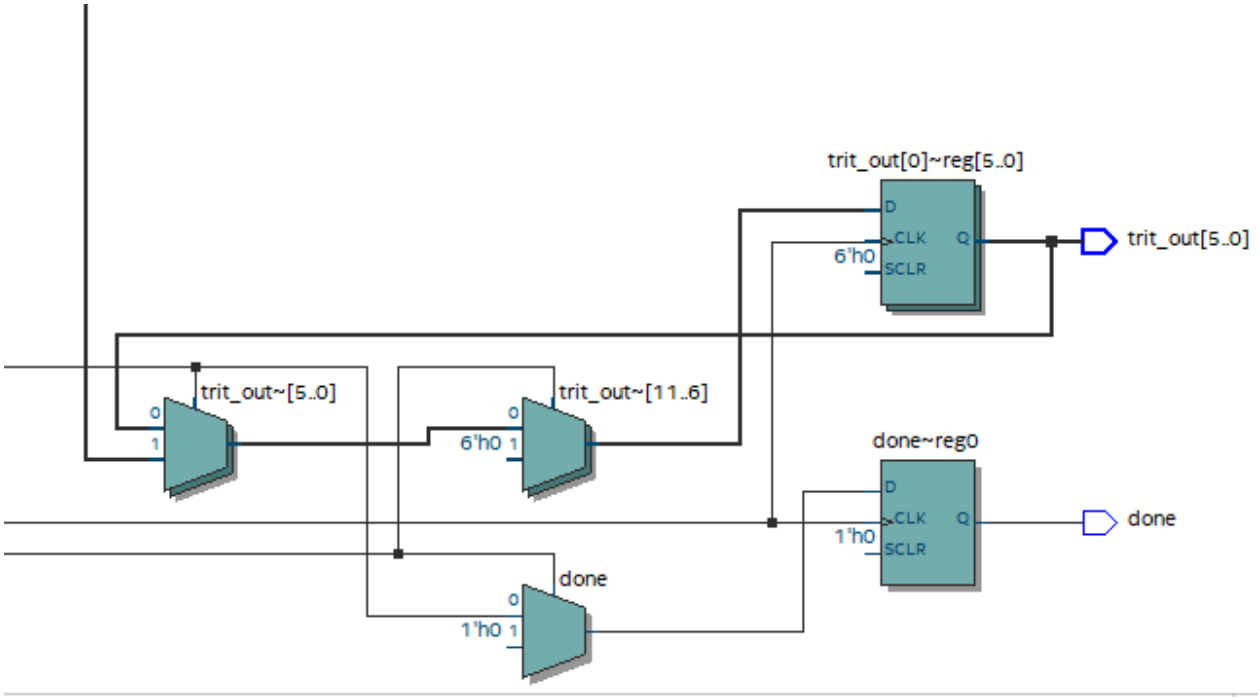



Рисунок 2.5 – Фрагмент RTL-схеми трійкового ALU, згенерований у Quartus Prime Lite

## 2.9 Детальний огляд симуляції апаратної моделі трійкового АЛП (VHDL) на прикладі операції ADD

Для демонстрації роботи апаратної моделі було наведено покрокову симуляцію кожного з семи основних сигналів окремо: a_in, b_in, op_code, start, clk, done. Для кожного з них створено окрему часову діаграму, що ілюструє вплив сигналу на обчислювальний процес. На прикладі операції додавання (ADD) продемонстровано типовий сценарій роботи:

Спочатку відбувається подача тритів на вхід a_in. Значення "01 10 11" кодує трити 0, 0, +1. Подання цих даних представлено на рисунку 2.6.

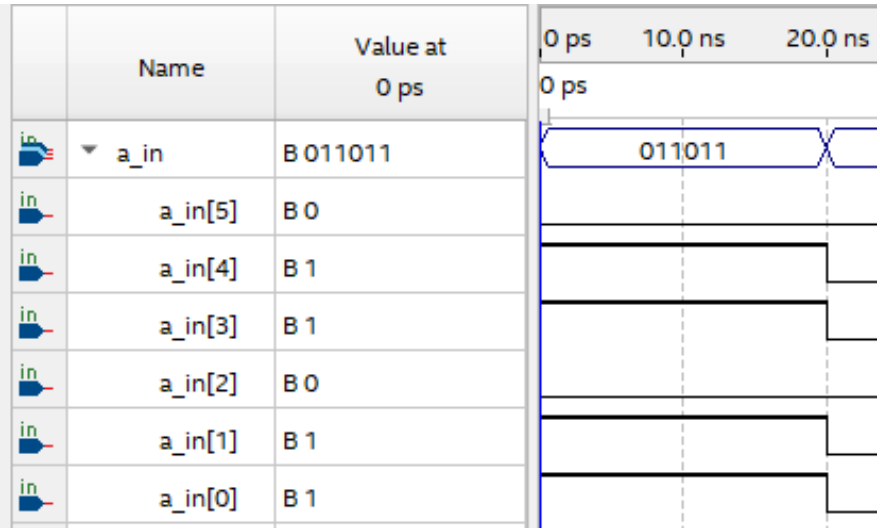



Рисунок 2.6 – Сигнал a_in: подача першого операнда

Після цього на вхід b_in подається другий операнд. У нашому випадку — "10 00 01", що відповідає тритам 0, -1, 0. Цей момент зображено на рисунку 2.7.

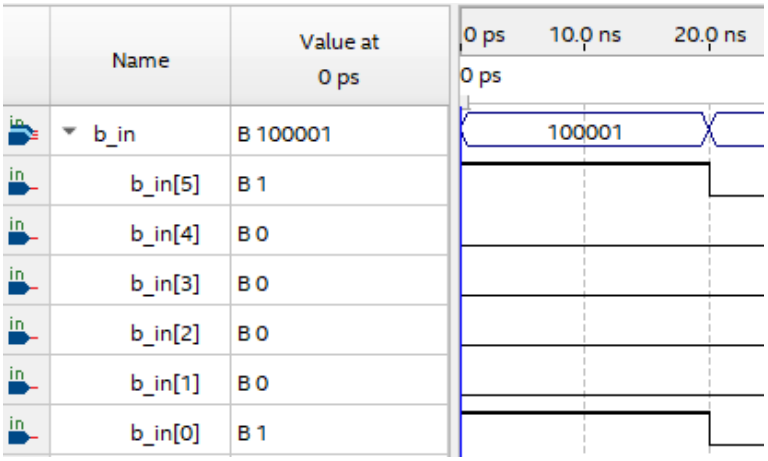



Рисунок 2.7 – Сигнал b_in: подача другого операнда

Далі активується сигнал op_code, який задає тип операції. У цьому прикладі — "0000" (додавання). Його фіксацію показано на рисунку 2.8.

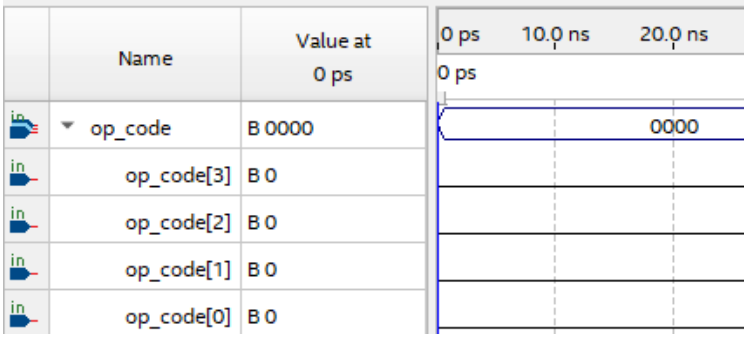



Рисунок 2.8 – Сигнал op_code: вибір операції

Після цього активується сигнал start, який ініціює обчислення. На рисунку 2.9 показано момент подачі start.

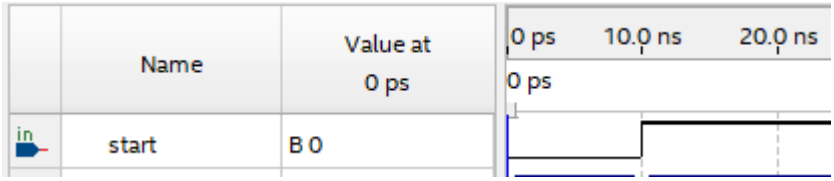



Рисунок 2.9 – Сигнал start: запуск операції

На фронт сигналу clk відбувається саме обчислення. Часову діаграму clk наведено на рисунку 2.10.

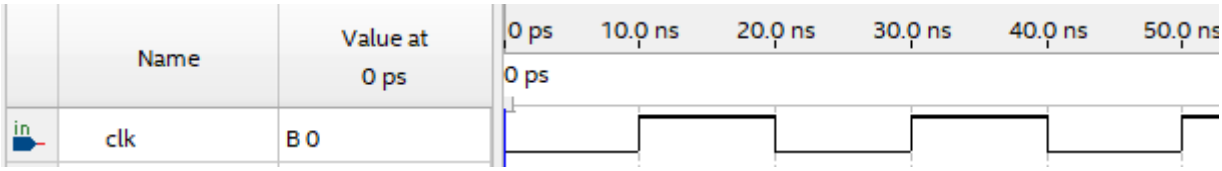



Рисунок 2.10 – Сигнал clk: такт синхронізації обчислення

Нарешті, на виході trit_out з’являється результат, а done встановлюється в '1'. Це зафіксовано на рисунку 2.11.

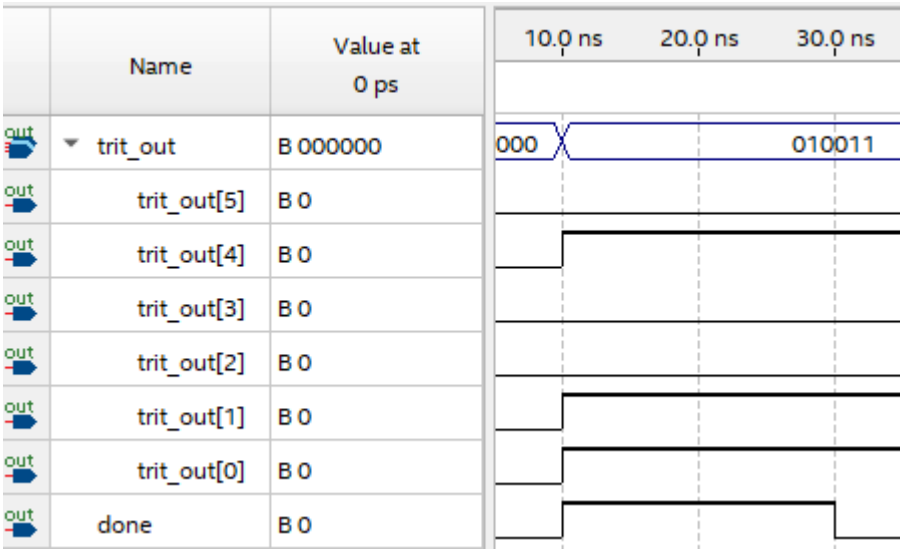



Рисунок 2.11 – Сигнали trit_out і done: результат і завершення обчислення

Цей підхід застосовувався для перевірки всіх восьми операцій.. У кожному випадку спостерігалась відповідність між очікуваними результатами та фактичними значеннями trit_out, що підтверджує коректність функціональної логіки.

## 2.10 Симуляція операцій ALU: порівняння реалізацій Python та VHDL

На рисунках 2.12–2.13 зображено приклад виконання операції ADD (додавання).

Вхідні значення:

– A = 0 0 + → 01 01 11

– B = 0 - 0→ 01 00 01

Очікуваний результат: A + B = [-1, +1] → 01 00 11


Рисунок 2.12 – Результат операції ADD у Python (GUI)

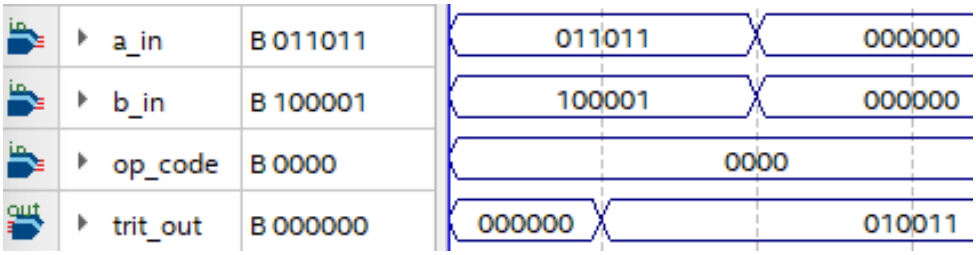



Рисунок 2.13 – Результат операції ADD у VHDL (ModelSim)

На рисунках 2.14–2.15 представлено приклад виконання операції SUB (віднімання).

Вхідні значення:

– A = + + + → 11 11 11

– B = 0 + 0 → 01 11 01

Очікуваний результат: A - B = [+1, 0, +1] → 11 01 11

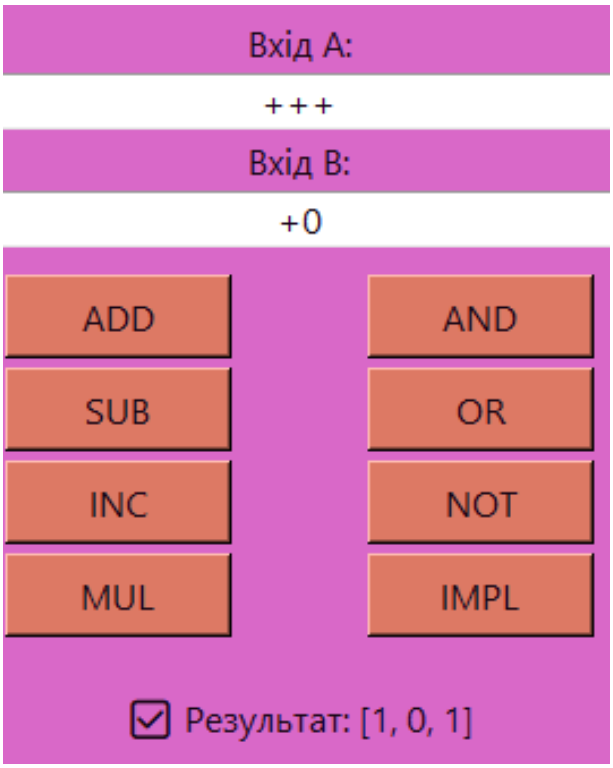



Рисунок 2.14 – Результат операції SUB у Python (GUI)

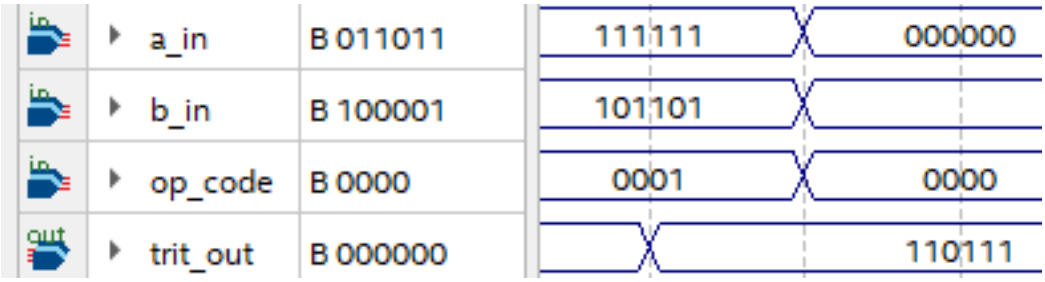



Рисунок 2.15 – Результат операції SUB у VHDL (ModelSim)

Рисунки 2.16–2.17 демонструють операцію INC –збільшення операнди A на 1.

Вхідні значення: A = 0 + + → 01 11 11

Очікуваний результат: INC(A) = [+1, -1, -1] → 11 00 00

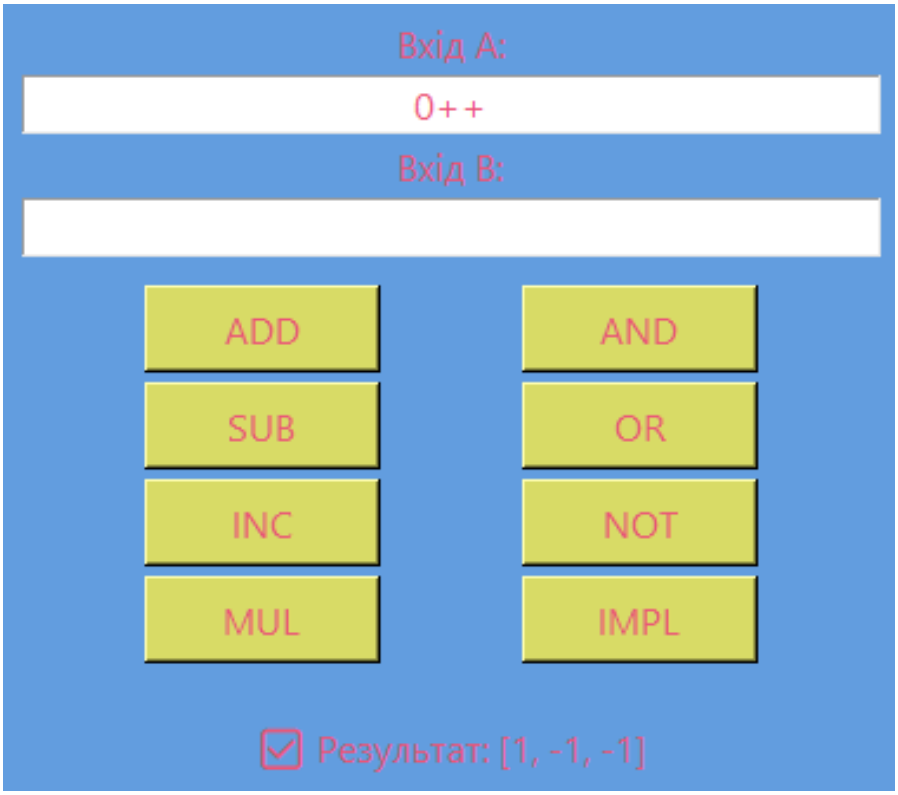



Рисунок 2.16 – Результат операції INC у Python (GUI)

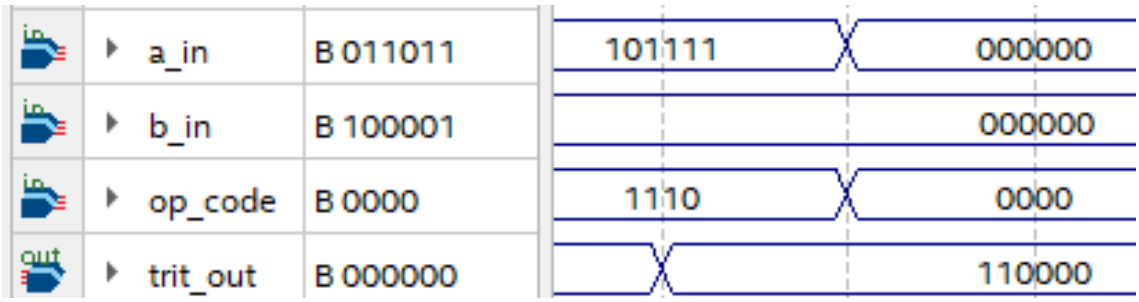



Рисунок 2.17 – Результат операції INC у VHDL (ModelSim)

Рисунки 2.18–2.19 ілюструють приклад множення MUL.

Вхідні значення:

– A = 0 - - → 01 00 00

– B = 0 + - → 01 11 00

Очікуваний результат: A * B = [-1, 0, +1] → 00 01 11

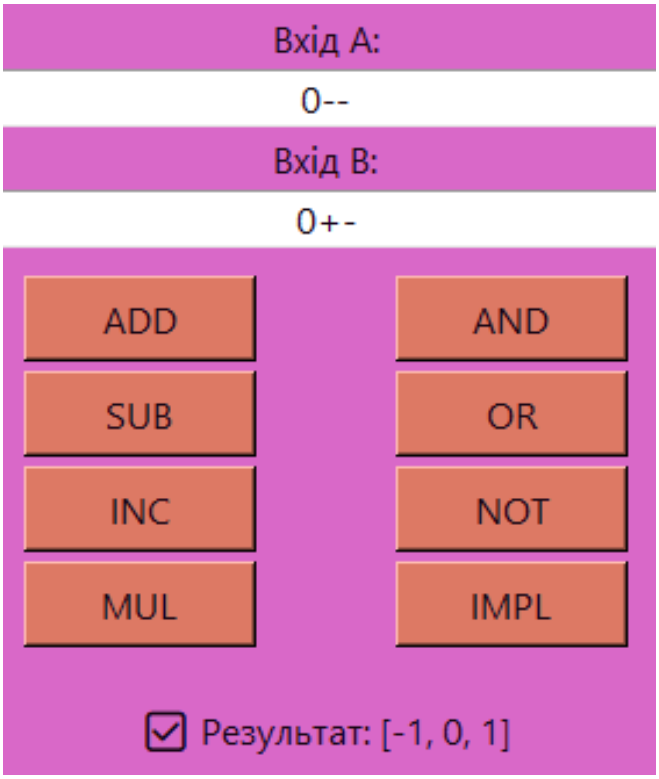



Рисунок 2.18 – Результат операції MUL у Python (GUI)

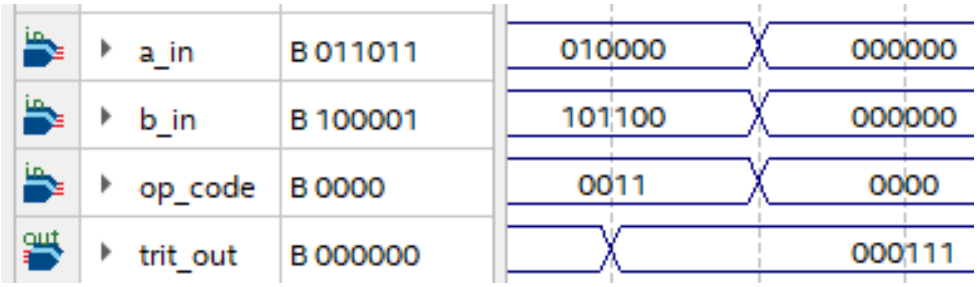



Рисунок 2.19 – Результат операції MUL у VHDL (ModelSim)

Рисунки 2.20–2.21 ілюструють приклад AND.

Вхідні значення:

– A = + - 0 → 11 00 01

– B = 0 0 0 → 01 01 01

Очікуваний результат: AND(A,B) = min(A,B) = [0, -1, 0] → 01 00 01

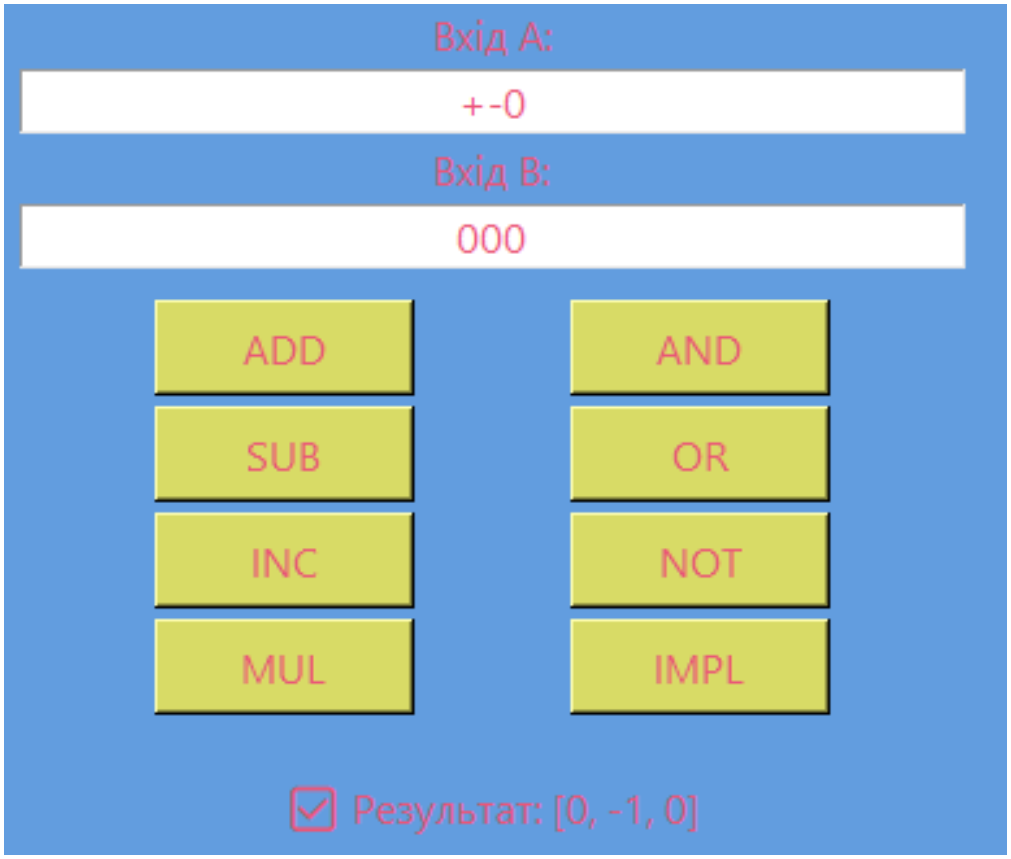



Рисунок 2.20 – Результат AND у Python (GUI)

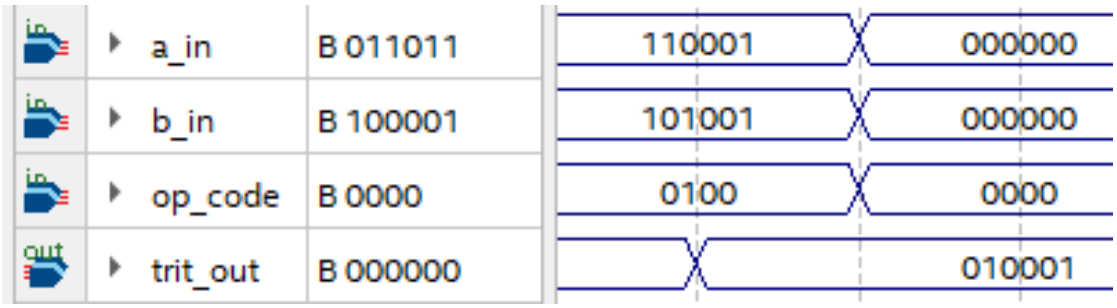



Рисунок 2.21 – Результат AND у VHDL

Рисунки 2.22–2.23 ілюструють приклад OR.

Вхідні значення:

– A = 0 0 + → 01 01 11

– B = 0 – 0 → 01 00 01

Очікуваний результат: OR(A,B) = max(A,B) = [0, 0, +1] → 01 01 11

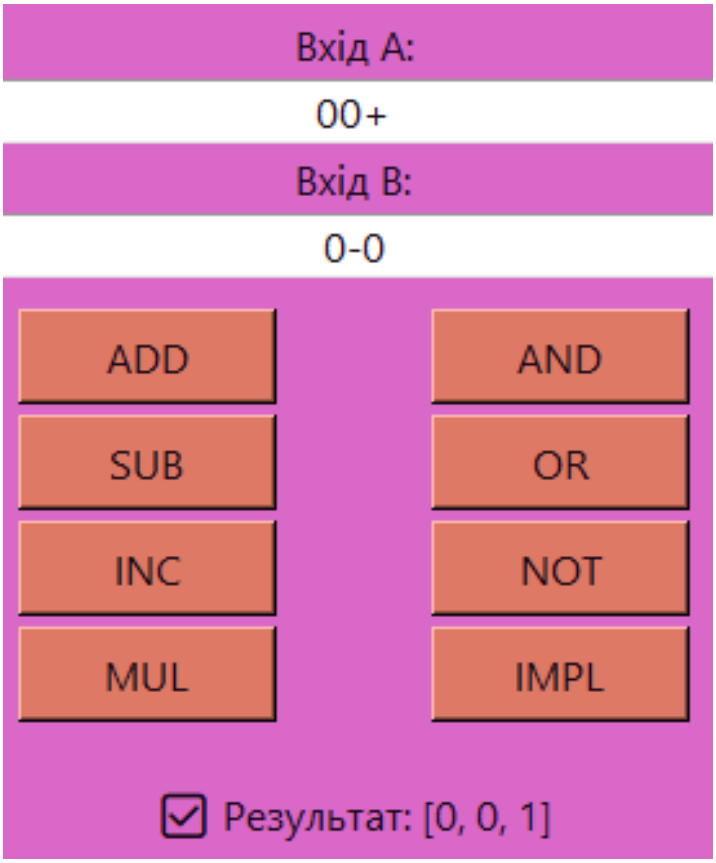



Рисунок 2.22 – Результат OR у Python (GUI)

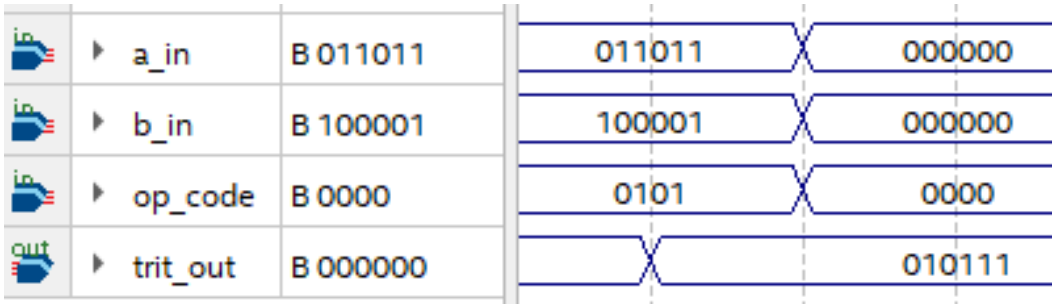



Рисунок 2.23 – Результат OR у VHDL

Рисунки 2.24–2.25 ілюструють приклад IMPL.

Вхідні значення:

– A = + 0 - → 11 01 00

– B = + 0 - → 11 01 00

Очікуваний результат: IMPL(A,B) = [+1, 0, +1] → 11 01 11

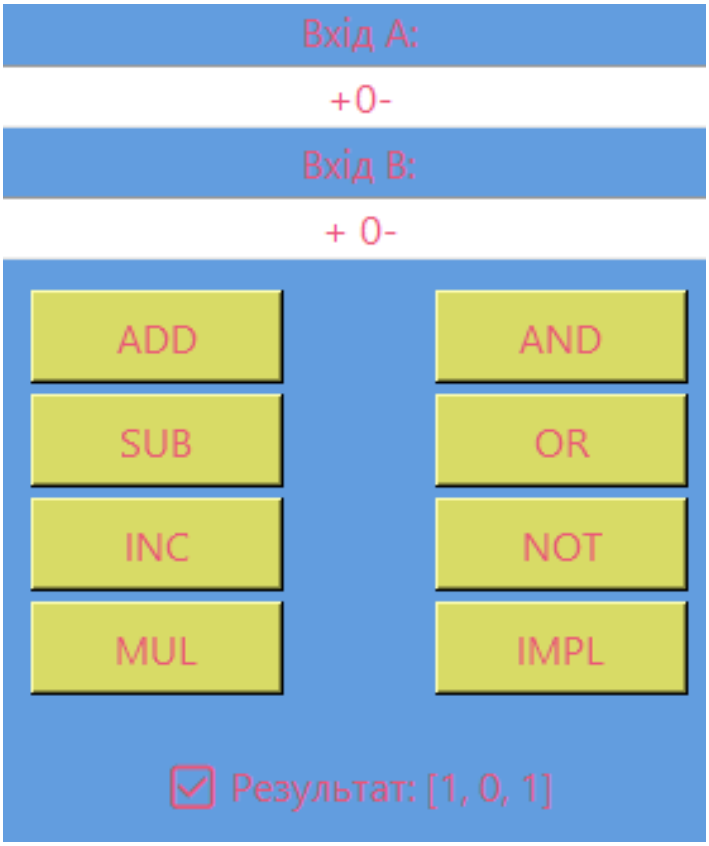



Рисунок 2.24 – Результат IMPL у Python (GUI)

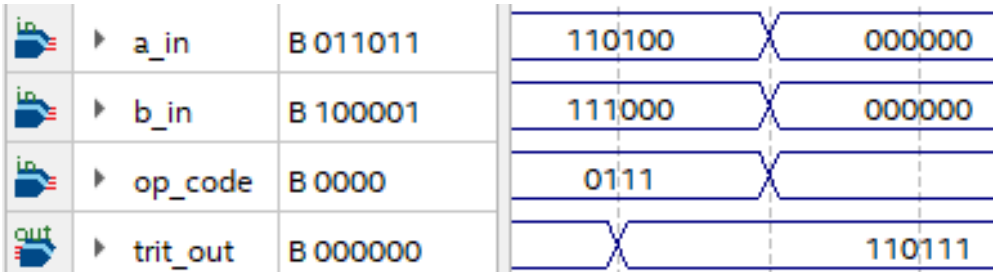



Рисунок 2.25 – Результат IMPL у VHDL

Рисунки 2.26–2.27 ілюструють приклад NOT.

Вхідні значення: A = + - 0 → 11 00 01

Очікуваний результат: NOT(A) = [-x for x in A] = [-1, +1, 0] → 00 11 01

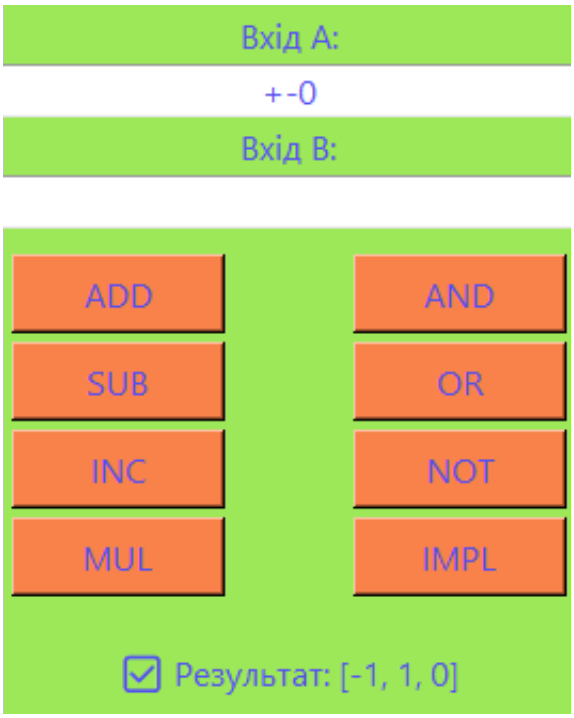



Рисунок 2.26 – Результат NOT у Python (GUI)

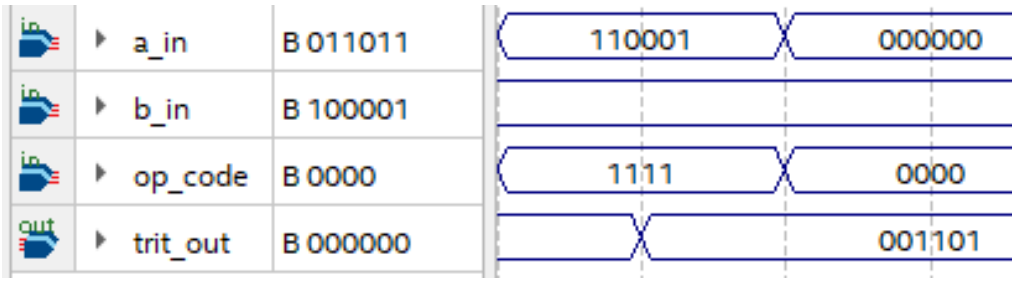



Рисунок 2.27 – Результат NOT у VHDL

Як видно з наведених прикладів, програмна реалізація та апаратна симуляція виводять ідентичні результати для всіх операцій. Це підтверджує коректність розробленого ALU, а також повну відповідність реалізації симетричній трійковій логіці.

# ВИСНОВКИ

У межах кваліфікаційної роботи було реалізовано симуляційну модель трійкового арифметико-логічного пристрою (АЛП), що функціонує за принципами симетричної трійкової логіки з множини значень {–1, 0, +1}. В результаті досягнуто основну мету дослідження — створення функціонального АЛП з підтримкою восьми базових операцій, серед яких: додавання, віднімання, інкремент, множення, логічні AND, OR, NOT та імплікація.

Для реалізації було використано мову програмування Python з графічною оболонкою на основі Tkinter. Розроблено модулі для виконання арифметичних і логічних операцій, модуль історії операцій, система довідки та підтримка зміни теми інтерфейсу. Окремим блоком реалізовано трійкову арифметику, адаптовану до симетричної логіки, що забезпечує коректну обробку чисел з базою 3.

У середовищі Quartus Prime Lite реалізовано симуляційну модель АЛП на VHDL. Уся логіка вводу тритів, прийому коду операції та формування результату організована через синхронний процес, що реагує на сигнали clk, start, reset та op_code. За допомогою Waveform було здійснено тестування працездатності моделі, що підтвердило правильність реалізації логіки та обчислень.

Практична цінність роботи полягає у демонстрації можливості побудови трійкових обчислювальних систем, які можуть виступати як альтернативні моделі цифрової логіки. Такий підхід може знайти застосування в енергоефективних і новітніх обчислювальних архітектурах, зокрема в галузях штучного інтелекту, нанотехнологій та багатозначних логічних систем.

У майбутньому проєкт може бути доповнений розширенням підтримуваних операцій, реалізацією модуля переведення між двійковою та трійковою системами числення.

# ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Łukasiewicz, J. On Three-Valued Logic. Polish Academy of Sciences, 1920.
2. Попов, В. І. Основи цифрової електроніки: навчальний посібник. — Київ: Ліра-К, 2018. — 368 с.
3. Волосюк, В. К. Теоретичні основи обчислювальної техніки. — Київ: Видавничий дім «Слово», 2020. — 284 с.
4. Brown, S., Vranesic, Z. Fundamentals of Digital Logic with VHDL Design. — New York: McGraw-Hill, 2009. — 816 p.
5. Kasperkovitz, P., Loerke, H. Ternary Computers: A New Dimension of Computing // ACM SIGARCH Computer Architecture News. — 2015. — Vol. 43, №2. — P. 44–49.
6. Hurst, S. L. Multiple-Valued Logic in VLSI: Challenges and Opportunities // IEEE Journal of Solid-State Circuits. — 1999. — Vol. 34, №10. — P. 1378–1390.
7. Zhao, Y., Hasan, M. M., Bayat-Sarmadi, S., et al. *Efficient Ternary Logic Circuits Optimized by Ternary Arithmetic Algorithms* // IEEE Transactions on Emerging Topics in Computing. — 2024. — Vol. 12, No. 3. — P. 410–421. — DOI: 10.1109/TETC.2024.3390261.
8. Python Software Foundation. Tkinter – Python interface to Tcl/Tk [Електронний ресурс]. — Режим доступу: https://docs.python.org/3/library/tkinter.html (дата звернення: 07.06.2025).
9. Щербина, О. А. Методичні вказівкидо використання мови VHDL для опису і моделювання цифрових електронних схем // Київ: КНУБА, 1999. — 35 с.
10. Якубовський, В. М. Трійкова логіка та її застосування в обчислювальних пристроях // Вісник ХНУРЕ. — 2021. — №4. — С. 88–93.

# ДОДАТОК А

## Лістинг програми

```
import tkinter as tk
from alu import TernaryALU
from help_system import open_help
import history
from theme_manager import user_theme, choose_color,
apply_custom_theme
def parse_input(input_str):
    symbols = {'-': -1, '0': 0, '+': 1}
    try:
        trits = [symbols[ch] for ch in input_str.strip() if ch
in symbols]
        return trits if trits else None
    except KeyError:
        return None

alu = TernaryALU()
def calculate(op):
    a = parse_input(entry_a.get())
    b = parse_input(entry_b.get())

    if a is None or (op not in [“INC”, “NOT”] and b is None):
        result_var.set(“❌ Невірне введення! Вводьте лише
символи -, 0 або +.”)
        history_text.configure(bg=“#ffdddd”)
        return
    else:
        history_text.configure(bg=user_theme[“bg”])
    try:
        if op == “ADD”:
            result = alu.add(a, b)
        elif op == “SUB”:
            result = alu.sub(a, b)
        elif op == “INC”:
            result = alu.inc(a)
        elif op == “MUL”:
            result = alu.mul(a, b)
        elif op == “AND”:
            result = alu.t_and(a, b)
        elif op == “OR”:
            result = alu.t_or(a, b)
        elif op == “NOT”:
            result = alu.t_not(a)
        elif op == “IMPL”:
            result = alu.t_impl(a, b)
```

Лістинг А.1 — Файл main.py

```
 else:
            result = [“?”]
    except Exception as e:
        result_var.set(f”❌ Помилка: {e}”)
        history_text.configure(bg=“#ffdddd”)
        return

    history.add(op, a, b, result)
    result_var.set(“☑ Результат: “ + str(result))
    update_history_display()

def update_history_display():
    history_text.configure(state=“normal”)
    history_text.delete(“1.0”, tk.END)
    for op, a, b, result in history.get():
        line = f”{op}: {a}, {b} => {result}\n”
        history_text.insert(tk.END, line)
    history_text.configure(state=“disabled”)

def open_color_settings():
    settings_window = tk.Toplevel(root)
    settings_window.title(“Налаштування кольорів”)

    widgets = [settings_window]

    label_bg = tk.Label(settings_window, text=“Фон”)
    label_bg.pack()
    bg_btn = tk.Button(settings_window, text=“Оберіть фон”,
command=lambda: choose_color(“bg”))
    bg_btn.pack()
    widgets.extend([label_bg, bg_btn])

    label_fg = tk.Label(settings_window, text=“Текст”)
    label_fg.pack()
    fg_btn = tk.Button(settings_window, text=“Оберіть текст”,
command=lambda: choose_color(“fg”))
    fg_btn.pack()
    widgets.extend([label_fg, fg_btn])

    label_btn = tk.Label(settings_window, text=“Кнопка”)
    label_btn.pack()
    btn_btn = tk.Button(settings_window, text=“Оберіть кнопку”,
command=lambda: choose_color(“button_bg”))
    btn_btn.pack()
    widgets.extend([label_btn, btn_btn])
    apply_btn = tk.Button(settings_window, text=“Застосувати”,
command=lambda: apply_custom_theme(root, all_widgets +
widgets))
```

Лістинг А.1, аркуш 2

```
apply_btn.pack(pady=10)
    widgets.append(apply_btn)

    apply_custom_theme(settings_window, widgets)

root = tk.Tk()
root.title("Трійковий АЛП")
root.geometry("700x500")

result_var = tk.StringVar()

label_title = tk.Label(root, text="Трійковий АЛП",
font=user_theme["title_font"])
label_title.pack(pady=(10, 0))

label_a = tk.Label(root, text="Вхід A:")
label_a.pack()
entry_a = tk.Entry(root, justify="center", width=40)
entry_a.pack()

label_b = tk.Label(root, text="Вхід B:")
label_b.pack()
entry_b = tk.Entry(root, justify="center", width=40)
entry_b.pack()

frame_ops = tk.Frame(root)
frame_ops.pack(pady=10)

arith_frame = tk.Frame(frame_ops)
logic_frame = tk.Frame(frame_ops)
arith_frame.pack(side="left", padx=30)
logic_frame.pack(side="left", padx=30)

arith_ops = ["ADD", "SUB", "INC", "MUL"]
logic_ops = ["AND", "OR", "NOT", "IMPL"]

buttons = []

for op in arith_ops:
    btn = tk.Button(arith_frame, text=op, width=10,
command=lambda o=op: calculate(o))
    btn.pack(pady=2)
    buttons.append(btn)
for op in logic_ops:
    btn = tk.Button(logic_frame, text=op, width=10,
command=lambda o=op: calculate(o))
    btn.pack(pady=2)
    buttons.append(btn)
```

Лістинг А.1, аркуш 3

```
label_result = tk.Label(root, textvariable=result_var,
font=("Arial", 12))
label_result.pack(pady=10)

btn_theme = tk.Button(root, text="🎨 Налаштувати кольори",
command=open_color_settings)
btn_theme.pack(pady=5)

btn_help = tk.Button(root, text="📘 Довідка", command=lambda:
open_help(root))
btn_help.pack(pady=5)

label_history = tk.Label(root, text="Історія обчислень:")
label_history.pack()
history_text = tk.Text(root, height=6, width=80)
history_text.pack(pady=(0, 10))
history_text.configure(state="disabled")

all_widgets = [
    label_title, label_a, entry_a, label_b, entry_b,
    frame_ops, arith_frame, logic_frame,
    label_result, btn_theme, btn_help,
    label_history, history_text
] + buttons

apply_custom_theme(root, all_widgets)
root.mainloop()
```

Лістинг А.1, аркуш 4

```
class TernaryALU:
    def __init__(self):
        self.base = 3  # симетрична трійкова система

    def parse(self, s):
        """Рядок типу '+-0' у список [-1, 0, 1]"""
        return [self.char_to_trit(c) for c in s]

    def format(self, trits):
        """Список [-1, 0, 1] у рядок '-0+'"""
        return ''.join(self.trit_to_char(t) for t in trits)
    def char_to_trit(self, c):
        if c == '-':
            return -1
```

Лістинг А.2 — Файл alu.py

```
elif c == '0':
            return 0
        elif c == '+':
            return 1
        else:
            raise ValueError(f"Недопустимий символ трита: {c}")

    def trit_to_char(self, t):
        if t == -1:
            return '-'
        elif t == 0:
            return '0'
        elif t == 1:
            return '+'
        else:
            raise ValueError(f"Недопустиме значення трита:
{t}")

    def _normalize(self, a, b):
        max_len = max(len(a), len(b))
        return (
            [0] * (max_len - len(a)) + a,
            [0] * (max_len - len(b)) + b,
        )
    def add(self, a, b):
        dec_a = self.ternary_to_decimal(a)
        dec_b = self.ternary_to_decimal(b)
        return self.decimal_to_ternary(dec_a + dec_b)

    def sub(self, a, b):
        dec_a = self.ternary_to_decimal(a)
        dec_b = self.ternary_to_decimal(b)
        return self.decimal_to_ternary(dec_a - dec_b)

    def inc(self, a):
        return
self.decimal_to_ternary(self.ternary_to_decimal(a) + 1)

    def mul(self, a, b):
        dec_a = self.ternary_to_decimal(a)
        dec_b = self.ternary_to_decimal(b)
        return self.decimal_to_ternary(dec_a * dec_b)

    def t_not(self, a):
        return [-t for t in a]

    def t_and(self, a, b):
        a, b = self._normalize(a, b)
        return [min(x, y) for x, y in zip(a, b)]
```

Лістинг А.2, аркуш 2

```
    def t_or(self, a, b):
        a, b = self._normalize(a, b)
        return [max(x, y) for x, y in zip(a, b)]

    def t_impl(self, a, b):
        a, b = self._normalize(a, b)
        return [max(-x, y) for x, y in zip(a, b)]

    def ternary_to_decimal(self, trits):
        return sum(val * (3 ** idx) for idx, val in
enumerate(reversed(trits)))

    def decimal_to_ternary(self, n):
        if n == 0:
            return [0]
        trits = []
        while n != 0:
            n, r = divmod(n, 3)
            if r == 2:
                r = -1
                n += 1
            trits.append(r)
        return trits[::-1]
```

Лістинг А.2, аркуш 3

```
_history = []

def add(op, a, b, result):
    _history.append((op, a, b, result))

def get():
    return _history
```

Лістинг А.3 — Файл history.py

```
import tkinter as tk
from tkinter import colorchooser
user_theme = {
    “bg”: “#f4f4f8”,
    “fg”: “#202124”,
```

Лістинг А.4 — Файл theme_manager.py

```
"entry_bg": "#ffffff",
    "button_bg": "#e0e0e0",
    "font": ("Segoe UI", 12),
    "title_font": ("Segoe UI", 16, "bold")
}

def choose_color(setting_key):
    color = colorchooser.askcolor(title="Оберіть колір")[1]
    if color:
        user_theme[setting_key] = color

def apply_custom_theme(root, widgets):
    root.config(bg=user_theme["bg"])
    for w in widgets:
        if isinstance(w, (tk.Tk, tk.Toplevel, tk.Frame,
tk.Canvas)):
            w.config(bg=user_theme["bg"])
        elif isinstance(w, tk.Label):
            w.config(bg=user_theme["bg"], fg=user_theme["fg"],
font=user_theme["font"])
        elif isinstance(w, tk.Entry):
            w.config(bg=user_theme["entry_bg"],
fg=user_theme["fg"],
                     insertbackground=user_theme["fg"],
font=user_theme["font"])
        elif isinstance(w, tk.Button):
            w.config(bg=user_theme["button_bg"],
fg=user_theme["fg"], font=user_theme["font"])
        elif isinstance(w, tk.Text):
            w.config(bg=user_theme["entry_bg"],
fg=user_theme["fg"], font=("Courier New", 10))
        elif isinstance(w, tk.Listbox):
            w.config(bg=user_theme["entry_bg"],
fg=user_theme["fg"], font=user_theme["font"])
```

Лістинг А.4, аркуш 2

```
import tkinter as tk
from theme_manager import apply_custom_theme
help_texts = {
    "Трійкова логіка": "Симетрична трійкова логіка використовує
значення: -1 (−), 0 та +1 (+).\n"
                       "Це дозволяє відображати від'ємні,
нульові й позитивні логічні стани.\n"
                       "Кожен трит зберігає більше інформації,
ніж біт: log₂(3) ≈ 1.58,\n"
```

Лістинг А.5 — Файл help_system.py

```
 “що забезпечує вищу щільність інформації та
енергоефективність.”,

    “АЛП”: “Трійковий АЛП (арифметико-логічний пристрій)
виконує обчислення\n”
           “у симетричній трійковій системі числення. Підтримує
операції:\n”
           “ADD, SUB, INC, MUL, AND, OR, NOT, IMPL — над
наборами тритів із значеннями –, 0, +.”
}

descriptions = {
    “ADD”: “Операція додавання (ADD): виконує складання двох
трійкових чисел у форматі [–, 0, +].\n”
           “Результат може містити трити з перенесенням у вищі
розряди.\n”
           “Приклад: [+, –] + [0, +] = [+, 0].”,

    “SUB”: “Операція віднімання (SUB): обчислює A – B через
десяткове представлення з подальшим\n”
           “переведенням назад у трійкову систему зі значеннями
–1, 0, +1.\n”
           “Приклад: [-, -, +] – [–, –] = [-, +, -].”,

    “INC”: “Операція інкременту (INC): додає один трит +1 до
числа.\n”
           “Приклад: INC([–, 0]) = [–, +].”,

    “MUL”: “Операція множення (MUL): перемножує два трійкові
числа через десяткові значення\n”
           “та переводить результат у симетричний трійковий
формат.\n”
           “Приклад: [+, –] × [0, +] = [ +, -].”,

    “AND”: “Операція AND (логічне І): повертає мінімальне
значення для кожної пари тритів.\n”
           “Приклад: [–, 0, +] AND [+ ,–, 0] → [–, –, 0].”,

    “OR”: “Операція OR (логічне АБО): повертає максимальне
значення з кожної пари тритів.\n”
          “Приклад: [–, 0, +] OR [+ ,–, 0] → [+ , 0, +].”,

    “NOT”: “Операція NOT (логічне НІ): інвертує кожен трит за
формулою –x.\n”
           “Приклад: NOT([–, 0, +]) → [+, 0, –].”,

    “IMPL”: “Операція IMPL (логічна імплікація): виконується за
правилом:\n”
```

Лістинг А.5, аркуш 2

```
 "якщо A ≤ B → результат + (істина), інакше – або 0.\n"
            "Приклад: [+, 0, –] → [–, 0, +] = [–, +, +].",
}

def create_truth_table(parent, headers, rows):
    table_frame = tk.Frame(parent)
    for col, header in enumerate(headers):
        tk.Label(table_frame, text=header, borderwidth=1,
relief="ridge", width=10, bg="#ddd").grid(row=0, column=col,
sticky="nsew")
    for row_num, row_data in enumerate(rows, start=1):
        for col, val in enumerate(row_data):
            tk.Label(table_frame, text=val, borderwidth=1,
relief="ridge", width=10).grid(row=row_num, column=col,
sticky="nsew")
    return table_frame

def open_help(root):
    help_window = tk.Toplevel(root)
    help_window.title("Довідка")
    help_window.geometry("580x540")
    help_window.resizable(False, False)

    canvas = tk.Canvas(help_window, bg=root.cget("bg"))
    scrollbar = tk.Scrollbar(help_window, orient="vertical",
command=canvas.yview)
    canvas.configure(yscrollcommand=scrollbar.set)

    scrollbar.pack(side="right", fill="y")
    canvas.pack(side="left", fill="both", expand=True)

    scrollable_frame = tk.Frame(canvas, bg=root.cget("bg"))
    canvas.create_window((0, 0), window=scrollable_frame,
anchor="nw", width=560)

    scrollable_frame.bind(
        "<Configure>",
        lambda e:
canvas.configure(scrollregion=canvas.bbox("all"))
    )

    widgets = [scrollable_frame, help_window, canvas]

    def add_text_section(title, content):
        btn = tk.Button(scrollable_frame, text=title,
anchor="w")
        btn.pack(fill="x", padx=10, pady=3, ipady=3)
        frame = tk.Frame(scrollable_frame)
```

Лістинг А.5, аркуш 3

```
 lbl = tk.Label(frame, text=content, justify="left",
wraplength=520, anchor="w")
        lbl.pack(anchor="w")
        widgets.extend([btn, frame, lbl])

        def toggle():
            if frame.winfo_ismapped():
                frame.pack_forget()
            else:
                frame.pack(after=btn, fill="x", padx=20,
pady=5)

        btn.configure(command=toggle)

    def add_table_section(title, headers, rows,
description=""):
        btn = tk.Button(scrollable_frame, text=title,
anchor="w")
        btn.pack(fill="x", padx=10, pady=3, ipady=3)
        frame = tk.Frame(scrollable_frame)

        if description:
            desc = tk.Label(frame, text=description,
justify="left", wraplength=520, anchor="w")
            desc.pack(anchor="w")
            widgets.append(desc)

        table = create_truth_table(frame, headers, rows)
        table.pack()
        widgets.extend([btn, frame, table])

        def toggle():
            if frame.winfo_ismapped():
                frame.pack_forget()
            else:
                frame.pack(after=btn, fill="x", padx=20,
pady=5)

        btn.configure(command=toggle)

    for title, content in help_texts.items():
        add_text_section(title, content)

    add_text_section("Операції / ADD", descriptions["ADD"])
    add_text_section("Операції / SUB", descriptions["SUB"])
    add_text_section("Операції / INC", descriptions["INC"])
    add_text_section("Операції / MUL", descriptions["MUL"])
```

Лістинг А.5, аркуш 4

```
    # Можливі значення тритів у симетричній системі
    trits = [-1, 0, 1]

    and_rows = [[a, b, min(a, b)] for a in trits for b in
trits]
    add_table_section("Операції / AND", ["A", "B", "A AND B"],
and_rows, descriptions["AND"])

    or_rows = [[a, b, max(a, b)] for a in trits for b in trits]
    add_table_section("Операції / OR", ["A", "B", "A OR B"],
or_rows, descriptions["OR"])

    not_rows = [[a, -a] for a in trits]
    add_table_section("Операції / NOT", ["A", "NOT A"],
not_rows, descriptions["NOT"])

    def impl(a, b):
        return max(-a, b)

    impl_rows = [[a, b, impl(a, b)] for a in trits for b in
trits]
    add_table_section("Операції / IMPL", ["A", "B", "IMPL"],
impl_rows, descriptions["IMPL"])

    apply_custom_theme(help_window, widgets)
```

Лістинг А.5, аркуш 5

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ternary_alu is
    Port (
        clk      : in  std_logic;
        reset    : in  std_logic;
        a_in     : in  std_logic_vector(5 downto 0);
        b_in     : in  std_logic_vector(5 downto 0);
        op_code  : in  std_logic_vector(3 downto 0);
        start    : in  std_logic;
        trit_out : out std_logic_vector(5 downto 0);
        done     : out std_logic
    );
end ternary_alu;
```

Лістинг А.6 — Файл ternary_alu.vhd

```
architecture Behavioral of ternary_alu is
    type integer_vector is array (natural range <>) of integer;
    type integer_array is array (0 to 2) of integer;

    function trit_to_int(t: std_logic_vector(1 downto 0))
return integer is
    begin
        if t = “00” then return -1;
        elsif t = “11” then return 1;
        else return 0;
        end if;
    end;

    function int_to_trit(i: integer) return std_logic_vector is
    begin
        if i = -1 then return “00”;
        elsif i = 0 then return “01”;
        else return “11”;
        end if;
    end;

    function trits_to_int(v: std_logic_vector(5 downto 0))
return integer is
        variable digits: integer_array;
        variable result: integer := 0;
    begin
        digits(2) := trit_to_int(v(5 downto 4));
        digits(1) := trit_to_int(v(3 downto 2));
        digits(0) := trit_to_int(v(1 downto 0));
        for i in 0 to 2 loop
            result := result + digits(i) * (3 ** i);
        end loop;
        return result;
    end;
    function int_to_trits(n: integer) return std_logic_vector
is
        type trit_table_t is array(-13 to 13) of
std_logic_vector(5 downto 0);
        constant table: trit_table_t := (
              -13 => “000000”,  -- -1 -1 -1
                     -12 => “000001”,  -- -1 -1  0
                     -11 => “000011”,  -- -1 -1 +1
                     -10 => “000100”,  -- -1  0 -1
                     -9  => “000101”,  -- -1  0  0
                     -8  => “000111”,  -- -1  0 +1
                     -7  => “001100”,  -- -1 +1 -1
                     -6  => “001101”,  -- -1 +1  0
                     -5  => “001111”,  -- -1 +1 +1
```

Лістинг А.6, аркуш 2

```
                    -4  => "010000",  --  0 -1 -1
                    -3  => "010001",  --  0 -1  0
                    -2  => "010011",  --  0 -1 +1
                    -1  => "010100",  --  0  0 -1
                       0  => "010101",  --  0  0  0
                       1  => "010111",  --  0  0 +1
                       2  => "011000",  --  0 +1 -1
                       3  => "011101",  --  0 +1  0
                       4  => "011111",  --  0 +1 +1
                       5  => "110000",  -- +1 -1 -1
                       6  => "110001",  -- +1 -1  0
                       7  => "110011",  -- +1 -1 +1
                       8  => "110100",  -- +1  0 -1
                       9  => "110101",  -- +1  0  0
                    10  => "110111",  -- +1  0 +1
                    11  => "111100",  -- +1 +1 -1
                    12  => "111101",  -- +1 +1  0
                    13  => "111111"   -- +1 +1 +1
        );
    begin
        if n >= -13 and n <= 13 then
            return table(n);
        else
            return (others => '0');  -- захист на випадок
виходу за межі
        end if;
    end;

    function t_min(a, b: integer) return integer is
    begin
        if a < b then return a; else return b; end if;
    end;

    function t_max(a, b: integer) return integer is
    begin
        if a > b then return a; else return b; end if;
    end;

    function t_not(x: integer) return integer is
    begin
        return -x;
    end;
    function t_and(a, b: integer) return integer is
    begin
        return t_min(a, b);
    end;
    function t_or(a, b: integer) return integer is
    begin
```

Лістинг А.6, аркуш 3

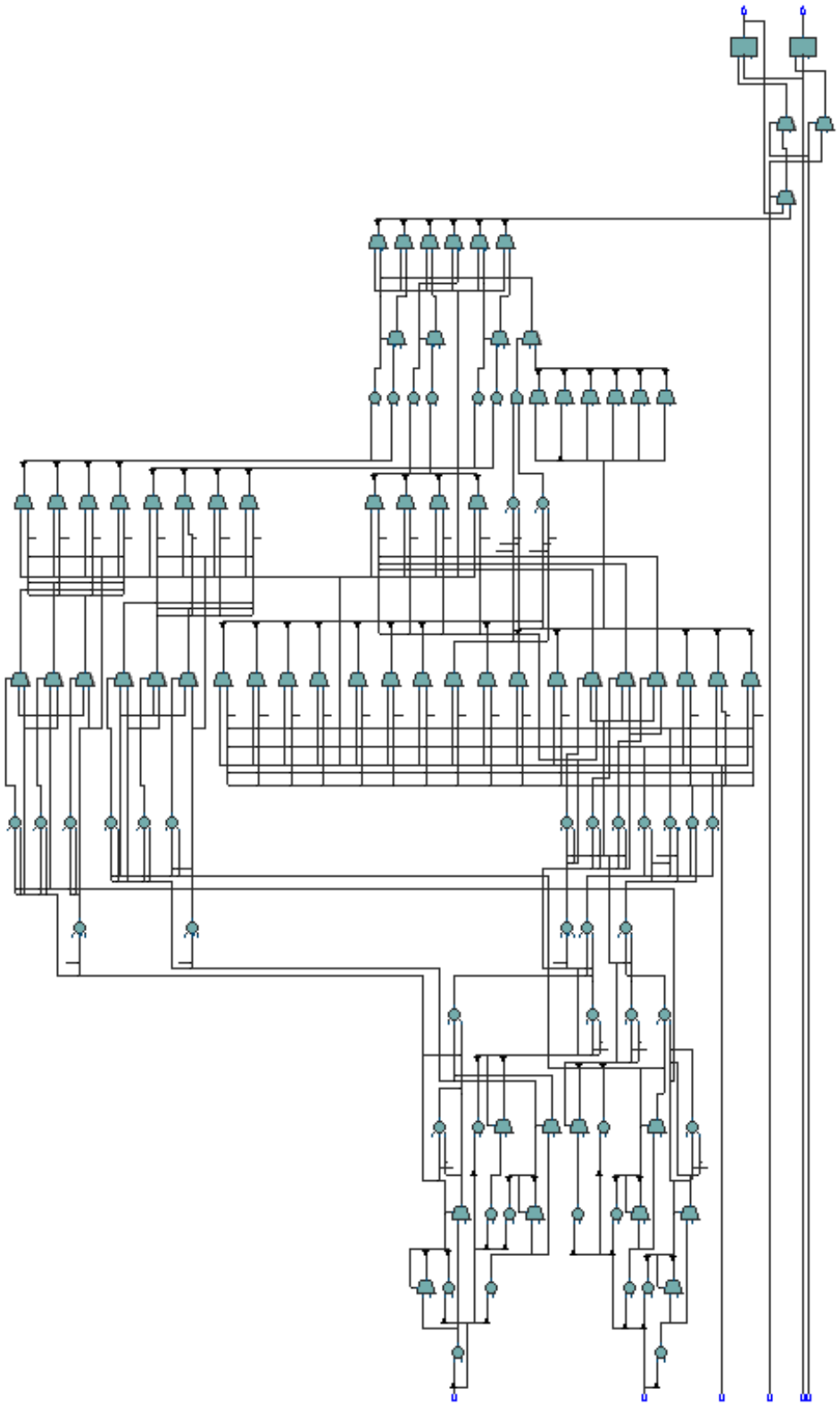
```
        return t_max(a, b);
    end;

    function t_impl(a, b: integer) return integer is
    begin
        return t_max(-a, b);
    end;

begin

    process(clk)
        variable a_val, b_val, result_val: integer;
        variable res_vec: std_logic_vector(5 downto 0);
        variable ra, rb, r: integer;
    begin
        if rising_edge(clk) then
            if reset = '1' then
                trit_out <= (others => '0');
                done <= '0';

            elsif start = '1' then

                case op_code is
                    when “0100” | “0101” | “1111” | “0111” =>
                        for i in 0 to 2 loop
                            ra := trit_to_int(a_in(i*2+1 downto
i*2));
                            rb := trit_to_int(b_in(i*2+1 downto
i*2));

                            case op_code is
                                when “0100” => r := t_and(ra,
rb);   -- AND
                                when “0101” => r := t_or(ra,
rb);    -- OR
                                when “1111” => r := t_not(ra);
-- NOT
                                when “0111” => r := t_impl(ra,
rb);  -- IMPL
                                when others => r := 0;
                            end case;

                            res_vec(i*2+1 downto i*2) :=
int_to_trit(r);
                        end loop;
                        trit_out <= res_vec;
                        done <= '1';
```

Лістинг А.6, аркуш 4

```
                    when others =>
                        a_val := trits_to_int(a_in);
                        b_val := trits_to_int(b_in);

                        case op_code is
                            when “0000” => result_val := a_val
+ b_val;  -- ADD
                            when “0001” => result_val := a_val
- b_val;  -- SUB
                            when “1110” => result_val := a_val
+ 1;      -- INC
                            when “0011” => result_val := a_val
* b_val;  -- MUL
                            when others => result_val := 0;
                        end case;

                        trit_out <= int_to_trits(result_val);
                        done <= '1';

                end case;

            else
                done <= '0';
            end if;
        end if;
    end process;

end Behavioral;
```

Лістинг А.6, аркуш 5

# ДОДАТОК Б



Б.1 — схема АЛУ побудована в RTL Viewer