Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет «Комп'ютерні технології і системи»
(назва факультету)

Кафедра «Електронні обчислювальні машини»
(повна назва кафедри)

21.01.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
до кваліфікаційної роботи
магістра
(ступінь вищої освіти)

на тему: Розробка з використанням ПЛІС і дослідження багатоядерного процесора
за освітньою програмою Комп'ютерна інженерія
зі спеціальності: 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

Виконав: студент групи: КС2221

(підпис студента) / Віктор ДЕМИДОВИЧ /
(Ім'я ПРІЗВИЩЕ)

Керівник: (підпис) /доц., Володимир ШАПОВАЛОВ
(посада, Ім'я ПРІЗВИЩЕ)

Нормоконтролер: (підпис) / доц., Володимир ШАПОВАЛОВ /
(посада, Ім'я ПРІЗВИЩЕ)

Консультанти:

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент (підпис)

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Computer Technologies and Systems
(faculty)

Electronic Computers
(department)

Explanatory Note
to Master's Thesis
(higher education degree)

on the topic: Multicore processor design and research using FPGA

according to educational curriculum Computer Engineering

in the Speciality: 123 Computer Engineering
(speciality and its code)

Done by the student of the group: КС2221 / Viktor Demydovych /
(name, surname)

Scientific Supervisor: / Docent, Volodymyr Shapovalov /
(position, name, surname)

Normative controller: / Docent, Volodymyr Shapovalov /
(position, name, surname)

Supervisors

(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)

Dnipro – 2024

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерні технології і системи
Кафедра: ЕОМ
Рівень вищої освіти: Другий (магістерський)
Освітня програма: Комп'ютерна інженерія
Спеціальність: 123 Комп'ютерна інженерія
(шифр та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ЕОМ
Ігор ЖУКОВИЦЬКИЙ
(підпис) (Ім'я ПРІЗВИЩЕ)
Дата 15.11.2023

ЗАВДАННЯ

на кваліфікаційну роботу **магістра**
(ступінь вищої освіти)

студенту **Демидовичу Віктору Миколайовичу**
(Прізвище, Ім'я По батькові)

1. Тема роботи: **Розробка з використанням ПЛІС і дослідження багатоядерного процесора**

Керівник роботи: **Шаповалов Володимир Олександрович, к.т.н., доцент**
(Прізвище, Ім'я, По батькові, науковий ступінь, вчене звання)

затверджені наказом від "21" квітня 2023р. №333ст

2. Строк подання студентом роботи: 17.01.2024 р.
3. Вихідні дані до роботи: Багатоядерний процесор, реалізований на ПЛІС, повинен виконувати на апаратному рівні матричні операції
4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):
4.1 Аналіз предметної області та постановка задачі
4.2 Обґрунтування кількості ядер, розробка структурної схеми та принципу функціонування багатоядерного процесора
4.3 Розробка багатоядерного процесора (апаратна та програмна частина, створення VHDL-опису)
4.4 Дослідження розробленого багатоядерного процесора

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Завдання видав: (підпис консультанта, дата)	Завдання прийняв: (підпис студента, дата)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та постановка задачі	16.09	15%
2	Обґрунтування кількості ядер, розробка структурної схеми та принципу функціонування багатоядерного процесора	23.10	20%
3	Розробка багатоядерного процесора (апаратна та програмна частина, створення VHDL-опису)	1.11	40%
4	Дослідження розробленого багатоядерного процесора	7.12	15%
5	Оформлення пояснювальної записки	15.01	10%
6	Подання кваліфікаційної роботи до кафедри	17.01	
7	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	22.01	

Студент _____ Віктор ДЕМИДОВИЧ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник роботи _____ Володимр ШАПОВАЛОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи **магістр**: **97 с., 58 рисунків, 1 табл., 3 додатки, 35 джерел.**

Об'єкт дослідження – розроблений засобами ПЛІС багатоядерний процесор для виконання матричних операцій.

Мета роботи – розробити засобами ПЛІС багатоядерний процесор для виконання матричних операцій та дослідити ефективність його роботи.

У першому розділі була описана класифікація архітектури комп'ютерів існуючих систем, зроблений огляд можливостей проектування за допомогою ПЛІС і на основі огляду була сформульована задача.

У другому розділі зроблений огляд існуючих систем, їх проектування та реалізація за допомогою ПЛІС і на основі огляду був обраний необхідний функціонал та логіка побудови багатоядерного пристрою.

У третьому розділі представлено обґрунтування кількості ядер процесора, розробка схеми пристрою, перелічені його модулі та описано загальний алгоритм роботи пристрою.

У четвертому розділі подані розроблені схеми кожного з блоків розробленої у попередньому розділі схеми та наведено опис їх портів.

У п'ятому розділі було проведено тестування розроблених блоків та дослідження ефективності їх роботи.

Результати роботи можуть стати основою (підсистемою) для побудови в майбутньому цілісної системи та проведення різноманітних матричних розрахунків.

Ключові слова: ПЛІС, багатоядерний процесор, матричні операції.

ЗМІСТ

ВСТУП

Однопроцесорні комп'ютери досягли високих показників продуктивності завдяки технологічним та архітектурним досягнення.

Але збільшення їх продуктивності стримується як фізичними, технологічними обмеженнями, так і обмеженими можливостями покращення традиційної архітектури Джона фон Неймана. Такі архітектурні рішення, як спрощена система команд, конвеєрна обробка команд та даних, суперконвеєрна та суперскалярна обробка команд, використання довгого формату команди, векторна обробка команд та природній паралелізм, практично виводять однопроцесорні комп'ютери до граничної межі.

Підвищення продуктивності комп'ютерів шляхом створення паралельних комп'ютерних систем. До таких систем належать як багатопроцесорні комп'ютерні системи, в яких використовується паралелізм розподілу задач на велику кількість вузлів обробки, так і комп'ютерні мережі, які мають іншу форму паралелізму – мережу структурно автономних комп'ютерів.

Сьогодні використання принципів паралельної обробки інформації в архітектурі комп'ютерів є звичним. Практично всі сучасні комп'ютери використовують той або інший вид паралельної обробки інформації.

Вони впроваджувалися в найбільш передових, дорогих, а тому одиничних, комп'ютерах свого часу. Пізніше, після належного відпрацювання технології і здешевлення виробництва, вони використовувалися в комп'ютерах, і нарешті сьогодні вони в повному обсязі втілюються у робочих станціях і персональних комп'ютерах.

Мультипроцесорна система – комп'ютерна система, що має мінімум два процесори, на відміну від однопроцесорної системи, яка має всього один процесор.

Багатоядерний мікропроцесор – мультипроцесор, у якого процесори розташовані на одному кристалі. Отже, процесори в багатоядерних кристалах часто називають ядрами.

Багатоядерний мікропроцесор – мікропроцесор, що містить декілька процесорів («ядер») на одній інтегральної мікросхеми. Процесор, який іноді називають «ядром», — це схема, яка виконує інструкції або обчислення. Оскільки багатоядерний процесор має більше ніж один процесор, він може виконувати обчислення та запускати програми з більшою швидкістю, ніж один процесор.

Багатоядерний процесор реалізує багатопроцесорність в одному фізичному пакеті. Багатоядерні процесори зазвичай використовуються в багатьох сучасних комп’ютерах, смартфонах і планшетах, завдяки чому наші пристрої працюють швидше, ніж з одноядерним процесором. Більша кількість процесорів на чіпі дозволяє виробникам пристроїв розмістити більшу обчислювальну потужність кількох ядер у просторі, який зайняв би одноядерний процесор, і покращити продуктивність електронного пристрою, будь то комп’ютер, ноутбук, смартфон або планшет.

Потрібно відзначити, що збільшення кількості процесорів у багатопроцесорній системі не завжди пропорційно зменшує час вирішення задачі.

Складності паралелізму створює не обладнання, насправді для прискорення виконання завдань на мультипроцесорах були переписані лише деякі відомі прикладні програми. Складно створити таке програмне забезпечення, яке використовує кілька процесорів для прискореного завершення одного завдання, і зі зростанням кількості процесорів ця проблема лише посилюється.

Перша причина полягає в тому, що ви повинні отримати більш високу продуктивність і ефективність від паралельного виконання програм на мультипроцесорній системі, інакше вийде, що ви будете просто використовувати послідовну програму на одному процесорі, оскільки таке програмування простіше. Насправді в таких технологіях конструювання однопроцесорних систем як суперскаляри та процесори зі зміненою послідовністю виконання інструкцій, використовується паралелізм на рівні інструкцій і для цього не потрібне втручання програміста. Подібні нововведення скорочують потреби у

переписуванні програм для мультипроцесорів, оскільки програмістам тут нема чого робити, і навіть їхні послідовні програми працюватимуть на нових комп'ютерах швидше.

Для паралельного програмування труднощі полягають у диспетчеризації, балансі завантаженості, часу на синхронізацію та витратах на обмін даними між учасниками. Ситуація ускладняється зростанням кількості процесорів для паралельного програмування.

Виявили ще одну перешкоду, а саме закон Амдала. Він нагадує нам про те, що у програмі, що розробляється з прицілом на раціональне використання безлічі ядер, необхідно розпаралелити навіть невеликі частини.

Дуже актуальними у світі обчислювальної техніки були і залишаються ПЛІС (програмовані логічні інтегральні схеми, FPGA – Field-Programmable Gate Array), вони мають безліч застосувань у різних галузях. Актуальне використання ПЛІС для прискорення обчислень у таких напрямках, як наукові дослідження, фінансові обчислення, машинне навчання, хмарні технології, криптографія. ПЛІС можуть прискорювати виконання алгоритмів навчання та виведення рішень у реальному часі. Також вони можуть використовуватися в техніці зв'язку для обробки сигналів в якості високопродуктивних цифрових сигнальних процесорів, особливо, в реальному часі. Використання ПЛІС дозволяє змінювати функціональність пристроїв без створення нових мікросхем (ASIC - Application-Specific Integrated Circuit). За допомогою ПЛІС можна створювати оригінальні цифрові пристрої, особливо там, де стандартні рішення на основі ASIC не підходять [1].

На основі проведених досліджень були опубліковані тези доповіді “Можливості розробки багатоядерного процесора з використанням ПЛІС” [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Класифікація архітектури комп’ютерних систем

Розробники архітектури комп'ютерів здавна вдавалися до методів проектування, відомих під загальною назвою "суміщення операцій", при якому апаратура комп'ютера у будь-який момент часу виконує одночасно більше однієї базової операції. Цей загальний спосіб включає два поняття: паралелізм та конвеєризацію.

Прослідкуємо за впровадженням значущих нововведень в архітектурі комп'ютера з точки зору паралельної обробки інформації, починаючи практично з часу створення перших комп'ютерів [2].

Загально розвиток у цій сфері починався з паралельною пам’яті та паралельної арифметики. Вони були характерними для всіх ранніх комп'ютерів, таких як EDSAC, EDVAC та UNIVAC, де слова читалися біт за бітом, тобто послідовно. Перший комерційно доступний комп'ютер із використанням паралельної пам'яті та паралельної арифметики - це IBM 701, який з'явився у 1953 році. У більш популярній та покращеній моделі IBM 704, вперше була використана пам'ять на феритових сердечниках та апаратний арифметичний пристрій з рухомою комою.

Наступним кроком у розвитку стала поява незалежних процесорів введення та виведення. В початкових моделях комп'ютерів процесори виконували не лише обробку інформації, але й керували процесами введення-виведення. Однак робота найшвидшого зовнішнього пристрою, яким була магнітна стрічка в ті часи, була на тисячу разів меншою, ніж швидкість процесора. Це призводило до того, що під час операцій введення-виведення процесор фактично простоював. Наслідком цього стало те, що до комп'ютера IBM 704 у 1958 році додали шість незалежних процесорів введення-виведення, які, отримавши команди від центрального процесора, могли працювати паралельно з ним.

Подальший розвиток архітектури комп’ютерів призвів до можливості розподілу пам'яті та проведення прогнозування. Компанія IBM у 1961 році завершила розробку комп'ютера STRETCH, обладнаного двома ключовими особливостями: передбаченням вибірки команд та розподілом пам'яті на два блоки для вирішення проблеми низької швидкості вибірки з пам'яті порівняно з високою швидкістю виконання операцій.

Невдовзі, архітектура була оновлена створенням конвеєру команд. У 1962 році вперше принцип конвеєрного виконання команд був застосований у комп'ютері ATLAS, який був розроблений в Манчестерському університеті.

Процес виконання команд було розділено на чотири етапи: вибірка команди, обчислення адреси операнда, вибірка операнда і виконання операції. Завдяки конвеєрній структурі вдалося скоротити час виконання команд з 6 мкс до 1,6 мкс.

Через незначний час з'являються незалежні операційні пристрої. У 1964 році був створений комп'ютер CDC-6600 – що став першим комп'ютер, який використовував кілька незалежних операційних пристроїв. CDC 6600 - перший у світі суперкомп'ютер, розроблений і створений американською компанією Control Data Corporation в 1963 під керівництвом інженера-електроника Сеймура Крея, згодом названого "батьком суперкомп'ютерів".

Високу швидкість вдалося досягти завдяки кільком новаторським рішенням: на відміну від поширеної тоді схеми, в CDC 6600 головний процесор комп'ютера виконував лише логічні та арифметичні операції. А робота з периферійними пристроями була покладена на 10 незалежних операційних пристроїв, головним призначенням яких було передача даних з пристроїв введення центральному процесору і забирати результати для відправки на пристрої виведення. Це дозволило розвантажити центральний процесор, скоротити набір його машинних команд до мінімуму та зробити їх виконання дуже швидким, тобто практично реалізувати ідею, яка пізніше, у 1970-х роках, була втілена у RISC-процесорах.

Але розробник Сеймур Крей на цьому не зупинився і в 1969 році компанія CDC випустила комп'ютер CDC-7600, що включав у себе вісім незалежних

операційних конвеєрів, в результаті чого забезпечувалася комбінація паралельної та конвеєрної обробки даних.

Наступним кроком став початок розробки матричного процесора. У 1967 році розпочалися роботи над розробкою матричного процесора ILLIAC IV, що включав досягнення таких параметрів: наявність процесорних елементів (ПЕ) кількістю 256 - 4 квадранти по 64 ПЕ, з можливістю їх реконфігурації по 128 ПЕ на 2 квадранти або із 256 ПЕ на 1 квадрант, довжина такту рівна 40 нс та продуктивністю, що становила 1 млрд. операцій з рухомою комою на секунду.

Як пристрій керування використовувався універсальний комп'ютер із маленькою продуктивністю. Усі ПЕ у складі мали повний набор команд у АЛП та об'єм пам'яті 64-розрядних слів кількістю 2К із циклом 350 нс. У 1974 році комп'ютер був запущений маючи лише 1 квадрант з тактом 80 нс, завдяки чому 50 млн операцій з рухомою комою в секунду стало його реальною продуктивністю.

Тим не менш цей проект значно вплинув на побудову архітектури подальших комп'ютерів, створених на основі подібного принципу, які були названі масивно-паралельними комп'ютерами із розподіленою пам'яттю.

Концепція цих комп'ютерів передбачала використання серійних мікропроцесорів із їхніми локальними пам'яттю, які були з'єднані за допомогою певного комунікаційного середовища. Переваги такої архітектури включають можливість збільшення продуктивності за рахунок збільшення кількості процесорів, або пристосування конфігурації відповідно до обмежених фінансів чи заздалегідь відомої потрібної продуктивності.

Проте існує суттєвий недолік. Міжпроцесорна взаємодія в таких комп'ютерах відбувається значно повільніше, ніж локальна обробка даних самими процесорами, що ускладнює написання ефективних програм для цих систем.

У 1976 році світ бачить перший векторно-конвеєрний комп'ютер CRAY-1. До значних покращень у швидкості призвела відмова від транзисторів на користь інтегральних мікросхем (ІВ), які давали таку щільність упаковки логічних елементів за високої надійності, яку неможливо було досягти за допомогою

транзисторів. Це дозволило без втрати продуктивності підвищити час на такт до 12,5 нс (80 МГц), замість амбітних 8 нс (125 МГц) CDC 8600.

Для CRAY-1 створили процесор, який швидко виконував і скалярні та векторні обчислення. Цього вдалося досягти через створення так званих «векторних регістрів» — модулів пам'яті невеликого об'єму, які були близько до процесора і працювали дуже швидко (але коштували дуже дорого). Таким чином центральний процесор брав дані з регістрів і записував дані також у регістри, реалізуючи новий принцип роботи з пам'яттю «реєстр-реєстр», тоді як CDC STAR-100 використовував колишній спосіб роботи з пам'яттю – «load-store», тобто читання та запис у пам’ять (яка була повільною) безпосередньо. Комп’ютер мав 12 конвеєрних операційних пристроїв. У 1974 році перші випробування цієї машини показали продуктивність на рівні 160 мільйонів операцій в секунду. Основна пам'ять складалася з 64-розрядних слів. Цикл пам'яті тривав 50 нс, а такт становив 12,5 нс.

З поступовим розвитком можливостей технологій та модернізацією виробництва компонентів комп'ютерів та їх архітектури, включаючи пам'ять, з'являється принципово новий тип комп'ютерів – паралельні системи зі спільною пам'яттю. У цих комп'ютерах вся основна пам'ять розділяється між кількома ідентичними процесорами. Хоча технічно не можна розширити кількість процесорів, які мають доступ до спільної пам'яті, безліч сучасних багатоядерних систем, комп'ютери такі як HP Exemplar і Sun StarFire, спрямовані в цьому напрямку.

У традиційній однопроцесорній системі присутній один потік інструкцій та один потік даних, в той час як звичайний мультипроцесор має декілька кілька потоків даних та потоків інструкцій. Для опису цих двох категорій використовуються абревіатури SISD і MIMD відповідно.

Найвідомішою класифікацією подібних архітектур обчислювальних систем, була запропонована 1966 року М.Флінном [3, 4]. Класифікація виходить з понятті потоку, під яким розуміється послідовність елементів, команд чи даних, оброблена процесором.

На основі числа потоків команд і потоків даних Флін виділяє чотири класи архітектур: SISD, MISD, SIMD, MIMD.

• SISD (single instruction stream / single data stream) - одиночний потік команд та одиночний потік даних. До цього класу належать, перш за все, класичні послідовні машини, або інакше, машини фон-нейманівського типу, наприклад PDP-11 або VAX 11/780. У таких машинах є лише один потік команд, всі команди обробляються послідовно одна за одною, і кожна команда ініціює одну операцію з одним потоком даних. Не має значення той факт, що для збільшення швидкості обробки команд і швидкості виконання арифметичних операцій може застосовуватися конвеєрна обробка - як CDC 6600 машина зі скалярними функціональними пристроями, так і CDC 7600 з конвеєрними потрапляють в цей клас.

• SIMD (single instruction stream/multiple data stream) - одиночний потік команд та множинний потік даних. В подібних архітектурах зберігається один потік команд, що включає, на відміну від попереднього класу, векторні команди. Це дозволяє виконувати одну арифметичну операцію відразу над багатьма даними елементами вектора. Спосіб виконання векторних операцій не обумовлюється, тому обробка елементів вектора може проводитися процесорною матрицею, як в ILLIAC IV, або за допомогою конвеєра, як, наприклад, в машині CRAY-1

• MISD (multiple instruction stream/single data stream) - множинний потік команд і одиночний потік даних. Визначення має на увазі наявність в архітектурі багатьох процесорів, які обробляють один і той же потік даних. Однак ні Флінн, ні інші фахівці в галузі архітектури комп'ютерів досі не змогли подати переконливий приклад реально існуючої обчислювальної системи, побудованої на цьому принципі. Ряд дослідників відносять конвеєрні машини до цього класу, проте це знайшло остаточного визнання у науковому співтоваристві.

• MIMD (multiple instruction stream/multiple data stream) - множинний потік команд та множинний потік даних. Цей клас передбачає, що у обчислювальної

системі є кілька пристроїв обробки команд, об'єднаних у єдиний комплекс і кожне зі своїм потоком команд і даних.

Безперечними представниками класу SIMD є матриці процесорів: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 і т.п.

У таких системах єдиний пристрій контролює безліч процесорних елементів. Кожен процесорний елемент отримує від пристрою управління кожен фіксований момент часу однакову команду і виконує її над своїми локальними даними. Для класичних процесорних матриць жодних питань не виникає, однак у цей же клас можна включити і векторно-конвеєрні машини, наприклад, CRAY-1. І тут кожен елемент вектора треба як окремий елемент потоку даних.

Найефективніше система SIMD працює у циклах for з масивами. Тому для того, щоб паралелізм функціонував у SIMD, потрібно мати велику кількість однакових структур даних, що відомо як паралелізм на рівні даних.

1.2 Можливості розробки багатоядерного процесора засобами ПЛІС

Мікросхеми ПЛІС — це спеціальні програмовані логічні пристрої, призначені для перенастроювання користувачами відповідно до їхніх потреб і бажань. Це дає змогу створювати, розробляти та тестувати будь-яку орієнтовану на користувача апаратну архітектуру обробки даних на одній інтегральній схемі. Опис необхідної конфігурації може бути вказаний у вигляді електронної електричної схеми або з використанням мови опису обладнання, наприклад VHDL, Verilog [5] або інших.

Функція масово-паралельних обчислень технології ПЛІС дає розробникам шанс прискорити швидкість алгоритму обробки даних шляхом поділу (якщо можливо) будь-якого окремого обчислювального процесу на багато незалежних одночасних потоків.

Існує багато обчислювальних проблем, які за своєю природою легко розпаралелювати.

За всіх цих обставин мікросхеми ПЛІС показують свої величезні переваги. Звичайно, рішення ПЛІС також має деякі недоліки; наприклад, час розробки

набагато довший, ніж для звичайної системи на основі програмного забезпечення. Однак це не применшує переваг ПЛІС.

Процесори на основі ПЛІС все частіше використовуються у вбудованих системах, призначених для спеціальних програм. Наприклад, Yiannacouras, Rose та Steffan [6] пропонують заснований на ПЛІС, автоматично згенерований векторний процесор для конкретної програми та демонструють можливість масштабування його продуктивності. У своїй статті [7] Coyne, Cyganski та Duckworth представляють співпроцесор на основі ПЛІС для прискорення методу SART для локалізації радіочастотного джерела. Система була розроблена на VHDL і використовує паралелізм на багатьох рівнях алгоритму SART. Іншим прикладом є процесор на базі ПЛІС, масивно-паралельний, з однією інструкцією, з кількома потоками даних (SIMD), представлений у [8].

ПЛІС можуть забезпечити прискорене виконання операцій, пов'язаних також з графічними обчисленнями (зараз для таких обчислень широко використовуються ASIC графічні процесори). У пристроях IoT (Internet of Things, інтернет речей) ПЛІС використовуються для обробки даних у реальному часі. Ефективно також використовувати їх для обробки мережевих пакетів та оптимізації обчислень у розподілених середовищах. З урахуванням своєї гнучкості та можливості програмування, ПЛІС продовжують залишатися актуальними в інноваційних та високотехнологічних галузях. Зокрема, ПЛІС можуть бути використані для матричних обчислень. Одним з можливих шляхів реалізації матричних обчислень є створення багатоядерного спеціалізованого процесора. Такий процесор повинен підтримувати виконання інструкцій над безліччю даних одночасно. З цього випливає, що архітектура такого процесора повинна бути SIMD (Single Instruction, Multiple Data).

Провідними фірмами з випуску ПЛІС та відповідного програмного забезпечення – систем автоматизованого проектування (САПР) є AMD (Xilinx) та Intel (Altera). У деяких ПЛІС, наприклад, таких як Xilinx Virtex і Xilinx UltraScale+ є виділені блоки цифрової обробки сигналів (прискорення обчислень) - DSP48E1, які можуть бути використані для реалізації паралельних

обчислень в ПЛІС і запрограмовані для ефективної обробки матричних операцій. У ці блоки закладені апаратні ресурси для виконання арифметичних операцій, у тому числі й множення. Таким чином, за допомогою блоків DSP48E1 можна виконувати як операції складання та віднімання, так і скалярне та векторне множення матриць. За основу ядра процесора можна взяти блок DSP48E1.

В даний час основним засобом проектування пристроїв на ПЛІС фірми Xilinx є САПР Vivado. Проектування ведеться за допомогою мов проектування апаратури (HDL – Hardware Description Language), наприклад, VHDL. Такий підхід називається низькорівневим проектуванням. Також є засоби високорівневого проектування, які дозволяють істотно скоротити час проектування. До таких засобів, наприклад, можна віднести програму Simulink у системі Matlab. При цьому можна провести моделювання багатоядерного процесора з урахуванням усіх модулів, форматів команд та даних. Після проходження етапу високорівневого проектування далі програмою HDL Coder генерується HDL-опис пристрою, наприклад, мовою VHDL. На основі отриманого VHDL-опису в САПР Vivado проводиться моделювання на рівні всіх сигналів, а також етап синтезу та реалізації пристрою в ПЛІС. При необхідності можна ввести корективи в пристрій, що розробляється, шляхом зміни VHDL-опису.

Матричне множення є обчислювально складною проблемою, особливо проектування та ефективна реалізація на ПЛІС, де ресурси дуже обмежені, були більш вимогливими. Проекти на основі ПЛІС зазвичай оцінюються за трьома показниками продуктивності: швидкість (затримка), площа та потужність (енергія). Реалізації з фіксованою точкою в ПЛІС є швидкими та мають мінімальне енергоспоживання. Крім того, блок матриці множника з фіксованою точкою часто потребує менше кремнію в ПЛІС або ASIC, ніж його аналог з плаваючою комою. Обмеження числа з фіксованою комою полягає в тому, що дуже великі та дуже малі числа не можуть бути представлені, а діапазон обмежений бітовою шириною числа. Була велика попередня робота в області

розробки системи на основі ПЛІС для обчислення множника матриці з фіксованою комою.

В даний час ведеться безліч досліджень у галузі ПЛІС реалізації матричних операцій для матриць з плаваючою комою. Реалізація матриці з плаваючою комою на ПЛІС і обчислення зворотної матриці пропонується з використанням VHDL і IP Cores в [9], з використанням Verilog в [10], з використанням синтезу високого рівня в [11], використання систем на кристалі (SoC) і концепцій обчислень, що реконфігуруються, описаних в [12]. Реалізація ПЛІС з плаваючою комою множення матриць пропонується з використанням VHDL [13], з використанням ModelSim в [14] і з використанням OpenCL в [15]. Проектування на основі моделей для реалізації ПЛІС з використанням MATLAB Simulink та DSP Blockset.

У [16] обговорюється методологія проектування для синтезу сімейства дуже компактних систолічних матриць на ПЛІС, що базується, по суті, на ручному відображенні на рівні CLB у поєднанні зі структурним рівнем VHDL. Автори [17] використовували множення матриць як еталон для порівняння продуктивності ПЛІС, DSP і вбудованих процесорів. Результати показують, що ПЛІС можуть множити дві матриці з меншою затримкою та меншим енергоспоживанням, ніж інші два типи пристроїв. Це робить ПЛІС ідеальним вибором для множення матриць у програмах обробки сигналів.

Аміра та ін. представив нову архітектуру, засновану на систолічній архітектурі для множення матриці [18]. Використано послідовно-паралельний матричний множник на основі алгоритму Боу-Вулі. Дизайн, заснований на систолічній архітектурі, був реалізований за допомогою Xilinx XCV1000E сімейства Virtex-E ПЛІС. Аміра та ін. розробив параметризовану систему для 8-розрядного множення матриці з фіксованою комою за допомогою ПЛІС [19]. Їх дизайн використовував як систолічну архітектуру, так і методологію розподіленого арифметичного проектування для реалізації множення матриці.

Дизайн на основі розподіленої арифметики забезпечує кращу продуктивність з точки зору швидкості та площі порівняно з дизайном на основі систолічного

масиву. Пропускна здатність вводу/виводу, необхідна за проектом, прямо пропорційна розміру проблеми. Конструкції, представлені в [20, 21], були обмежені малим розміром матриці. Для множення великих матриць (n=128, 256 і 512) Bensaali та інші розробив співпроцесор на основі ПЛІС [22]. Розроблений співпроцесор спочатку розбиває вхідні матриці на менші підматриці, а потім обчислює добуток.

У [23] Mencer та ін. реалізовано множення матриць на пристрої Xilinx XC4000E ПЛІС. У їх конструкції використовуються бітові послідовні помножувачі з використанням кодування Бута. Вони зосередилися на компромісах між площею та максимальною робочою частотою з генераторами параметризованої схеми. Їх дизайн був вдосконалений Амірою та ін. в [24, 25] з використанням модифікованого множення кодувальника кабіни разом із додаванням дерева Уоллеса. Для n=4 було використано 296 CLB для досягнення максимальної робочої частоти 60 МГц за допомогою Xilinx XCV1000E ПЛІС.

Jang та ін. покращив дизайн у [23-25] щодо площі, швидкості [26] та енергії [27], використовуючи переваги повторного використання даних. Вони зменшили затримку для обчислення матричного продукту, використовуючи внутрішні регістри зберігання в елементі обробки (PE). Для алгоритмів потрібні n множників, n суматорів і загальна пам’ять розміром n2 слів. Для множення матриці 4 × 4 затримка схеми в [28] становить 0,57 мкс, тоді як схема в [26, 27] використовує 0,15 мкс, використовуючи на 18 % менше площі порівняно з [23]. Belkacemi та ін. [28] представив дизайн і реалізацію високопродуктивного, повністю паралельного ядра множення матриць. Ядро було параметризоване та масштабоване з точки зору розмірів матриці (тобто кількості рядків і стовпців) і довжини слова вхідних даних. Повністю заплановані конфігурації ПЛІС були створені автоматично з високорівневих описів операції множення матриць у формі списку з’єднань у форматі електронного дизайну (EDIF) менш ніж за одну секунду. Що були спеціально оптимізовані для мікросхем Xilinx Virtex ПЛІС.

Використовуючи велику кількість логічних ресурсів у Xilinx Virtex ПЛІС (LUT, логіка швидкого перенесення, регістри зсуву, тригери тощо), досягається

повністю паралельна реалізація ядра матричного множника; з результатом повної матриці, що генерується кожного такту. Екземпляр помножувача матриці 3 × 3 споживає 2448 сегментів Virtex і може працювати на 175 МГц на чіпі XCV1000E-6 Virtex-E. Традиційно показниками продуктивності для проектів на основі ПЛІС були затримка та площа. Однак із поширенням портативних мобільних пристроїв стає все більш важливим, щоб системи також були енергоефективними та мали низьке енергоспоживання. У пристроях ПЛІС основна частина енергії споживається програмованими з'єднаннями, тоді як решта енергії споживається блоками синхронізації, логіки та введення/виведення.

Іншим джерелом розсіювання потужності в ПЛІС є використання ресурсів і комутаційна діяльність [29]. Дослідницькі зусилля щодо розробки енергоефективного матричного помножувача описані в [27], [30-32]. Більшість попередніх робіт із множення матриці з фіксованою точкою зосереджувалися лише на зменшенні затримки та області.

Чой та ін. розробив нові конструкції та архітектури для ПЛІС, які мінімізують енергоспоживання разом із затримкою та площею [30, 31]. Вони використовували лінійну систолічну архітектуру для розробки енергоефективних конструкцій. Для архітектури лінійного масиву обсяг пам'яті на елемент обробки впливає на загальносистемну енергію. Таким чином, вони використали максимальний обсяг пам'яті на елемент обробки та мінімальну кількість помножувачів, щоб отримати енергоефективний матричний помножувач. Функція часткової реконфігурації була вперше використана для обчислення множення матриці Jianwen та ін. в [33]. Пристрої, що частково реконфігуруються, пропонують можливість зміни реалізації проекту без зупинки всього процесу виконання. Матричний множник реалізовано в пристрої Xilinx Virtex-II, який підтримує часткову реконфігурацію. Конструкцію оцінювали з точки зору затримки та площі, і було виявлено, що площа зменшується на 72%-81% для розмірів матриці від 3 × 3 до 48 × 48 порівняно з [26], а продуктивність ще більше покращується для великих матриць.

Підхід до проектування системи на основі підключення, що скорочує час розробки та забезпечує швидке створення прототипів. Інші переваги Проектування на основі моделей включає візуальне представлення реальної системи, що легко масштабується для проектів, що вимагають більш високої точності, включення індивідуальних модулів, автоматичне створення вбудованого програмного коду та інтегрований дизайн з легкістю налагодження та апаратного спільного моделювання.

У наступному розділі більш детально наведені приклади розроблених архітектур та алгоритмів для роботи безпосередньо з матрицями та проведення розрахунків над ними.

1.3 Постановка задачі

Розробка багатоядерного процесора за допомогою ПЛІС та його дослідження є складним процесом та вимагає великого об’єму попереднього проектування. Результатом розробки є пристрій з багатоядерним процесором який вміє виконувати матричні операції таких як додавання, віднімання та множення.

Саме розробка багатоядерного процесора та алгоритм роботи пристрою з ним для вирішення задачі матричної арифметики та дослідження його ефективності є головними результатами кваліфікаційної роботи.

З урахуванням виконаного аналізу існуючих можливостей систем розроблених на ПЛІС щодо роботи з матрицями та побудови багатоядерного процесору необхідно розробити пристрій для вирішення наступних задач:

- завантаження на пристрій інструкцій програм для процесору;
- завантаження даних матриць над якими будуть виконуватись матричні арифметичні операції;
- розшифрування інструкції та згідно з отриманими даними передача даних матриць до відповідних модулів розробленої системи для проведення подальших розрахунків;
- формування результуючої матриці та її передача до оперативної пам’яті;

- виведення даних з оперативної пам’яті за умови закінчення усіх розрахунків.

2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ РЕАЛІЗАЦІЇ МНОЖЕННЯ МАТРИЦЬ НА ПЛІС

2.1 Розробка та оцінка операцій з матрицею з плаваючою комою для проектування системи на основі FPGA

У роботі Chetan S, Sourabh KS, Lekshmi V, Sudhakar S, Manikandan J «Design and Evaluation of Floating point Matrix Operations for FPGA based system design» [34] розроблявся спеціалізований процесор для матричного множення та знаходженню зворотньої матриці з використанням чисел з плаваючою комою на платі ПЛІС.

У цій статті розглядалась апаратна реалізація та оцінка двох основних матричних операцій (матриця з плаваючою комою) пропонується отримання зворотної матриці та множення матриць з плаваючою комою з використанням системного проектування на основі моделей для ПЛІС. Подробиці про різні архітектури/підходи проектування матричного помножувача та масштабування пропонованої конструкції для двох матричних операцій з використанням великих за розміром матриць.

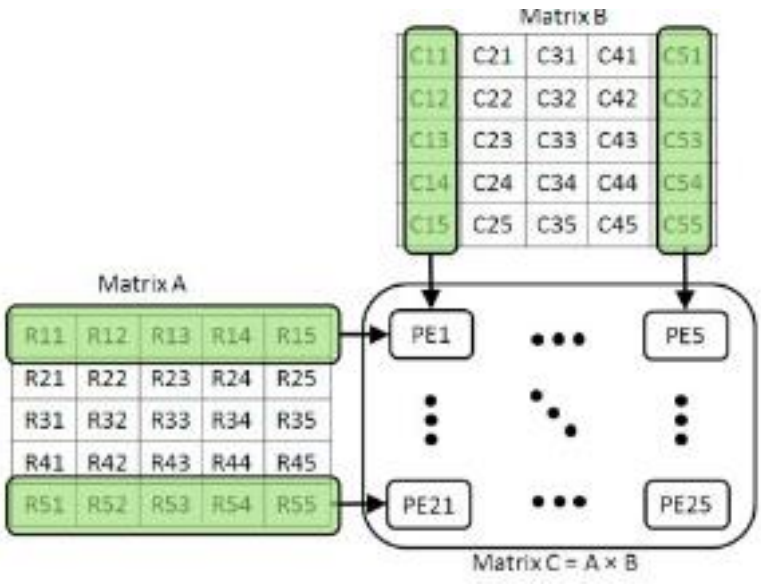

Рисунок 2.1 – Запропоноване матричне множення – архітектура 1

Описані різні архітектури/підходи проектування матричного помножувача та масштабування пропонованої конструкції для двох матричних операцій з використанням великих за розміром матриць.

Для операції матричного множення розгядалося три види архітектур. Нижче наведено детальний опис кожної з них.

У першій архітектурі використовуються N^2 ідентичні процесорні елементи (PE), як показано на рисунку. Рисунку 2.1 для матриці розміру NxN. Функціональність кожного PE-модуля показана на рисунку 2.2 для матриці 5 × 5. Основна перевага цього підходу полягає в тому, що він використовує функцію паралелізму ПЛІС, що забезпечує швидше результат за рахунок збільшення ресурсів.

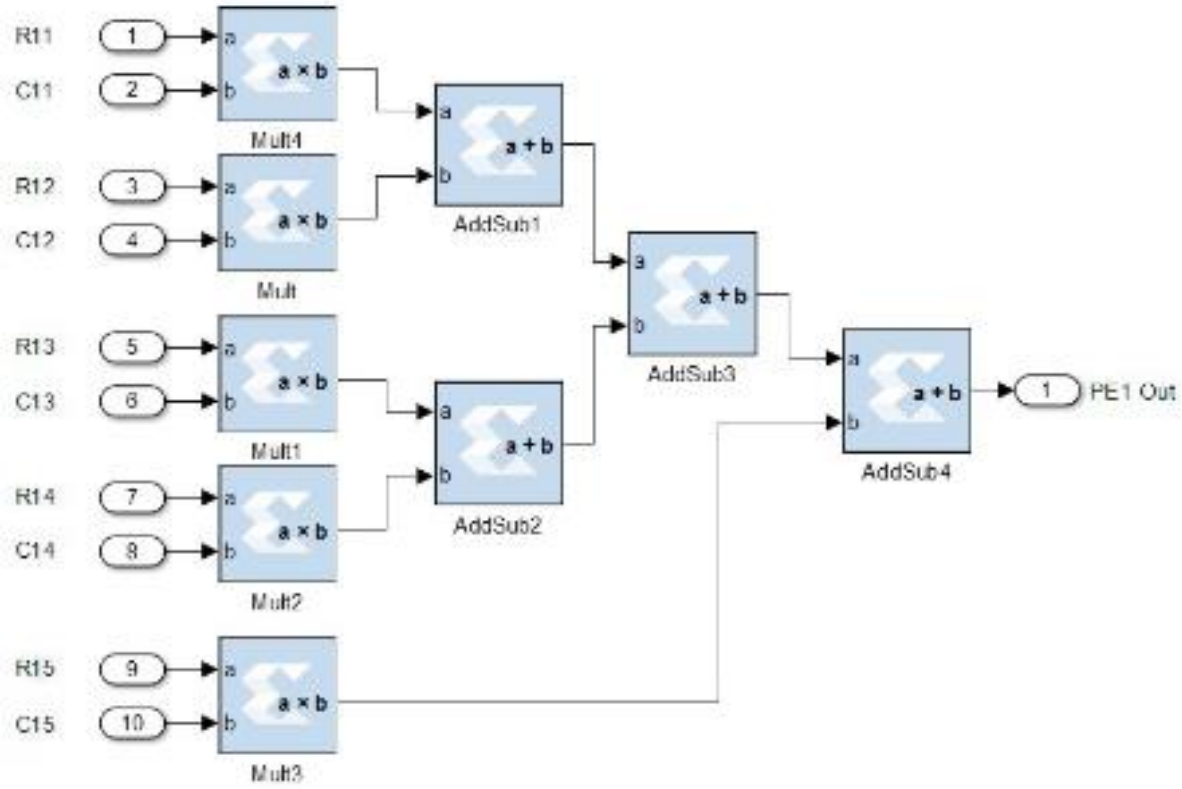

Рисунок 2.2 – Процесорний елемент для архітектури 1

У другій архітектурі використовується інший PE з N ідентичними PE для розміру матриці N на N, як показано на рисунку 2.3.

Функціональність кожного модуля PE для матриці 5×5 показана на рисунку 2.4.

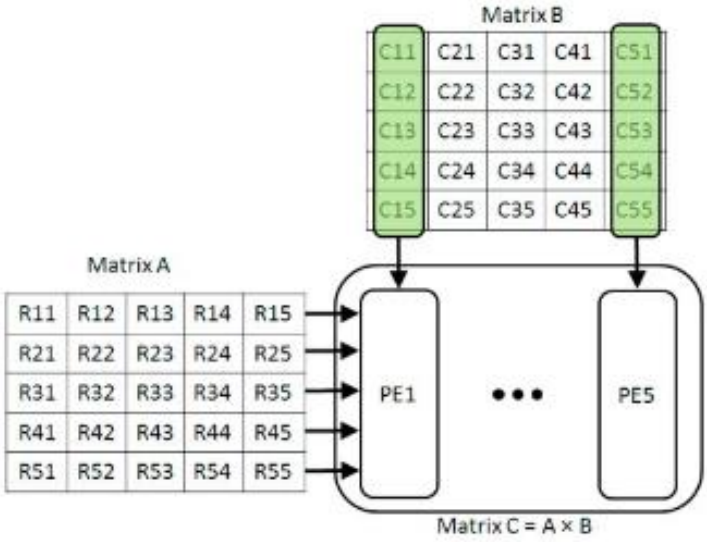

Рисунок 2.3 – Запропоноване матричне множення – архітектура 2

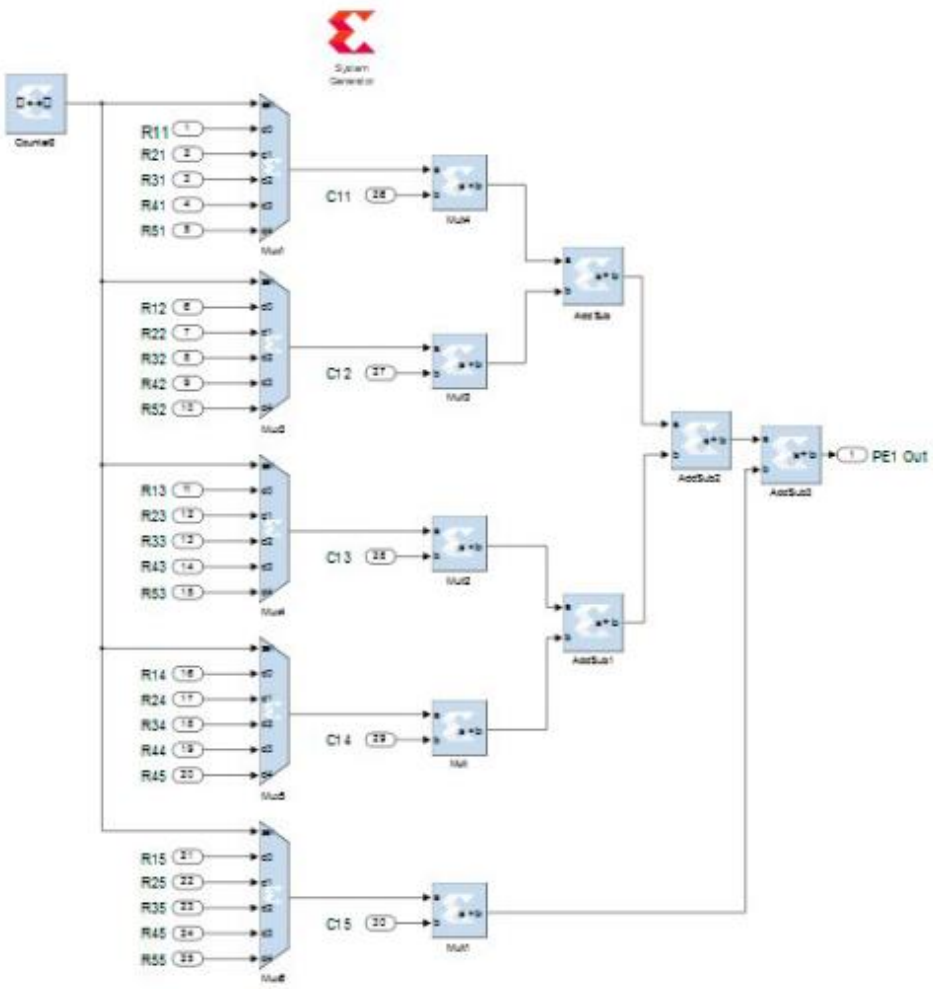

Рисунок 2.4 – Процесорний елемент для архітектури 2

PE використовується мультиплексор для послідовного зчитування рядків матриці A, а ресурси всередині PE повторно використовуються для обчислення вихідних елементів.

У третій архітектурі використовується лише один PE, як показано на рисунку 2.6, який використовується повторно N2 разів, а функціональність PE показана на рисунку 2.5.

Цей підхід використовує мінімальні ресурси рахунок збільшення затримки виводу. Вищезгадані підходи мають свої плюси та мінуси. Залежно від проектних вимог та через апаратні обмеження можна використовувати один із цих підходів.

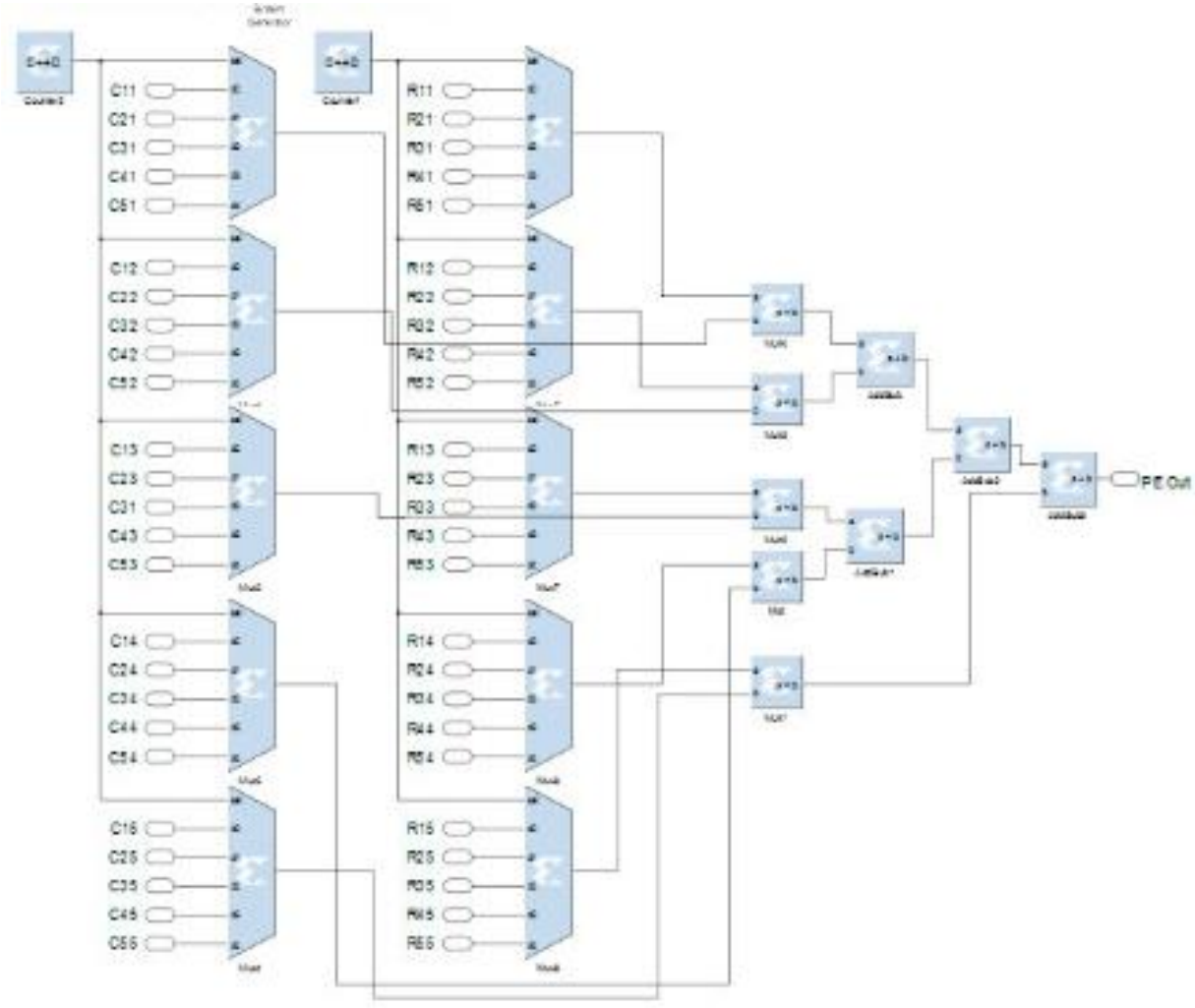

Рисунок 2.5 – Процесорний елемент для архітектури 3

Запропоновані підходи до множення матриць можна легко масштабувати до великих розмірів матриць за рахунок збільшення кількості PE для першого та другого підходу, тоді як третій підхід вимагає збільшення кількості ітерацій.

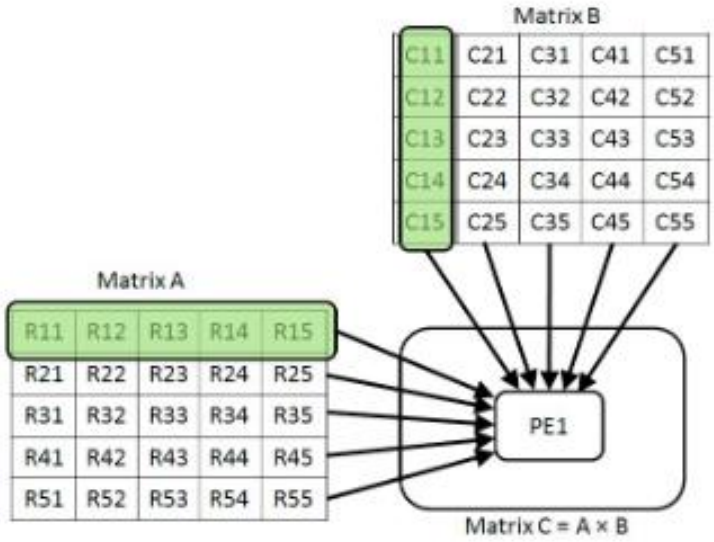

Рисунок 2.6 – Запропоноване матричне множення – архітектура 3

2.2 Проектування та реалізація спрощеного матричного процесора на ПЛІС

У роботі Mostafa I. Soliman, Elsayed A. Elsayed «Design and FPGA Implementation of a Simplified Matrix Processor» [35] був спроектований простий матричний процесор (SMP) для виконання скалярних/векторних/матричний інструкцій.

У цій статті пропонується простий матричний процесор під назвою SMP для виконання скалярних/векторних/матричних інструкцій по одному операційному автомату (рисунку 2.7). SMP використовує явний паралелізм, виражений високорівневими інструкціями замість динамічного вилучення за допомогою складної логіки або статичного вилучення за допомогою складних компіляторів.

Крім векторних інструкцій, SMP використовує інструкції "матриця-скаляр", "матриця-вектор" та "матриця-матриця" для вираження паралелізму з апаратним забезпеченням. За допомогою SMP можна використовувати до трьох рівнів паралелізму даних (скалярні операції для векторної обробки, скалярні операції

для матрично-векторної обробки та скалярні операції для матричної обробки). Це призводить до високої продуктивності, простої моделі програмування та компактного виконуваного коду. SMP спроектовано як універсальну архітектуру завантаження-зберігання, в якій доступ до пам'яті можливий лише за допомогою інструкцій завантаження або збереження. Це зменшує трафік пам'яті, підвищує щільність коду та кодує інструкції фіксованої довжини.

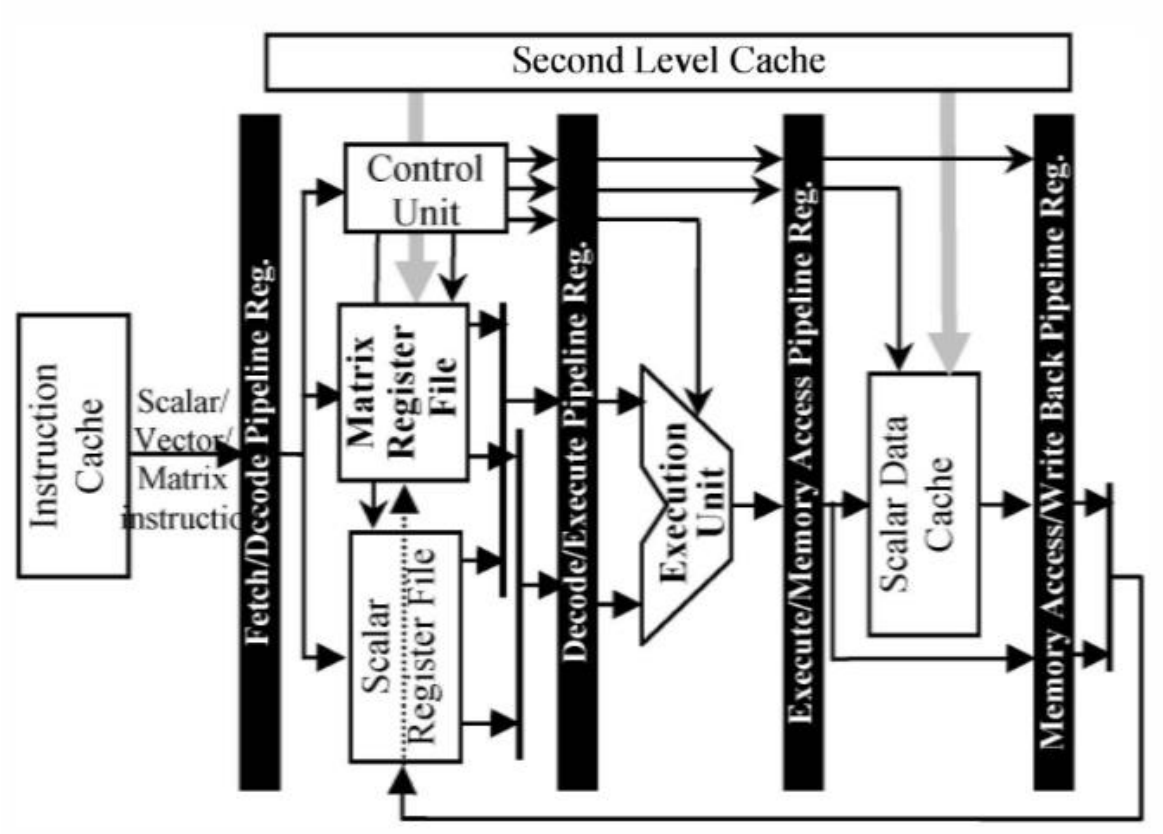

Рисунок 2.7 – Функціональна схема SMP процесору

Процесор SMP розширює скалярну ISA високорівневими інструкціями з обробки матричних і векторних даних на одному і тому операційному автоматі виконання. Матричні регістри та їх пов'язані регістри управління додаються до скалярної ISA. Традиційно векторні архітектури обробляють векторні дані під контроль довжини вектора, кроку, максимальної довжини вектора та маски регістрів. В якості прямого розширення обробки матриць необхідні наступні регістри управління: Strps і Wstrp, які містять кількість рядків і кількість елементів на рядок відповідно. Strps x Wstrp Елементи блоків обробляються за допомогою векторних/матричних інструкцій. Для по елементних

векторних/матричних інструкцій, таких як по елементне додавання, віднімання, множення і т. д., Strps і Wstrp зчитуються блоком управління для генерації правильних сигналів керування для обробки блоків матриць Strps Wstrp або Strps*Wstrp рядків векторів. Інші інструкції, такі як множення матриці на матрицю, вимагають трьох параметрів обробки блоків даних. Регістр керування Dim використовується для зберігання третього параметра. Залежно від коду операції інструкції, що виконується, блок управління генерує сигнали управління після читання регістрів управління Strps/Wstrp або Strps/Wstrp/Dim.

На рисунку 2.8 показано мікроархітектура пропонованого SMP-процесора. Скалярна/векторна/матрична інструкція програми витягується з кешу команд на етапі вибірки та зберігається у регістрі конвеєра вибірки/декодування.

Стадія декодування має скалярні та матричні регістри, а також основний блок управління. Вхідна скалярна/векторна/матрична інструкція з регістру конвеєра вибірки/декодування декодується і її операнди читаються з скалярних або матричних регістрів. Крім того, генеруються керуючі сигнали для керування виконанням вибраної команди. Скалярна інструкція виконується один такт на загальних функціональних блоках. Проте векторна/матрична інструкція виконує n кроків загальних функціональних блоків. Кожен етап включає вибірку елементів з матричних регістрів на етапі декодування і виконання операції над обраними елементами на етапі виконання. n залежить від вмісту регістрів керування Strps, Wstrp та Dim.

Інструкції завантаження/збереження в процесорі SMP переміщують скалярні (0-D), векторні (1-D) та матричні (2-D) дані між скалярними/матричними регістрами та пам'яттю. Скалярна пам'ять доступ здійснюється через кеш даних першого рівня (L1), у якому зберігається лише скалярні дані. Вектора і матриці мають доступ безпосередньо в кеш-пам'ять другого рівня (L2).

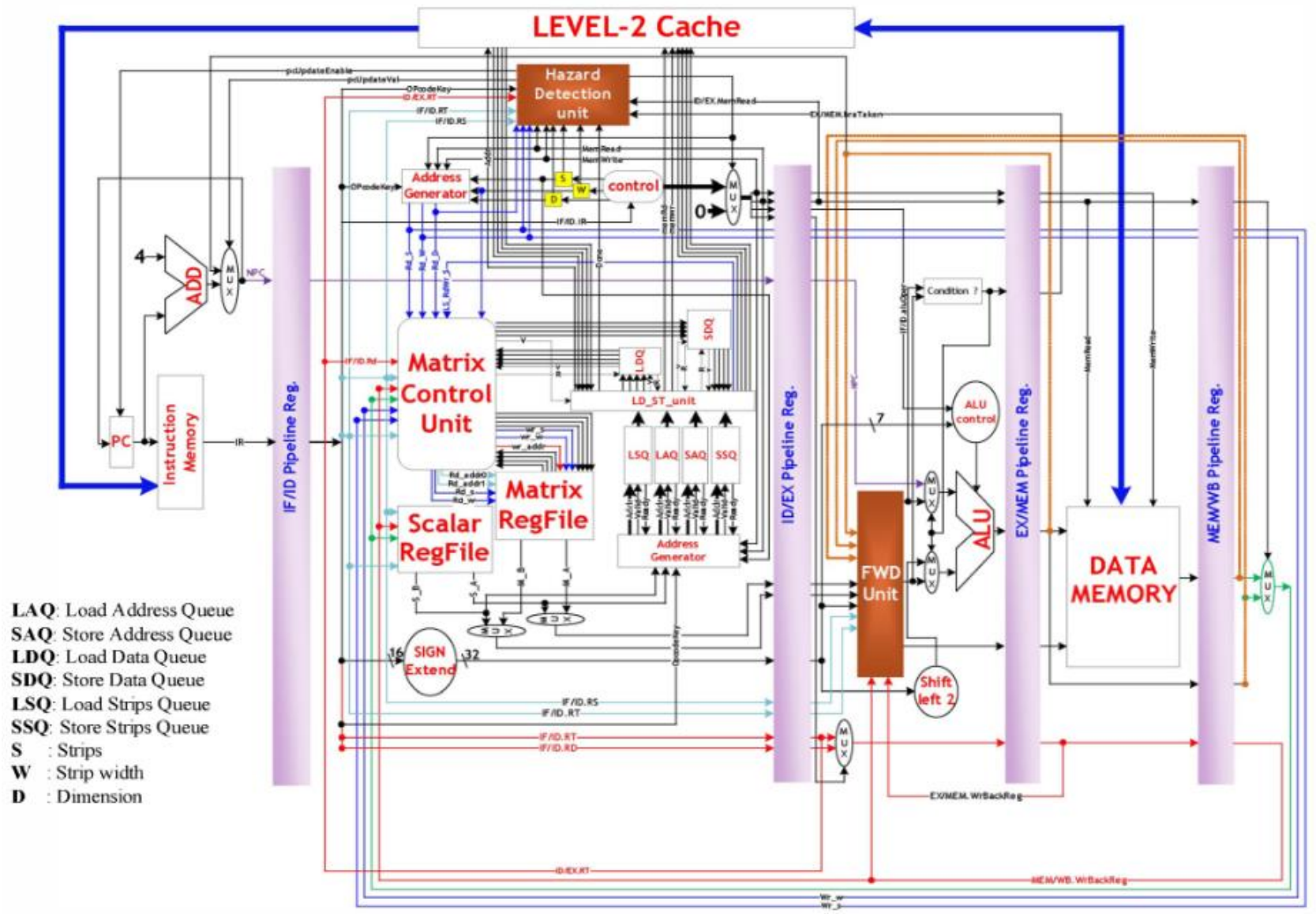

Рисунок 2.8 – Мікроархітектура SMP процесора

Найпростішою формою завантаження/зберігання блоку даних є зміщенням для вектор завантаження/збереження, яке передає набір елементів (одномірний масив) між суміжними осередками пам'яті та регістрів. Базова адреса цих суміжних одномірних елементів зазвичай визначається вмістом скалярного регістру. Адреса генератор на етапі декодування генерує у пам'яті серію адреси (тільки одну адресу за такт); кожна адреса переміщає чотири елементи з пам'яті.

Оскільки дані матриці (двовимірний масив) зберігаються в пам'яті як одновимірний масив, завантаження/збереження зсуву для матриці може виконуватися аналогічно завантаженню/збереженню зміщення для вектора.

2.3 Розробка та реалізація архітектур матричних помножувачів для програм обробки зображень і сигналів

У роботі «FPGA design and Implementation of Matrix Multiplier Architectures for Image and Signal Processing Applications» Qasim, S. M., Telba, A. A. та AlMazroo, A. Y. у своїй роботі [32] представили дизайн і реалізацію програмованої вентильної матриці (ПЛІС) архітектур матричних помножувачів для використання в програмах обробки зображень і сигналів. Конструкції оптимізовані для швидкості, яка є основною вимогою в цих програмах. Перший дизайн передбачає обчислення щільного множення матриці-вектора, яке використовується в програмі обробки зображень. Проект реалізовано на ПЛІС Virtex-4, і продуктивність оцінюється шляхом обчислення часу виконання на ПЛІС. Результати впровадження показують, що він може забезпечити пропускну здатність 16970 кадрів на секунду, що цілком достатньо для більшості програм обробки зображень. Друга конструкція передбачає множення триматриці (трьох матриць), яка використовується в програмі обробки сигналів. Запропонована конструкція для множення трьох матриць була реалізована на платформі ПЛІС Spartan-3 і Virtex-II Pro відповідно. Представлені результати впровадження, які демонструють придатність ПЛІС для таких застосувань.

Нижче наведено методику проектування апаратної архітектури для реалізації алгоритму множення матриці-вектора на FPGA. Множення матриці на вектор є операцією ядра та задля ефективної реалізації та максимального прискорення використовується цілочисельна арифметика. Оскільки блок арифметики з плаваючою комою споживає більше кремнієвої нерухомості FPGA і працює повільніше порівняно з цілочисельною арифметикою, ми використовували цілочисельну арифметику для наших проектів.

Проект передбачає обчислення G = AC, де A — матриця, C і G — вектори, підсумовані в таблиці 1. Нам потрібно обчислити вектор G. Для множення матриці на вектор прийнято широкомовний алгоритм. Множення матриця–вектор виконується трансляцією рядків матриці A та множенням відповідних елементів стовпця вектора C.

Задіяні наступні операції:

• Зчитування окремих елементів рядка матриці A та окремих елементів стовпця вектора C

• Зберігання їх у внутрішніх буферах рядка і по стовпцях відповідно

• Перемноження елементів рядків і стовпців

• Накопичення виводу множника та записування результатів у вихідні буфери.

Таблиця 2.1 – Матриці, що приймали участь у проведених розрахунках

Matrix Symbol	**Matrix Dimension**
A	1024×28
C	28×1
G	1024×1

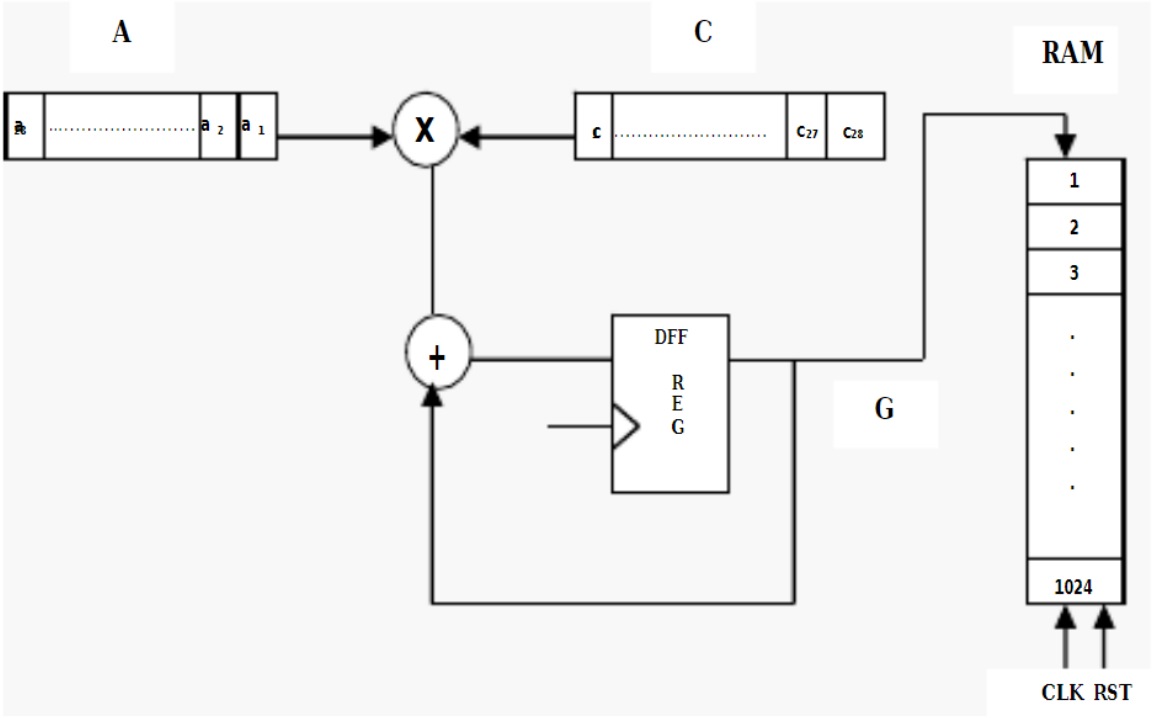

Рисунок 2.9 – Блок-діаграма множення матриця-вектор

Вхідні та вихідні буфери реалізовані на ПЛІС. Матричні векторні множення містять операції множення та накопичення. Блок множення-накопичування складається з множника та суматора. Елементи рядка та стовпця подаються як два входи до множника. Вихід множника безпосередньо передається суматору як один із входів. Попередній вихід суматора подається назад як другий вхід суматора. Блок множення-накопичення бере кожен елемент матриці A у форматі головного рядка та кожен елемент вектора C, перемножує їх і додає результат до поточного підсумку. Цей процес повторюється до останнього елемента рядка A та стовпця C. Значення подаються послідовно. Якщо сигнал скидання встановлений високим, вміст регістрів A і C очищається. Після затримки, визначеної результатами реалізації, перший елемент вектора G стає доступним на послідовному виході, і цей вихід зберігається у вбудованій пам’яті. Ця операція повторюється, і процес триває, доки не будуть оброблені всі рядки матриці A. Нарешті, вихідний вектор G доступний з усіма елементами, що

зберігаються в місцях пам'яті. Спрощена структурна схема множення матриці-вектора наведена на рисунку 2.9.

Результати вказують на доцільність використання FPGA для програм високошвидкісної обробки зображень у режимі реального часу з використанням множення матриці на вектор.

Друга частина дослідження стосувалася дизайн триматричного помножувача, який зазвичай використовується в додатках DSP [36]. Матрицю T можна записати як T = XYZ, де X і Z — прямокутні матриці, визначені на рисунку 2.10 і рисунку 2.11 відповідно. Y – діагональна квадратна матриця, де n = 0, 1, …, N-1.

$$X = \begin{bmatrix} 0 & 0 & \cdots & 0 & x_1(0) & \cdots & x_1(N-k-1) \\ 0 & \cdots & \cdots & x_1(0) & x_1(1) & \cdots & x_1(N-k) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & x_1(0) & \cdots & \cdots & \cdots & \cdots & x_1(N-2) \\ x_1(0) & x_1(1) & \cdots & \cdots & \cdots & \cdots & x_1(N-1) \\ x_1(1) & x_1(2) & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1(k) & x_1(k+1) & \cdots & x_1(2k-1) & x_1(2k) & \cdots & 0 \end{bmatrix}$$

Рисунок 2.10 – Прямокутна матриця X

$$Z = \begin{bmatrix} 0 & 0 & \cdots & x_2(0) & x_2(1) & \cdots & x_2(k) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & x_2(0) & \cdots & \cdots & \cdots & \cdots & x_2(2k-1) \\ x_2(0) & x_2(1) & \cdots & \cdots & \cdots & \cdots & x_2(2k) \\ x_2(1) & x_2(2) & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_2(N-k-1) & x_2(N-k) & \cdots & x_2(N-1) & 0 & \cdots & 0 \end{bmatrix}$$

Рисунок 2.11 – Прямокутна матриця Z

Система для наведеної вище математичної формулювання перетворюється на два блоки, у яких перший блок множить матрицю X на діагональну матрицю Y, а потім подає вихідні дані цього блоку в інший блок, який множить добуток XY

на Z. Ми використали два- архітектура на основі розмірної систолічної матриці, як показано на рисунку 2.12 та рисунку 2.13 для множення матриці.

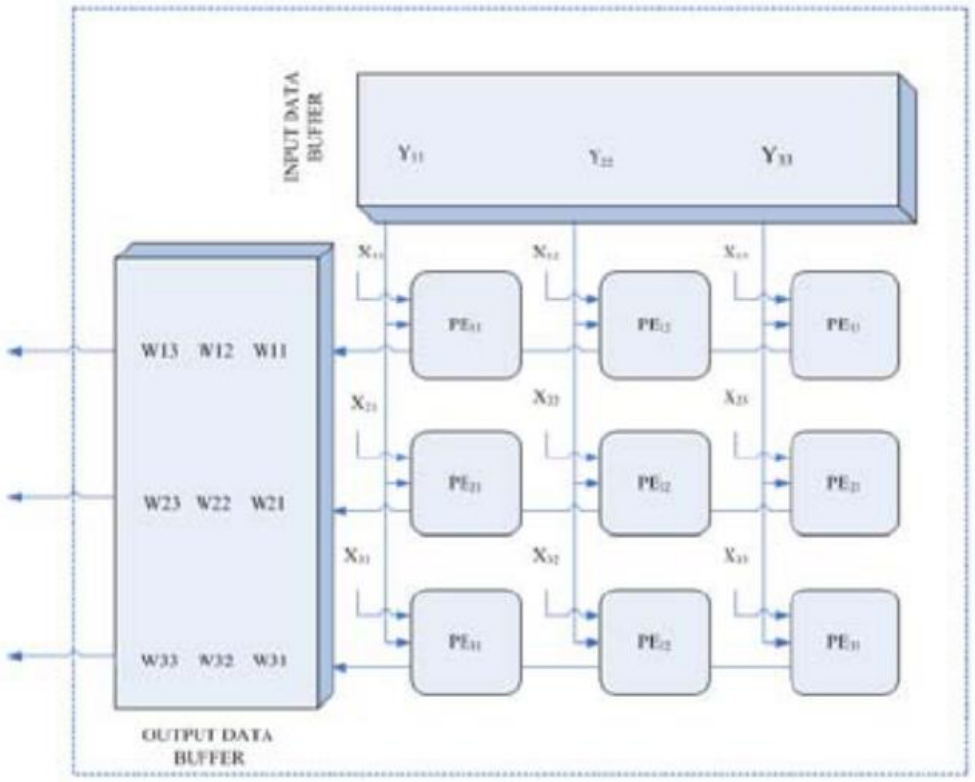

Рисунок 2.12 – Прямокутна матриця Z

Систолічні масиви прискорюють множення матриць середнього розміру, використовуючи притаманний паралелізм даних у множенні матриць. Множення матриці X на діагональну квадратну матрицю Y еквівалентно множенню першого діагонального елемента на елементи першого рядка X, другого діагонального елемента на елементи другого рядка X і так далі. Рисунку 2.12 і рисунку 2.13 показана систолічна архітектура для обох модулів для N1=3 і N2=3 відповідно. Обидва блоки помножувача матриці складаються з дев'яти ідентичних елементів обробки PE1 і PE2 відповідно. PE1 складається з помножувача, тоді як PE2 складається з блоку MAC, де кожен блок MAC складається з помножувача та суматора.

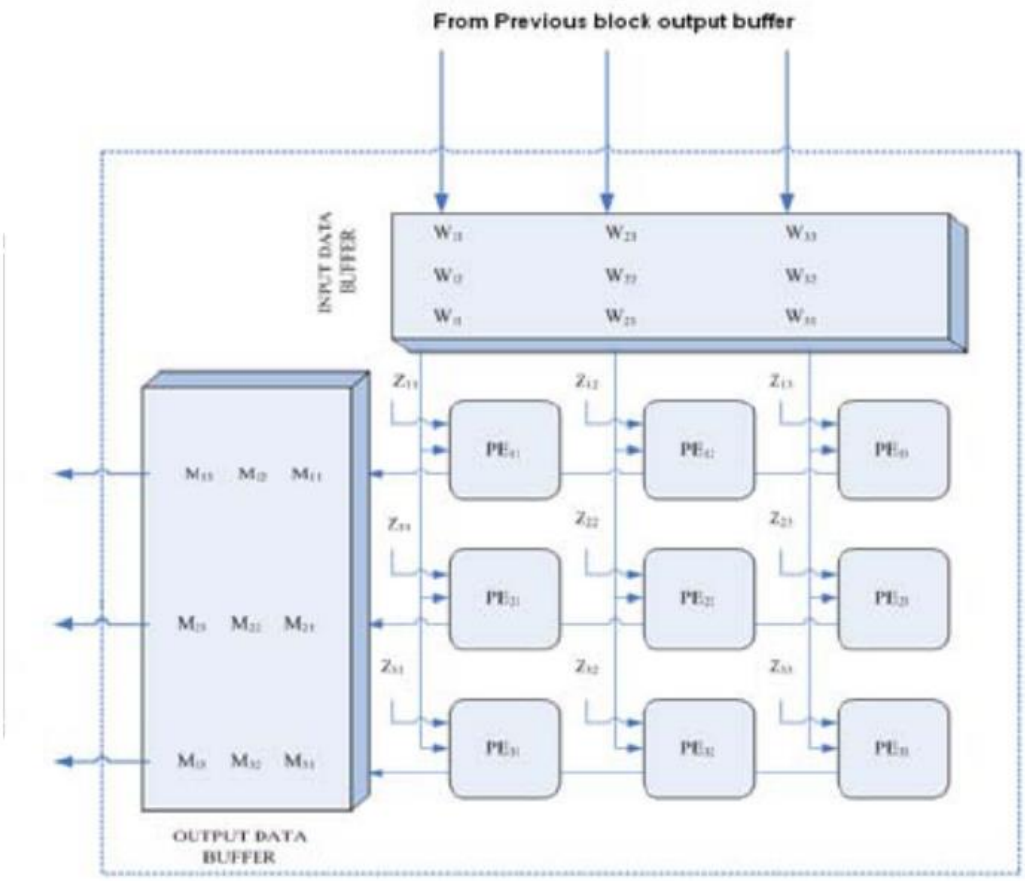

Рисунок 2.13 – Прямокутна матриця Z

Функція кожного PE1 у першому масиві полягає в тому, щоб множити діагональні елементи Y на один елемент матриці X протягом кожного тактового періоду. Перший стовпець PE1 відповідає за створення першого стовпця продукту XY, позначеного W на рисунку 4, другий стовпець створює другий стовпець і так далі. Записи зберігаються у внутрішньому буфері для подальшого використання наступним масивом. Аналогічно, другий масив, як показано на рисунку 14 виконує множення (XY) на Z.

3 ОБҐРУНТУВАННЯ КІЛЬКОСТІ ЯДЕР, РОЗРОБКА СТРУКТУРНОЇ СХЕМИ ТА ПРИНЦИПУ ФУНКЦІОНУВАННЯ БАГАТОЯДЕРНОГО ПРОЦЕСОРА

3.1 Обґрунтування кількості ядер

В ході дослідження і аналізу існуючих рішень задачі побудови багатоядерного процесору для виконання матричних операції та їх розрахункових можливостей, для створення багатоядерної системи був використаний підхід на основі розробки спеціалізованого процесору для матричного множення та знаходженню зворотної матриці з використанням чисел з плаваючою комою на платі ПЛІС [34], де для отримання кожного елементу результуючої матриці використовується окремий процесорний елемент.

Наведемо алгоритм такого множення на рисунку 3.1.

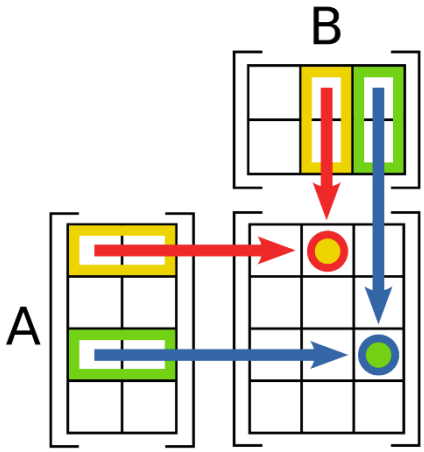

Рисунок 3.1 – Алгоритм множення матриць

Таким чином, через проведення паралельних розрахунків елементів матриці, уся результуюча матриця розраховується за один такт. Оскільки створений багатоядерний процесор буде виконувати операції з матрицями 4 на 4, то загальна кількість ядер процесору становить 16.

3.2 Розробка структурної схеми

Для роботи подібного спеціалізованого багатоядерного процесору необхідно розробити структуру пристрою, який буде подавати інструкції та данні до процесору, необхідно включити до схеми, що розроблюється, відповідний порт для їх завантаження, а також виведення результуючих даних.

Також пристрій повинен мати у своєму складі оперативну пам’ять для збереження інструкцій, та оперативну пам’ять даних для зберігання елементів матриць, задіяних в розрахунках (початкові та результуючі). Необхідно включити до складу пристрою перелік регістрів окремо для двох матриць, над якими виконуються розрахунки та результуючої матриці.

Наступним елементом пристрою має бути генератор адреси регістрів для запису даних матриць з оперативної пам’яті даних до відповідного їй регістру й також для проведення зворотної операції зчитування елементів результуючої матриці з відповідних регістрів до оперативної пам’яті даних.

Останній необхідний модуль пристрою – це кінцевий автомат, що буде відповідальним за відправлення керуючих сигналів у системі в залежності від загального стану пристрою.

На підставі усього вище сказаного, узагальнена структура пристрою з багатоядерним процесором для реалізації матричних операцій (додавання, віднімання та множення) має наступний вигляд, представлений на рисунку 3.2.

Модуль PORT_A відповідальний за зв’язок з зовнішнім пристроєм. На нього передається наступна інформація: елементи матриць для проведення подальших розрахунків та відповідні інструкції щодо реалізації матричних операцій (код операції, адреса початкового елементу матриці, розмір матриці та кількість її стовпців, адреса першого елемента другої матриці, кількість елементів та кількість стовпців другої матриці, адреса першого елементу матриці-результату).

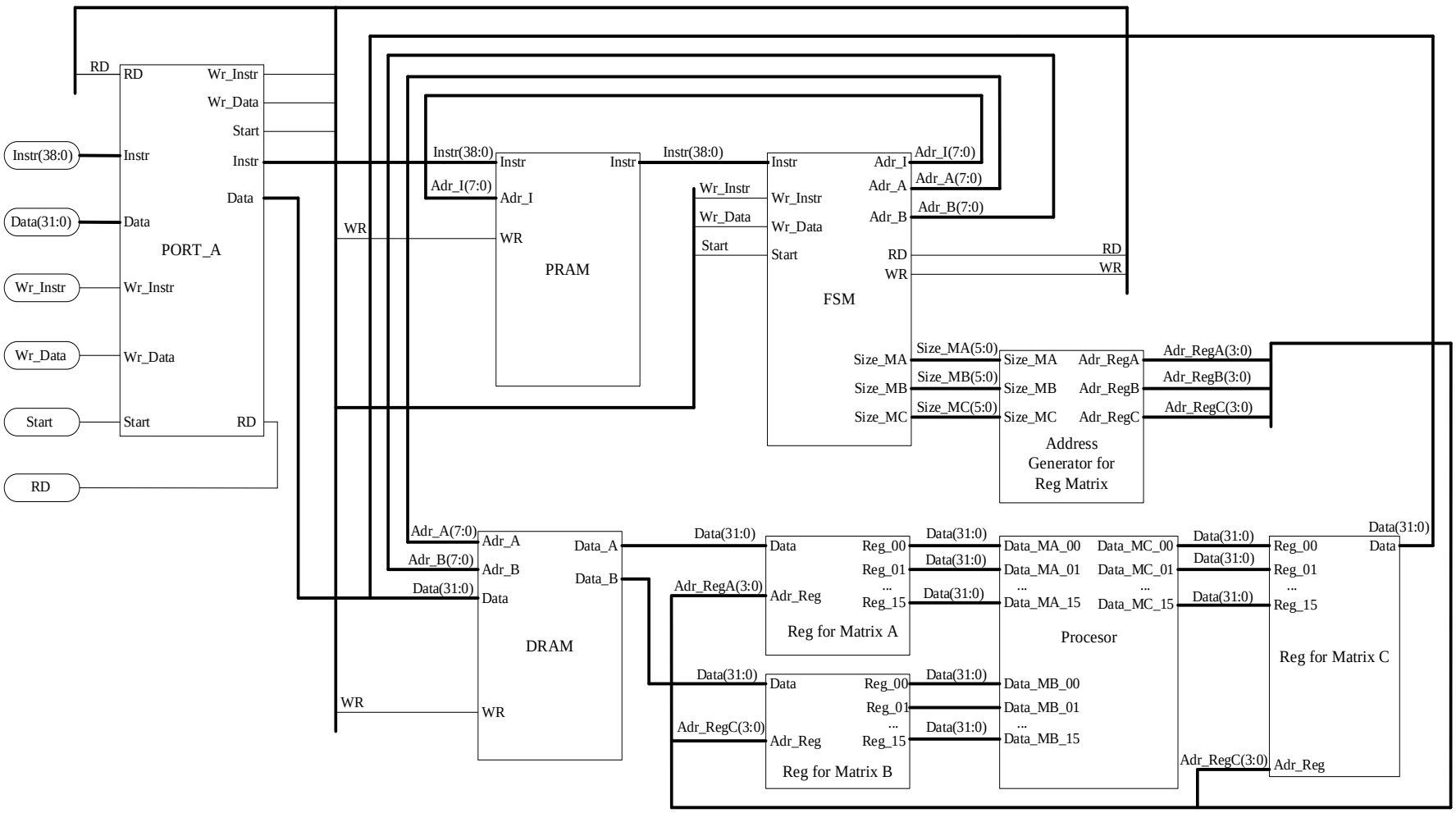

Рисунок 3.2 – Узагальнена структура пристрою з багатоядерним процесором , що виконує матричні операції

Також з цього порту будуть виводитись отриманні результати після закінчення проведення розрахунків.

Модель взаємодіє з PRAM, DRAM та FSM. До PRAM надсилаються інструкції, до DRAM надсилаються данні матриць (елементи матриць) після відправлення інструкцій до PRAM. Та взаємодія з FSM полягає у відправці керуючих сигналів, таких як: початок завантаження PRAM, початок завантаження DRAM та старту розрахунків.

Модуль PRAM являє собою оперативну пам’ять програми, зберігає інструкції програми та подає їх до FSM. Оперативна пам’ять програми буде мати 256 адрес, кожна з яких зберігає 39 біт (розмір інструкції). Загальний обсяг складає 1248 байт.

Модуль DRAM двоканальна оперативна пам’ять для даних, зберігає елементи матриці та передає їх до регістрів матриці A та регістрів матриці B. Регістр матриці C записає результуючі данні після проведення розрахунків у цей модуль для подальшого їх виведення на порт A. Оперативна пам’ять даних буде мати 256 адрес, кожна з яких зберігає 32 біта (розмір даних). Загальний обсяг складає 1024 байт.

Модуль FSM являє собою кінцевий автомат, що розшифровує інструкцію, отримує з неї переданий код операції, адресу першого елементу матриці A, адресу першого елементу матриці B, а також кількість елементів кожної з матриці та кількість їх рядків (size_MA, size_MB). Також розшифровує адресу для першого елементу результуючої матриці C, яку передає DRAM для запису отриманих даних після проведення потрібних розрахунків. Проводить розрахунки розміри результуючої матриці C (size_MC). Передає керуючі сигнали такі, як WR для зчитування та запису даних з оперативної пам’яті даних та сигнал Enable кожного з представлених елементів, що відображає готовність елементу до роботи.

Модуль Reg for Matrix A зберігає елементи матриці A для подальшої передачі до процесору. Модуль складається з 16 регістрів.

Модуль Reg for Matrix B зберігає елементи матриці B для подальшої передачі до процесору. Модуль так само складається з 16 регістрів.

Модуль Reg for Matrix C зберігає елементи матриці C для подальшого її запису до оперативної пам’яті даних. Як і попередні, цей модуль також складається з 16 регістрів.

Модуль PE представляє собою багатоядерний процесор, що складається з 16 ядер. Працює з елементами матриці та проводить відповідні до інструкції розрахунки.

Модуль Address Generator for Reg Matrix відповідальний за розрахунок адреси регістру матриці A та регістру матриці B до якого будуть записуватися данні з оперативної пам’яті. Відповідно розраховує регістр матриці C для подальшої передачі даних з регістру матриці C до оперативної пам’яті даних.

3.3 Алгоритм роботи пристрою

Нижче наведено алгоритм роботи пристрою. Представлений алгоритм в загальному вигляді візуалізовано на діаграмі станів FSM, що зображено на рисунку 3.3.

1. Першим кроком є передача до порту A з зовнішньогог пристрою сигналу Wr_Instr та інструкція Instr (38:0), Wr_Instr подається до FSM кінцевого автомату, що виставляє по цьому сигналу адресу інструкції для оперативної пам’яті програми, в яку вона буде записана. З наступним синхросигналом зовнішній пристрій подає наступну інструкцію, а FSM інкрементує адресу її зберігання в оперативній пам’яті програми.

2. Другим кроком, при умові, що усі інструкції були передані до пристрою, з зовнішнього пристрою передається сигнал Wr_Data та Data (31:0), Wr_Data подається до FSM кінцевого автомату, що виставляє по цьому сигналу адресу даних для оперативної пам’яті даних, в яку вони будуть записані. З

наступним синхросигналом зовнішній пристрій подає наступні дані, а FSM інкрементує адресу їх зберігання в оперативній пам’яті даних.

3. Третім кроком за умови, що усі дані були передані до пристрою, з зовнішнього пристрою передається сигнал Start для початку роботи пристрою.

4. Наступним кроком FSM виставляє адресу першої інструкції.

5. Оперативна пам’ять видає інструкцію кінцевому автомату.

6. FSM згідно отриманої інструкції виставляє адресу першого елементу матриці A та адресу першого елементу матриці B і відправляє ці адреси до оперативної пам’яті даних, також відправляє дані розмірів матриць над якими проводяться розрахунки та результуючої матриці до генератору адрес регістрів матриць. Окрім того, кінцевий автомат відправляє код операції до ядер процесору. Якщо інструкція є кодом кінці програми, переходимо на п.16.

7. Генератор адрес регістрів матриць генерує адресу першого елементу матриці A та матриці B для регістрів матриці A та регістрів матриці B.

8. DRAM оперативна пам’ять даних відправляє за виставленою адресою елемент матриці A до регістрів матриці A та за встановленою адресою елемент матриці B так само до регістрів матриці B.

9. FSM інкрементує адресу оперативної пам’яті даних для наступного елементу матриці A та інкрементує адресу оперативної пам’яті даних для наступного елементу матриці B. Генератор адрес регістрів матриць генерує адресу наступних елементів матриці A та матриці B для регістрів матриці A та регістрів матриці B. Після цього повторюється п.8 цього алгоритму до моменту запису усіх елементів обох матриць до їх відповідних регістрів.

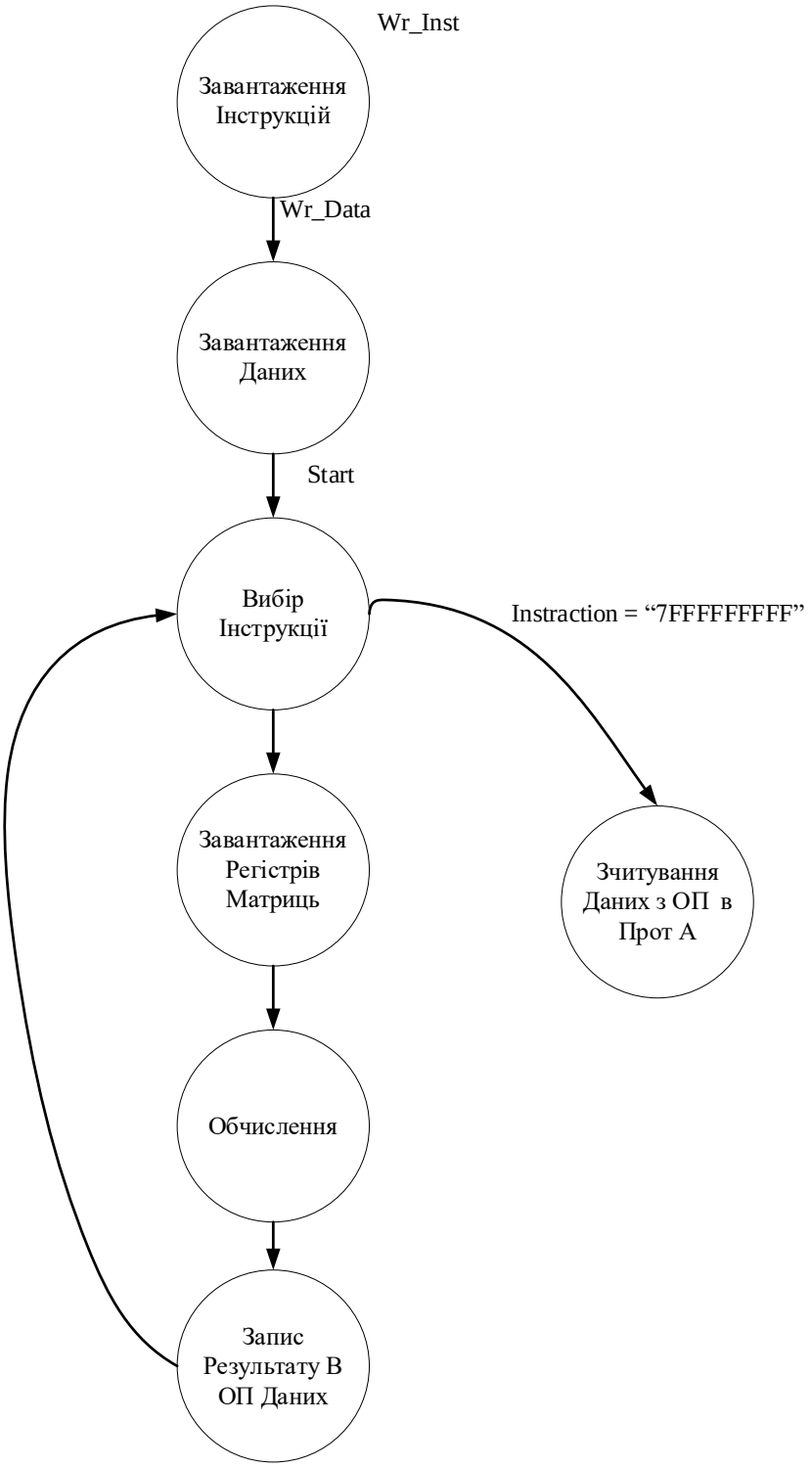

Рисунок 3.3 - Діаграма станів кінцевого автомату FSM

10. Коли усі елементи матриць A та B будуть записані до відповідних регістрів, ці регістри передаватимуть їх до ядер процесору для проведення потрібних математичних розрахунків, згідно коду операції з інструкції.

11. Результати розрахунків кожного елементу результуючої матриці C передаються до регістрів матриці C.

12. FSM виставляє адресу оперативної пам’яті даних для запису першого елементу результуючої матриці C. Генератор адрес регістрів матриць генерує адресу першого елементу результуючої матриці C з її регістрів.

13. До оперативної пам’яті записуються елемент матриці C.

14. FSM інкрементує адресу оперативної пам’яті даних для наступного елементу матриці C, генератор адрес регістрів матриць генерує адресу наступного елементу результуючої матриці C з її регістрів. Після цього повторюється п.13 цього алгоритму до моменту запису усіх елементів матриці до відповідних регістрів.

15. Далі FSM інкрементує адресу інструкції та повертається на п.5.

16. За умови закінчення виконання програми кінцевим автоматом виставляється сигнал RD, який передаєть з порту A до зовнішнього пристрою та FSM виставляє початкову адресу пам’яті даних.

17. Данні з оперативної пам’яті передаються до зовнішнього пристрою через порт A.

18. Кінцевий автомат FSM інкрементує адресу оперативної пам'яті даних та повторюється п.17 поки усі дані з оперативної пам'яті даних не будуть передані до зовнішнього пристрою.

3.4 Загальна структура інструкції пристрою

Формат інструкцій, що подається на порт A, зберігається в оперативній пам’яті програми та розшифровується FSM представлено на рисунку 3.4.

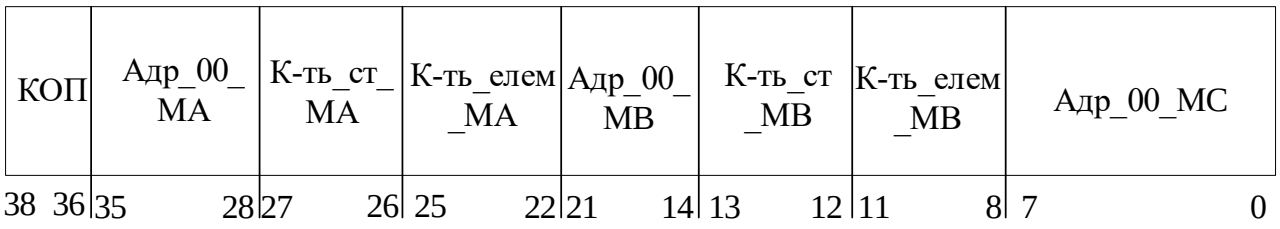

Рисунок 3.4 – Формат інструкцій

Наведемо та розшифруємо основні позначення з інструкції:

- КОП – код операції;
- Адр_00_МА – адреса першого елементу матриці А в оперативній пам’яті даних;
- К-ть_ст_МА – загальна кількість стовпців матриці А;
- К-ть_елем_МА – загальна кількість елементів матриці А;
- Адр_00_МВ – адреса першого елементу матриці В в оперативній пам’яті даних;
- К-ть_ст_МВ – загальна кількість стовпців матриці В;
- К-ть_елем_МВ – загальна кількість елементів матриці В;
- Адр_00_МС – адреса першого елементу матриці С в оперативній пам’яті даних.

Розмір інструкції складає 39 біт, де КОП (38:36), Адр_00_МА(35:28), К-ть_ст_МА (27:26), К-ть_елем_МА (25:22), Адр_00_МВ (21:14), К-ть_ст_МВ (13:12), К-ть_елем_МВ (11:8), Адр_00_МС (7:0) відповідно.

3.5 Логіка роботи ядер процесору з матрицями

На рисунку 3.5 представлено логіку передачі даних до ядер процесору. Наприклад: до Core-1 передається перший рядок матриці А та перший стовпець матриці В для проведення матричних операцій над ними. До Core-6 передаються другий рядок матриці А та другий стовпець матриці В, у той час, як до Core-15 буде передаватися четвертий рядок матриці А та третій стовпець матриці В.

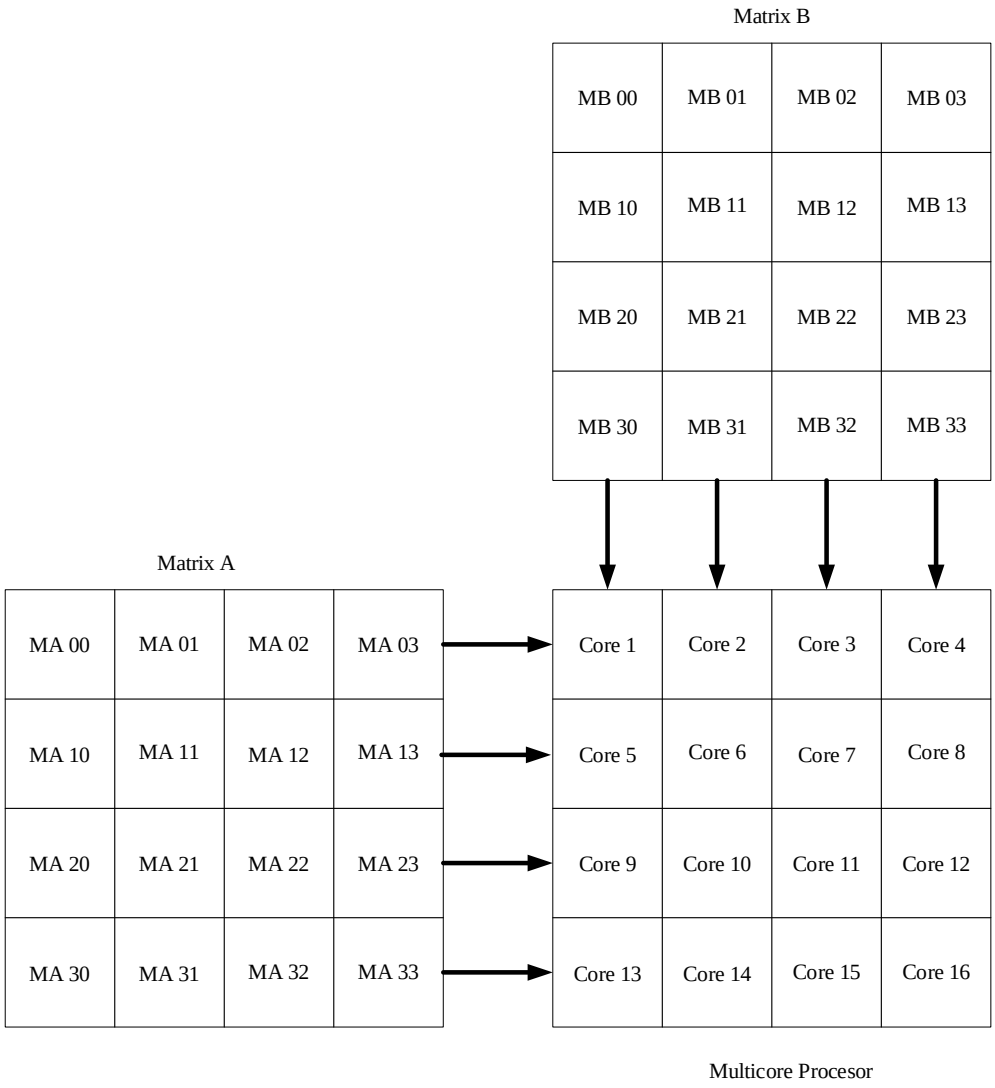

Рисунок 3.5 – Передача елементів матриць до ядер процесора для проведення арифметичних операцій

4 РОЗРОБКА БАГАТОЯДЕРНОГО ПРОЦЕСОРА

Для забезпечення необхідного функціоналу та побудови відповідної до розробленої у попередньому розділі системи, було створено наступні елементи пристрою.

4.1 Блок Core

Блок Core – відповідальний за проведення матричних розрахунків над елементами матриці, що передаються до нього. На рисунку 4.1 наведена схема портів цього елементу: MA_00, MA_01, MA_02 та MA_03, що містять данні рядка матриці A; MB_00, MB_10, MB_20 та MB_30, що відповідно містять данні стовпця матриці B; COP – код необхідної операції для розрахунку; CLK – синхросигнал та ENA – сигнал enable для підтвердження готовності елементу до початку обчислень.

Вихідним сигналом є MC_00, що містить результуючий елемент матриці C, отриманий в результаті проведених обчислень.

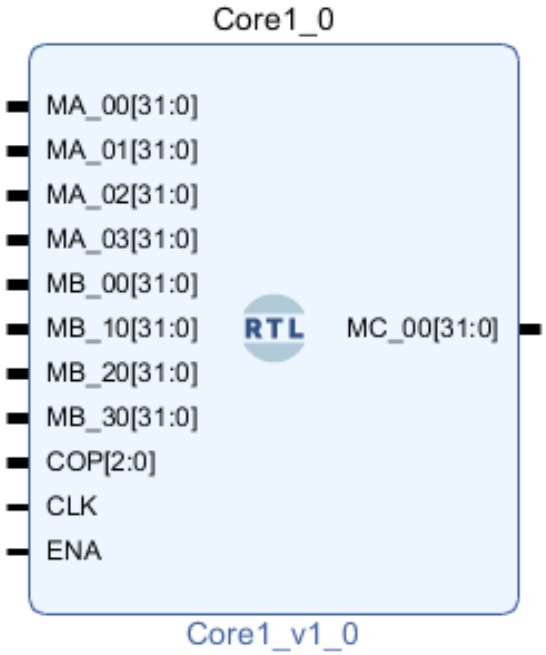

Рисунок 4.1 – Схема портів ядра процесору

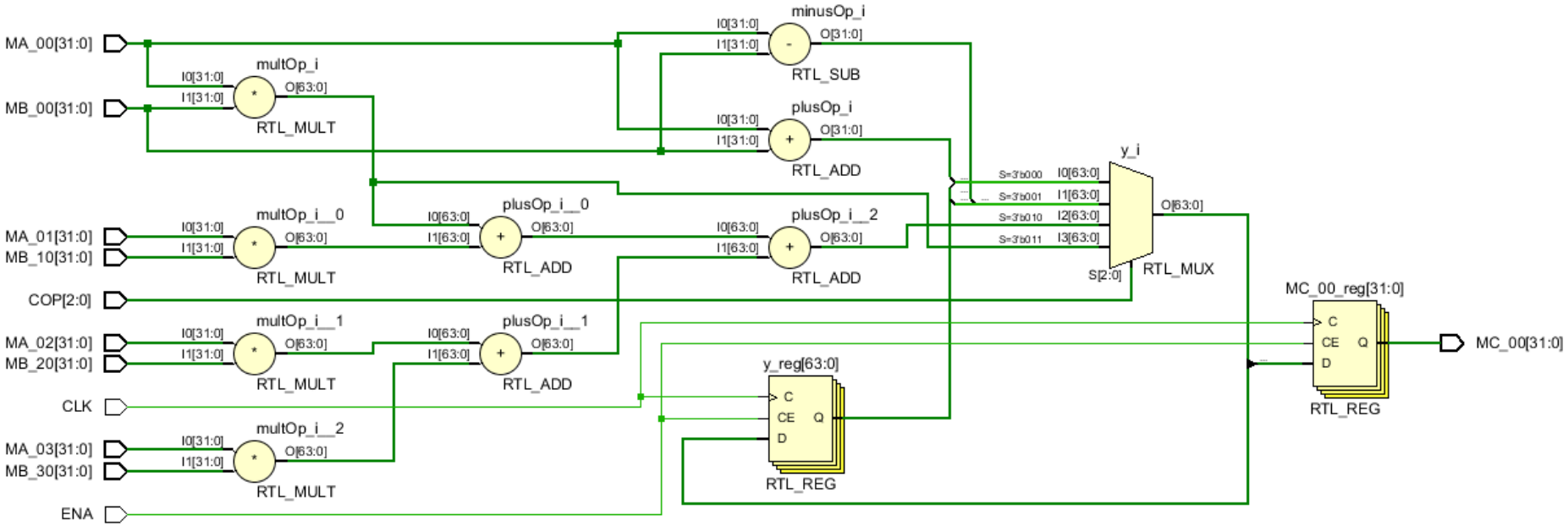

Рисунок 4.2 – Схема блоку Core

Загальна схема блоку Core представлена на рисунку 4.2 та має у своєму складі наступні елементи: мультіплексор, 4 помножувача, 4 суматора, 1 суматор, відповідальний за віднімання та регістри для збереження результату.

4.2 Блок Registers

Наступним розглянемо блок Registers, представлений на рисунку 4.3.

До його портів належать наступні: Adr_Reg, що отримує адресу регістру для зберігання елементу матриці, Data – саме значення для збереження, ENA – сигнал enable для підтвердження готовності елементу до роботи, CLK – синхросигнал та RST – сигнал для обнуління регістрів.

Вихідні порти блоку Registers призначені для передачі даних кожного з регістрів у блоці Reg_00, Reg_01,…Reg_15 відповідно.

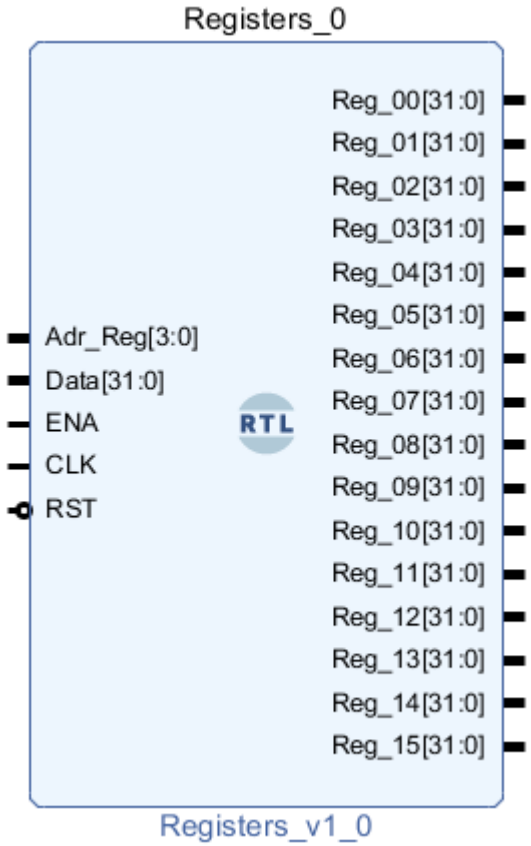

Рисунок 4.3 – Схема портів регістрів

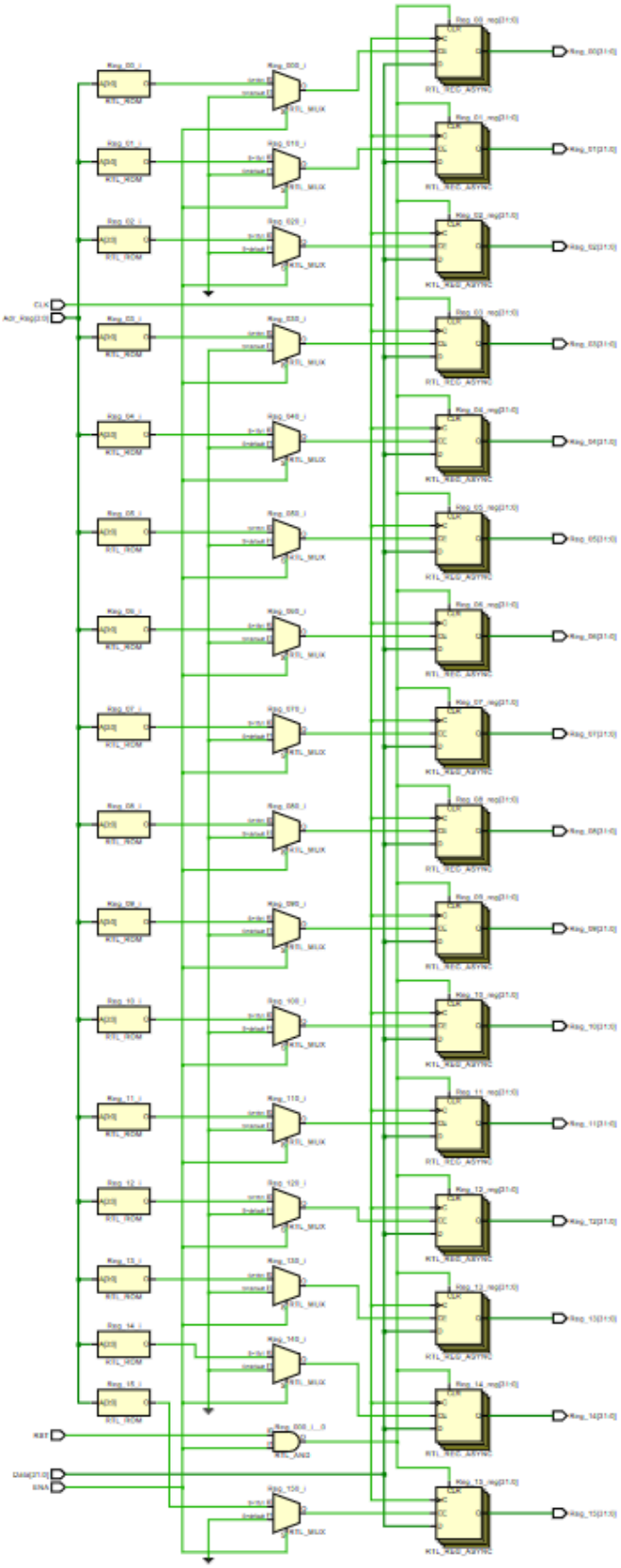

Рисунок 4.4 – Схема блоку Registers

Цей блок є практичною реалізацію модулів **Reg for Matrix A** та **Reg for Matrix B**, що обули описані у попередньому розділі.

Загальна схема блоку Registers представлена на рисунку 4.4. За сигналом Adr_Reg видається адреса регістру, у який буде записано данні елементу матриці з порту Data. Дані усіх регістрів передаються до блоків Core.

4.3 Блок Register MC

На наступному рисунку 4.5 наведено блок Register MC, що відповідальні за збереження елементів результуючої матриці C.

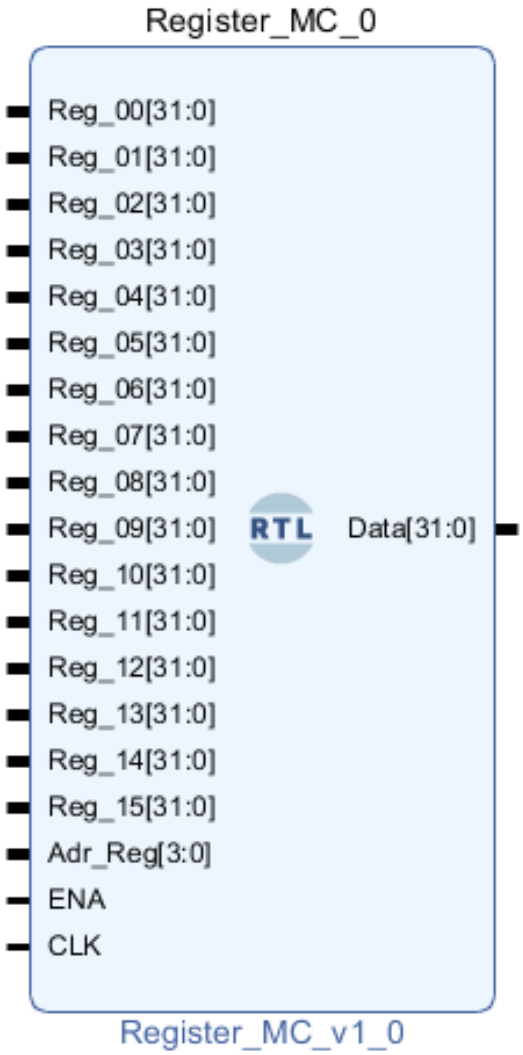

Рисунок 4.5 – Схема регістрів результуючої матриці C

Його вхідні порти Reg_00, Reg_01,…Reg_15 призначені для отримання даних результуючої матриці з блоків Core, Adr_Reg, що отримує адресу регістру для видачі елементу матриці, для його запису у оперативну пам’ять даних, ENA – сигнал enable для підтвердження готовності елементу до роботи, CLK – синхросигнал.

Вихідний порт Data передає дані з регістру за адресою Adr_Reg до оперативної пам’яті даних.

Загальна схема блоку Register MC представлена на рисунку 4.6 та складається з мультиплексора та регістрів.

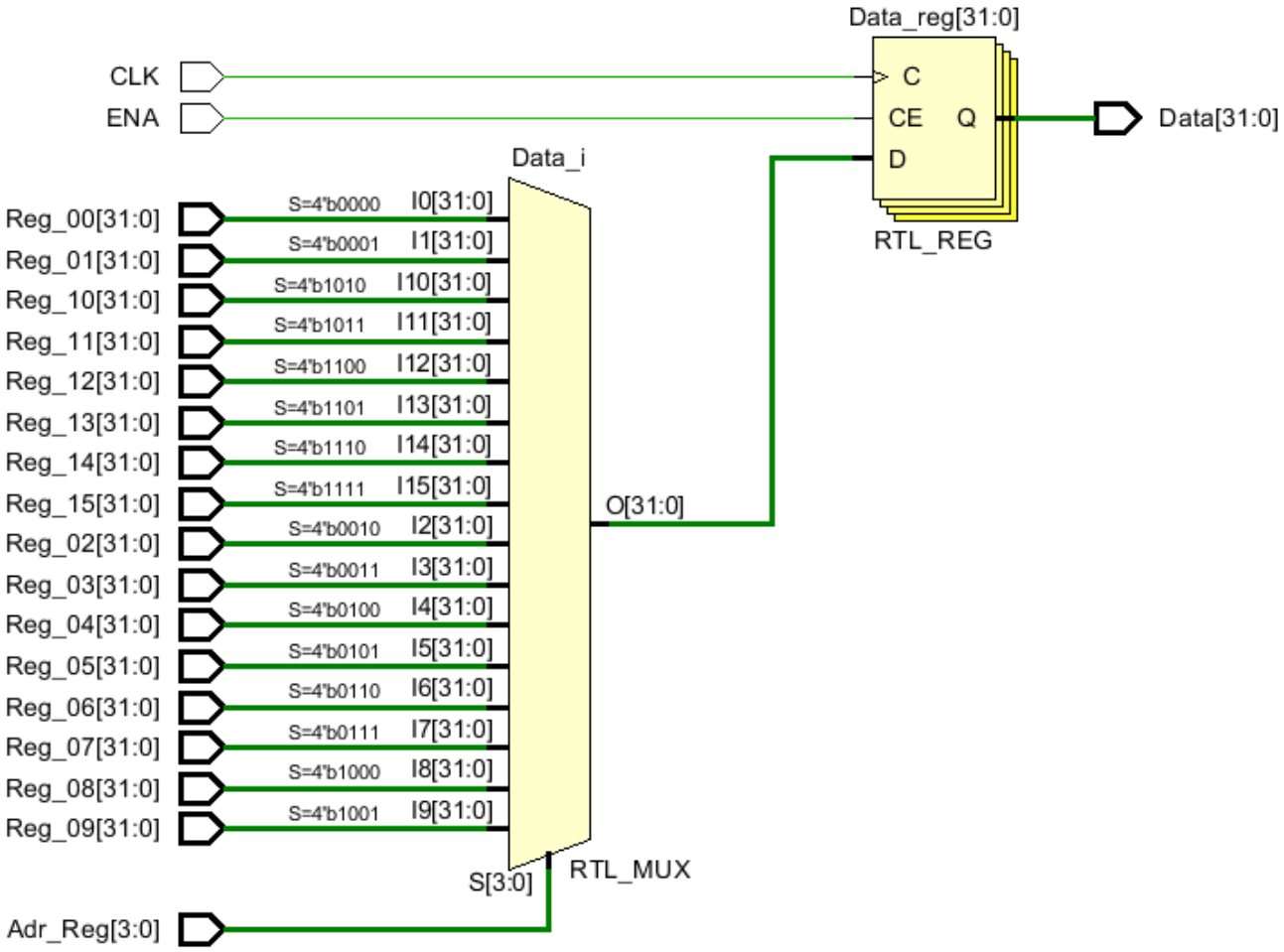

Рисунок 4.6 – Схема блоку Register MC

4.4 Блок Adress Generator for Matrix

Порти блоку Adress Generator for Matrix зображено на рисунку 4.7.

Цей блок відповідальний за розрахунок адреси регістру матриці A та регістру матриці B до якого будуть записуватися данні з оперативної пам’яті. Відповідно розраховує регістр матриці C для подальшої передачі даних з регістру матриці C до оперативної пам’яті даних

На вхід до цього блоку через порти передаються розміри матриці A, B та C (Size_MA, Size_MB та Size_MC відповідно), початок адресування регістрів матриць A та B AB_adr, а також для матриці C C_Adr.

Як і в попередніх блока, отримується через порти ENA – сигнал enable для підтвердження готовності елементу до роботи, CLK – синхросигнал та RST – сигнал для обнуління регістрів.

Загальна схема блоку наведена на рисунку 4.8.

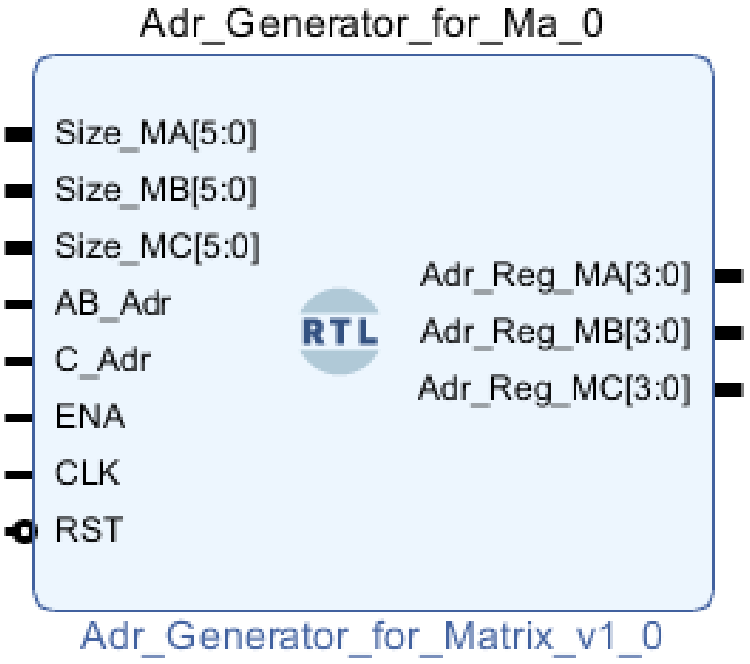

Рисунок 4.7 – Схема портів генератора адреси регістрів

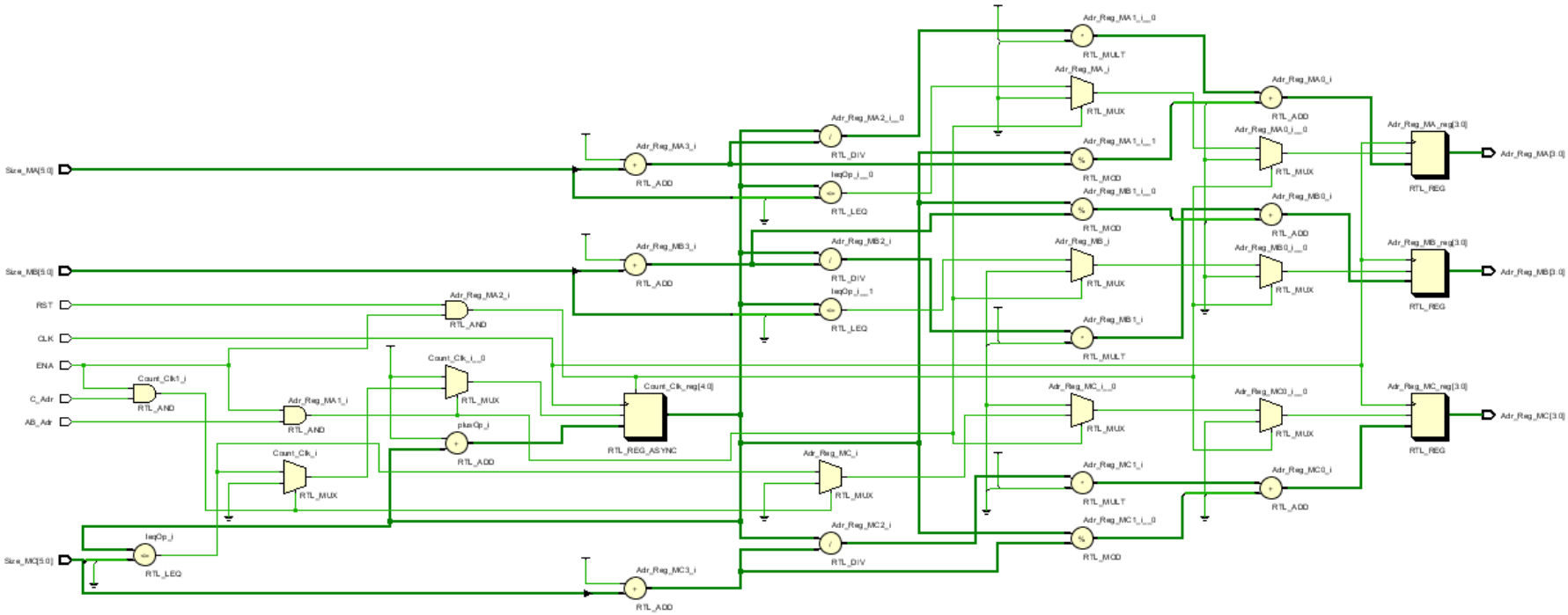

Рисунок 4.8 – Схема блоку Adress Generator for Matrix

4.5 Блок D_RAM

Наступним розглянемо блок D_RAM, двоканальна оперативна пам’ять даних, схема портів якого представлена на рисунку 4.9.

Вхідні порти цього блоку отримують наступні сигнали: ClkA та ClkB – це синхросигнали для роботи з каналами оперативної пам’яті, EnA та EnB – сигнал готовності до роботи каналу А та В оперативної пам’яті, WeA та WeB – запис при значені сигналу «1» та читання при значені «0», AddrA та AddrB – адреса опреативної пам’яті для каналу А та В відповідно, DIA та DIB – дані, що записуються в оперативну пам’ять для каналу А та В.

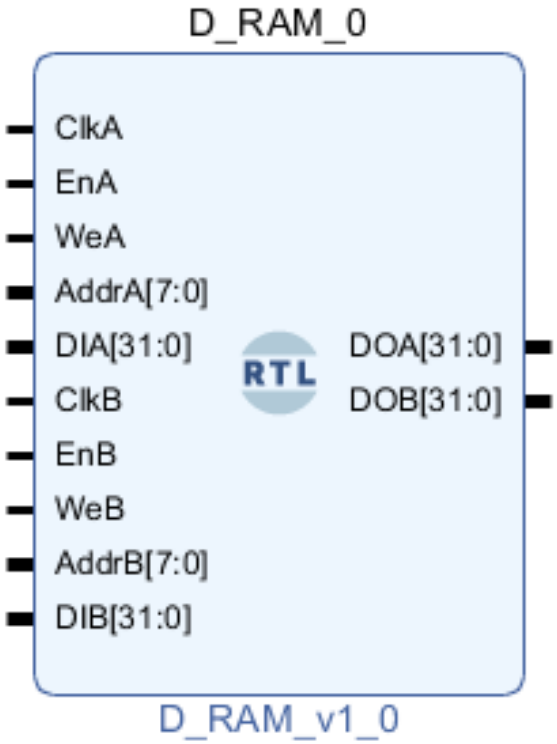

Рисунок 4.9 – Схема портів оперативної пам’яті даних

Канал А відповідальний за запис даних та зчитування елементів матриці А.

Канал В відповідальний за зчитування елементів матриці В.

Вихідні порти D_RAM: DOA та DOB призначені для зчитування даних з каналу А та каналу В відповідно.

Загальні схема блоку D_RAM представлено на рисунку 4.10.

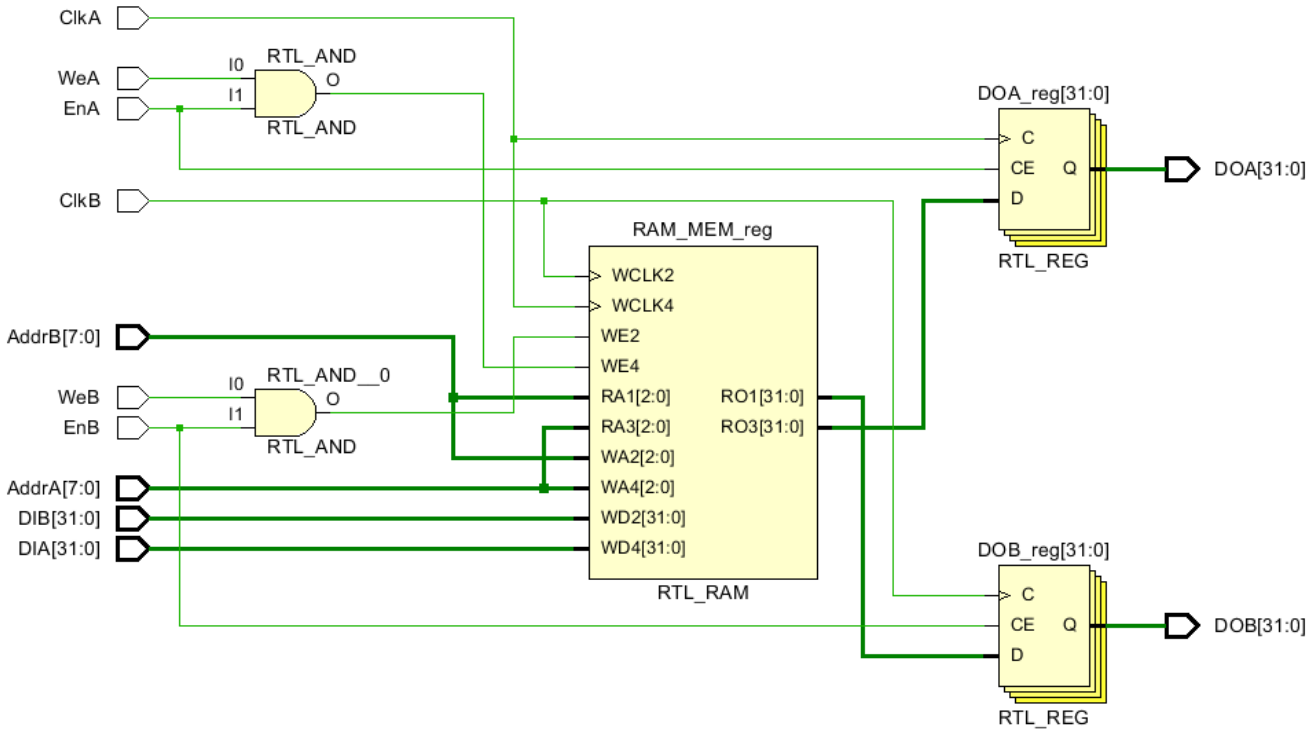

Рисунок 4.10 – Схема блоку D_RAM

4.6 Блок P_RAM

Блок P_RAM – це оперативна пам’ять програми, його порти зображено на рисунку 4.11.

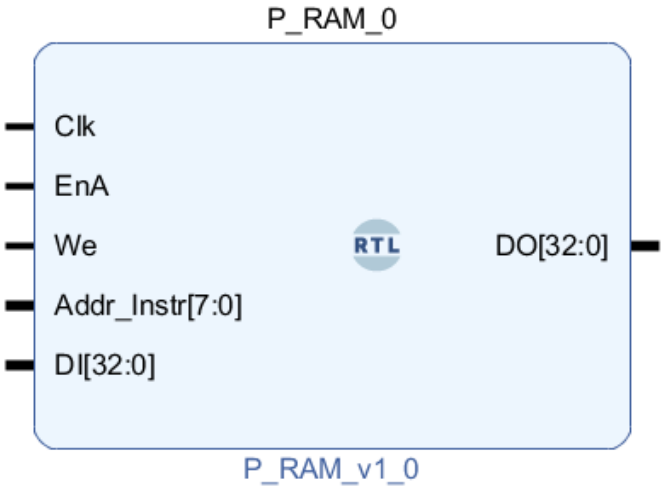

Рисунок 4.11 – Схема портів оперативна пам’ять програми

До вхідних портів цього блоку належать Clk – це синхросигнал, EnA– сигнал готовності до роботи, We – запис при значені сигналу «1» та читання при значені

«0», Addr_Instr – адреса опреативної пам’яті, DI – дані, що записуються в оперативну пам’ять.

Вихідні порти P_RAM: DO призначені для зчитування.

Загальна схема оперативної пам'яті програми зображено на рисунку 4.12.

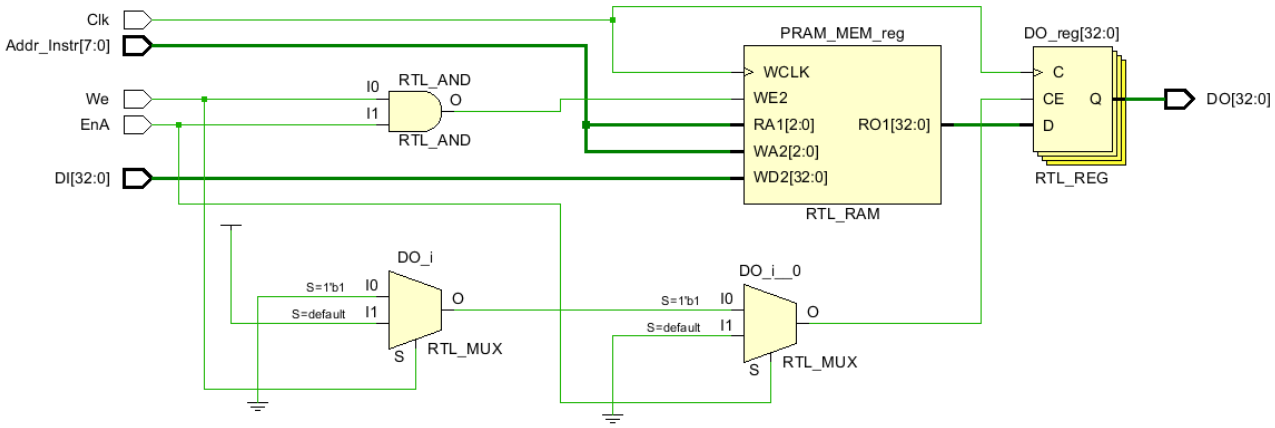

Рисунок 4.12 – Схема блоку P_RAM

4.7 Блок Port_A

Наступний блок Port_A відповідальний за зв’язок з зовнішнім пристроєм. Схема його портів наведено на рисунку 4.13.

Його вхідними сигналами є:

- Instraction_PortIn – отримує інструкції з зовнішнього пристрою;
- Data_PortIn – отримує дані з зовнішнього пристрою;
- Wr_Inst – запис інструкції;
- Wr_Data – запис даних;
- Rd_in – зчитування даних;
- Clk – синхросигнал;
- Start – старту виконання програми;
- Instraction_In – вхідні інструкції з оперативної пам’яті для подальшої видачі на зовнішній пристрій;

- Data_In – вхідні дані з оперативної пам’яті для подальшої видачі на зовнішній пристрій.

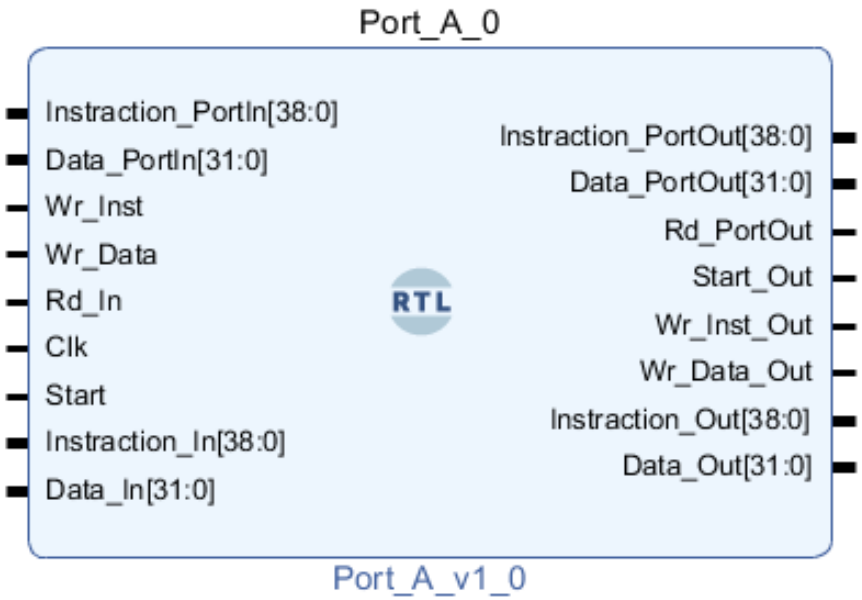

Рисунок 4.13 – Схема портів порту А

Його вихідними сигналами є:

- Instraction_PortOut – передає інструкції на зовнішній пристрій;
- Data_PortOut – передає дані на зовнішній пристрій;
- Wr_Inst_Out – передає сигнал запису інструкції на FSM;
- Wr_Data_Out – передає сигнал запису даних на FSM;
- Rd_PortOut – передає на зовнішній пристрій сигнал зчитування даних;
- Start_Out – передає на FSM сигнал старту роботи програми;
- Instraction_Out – передає інструкції з порту А в оперативну пам’ять програми;
- Data_Out – передає дані з порту А в оперативну пам’ять даних.

На рисунку 4.14 представлено загальну схему порту А.

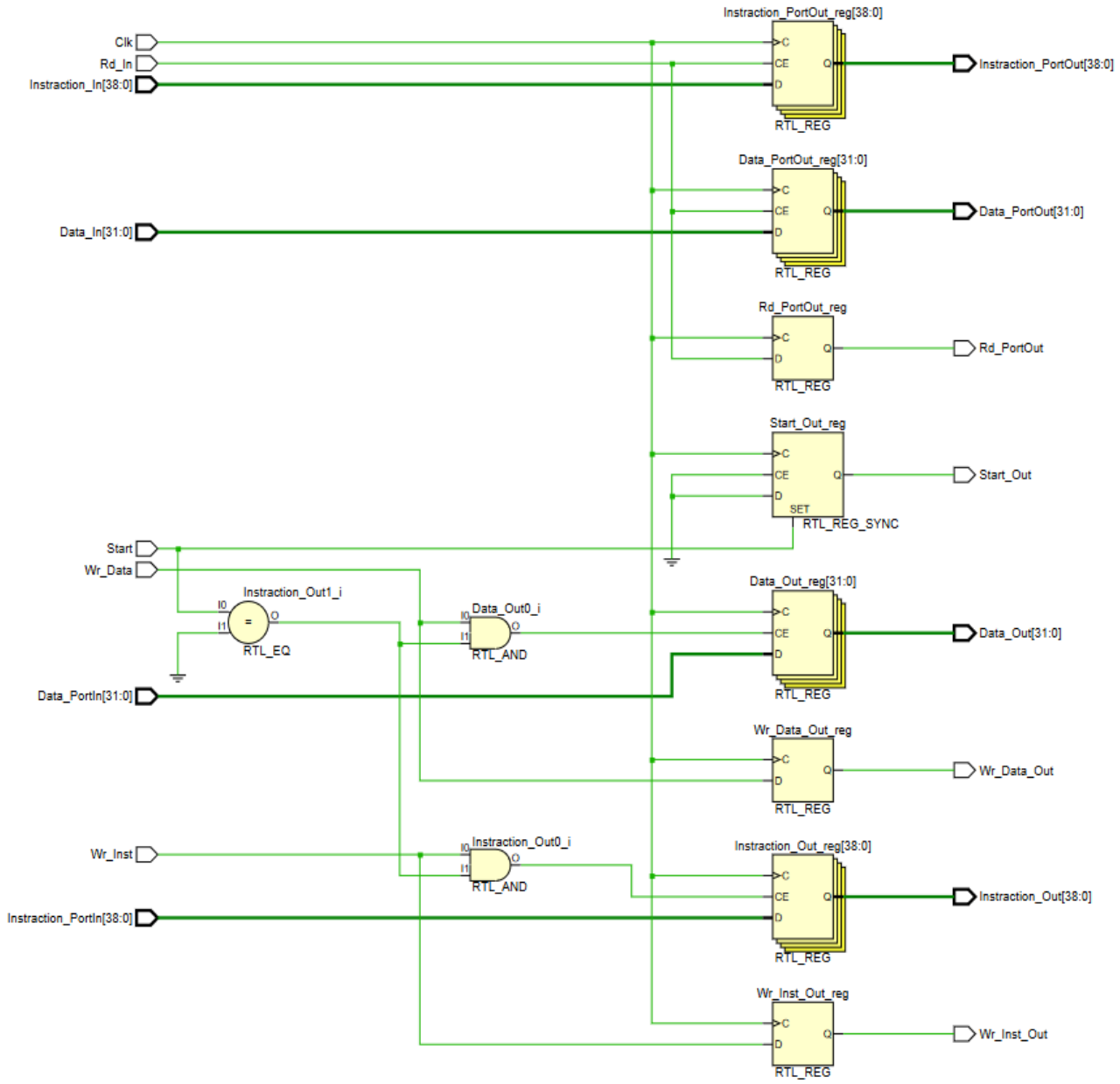

Рисунок 4.14 – Схема блоку Port_A

4.8 Блок FSM

Останній блок FSM є кінцевим автоматом і схема його портів зображена на рисунку 4.15.

Вхідні сигнали FSM:

- Instraction – інструкція, яку виконує пристрій;
- Start – старту роботи програми;

- Rst –
- Clk – синхросигнал;
- Wr_Inst – запис інструкції в оперативну пам’ять;
- Wr_Data –запис даних в оперативну пам’ять;

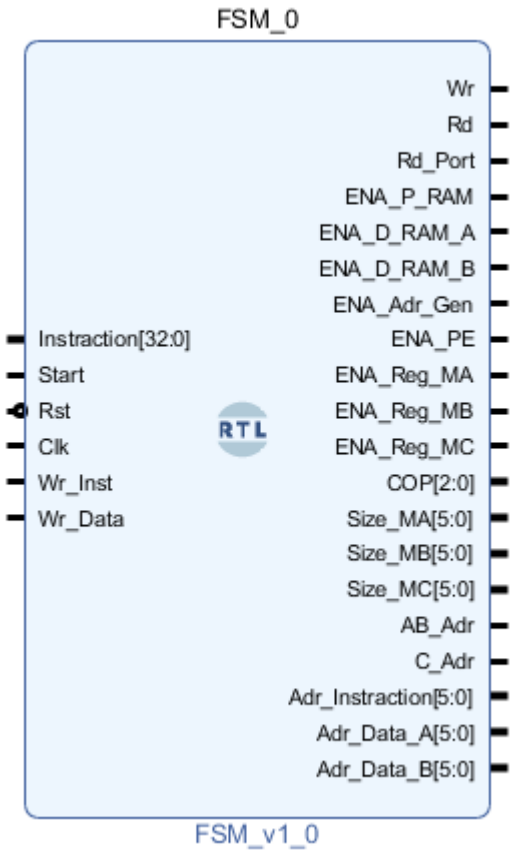

Рисунок 4.15 – Схема портів кінцевого автомату

Вихідні сигнали FSM:

- Wr – запис в оперативну пам’ять даних та програми;
- Rd_Port – початок зчитування з оперативну пам’яті даних та програми;
- ENA_P_RAM – готовність до роботи P_RAM;
- ENA_D_RAM_A – готовність до роботи D_RAM_A;
- ENA_D_RAM_B – готовність до роботи D_RAM_B;
- ENA_Adr_Gen – готовність до роботи Adr_Gen;
- ENA_PE – готовність до роботи PE;

- ENA_Reg_MA – готовність до роботи Reg_MA;
- ENA_Reg_MB – готовність до роботи Reg_MB;
- ENA_Reg_MC – готовність до роботи Reg_MC;
- COP – код операції, який йде до ядер процесору;
- Size_MA – розмір матриці A, що передаємо на генератор адрес регістрів матриць;
- Size_MB – розмір матриці B, що передаємо на генератор адрес регістрів матриць;
- Size_MC – розмір матриці C, що передаємо на генератор адрес регістрів матриць;
- AB_Adr – початок адресації регістрів матриць A та B, передаємо на генератор регістрів матриць;
- C_Adr – початок адресації регістрів матриці C, передаємо на генератор регістрів матриць;
- Adr_Instraction – адрес інструкції, що буде виконуватись наступною, предається на оперативну пам’ять програми;
- Adr_Data_A – адреса даних каналу A оперативної пам’яті даних, з якого будуть зчитуватись або записуватись дані;
- Adr_Data_B – адреса даних каналу B оперативної пам’яті даних, з якого будуть зчитуватись дані;

5 ДОСЛІДЖЕННЯ РОЗРОБЛЕНОГО БАГАТОЯДЕРНОГО ПРОЦЕСОРА

При виконанні синтезу схеми та підрахунку ресурсів, що були витрачені під час реалізації, за основу була взята ПЛІС Virtex 7 xc7vx690t ffg1157-2.

5.1 Тестування роботи ядер процесору

Як було зазначено раніше, багатоядерний процесор пристрою складається з 16 ядер для обробки максимальної матриці 4 на 4.

Для тестування коректності проведення розрахунків та їх швидкості були проведені наступні тести на множення матриць (рисунку 5.1 - рисунку 5.7), додавання матриць та віднімання матриць.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \cdot \begin{pmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \end{pmatrix} = \begin{pmatrix} 100 & 110 & 120 & 130 \\ 228 & 254 & 280 & 306 \\ 356 & 398 & 440 & 482 \\ 484 & 542 & 600 & 658 \end{pmatrix}$$

Рисунок 5.1 – Тестовий приклад 1

У першому тестовому прикладі на множення подавалися дві матриці A та B розміром 4 на 4.

Розрахунки та їх результати для тесту 1 можна побачити на рисунку 5.10 на 20 нс.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} 40 \\ 96 \\ 152 \\ 208 \end{pmatrix}$$

Рисунок 5.2 – Тестовий приклад 2

У другому тестовому прикладі проводилось множення матриці A розміром 4 на 4 та вектор-стовпець B.

Розрахунки та їх результати для тесту 2 можна побачити на рисунку 5.10 на 140 нс.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} 40 \end{pmatrix}$$

Рисунок 5.3 – Тестовий приклад 3

У третьому тестовому прикладі розглядалось множення вектору-рядку А на вектор-стовпець В.

Розрахунки та їх результати для тесту 3 можна побачити на рисунку 5.10 на 180 нс.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \end{pmatrix} = \begin{pmatrix} 100 & 110 & 120 & 130 \end{pmatrix}$$

Рисунок 5.4 – Тестовий приклад м 4

Четвертий тест полягав у множенні вектора-рядка А на матрицю В розміром 4 на 4.

Розрахунки та їх результати для тесту 4 можна побачити на рисунку 5.10 на 220 нс.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & 10 & 11 \end{pmatrix} \cdot \begin{pmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 44 & 50 & 56 \\ 116 & 134 & 152 \\ 188 & 218 & 248 \end{pmatrix}$$

Рисунок 5.5 – Тестовий приклад 5

У п’ятому тесті процесором виконувалось множення матриць А та В розміром 3 на 3 кожна.

Розрахунки та їх результати для тесту 5 можна побачити на рисунку 5.10 на 260 нс.

Наступні тести 6 та 7 проводились для перевірки коректності та швидкості роботи ядер процесору при проведенні матричних операцій додавання та віднімання відповідно.

Тестові приклади надано на рисунку 5.8 та рисунку 5.9.

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} + \begin{pmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \end{pmatrix} = \begin{pmatrix} 3 & 5 & 7 & 9 \\ 11 & 13 & 15 & 17 \\ 19 & 21 & 23 & 25 \\ 27 & 29 & 31 & 33 \end{pmatrix}$$

Рисунок 5.6 – Тестовий приклад 6

Для тестового прикладу 6 проводилась матрична операція додавання двох матриць A та B розміром 4 на 4.

Розрахунки та їх результати для тесту 6 можна побачити на рисунку 5.10 на 60 нс.

$$\mathbf{A} - \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} - \begin{pmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix}$$

Рисунок 5.7 – Тестовий приклад 7

Останній тест на матричні операції – обчислення різниці двох матриць A та B розмірами 4 на 4.

Розрахунки та їх результати для тесту 7 можна побачити на рисунку 5.10 на 100 нс. Згідно даних на рисунку 5.10 процесор виконує усі розрахункові тести 1-5 за 1 такт. Результат роботи процесору відповідає очікуваним результатам тестів.

Нижче на рисунках 5.8-5.10 наведено процес виконання матричних операцій розробленим пристроєм.

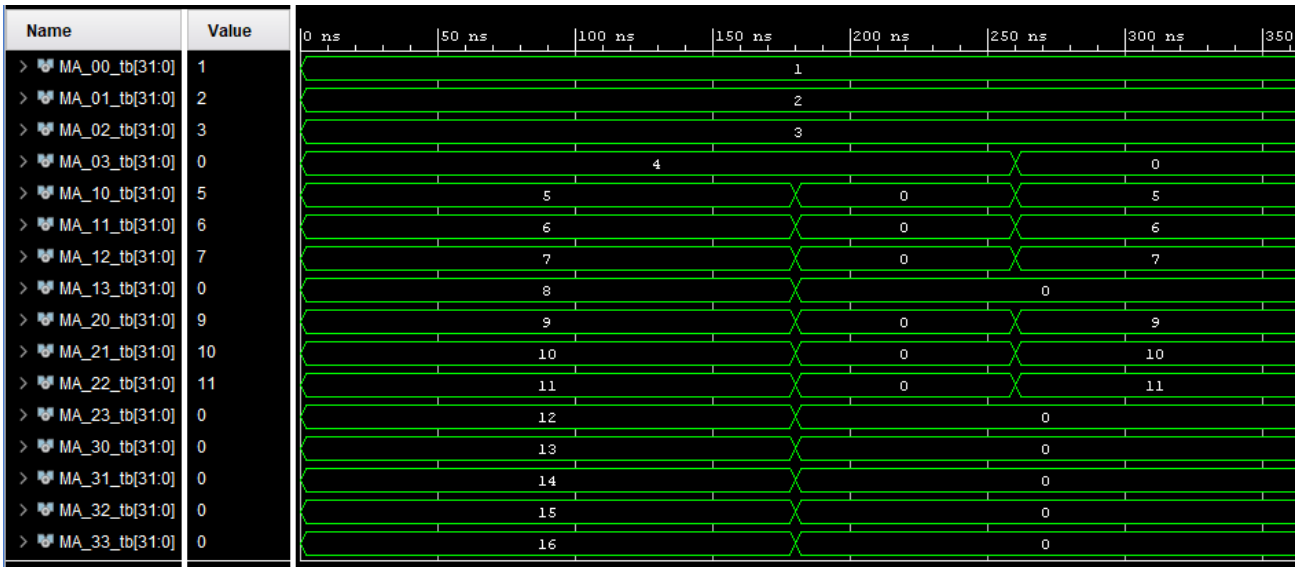

Рисунок 5.8 – Дані матриць A, що передавались у багатоядерний процесор

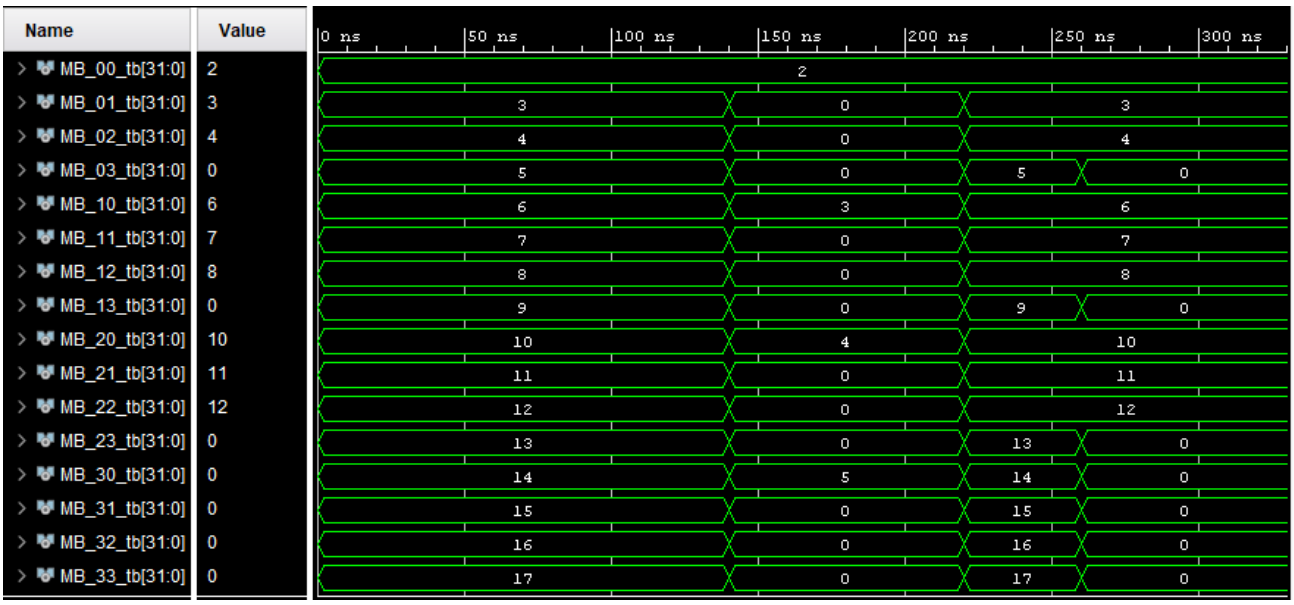

Рисунок 5.9 – Дані матриць B, що передавались у багатоядерний процесор

Також на рисунку 5.10 можна побачити передачу коду операції ядрам процесора COP (2:0). Де значення 0 відповідає операції додавання, значення 1 відповідає операції віднімання матриць та значення 2 – множенню матриць.

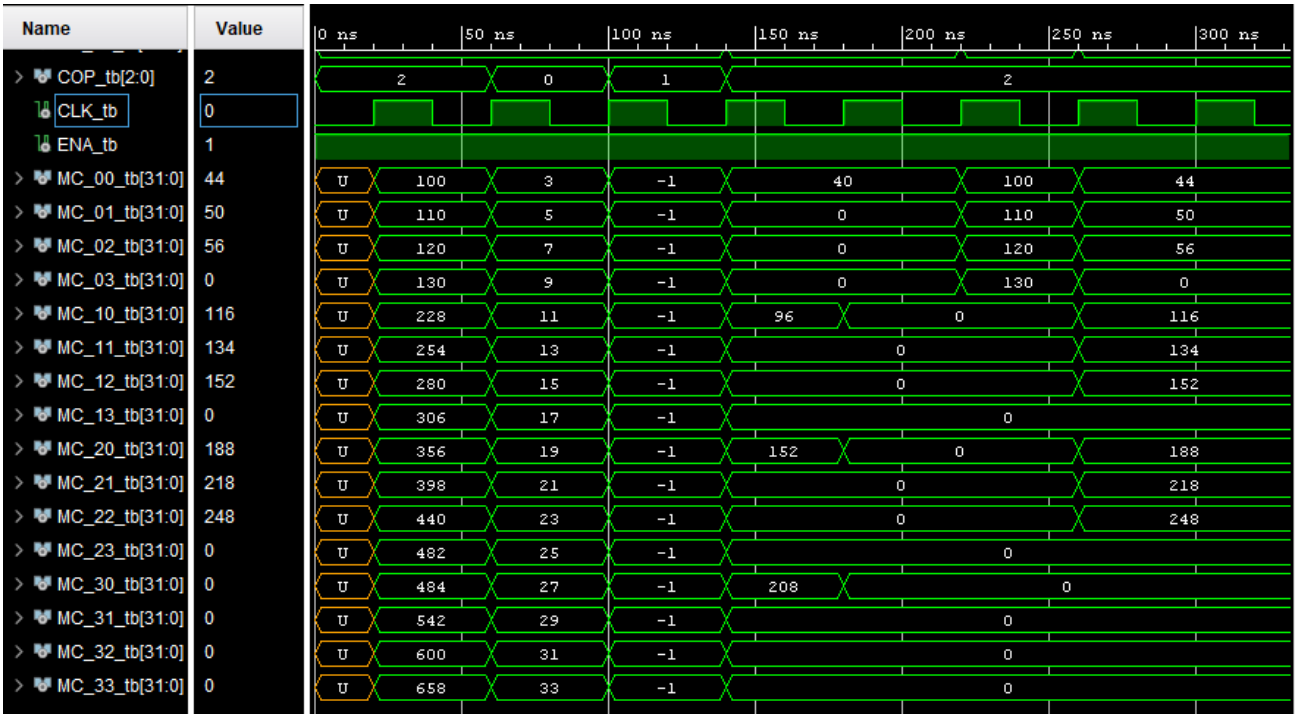

Рисунок 5.10 – Результати роботи ядер процесора

Також були проведені дослідження завантаження блоку проведеними тестами. На рисунку 5.11 наведені числові значення завантаженості ресурсів ПЛІС для ядра процесору.

Resource	Estimation	Available	Utilization %
LUT	244	303600	0.08
FF	32	607200	0.01
DSP	12	2800	0.43
IO	292	600	48.67
BUFG	1	32	3.13

Рисунок 5.11 – Завантаженість ресурсів ПЛІС при тестуванні

Згідно отриманих даних, найбільше навантаження припадає на порти вводу-виводу даних. Оскільки цей блок не має зовнішніх портів, то порти вводу-виводу не мають критичного значення.

На рисунку можна побачити серед використаних ресурсів ПЛІС певну кількість елементів для цифрової обробки сигналів через виконання блоком матричних розрахунків.

Для більш наглядної демонстрації дані використання ресурсів ПЛІС також представлено у вигляді стовпчикової діаграми у відсотках від загальної кількості елементів на рисунку 5.12.

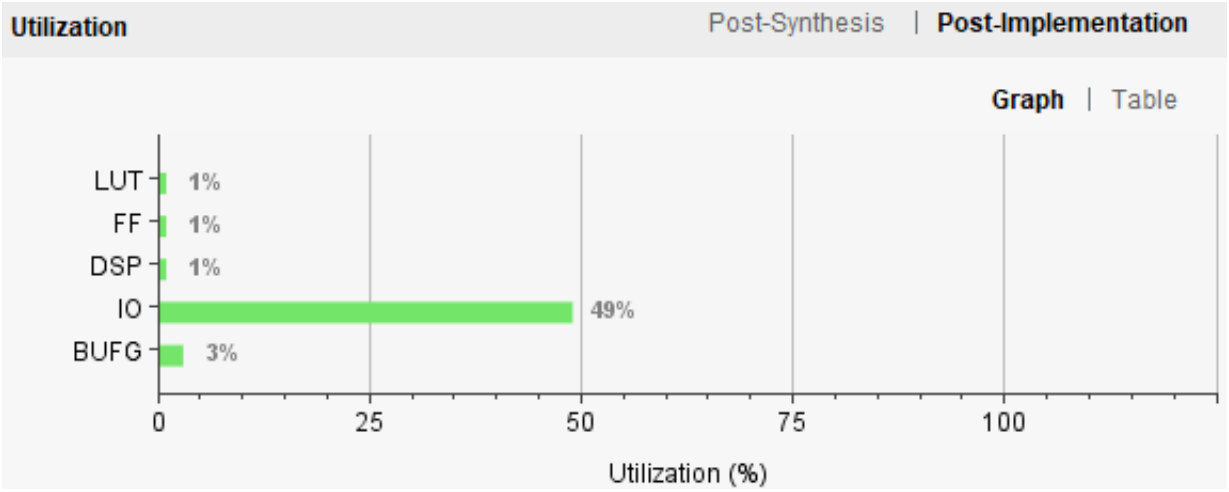

Рисунок 5.12 – Завантаженість ресурсів ПЛІС при тестуванні у відсотковому еквіваленті

5.2 Тестування роботи блоку Registers

Блок Registers відповідальний за видачу матриці A та матриці B багатоядерному процесору для проведення подальших матричних розрахунків.

У тестовому прикладі блоку подавалася матриця з 16 елементів наведена на рисунку 5.13.

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 30 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

Рисунок 5.13 – Матриця A для тестування роботи блоку Registers

Кожний такт на блок подавався елемент матриці A по рядках, також подавалася адреса регістру для кожного елементу. Для запису у регістри 16 елементів матриці A знадобилося 16 тактів.

Необхідно також відмітити, що перед початком завантаження потрібно обнулити усі регістри сигналом RST. Це необхідно для того, якщо буде заповнюватись матриця з меншою кількістю елементів – відсутні елементи будуть заповнені нулями.

Процес тестування блоку зображено на рисунку 5.14.

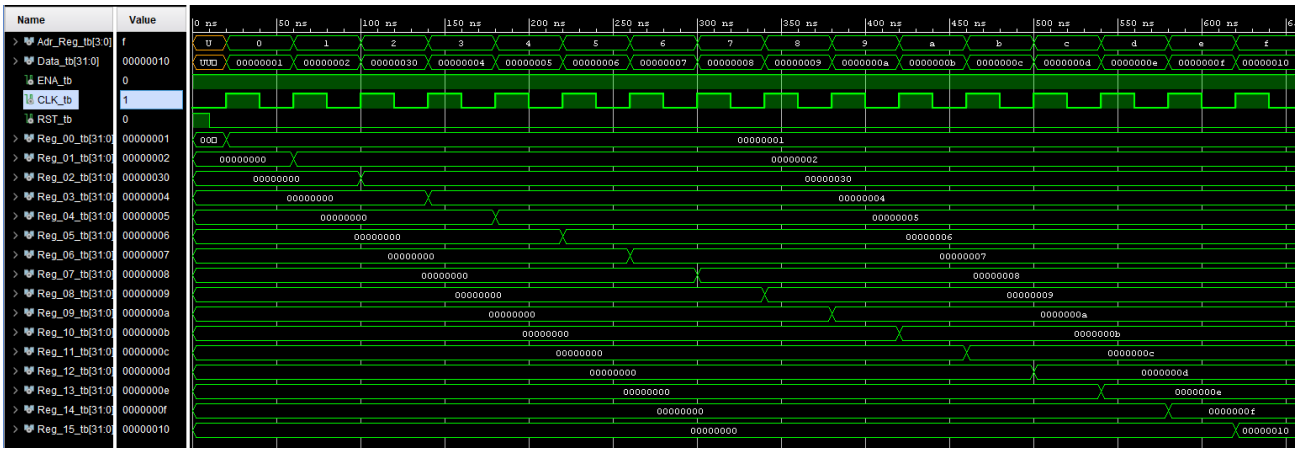

Рисунок 5.14 – Результати тестування блоку регістрів

Також були проведені дослідження завантаження блоку проведеними тестами. На рисунку 5.15 наведені числові значення завантаженості ресурсів ПЛІС для блоку регістрів.

Resource	Utilization	Available	Utilization %
LUT	17	433200	0.01
FF	512	866400	0.06
IO	551	600	91.83
BUFG	1	32	3.13

Рисунок 5.15 – Завантаженість ресурсів ПЛІС при тестуванні

Згідно отриманих даних, найбільше навантаження припадає на порти вводу-виводу даних. Оскільки цей блок не має зовнішніх портів, то порти вводу-виводу не мають критичного значення.

Для більш наглядної демонстрації дані використання ресурсів ПЛІС також представлено у вигляді стовпчикової діаграми у відсотках від загальної кількості елементів на рисунку 5.16.

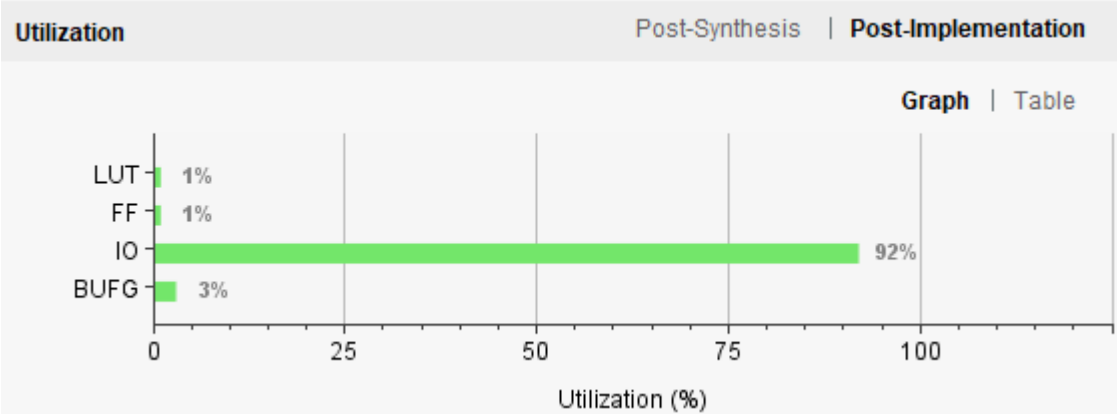

Рисунок 5.16 – Завантаженість ресурсів ПЛІС при тестуванні у відсотковому еквіваленті

5.3 Тестування блоку Port_A

Блок Port_A приймає з зовнішнього пристрою інструкцію для виконання матричних операцій та данні матриць для розрахунків, що далі передає до оперативної пам’яті програми та оперативної пам’яті даних відповідно.

Сигнали, які передають інструкції з зовнішнього пристрою – це Instraction_PortIn. По сигналу Wr_Inst порт передає інструкції Instraction_Out далі до оперативної пам’яті програми, з одночасною передачею сигналу-попередження Wr_Inst_Out до FSM. Кожного такту передається нова інструкція.

Сигнали, які передають дані з зовнішнього пристрою – це Data_PortIn. По сигналу Wr_Data порт передає дані Data_Out далі до оперативної пам’яті даних, з одночасною передачею сигналу-попередження Wr_Data_Out до FSM. Кожного такту передається нові дані.

Сигнал Start, що передає зовнішній пристрій, передається далі через порт Start_Out до FSM та визначає початок роботи програми.

По сигналу Rd_in, який передається з FSM на Port_A, відбувається зчитування даних. Port_A передає цей сигнал Rd_PortOut до зовнішнього пристрою як попередження зовнішньому пристрою про наступну передачу даних за допомогою сигналу Data_PortOut для даних та Instraction_PortOut для інструкцій.

Data_In та Instraction_In безпосередні дані, що передаються на порт A з оперативної пам’яті даних та оперативної пам’яті програми.

Весь процес роботи блоку зображено в режимі тестування на рисунку 5.17.

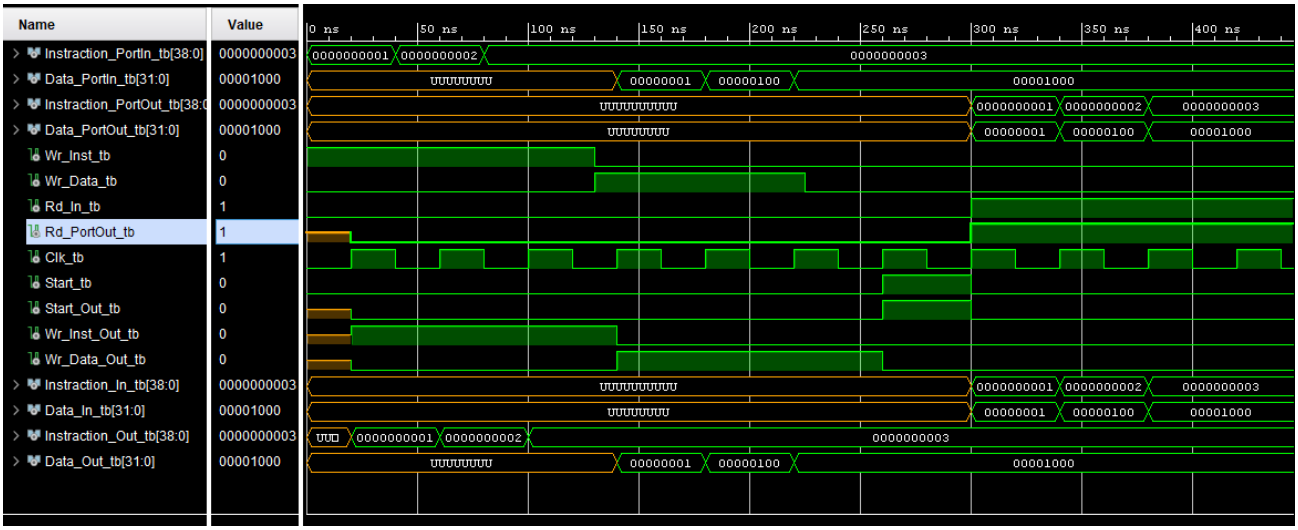

Рисунок 5.17 – Результати тестування блоку порт A

Також були проведені дослідження завантаження блоку проведеними тестами. На рисунку 5.18 наведені числові значення завантаженості ресурсів ПЛІС для порту A.

Resource	Estimation	Available	Utilization %
LUT	2	433200	0.01
FF	146	866400	0.02
IO	293	600	48.83
BUFG	1	32	3.13

Рисунок 5.18 – Завантаженість ресурсів ПЛІС при тестуванні

Згідно отриманих даних, найбільше навантаження так само припадає на порти вводу-виводу даних. Цей блок має зовнішні порт, але отримане навантаження не є критичним для роботи блоку.

Для більш наглядної демонстрації дані використання ресурсів ПЛІС також представлено у вигляді стовпчикової діаграми у відсотках від загальної кількості елементів на рисунку 5.19.

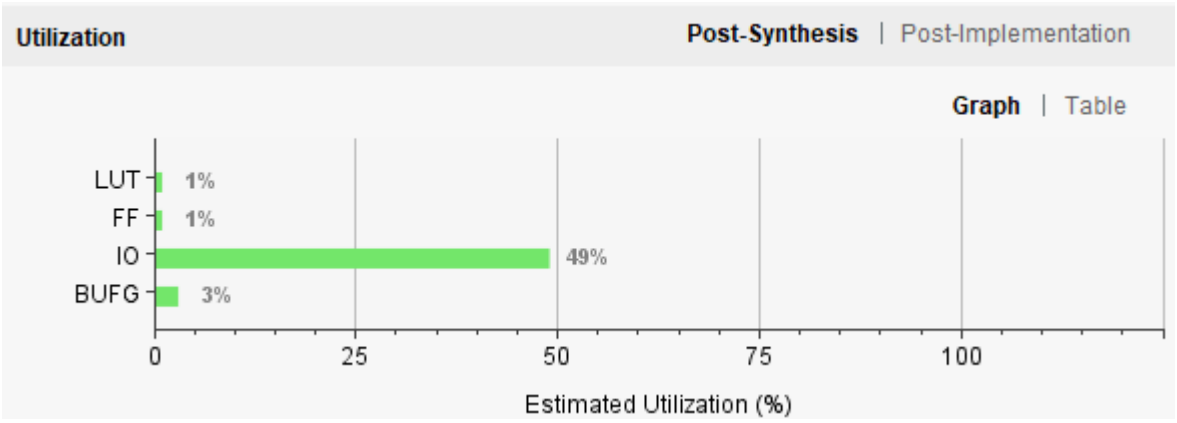

Рисунок 5.19 – Завантаженість ресурсів ПЛІС при тестуванні у відсотковому еквіваленті

5.4 Тестування блоку P_RAM

Блок P_RAM по сигналу We, рівному «1», записує дані, подані сигналом DI в оперативну пам’ять програми за адресою, яка передається сигналом Addr_Instr. По сигналу We, рівному «0» зчитуються дані за адресою, що передається сигналом Addr_Instr. Ці дані – це сигнал DO.

Сигнал EnA – свідчить про готовність блоку до роботи.

Весь процес роботи блоку зображено в режимі тестування на рисунку 5.20.

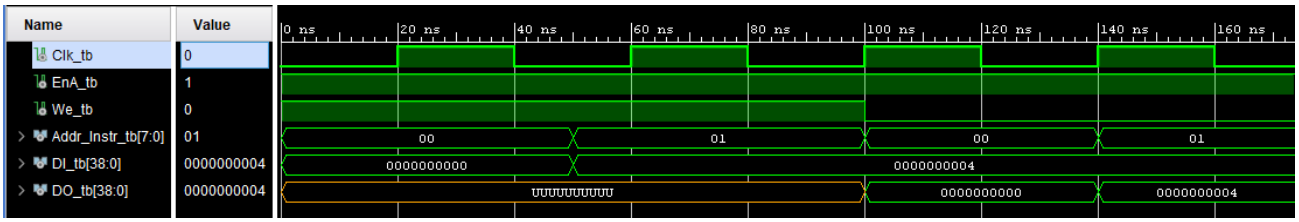

Рисунок 5.20 – Результати тестування блоку оперативної пам’яті програми

Також були проведені дослідження завантаження блоку проведеними тестами. На рисунку 5.21 наведені числові значення завантаженості ресурсів ПЛІС для блоку оперативної пам’яті програми.

Resource	Estimation	Available	Utilization %
LUT	41	433200	0.01
LUTRAM	39	174200	0.02
FF	39	866400	0.01
IO	84	600	14.00
BUFG	1	32	3.13

Рисунок 5.21 – Завантаженість ресурсів ПЛІС при тестуванні

Згідно отриманих даних, найбільше навантаження так само припадає на порти вводу-виводу даних.

Оскільки це елемент пам’яті, на рисунку можна побачити більшу кількість використаних LUTRAM.

Для більш наглядної демонстрації дані використання ресурсів ПЛІС також представлено у вигляді стовпчикової діаграми у відсотках від загальної кількості елементів на рисунку 5.22.

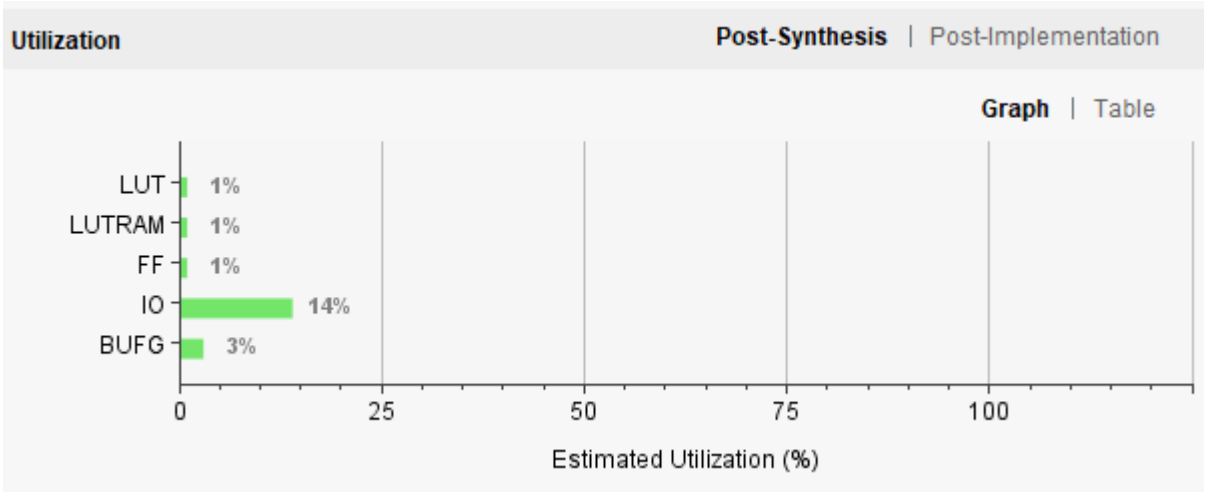

Рисунок 5.22 – Завантаженість ресурсів ПЛІС при тестуванні у відсотковому еквіваленті

5.5 Тестування блоку D_RAM

Блок D_RAM по сигналу WeA рівному «1» для каналу A та WeB рівному «1» для каналу B, за умови отримання сигналів EnA та EnB (сигналів готовності до роботи каналу A та B оперативної пам’яті), записує дані, подані сигналами DIA

та DIB в оперативну пам’ять даних за адресою, яка передається сигналами AddrA та AddrB для каналу A та каналу B відповідною.

По сигналу WeA рівному «0» для каналу A та WeB рівному «0» для каналу B, за умови отримання сигналів EnA та EnB (сигналів готовності до роботи каналу A та B оперативної пам’яті), зчитує дані з оперативної пам’яті та виводить їх сигналами DOA та DOB. Адреса зчитуваних даних задається сигналами AddrA та AddrB для каналу A та каналу B відповідною.

Весь процес роботи блоку зображено в режимі тестування на рисунку 5.23.

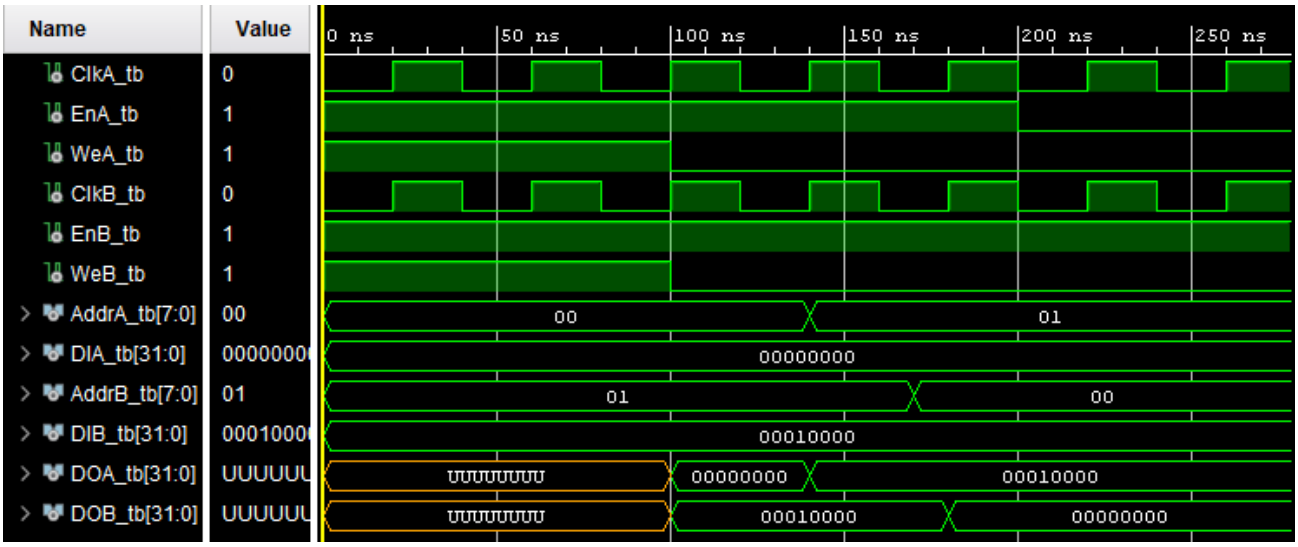

Рисунок 5.23 – Результати тестування блоку оперативної пам’яті даних

Також були проведені дослідження завантаження блоку проведеними тестами. На рисунку 5.24 наведені числові значення завантаженості ресурсів ПЛІС для блоку оперативної пам’яті даних.

Resource	Estimation	Available	Utilization %
BRAM	1	1470	0.07
IO	140	600	23.33
BUFG	2	32	6.25

Рисунок 5.24 – Завантаженість ресурсів ПЛІС при тестуванні

Згідно отриманих даних, найбільше навантаження так само припадає на порти вводу-виводу даних.

Для моделювання цього блоку використовувались лише ресурси вводу-виводу, буфер та блоки пам’яті, оскільки цей блок відповідальний за двоканальну оперативну пам’ять.

Для більш наглядної демонстрації дані використання ресурсів ПЛІС також представлено у вигляді стовпчикової діаграми у відсотках від загальної кількості елементів на рисунку 5.25.

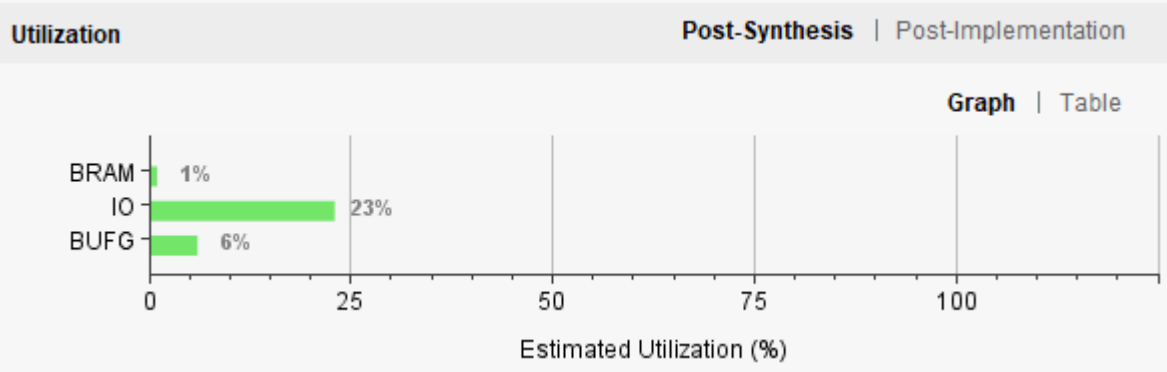

Рисунок 5.25 – Завантаженість ресурсів ПЛІС при тестуванні у відсотковому еквіваленті

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було розроблено багатоядерний процесор на основі ПЛІС і виконано тестування щодо його можливостей виконання матричних операцій.

В кваліфікаційній роботі здійснена розробка апаратної частини та алгоритмів, створенні програми функціонування та відлагодження багатоядерного процесора. Для дослідження його можливостей щодо виконання матричних операцій було розроблено та реалізовано відповідні тести та проведено аналіз отриманих результатів.

Для роботи цього процесору були розроблені наступні модулі: порт, з якого подаються дані та інструкції, двоканальну оперативну пам’ять даних, оперативну пам’ять програми, регістри, в яких зберігаються елементи матриць над якими проводяться матричні розрахунки, регістри зберігання елементів результуючої матриці, генератор адрес регістрів, що зберігають елементи матриць та багатоядерний процесор, що складається з 16 ядер та проводить усі матричні розрахунки.

Розробка блоків відповідних модулів та процес симуляції їх роботи проводились з використанням САПР Vivado 18.2, опис створених функціональних блоків реалізовано на мові VHDL. Побудовані та представлені у роботі схеми відповідних блоків отримані за допомогою синтезу схем цього ж інструменту.

У ході роботи були проведені дослідження роботи розроблених блоків, де було продемонстровано навантаження на ресурси ПЛІС при синтезі кожного з побудованих блоків. Також була проведена симуляція роботи кожного з блоків на тестових прикладах, що полягали у роботі з матрицями:

- отримання матриці та інструкцій з зовнішнього пристрою на порт;
- запису інструкції у оперативну пам’ять програми;
- запису даних у оперативну пам’ять даних;

- завантаження елементів матриці у регістри;

- проведення матричних обчислень (додавання, віднімання та множення);

- вивід з порту даних;

- зчитування даних з оперативної пам’яті даних та зчитування інструкцій з оперативної пам’яті програми.

Запропонований підхід до множення матриць можна легко масштабувати до великих розмірів матриць за рахунок збільшення кількості ядер процесору, що є актуальним рішенням, оскільки множення матриць має широке використання у низці сфер програмування, воно широко застосовується в різних чисельних методах.

У комп'ютерній графіці матричне множення широко застосовується для трансформації та маніпуляції графічними об'єктами, обробці зображень та відео. Наприклад, застосування матриці трансформації до координат точок дозволяє здійснювати переміщення, масштабування та поворот об'єктів. При розробці комп'ютерних ігор матричне множення застосовується для різних обчислень, таких як перетворення координат, обробка фізики та інші аспекти.

В машинному навчанні дані матричного множення використовуються при навчанні моделей і виконанні операцій над операціями. Нейронні мережі, наприклад, часто застосовують матричні операції для обчислення вагів та активації.

В області аналізу даних і статистики матричне множення може використовуватися для обробки даних, розрахунку кореляцій і виконання інших математичних операцій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Демидович В.М, Шаповалов В. О. (2023) “Можливості розробки багатоядерного процесора з використанням ПЛІС”. Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті. с. 27.
2. Мельник А. О. (2008) “Архітектура комп'ютера: підруч. для студ. вищ. навч. закл.”, Луцьк, Волинська обласна друкарня, 470с.
3. Flynn, M. J. (1966). Very high-speed computing systems. Proceedings of the IEEE, 54(12), 1901-1909. doi:10.1109/PROC.1966.5273.
4. Flynn, M. J. (1972). Some computer organizations and their effectiveness. IEEE transactions on computers, 100(9), 948-960. doi:10.1109/TC.1972.5009071.
5. Verilog Hardware Description Language Reference Manual, http://ecad. tu-sofia.bg/soc/data/verilog/verilog.pdf
6. Yiannacouras P., Rose J., and Steffan J. G. (2005) “The Microarchitecture of FPGABased Soft Processors”, International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), September, San Francisco, CA.
7. Coyne J., Cyganski D., Duckworth R. J.: FPGA-Based Co-processor for Singular Value Array Reconciliation Tomography. In Kenneth L. Pocek, Duncan A. Buell, editors, 16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008, 14–15 April 2008, Stanford, Palo Alto, California, USA. pp. 163–172, IEEE Computer Society, 2008.
8. Stanley Y. C. Li, Gap C. K. Cheuk, Kin-Hong Lee, Philip Heng Wai Leong,: FPGAbased SIMD Processor. In 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003), 8–11 April 2003, Napa, CA, Proceedings. pp. 267–268, IEEE Computer Society, 2003.
9. Arias−García, Pezzuol Jacobi, CH Llanos, and Ayala-Rincón. (2011) “A suitable FPGA implementation of floating-point matrix inversion based on Gauss-Jordan

elimination.” VII Southern Conference on Programmable Logic, Cordoba, Argentina, 13−15 April, pp. 263-268.

10. WWang, J Szefer, and R Niederhagen. (2016) “Solving large systems of linear equations over GF (2) on FPGAs.” International Conference on ReConFigurable Computing and FPGAs, Cancun, Mexico, 30 November − 2 December, pp. 1-7.
11. Matt Ruan. (2017) “Scalable Floating-point Matrix Inversion Design using Vivado High Level Synthesis.” Xilinx Application Note XAPP 1317, Xilinx Inc.,Version 1.0, pp. 1-20.
12. L Jianwen, and JC Chuen. (2007) “A System-on-Chip Dynamically Reconfigurable FPGA Platform for Matrix Inversion.” International Symposium on Integrated Circuits, Singapore, 26 − 28 September, pp. 465-468.
13. Ahmad Khayyat and Naraig Manjikian. (2011) “Controller design for matrix multiplication on FPGAs.” 24th Canadian Conference on Electrical and Computer Engineering, Niagara Falls, ON, Canada, 8 − 11 May, pp 1327-1332.
14. Rasha El-Atfy, Mohamed A. Dessouky, and Hassan El-Ghitani. (2007) “Accelerating Matrix Multiplication on FPGAs.” 2nd International Design and Test workshop, Cairo, Egypt, 16 − 18 December, pp 203-204.
15. Yiyu Tan, and Toshiyuki Imamura. (2017) “An Energy-Efficient FPGA-based Matrix Multiplier.” 24th IEEE International conference on Electronics, Circuits and Systems, Batumi, Georgia, 5 − 8 December, pp 514-517.
16. Oudjida, A. K., Titri, S., & Hamarlain, M. (2000, February). Synthesizing full-systolic arrays for matrix product on Xilinx's XC4000 (E, EX) FPGAs. In Field-Programmable Gate Arrays, International ACM Symposium on (pp. 222-222). IEEE Computer Society.
17. Scrofano, R., Choi, S., & Prasanna, V. K. (2002, December). Energy efficiency of FPGAs and programmable processors for matrix multiplication. In 2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings. (pp. 422-425). IEEE.

18. Amira, A., Bouridane, A., Milligan, P., & Sage, P. (2000, August). A high throughput FPGA implementation of a bit-level matrix-matrix product. In Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems (Cat. No. CH37144) (Vol. 1, pp. 396-399). IEEE.

19. Amira, A., & Bensaali, F. (2002, October). An FPGA based parameterizable system for matrix product implementation. In IEEE workshop on signal processing systems (pp. 75-79). IEEE.

20. Bensaali, F., Amira, A., & Bouridane, A. (2003, December). An FPGA based coprocessor for large matrix product implementation. In Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT)(IEEE Cat. No. 03EX798) (pp. 292-295). IEEE.

21. Mencer, O., Morf, M., & Flynn, M. J. (1998, April). PAM-Blox: High performance FPGA design for adaptive computing. In Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB100251) (pp. 167-174). IEEE.

22. Amira, A., Bouridane, A., & Milligan, P. (2001, August). Accelerating matrix product on reconfigurable hardware for signal processing. In International Conference on Field Programmable Logic and Applications (pp. 101-111). Berlin, Heidelberg: Springer Berlin Heidelberg.

23. Bensaali, F., Amira, A., & Bouridane, A. (2005). Accelerating matrix product on reconfigurable hardware for image processing applications. IEE proceedings-Circuits, Devices and Systems, 152(3), 236-246.

24. Jang, J. W., Choi, S., & Prasanna, V. K. K. (2002, December). Area and time efficient implementations of matrix multiplication on FPGAs. In 2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings. (pp. 93-100). IEEE.

25. Jang, J. W., Choi, S. B., & Prasanna, V. K. (2005). Energy-and time-efficient matrix multiplication on FPGAs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 13(11), 1305-1319.

26. Belkacemi, S., Benkrid, K., Crookes, D., & Benkrid, A. (2003, May). Design and implementation of a high performance matrix multiplier core for Xilinx Virtex FPGAs. In 2003 IEEE International Workshop on Computer Architectures for Machine Perception (pp. 4-pp). IEEE.

27. Shang, L., Kaviani, A. S., & Bathala, K. (2002, February). Dynamic power consumption in Virtex™-II FPGA family. In Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays (pp. 157-164).

28. Jang, J. W., Choi, S., & Prasanna, V. K. (2002, August). Energy-efficient matrix multiplication on FPGAs. In International Conference on Field Programmable Logic and Applications (pp. 534-544). Berlin, Heidelberg: Springer Berlin Heidelberg.

29. Choi, S., Prasanna, V. K., & Jang, J. W. (2002, July). Minimizing energy dissipation of matrix multiplication kernel on Virtex-II. In Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications IV (Vol. 4867, pp. 98-106). SPIE.

30. Choi, S., Scrofano, R., Prasanna, V. K., & Jang, J. W. (2003, February). Energy-efficient signal processing using FPGAs. In Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays (pp. 225-234).

31. Jianwen, L., & Chuen, J. C. (2004, August). Partially reconfigurable matrix multiplication for area and time efficiency on FPGAs. In Euromicro Symposium on Digital System Design, 2004. DSD 2004. (pp. 244-248). IEEE.

32. Qasim, S. M., Telba, A. A., & AlMazroo, A. Y. (2010). FPGA design and implementation of matrix multiplier architectures for image and signal

processing applications. International Journal of Computer Science and Network Security, 10(2), 168-176.

33. Lshebeili, S. A. (2001). Computation of higher-order cross moments based on matrix multiplication. Journal of the Franklin Institute, 338(7), 811-816.
34. Chetan, S., Sourabh, K. S., Lekshmi, V., Sudhakar, S., & Manikandan, J. (2020). Design and Evaluation of Floating point Matrix Operations for FPGA based system design. Procedia Computer Science, 171, 959-968.
35. Soliman, M. I., & Elsayed, E. A. (2011, December). Design and FPGA implementation of a simplified matrix processor. In 2011 Seventh International Computer Engineering Conference (ICENCO'2011), pp. 31-36.

ДОДАТОК А ТЕЗИ

Можливості розробки багатоядерного процесора з використанням ПЛІС

Демидович В. М., Шаповалов В. О., Український державний університет науки і технологій

Дуже актуальними у світі обчислювальної техніки були і залишаються ПЛІС (програмовані логічні інтегральні схеми, FPGA – Field-Programmable Gate Array), вони мають безліч застосувань у різних галузях. Актуальне використання ПЛІС для прискорення обчислень у таких напрямках, як наукові дослідження, фінансові обчислення, машинне навчання, хмарні технології, криптографія. ПЛІС можуть прискорювати виконання алгоритмів навчання та виведення рішень у реальному часі. Також вони можуть використовуватися в техніці зв'язку для обробки сигналів в якості високопродуктивних цифрових сигнальних процесорів, особливо, в реальному часі. Використання ПЛІС дозволяє змінювати функціональність пристроїв без створення нових мікросхем (ASIC - Application-Specific Integrated Circuit). За допомогою ПЛІС можна створювати оригінальні цифрові пристрої, особливо там, де стандартні рішення на основі ASIC не підходять.

ПЛІС можуть забезпечити прискорене виконання операцій, пов'язаних також з графічними обчисленнями (зараз для таких обчислень широко використовуються ASIC графічні процесори). У пристроях IoT (Internet of Things, інтернет речей) ПЛІС використовуються для обробки даних у реальному часі. Ефективно також використовувати їх для обробки мережевих пакетів та оптимізації обчислень у розподілених середовищах. З урахуванням своєї гнучкості та можливості програмування, ПЛІС продовжують залишатися актуальними в інноваційних та високотехнологічних галузях. Зокрема, ПЛІС можуть бути використані для матричних обчислень. Одним з можливих шляхів реалізації матричних обчислень є створення багатоядерного спеціалізованого процесора. Такий процесор повинен підтримувати виконання інструкцій над безліччю даних одночасно. З цього випливає, що архітектура такого процесора повинна бути SIMD (Single Instruction, Multiple Data).

Провідними фірмами з випуску ПЛІС та відповідного програмного забезпечення – систем автоматизованого проектування (САПР) є AMD (Xilinx) та Intel (Altera). У деяких ПЛІС, наприклад, таких як Xilinx Virtex і Xilinx UltraScale+ є виділені блоки цифрової обробки сигналів (прискорення обчислень) - DSP48E1, які можуть бути використані для реалізації паралельних обчислень в ПЛІС і запрограмовані для ефективної обробки матричних операцій. У ці блоки закладені апаратні ресурси для виконання арифметичних операцій, у тому числі й

множення. Таким чином, за допомогою блоків DSP48E1 можна виконувати як операції складання та віднімання, так і скалярне та векторне множення матриць. За основу ядра процесора можна взяти блок DSP48E1.

В даний час основним засобом проектування пристроїв на ПЛІС фірми Xilinx є САПР Vivado. Проектування ведеться за допомогою мов проектування апаратури (HDL – Hardware Description Language), наприклад, VHDL. Такий підхід називається низькорівневим проектуванням. Також є засоби високорівневого проектування, які дозволяють істотно скоротити час проектування. До таких засобів, наприклад, можна віднести програму Simulink у системі Matlab. При цьому можна провести моделювання багатоядерного процесора з урахуванням усіх модулів, форматів команд та даних. Після проходження етапу високорівневого проектування далі програмою HDL Coder генерується HDL-опис пристрою, наприклад, мовою VHDL. На основі отриманого VHDL-опису в САПР Vivado проводиться моделювання на рівні всіх сигналів, а також етап синтезу та реалізації пристрою в ПЛІС. При необхідності можна ввести корективи в пристрій, що розробляється, шляхом зміни VHDL-опису.

ДОДАТОК Б VHDL-ОПИС БЛОКІВ

PE.vhd

```
entity Core1 is
    Port ( MA_00 : in STD_LOGIC_VECTOR (31 downto 0);
           MA_01 : in STD_LOGIC_VECTOR (31 downto 0);
           MA_02 : in STD_LOGIC_VECTOR (31 downto 0);
           MA_03 : in STD_LOGIC_VECTOR (31 downto 0);
           MB_00 : in STD_LOGIC_VECTOR (31 downto 0);
           MB_10 : in STD_LOGIC_VECTOR (31 downto 0);
           MB_20 : in STD_LOGIC_VECTOR (31 downto 0);
           MB_30 : in STD_LOGIC_VECTOR (31 downto 0);
           COP : in STD_LOGIC_VECTOR (2 downto 0);
           CLK : in STD_LOGIC;
           ENA : in STD_LOGIC;
           MC_00 : out STD_LOGIC_VECTOR (31 downto 0));
end Core1;

architecture Behavioral of Core1 is
begin
    process (CLK, ENA)
        variable y, a,b,c,d,e,f: std_logic_vector(63 downto 0);
    begin
        if Rising_edge(CLK) and ENA='1'  then
            case COP is
                when "000" => y(31 downto 0):= MA_00 + MB_00;
                when "001" => y(31 downto 0):= MA_00 - MB_00;
                when "010" => a:= MA_00 * MB_00;
                              b:= MA_01 * MB_10;
                              c:= MA_02 * MB_20;
                              d:= MA_03 * MB_30;
                              e:= a + b;
                              f:= c + d;
                              y:= e + f;
                when "011" => y:= MA_00 * MB_00;
                when                          others                          =>
y:="UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
            end case;
            MC_00<=y(31 downto 0);
```

```
            end if;
        end process;
   end Behavioral;
```

Registers.vhd

```
entity Registers is
    Port ( Adr_Reg : in STD_LOGIC_VECTOR (3 downto 0);
           Data : in STD_LOGIC_VECTOR (31 downto 0);
           ENA : in STD_LOGIC;
           CLK : in STD_LOGIC;
           RST : in STD_LOGIC;
           Reg_00 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_01 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_02 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_03 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_04 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_05 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_06 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_07 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_08 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_09 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_10 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_11 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_12 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_13 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_14 : out STD_LOGIC_VECTOR (31 downto 0);
           Reg_15 : out STD_LOGIC_VECTOR (31 downto 0));
end Registers;

architecture Behavioral of Registers is

begin
process (CLK, ENA, Adr_Reg, RST)
begin
    if RST='1' and ENA='1' then
        Reg_00 <= x"00000000";
        Reg_01 <= x"00000000";
        Reg_02 <= x"00000000";
```

```
        Reg_03 <= x"00000000";
        Reg_04 <= x"00000000";
        Reg_05 <= x"00000000";
        Reg_06 <= x"00000000";
        Reg_07 <= x"00000000";
        Reg_08 <= x"00000000";
        Reg_09 <= x"00000000";
        Reg_10 <= x"00000000";
        Reg_11 <= x"00000000";
        Reg_12 <= x"00000000";
        Reg_13 <= x"00000000";
        Reg_14 <= x"00000000";
        Reg_15 <= x"00000000";
    else
        if Rising_edge(CLK) and ENA='1'  then
            case Adr_Reg is
                when x"0" => Reg_00 <= Data;
                when x"1" => Reg_01 <= Data;
                when x"2" => Reg_02 <= Data;
                when x"3" => Reg_03 <= Data;
                when x"4" => Reg_04 <= Data;
                when x"5" => Reg_05 <= Data;
                when x"6" => Reg_06 <= Data;
                when x"7" => Reg_07 <= Data;
                when x"8" => Reg_08 <= Data;
                when x"9" => Reg_09 <= Data;
                when x"A" => Reg_10 <= Data;
                when x"B" => Reg_11 <= Data;
                when x"C" => Reg_12 <= Data;
                when x"D" => Reg_13 <= Data;
                when x"E" => Reg_14 <= Data;
                --when x"F" => Reg_15 <= Data;
                when others => Reg_15 <= Data;
            end case;
        end if;
    end if;
end process;
end Behavioral;
```

```
Register_MC.vhd
entity Register_MC is
    Port ( Reg_00 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_01 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_02 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_03 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_04 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_05 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_06 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_07 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_08 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_09 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_10 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_11 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_12 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_13 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_14 : in STD_LOGIC_VECTOR (31 downto 0);
           Reg_15 : in STD_LOGIC_VECTOR (31 downto 0);
           Adr_Reg : in STD_LOGIC_VECTOR (3 downto 0);
           ENA : in STD_LOGIC;
           CLK : in STD_LOGIC;
           Data : out STD_LOGIC_VECTOR (31 downto 0));
end Register_MC;

architecture Behavioral of Register_MC is

begin
process (CLK, ENA, Adr_Reg)
begin
    if Rising_edge(CLK) and ENA='1'  then
        case Adr_Reg is
                    when x"0" => Data <= Reg_00;
                    when x"1" => Data <= Reg_01;
                    when x"2" => Data <= Reg_02;
                    when x"3" => Data <= Reg_03;
                    when x"4" => Data <= Reg_04;
                    when x"5" => Data <= Reg_05;
```

```
                    when x"6" => Data <= Reg_06;
                    when x"7" => Data <= Reg_07;
                    when x"8" => Data <= Reg_08;
                    when x"9" => Data <= Reg_09;
                    when x"A" => Data <= Reg_10;
                    when x"B" => Data <= Reg_11;
                    when x"C" => Data <= Reg_12;
                    when x"D" => Data <= Reg_13;
                    when x"E" => Data <= Reg_14;
                    when x"F" => Data <= Reg_15;
        end case;
    end if;
end process;
end Behavioral;

D_RAM.vhd
entity D_RAM is
    Port ( ClkA : in  STD_LOGIC;
       EnA : in  STD_LOGIC;
       WeA : in  STD_LOGIC;
       AddrA : in  STD_LOGIC_VECTOR (7 downto 0);
       DIA : in  STD_LOGIC_VECTOR (31 downto 0);
       DOA : out  STD_LOGIC_VECTOR (31 downto 0);
       ClkB : in  STD_LOGIC;
       EnB : in  STD_LOGIC;
       WeB : in  STD_LOGIC;
       AddrB : in  STD_LOGIC_VECTOR (7 downto 0);
       DIB : in  STD_LOGIC_VECTOR (31 downto 0);
       DOB : out  STD_LOGIC_VECTOR (31 downto 0));

end D_RAM;

architecture Behavioral of D_RAM is
  Type ram_type is array(7 downto 0) of std_logic_vector(31 downto 0);
  Shared Variable   RAM_MEM : ram_type;
  --signal   RAM_MEM : ram_type;
  begin
    process (CLKA)
```

```
    begin
      if Rising_Edge(ClkA)then
            if EnA = '1' then
          if WeA = '1' then
                   RAM_MEM(conv_integer(Unsigned(AddrA))) := DIA;
                   --RAM_MEM(conv_integer(Unsigned(AddrA))) <= DIA;
            else
          DOA <= RAM_MEM(conv_integer(AddrA));
          end if;
            end if;
      end if;
    end process;
    process (CLKB)
    begin
      if Rising_Edge(ClkB)then
            if EnB = '1' then
          if WeB = '1' then
                   RAM_MEM(conv_integer(Unsigned(AddrB))) := DIB;
              --RAM_MEM(conv_integer(Unsigned(AddrB))) <= DIB;
            else
          DOB <= RAM_MEM(conv_integer(AddrB));
          end if;
            end if;
      end if;
    end process;
end Behavioral;

P_RAM.vhd
entity P_RAM is
    Port ( Clk : in  STD_LOGIC;
           EnA : in  STD_LOGIC;
           We : in  STD_LOGIC;
           Addr_Instr : in  STD_LOGIC_VECTOR (7 downto 0);
           DI : in  STD_LOGIC_VECTOR (38 downto 0);
           DO : out  STD_LOGIC_VECTOR (38 downto 0));
end P_RAM;

architecture Behavioral of P_RAM is
```

```
  Type ram_type is array(7 downto 0) of std_logic_vector(38 downto 0);
  Shared Variable   PRAM_MEM : ram_type;
  --signal   PRAM_MEM : ram_type;
begin
    process (CLK)
     begin
      if Rising_Edge(Clk)then
          if EnA = '1' then
          if We = '1' then
                PRAM_MEM(conv_integer(Unsigned(Addr_Instr))) := DI;
                --PRAM_MEM(conv_integer(Unsigned(Addr_Instr))) <= DI;
             else
          DO <= PRAM_MEM(conv_integer(Addr_Instr));
          end if;
          end if;
      end if;
    end process;
end Behavioral;

Adr_Generator_for_Matrix.vhd
entity Adr_Generator_for_Matrix is
    Port ( Size_MA : in STD_LOGIC_VECTOR (5 downto 0);
           Size_MB : in STD_LOGIC_VECTOR (5 downto 0);
           Size_MC : in STD_LOGIC_VECTOR (5 downto 0);
           AB_Adr : in STD_LOGIC;
           C_Adr : in STD_LOGIC;
           ENA : in STD_LOGIC;
           CLK : in STD_LOGIC;
           RST : in STD_LOGIC;
           Adr_Reg_MA : out STD_LOGIC_VECTOR (3 downto 0);
           Adr_Reg_MB : out STD_LOGIC_VECTOR (3 downto 0);
           Adr_Reg_MC : out STD_LOGIC_VECTOR (3 downto 0));
end Adr_Generator_for_Matrix;

architecture Behavioral of Adr_Generator_for_Matrix is
signal Count_Clk : std_logic_vector(4 downto 0);
begin
process (CLK, ENA, RST)
```

```
    variable Adr_MA, Adr_MB, Adr_MC: std_logic_vector(3 downto 0);
    variable a, b, c: integer;
    begin
    if RST='1' and ENA='1' then
        Count_Clk <= "00000";
    else
        if  Rising_edge(CLK) then
        if ENA='1' and AB_Adr='1' then
            if Count_Clk <= Size_MA(3 downto 0) then
                a := (to_integer(unsigned(Count_Clk))) mod (to_integer(unsigned(Size_MA(5
downto 4 )))+1);
                b  :=  to_integer(unsigned(Count_Clk))  /  (to_integer(unsigned(Size_MA(5
downto 4 )))+1);
                c := ((b*4)+a);
                Adr_Reg_MA <= std_logic_vector(to_unsigned(c, Adr_Reg_MA'length));
            end if;
            if Count_Clk <= Size_MB(3 downto 0) then
                        a         :=         (to_integer(unsigned(Count_Clk)))         mod
(to_integer(unsigned(Size_MB(5 downto 4 )))+1);
                        b          :=          to_integer(unsigned(Count_Clk))          /
(to_integer(unsigned(Size_MB(5 downto 4 )))+1);
                        c := ((b*4)+a);
                        Adr_Reg_MB           <=           std_logic_vector(to_unsigned(c,
Adr_Reg_MB'length));
            end if;
            Count_Clk <= Count_Clk + 1;
        else
            if ENA='1' and C_Adr='1' then
                 if Count_Clk <= Size_MC(3 downto 0) then
                            a       :=       (to_integer(unsigned(Count_Clk)))       mod
(to_integer(unsigned(Size_MC(5 downto 4 )))+1);
                            b        :=        to_integer(unsigned(Count_Clk))        /
(to_integer(unsigned(Size_MC(5 downto 4 )))+1);
                            c := ((b*4)+a);
                            Adr_Reg_MC         <=         std_logic_vector(to_unsigned(c,
Adr_Reg_MC'length));
                            Count_Clk <= Count_Clk + 1;
                end if;
```

```
         end if;
    end if;
    end if;
end if;
end process;
end Behavioral;

Port_A.vhd
entity Port_A is
    Port ( Instraction_PortIn : in STD_LOGIC_VECTOR (38 downto 0);
           Data_PortIn : in STD_LOGIC_VECTOR (31 downto 0);
           Instraction_PortOut : out STD_LOGIC_VECTOR (38 downto 0);
           Data_PortOut : out STD_LOGIC_VECTOR (31 downto 0);
           Wr_Inst : in STD_LOGIC;
           Wr_Data : in STD_LOGIC;
           Rd_In : in STD_LOGIC;
           Rd_PortOut : out STD_LOGIC;
           Clk : in STD_LOGIC;
           Start : in STD_LOGIC;
           Start_Out : out STD_LOGIC;
           Wr_Inst_Out : out STD_LOGIC;
           Wr_Data_Out : out STD_LOGIC;
           Instraction_In : in STD_LOGIC_VECTOR (38 downto 0);
           Data_In : in STD_LOGIC_VECTOR (31 downto 0);
           Instraction_Out : out STD_LOGIC_VECTOR (38 downto 0);
           Data_Out : out STD_LOGIC_VECTOR (31 downto 0));
end Port_A;


architecture Behavioral of Port_A is
begin
    process (Clk)
   begin
      if Rising_Edge(Clk)then
       Wr_Inst_Out<=Wr_Inst;
       Wr_Data_Out<=Wr_Data;
       Rd_PortOut<=Rd_In;
       Start_Out<=Start;
```

```
        if Wr_Inst = '1' and Start = '0' then
            Instraction_Out<=Instraction_PortIn;
        end if;
        if Wr_Data = '1' and Start = '0' then
            Data_Out<=Data_PortIn;
        end if;
        if Rd_In ='1' then
         Instraction_PortOut<=Instraction_In;
         Data_PortOut<=Data_In;
        end if;
      end if;
   end process;
end Behavioral;
```

ДОДАТОК В TESTBENCH

PE_tb.vhd

```
Clk_tb<=not Clk_tb after 20 ns;

 MA_00_tb <= x"00000001";
 MA_01_tb <= x"00000002";
 MA_02_tb <= x"00000003";
 MA_03_tb <= x"00000004", x"00000000" after 260 ns;

 MA_10_tb <= x"00000005", x"00000000" after 180 ns, x"00000005" after 260 ns;
 MA_11_tb <= x"00000006", x"00000000" after 180 ns, x"00000006" after 260 ns;
 MA_12_tb <= x"00000007", x"00000000" after 180 ns, x"00000007" after 260 ns;
 MA_13_tb <= x"00000008", x"00000000" after 180 ns, x"00000000" after 260 ns;

 MA_20_tb <= x"00000009", x"00000000" after 180 ns, x"00000009" after 260 ns;
 MA_21_tb <= x"0000000A", x"00000000" after 180 ns, x"0000000A" after 260 ns;
 MA_22_tb <= x"0000000B", x"00000000" after 180 ns, x"0000000B" after 260 ns;
 MA_23_tb <= x"0000000C", x"00000000" after 180 ns, x"00000000" after 260 ns;

 MA_30_tb <= x"0000000D", x"00000000" after 180 ns;
 MA_31_tb <= x"0000000E", x"00000000" after 180 ns;
 MA_32_tb <= x"0000000F", x"00000000" after 180 ns;
 MA_33_tb <= x"00000010", x"00000000" after 180 ns;

 MB_00_tb <= x"00000002", x"00000002" after 140 ns, x"00000002" after 220 ns;
 MB_10_tb <= x"00000006", x"00000003" after 140 ns, x"00000006" after 220 ns;
 MB_20_tb <= x"0000000A", x"00000004" after 140 ns, x"0000000A" after 220 ns;
 MB_30_tb <= x"0000000E", x"00000005" after 140 ns, x"0000000E" after 220 ns,
x"00000000" after 260 ns;

 MB_01_tb <= x"00000003", x"00000000" after 140 ns, x"00000003" after 220 ns;
 MB_11_tb <= x"00000007", x"00000000" after 140 ns, x"00000007" after 220 ns;
 MB_21_tb <= x"0000000B", x"00000000" after 140 ns, x"0000000B" after 220 ns;
 MB_31_tb <= x"0000000F", x"00000000" after 140 ns, x"0000000F" after 220 ns,
x"00000000" after 260 ns;

 MB_02_tb <= x"00000004", x"00000000" after 140 ns, x"00000004" after 220 ns;
 MB_12_tb <= x"00000008", x"00000000" after 140 ns, x"00000008" after 220 ns;
```

```
    MB_22_tb <= x"0000000C", x"00000000" after 140 ns, x"0000000C" after 220 ns;
    MB_32_tb <= x"00000010", x"00000000" after 140 ns, x"00000010" after 220 ns,
x"00000000" after 260 ns;

    MB_03_tb <= x"00000005", x"00000000" after 140 ns, x"00000005" after 220 ns,
x"00000000" after 260 ns;
    MB_13_tb <= x"00000009", x"00000000" after 140 ns, x"00000009" after 220 ns,
x"00000000" after 260 ns;
    MB_23_tb <= x"0000000D", x"00000000" after 140 ns, x"0000000D" after 220 ns,
x"00000000" after 260 ns;
    MB_33_tb <= x"00000011", x"00000000" after 140 ns, x"00000011" after 220 ns,
x"00000000" after 260 ns;

    COP_tb <= "010", "000" after 60 ns,  "001" after 100 ns, "010" after 140 ns; --,
"000" after 180 ns,  "001" after 220 ns;
    ENA_tb <= '1';

   Registers_tb.vhd
   CLK_tb <= not CLK_tb after 20 ns;
   ENA_tb <= '1', '0' after 680 ns;
   RST_tb <= '1', '0' after 10 ns;
   Data_tb <= x"00000001" after 20 ns, x"00000002" after 60 ns, x"0000003" after 100 ns,
x"00000004"  after 140 ns, x"00000005"  after 180 ns, x"00000006"  after 220 ns,
x"00000007"  after 260 ns, x"00000008"  after 300 ns, x"00000009"  after 340 ns,
x"0000000A"  after 380 ns, x"0000000B"  after 420 ns, x"0000000C"  after 460 ns,
x"0000000D"  after 500 ns, x"0000000E"  after 540 ns, x"0000000F"  after 580 ns,
x"00000010"  after 620 ns;
   Adr_Reg_tb <= x"0"  after 20 ns, x"1" after 60 ns, x"2" after 100 ns, x"3" after 140
ns, x"4" after 180 ns, x"5" after 220 ns, x"6" after 260 ns, x"7" after 300 ns, x"8" after
340 ns, x"9"  after 380 ns, x"A" after 420 ns, x"B" after 460 ns, x"C" after 500 ns, x"D"
after 540 ns, x"E" after 580 ns, x"F" after 620 ns;

   D_RAM_tb.vhd
       ClkA_tb<=not ClkA_tb after 20 ns;
       ClkB_tb<=not ClkB_tb after 20 ns;

      EnA_tb<='1', '0' after 200 ns;
            EnB_tb<='1', '0' after 300 ns;
```

```
      WeA_tb<='1', '0' after 100 ns;
      WeB_tb<='1', '0' after 100 ns;

      AddrA_tb<="00000000", "00000001" after 140 ns;
      AddrB_tb<="00000001", "00000000" after 170 ns;

      DIA_tb<=X"00000000";
      DIB_tb<=X"00010000";

   P_RAM_tb.vhd
   Clk_tb<=not Clk_tb after 20 ns;

    EnA_tb<='1', '0' after 200 ns;

    We_tb<='1', '0' after 100 ns;

    Addr_Instr_tb<="00000000", "00000001" after 50 ns, "00000000" after 100 ns,
"00000001" after 140 ns;
    DI_tb<="00000000000000000000000000000000000000000",
"00000000000000000000000000000000000000100" after 50 ns;

   Port_A_tb.vhd
   Clk_tb<=not Clk_tb after 20 ns;
    Wr_Inst_tb<='1', '0' after 130 ns;
    Wr_Data_tb<='0', '1' after 130 ns, '0' after 225 ns;
    Rd_In_tb<='0', '1' after 300 ns;
    Start_tb<='0', '1' after 260 ns, '0' after 300 ns;
    Instraction_PortIn_tb<="00000000000000000000000000000000000000001",
"00000000000000000000000000000000000000010" after 40 ns,
"00000000000000000000000000000000000000011" after 80 ns;
    Data_PortIn_tb<=x"00000001" after 140 ns, x"00000100" after 180 ns, x"00001000" after
220 ns;
    Instraction_In_tb<="00000000000000000000000000000000000000001" after 300 ns,
"00000000000000000000000000000000000000010" after 340 ns,
"00000000000000000000000000000000000000011" after 380 ns;
    Data_In_tb<=x"00000001" after 300 ns, x"00000100" after 340 ns, x"00001000" after
380 ns;
```