НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський політехнічний інститут ім. Ігоря Сікорського»

ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ
КАФЕДРА КОНСТРУЮВАННЯ ЕЛЕКТРОННО-ОБЧИСЛЮВАЛЬНОЇ АПАРАТУРИ

«На правах рукопису»
УДК 6.621.3

«До захисту допущено»
Завідувач кафедри
_____ Лисенко О.М.
(підпис) (ініціали, прізвище)

“23” _грудня_ 2022р.

Магістерська дисертація

зі спеціальності *172 "Телекомунікації та радіотехніка*
(код та назва напряму підготовки або спеціальності)

на тему Апаратно-програмний комплекс для маніпулювання складними типами даних

Виконав: студент II курсу, групи ДК-11мп

Кудлай Станіслав Васильович _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник професор д.ф.-м. н. І. В. Редько _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Консультант ст. викл., к.т.н., О. І. Антонюк _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент доцент к.т.н. К. О. Трапезон _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій мігастерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2022 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) електроніки
(повна назва)

Кафедра конструювання електронно-обчислювальної апаратури
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою
Спеціальність 172 – Телекомунікації та радіотехніка
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
О.М. Лисенко
(підпис) (ініціали, прізвище)

«05» вересня 2022 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Кудлаю Станіславу Васильовичу
(прізвище, ім'я, по батькові)

1. Тема проекту Апаратно-програмний комплекс для маніпулювання складними типами даних

Науковий керівник Редько Ігор Володимирович, професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 08» грудня 2022 р. № 4092-с

2. Термін подання студентом проекту 21 грудня 2022 року

3. Об'єкт дослідження клас конструктивних маніпуляційних функцій над реляціями та вивчення можливості їхньої реалізації в комп'ютерній архітектурі у вигляді реляційного процесору

4. Предмет дослідження Процеси обробки структурованої інформації, зокрема реляційних баз даних, та можливості їх прискорення

5. Перелік завдань, які потрібно розробити: виведення повної сукупності породжуючих класу р-функцій та чр-предикатів над реляціями як функції над багатовимірним типом даних; дослідження архітектури матричного

процесору на предмет його можливості підтримки операцій над реляціями та можливістю його реконфігурації для прискорення довільного n-вимірного типу даних; розробка архітектури реляційного процесору на основі матричного прискорювача з введенням необхідних модифікацій та можливістю реконфігурації

6. Перелік графічного матеріалу (із зазначенням обов'язкових креслень, плакатів, презентацій тощо):

— схема електрична функціональна реляційного процесору;

— графічна презентація.

7. Орієнтовний перелік публікацій: 1 публікація.

8. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
3 Реляційний процесор	Антонюк О. І., ст. викл.		

9. Дата видачі завдання 01.09.22

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів проекту	Примітка
1	Узгодження плану роботи з керівником	01.09.22 – 01.09.22	виконано
2	Аналіз останніх робіт по обранному напрямку	02.09.22 – 15.09.22	виконано
3	Формування вимог та задач до об’єкту вивчення, проектування	16.09.22 – 17.09.22	виконано
4	Опрацювання теоретичних матеріалів	18.09.22 – 30.09.22	виконано
5	Виведення алгебраїчної характеристики чр-функцій та предикатів над реляціями з подальшим уточненням іменною моделлю даних	01.10.22 – 15.10.22	виконано з запізненням
6	Розробка архітектури реляційного процесору на основі прискорювача матричних операцій	16.10.22 – 22.10.22	виконано
7	Введення механізму стекування та модифікації керуючого ядра	23.10.22 – 31.10.22	виконано
8	Модифікація пам’яті реляцій та блоку обчислення даних	01.11.22 – 10.11.22	виконано
9	Модифікація блоку обчислення адрес	11.11.22 – 17.11.22	виконано
10	Розробка блоку керування операціями	18.11.22 – 01.12.22	виконано з запізненням
11	Виконання стартап проекту	01.12.22 – 04.12.22	виконано з запізненням
12	Оформлення креслення	05.12.22 – 06.12.22	виконано з запізненням

Студент _____ Кудлай С. В.
(підпис) (прізвище та ініціали)

Науковий керівник дисертації _____ Редько І. В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Магістерська дисертація складається з 180 сторінок, в якій міститься 84 рисунка, 37 таблиць, використано 48 джерел.

Актуальність. Бази даних це одне з ключових направлень в індустрії інформаційних технологій. Проблема збереження та зміни даних виникла ще задовго до формування індустріальних технологій, та навіть паперу. При зародженні ж комп'ютерних наук дана проблема постала відразу. Згодом, бази даних стали окремою гілкою комп'ютерних наук. По мірі розвитку цифрових технологій та діджиталізації суспільства проблематика баз даних стає дедалі актуальнішою. На сьогоднішній день усі мастадонти сфери інформаційних технологій мають свої сервери для зберігання даних, а деякі компанії працюють лише з ними. Бази даних використовуються в сучасному житті майже усюди, від звичайних сайтів до величезних державних та наукових мереж. А розвиток хмарних технологій постійно підвищує вимоги до баз даних. Зараз, хмарні технлології це не лише зберігання даних, а й їх обчислення. Так, з кожним роком розвиваються ігрові хмарні сервіси, які дають змогу користувачам грати в комп'ютерні ігри на будь якому апараті -- всі обчислення відбуваються на стороні серверу. Також до хмарних обчислень поступово входять й обчислення нейроних мереж. Самі ж хмарні сервери є величезним кластером комп'ютерів, які споживають величезну кількість електроенергії, охолоджуються, вимагають обсуговування, тощо.

Одним з найпоширеніших типів баз даних є реляційний. Дані у ньому пов'язані між собою певним відношенням, а відношення представлені у вигляді двовимірних таблиць. Таблиці, в свою чергу, ідентичні до матриць у плані їх збереження в пам’яті. Тому апаратне прискорення базових операцій над реляціями не є складним, проте дозволить значно збільшити продуктивність всієї системи.

Мета та завдання. Метою даної роботи є дослідження класу конструктивних маніпуляційних функцій над реляціями та вивчення можливості їхньої адаптації апаратної реалізації; а також дослідження

архітектури раніше розробленого матричного процесору (останньої модифікації) на предмет його повноти для n-вимірних типів даних з можливістю зміни базових операцій та, у разі потреби, покращення архітектури раніше розробленого матричного процесору задля забезпечення вищезазначеного.

Для досягнення мети, в роботі ставляться та вирішуються наступні задачі:

1. відображення типу даних реляції на матричний тип з послідуючим їх уточненням іменною моделлю даних;
2. виведення повної сукупності породжуючих класу р-функцій та чр-предикатів над реляціями як функції над багатовимірним типом даних;
3. дослідження архітектури матричного процесору на предмет його можливості підтримки операцій над реляціями та можливістю його реконфігурації для прискорення довільного n-вимірного типу даних.
4. створення архітектури реляційного процесору на основі матричного для підтримки операцій над реляціями та його реконфігурації.

Об'єкт дослідження. Процеси обробки структурованої інформації, зокрема реляційних баз даних, та можливості їх прискорення.

Предмет дослідження. Алгебраїчна характеристика класу частково-рекурсивних багатомісних функцій та предикатів над складними типами даних, зокрема реляції, та апаратно-програмна реалізація базових реляційних маніпуляцій.

Методи. Проведені в роботі дослідження базуються на низці загальнометодологічних та логіко-математичних методів. До перших відносяться, перед усім, методи введення та виключення абстракції. До других – метод уточнення складних даних іменною множиною та метод ізоморфних відображень.

Наукову новизну складає виведена алгебраїчна характеристика класу частково-рекурсивних функцій та частково-рекурсивних предикатів над реляціями та отримана теорема про її повноту в примітивній програмній

алгебрі. Потужність даної алгебри була доказана виведенням з її допомогою примітивних операцій алгебри Кодда. Також було отримано результат по управлінню складною архітектурою за допомогою скінченого автомату для реалізації функцій над багатомісними структурованими типами даних.

Зв'язок роботи з науковими програмами, планами, темами. Дослідження проводилися відповідно до наукових напрямків діяльності кафедри конструювання електронно-обчислювальної апаратури, а також пріоритетного напрямку розвитку науки і техніки України "Інформаційні та комунікаційні технології". Основні результати були отримані в рамках науково-дослідної роботи № д.р. 0113U001874 (шифр «ФЕЛ-4/5») «**Прискорення обчислень з використанням логічних пристроїв, що реконфігуруються**».

Практичну значимість отриманих результатів складає:

- розробка архітектури реляційного процесора як програмно-апаратної реалізації алгебри реляцій;
- апаратно-програмна реалізація всіх частково-рекурсивних багатомісних функцій над реляціями з алгебри Кодда, з повної сукупності породжуючих класу функцій та додаткових функцій;
- використання результатів дослідження при викладанні дисципліни «Основи побудови інформаційно-обчислювальнх засобів інтеграції» (лабораторний практикум) для студентів спеціальності 172 «Телекомунікації та радіотехніка» освітньо-професійної програми «Інформаційно-обчислювальні засоби радіоелектронних систем» на кафедрі КЕОА факультету електроніки КПІ ім. Ігоря Сікорського.

Апробація роботи. Основні результати роботи пройшли апробацію на Міжнародному змаганні InnovateFPGA 2021-2022. Робота під назвою Reconfigurable matrix co-processor від представленої автором команди EM029 стала переможцем першого туру європейсько-африканського регіону. Також окрема робота в даному напрямку була представлена на конференції IEEE UkrMiCo'2021 під назвою «Digital Equalizer Model for the Microcontroller»

Публікації. За матеріалами досліджень опубліковано 1 друковану статтю [31].

ABSTRACT

The master's thesis consists of 180 pages, of which 84 figure, 37 tables, 48 sources are used.

Actuality. Databases are one of the key directions in the information technology industry. The problem of saving and changing data arose long before the formation of industrial technologies, and even paper. At the fondations of computer science, this problem arose immediately. Later, databases became a separate branch of computer science. With the development of digital technologies and the digitalization of society, the issue of databases is becoming more and more relevant. Today, all giant companies in the field of information technology have their own servers for data storage, and some companies work only with them. Databases are used in modern life almost everywhere, from ordinary websites to huge government and scientific networks. The development of cloud technologies constantly increases the requirements for databases. Now, cloud technologies not only store the data, but also make the calculations. Thus, cloud gaming services are developing every year, which allow users to play computer games on any device - all calculations take place on the server side. Cloud computing is also gradually including neural network computing. The cloud servers themselves are a huge cluster of computers that consume a huge amount of power, cooled down, require maintenance, etc.

One of the most common types of databases is relational. The data in it are related to each other by a certain relationship, and the relationships are presented in the form of two-dimensional tables. Tables, in turn, are identical to matrices in terms of their storage in memory. Therefore, hardware acceleration of basic operations on relations is not difficult, but it will allow to significantly increase the performance of the entire system.

Purpose and tasks. The purpose of this work is to study the class of constructive manipulation functions over relations and study the possibility of their hardware acceleration. And also examining the architecture of the previously developed matrix processor (latest modification) for its completeness for n-dimensional data types with the possibility of changing the basic operations and, if necessary, improving the architecture of the previously developed matrix processor to provide the above.

To achieve the goal, the following tasks are set and solved in the work:

1. mapping the data type of the relation to the matrix or vector types with their subsequent refinement by the nominal data model;
2. derivation of the complete set of class-generating partially recursive functions and predicates over relations as functions over a multidimensional data type;
3. a study of the matrix processor architecture for its ability to support relational operations and its reconfiguration to accelerate an arbitrary n-dimensional data type.
4. developing relations processor architecture based on the matrix processor to support relational operations and its reconfiguration.

Object of study. Structured information processing, in particular relational databases, and the possibilities of their hardware-software acceleration.

Subject of study. Algebraic characterization of a class of partially recursive multi-place functions and predicates over complex data types, in particular relations, and hardware-software implementation of basic relational manipulations.

Methods. The research is based on a number of general methodological and logical-mathematical methods. The first include, first of all, the methods of inclusion and exclusion of abstraction. The second is the method of specifying complex data with a nominal set and the method of isomorphic mappings.

The scientific novelty is the derived algebraic characterization of the class of partially recursive functions and partially recursive predicates over relations and the obtained theorem on its completeness in primitive programming algebra. The power of derived algebra was proved by implementing primitive operations of the Codd algebra using the above algebra. Also, a result was obtained on the management of a complex architecture using a finite state machine for the implementation of functions over multi-place structured data types.

Connection of work with scientific programs, plans, topics. The research was carried out in accordance with the scientific directions of the department of design of electronic and computing equipment, as well as the priority direction of the development of science and technology of Ukraine "Information and communication technologies". The main results were obtained within the framework of research work No. Dr. 0113U001874 (code "FEL-4/5") "Acceleration of calculations using reconfigurable logical devices."

The practical significance of the obtained results is:

- development of relational processor architecture as a hardware-software implementation of relational algebra;
- hardware and software implementation of all partially recursive multi-place functions over relations from the Codd algebra, from the complete set of functions generating a class of functions and additional functions;\
- the use of research results in the teaching of the discipline "Fundamentals of the construction of information and computing means of integration" (laboratory practicum) for students of the specialty 172 "Telecommunications and radio engineering" of the educational and professional program "Information and computing means of radio electronic systems" at the KEOA department of the Faculty of Electronics of Igor Sikorsky Kyiv Polytechnic Institue.

Approbation of work. The main results of the work were verified at the International InnovateFPGA 2021-2022 competition. The work entitled Reconfigurable matrix co-processor from the EM029 team represented by the author became the winner of the first round of the European-African region. Also, a separate work in this direction was presented at the IEEE UkrMiCo'2021 conference under the title "Digital Equalizer Model for the Microcontroller"

Publications. Based on research materials, one printed article was published [31].

Зміст

Перелік скорочень, умовних позначень, термінів

RISC	Reduced Instructions Set Computer
CISC	Complex Instructions Set Computer
чр	частково-рекурсивні
SIMD	Single Instruction - Multiple Data
MIPS	Microprocessor without Interlocked Pipeline Stages
ALU	Arithmetic-Logic Unit
TPU	Tensor Processing Unit
SM	Stream Processor
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
DNN	Deep Neural Network
FPU	Floating-Point Unit
PCI-E	Peripheral Component Interconnect Express
MAC адреса	Media Access Control адреса
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
GPU	Graphics Processing Unit
CPU	Central Processing Unit
I-кадр	Інтракодований кадру
ППА	Примітивна Програмна Алгебра
SQL	Structured Query Language
I/O	Input/Output
NZCV	прапорці Negative, Zero, Carry out, Overflow
MAC	Multipy-accumulate
БД	База Даних
СУБД	Система Управління Базою Даних
MVP	Minimal Viable Product

EATX	Extended ATX
ATX	Advanced Technology Extended
OC	Overclocking
ОС	Операційна Система
API	Application Programming Interface
OEM	Original Equipment Manufacturer

Вступ

Дана дисертаційна робота продовжує тему досліджень апаратно-програмного прискорення обчислень багатовимірних масивів даних. У бакалаврській дипломній роботі «Прискорювач матричних операцій» [1] було досліджено апаратно-програмне прискорення матриць з виведенням його алгебраїчної характеристики та побудови архітектури, що підтримує в повному обсязі цю характеристику. В цій же роботі порушується питання прискорення операцій з реляціями та таблицями. Насамперед це проведення аналізу класу реляційних рекурсивних функцій (далі р-функцій) та частково рекурсивних предикатів (далі чр-предикатів), а також можливості їх імплементації в архітектуру матричного процесору з мінімальними змінами самої архітектури. Дослідження проводяться з перспективою виходу на визначення класу задач з обчислення багатовимірних даних, де матриці та таблиці є представниками цього класу.

Актуальність. Бази даних це одне з ключових направлень в індустрії інформаційних технологій. Проблема збереження та зміни даних виникла ще задовго до формування індустріальних технологій, та навіть паперу. При зародженні ж комп'ютерних наук дана проблема постала відразу. Згодом, бази даних стали окремою гілкою комп'ютерних наук. По мірі розвитку цифрових технологій та діджиталізації суспільства проблематика баз даних стає дедалі актуальнішою. На сьогоднішній день усі мастадонти сфери інформаційних технологій мають свої сервери для зберігання даних, а деякі компанії працюють лише з ними. Бази даних використовуються в сучасному житті майже усюди, від звичайних сайтів до величезних державних та наукових мереж. А розвиток хмарних технологій постійно підвищує вимоги до баз даних. Зараз, хмарні технлології це не лише зберігання даних, а й їх обчислення. Так, з кожним роком розвиваються ігрові хмарні сервіси, які дають змогу користувачам грати в комп'ютерні ігри на будь якому апараті -- всі обчислення відбуваються на стороні серверу. Також до хмарних

обчислень поступово входять й обчислення нейроних мереж. Самі ж хмарні сервери є величезним кластером комп'ютерів, які споживають величезну кількість електроенергії, охолоджуються, вимагають обсуговування, тощо.

Одним з найпоширеніших типів баз даних є реляційний. Дані у ньому пов'язані між собою певним відношенням, а відношення представлені у вигляді двовимірних таблиць. Таблиці, в свою чергу, ідентичні до матриць у плані їх збереження в пам’яті. Тому апаратне прискорення базових операцій над реляціями не є складним, проте дозволить значно збільшити продуктивність всієї системи.

Мета та завдання. Метою даної роботи є дослідження класу конструктивних маніпуляційних функцій над реляціями та вивчення можливості їхньої адаптації апаратної реалізації; а також дослідження архітектури раніше розробленого матричного процесору (останньої модифікації) на предмет його повноти для n-вимірних типів даних з можливістю зміни базових операцій та, у разі потреби, покращення архітектури раніше розробленого матричного процесору задля забезпечення вищезазначеного.

Для досягнення мети, в роботі ставляться та вирішуються наступні задачі:

1. відображення типу даних реляції на матричний тип з послідуючим їх уточненням іменною моделлю даних;
2. виведення повної сукупності породжуючих класу p-функцій та чр-предикатів над реляціями як функції над багатовимірним типом даних;
3. дослідження архітектури матричного процесору на предмет його можливості підтримки операцій над реляціями та можливістю його реконфігурації для прискорення довільного n-вимірного типу даних.
4. створення архітектури реляційного процесору на основі матричного для підтримки операцій над реляціями та його реконфігурації.

Об’єкт дослідження. Процеси обробки структурованої інформації, зокрема реляційних баз даних, та можливості їх прискорення.

Предмет дослідження. Алгебраїчна характеристика класу частково-рекурсивних багатомісних функцій та предикатів над складними типами даних, зокрема реляції, та апаратно-програмна реалізація базових реляційних маніпуляцій.

Методи. Проведені в роботі дослідження базуються на низці загальнометодологічних та логіко-математичних методів. До перших відносяться, перед усім, методи введення та виключення абстракції. До других – метод уточнення складних даних іменною множиною та метод ізоморфних відображень.

Зв'язок роботи з науковими програмами, планами, темами. Дослідження проводилися відповідно до наукових напрямків діяльності кафедри конструювання електронно-обчислювальної апаратури, а також пріоритетного напрямку розвитку науки і техніки України “Інформаційні та комунікаційні технології”. Основні результати були отримані в рамках науково-дослідної роботи № д.р. 0113U001874 (шифр «ФЕЛ-4/5») «**Прискорення обчислень з використанням логічних пристроїв, що реконфігуруються**».

Практичну значимість отриманих результатів складає:

- розробка архітектури реляційного процесора як програмно-апаратної реалізації алгебри реляцій;
- апаратно-програмна реалізація всіх частково-рекурсивних багатомісних функцій над реляціями з алгебри Кодда, з повної сукупності породжуючих класу функцій та додаткових функцій;
- використання результатів дослідження при викладанні дисципліни «Основи побудови інформаційно-обчислювальнх засобів інтеграції» (лабораторний практикум) для студентів спеціальності 172 «Телекомунікації та радіотехніка» освітньо-професійної програми

«Інформаційно-обчислювальні засоби радіоелектронних систем» на кафедрі КЕОА факультету електроніки КПІ ім. Ігоря Сікорського.

Особистий внесок автора. Усі результати, які складають суть даної роботи, отримані автором самостійно. З робіт, виконаних зі співавторами, в роботі представлені тільки результати, отримані особисто автором.

Апробація роботи. Основні результати роботи пройшли апробацію на Міжнародному змаганні InnovateFPGA 2021-2022. Робота під назвою Reconfigurable matrix co-processor від представленої автором команди EM029 стала переможцем першого туру європейсько-африканського регіону. Також окрема робота в даному напрямку була представлена на конференції IEEE UkrMiCo’2021 під назвою «Digital Equalizer Model for the Microcontroller»

Публікації. На основі роботи з конференції UkrMiCo’2021 була опублікована наукова робота під назвою «Побудова та верифікація моделі цифрового еквалайзера» в журналі «Вісник Хмельницького національного університету» - 2022. -№5 (313) С.178-184. DOI:10.31891/2307-5732-2022-313-5-178-184.

Структура та обсяг роботи. Дана робота складається зі вступу, шести розділів, висновків, списку використаних джерел (48 найменувань). Загальний обсяг роботи складає 180 стор.

1. Оглядовий розділ

1.1 Типи комп’ютерних архітектур

З розвитком комп’ютерних технологій з’являлося все більше нових архітектур. Згодом їх почали класифікувати та відокремлювати в окремі групи. Типи комп’ютерних архітектур можна умовно поділити на два типи: універсальні та спеціалізовані. Універсальні комп’ютерні архітектури здатні виконувати будь які дії з будь якими даними. Важливою характеристикою при їх розробці виступає повнота по Тьюрінгу. Ця характеристика означає, що комп’ютерна архітектура здатна в повному обсязі імітувати так звану машину Тьюрінга [18], яка здатна виконувати будь які операції з будь якими даними. Спеціалізовані ж заточені під виконання конкретних задач або роботи з певним типом даних. Зазвичай, такі архітектури не можуть працювати самостійно, тому вони знаходяться в тандемі з універсальними в якості сопроцесорів.

На сучасні універсальні архітектури прийнято ділити за двома найбільш поширеними сьогодні наборами інструкцій: обмежений (RISC) та комплексний (CISC). У RISC архітектурах довжина однієї інструкції має фіксований розмір, зазвичай одного машиного слова. При цьому майже всі регістри комп’ютеру, окрім функціональних, є універсальними та можуть бути використані для будь яких даних та операцій. У CISC архітектурі довжина однієї інструкції немає фіксованого розміру, а саме ядро за одну інструкцію виконує декілька дій. До того ж кожен регістр CISC має своє фіксоване функціональне призначення. Простий приклад порівняння двох архітектур це додавання одиниці до значення в пам’яті. Так, у RISC архітектурі ця процедура займає 3 інструкції:

1. вивантажити значення з пам’яті,
2. додати до значення одиницю,
3. завантажити значення в пам’ять.

Та ж сама операція в CISC займає всього лише одну інструкцію. Таким чином програма займає менше місця в пам’яті, що в роки виникнення CISC архітектури зіграло ключову роль для неї, адже тоді пам’ять була повільною та дефіцитною; також додаткова логіка дозволяє виконувати комплексні операції швидше. Проте саме ця додаткова логіка ускладнює цифрову схему, що в наслідку дає більшу площу на кристалі на одне ядро та більше енергоспоживання. Напротивагу CISC, RISC архітектура містить лише саме необхідне, і як наслідок, займає менше місця та споживає менше енергії. До виникнення тренду на багатопоточність домінуючою архітектурою була CISC, RISC використовувався здебільшого лише в малопотужних вбудованих системах. А вже після, а також за рахунок потреби в портативних приладах, архітектура RISC отримала новий поштовх у розвитку.

Спеціалізовані архітектури можна розділити на системно орієнтовані та об'єктно орієнтовані. Про повноту по Тьюрингу в таких архітектурах говорити не доводиться, адже вони націлені на роботу з конкретним класом зачад. Їхній функціонал дозволяє працювати над своїми задачами найбільш ефективно, як у плані часу, так і в плані розміру кристалу. Системно орієнтовані розраховані для роботи з електроно-логічними системами. До їх завдань входять: підтримка інтерфейсів, приймання/передача радіосигналів, кодування/декодування даних, регулювання електричних величин, логічна обробка певних форматів даних, контроль трансмісії даних, тощо. Контролери з такою архітектурою зазвичай слугують периферійними пристоюями універсального обчислювального пристрою (як вбудовані, так і дисктретні). Також існують і самостійні контролери з системно орієнтованою архітектурою, якщо по тим чи іншим причинам не підходить універсальний контролер. Об'єктно орієнтовані архітектури розраховані на обробку певних типів даних, зазвичай складних. Процесори з такою архітектурою називають прискорювачами. Серед найбільш відомих на сьогодні обчислювачів числяться: графічні, нейронні, сигнальні процесори, прискорювачі операцій з

комплексними числами. У загальному випадку такі процесори не можуть працювати самостійно, тому використовуються як сопроцесори у поєднанні з універсальними процесорами.

Методи паралелізації обчислень

Існує два методи розпаралелювання обчислень: векторизація та додавання ядер. При векторизації обчислень в одному слові міститься декілька значень, ці значення оброблюються окремими блоками, які частіше за все можуть оброблювати як одне велике число, так і декілька маленьких. При додаванні ядер програма може розподілити виконання декількох функцій на різні ядра. При цьому ядра працювати повністю незалежно, так і можуть мати спільні дані.

В залежності від конфігурації, ядро може мати механізм векторизації обчислень. Одним з найчастіше використовуваних це принцип виконання команд одна інструкція – багато даних (single instruction-multiple data, SIMD). Зазвичай, у сучасних процесорах використовуються обидва способи паралелізації обчислень.

1.2 Універсальні комп’ютери

Розглянемо архітектури одних з найбільш відомих представників RISC та CISC: MIPS та x86 відповідно. Розглядатися буде безпосередньо сама архітектура ядра, що відповідає за виконання команд та основних обчислень.

Таблиця 1.1. Порівняння архітектур MIPS та x86.

Характеристики	MIPS	x86
Кількість регістрів	32, загального призначення	8, з деякими обмеженнями по використанню
Кількість операндів	3 (2 джерела, 1 призначення)	2 (1 джерело, 1 джерело/призначення)
Розташування операндів	Регістри або безпосередньо операнди	Регістри, безпосередньо операнди або пам’ять
Розмір операнду	32 бити	8, 16 або 32 бити
Коди умов	Ні	Так
Типи команд	Прості	Прості та комплексні
Розмір команд	Фіксований, 4 байти	Змінний, 1–15 байт

У таблиці 1.1 наведені основні відмінності архітектур MIPS та x86. Фактично це підсумок всього вищесказаного. Варто лише зазначити, що в x86 операції відбуваються з накопиченням, тому потрібно всього лише два операнда. Тепер розглянемо детальніше ці дві архітектури.

MIPS є класичним представником RISC архітектури. Довжина однієї інструкції дорівнює довжині машинного слова, має регістри загального призначення.

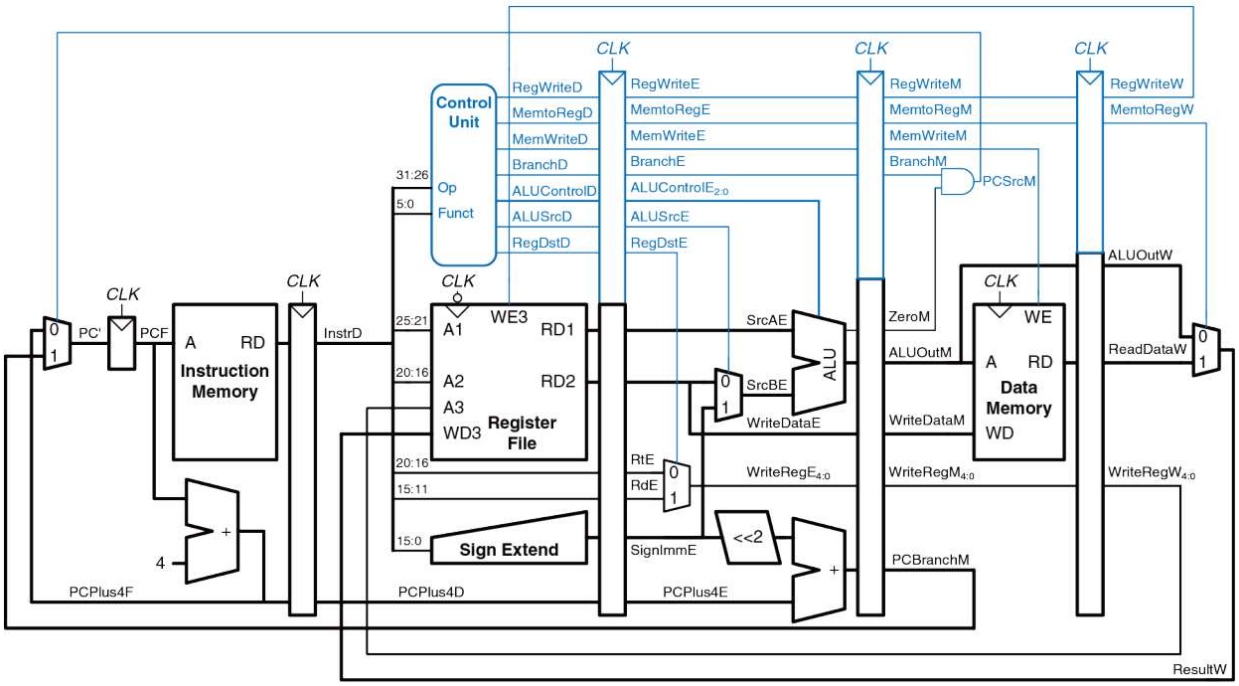


Рисунок 1.1 – Структура архітектури MIPS.

На рисунку 1.1 представлена спрощена структурна схема ядра MIPS. На цій схемі ядро розділено на так звані стадії виконання інструкції:

1. Fetch: вилучення інструкції за адресою в регістрі програмного лічильника PC;
2. Decode: декодування інструкції;
3. Execute: виконання інструкції арифметично-логічним блоком (ALU);
4. Memory/Store: збереження даних до пам’яті;
5. Write back: зчитування пам’яті.

На етапі вилучення з пам’яті програм вилучається інструкція за адресою, збереженою в PC. Одночасно з цим до значення PC додається 4, щоб з наступним тактом зчитати наступну інструкцію. Наступним етапом інструкція декодується. MIPS підтримує 3 типи інструкцій: Register (R), Immediate (I) та Jump (J).

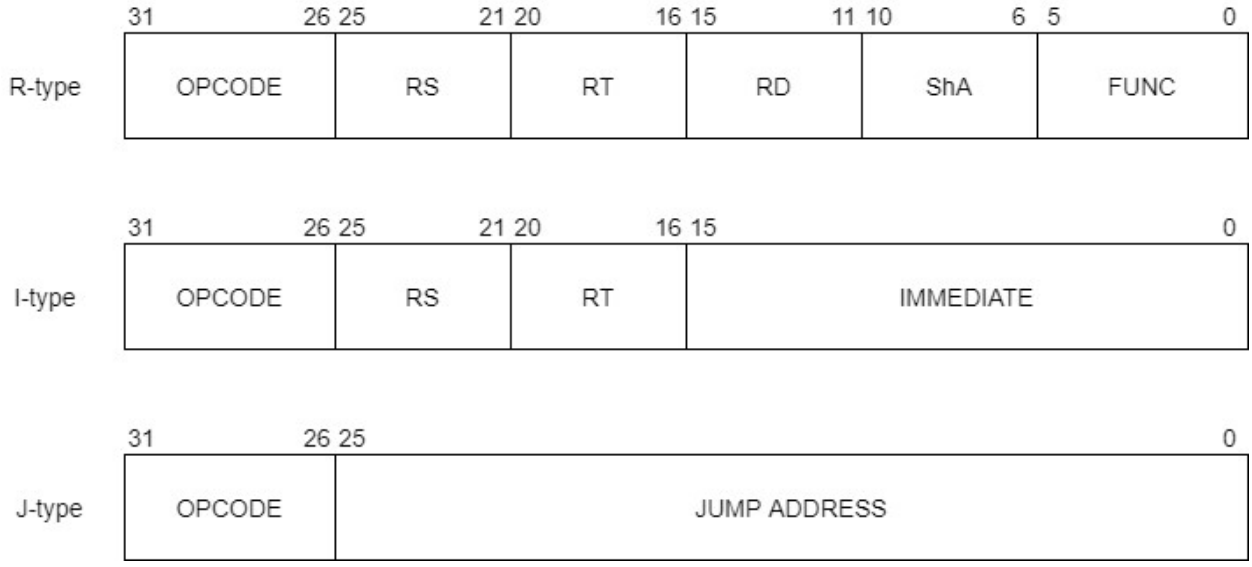


Рисунок 1.2.а – R-тип інструкцій, 1.2.б I-тип інструкцій, 1.2.в J-тип інструкцій.

R-тип інструкцій працює лише з регістрами. Операція вказується в полі FUNC, а в полі OPCODE вказується нульове значення, що вказує на цей тип інструкцій. Значення аргументів беруться з регістрів RS, RT і записуються до регістру RD. Якщо вказана команда зсуву, тоді значення регістру RS зсувається на значення ShA і записується в регістр RD. I-тип, на відміну від попереднього, має поле IMMEDIATE. У ньому вказується константне двійкове значення, яке виступає одним з аргументів операції. Таким чином ядру не треба постійно зчитувати константи з пам’яті. Інструкції J-типу виконують «стрибок» на інструкцію за адресою JUMP ADDRESS. Існують умовні та безумовні стрибки. Цей механізм нелінійного виконання програми взятий напряму з машини Тьюрінга.

Наступним етапом іде виконання інструкції, зазвичай з використанням ALU. Потім відбувається етап збереження даних до пам’яті, сам етап відбувається навіть якщо запису до пам’яті не потрібно. Останнім етапом іде збереження даних до регістрів ядра, цим же етапом зберігається і результат з ALU.

У прикладі використовується принцип конвеєризації (pipeline), тобто між стадіями встановлені буфери даних. Без них усі стадії опрацьовують

одну інструкцію за один такт, з буферами стадії виконують різні інструкції одночасно. До того ж таке рішення дозволяє підняти тактову частоту за рахунок зменшення мінімальної необхідної затримки. Проте, таке рішення породжує новий клас проблем, а саме конвеєрні конфлікти (pipeline hazards). Існує 2 типи конфліктів: конфликти даних (data hazards) та конфликти керування (control hazards) . Для їх вирішення існує декілька методів: обхід певної стадії, призупинка конвеєра, виконання одночасно обох випадків гілкування, передбачення стрибку. Детальніше про MIPS описано в [17].

Архітектура x86 є класичним представником CISC. Вона містить спеціальні регістри та додаткову логіку для забезпечення виконання комплексних інстркцій.

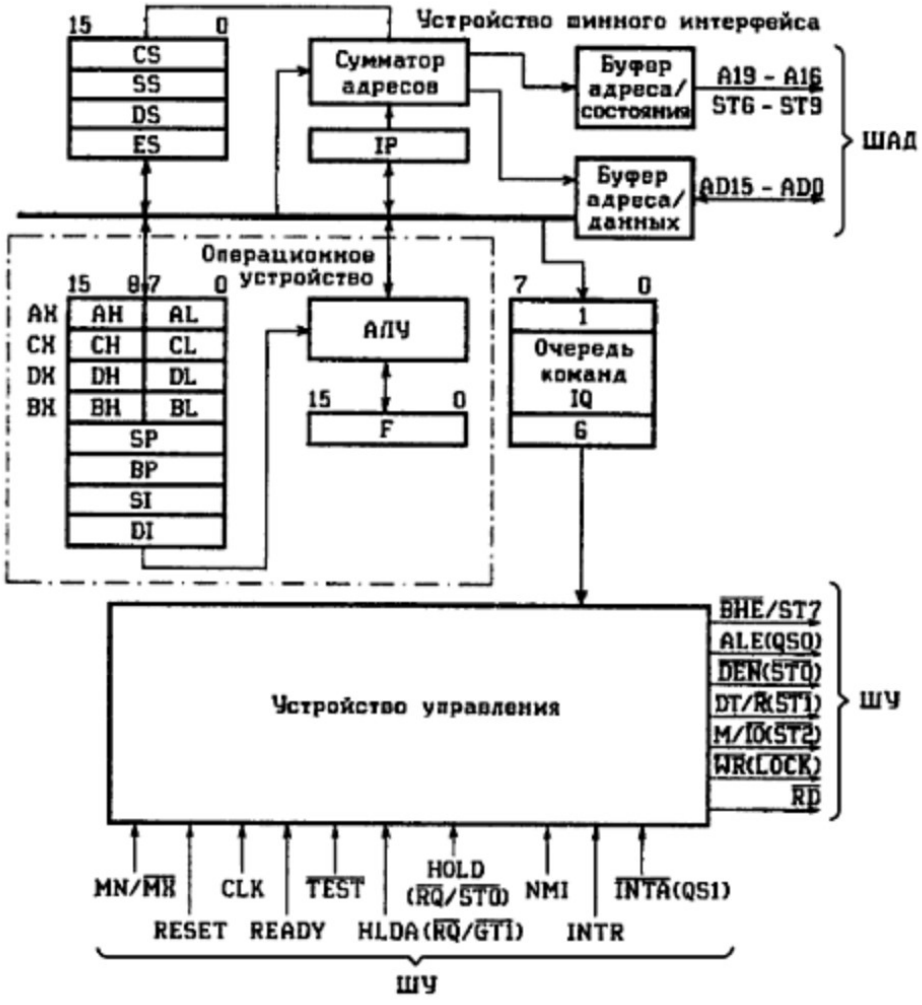


Рисунок 1.3 – Структура мікропроцесору i8086.

Мікропроцесор містить дві основні частини (рисунок 1.3): виконавчий блок (Execution Unit, EU) та блок шинного інтерфейсу (Bus Interface Unit, BIU). Обидва пристрої можуть працювати паралельно, що забезпечує суміщення у часі процесів вибірки та виконання команд. Виконавчий блок містить регістри загального призначення, арифметико-логічний блок (АЛУ, ALU), регістр прапорців (F) та блок управління, що забезпечує виконання команд. Блок шинного інтерфейсу містить блок сегментних регістрів, покажчик команд (IP), суматор адреси, чергу команд, буфери, що забезпечують зв'язок з шиною і призначені для виконання (функцій, пов'язаних з вибіркою операндів, встановленням черговості команд та формування адрес операндів та команд).

Мікропроцесор має такі програмно доступні регістри:

- AX(AH/AL) – регістр-акумулятор;
- BX(BH/BL) – регістр індексу базової адреси;
- CX(CH/CL) – регістр-лічильник;
- DX(DH/DL) – регістр даних;
- SP – регістр-покажчик стека;
- BP – регістр-покажчик базової адреси;
- SI – індекс джерела;
- DI – індекс приймача;
- CS – регістр сегмента коду;
- DS – регістр сегмента даних;
- SS – регістр сегмента стека;
- ES – регістр додаткового сегмента даних.

Деякі з них дозволяли здійснювати доступ окремо до старших та молодшим восьми біт. Коли була представлена 32-бітна архітектура 80386, регістри були просто розширені до 32 біт. Розширені регістри називаються EAX, ECX, EDX, EBX, ESP, EBP, ESI та EDI. Ці вісім регістрів можна, за деяким винятком, вважати регістрами загального призначення. Деякі

команди не можуть використовувати декілька з них. Інші команди завжди записують результат у певні регістри. Також як регістр показнику стеку в MIPS, регістр ESP, як правило, використовується як покажчик стека.

Регістр прапорців містить такі прапори:

- CF (Carry Flag) – прапор перенесення;
- PF (Parity Flag) – прапор паритету;
- AF (Auxiliary Flag) – допоміжний прапор переносу;
- ZF (Zero Flag) – прапор нуля;
- SF (Sign Flag) – прапор знака;
- OF (Overflow Flag) – прапор переповнення;
- TF (Trap Flag) – прапор трасування;
- IF (Interrupt Flag) – прапор переривання:
- DF (Direction Flag) – прапор напряму.

Кодування команд x86 – це тяжка спадщина десятиліть поступових змін. На відміну від MIPS, де команди завжди мають довжину 32 біти, довжина команди x86 може становити від 1 до 15 байтів.

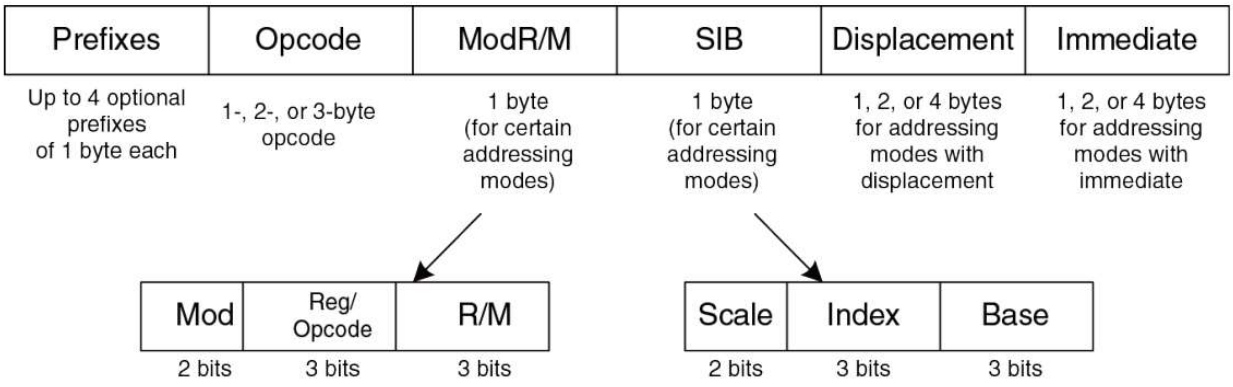


Рисунок 1.4 – Кодування команд x86.

Код операції (Opcode) може становити 1, 2 або 3 байти. Далі слідують чотири додаткові поля: ModR/M, SIB, Displacement та Immediate. Поле ModR/M визначає режим адресації. Поле SIB визначає коефіцієнт масштабування (scale), індексний (index) та базовий (base) регістри у деяких режимах адресації. Поле Displacement містить 1-, 2- або 4-байтове зміщення,

використовується у відповідних режимах адресації. Поле Immediate містить 1-, 2- або 4-байтову константу для команд, що використовують безпосередній операнд. Більше того, команда може мати до чотирьох однобайтних префіксів, що змінюють її поведінку. Однобайтове поле ModR/M використовує 2-бітне поле режиму Mod 3-бітове поле R/M для завдання режиму адресації одного з операндів. Операнд може бути в одному з восьми регістрів, або його можна вказати за допомогою одного з 24 режимів адресації пам'яті. Через помилки в кодуванні регістри ESP та EBP не можуть використовуватися як базовий або індексний регістри деяких режимах адресації. У полі Reg вказується регістр, який використовується як другий операнд. Для деяких команд, які не мають другого операнда, поле Reg використовується для зберігання трьох додаткових біт коду операції. У режимах адресації, які використовують регістр масштабованого індексу, байт SIB визначає індексний регістр та коефіцієнт масштабування (1, 2, 4 чи 8). Якщо під час адресації використовуються базова адреса та індекс, то SIB також визначає регістр базового адреси.

Розглянемо сучасний процесор, що містить архітектуру x86. У якості прикладу було вибрано архітектуру Intel Alder Lake.

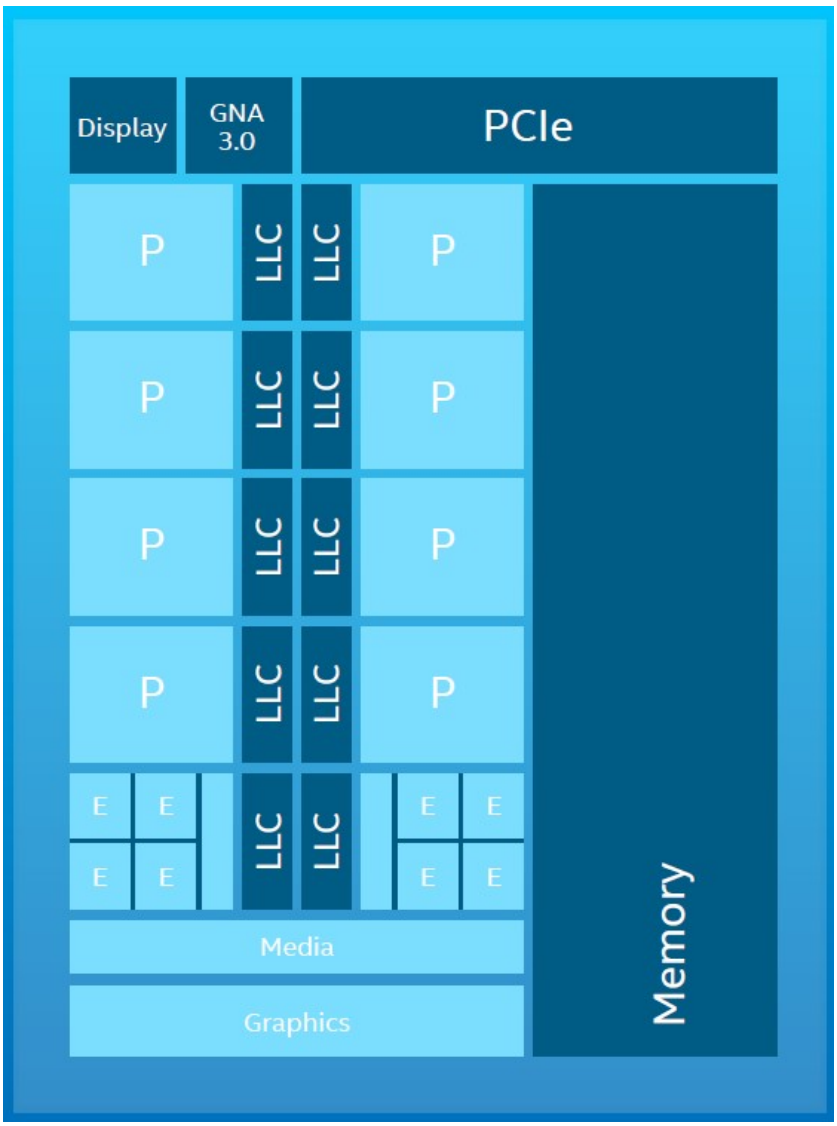


Рисунок 1.4 – Архітектура Intel Alder Lake.

В архітектурі присутні 2 види ядер: продуктивні (P) та енергоефективні (E). Такий розділ ядер дозволяє запускати операційну систему більш оптимально з точки зору споживання енергії. Об’єднує ядра кеш останнього рівня (last-level cache, LLC). Архітектура продуктивних ядер називається Golden Cove, а енергоефективних Gracemon. Розглянемо для прикладу архітектуру продуктивних ядер Golden Cove.

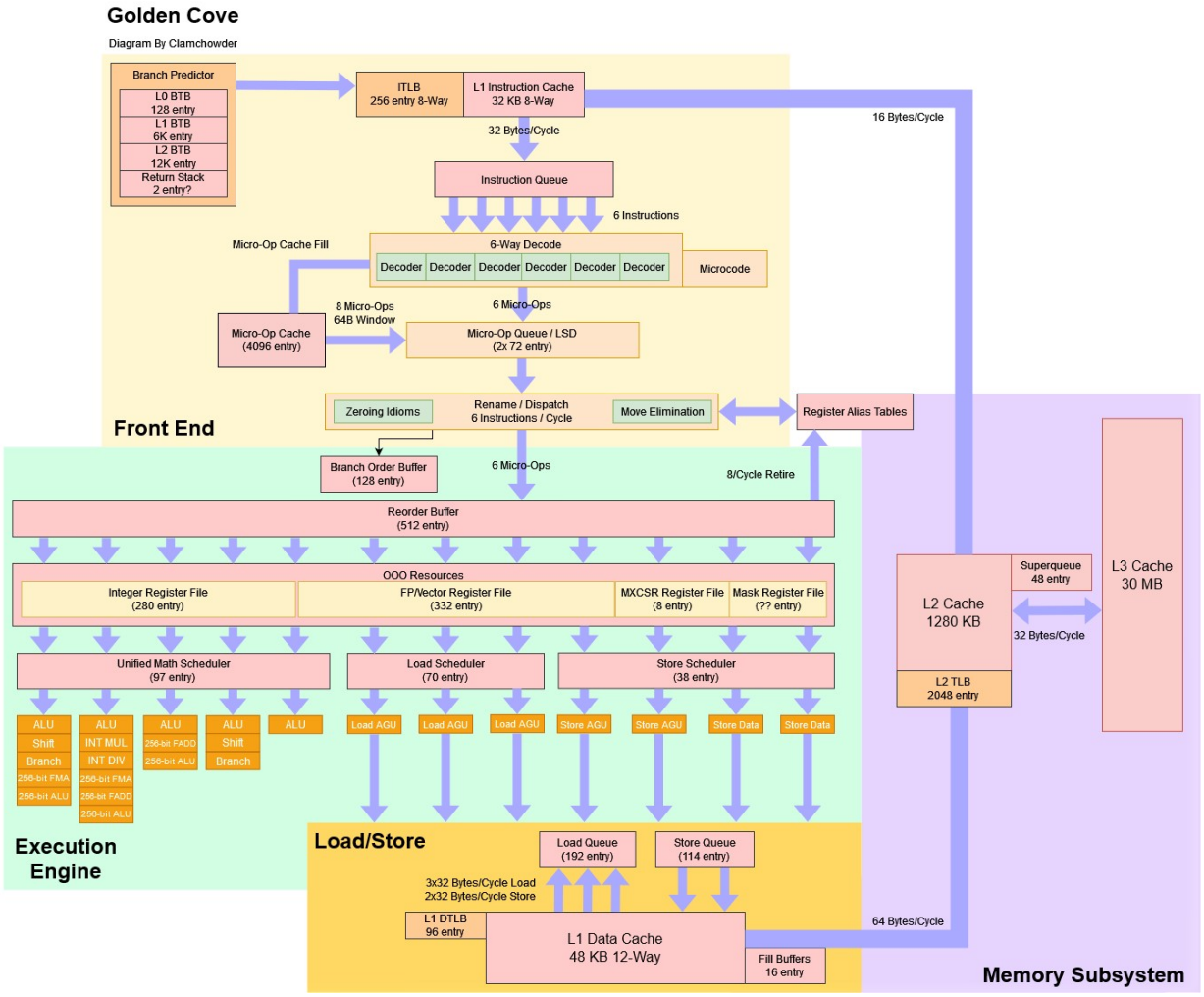
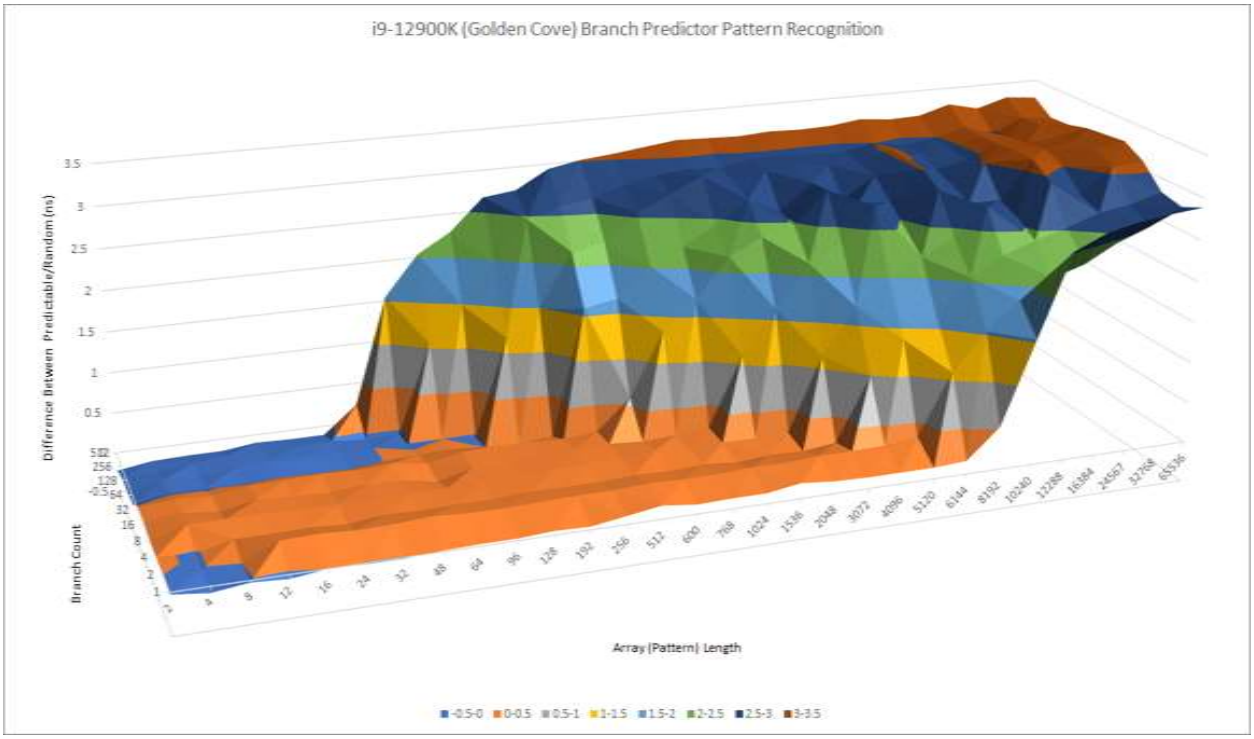


Рисунок 1.5 – архітектура ядра Golden Cove.

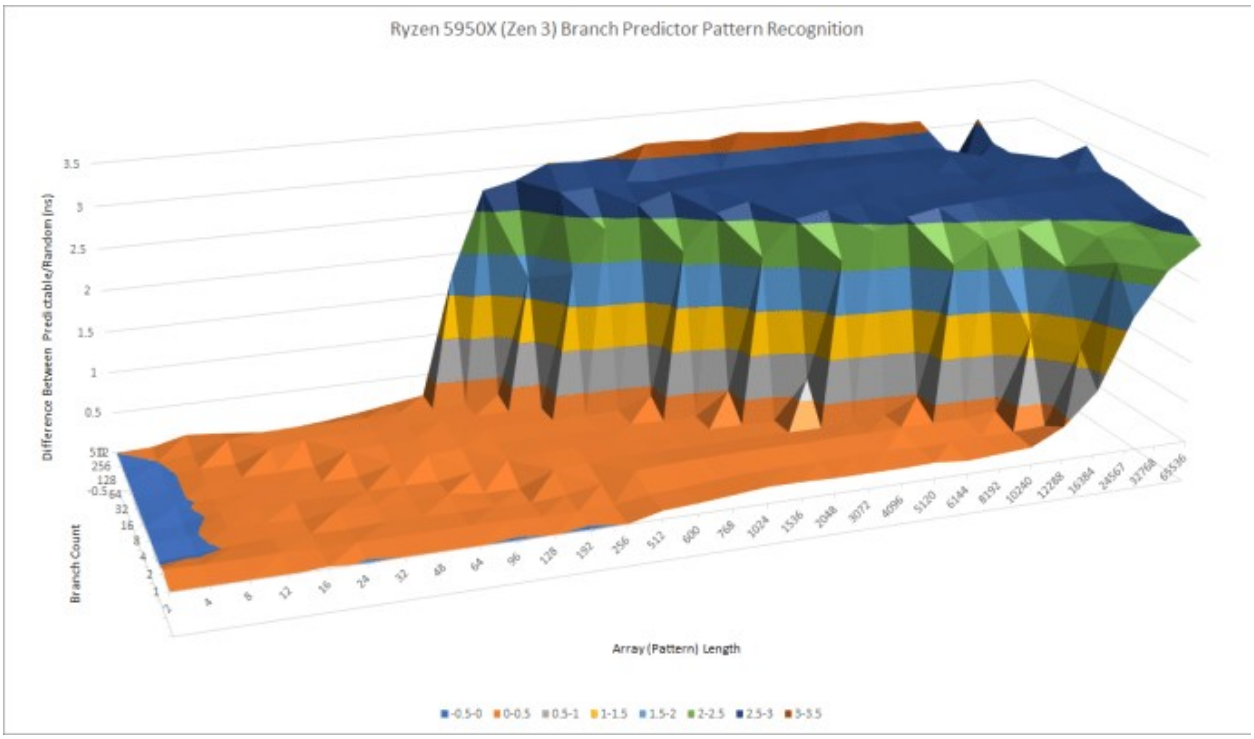
Конвеєр Golden Cove вже більше схожий на MIPS. Перші стадії, а саме вилучення та декодування представлені єдиним блоком Front End. Далі йде потужна виконавча стадія, представлена виконавчим двигуном (Execution engine). Стадії Load/Store вже пов’язані з кешем L1.

Окрім збільшеного розміру самого ядра, головною відмінністю є налічення черг та планувальників черговості виконання. Все це розраховано на роботу з надвеликою кількістю потоків та на часте перемикання контексту. А також дана архітектура має потужний механізм передбачування гілкування програми, щоб мінімізувати пузирі в конвеєрі (ситуація, коли конвеєр зупиняється і стадію простоюють). Отже вся архітектура націлена на обробку великої кількості потоків з мінімізацією простоїв.

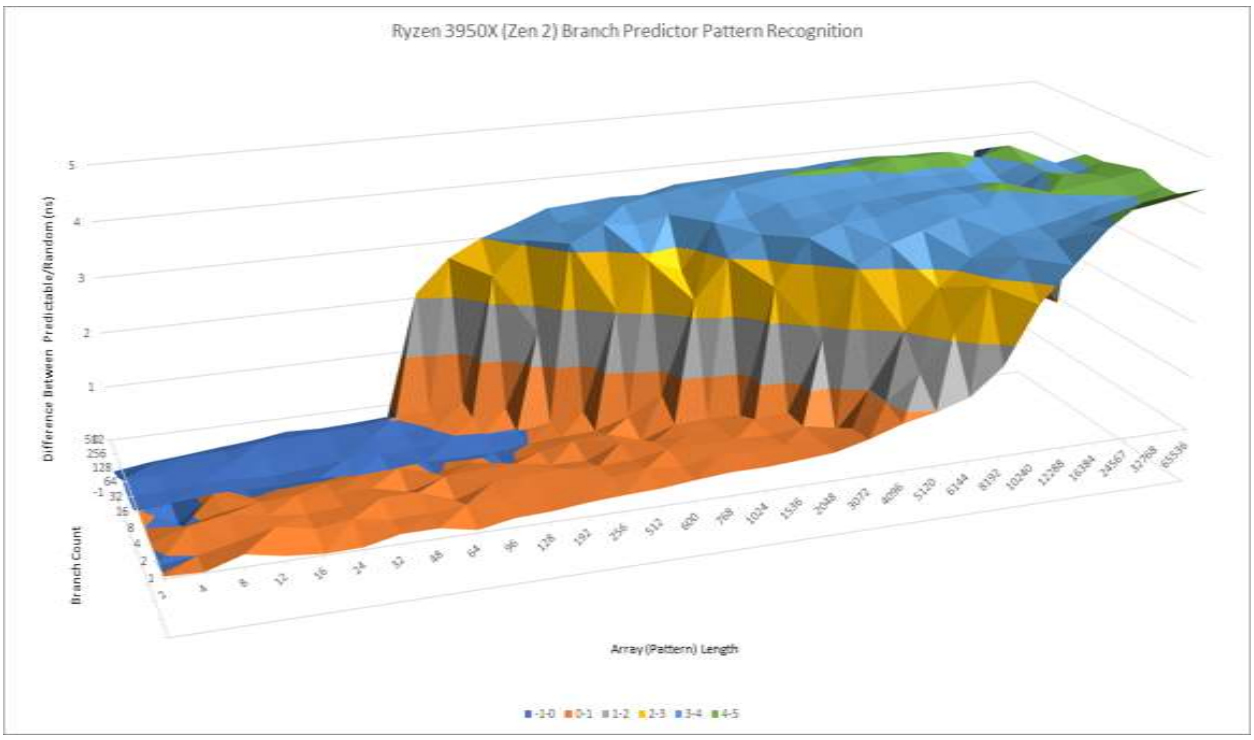
Розглянемо можливість предикації гілок в порівнянні з архітектурами конкурента AMD Zen3 та Zen2.



а)



б)



в)

Рисунок 1.6а – результат предикації Golden Cove; 1.6б – результат предикації для Zen3; 1.6в – результат предикації для Zen2.

По вісі 0x на рисунку 1.6 представлена довжина патерну, який представлений у вигляді довжини масиву даних; по вісі 0y представлений лічильник гілок; по вісі 0z представлене відношення між часом виконання предикованого патерну та довільного у наносекундах (чим менше, тим краще). Zen 3 може розпізнавати довші патерни, ніж Golden Cove. Порівняно з Zen 2, Golden Cove приблизно рівний, коли задіяна одна гілка (без урахування величезного штрафу перезапису даних при предикації L2 TAGE для Zen 2). Обидві архітектури AMD можуть краще впоратися з безліччю гілок. З 512 розгалуженнями в циклі Zen 2 і Zen 3 можуть добре триматися, коли повторюваний патерн має довжину 64 або 96 відповідно. Golden Cove страждає, коли довжина патерну перевищує 48.

Цільовий буфер гілки (branch target buffer (BTB)) Golden Cove має три рівні. Промах на одному рівні та попадання на наступному тягне за собою 1 цикл штрафу, що досить швидко порівняно з 3 циклами AMD для попадання L2 BTB. І як Sunny Cove (у формі Rocket Lake, а не Ice Lake з невідомих

причин), Golden Cove може розгортати цикли в межах своєї черги uop, надаючи йому неймовірну пропускну здатність двох взятих розгалужень за цикл. У певному сенсі черга uop від Intel діє як крихітний кеш трасування, зменшуючи накладні витрати на розгалуження для невеликих циклів без необхідності розгортання циклу від розробника чи компілятора. Проте в деяких випадках Zen 3 все ще може мати перевагу у швидкості. Він може відстежувати до 1024 гілок і обробляти їх один за одним без втрачених циклів між ними. Golden Cove може обслуговувати 128 гілок із пропускною здатністю приблизно 1 за цикл. Це еквівалентно здатності Haswell, але є регресією порівняно з Sunny Cove. Intel, ймовірно, пішла на компроміси, щоб такий великий BTB працював на частоті вище 5 ГГц.

1.3 Спеціалізовані процесори

В якості процесорів спеціального призначення було обрано Tensor Processing Unit (TPU) від Google та загальновживаний Stream Processor (SM), який і по сьогоднішній день використовується у графічних прискорювачах.

Походження TPU, архітектура та реалізація. Ще в 2006 році компанія Google розглядала можливість розгортання у своїх центрах обробки даних графічних процесорів, програмованих вентильних матриць (FPGA) або спеціалізованих інтегральних схем (ASIC). Висновок полягав у тому, що ті небагато програм, які могли працювати на спеціальному устаткуванні, можна було зробити практично безкоштовно, використовуючи надмірні потужності великих центрів обробки даних Google, а покращити безкоштовно складно. Ситуація змінилася в 2013 році, коли прогноз, згідно з яким користувачі Google виконували голосовий пошук протягом трьох хвилин на день за допомогою глибинних нейронних мереж (deep neural network, DNN) з розпізнаванням мови, подвоїла обчислювальні потреби центрів обробки даних Google, які були б дуже дорогими при використанні звичайних процесорів. Таким чином, Google запустив високопріоритетний проект

швидкого виробництва спеціалізованого чіпа для логічних висновків і закупив готові графічні процесори для навчання. Мета полягала в тому, щоб покращити співвідношення ціни та якості у 10 разів. Враховуючи цей мандат, TPU було спроектовано, перевірено, побудовано та розгорнуто в центрах обробки даних Google всього за 15 місяців.

Щоб знизити ризик затримки розгортання, інженери Google розробили TPU як співпроцесор на шині вводу-виводу, а не тісно інтегрований з центральним процесором, що дозволяє йому підключатися до існуючих серверів так само, як це робить графічний процесор. Крім того, щоб спростити проектування та налагодження обладнання, хост-сервер відправляє інструкції TPU для виконання, а не витягує їх сам. Таким чином, TPU ближче за духом до співпроцесора з плаваючою комою (FPU), ніж до GPU.

Інженери Google оптимізували дизайн із погляду системи. Щоб зменшити взаємодію із центральним процесором, TPU запускає цілі моделі логічного висновку, але при цьому пропонує гнучкість для відповідності DNN 2015 року та пізніших версій, не обмежуючи фокус DNN 2013 року.

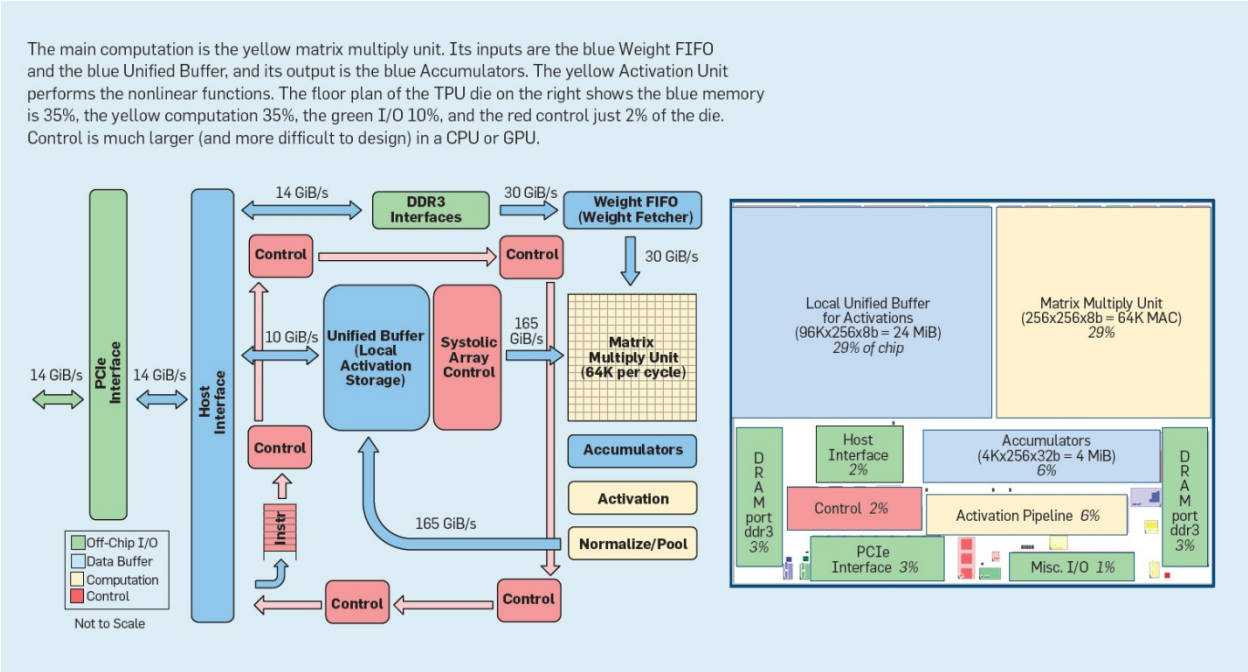


Рисунок 1.7 – блок-схема TPU та її топологія.

На рисунку 1.7 представлена блок-схема TPU. Інструкції TPU відправляються з хоста по шині периферійних компонентів PCI-E Gen3 x16 в буфер інструкцій. Внутрішні блоки зазвичай з'єднуються між собою шляхами завширшки 256 байт. Починаючи з правого верхнього кута, блок матричного множення є серцем TPU з MAC-адресами 256 × 256, які можуть виконувати 8-бітне множення та додавання цілих чисел зі знаком або без знака. 16-бітові продукти збираються у чотирьох мегабайтах 32-бітових акумуляторів під матричним блоком. Чотири мегабайти є 4096 256-елементних 32-розрядних акумуляторів. Матричний блок виготовляє одну часткову суму з 256 елементів за цикл.

Зважені коефіцієнти для матричного блоку розподіляються через вбудований у чіп «Ваговий FIFO», який зчитується із зовнішньої 8-гігабайтної DRAM, яку названо «пам'яттю вагів»; з логічних міркувань, ваги доступні лише читання; 8 гігабайт підтримує багато одночасно активних моделей. Ваговий FIFO має глибину в 4 буфери. Проміжні результати зберігаються у вбудованому «уніфікованому буфері» об'ємом 24 MiB, який може надсилати вхідні дані для матричного блоку. Програмований контролер прямого доступу до пам'яті (DMA) передає дані або до пам'яті хоста центрального процесору, або до уніфікованого буферу. Щоб мати можливість надійного розгортання в масштабі Google, внутрішня та зовнішня пам'яті включають у себе вбудовану апаратну частину для виявлення та виправлення помилок.

Філософія мікроархітектури TPU полягає у тому, щоб матричний блок залишався зайнятим. З цією метою інструкція, яка зчитує ваги, слідує філософії розв'язаного доступу/виконання у тому сенсі, що вона може завершитися після відправки своєї адреси, але до того, як вага буде обрана з пам'яті ваг. Матричний блок зупиниться, якщо активація входу або дані про вагу не готові.

Оскільки читання великої статичної пам'яті з довільним доступом (SRAM) вимагає набагато більше енергії, ніж арифметичні операції, матричний модуль використовує “систолічне виконання” для економії енергії за рахунок скорочення операцій читання та запису в уніфікованому буфері. осередки в масиві через рівні проміжки часу, де вони об'єднані. Задана операція множення вектор-матриці з 65536 елементів проходить через матрицю як діагональний хвильовий фронт. Ваги попередньо завантажуються і вступають в силу з хвилею, що настає, разом з першими даними нового блоку. Управління та дані конвеєризовані, щоб дати програмісту ілюзію того, що 256 вхідних даних зчитуються одночасно і миттєво оновлюють одне місце кожного з 256 акумуляторів. З погляду коректності програмне забезпечення не знає про систолічну природу одиниці матриці, але для продуктивності має враховувати затримку одиниці.

Програмний стек TPU має бути сумісним зі стеком, розробленим для CPU та GPU, щоб програми можна було швидко переносити на TPU. Частина програми, що працює на TPU, зазвичай пишеться на TensorFlow і компілюється в API, який може працювати на GPU або TPU.

Тепер розглянемо порівняння TPU від Google з представниками класичних процесорів та відеокарт. В якості центрального процесору було взято архітектуру від Intel Haswell, в якості відеокарти була взята модель від NVidia Tesla K80.

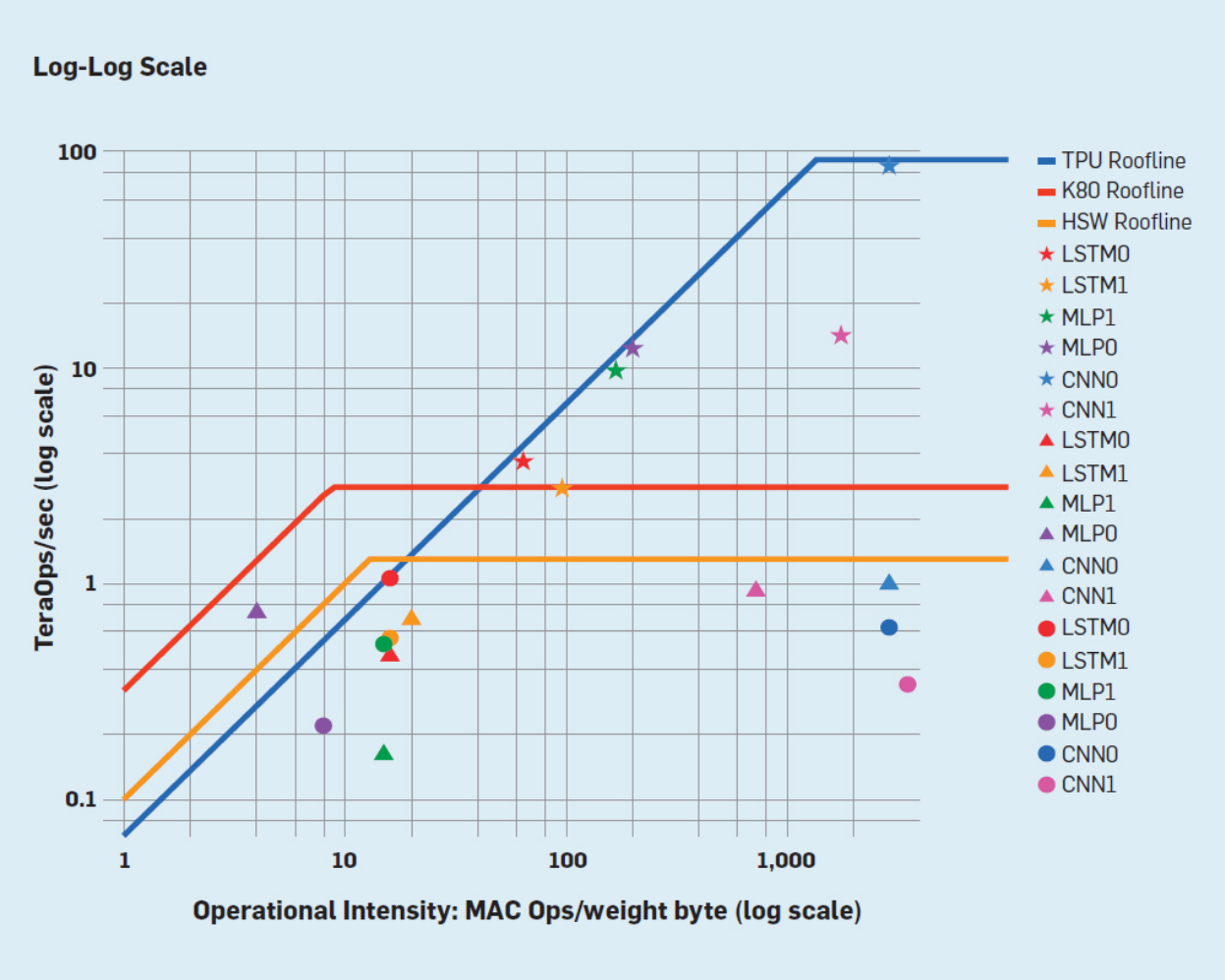


Рисунок 1.8 – Roofline графіки TPU, CPU та GPU.

На рисунку 1.8 показано моделі Roofline для кристалів TPU, CPU та GPU у логарифмічній шкалі. Зірки для TPU, трикутники для K80 і кола для Haswell; усі зірки TPU знаходяться на двох інших лініях вгорі або вище. TPU має довгу «похилу» частину Roofline, де інтенсивність роботи означає, що продуктивність обмежена пропускною здатністю пам’яті, а не піковими обчисленнями. П'ять із шести додатків “б'ються головою об стелю”; багатошаровий персептрон (MultiLayer Perceptron, MLP) та мережа довгої короткострокової пам’яті (long short-term memory, LSTM) прив’язані до пам’яті, а згорткові нейроні мережі (convolution neural network, CNN) – до обчислень.

Шість додатків DNN, як правило, нижчі за межі для Haswell і K80, в порівнянні з TPU. Причиною є час відгуку. Багато з цих програм DNN є частинами служб, орієнтованих на кінцевих користувачів. Дослідники

продемонстрували, що навіть незначне збільшення часу відповіді змушує клієнтів менше використовувати послугу. Хоча навчання може не мати жорстких термінів часу відповіді, логічний висновок зазвичай має, або надає перевагу затримці над пропускною здатністю.

Наприклад, обмеження часу відповіді 99-го процентиля для MLP0 становило 7 мс, як того вимагає розробник програми. (Висновки за секунду та затримка 7 мс включають час хоста сервера, а також час прискорення.) Haswell і K80 працюють лише на 42% і 37% відповідно від найвищої пропускної здатності, доступної для MLP0, якщо обмеження часу відповіді пом'якшено. Ці обмеження також впливають на TPU, але при 80% працюють набагато ближче до найбільшої пропускної здатності TPU MLP0. Порівняно з процесорами та графічними процесорами, однопотоковий TPU не має жодної складної мікроархітектури, яка витрачає транзистори та споживає енергію для покращення середнього випадку, але не 99-го процентиля; тобто відсутні кеші, предикація гілок, виконання поза порядком, багатопроцесорна обробка, спекулятивна попередня вибірка, об'єднання адрес, багатопотоковість, перемикання контексту тощо. Мінімалізм є перевагою доменно-орієнтованих процесорів.

Таблиця 1.2. Продуктивність графічного процесора K80 і кристала TPU відносно центрального процесора для робочого навантаження DNN.

	DNN		LSTM		CNN		Weighted
Type	0	1	0	1	0	1	Mean
GPU	2.5	0.3	0.4	1.2	1.6	2.7	1.9
TPU	41.0	18.5	3.5	1.2	40.3	71.0	29.2
Ratio	16.7	60.0	8.0	1.0	25.4	26.3	15.3

У таблиці 1.2 наведено кінцевий рядок відносної продуктивності висновку на матрицю, включаючи накладні витрати хост-сервера для двох прискорювачів проти центрального процесора, показуючи середньозважену відносну продуктивність для шести додатків DNN, припускаючи, що матриця

K80 в 1,9 рази перевищує швидкість. матриці Haswell, що матриця TPU у 29,2 рази швидша, а отже, матриця TPU у 15,3 рази швидша за матрицю GPU.

Тепер порівняємо споживання енергії цими трьома процесорами.

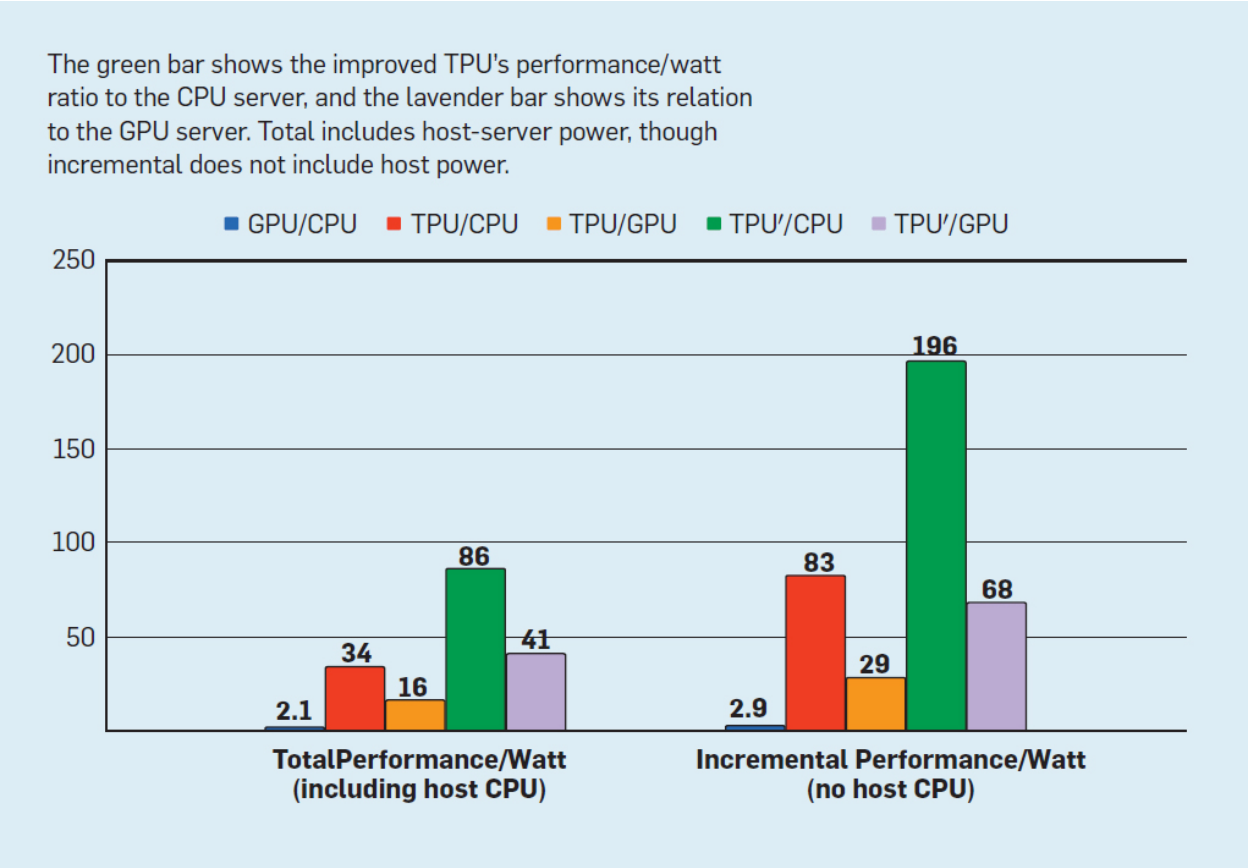


Рисунок 1.9 – Відносна продуктивність на ват (TDP) сервера GPU (синя смужка) і сервера TPU (червона смуга) до сервера CPU і сервера TPU до сервера GPU (помаранчева смуга). TPU' — це вдосконалений TPU, який використовує пам'ять GDDR5 K80.

На рисунку 1.9 наведено середню продуктивність/Вт для графічного процесору K80 і TPU відносно процесора Haswell. Представлено два різні розрахунки продуктивності/Вт. Перша «загальна» включає потужність, споживану сервером центрального процесора під час розрахунку продуктивності/Вт для GPU та TPU. Другий «інкрементний» віднімає потужність сервера центрального процесора від графічного процесора та процесора. За сумарною продуктивністю на ват сервер K80 у 2,1 рази кращий, ніж Haswell. Для додаткової продуктивності на ват, якщо не врахувати потужність сервера Haswell, сервер K80 у 2,9 разів перевищує

потужність сервера Haswell. Сервер TPU забезпечує в 34 рази кращу загальну продуктивність на ват, ніж Haswell, що робить сервер TPU в 16 разів більшим за продуктивність на ват в порівнянні з сервером K80. Відносна додаткова продуктивність/WattGoogle обґрунтовує спеціальний ASIC 83 для TPU, таким чином підвищуючи продуктивність TPU до 29 × продуктивності/Watt GPU.

Архітектура Stream processors. Щоб проілюструвати модель потокової архітектури, розглянемо потоковий кодувальник інтракодованого кадру (I-кадру) кодера MPEG2.

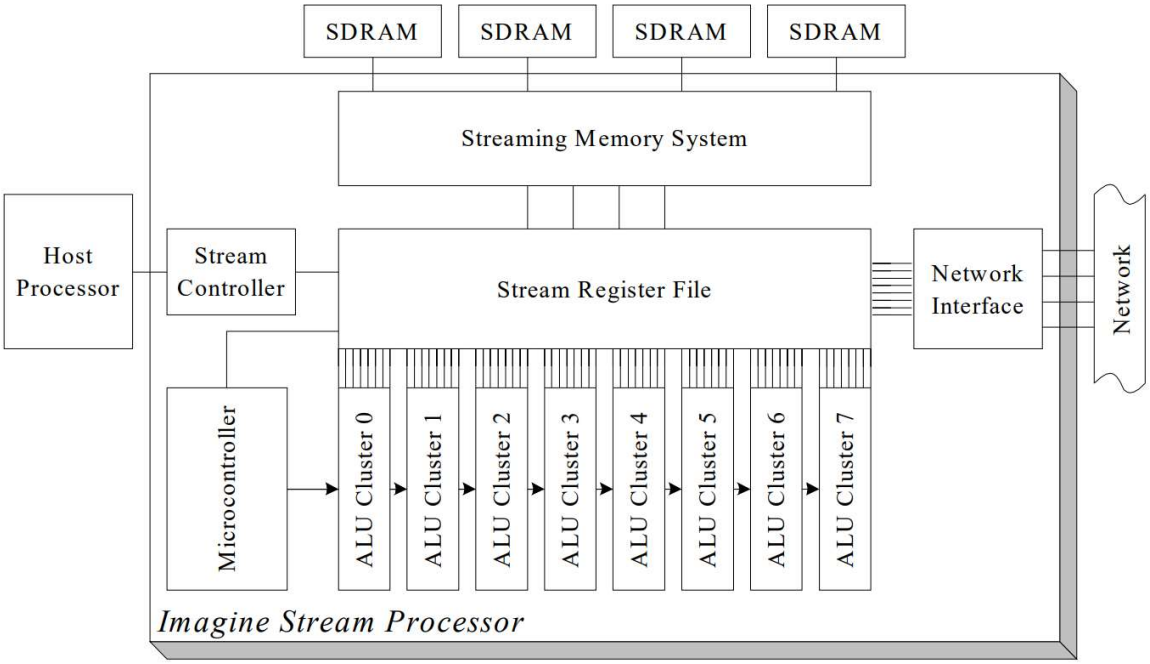


Рисунок 1.10 – Imagine Stream Processor.

Цей кодувальник I-кадрів (рисунок 1.10) ілюструє три ключові властивості додатків для обробки мультимедіа, згаданих раніше: дуже мало даних використовується повторно, багато паралелізму даних та потрібно багато операцій для обробки одного блоку даних. По-перше, дані рідко використовуються повторно в при кодуванні; наприклад, після перетворення кольору вихідне вхідне зображення більше ніколи не використовується. По-друге, більшість обчислень у кодері виконуються паралельно. Якби було достатньо обчислювальних блоків, всі фрагменти зображення могли б бути

конвертовані одночасно і паралельно, оскільки немає залежності одного фрагмента від іншого. Врешті, припускаючи, що лише оригінальне зображення, остаточний закодований бітовий потік та референсне зображення зберігаються у пам'яті, цей кодувальник I-кадрів виконуватиме 49,5 арифметичних операцій на звернення до пам'яті.

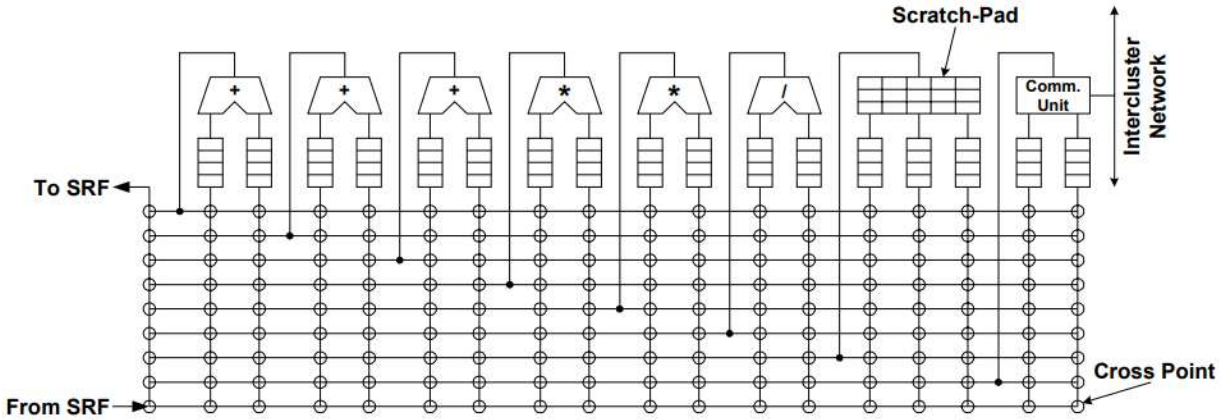


Рисунок 1.11 – обчислювальні кластери Imagine Stream Processor.

Imagine Stream Processor містить 8 обчислювальних кластерів, керованих одним мікроконтролером у SIMD режимі. Арифметичні кластери працюють одночасно з даними, що чергуються. записи, переданими з SRF, що дозволяє обробляти 8 записів даних паралельно. Як показано на рисунку 8 кожен арифметичний кластер містить 3 суматора, 2 помножувачі, і 1 блок поділу/квадратного кореня, одну «блокнотну» пам'ять на 256 записів і один блок міжкластерного зв'язку. Ці пристрої контролюються статично запланованими VLIW інструкціями, що видаються мікроконтролером. Через 8 арифметичних кластерів, 48 арифметичних блоків Imagine Stream Processor забезпечують потрібні високі та стійкі швидкості обчислень необхідні для обробки мультимедіа. Поєднання арифметичних блоків в Imagine Stream Processor відповідає вимогам цільових мультимедійних програм, але архітектурна концепція сумісна з іншими поєднаннями арифметичних блоків усередині кластерів.

Усі арифметичні пристрої підтримують як операції над 32-розрядними числами з плаваючою комою одинарної точності, так і операції над 32-

розрядними цілочисленними значеннями. Крім того, суматори та помножувачі також підтримують 16-бітні та 8-бітові паралельні операції з напівсловами для підмножини цілих операцій. Ці арифметичні операції добре підходять для низькоточних типів даних, які зазвичай зустрічаються в програмах для обробки мультимедіа. Ці інструкції з паралельними напівсловами, подібно до мультимедійних інструкції, зазвичай додаються в процесори загального призначення [Lee96] [PW96] [TONH96], використовують дрібнозернистий паралелізм даних, властивий мультимедійним додаткам. Суматори, помножувачі, блокнотна пам’ять та блок міжкластерного зв'язку повністю конвеєризовані, що дозволяє виконувати нову операцію у кожному циклі. Блок поділу/квадратичного кореня має два ядра SRT, тому «на льоту» може бути не більше двох операцій поділу або квадратного кореня будь-якої миті часу.

1.4 Висновок

Отже, в даному розділі були розглянуті та порівнянні універсальні та спеціалізовані комп’ютерні архітектури. Було розглянуто як класичні рішення, так і сучасні.

Універсальні архітектури поділяються на RISC та CISC. Основна відмінність між ними в розмірі інструкцій та підході до реалізації потенціалу. Так, RISC процесори мають обмежений, проте повний потенціал; а їхні інструкції мають фіксовану довжину. CISC процесори мають інструкції різної довжини, а також мають засоби прискорення певних операцій. Сучасні універсальні процесори проектуються так, щоб мінімізувати простої в роботі. До таких засобів належить продвинутий передбачувати гілкування інструкцій та буфери для виконання інструкцій не по черзі (out-of-order).

Спеціалізовані процесори не мають такої повноти, як універсальні, проте здатні швидко виконувати конкретні операції. Для обробників великих масивів даних характерне надвелике розпараллелювання обчислень, щоб

конвеєрно виконувати багато одноманітних операцій. При цьому, вони можуть мати апаратні прискорювачі певних дій, важливих для їх типів даних. Так, наприклад, тензорний прискорювач від компанії Google має FIFO буфер для зважувальних коефіцієнтів. Також було розглянуто принципи, покладені в архітектурі Streaming Mulitprocessor на прикладі кодувальника інтракодованого кадру. Так, архітектура будується на принципі мінімального повторного використання попередніх даних. Так, між TPU та SM наявні принципові відмінності. Її суть полягає в тому, що SM не використовує дані повторно, коли як TPU має спеціальний буфер для повторного використання зважувальних регістрів.

Для реляційного процесору питання повноти в рамках обранного типу даних є вкрай важливою. Тому у ньому зібрано принципи як від універсальних, так і від спеціалізованих процесорів. Так, від універсальних процесорів реляційному дістався механізм роботи з інструкціями, зокрема механізму стрибку між інструкціями. Це забезпечує повноцінну підтримку сигнатурних операцій примітивно-програмної алгебри. Від спеціалізованих процесорів реляційному дістався принцип роботи з даними. Таким чином забезпечуючи високу продуктивність в обчисленнях даних. При цьому, реляційний процесор використовує як принципи від SM, так і від матричного прискорювача.

Розділ 2. Алгебраїчна характеристика реляційної алгебри

У минулій роботі [1] було розглянуто частково рекурсивні функції та частково рекурсивні предикати (далі чр-функції та чр-предикати) з використанням сигнатурних операцій примітивної програмної алгебри (далі ППА) над структурними типами даних типу вектора, моделювання матричних чр-функцій та чр-предикатів векторними чр-функціями та чр-предикатами, та структурологічне збагачення складних типів даних іменною моделлю даних. В цій же роботі буде проведено аналогічне дослідження, проте вже реляцій та реляць. Дані концепції є стержнем релятивійських баз даних, тому їхнє вивчення є не менш значущим.

Як і в минулій роботі, тут для дослідження використовується ППА через її потужний потенціал, простоту використання та поширеність. Строгі визначення сигнатурних операцій ППА дані у [4]. Наведемо їх повне визначення тут, оскільки вони є вкрай необхідними.

Носій ППА становлять багатомісні функції і предикати (далі - функції і предикати) ($n = 1,2,\cdots$). Сигнатуру ж ППА (позначається як Ω) складають операції суперпозиції, галуження і циклування, як уточнення основних методів конструювання програм [2]. Для зручності і компактності викладу при позначенні функцій і предикатів будемо користуватися, як операторною, так і термальною формою запису [5]. Використання кожної з них в будь-якому конкретному випадку обумовлюється тим, що вони відображають принципово різні точки зору на одну і ту ж сутність, описом якої вони є. Операторну форму запису використовуватимемо, якщо важливо відобразити генезис описуваної функції, термальна ж буде використовуватись там, де важливо відобразити результат генезису. І хоча, як тексти ці форми, в певних сенсах, взаємозамінні, наведена розстановка акцентів важлива, тому що фіксує абсолютно природну домінанту генезису по відношенню до його результату.

Нехай задані m функцій $f_1,...,f_m$ однакової арності (наприклад, k) виду $A^k \to B$, визначені на деякій зафіксованій множині A, зі значеннями з множини B і нехай на множині B визначена m-арна функція f, що приймає значення з деякої множини C. Розглянемо нову k-арну функцію $g: A^k \to C$ таку, що її значення на аргументі $< a_1,..., a_k >$ задаються так: $g(< a_1,...,a_k >) \cong f(f_1(< a_1,...,a_k >),...,f_m(< a_1,...,a_k >)))$. Тоді кажуть, що функція g є результатом застосування $(m+1)$-арної суперпозиції S^{m+1} до кортежу функцій $< f, f_1,...,f_m >$, тобто $g = S^{m+1}(< f, f_1,...,f_m >)$.

Нехай тепер крім уже даних функцій $f_1,...,f_m$ додатково задані функція h того ж виду $A^k \to B$ і m-значна функція $\beta: B \to \{1,2,...,m\}$. Будемо говорити, що k-арна функція $g: A^k \to B$ виникає з функцій $h, f_1, \ldots, f_m$ за допомогою $(m+1)$-арної параметричної операції розгалуження $\Diamond_\beta^{m+1}$, якщо для довільного аргументу $< a_1,...,a_k > \in A^k$ значення функції $g(< a_1,...,a_k >)$ задається наступним чином: $g(< a_1,...,a_k >) \cong f_r(< a_1,...,a_k >)$, якщо $\beta(h(< a_1,...,a_k >)) = r$, $(1 \le r \le m)$.

Зауважимо, що так введена параметрична операція галуження представляє адекватне уточнення відомого методу конструювання програм типу $case\_of$. Корисним окремим випадком цієї операції, очевидно, є тернарна операція галуження $\Diamond$, що зіставляє двом функціям $f_i: A^k \to B, i = 1,2$ і одному предикату $p: A^k \to \{T, F\}$ k-арну функцію $g \equiv \Diamond(< p, f_1, f_2 >)$, значення якої на довільному аргументі $< a_1,...,a_k > \in A^k$ задається так:

$$g(< a_1,..., a_k >) \cong \begin{cases} f_1(< a_1,..., a_k >), p(< a_1,..., a_k >) \cong T \\ f_2(< a_1,..., a_k >), p(< a_1,..., a_k >) \cong F \end{cases}$$

Нарешті, нехай дано також k-арний предикат $p: A^k \to \{T, F\}$. Розглянемо k-арну функцію $g: A^k \to B$, значення якої $g(< a_1,...,a_k >)$ на довільному аргументі $< a_1,...,a_k > \in A^k$ вважається рівним першому компоненту першого кортежу

послідовності кортежів $[< a_1^i,...,a_k^i >]_{i=0,1,2,...}$, де $a_j^0 = a_j, j = 1,2,...,k$ і $a_j^{i+1} = f_j(< a_1^i,...,a_k^i >), j = 1,2,...,k$, для якого (позначимо його, наприклад, $< a_1^s,...,a_k^s >$) $p(< a_1^s,...,a_k^s >) = F$, за умови, що для всіх $r < s$, якщо такі є, значення $p(< a_1^r,...,a_k^r >) \cong T$. Функція g виникає застосуванням $(m+1)$-операції циклювання до функцій кортежу $< p, f_1,\ldots, f_m >$, тобто $g = *^{m+1}(< p, f_1,\ldots, f_m >)$. Таким чином, відповідно до сказаного, $g(< a_1,...,a_k >) = a_1^s$.

До цих пір для позначення введених операцій використовувалося виключно операційна форма запису. Використовуючи ж термальну форму запису операцій сигнатури ППА, будемо явно вказувати тільки ті змінні, значення яких істотно використовуються. Наприклад, стосовно операції циклування будемо використовувати запис вигляду: $p(x_1,\ldots,x_m)\underset{y_1\ldots y_k}{*} < f_1(z_1^1,\ldots,z_{m_1}^1),\ldots,f_k(z_1^k,\ldots,z_{m_k}^k) >$, вказуючи явно тільки ті змінні, від яких функції і предикат істотно залежать. При цьому функція $f_j(z_1^j,\ldots,z_{m_j}^j)$ «управляє» змінними y_j, $j = 1,2,...,k$, а змінна y_1 вважається «вихідною».

2.1 ППА чр-функцій та чр-предикатів над структурними типами даних

Розглянемо спочатку обчислюваність на множині векторів (кортежів) натуральних чисел довільної довжини, тобто $D \equiv N^* \equiv \bigcup_i N^i$, $i \in N \cup \{0\}$, $N^0 \equiv \{\Lambda\}$, $N^1 \equiv N$, $N^s \equiv \underbrace{N \times \ldots \times N}_{s}$, Λ - кортеж нульової довжини або так званий пустий кортеж.

Багатомісні функції векторного аргументу та значення назвемо векторними функціями; при їх запису в якості функціональних символів використовуємо великі літери $F, G, H, \ldots$ Векторні предикати визначаються аналогічно, а літери $P, Q, R, \ldots$ відіграватимуть роль предикатів символів. Малі прописні літери *x, y, z, ...*позначають змінні над *N**. Вектори позначимо *u, v, w, ...,*

натуральні числа – *a, b, c ...* . Нагадаємо важливий результат про повноту алгебри векторних чр-функцій та чр-предикатів.

Покладемо $\sigma_{N^*} \equiv \{\Pi, \circ, C_0, S, =_{N^*}, I_m^n\}$, де $\circ$ – позначає собою конкатенацію кортежів. Запис виду $u_1 u_2 .. u_n$ означає вектор, отриманий конкатенацією векторів u_1, u_2, …, u_n у вказаному порядку; зокрема, $u = a_1 a_2 .. a_n$ – приклад розгорнутого запису вектора натуральних чисел. Через Π, C_0, S позначимо наступні векторні функції:

$\Pi(\Lambda) = S(\Lambda) = \Lambda$, $C_0(u) = 0$, $S(au) = (a+1)u$, Λ – «пустий» кортеж, а I_m^n – звичайна селекторна функція [5].

Справедливим є твердження.

Твердження. σ_{N^*} – I_m^n–базис алгебри чр-функцій та чр-предикатів над кортежами $A_{N^*}^{чр}$ [4].

Термін I_m^n–базис алгебри означає, що наведена сукупність функцій та предикатів не може бути скорочена без втрати базисності.

Спираючись на цей результат, розглянемо тепер алгебраїчну характеристику класу чр-функцій та чр-предикатів над матрицями.

Під матрицею (натуральних чисел) розміру $m \times n$, $(m, n = 1,2,...)$ розуміється відображення $\{1,...,m\} \times \{1,...,n\} \to N$. Множину всіх матриць розміру $m \times n$ позначимо $\mathrm{M}_{m,n}$, а Λ – означає «порожню» матрицю, по аналогії з уведеним вище «порожнім» кортежем. Покладемо $\mathrm{M} \equiv \{\Lambda\} \cup \bigcup_{m,n=1}^{\infty} \mathrm{M}_{m,n}$ – множина всіх матриць.

Матриці позначатимемо прописними буквами $A, B, C, ...$, їх елементи – відповідними рядковими буквами з індексами. Наприклад, $A = (a_{ij})_{mn}$ – позначення матриці розмірності $m \times n$, $[a_1,...,a_n]$ – рядкова і $\begin{bmatrix} a_1 \\ ... \\ a_n \end{bmatrix}$ – стовпчикова матриці. Матричні функції та предикати вводяться та позначаються

аналогічно до уведених вище векторних функцій та предикатів. Зокрема, у якості функціональних символів використовуватимемо букви $\Im, \aleph, \ldots$, у якості предикатних – $\Re, \mathrm{P}, \ldots$, відповідні змінні позначатимемо $\varsigma, \xi, \eta, \ldots$

Таким чином, уведенням матриць та класів чр-функцій Ф та чр-предикатів Р над ними отримуємо ППА матричних чр-функцій та чр-предикатів $A_M^{чр} = <\Omega, \Phi_M \cup \mathrm{P}_M>$.

Співставляючи деяким чином кожній матричній функції відповідну їй кортежну функцію та роблячи теж саме з предикатами отримуємо взаємно однозначне відображення $\Theta: \Phi_M \cup \mathrm{P}_M \to \Phi_{N^*} \cup \mathrm{P}_{N^*}$ Виходячи ж з того, що сигнатурами обох алгебр $A_{\mathrm{N}^*}^{чр}$ та $A_{\mathrm{M}}^{чр}$ є одні й ті самі операції суперпозиції, розгалуження та циклування, отримуємо просте, але дуже корисне твердження про ізоморфізм.

Твердження (про ізоморфізм ППА). Відображення Θ є ізоморфізмом ППА $A_{\mathrm{M}}^{чр}$ на ППА $A_{\mathrm{N}^*}^{чр}$.

Таким чином, залишається тільки наповнити цей ізоморфізм Θ конкретним змістом. Про його існування можна стверджувати через результати [9] про нумерації множини векторів та матриці. Окремі випадки таких нумерацій були побудовані, зокрема, у [10, 11, 12].

Виходячи з твердження про I_m^n–базис $A_{\mathrm{N}^*}^{чр}$, а також того, що при ізоморфізмі Θ селекторні функції переходять у селекторні, а повні системи уповні системи [5], то має місце наступне твердження.

Твердження. Існує I_m^n–базис алгебри $A_{\mathrm{M}}^{чр}$, що складається з точністю до селекторних функцій з чотирьох чр-чункцій та одного чр-предикату.

Це твердження дає принципову відповідь на питання про існування I_m^n–базису алгебри $A_{\mathrm{M}}^{чр}$. Однак, знаходження конкретного I_m^n–базису алгебри $A_{\mathrm{M}}^{чр}$ залишається поки відкритим. Його рішення залежить від того, яким чином буде реалізовано згаданий вище ізоморфізм ППА $A_{\mathrm{M}}^{чр}$ на ППА $A_{\mathrm{N}^*}^{чр}$. Саме цьому присвячено наступний підрозділ.

2.2 Ізоморфне відображення реляційної алгебри на векторну в рамках ППА

В данному підрозділі за допомогою методу ізоморфних відображень проводиться семантико-синтаксичне дослідження введеної вище ППА $A_{\text{M}}^{чр}$. Будується ізоморфне відображення ППА $A_{\text{M}}^{чр}$ на ППА $A_{\text{N}^*}^{чр}$. На основі твердження про повноту ППА $A_{\text{N}^*}^{чр}$ дається алгебраїчна характеристика класу матричних чр-функцій та чр-предикатів.

Ізоморфізм векторної алгебри над реляційною будується за принципом, що певний вектор можна розглядати як однострочну реляцію. Сам ізоморфізм базується на кодуючому та декодуючому відображеннях. Очевидно, що відображення однострочної реляції натуральних чисел на вектори є бієкційним, оскільки однострочна реляція фактично є множиною, що складається з одного кортежа. Позначимо вектори як $(a_1, \dots a_n)$, а кортежі як $< a_1, \dots a_k >$, де n— розмір вектору, k — розмір кортежу.

Введемо кодуюче відображення однострочних реляцій натуральних чисел на вектори $\psi_1 : R \to N^*$.

$$\psi_1(\{\Lambda\}) = \Delta, \psi_1(\{< a_1, \dots a_n >\}) = (a_1, \dots a_n)$$

А також декодуюче зображення $\chi_1 : N^* \to R$, що буде оберненою функцією до ψ_1. Тобто дана функція перетворює вектор у однострокову реляцію.

$$\chi_1(\Delta) = \{\Lambda\}, \chi_1((a_1, \dots a_n)) = \{< a_1, \dots a_n >\}$$

Таким чином, функції $\psi_1 та \chi_1$представляють пряме ізоморфне відображення однострокових реляцій на вектори. А отже для таких реляцій можна застосовувати операції над векторами, подаючи на вхід функції закодовану у вектор реляцію, та декодуючи її результат.

$\chi_1(f^{N^*}(\psi_1(R_1), \dots \psi_1(R_n)))$,

де f^{N^*} — n-арна векторна функція $f^{N^*} : N^* \to N^{*n}$. А це означає, що для таких реляцій можна використовувати векторну алгебру, повнота якої доказано в

[4]. Таким чином за допомогою ізоморфізму однострочних реляцій на вектори досягається повнота, проте саме для однострочних реляцій.

Однак, повнота лише для однострочних реляцій ще не означає повноту для всіх реляцій. Проте саме за допомогою повноти однострокових реляцій можна досягнути повноти для всіх реляцій. Для початку введемо функцію, яка перетворює будь-яку реляцію в однострочну із заздалегідь заданим кодуванням для оберненого переходу $\varphi: R \to R$.Найбільш очевидним способом є зберігання кількості стовпців (довжини кортежей). А оскільки цей показник є вкрай важливим, тому він буде записуватись у перший стовпчик.

$$\varphi(\{\Lambda\}) = \{\Lambda\}, \varphi(\{< a_1^i, \dots a_n^i > |\ i = \overrightarrow{1, m}\}) = \{< n, a_1^1, \dots a_n^1, \dots, a_1^m, \dots a_n^m >\}$$

Фактично реляція з m-вимірних кортежів витягується у довгий кортеж $n \times m + 1$, де на його початку вказано розмір строки. Це необхідно для декодування, тобто для оберненої дії.

Очевидно що φ — ін'єкція.Оскільки, в даному випадку, є 2 типи однострокових реляцій: закодовані та незакодовані. Закодованими вважаються такі однострокові реляції, перший елемент яких є кратним їхній довжині (звісно, віднявшии одиницю). З інших однострокових реляцій просто не вдасться побудувати багатострочні реляції. Також очевидно, що $\omega \rightleftharpoons \varphi(R)$рекурсивно (в нумерації$\alpha_{N^*}$).

Введемо дію декодування однострокової реляції в багатострокову $\Phi: R \to R$.

$$\Phi(\{\Lambda\}) = \Lambda, \Phi(\{< n, a_1^1, \dots a_n^1, \dots, a_1^m, \dots a_n^m >\}) = \{< a_1^i, \dots a_n^i > |\ i = \overrightarrow{1, m}\},$$

При цьому, якщо на вхід функції Φ подати не закодовану однострокову реляцію, то функція її ж і поверне.

$$\Phi(\{< k, a_1^1, \dots a_n^1, \dots, a_1^m, \dots a_n^m >\}) = \{< k, a_1^1, \dots a_n^1, \dots, a_1^m, \dots a_n^m >\}$$

Тепер можна ввести нові кодуючу та декодуючу функції, які працюють з усіма реляціями. Так, нова кодуюча функція $\psi(R) = \psi_1(\varphi(R))$вже перетворює багатострокові реляції у вектор. А функція

$\chi(N^*) = \Phi(\chi_1(N*))$ перетворює вектор у багатострокову реляцію.

Далі, необхідно розширити векторні базисні операції на реляційні. Принцип розширення є таким, що самі розширені функції для однострокових реляцій будуть працювати так само; а для багатострокових буде працювати за принципами операцій над реляціями.

Надалі ізоморфні функції над реляціями будуть позначатися нижнім індексом R, а всі ці операції призначатимутсья саме для закодованих у вектори реляції. Для початку розберемо операцію видалення першого рядка на одиницю $П$. Для вектору, а також однострокової реляції, дана операція видаляє перший елемент. А для багатострокової принцип дії є таким, що у кожної строки видаляється перший елемент.

$$П_R(\{< a_1^i, a_2^i, \dots a_n^i >| \ i = \overrightarrow{1, m}\}) = \{< a_2^i, \dots a_n^i >| \ i = \overrightarrow{1, m}\}$$

$П_R$ — функція видалення першого стовпчика.

$$S_R(\{< a_1^i, a_2^i, \dots a_n^i >| \ i = \overrightarrow{1, m}\}) = \{< a_1^i + 1, a_2^i, \dots a_n^i >| \ i = \overrightarrow{1, m}\}$$

S_R — функція збільшення всіх елементів першого стовпчика на одиницю.

$$C_{\{0\}}(\{< a_1^i, \dots a_n^i >| \ i = \overrightarrow{1, m}) = \{0\}$$

$C_{\{0\}}$ — функція співставлення будь-якій реляції нульової реляції (реляція з єдиним нульовим елементом).

$$\{< a_1^i, \dots a_n^i >| \ i = \overrightarrow{1, l}\} \circ_R \{< b_1^j, \dots b_m^j >| \ j = \overrightarrow{1, t}\} = \{< a_1^k, \dots a_n^k, b_1^k, \dots b_m^k >| \ k = \overrightarrow{1, l \times t}\}$$

$\circ_R$ — функція конкатенації двох реляцій, тобто декартовий добуток.

$$\{a_n^m\} =_R \{b_n^k\} = \begin{cases} T, \forall a \in \{a_n^m\}, a \in \{b_n^k\} \\ F, \textit{інакше} \end{cases}$$

$=_R: R^2 \to \mathbb{B}$ — предикатрівності двох реляцій. Порівняння реляцій на рівність відбувається за таким же принципом, що й порівняння множин, тобто якщо всі їхні елементи (в даному випадку кортежі) співпадають.

А також введемо обмежені на реляціях функції об'єднання $\cup_R$ та різниці$\setminus_R$множин. А також звичайну селекторну функціюю I_m^n.

Теорема. $\sigma_R \rightleftharpoons \{\Pi_R, S_R, C_{\{0\}}, \circ_R, =_R, \cup_r, \setminus_R, \psi_R, \chi_R, I_m^n\}$ — повна сукупність алгебри чр-функцій та чр-предикатів над реляціями $A_R^{чр}$.

2.3 Ізоморфне відображення реляційної алгебри на матриці в рамках ППА

Побудувати ізоморфне відображення реляційної алгебри на матричну вже не є таким тривіальним, як для векторної. Одна з головних відмінностей полягає у тому, що матриця є повністю впорядкована по всім вимірам (фактично можна вважати кортежем кортежів); тоді як реляція по строкам не є впорядкованою, оскільки вона є множиною, тобто множина кортежів. Розглянемо відмінність по впорядкованості. Допустимо, що існує реляція R розміром $m \times n$, а всі її атрибути є простими типами даних. Враховуючи вищезазначені визначення для відповідності реляцій до матриць, перенесемо дані з усіх комірок реляції R до матриці $A_{m \times n}$ відповідно до нумерації (тобто фактично скопіюємо дані з двовимірної реляції до матриці). А тепер створимо реляцію R', яка складається з тих же самих кортежів, що й R, тільки при записі поміняємо кортежи i-й та $i+1$-й місцями. Фактично, така перестановка кортежів нічого семантично не змінює, адже для усіх можливих операцій над реляціями R та R' результати будуть однаковими. При цьому мається на увазі що інформації про порядок розміщення кортежів у реляції немає. Тепер, аналогічно до матриці $A_{m \times n}$, перенесемо всі комірки з реляції R' до $A'_{m \times n}$. Хоч матриці $A_{m \times n}$ та $A'_{m \times n}$ утворились від рівних між собою реляції, проте самі матриці не є рівними.

$$\begin{cases} R = R' \\ A_{M \times N} \neq A'_{M \times N} \end{cases}$$

Для наглядності введемо просту кодуючу та декодуючу функції $\psi'(\cdot)$ та $\chi'(\cdot)$ відповідно. Проста кодуюча функція $\psi'(\cdot)$ перетвоює реляцію на матрицю шляхом звичайного переносу даних з рядків (кортежи) та стовпчиків (атрибути) до матриці. Проста декодуюча функція $\chi'(\cdot)$ робить з

рядків кортежи та записує їх до матриці. Очевидно, що функції кодування та декодування оберненні одна до одної.

$$\begin{cases} \psi' = \chi'^{-1} \\ \chi' = \psi'^{-1} \end{cases}$$

Представимо матрицю як кортеж кортежів, тобто:

$$M_{m\times n} = < \begin{matrix} < a_1^1, a_2^1, \dots a_n^1 >, \\ < a_1^2, a_2^2, \dots a_n^2 >, \\ \dots, \\ < a_1^m, a_2^m, \dots a_n^m > \end{matrix} >=<< a_1^i, a_2^i, \dots a_n^i >, i = \overrightarrow{1,m} >$$

Врахуємо також що реляція розміром $m \times n$ позначається як:

$$R = \{< a_1^i, a_2^i, \dots a_n^i >, i = \overrightarrow{1,m}\}$$

Тоді, функція ψ' напряму перетворює множину кортежів у кортеж кортежів.

$$\psi'\left(\{< a_1^i, a_2^i, \dots a_n^i >, i = \overrightarrow{1,m}\}\right) =<< a_1^i, a_2^i, \dots a_n^i >, i = \overrightarrow{1,m} >$$

Перепишемо переведення реляцій з різним порядком строк більш формально.

$$\psi'(R) = A_{M\times N}$$

$$\psi'(R') = A'_{M\times N}$$

І вже перепишемо вираз (1) з використанням простих кодуючої та декодучої функцій.

$$\begin{cases} R = R' \\ \psi'(T) \neq \psi'(T') \end{cases}$$

Отриманий вираз доказує, що функції кодування та декодування не є бієкцією. Відсутність даної властивості є критичною, адже може призвести до конфліктів при виконанні або коли відбувається обмін інформацією між системами/програмами, що для зв’язку використовують операції ψ' та χ'.

Розглянемо рішення, яке було виведено в рамках цієї роботи, що дозволяє досягти властивість бієкції для кодуючої та декодуючої функцій. Один з найбільш очевидних методів упорядкування множини довільних значень є їхнє сортування. Для сортування числових значень зазвичай

використовують предикатів порівняння: більше $>$ та менше $<$. Таким чином можна відсортувати числа за зменшенням чи збільшенням, проте для кортежів такі предикати не визначені. А отже дані предикати необхідно розширити в рамках сортування строк реляцій. При чому розширити таким чином, щоб функція працювала як для одноелементних строк, так і для багатоелементних. Почнемо з порівняння одноелементних строк, у них порівнюються перші елементи строк. Спробуємо застосувати дану функцію для багатомісних строк, тобто порівнювати їхні перші елементи.

$$< a_1, \dots, a_n > < < b_1, \dots, b_n >= \begin{cases} T, a_1 < b_1 \\ F, \text{інакше} \end{cases}$$

Розберемо недоліки даного предикату. Очевидно, що дана функція не охоплює усієї строки. Тобто для випадку, коли у абсолютно різних строк будуть однакові лише перші елементи, дана функція не зможе розпізнати їхню нерівність. Її природнім розширенням буде охоплення всіх значень строки. У такому випадку функція буде завжди працювати коректно, адже в реляції не може повторюватись одна й та сама строка. Розширимо її так, що якщо два відповідні елементи строк рівні, тоді функція порівнює наступні елементи.

$$< a_1, \dots, a_n > <_n < b_1, \dots, b_n >= *_{\overrightarrow{1,n}} ((a_i < b_i) \diamond \Big(\text{повернути } T, \big((a_i > b_i) \diamond \text{ повернути } F\big)\Big)$$

Така функція проходиться по всім значенням строк $< a_n >, < b_n >$ у циклі. Якщо $a_i < b_i$, то функція повертає істинну; якщо $a_i > b_i$, то повертає хибу. Очевидно, що якщо двох однакових строк в одній реляції нема, то функція визначена для всіх строк в межах одної реляції. Саму функцію сортування назвемо $sort_R: R \times p \to R$, де p є предикатом, по якому функція сортує строки реляції. Очевидно, що дана функція є сюр'єкцією над множиною реляцій. Проте така сюр'єкція дозволить без помилок кодувати реляції в матриці та маніпулювати ними.

Тепер, введемо нову кодуючу фунцію $\psi_R^M : R \to M$, що на вхід функції ψ' подає відсортовану реляцію, $\psi_R^M(R) = \psi'(sort(R,p))$. При чому, p в даному випадку є параметром. Тепер побудуємо обернену функцію, тобто декодування $\chi_R^M = \psi_R^{M^{-1}}$. Але є один нюанс, дані про реляцію до сортування не зберігаються. Проте, в межах множини, реляції з різним порядком строк є тотожними. Тому, можемо обійтися простим прямим перетворенням матриць на реляції $\chi_R^M = \chi_R^{M'}$.

Розглянемо функції, що однаково працюють як на відсортованих матриць, та і на невідсортованих. Мова йде про $\Pi_M, S_M, C_{(0)_{1\times1}}$. Так, функції видалення першого стовпчика Π_M, збільшення всіх елементів першого стовпчика на одиницю S_M будуть однаково працювати з першим стовпчиком, не дивлячись на порядок елементів в ньому. Функція співставлення нульової матриці $(0)_{1\times1}$завжди буде повертати матрицю розміром 1×1з нулем. Ці функції є прямими аналогами функцій для реляцій $\Pi_R, S_R, C_{\{0\}}$, і працюватимуть навіть якщо кодувати реляцію в матрицю без урахування різниці між множиною та кортежем.

Про операції конкатенацій та порівняння матриць $\circ_M, =_M$такого вже сказати не можна, саме через відмінність кортежу від множини. Проте упорядяркування при кодуванні дозволяє вже напряму використати функцію порівняння матриць $=_M$, суть якої полягає в поелементному порівнянні двох матриць. Оскільки в загальному випадку матриці розміром $m \times n$відповідають $n!$реляцій, а вже при впорядкуванні реляції одній матриці відповідає одна реляція. А от функцію горизонтальної конкатенації матриць $\circ_M$ використати не вдасться, оскільки суть конкатенації матриць полягає в декартовому добутку, коли як ця функція просто приєднує до рядків одної матриці відповідні рядки іншої. Тому потрібно визначити нову функцію конкатенації реляцій $\circ_R^M$, причому працює призначена вона саме для упорядкованих матриць.

$$(a_{ij})_{m\times n} \circ_R^M (b_{ij})_{l\times k} = (c_{ij})_{(m\cdot l)\times(n+k)},$$

$$c_{ij} = \begin{cases} a_{\lfloor i/l+1 \rfloor j} & j \leq n; \\ b_{((i \bmod l)j-n} & j \geq n; \end{cases}$$

Такий складний запис обумовлений моделюванням декартового добутку. Так, перший рядок матриці $(a_{ij})_{m\times n}$ конкатенується з усіма рядками матриці $(b_{ij})_{l\times k}$, і так для всіх рядків матриці $(a_{ij})_{m\times n}$. При цьому, дані матриці є закодованими реляціями, з попереднім сортуванням строк, а оскільки дана функція покроково працює від першого до останнього рядка, то порядок такої матриці-реляції зберігається. Важливо зазначити, що матриці в даному прикладі є лише показником того, що $\circ_R^M$ є відображенням на матричну алгебру, насправді ж $\circ_R^M: R^2 \to R$.

Аналогічним образом робляться функції об'єднання $\cup_R^M$ та різниці $\setminus_R^M$ множин для матричного відображення.

В результаті отримуємо систему породжуючих функцій реляційної алгебри, вираженої через матричне відображення $\sigma_R^M \rightleftharpoons \{\Pi_R^M, S_R^M, C_{\{0\}}^M, \circ_R^M, =_R^M, \cup_r^M, \setminus_R^M, \psi_R^M, \chi_R^M I_m^n\}$. Дана система повністю аналогічна попередньо виведеній, за єдиними вийнятком, що відображення будується на матричну алгебру.

Теорема. $\sigma_R^M \rightleftharpoons \{\Pi_R^M, S_R^M, C_{\{0\}}^M, \circ_R^M, =_R^M, \cup_r^M, \setminus_R^M, \psi_R^M, \chi_R^M I_m^n\}$ — повна сукупність алгебри чр-функцій та чр-предикатів над реляціями $A_R^{чр}$.

Висновок. Завдяки ізоморфному відображенню реляційної алгебри на векторну та матричну вдалося побудувати її базис $A_R^{чр}$. Разом з сигнатурними операціями, базисні функції дозволяються реалізувати будь-яку функцію, визначену на множині R.

Порівнюючи відображення на векторну та матричну алгебру, стає очевидним, що реалізація на векторній алгебрі є простішою та очевиднішою. Виражається це насамперед тим, що вектор є простішим типом даних, та є прямою відповідністю одностроковій реляції. Матриця, в свою чергу, є більш складним типом даних, що можна виразити як кортеж кортежів; тоді як

реляція є множиною кортежів. І саме різниця між кортежем та множиною викликала основні незручності при побудові ізоморфного відображення.

Також необхідно пам’ятати, що в попередній роботі матрична алгебра була отримана в результаті її відображення на векторну. Тому буде справедливим використовувати в подальшому саме відображення на векторну алгебру σ_R. Проте рішення, що застосовувались при побудові ізоморфізму реляційної алгебри на матричну будуть використані в подальшому на етапі безпосередньої реалізації.

2.4 Реалізація функцій алгебри Кодда на основі виведеного відображення

Алгебра Кодда вважається одним з найпотужніших інструментів для створення функцій над реляціями, а також є фундаментом для мови програмування SQL.

Реляційна алгебра складається з 6 примітивних операцій, 3 з них які є теоретико-множинними, і призначені суто для реляцій:

1. Об’єднання множин UNION $A \cup B$;
2. Віднімання множин MINUS $A \backslash B$;
3. Декартів добуток TIMES $A \times B$;
4. Вибірка з реляції SEL$\sigma_{\varphi}(R)$

$$\sigma_{\varphi}(\{< a_1^i, \dots a_n^i > |\ i = \overrightarrow{1, \mathrm{m}}\}) = \{< a_1^i, \dots a_n^i > |\ p(< a_1^i, \dots a_n^i >) = T, i = \overrightarrow{1, \mathrm{m}}\}$$

де $p(< a_1^i, \dots a_n^i >)$ є предикатом над кортежем, що визначає чи буде кортеж у вихідній реляції.

5. Проекція реляції на підмножину атрибутів (стовпців) PROJ $\pi_{n_1,\dots,n_s}(R)$, де $n_1, \dots, n_s$ є множиною імен атрибутів. На виході функція залишить лише атрибути $n_1, \dots, n_s$.

$$\sigma_{\varphi}(\{< a_1^i, \dots a_n^i >| \; i = \overrightarrow{1, \mathrm{m}}\}) = \{< a_{n_1}^i, \dots a_{n_s}^i >|, i = \overrightarrow{1, \mathrm{m}}\}$$

6. Перейменування атрибутів $\rho_{a/b}(R)$. Дана операція ніяких значень в реляці не змінює, проте атрибутbв усіх кортежах переіменовується на a.

Всі ці операції є невід'ємною частиною алгебри, які не можна моделювати іншими. Щоправда, операція перейменування атрибутів є більше практичним інструментом, аніж теоретичним, тому тут ми її опустимо. А замість неї наведемо ще 3 важливі для SQL, та самої алгебри Кодда,операції: перетин множин $A \cap B$, ділення множини $A \div B$, та операцію так званого натурального з'єднання A JOIN B.

Тепер, реалізуємо вищезазначені операції з використанням виведених базисних фукнцій $\sigma_R \rightleftharpoons \{\Pi_R, S_R, C_{\{0\}}, \circ_R, =_R, \cup_r, \setminus_R, \psi_R, \chi_R, I_m^n\}$.

1. Об'єднання. Функція об'єднання двох реляцій вже наявна в σ_R, тому дана операція реалізується банальною підстановкою.
   A UNION B = $A \cup_R B$
2. Віднімання. Віднімання множин також присутнє в σ_R, тому робимо аналогічно об'єднанню.
   A MINUS B = $A \setminus_R B$
3. Декартовий добуток. Операція декартового добутку також є в σ_R.
   A TIMES B = $A \circ_R B$
4. Перетин. Операції перетину вже в σ_R безпосередньо нема, проте вона доволі просто реалізується за допомогою операції віднімання множин.
   A INTERSECT B = $A \setminus_R (A \setminus_R B)$
5. Вибірка. Це вже більш складна операція, яка потребує додаткових функцій. Так, введемо функцію видалення першої строчки $D: R \to R$

   $$D_R(\{< a_1^i, a_2^i, \dots a_n^i >| \; i = \overrightarrow{1, \mathrm{m}}\}) = \{< a_1^i, a_2^i, \dots a_n^i >| \; i = \overrightarrow{2, \mathrm{m}}\}$$

   Далі введемо функцію вибору k-ї строки з реляції $J_k^n: R \to R, k \leq n$

   $$J_k^n(\{< a_1^i, a_2^i, \dots a_n^i >| \; i = \overrightarrow{1, \mathrm{n}}\}) = \{< a_1^k, a_2^k, \dots a_n^k >\}$$

   Вибірка вважається параметричною фукнцією, в якості параметру виступає умова, яка визначає чи буде кортеж у вихідній реляції.

SEL A WHERE p =

$R_{ret} = \emptyset,$
$(A =_R \emptyset) * ($
$R_1 = J_1^n(A),$
$(p(R_1)) \diamond (R_{ret} \cup_R R_1),$
$)$
$returnR_{ret}$

Так, функція покроково вибирає першу строку вхідної матриці A, і якщо та відповідає умові параметричного предикату p, то строка додається до вихідної реляції дією об'єднання.

6. З'єднання. Дана операція дещо відрізняється від операції декартового добутку. Проте при її реалізації використовується саме він. Так з декартового добутку вибираються строчки, які відповідають умовірівності значень з вхідних реляцій. Тож дану операцію можна реалізувати на вже реалізованих TIMES та SEL
A JOIN B = SEL (A TIMES B) WHERE A=B
7. Проекція. Для зручності реалізації даної операції введемо додаткову функцію видалення з реляції -го стовпчика $\Pi_R^j: R \to R, j \leq n.$ Її позначення, очевидно, нагадує запис операції видалення першого стовпчика, це зроблено для підкреслювання суті даної операції.

$$\Pi_R^j(\{< a_1^i, a_2^i, \dots a_n^i > | \ i = \overrightarrow{1, \mathrm{m}}\}) = \{< a_1^i, \dots a_{j-1}^i, a_{j+1}^i, \dots a_n^i > | \ i = \overrightarrow{1, \mathrm{m}}\}$$

Очевидно що проекція є параметричною операцією, тому її параметр позначимо як кортеж $T =< t_1, t_2, \dots t_k >$;причому це кортеж не стовпців, які повинні залишитись, а які потрібно видалити. Так, операція проекції реалізується наступним чином.

A$[X \notin T] = \Pi_R^{t_1}(\Pi_R^{t_2}(\dots \Pi_R^{t_k}(A)))$

Тобто операція проекції реалізується як рекурсивне застосування операції видалення t_i-го стовпчика.При цьому $X \cup T = \overrightarrow{1, \mathrm{n}}$

8. Ділення. Дану операцію можна виразити через операції проекції, декартового добутку та віднімання множин наступним чином. A DIVIDEBY B = A[X] MINUS((A[X] TIMES B) MINUS A)[X]

Таким чином, на основі базисних функцій з σ_R було реалізовано основні операції з алгебри Кодда. Це доводить потужність виведеної алгебраїчної характеристики. Проте, сама алгебра Кодда не доскональна, і в ній не можна реалізувати, наприклад, операцію транзитивного замикання. А за допомогою даної алгебраїчної характеристики можна реалізувати більш широкий обсяг операцій.

2.5 Уточнення реляцій іменною моделлю даних

Як було зазначено у вступі, реляції представляються у вигляді двовимірних реляць зі стовпчиками і рядками. Кожна строка реляці це набір пов'язаних між собою значень, які стосуються одного кортежу чи сутності. У кожному стовпці реляці зберігається певний тип даних, атрибут кортежу чи сутності; а у кожній комірці – значення атрибута. Кожен рядок у реляці може бути позначений унікальним ідентифікатором, який називається первинним ключем, а рядки з кількох реляць можуть бути пов'язані за допомогою зовнішніх ключів. До цих даних можна отримати доступ багатьма способами і при цьому реорганізовувати реляці бази даних не потрібно. Розглянемо вищезазначене більш формально.

Кортеж, як було зазначено, у реляційному типі даних представлений як сукупність пов’язаних між собою значень. При цьому, оскільки у кожному стовпці зберігається певний тип даних, тому справедливим є те, що сукупність даних в кортежі є строго впорядкованою. Таким чином запишемо кортеж з n-ї довжини як кортеж:

$$tuple =< a_1, a_2, \dots a_n >,$$

Далі будемо згадувати елементи кортежу як атрибути реляції. При цьому варто уточнити сам атрибут іменною моделлю даних. Зазвичай, іменна

модель даних представляє змінну парою $< v, a >$, де v – це значення, що зберігається за адресою A. Проте для атрибуту також важливим параметром є і його тип даних. Тип даних визначає не лише розмір та формат збереження змінної, а й базові операції над нею. Отже іменна модель даних атрибуту виглядає так: $< v, a, t >$, де t вказує на тип змінної.

Повернемося до кортежу. Перепишемо його для іменних множин.

$$tuple^n = << v_1, a_1, t_1 >, < v_2, a_2, t_2 >, \dots, < v_n, a_n, t_n >>,$$

де $tuple^n$позначає іменну модель кортежу. При цьому варто згадати, що кортеж є впорядкованою послідовністю з атрибутів, тому відокремимо різні елементи кортежу іменної множини та об'єднаємо їх в один великий кортеж, так іменну модель кортежу можна представити трьома кортежами.

$$tuple^n \equiv << v_1, v_2, \dots v_n >, < a_1, a_2, \dots a_n >, < t_1, t_2, \dots t_n >>,$$

де у першому кортежі зберігаються значення атрибутів, у другому зберігаються їхні адреси, а у третьому їхні типи.

Тепер перейдемо від абстракції до реального спососу зберігання даних у пам'яті для розглядання можливості скоротити вираз кортежу. У комп'ютерній архітектурі пам'ять Mem є вектором, для зчитування та запису якого необхідно вказати індекс елементу, що називається адресою. Введемо поняття компактності. Компактність позначає те, що всі елементи складного впорядкованого типу даних v зберігаються щільно одна за іншою, без розривів; а самі дані займають лише один діапазон адрес, тобто починаючи з i-ї адреси, закінчуючи $i + L$-тою адресою $v = read(Mem, < i, i + L >)$, де $read$ це функція зчитування з пам'яті значень, n це розмір типу даних.

Якщо припустити, що кортежу властива компактність зберігання, то, враховуючи впорядкованість кортежу, можна скоротити кортеж адрес до двох елементів: адреси початкового елементу та кількість атрибутів.

$$tuple^n \equiv << v_0, v_1, \dots v_{N-1} >, < a_0, n >, < t_0, t_1, \dots t_{N-1} >>$$

А тепер розглянемо реляції. Реляція є сукупністю кортежів, при цьому впорядкованість немає значення, адже перший елемент дозволяє

безпомилково визначити кортеж. Так, реляцію розміром m можна представити у вигляді:

$$R = \{tuple_1, tuple_2, \dots tuple_m\}$$

Фігурними скобками позначається невпорядкована множина. Далі береться виведена іменна модель даних кортежу, і підставляється до визначення реляції. Відразу ж варто скоротити повторювані дані. Так, можна прибрати кортеж з типами даних, оскільки він є спільним для усіх кортежів. А також, якщо припустити, що реляції теж властива компактність, тоді можна замінити кортежі адрес кортежів на один спільний кортеж для реляції з її початковим елементом та розміром, а також усі кортежі значень атрибутів кортежів щільно слідують один за одним у пам'яті. Враховуючи все це, іменна модель реляції записується як:

$$R^n \equiv < \begin{Bmatrix} < v_1^1, v_2^1, \dots v_n^1 >, \\ < v_1^2, v_2^2, \dots v_n^2 >, \\ \dots, \\ < v_1^m, v_2^m, \dots v_n^m > \end{Bmatrix}, < a_0^R, m, n >, < t_0, t_1, \dots t_{N-1} >>$$

де v_i^j – i-й атрибут j -го кортежу,

a_0^R – адреса початкового елементу реляції,

m, n – розмір реляції.

Важливо зазначити що порядок розміщення кортежів у реляці не важливий. Навіть у ній будуть наявні два ідентичні записи, їх все ще можна розрізнити завдяки первинному ключу.

2.6 Співставлення іменної моделі реляцій до іменної моделі векторів та матриць

Раніше було введено кодування та декодування даних реляцій у вектори та матриці. Тепер, розширимо це представлення для іменної моделі даних. Для зручності, представимо кортежі адрес та типів як однострочні

реляції. Тоді кодуючу функцію для іменних множин $\psi_{R^n}: R^n \to N^{*n}$ представимо так:

$$\psi_{R^n}\left(< \left\{< a_1^i, a_2^i, \dots a_n^i >, i = \overrightarrow{1,m}\right\}, < a_0^R, m, n >, < t_0, t_1, \dots t_{N-1} >\right) =$$
$$< n, a_1^1, a_2^1, \dots a_n^m >, < a_0^R, m, n >, < t_0, t_1, \dots t_{N-1} >>$$

Результатом є іменна модель кодованої реляції в іменну модель кодуючого вектора N^{*n}.

$$\psi_{R^n}(R^n) = N^{*n}$$

Відповідно, декодуючи функція χ_{R^n} декодує іменну модель кодую чого вектора в іменну модель реляції.

$$\chi_{R^n}(N^{*n}) = R^n$$

Коли кодуючи та декодуюча функції для іменних моделей реляцій введені, тепер можна перевизначити інші базисні функції саме для іменних моделей реляцій. В результаті вийде система

$$\sigma_{R^n} \rightleftharpoons \{\Pi_{R^n}, S_{R^n}, C_{\{0\}^n}, \circ_{R^n}, =_{R^n}, \cup_{R^n}, \setminus_{R^n}, \psi_{R^n}, \chi_{R^n}, I_m^n\}$$

Теорема. $\sigma_{R^n} \rightleftharpoons \{\Pi_{R^n}, S_{R^n}, C_{\{0\}^n}, \circ_{R^n}, =_{R^n}, \cup_{R^n}, \setminus_{R^n}, \psi_{R^n}, \chi_{R^n}, I_m^n\}$— повна сукупність алгебри чр-функцій та чр-предикатів над над іменною моделлю реляцій $A_{R^n}^{чр}$.

2.7 Апаратна підтримка різноманітних типів даних у реляці

Як вже було зазначено, всі елементи векторів та матриць належать до одного й того ж типу, зазвичай чисельного. Атрибути реляцій можуть мати різні типи даних, у тому числі складні структури або символьні послідовності. Це призводить до складності відтворення деяких атрибутів у матриці. Також варто врахувати фактор архітектури, адже саме вона обмежує функціональність.

На щастя, усі дані, з якими працює комп’ютер, є числами. Таким чином усі символи закодовані своїми числовими значеннями, логічні прапорці є нічим іншим як розрядами двійкових чисел, а вся інша інформація

представляється у цифровому вигляді. Єдине чим вони відрізняються то це розрядністю та набором базових операцій над даними. Такий нюанс значно облегшує представлення даних атрибутів у матричному обчислювачі. Проте, якщо кодувати реляцію в матрицю чи вектор, то кожен атрибут повинен поміститися в комірку, відведену для одного елементу. Для цього є низка технологічно-логічних рішень, викладених нижче.

1) Вказівники. У комп'ютерних науках проблему зберігання структурних та багатовимірних даних вирішують за допомогою вказівників. Вказівник на кортеж це така спеціальна змінна, що зберігає початкову адресу кортежу. При цьому у багатьох мовах програмування, де наявні вказівники, необхідно вказувати на тип кортежу, на який вказує вказівник.

Рисунок 2.1 – Вказівник у пам'яті.

Таким чином значення атрибуту може бути не сам рядок, а лише посилання на нього. Таке рішення має як свої переваги, так і недоліки.

До переваг можна віднести, що такий спосіб є більш економним, якщо програми чи апаратні комплекси мають єдину пам'ять. У такому випадку не доведеться кожного разу копіювати увесь текст чи велику структуру між суб'єктами виконання. Друга перевага це економія пам'яті, адже в матриці

можна зберігати лише вказівник, а сам рядок буде займати стільки місця, скільки він фактично потребує.

До недоліків можна віднести те, що зазвичай вказівниками користуються універсальні машини, що мають достатньо ресурсів для різноманітної обробки однієї змінної. Проте для процесорів, які обробляють великі масиви даних, це може бути проблемою. Адже такі процесори, зазвичай, їхні ядра мають лише обчислювальний потенціал і не працюють напряму з вказівниками. Насамперед до таких належить і SM. Також якщо необхідно обмінюватись даними з сутністю, у якої інша система пам’яті, тоді потрібно передати окремо й усю структуру, на яку посилається вказівник. Також усі елементи матриці повинні

Рішення з вказівниками не підходить для матричних обчислювачів, оскільки вони не пристосовані до операцій зі вказівниками, а для їхньої реалізації потрібно буде чимала кількість ресурсів кремнію.

2) Розбиття складних атрибутів на декілька (нормалізація). Ця операція фактично входить до приведення реляцій до нормальної форми [?], тому є досить нативною для реляцій. Називається вона нормалізацією. Проте у низькорівневому випадку доведеться йти ще далі: так атрибути мають підлягати розбиттю не лише логічно, але й на хардварному рівні. Перш за все це стосується символьних рядків. Розглянемо це на прикладі. У класичному випадку атрибут «Прізвище, Ім’я, По батькові» розбивається на «Прізвище», «Ім’я», «По батькові». У нашому випадку необхідно ще додатково розбити атрибути «Прізвище», «Ім’я», «По батькові» на атрибути, які позначають позицію символу в слові. Тобто як: «П», «р», «і», «з», «в», «и», «щ», «е», «І», «м», «’», «я», «П», «о» «б», «а», «т», «ь», «к», «о», «в», «і».

Таблиця 1. Приклади нормалізації реляції

Form	Not NF	Normal form		Hardware implementation						
Attribute	Full name	Name	Surname	Name[0]	Name[1]	Name[2]	Name[3]	Surname[0]	Surname[1]	Surname[2]
Value	John Doe	John	Doe	J	o	h	n	D	o	e

Як видно з таблиці 1, не нормалізована форма містить у собі один атрибут «повне ім’я», класично нормалізована містить два атрибути «ім’я» та

«прізвище», а в нормалізованій для заліза вже кожний символ виступає окремим атрибутом.

Хоч проблема розбиття символьних рядків виглядає як питання зручності, проте насправді воно за собою несе проблему невизначеності для розміру рядкового буферу. Коли програма приймає довільний символьний рядок у буфер, то невідомо якої довжини цей рядок буде. Так, якщо буфер буде замалим, то користувач не зможе записати усю необхідну інформацію. Якщо буфер буде завеликим, то частина пам’яті не буде використовуватись взагалі. Як компроміс, програмісти в документації вказують розмір вхідного буферу, що повинен обмежити занадто довгі рядки, проте помістити більшу частину.

Також накладається обмеження по розміру комірки для одного елементу. Так, його мінімальний розмір повинен бути не меншим за розрядність кодування символу. Наприклад, якщо ми використовуємо символи з ASCII, то розрядність одного елементу матриці повинна бути не меншою за 8 біт. При цьому, якщо розрядність одного елемента є більшою за розрядність кодування символу та є кратним йому, то в одній комірці можна зберігати декілька значень. Завдяки бітовим полям можна розрізнити окремі літери в такому записі.

Переваги нормалізації:

1. Нативна імплементація для обробки багатовимірних даних;
2. Нативна імплементація для розбиття реляцій;
3. Простота у реалізації та обробці.

Недоліки нормалізації:

1. Проблема невизначеності розміру буфера;
2. Розмір самих матриць стає значно більшим.

Загалом, цей метод підтримки складних атрибутів є простим у програмній імплементації та не вимагає жодних модифікацій заліза чи складних методів кодування та декодування.

3) Обмеження лише числові дані одного типу. Такий метод є найбільш простим, проте він сильно обмежує функціональність. Фактично, обмеження йде за рахунок того, що ми обмежуємось лише тою функціональністю, яка необхідна для матриць. Таке рішення дозволить напряму звертатись до реляцій як до матриць чи векторів. Проте воно не дозволяє використовувати в реляціях більш складні структури, включаючи символьний рядок, що сильно обмежує використання таких реляцій у практичних випадках. Тим не менше, це рішення цілком підходить для теоретичних задач.

Перевага такого рішення є пряме використання реляць як матриць. Таке рішення є найбільш простим та найбільш показовим.

До недоліків такого рішення є сильно обмежений функціонал, що фактично не дозволяє його використовувати для практичних реляць.

Отже таке рішення придатне лише для теоретичних випадків, адже обмеження функціональності у такому випадку не вплине не важливе.

Підсумки. Серед розглянутих трьох варіантів найбільш оптимальним є нормалізація. Цей метод дозволяє при мінімальних програмних затратах підтримати повний функціонал для реляцій. Використання вказівників є дещо ускладненим рішенням. Воно потребує значного розширення цифрової логіки ядра та знижує продуктивність багатопотокових процесорів. Тому таке рішення без ряду досліджень є нежиттєздатним.

Отже, в даній роботі було обрано варіант з нормалізацією реляції.

2.8 Висновок по розділу 2

Отже, в даному розділі було виведено алгебраїчну характеристику реляційних чр-функцій та чр-предикатів для ППА методом ізоморфного відображення на векторну та матричну алгебри. Було доведено потужність даної алгебраїчної характеристики шлях моделювання операцій з алгебри Кодда за допомогою базисних операцій виведеної алгебри. Було її уточнено

для іменної моделі даних, а також була представлена можливість апаратної підтримки складних типів даних.

Алгебраїчна характеристика являє собою множину операцій $\sigma_R \rightleftharpoons \{\Pi_R, S_R, C_{\{0\}}, \circ_R, =_R, \cup_r, \setminus_R, \psi_R, \chi_R, I_m^n\}$ для векторів; і для матриць $\sigma_R^M \rightleftharpoons \{\Pi_R^M, S_R^M, C_{\{0\}}^M, \circ_R^M, =_R^M, \cup_r^M, \setminus_R^M, \psi_R^M, \chi_R^M I_m^n\}$. Сюди входять наступні операції Π_R – видалення першого стовпчика реляції; S_R – функція слідування (збільшення всіх елементів першого стовпчика на 1); $C_{\{0\}}$ – співставлення будь-якій реляції реляції з єдиним елементом «0»; $\circ_R$ – множення двох реляцій; $=_R$ – предикат рівності двох реляцій; $\cup_r$ – об'єднання двох реляцій; $\setminus_R$ – різниця двох реляцій; ψ_R – кодуюча функція; χ_R – декодуюча функція; I_m^n – селекторна функція. І це є повна сукупність алгебри чр-функцій та чр-предикатів над реляціями $A_R^{чр}$.

За допомогою виведеної сукупності було реалізовано 5 примітивних операцій алгебри Кодда. А вже з цих примітивних операцій можна утворити й інші. Таким чином було доведено, що дана алгебраїчна характеристика повністю підтримує мову SQL, яка базується на алгебрі Кодда.

Було уточнено реляцію іменною множиною даних, а також було уточнено ізоморфне відображення іменною моделлю даних. В іменній моделі даних змінна представлена парою з адреси та значенням, що зберігається за адресою, що є важливим кроком для апаратної реалізації. Серед способів інтерпретації складних типів даних було обрано метод нормалізації таблиці як найбільш інтуїтивний та оптимальний по використанню ресурсів.

Розділ 3. Реляційний процесор

В даному розділі описується реляційний процесор, в основу якого покладено матричний процесор з бакалаврської роботи [1]. До його архітектури були внесені модифікації та реорганізації, а потім адаптовано для використання з реляціями. Тепер керуючий блок та виконавчий блок називаються ядрами, щоб підкреслити їхню абстрактність над іншими блоками та прибрати плутанину з їхніми внутрішніми блоками керування. В ході виконання роботи було модифіковано керуюче ядро, пам'яті імен та реляцій, а також значно реорганізовано виконавче ядро. Характер модифікацій полягає у покращенні та полегшенні процесу програмування обчислювача та його реконфігурації, і разом з цим прибирання надлишкових компонентів. Проте, загальна структура залишилась незмінною.

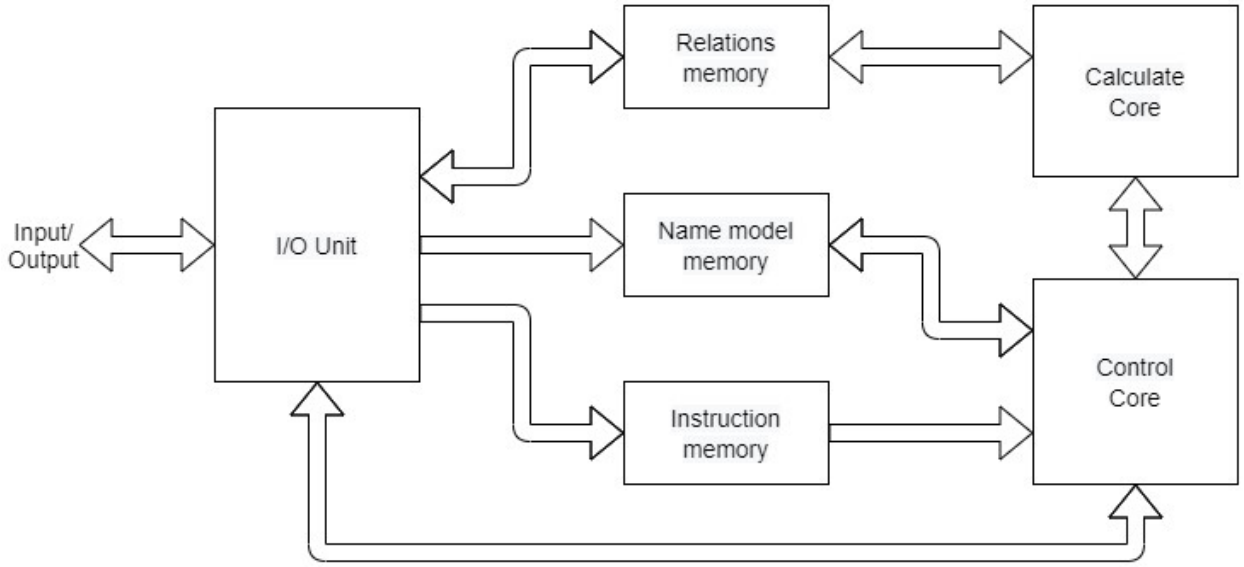


Рисунок 3.1 – структура реляційного процесору

Представлена на рисунку 3.1 структура ідентична структурі матричного процесору з [1]. Блок *I/O Unit* призначений для обміну даних між зовнішнім пристроєм та реляційним прискорювачем. Окрім обміну даних, він також обробляє переривання та забезпечує безпечне зчитування/надсилання даних. Даний блок розподіляє дані з входу пристрою до трьох блоків пам'яті. Блок пам'яті *Instruction memory* зберігає інструкції, які виконуються прискорювачем. *Name model memory* зберігає імена з іменні моделі реляцій, в

яких міститься основна інформація про реляції, що забезпечує можливість правильного зчитування реляції з пам’яті. *Relations memory* зберігає елементи реляцій. В даній реалізації адресні шини усіх блоків пам’яті займають 16 біт. *Control Core* відповідає за виконання програм. Він послідовно обробляє інструкції та декодує інструкції для *Calculate Core*, виконує операції з пам’яттю імен, реалізує команду «стрибку» до інструкції, а також саме він генерує переривання до зовнішнього пристрою. *Calculate unit*, в свою чергу, виконує основні операції з елементами реляцій, та самими реляціями.

Керуюче ядро зберігає, які представляють з себе так звані метадані про реляцію у вигляді початкової адреси в пам'яті та її розмірність. Фактично ці дані дозволяють отримати коректний доступ до кожного елемента реляції. В імені реляцій налічується 3 бітових поля фіксованої довжини. В цих бітових полях записані значення адреси початкового елементу матриці, кількості строк і атрибутів. Саме вони дозволяють значно спростити спосіб задання команд та їх обробку. Також це дозволяє розділити цифрові схеми для різного призначення. Далі машине слово, яке містить інформацію про реляції буде називатися іменем реляції. Їх структура наведена на рисунку 3.2.

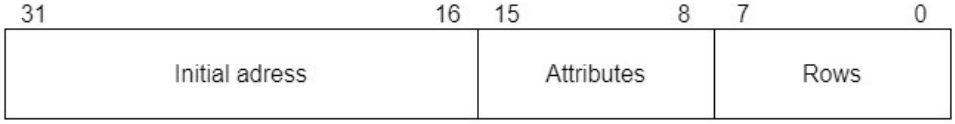


Рисунок 3.2 – Бітові поля імені реляції

На рисунку 3.2 показано приклад імені реляції з довжиною 32 біта. Тут, *Initial address* – адреса першого елементу реляції; *Rows* – кількість строк; Attributes – кількість атрибутів. Саме ім’я такої довжини використовується в даній реалізації. Кількість бітів на адресне поле залежить від розрядності адресної шини пам’яті елементів. Такі слова керуюче ядро пересилає виконавчому разом з інструкцією та параметрами.

3.1 Інструкції

В цій роботі набір інструкцій значно розширився, в порівнянні з попередньою версією. Новий процесор реляцій/таблиць має 7 типів інструкцій: реляційні, арифметичні, порівняння, селекторні, стрибок, читання/запис імен, та переривання. Такий набір дозволить повноцінно працювати з таблицями та їхніми даними, забезпечуючи підтримку мови програмування SQL. Причому підтримка для більшості операцій цієї мови забезпечена на рівні прямого переводу базових конструкцій в асемблерні інструкції.

Реляційні інструкції до себе включають як множинні операції, так і операції над залежностями атрибутів. Під операціями над залежностями атрибутів розуміється що можна з одних реляцій робити реляції іншого рангу, або з іншим порядком атрибутів. Операції даного типу можуть бути як бінарними, так і унарними.

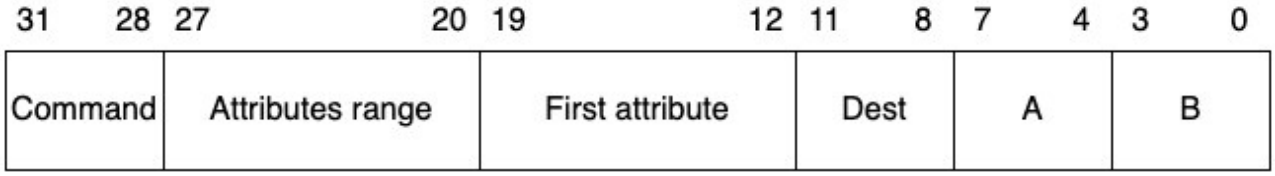


Рисунок 3.3 — Бітові поля реляційних інструкцій

Перші 12 бітів інструкції займають 3 адреси операндів у регістровому файлі: таблиці результату (*Dest*), операнду A (*A*), та B (*B*).На кожен операнд відводиться по 4 біти. З 12 по 19 біти займає поле *First attribute.* У ньому зберігається індекс атрибуту, з якого обчислювач буде починати обробку операції. Поле *Attribute range* (20 - 27 біти) вказує кількість атрибутів, яку обчислювачу необхідно обробити. Останнє поле відповідає за код операції *Command* (31 - 28 біти), яку необхідно виконати. При чому, дане поле є в усіх операцій над реляціями.

До операцій даного типу належать: об’єднання (UNION), перетин (INTERSECT), віднімання множин (MINUS), декартовий добуток (TIMES), поділ множин (DIVIDEBY), натурального об’єднання (JOIN) а також проекції (PROJ), яка може імітувати операцію копіювання усієї таблиці (для цього типу операцій і потрібні поля для атрибутів в інструкції). Дані операції є типовими для реляцій та таблиць, зокрема в мові програмування SQL.

Таблиця 3.1 – коди поля *Command*

Тип	command	Назва
Реляційні	0001	UNION
	0010	MINUS
	0011	INTERSECT
	0100	PROJ
	0101	TIMES
	0110	DIVIDEBY
	0111	JOIN
Селекторні	1000	SEL.M
	1001	SEL.S
	1010	SEL
Порівняння	1011	COMP
	1100	COMP.S
Арифметичні	1101	ARITHM
	1110	ARITHM.V
	1111	ARITHM.S

Як видно з таблиці 3.1, поле *Command* використовується не лише для реляційних операцій, а для всіх операцій над реляціями. Далі розглядаються інші операції, що згадані в даній таблиці.

Арифметичні інструкції націлені на арифметичні маніпуляції з атрибутами. Дані маніпуляції є виключно бінарними. Існує 3 різновиди

арифметичних операцій: $(R_{m\times n} \times R_{m\times n})$, $(R_{m\times n} \times R_{1\times n})$, $(R_{m\times n} \times R_{1\times 1})$. $R_{m\times n}$ — реляція з nкортежів розміру m, $R_{1\times n}$ — реляція з n одновимірних кортежів, $R_{1\times 1}$ — реляція зодним однорозмірним кортежем. Перший тип операцій застосовується для одногангових реляцій з однаковими атрибутами. У другому типі операції проводяться над одним певним атрибутом реляції та реляції з єдиним атрибутом. Третій тип дозволяє провести операцію з лише з одним значенням, яке буде застосовуватись до усіх значень певного атрибуту.

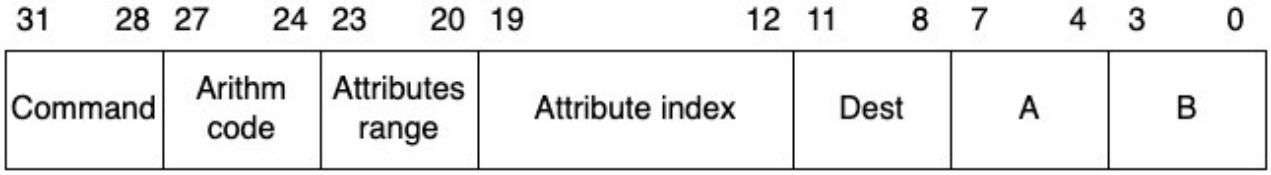


Рисунок 3.4 — Бітові поля арифметичних інструкцій

Як і з реляційними інструкціями, перші 12 бітів відведені для 3-х адрес операндів у регістровому файлі: таблиці призначення (*Dest*), операнду A(*A*), та B(*B*). Оскільки арифметичні операції виконуються лише над одним атрибутом, тому з необхідної інформації залишається лише поле із значенням зсуву атрибуту, над яким виконуються арифметичні дії *Attribute index.* У полі *Command* (31 - 28 біти) вказується код типу арифметичної операції, причому у цьому полі мають бути зазначені обмежена кількість операцій. У полі *Arithm code* (27 - 24 біти) зазначається яка саме арифметична дія повинна бути виконана над числами. На поле *Attributes range* відведено 4 біти (23 - 20). Так, діапазон охоплення скорочується до 16 атрибутів, проте цього цілком достатньо для інструкцій даного типу.

Таблиця 3.2 – Відповідність коду поля *Arithm code* до операцій

Код	Дія
1001	$c = a - b$
1101	$c = a + b$
0100	$c = a\ /b$
1100	$c = a \times b$

Продовження Таблиці 3.2

0111	$c = a + c_{i-1}$
0011	$c = a - c_{i-1}$
0010	$c_i = a \,/\, b - c_{i-1}$
0110	$c_i = a \,/\, b + c_{i-1}$
1010	$c_i = a \times b - c_{i-1}$
1110	$c_i = a \times b + c_{i-1}$

Завдяки кодам з таблиці 3.2, одна арифметична інструкція розбивається на 8. Самі ж коди операцій обумовлені прапорцями арифметично-логічного блоку (АЛУ), що використовується у ядрі обчислень. Далі більш детально розглядається про АЛУ та його структура.

Інструкції порівняння тепер значно розширились. Якщо у попередній версії було лише порівняння на рівність двох матриць, то в поточній версії доступні порівняння з усіма видами нерівностей. Це стало можливим за допомогою використання прапорців для вихідного результату віднімання двох чисел, а саме від'ємного результату(N), нуля (Z), переносу (C) та переповнення (V). Дані прапорці активно використовуються в комп'ютерній архітектурі, тож імплементація такого механізму є досить інтуїтивною.

Таблиця 3.3 – Відповідність предикатів порівняння та їхні формули

Предикат	Знакові	Беззнакові
=	Z	Z
≠	¬Z	¬Z
<	N⊕V	¬C
≤	Z + (N⊕V)	Z + ¬C
>	Z • ¬(N⊕V)	¬Z • C
≥	¬(N⊕V)	C

Як видно, в таблиці 3.3 для обчислення значення порівнянь використовуються саме прапорці NZCV. Таким чином можна імплементувати основні операції порівняння в процесор.

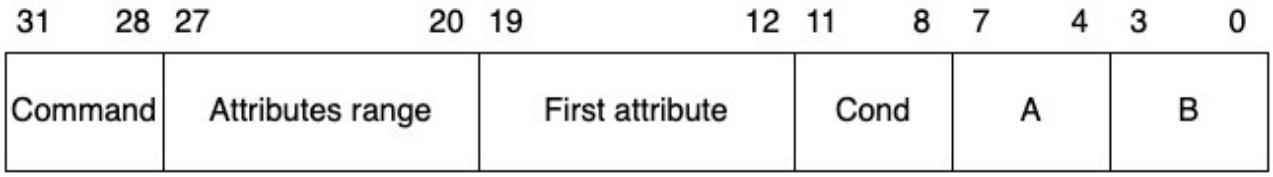


Рисунок 3.5 — Бітові поля інструкцій порівняння

Усі поля даного типу інструкцій ідентичні до попередніх (*Command, Attributes range, First attribute, A, B*), окрім поля *Cond* (11 - 8 біти). Саме в це поле записується умова для порівняння.

Таблиця 3.4 – Відповідність кодів *Cond* до операцій порівняння

Предикат	Код
=	0001
≠	0010
<	0100
≤	0101
>	1000
≥	1001

У виконавчому ядрі обчислене значення NZCV також транслюється у коди з таблиці 3.4 та порівнюється з тим, що було записано у полі *Cond.*

Порівнювати можна як одне значення, так і декілька атрибутів, що розміщюються в пам’яті поруч. Інструкція для порівняння одного значення називається COMP.S, а порівняння групи атрибутів (в тому числі повністю порівняти дві таблиці) використовується функція COMP.

Селекторні інструкції дозволяють обрати із існуючих записів ті, що відповідають певним умовам. Для того, щоб з таблиці A виділити необхідні

записи, використовується референсна таблиця B здовжиною кортежів у кількість атрибутів для перевірки; усі підходящі кортежі записуються в таблицю Dest.

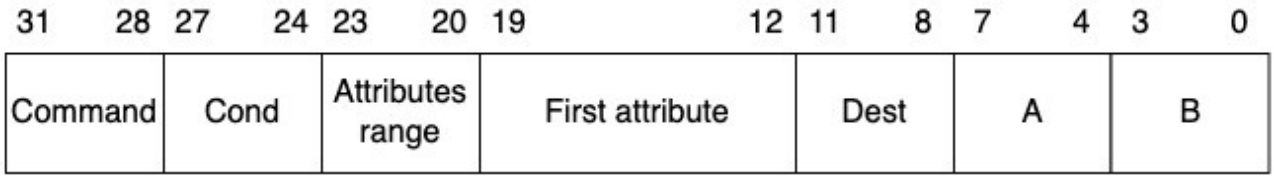


Рисунок 3.6 — Бітові поля селекторних інструкцій

Всі поля (*Command, NZCV, Attributes range, First attribute, Dest, A, B*) аналогічні до попередньо розглянутих. Проте в даному типі інструкцій поле *Attributes range* скорочено до 4 бітів з 8-ми. Залишившися 4 біти відведено для поля *Cond,* щоб вказати предикат для порівняння. В даному випадку, зазвичай, забагато атрибутів не перевіряються. Коди предикатів такі ж, як в таблиці 3.4.

Інструкції читання/запису імен призначені для запису до регістрів керуючого блоку імен таблиць, чи їх вивантаження до пам’яті. Даний тип також включає в себе операції зі стеком.

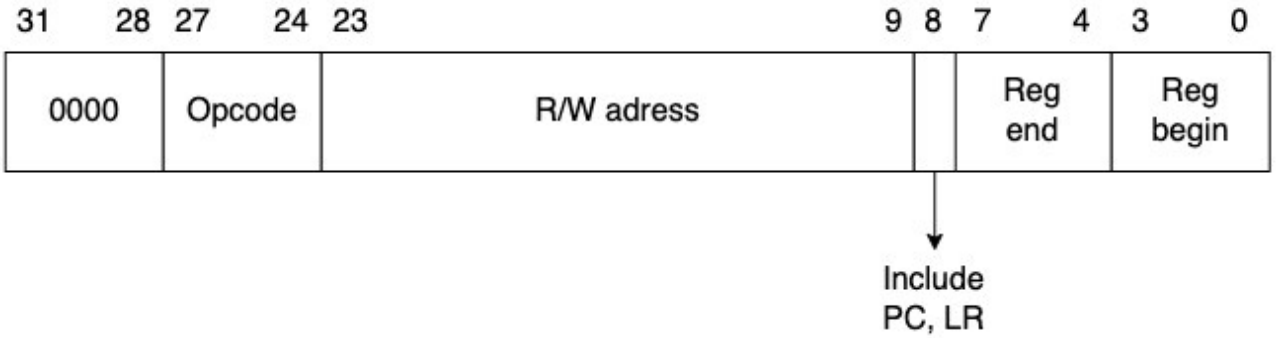


Рисунок 3.7 – Бітові поля інструкцій читання/запису

Бітові поля данного типу інструкцій вже суттєво відрізняються від вищеоглянутих, оскільки вони призначені зовсім для іншої частини процесора.

Команда типу R/W містить 5 бітових поля:

- *Opcode* (27 – 24) – поле для вказання типу операцій;
- *R/W address* (23 – 9) – адреса зчитування/запису з пам’яті;

- *Include PC, Ret* (8) – прапорець включення програмного лічильника та адреси переходу;
- *Reg end* (7 – 4) – кінцевий регістр зчитування/запису даних;
- *Reg begin* (3 – 0) – початковий регістр зчитування/запису даних.

Старші 4 біти в нульовому значенні вказують на те, що інструкція належить саме до типу R/W.

При виклику інструкції зчитування відбувається читання значення з комірок пам'яті імен, починаючи з *R/W address*, та закінчуючи *R/W address* + *Reg end,* та аналогічно для інструкції запису. Коли вибирається інструкція читання/запису стеку, то адреса *R/W address* не працює, замість неї використовується внутрішній регістр зі значенням вказівника стеку *stack pointer*. Коли встановлений прапорець *Include PC, Ret*, зі стеку зчитується значення програмного лічильника та так званої «точки повернення» в програму. Точка повернення включає в себе значення регістру повернення з функції, а також індекси регістрів, які були збережені в стек до виклику функції, з якої було викликано поточну.

Таблиця 3.5 – відповідність значення *Opcode* та операцій Read/Write

Opcode	Операція
0001	Read
0011	Write
0101	Read stack
0111	Write stack

Поле *Opcode* налічує всього 4 операції. Причому 3-й біт є прапорцем, що переключає читання/запис на стек. А 2-й біт є прапорцем переключення з читання на запис. При цьому, 4-й та 1-й біт зафіксовані в «0» та «1» відповідно, оскільки біти в даному полі не можуть бути усі нульовими та усі одиничними.

Інструкції типу Jump записують в програмний лічильник адресу стрибку, тобто інструкції, з якої програма почне виконувати дії наступного такту; звідси й назва даного типу команд. Існує інструкція умовного та безумного переходу. Умовний стрибок використовує значення предикату, розрахованого в минулій інструкції. Якщо умова істина, тоді відбувається стрибок.

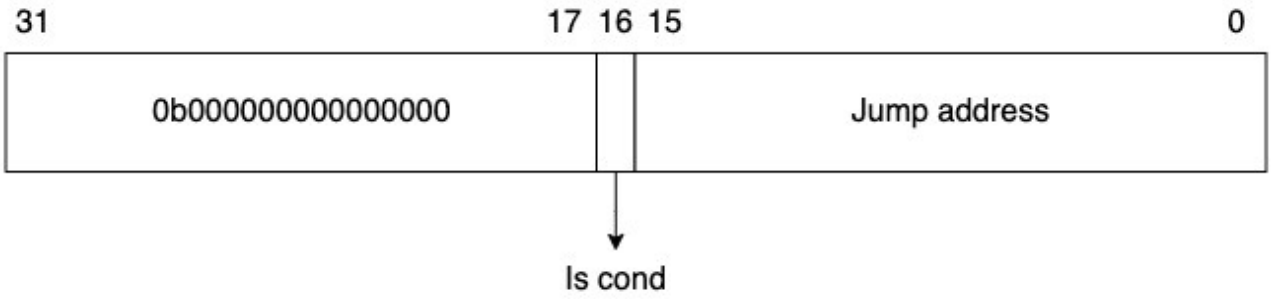


Рисунок 3.8 – Структура інструкції типу Jump

Інструкція типу Jump містить 5 полів:

- *Is cond* – біт, який указує чи перехід є умовним (логічна одиниця), чи безумовним (логічний нуль);
- *Jump address* – адреса програмного переходу.

Нульові старші 15 бітів вказують на інструкцію даного типу. Виконання інструкції типу J переводить програмний лічильник замість наступної позиції на позицію, вказану в полі *Jump address*. «Стрибок» буває умовним та безумовним. В першому випадку перехід відбувається при виконанні певної умови, яка визначається предикатом. В першому розділі було виведено матричний предикат рівності двох матриць, саме він використовується для визначення умови. При цьому також наявний обернений до нього, а саме предикат нерівності матриць. А для безумовного переходу відповідно ніякої умови не потрібно.

При застосуванні інструкції стрибку буферам всередині керуючого блоку забороняється вести запис. Така особливість пов’язана зі запобіганням так званих pipeline hazards [17] (див. 286 с.). Це ситуація коли при стрибку інші стадії ядра виконують операції так, ніби переходу й не було. Дана

проблема є дуже значущою при побудові конвеєра, тому її вирішення є пріоритетним.

Інструкції типу IR призначені для взаємодії з зовнішнім пристроєм, а також очікування надходження сигналів.

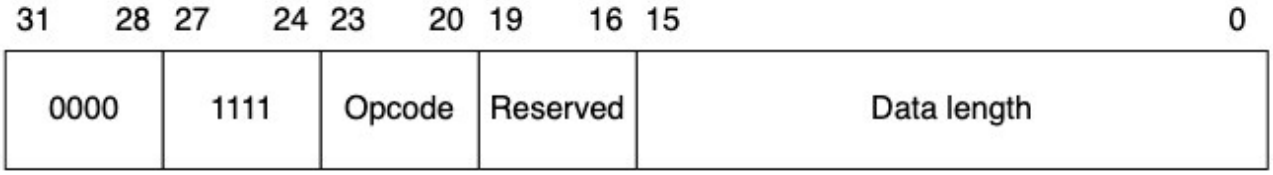


Рисунок 3.9 – Структура інструкції типу IR

Команда типу IR містить 4 поля:

- *Opcode* – операція інструкції типу IR;
- *Reserved* – біти, що не використовуються;
- *Data length* – розмір вхідних/вихідних даних у кількості комірок пам’яті реляцій.

Розмір вхідних/вихідних даних потрібен блоку вводу-виводу для обробки запитів на запис та надсилання даних. Цей параметр потім записується до лічильника даних, який відповідає за адресацію даних в просторі вхідних/вихідних даних пам’яті. При команді очікування переривання від блоку вводу-виводу поле *Data length* не використовується. Останні 8 бітів встановлені в 0b0001111 вказують на причетність інструкції до IR.

Таблиця 3.6 – відповідність значення *Opcode* та операцій IR

Opcode	Операція
0001	Data Request
0010	Data Send
0100	Send Ready
…	

У даного типу операцій всього 3 операції. Операція Data Request надсилає зовнішньому пристрою сигнал, що процесору необхідні нові дані.

Операція Data Send надсилає зовнішньому пристрою запис на надсилання даних. Операція Send Ready просто надсилає зовнішньому пристрою сигнал, що всі операції виконані, а дані оброблені.

3.2 Керуюче ядро

Покращення даного блоку полягає в додаванні нового функціоналу та прибиранні непотрібного. Суть додавання та прибирання полягає в розподілі завдань між апаратною та програмною частинами. Так було прибрано блок перевірки розмірностей матриць, тепер ця функція покладається на компілятор. Проте була додана можливість завантажувати/вивантажувати вміст регістрів до/з стеку. При цьому є апаратна підтримка технології fast stacking. Суть цієї технології полягає у завантаженні/вивантаженні частини регістрів, у яких зберігаються дані, необхідні для подальшого виконання функції при виклику іншої.

Відмова від перевірки розмірностей. Даний блок було прибрано оскільки його задачу можна виконувати й програмно, при компіляції коду; проте сам блок займає багато місця та потребує апаратного множення, що є дуже ресурсозатратним. Нижче наведена таблиця перевірки розмірностей:

Таблиця 3.7 – Предикат перевірки розмірів матриць від операції

Операція	**Функція**
Множення матриць	$dest.row == A.row$ $\wedge dest.col == B.col \wedge A.row == B.col$
Поелементні	$dest.row == A.row == B.row$ $\wedge dest.col == A.col == B.col$
Транспонування	$dest.row == A.col \wedge dest.col == A.row$

Необхідність в апаратному множенні була обумовлена кодуванням кількості стовпчиків.

$$col_{cd} = col \cdot ceil(\frac{row}{k}),$$

де col_{cd} – закодоване значення кількості стовпчик;

col – кількість стовпчиків;

$ceil$ – операція округлення до більшого

row – кількість рядків

k – параметр векторізації;

Тому цей блок повинен містити до трьох включно блоків множення, що є нераціональним використанням ресурсів. І хоча в новій реалізації, навіть без операції транспонування, не потрібні блоки множення; проте перевірка розмірностей реляцій також потребують забагато ресурсів.

Також проблема полягагає в тому, що при введенні нових операцій блок перевірки розмірностей доведеться також розширювати. При цьому, як було сказано раніше, дана задача може вирішуватись на етапі компіляції коду. Таке рішення не вплине на кінцеву продуктивність та не відбирає кремнієві ресурси.

Стек. Введення стеку обумовлює потребу зберігати поточні дані регістрів в іншому місці, окрім як в регістрах. Цим місцем слугує частина пам’яті матричних імен, відведена під стек. Зберігання поточних даних до стеку є перевіреним практикою інструментом в комп’ютерній архітектурі. Такий прийом найчастіше використовується при переході до нової функції. Зберігання даних у стек дозволить запам’ятати точку повернення та повністю відновити перерваний контекст виконання. Проте одного виділення пам’яті під стек не вистачить. Необхідно забезпечити функціональну підтримку читання/запису до стеку.

Для оперування зі стеком необхідно знати адресу початку стеку та адресу кінцевого елементу стеку. Для спрощення механізму взаємодії зі стеком, його розмір вибирають таким чином, щоб певна кількість старших бітів усіх можливих адрес стеку була однаковою. Тоді для читання та запису до стеку можна керувати залишеними молодшими розрядами. У даному проекті під стек виділяється 64 32-бітові комірки пам’яті. Для адресування по

такому стеку необхідно 6 бітів. Усі ж старші біти виділяються під маску стеку. Стек було вирішено розмістити у вершині пам’яті. Враховуючи, що пам’яті імен потрібно 16 біт, маємо:

Базова адреса (маска стеку): 0xFFC0;

Максимальна адреса: 0xFFFF;

Наступним необхідним компонентом є адреса кінцевого елементу, вона зазвичай зберігається в спеціальному регістрі під назвою «stack pointer». Для можливості адресації всього стеку він якраз займає 6 біт. Адреса стеку формується шляхом конкатенації значення stack pointer та маски стеку <stack mask, stack pointer>. Коли до стеку кладеться один елемент, значення stack pointer збільшується на одиницю, а при вилученні зменшується на одиницю. Такий принцип дії властивий для двонаправленного лічильника, тому stack pointer реалізується як лічильник «stack counter».

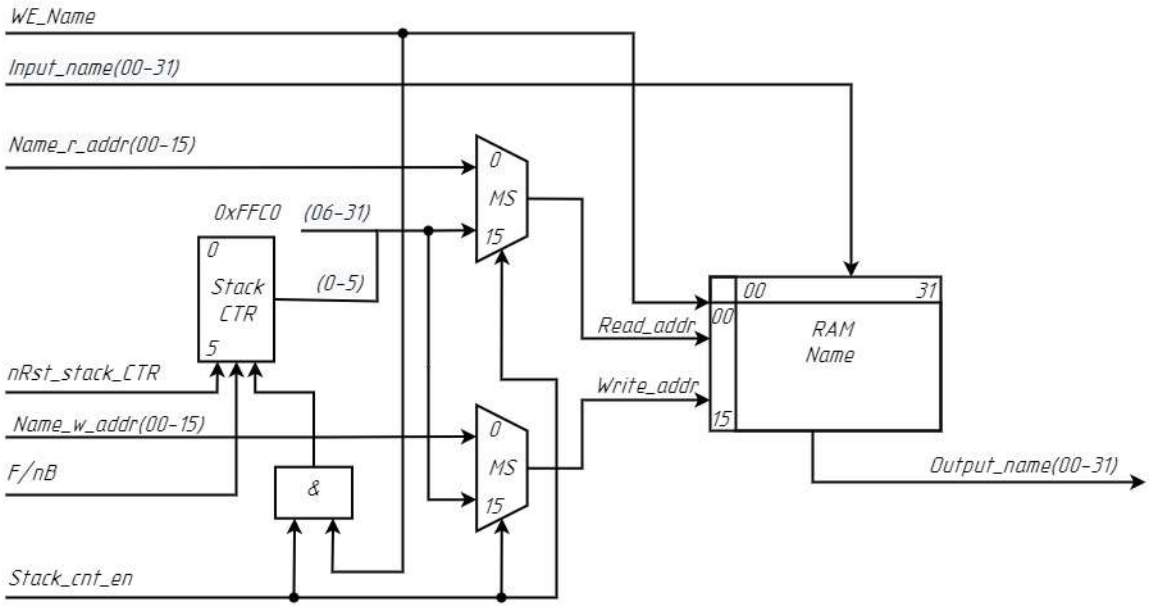


Рисунок 3.10 – Функціональна схема пам’яті імен зі стеком.

Схема на рисунку 3.10 враховує два режими роботи: звичайний та зі стеком. Коли вхід *Stack_cnt_en* знаходиться в одиничному стані, схема перемикається у режим роботи зі стеком, а лічильник починає лічити. У звичайному режимі до пам’яті записуються дані з блоку входу/виходу *Input_name* за адресою *Name_r_addr*; а зчитуються дані за адресою

Name_w_addr. При режимі роботи зі стеком до пам’яті записуються дані stack_data за адресою, вказаною у *stack_CTR*. Коли вхід *F/nB* знаходиться в одиничному стані, лічильник рахує у прямому напрямку (Forward); при нулі у зворотньому (Backward). А при нульовому *nRst_stack_CTR* лічильник скидається. Для того, щоб лічильник почав лічити, необхідно щоб входи дозволу лічення *Stack_cnt_en* та вхід дозволу запису *WE_Name* були активними.

Для взаємодії зі стеком було також модифікована схема керуючого ядра. До неї додано лічильник регістрів та схема управління операціями зі стеком. Функціональну схему модифікованого керуючого ядра представлено в додатку А. Лічильник регістрів, як і *stack counter*, є двонаправленним. Лічильник регістрів лічить від адреси початку зберігання до адреси кінця зберігання при прямому ліченні; і навпаки при зворотньому. Даний механізм дозволяє записувати до стеку лише ті регістри, що дійсно необхідно зберегти, таким чином не вивантажуючи повністю регістровий файл. Дана функція основана на динамічному стекуванні в Cortex [32]. Модуль stack control забезпечує читання/запис даних за наступним форматом:

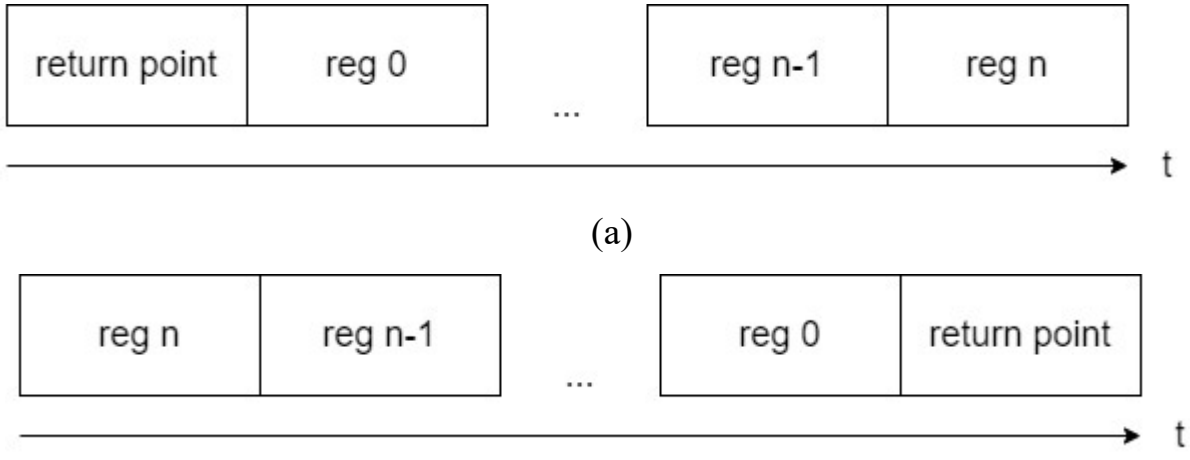


Рисунок 3.11а – Формат даних для запису до стеку; 3.11б – для читання зі стеку.

Діаграми на рисунку 3.11 показують що формати читання та запису фактично є інверсними, що відповідає порядку їх запису та вилучення зі

стеку. Цим і пояснюється необхідність у двонаправленому ліченні. Return point зберігає значення програмного лічильника, до якого програма повинна повернутись та лічильника регістрів до переходу до нової функції, чим гарантує повне повернення до початкового стану.

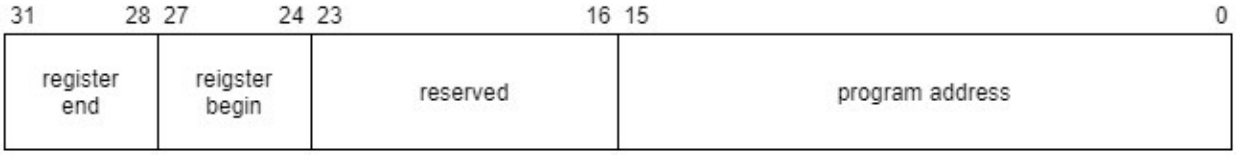


Рисунок 3.12 – Структура return point.

У молодших 16 розрядах зберігається значення програмного лічильника до якого програма повинна повернутися після завершення виконання функції *program address*. В старших полях останні 4 розряди відведені кількість регістрів *register end*, що були записані до стеку до виконання функції. Поле *register begin* (24-27 біти) вказує початковий регістр, з якого починається запис до стеку, аж до *register end*. Біти з 16 по 23 не використовуються, тому вони зарезервовані для можливого майбутнього використання.

При цьому всьому для стекування даних з регістрів (або вивантаження зі стек), необхідні лише 2 команди, що вказані в таблиці 3.5. При виконанні операцій зі стеком програмний лічильник не працює, оскільки інструкції роботи зі стеком є багататактними. Тому для атомарного виконання необхідно призупинити роботу усього конвеєра керуючого ядра.

3.3 Виконавче ядро, обчислення та зберігання даних реляції

Структура виконавчого ядра. У матричному процесорі блок підтримки ізоморфізму брав на себе забагато задач одночасно та мав завелику кількість входів та виходів. Тому, було вирішено реорганізувати структуру виконавчого ядра. Так, блок підтримки ізоморфізму був поділений на 2 окремих блоки, що виконують конкретну задачу. Тепер виконавче ядро розділяється на 3 складові: блок обчислення адрес, блок обчислення даних,

та блок контролю операцій. Причому, його архітектура зберігає логіко-предметний принцип, згідно з яким виконавча та керуючі частини розділяються та поєднуються прямим та зворотнім зв’язком.

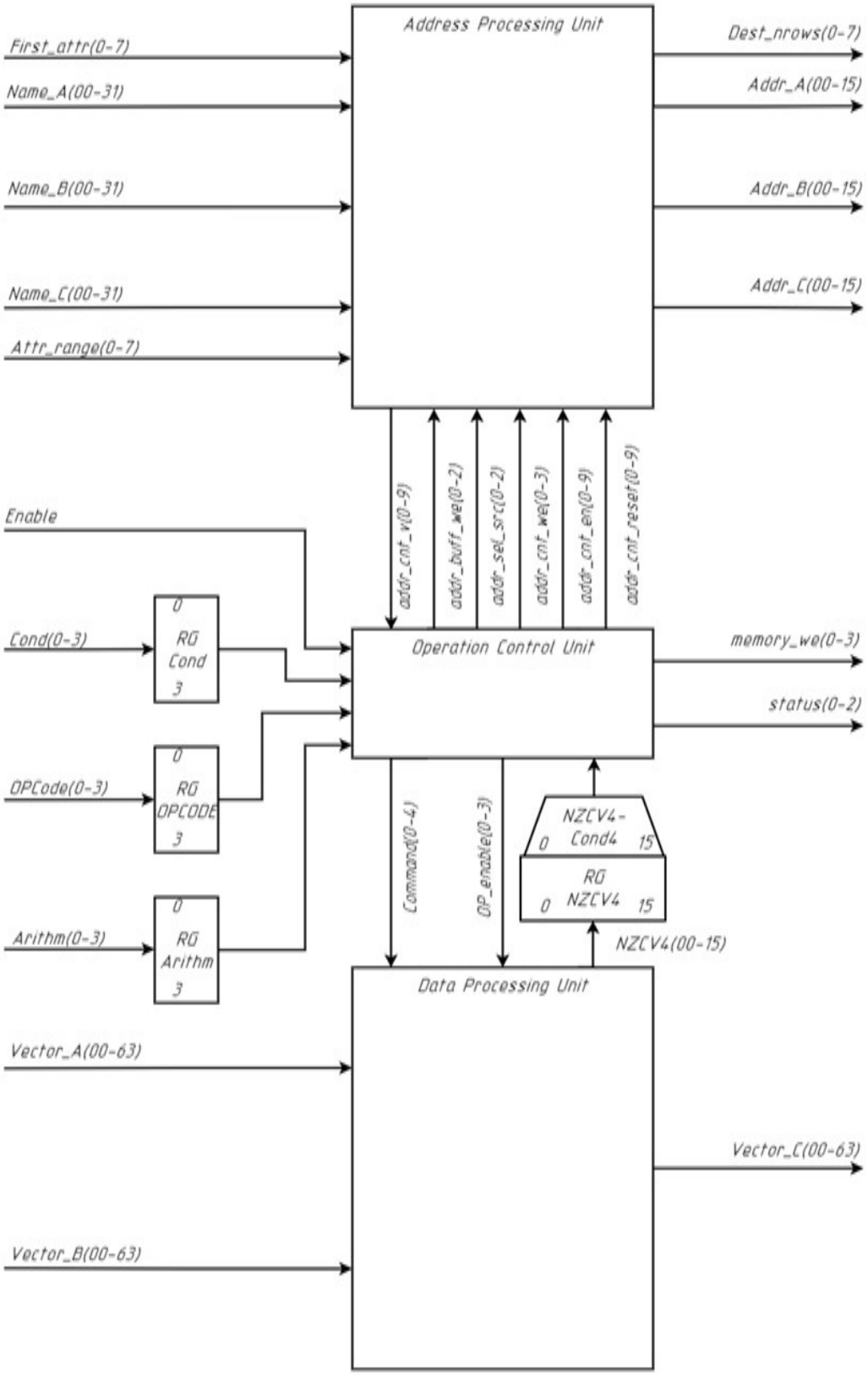


Рисунок 3.13 – структура обчислювального блоку.

Так, блок обчислення адрес приймає на вхід імена матриць, а повертає адреси елементів кожної з матриць. Обчислювальний блок приймає на вхід два вектори, та повертає результуючий. А блок контролю операцій є кінцевим автоматом, що керує ними обома.

Арифметичний блок. Даний блок зазнав деяких змін, в порівнянні з матричним. Схема транспонування була прибрана через її непотрібність. З самої арифметичної схеми буі прибраний регістр зсуву, оскільки з новим методом адресації він більше не потрібен. При цьому, з виходів АЛУ додаються спеціальні виходи, що сигналізують про стан результату, вищезгадані NZCV. З кожного АЛУ виходи статусу збираються в єдину шину. Принцип векторизації обчислень зберігається і в поточній версії. Так, в одній комірці зберігається k елементів реляції, де k є параметром векторизації. Цей параметр визначає скільки елементів міститься в одному машиному слові даних. Також згідно з цим параметром будується й арифметична схема. Всередині неї одне машине слово даних розбивається на k елементів, кожне з яких опрацьовується окремим АЛУ, а потім збирається знову в одне машине слово даних.

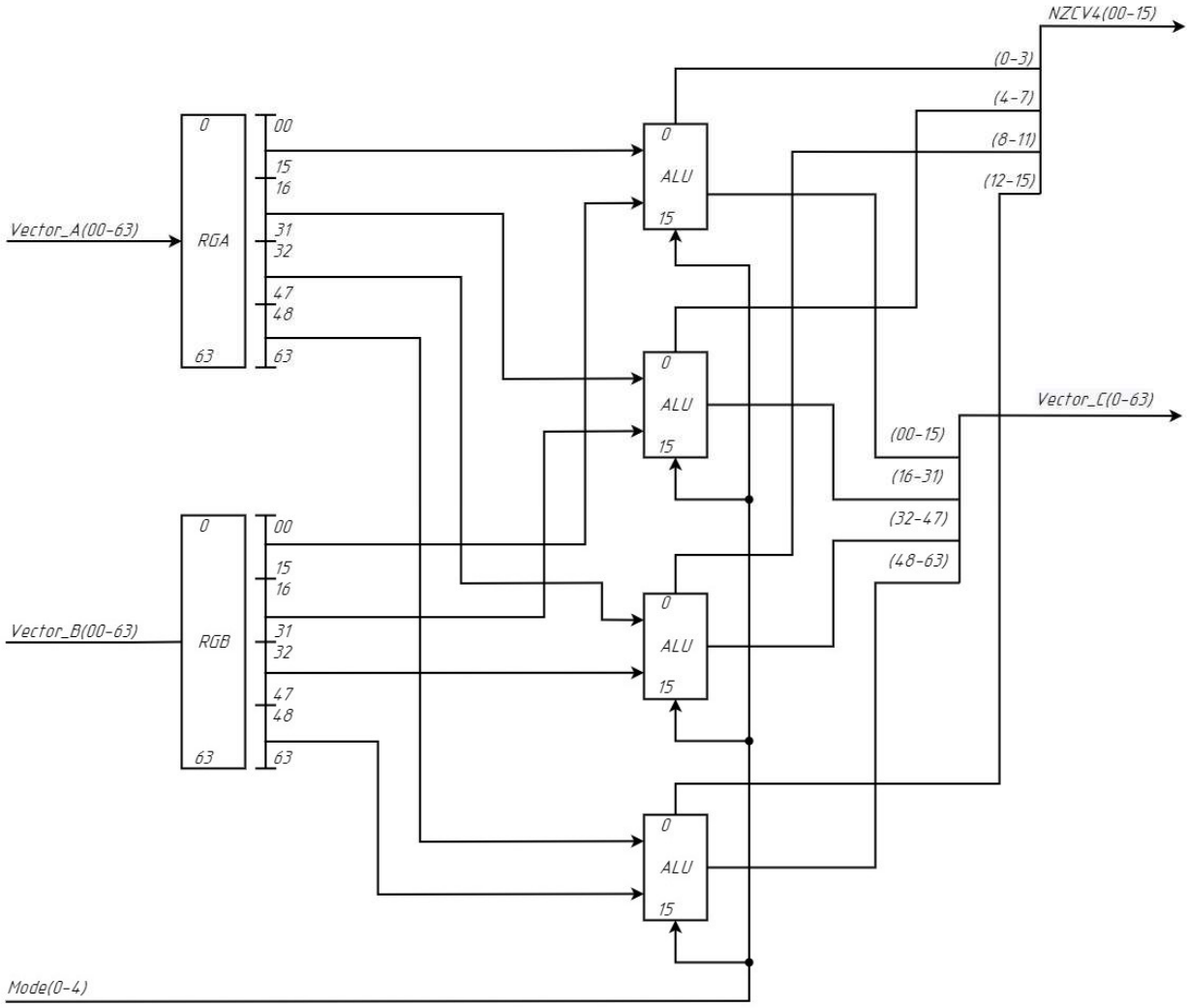


Рисунок 3.14 – Функціональна схема арифметичного блоку

Схема на рисунку 3.14 приймає на вхід 2 вектори *Vector_A*, *Vector_B* та шину задання режиму роботи АЛУ *Mode*.

В АЛУ наявні спеціальні арифметичні блоки з можливістю накопичення результату (Multiply-accumulate, MAC). У типовій реалізації MAC два вхідні операнди перемножуються та додаються до попереднього значення шляхом зворотнього зв’язку.

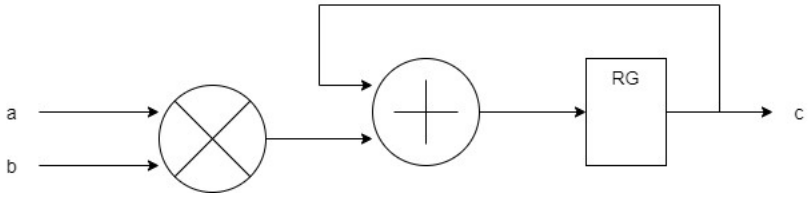


Рисунок 3.15 – Типова реалізація MAC

У схемі вже наявні множення і суматор, які використовуються для виконання однієї дій. Додаючи до схеми інші арифметичні операції, як віднімання чи ділення, можемо значно розширити потенціал АЛУ. При цьому, схема віднімання реалізується на суматорі, який вже наявний у MAC, тож зворотній зв’язок з регістром накопичення може бути використаний і с дією віднімання. Також додаємо до схеми АЛУ мультиплексорів таким чином, щоб можна було міняти різні функції обчислення за допомогою їх перемикання. Таким чином маємо повністю налаштовуючийся АЛУ.

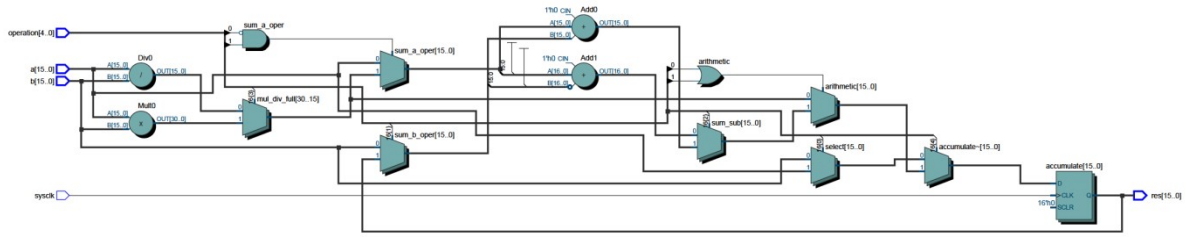


Рисунок 3.16 – Схема ALU.

Схема на рис. 2. була згенерована інструментом RTL viewer програмного оточення Quartus 21.1. Функція ALU задається входом Mode. Кожен біт входу Mode перемикає певну функцію, які в решті утворюють одну складну. Відповідність бітам та перемиканням показана в таблиці 3.8.

Таблиця 3.8 – Прив’язка бітів Mode до перемикання локальних функцій

Біт Mode	Тип перемикання
0	Перемикання джерела
1	Включити/виключити накопичення
2	Сума/віднімання
3	Множення/ділення
4	Арифметика/селекція

Зауваження: нульовий біт з таблиці 3.8 працює в по-різному при різних значеннях біту ввімкнення накопичення. Так, коли перший біт знаходиться в логічному «0», результат буде зчитуватися з суматору при логічній «1» на нульовому біті, і з множення/ділення при логічному «0». А

коли перший біт знаходиться в логічній «1», то при логічному «0» на нульовому біті другим операндом для додавання/віднімання, окрім регістру накопичення, вибирається вихід множення/ділення; а якщо нульовий біт знаходиться в логічній «1», то другим операндом стає вхідне число A. З врахуванням з'єднань всередині ALU, можна знайти функцію виходу в залежності від входу Mode, що наведено в таблиці 3.9.

Таблиця 3.9 – Режими роботи ALU

Код	Дія
0???0	$c = a$
0???1	$c = b$
11001	$c = a - b$
11101	$c = a + b$
10?00	$c = a / b$
11?00	$c = a \times b$
1?111	$c = a + c_{i-1}$
1?011	$c = a - c_{i-1}$
10010	$c_i = a / b - c_{i-1}$
10110	$c_i = a / b + c_{i-1}$
11010	$c_i = a \times b - c_{i-1}$
11110	$c_i = a \times b + c_{i-1}$

Де c_{i-1} означає попереднє знечення, а знаком «?» позначено біти, які не впливають на вихід при встановлених інших бітах. Останнє пов'язано з тим, що перший біт Mode вимикає накопичення, а нульовий переводить вхід регістру на множення/ділення.

Схема пам'яті реляцій. У даній роботі представлена нова схема пам'яті, в порівнянні з матричним процесором. Її ключовою відмінністю є те, що тепер мінімальний крок в адресації дорівнює одному елементу реляції. Раніше, мінімальний крок становив розмір під вектору, параметру

векторизації k, тобто мінімально можна було зсунутись на k елементів. Досягається це завдяки зміненій лозіці розбиття та маніпулювання чанками пам’яті. Відтак, на кожен елемент підвектору $a_i, i = \overrightarrow{1,k}$ відводиться окремий чанк пам’яті, які потім логічно збираються в один великий. Без зсуву нова пам’ять видає той же результат, що й стара. Проте зі зсувом схема пам’яті видає на виході суміш з елементів поточного та наступного підвекторів, розміщенних у тому порядку, в якому вони повинні бути у реляції. Наступний рисунок наглядно демонструє принцип зсуву підвектору.

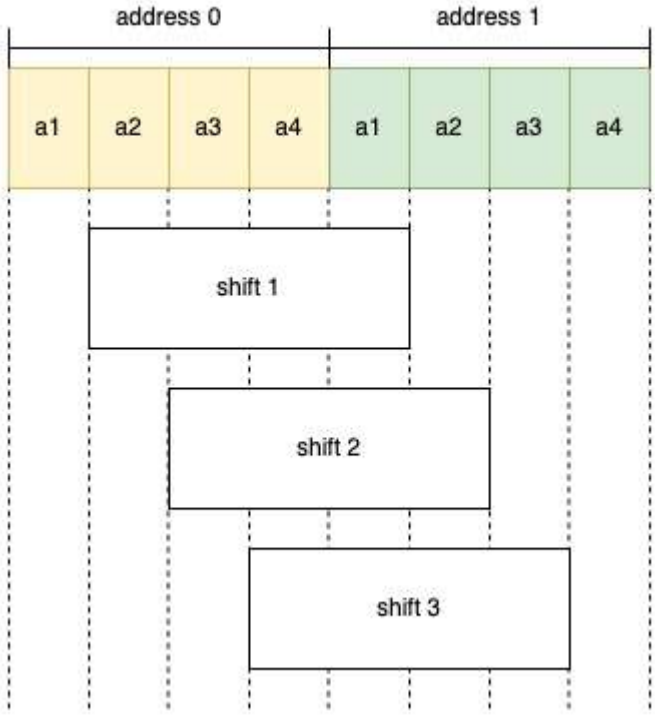


Рисунок 3.17 – принцип зсуву підвектора

На рисунку 3.17 представлена частина пам’яті з параметром векторизації $k = 4$, а конкретно значення у двох сусідніх адресах. У прикладі показано які елементи та в якому порядку з обох адрес входять до підвекторів з різним значенням зсуву. Так, при одиничному зсуві до вихідного підвектору входять 3 кінцеві елементи з адреси *address 0*, (a2, a3, a4), і в кінці підвектору розміщюється перший елемент з адреси *address 1*, (a1). При зсуві на 2 елементи вже до результуючого вектору входять 2 останні елементи з адреси *address 0* на початку, (a3, a4), та 2 перші елементи

з адреси *address 1* в кінці, (a1, a2). І при зсуві на 3 елементи в результуючому векторі наявний лише 1 останній елемент з адреси *address 0* на початку, (a4), всі інші з *address 1*, (a1, a2, a3).

Розберемо принцип кодування та декодування зсунутих векторів для пам'яті. Позначимо зсув підвектору як s, (від англ. shift, зсув). Тоді, перші $k - s$ елементів беруться з поточної адреси, та s елементів з наступної. Для взяття елементів наступної адреси потрібно інкрементувати адресу виходу. Проте розміщення чанків є статичним, як і конкатенація їхніх виходів. Тобто дані з першого чанку завжди будуть на першому місці, а так далі для інших. Рішенням статичної конкатенації є циклічний зсув. Так, щоб елемент першого чанку опинився в кінці необхідно конкатенацію даних циклічно зсунути вправо на $s = 1$.

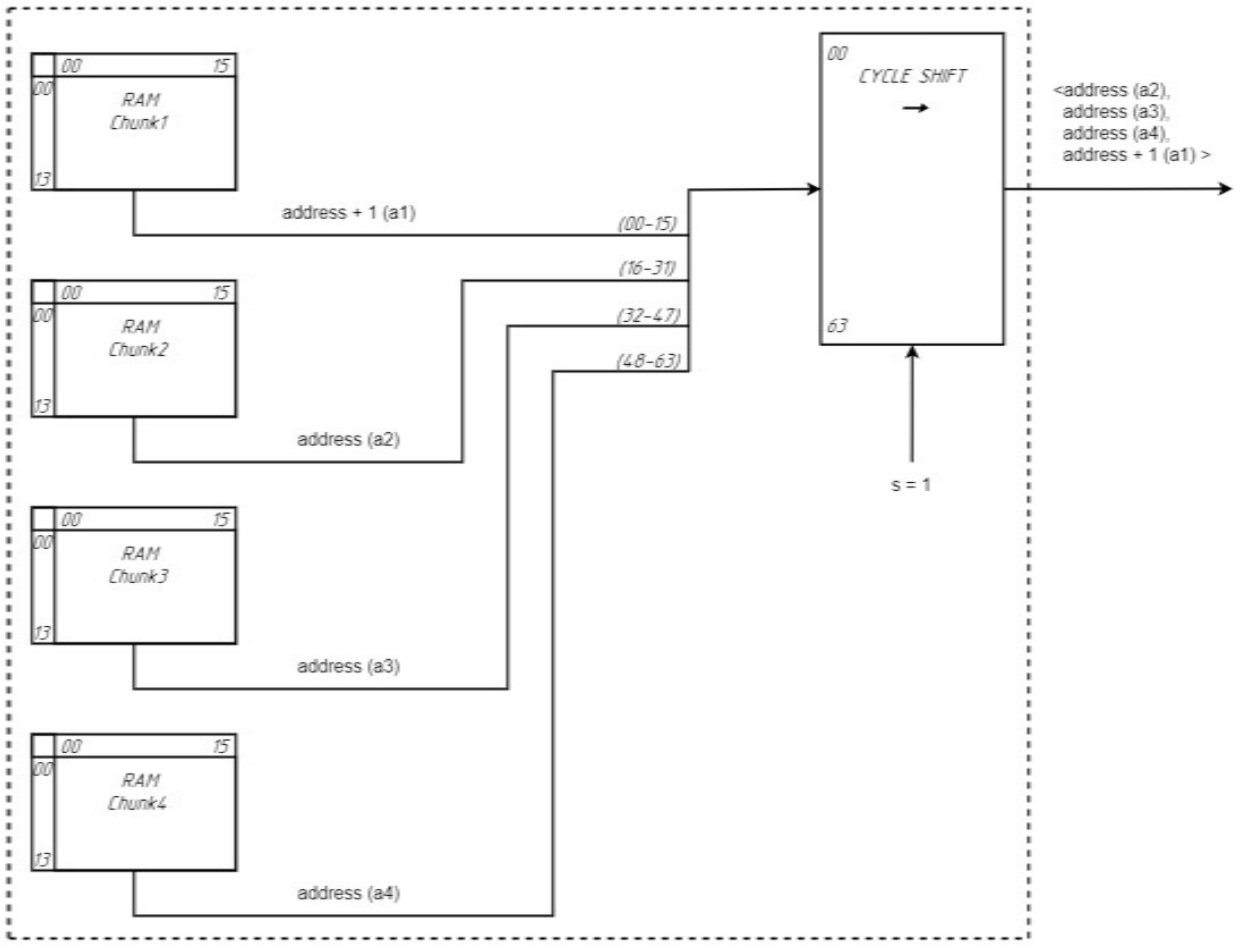


Рисунок 3.18 – приклад циклічного зсуву конкатенації даних з чанків пам'яті

З прикладу видно, що дані з чанків пам’яті *RAM Chunk* конкатенуються за порядком, згідно їх номеру. У *RAM Chunk 1* зберігається перший елемент всіх підвекторів, і так для всіх чанків. Тобто без зсуву вектора конкатенація видає правильний результат, у плані впорядкованості елементів. У прикладі ж вектор зсунуто на $s = 1$, тому з першого чанку *RAM Chunk 1* береться наступна адреса, після поточної *address*. Проте так при конкатенації виходить, що елемент, який мав бути в кінці, тепер перший. Саме це і виправляє циклічний зсув. Схема переносить перший елемент вектору, тобто *address + 1 (a1)*, у кінець вектору, при цьому зсуваючи інші елементи на одну позицію вперед. Таким чином отримується вектор з правильним порядком елементів, (a2, a3, a4, address + 1 (a1)).

Таким чином зсунутий вектор, що зчитується з незсунутої пам’яті, утворюється у 2 кроки:

1. Збільшити адреси зчитування для перших s чанків на одиницю;
2. Циклічно зсунути вправо конкатенацію вихідних даних на s.

А для декодування зсунутого вектору в звичайний потрібно поміняти порядок дій та змінити циклічний зсув з права на ліво. Таким чином для запису зсунутого вектору до пам’яті необхідно:

1. Циклічно зсунути вліво зсунутий вектор на s;
2. Збільшити адреси запису для перших s чанків на одиницю.

При цьому, необхідно врахувати порядок розміщення даних у пам’яті, тобто так званий Endianess англійською. Так, елементи у пам’яті розміщенні в порядку Little-Endian, що можна спостерігати з рисунку 3.18.

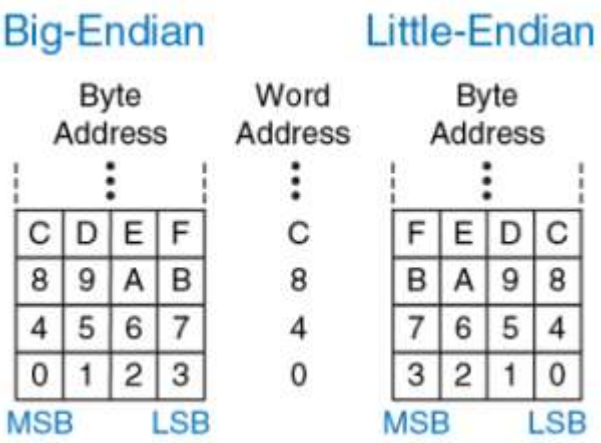


Рисунок 3.19 – різниця зберігання даних між Big-Endian і Little-Endian

На рисунку 3.19, MSB перекладається як Most Significant Bit, тобто найстарший у машиному слові; LSB, навпаки, означає Least Significant Bit, що означає наймолодший. Як видно, у Little-Endian дані зберігаються у прямому порядку, тобто молодші байти зберігаються на початку слова, а старші в кінці. Коли як у Big-Endian молодші байти розміщуються в кінці.

З врахуванням Little-Endian, вищезазначені алгоритми кодування та декодування працюють коректно, що доказується у реальній розробці.

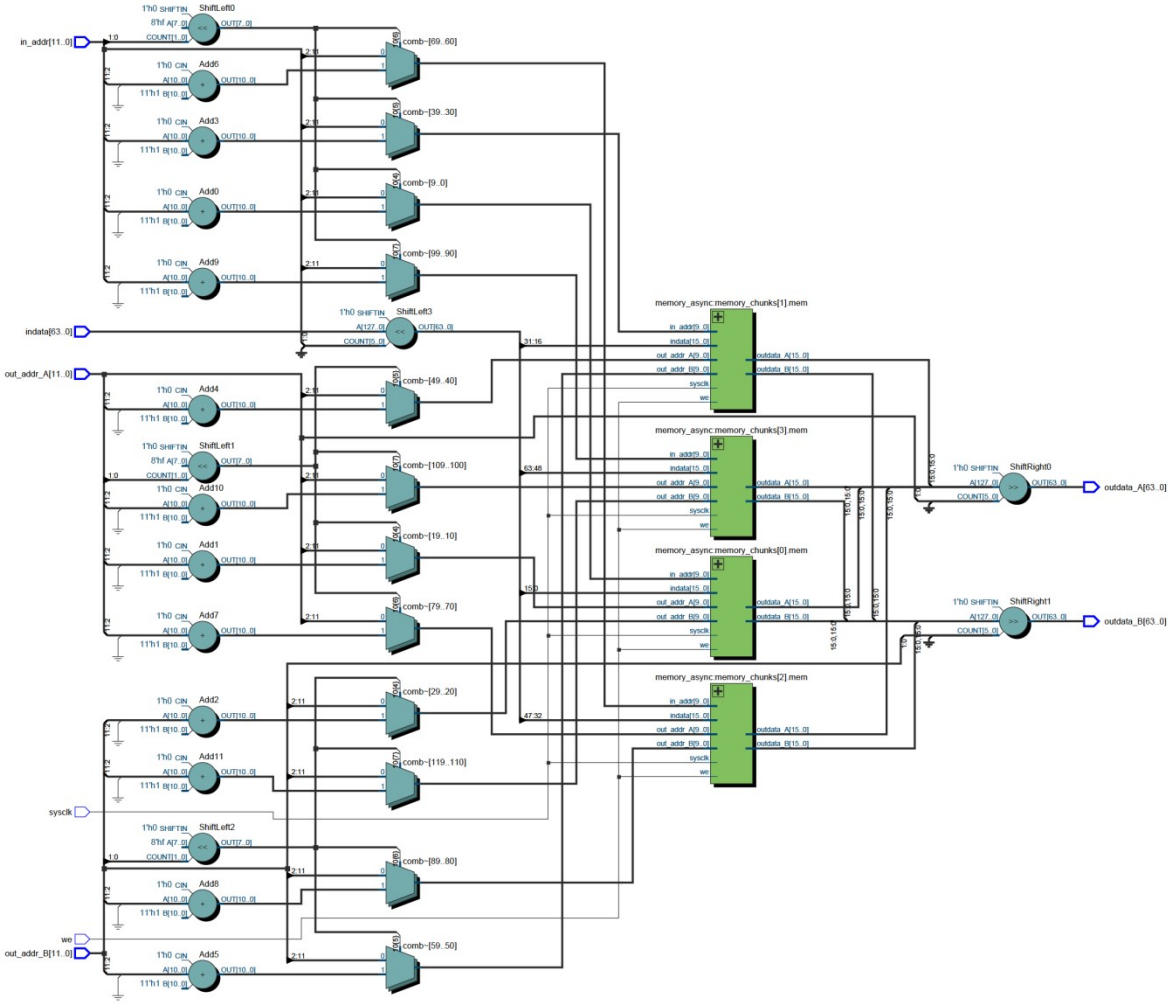


Рисунок 3.20 – Модифікована пам’ять, зображеня в RTL Viewer.

На рисунку 3.20 зображено графічне відображення змісту модулю модифікованої пам’яті на мові опису цифрових пристроїв Verilog HDL. Дане зображення згенеровано в САПР Quartus 21.1 інструментом RTL Viewer. Дане зображення не є прямим відображенням того, як уся логіка реалізована апаратно, проте є дає графічне уявлення як дана схема влаштована логічно. З результату графічного відображення помилок у схемі не знайдено, тож можна провести верифікацію пам’яті.

Рисунок 3.21 – верифікація модулю модифікованої пам’яті.

Результат симуляції отриманий використанням інструменту Waveform САПРу Quartus 21.1. Хоча даний інструмент не є серйозним засобом верифікації, проте він все ж дозволяє перевірити основну функціональність нової пам’яті. Тут, у 3 підвектори, тобто комірки пам’яті найвищої абстракції, були записані дані, відповідно до табиці 3.10.

Таблиця 3.10 – Запис даних за адресами до тестової пам’яті

Адреса	Дані
0x0	0x4444333222221111
0x4	0x8888777766665555
0x8	0xCCCCBBBBAAAA9999

Тобто дані були записані за адресами, кратними k, та імітують запис вектору $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)$. Після чого дані зчитувались з адрес, не кратних k, результат наведено в табиці 3.11.

Таблиця 3.11 – Зчитування даних за адресами з тестової пам’яті

Адреса	Дані
0x0	0x4444333222221111
0x1	0x5555444433332222
0x2	0x6666555544443333
0x3	0x7777666655554444
0x4	0x8888777766665555
0x5	0x9999888877776666
0x6	0xAAAA999988887777
0x7	0xBBBBAAAA99998888
0x8	0xCCCCBBBBAAAA9999

Отже, зігдно таблиці 3.11, новий стиль адресацій працює успішно.

3.4 Блок обчислення адрес

В універсальних процесорах роль обчислення адрес та обчислення даних виконує одне й те саме залізо. Операції з адресами можуть виконуватись універсальним процесором, оскільки адреси є фактично цілими числами. Хоча таке рішення і є універсальним, проте воно має властиві для універсальних процесорів проблему — повільна швидкість обробки складних типів даних. Зазвичай під обробкою складних типів даних мається на увазі обробка усіх значень. Тобто необхідно пройтися по усім адресам підтипів складного типу. Так, наприклад, для знаходження адреси одного елементу складного типу даних, визначеного на N^n, необхідно врахувати індекси по усіх його n вимірів. Така операція може стати доволі складною для універсальних процесорів, і таким чином значимо знизити їхню продуктивність. Об’єктно орієнтовані процесори заточені під специфічні обчислення, зокрема stream processor виконує паралельно багато однотипових операцій, нейронні процесори зазвичай заточені під виконання операції множення з накопиченням, а матричний процесор обробляє підвектора матриць. Хоча на такому залізі й можна обчислювати адреси, проте саме заточеність під певний тип даних робить це рішення нелогічним і не ефективним. Тому для збільшення продуктивності такі архітектури зазвичай мають блоки обчислення адрес, що дозволяють влаштувати конвеєрне виконання дій над склданими типами даних.

Для виконання операцій над матрицями або реляціями необхідно послідовно проходитись по усім чи певним елементам, тобто дані слідують один за одним. У матричному процесорі блок обчислення адрес складається з лічильників, що виконують функцію слідування (покрокове збільшення початкової адреси на одиницю) для кожного простору. Також модифікація лічильників з можливістю перезапису їх змісту дозволяє нелінійне проходження по елементам, що підвищує критерій універсальності.За кожен вимір (декартову множину) відповідає свій лічильник, таким чином адресація

відбувається нативно. При самому розрахунку адрес -вимірний простір проецюється на одновимірний (пам’ять компютеру є одновимірним масивом, тобто вектором). Проекція відбувається за рахунок “витягнення” багатовимірного типу даних у вектор. Під витягненням розуміється розкладання багатовимірного типу даних на одновимірні вектори з їхнею подальшою конкатенацією. Нехай існує матриця $M_{l\times k}$, розкладемо її на kвекторів довжиною l. Потім до цих k векторів застосовуєм операцію конкатенації $v_{mem} =*^{i=2,k} cat(v_1, v_i)$, в результаті на початку вектору розміщуються елементи v_0, а в кінці елементи вектору v_k. І таким чином, щоб считати перший рядок другого стовпчика матриці $M_{l\times k}$ необхідно від адреси першого елементу v_{res} відступити на l елементів. Для зчитування всього j-го рядка треба послідовно додавати до адреси його першого елементу (тобто j) l, $row_j =*^{i=2,k} cat(v, v_{mem}(j + i \times l))$, де vдо початку циклування містить j-й елемент першого стовпчика. Дуже важливим уточненням є те, що матриця в пам’яті може зберігатись не на самому початку. Тобто матриця розміщюється в пам’яті з деяким зміщенням відносно початкової адреси. Очевидно що це зміщення дорівнює адресі першого елемента матриці. Так, зчитування рядка, з урахуванням зміщення адреси, виглядає наступним чином: $row_j =*^{i=2,k} cat(v, v_{mem}(addr_{init} + j + i \times l))$, де $addr_{init}$ є адресою початкового елементу матриці у пам’яті v_{mem}.

Так, для матриць, а також для реляцій, використовується 2 лічильники: для стовпчиків (атрибутів) та рядків (записів). Схемотехнічно, найбільша арність операцій є бінарною. Для бінарної операції потрібно 3 операнди: 2 вхідних операнда функції (*A, B*) та один вихідний (*Dest*). Тож схемотехніка блоку обчислення адрес повинна підтримувати всі вищезазначені операнди. А отже даний блок налічує як мінімум 6 лічильників. Додатково до них можуть бути наявними службові лічильники. У матричному блоку обчислення адрес наявний лічильник, який відлічує такти для формування квадратних підматриць. Його необхідність обумовлена тим, що в одній

комірці пам’яті зберігається декілька елементів матриці, ця кількість характеризується параметром векторизації виконання операцій k_{vect}. Таким чином лічильник відлічує k_{vect} тактів, а потім переповнюється. Сигнал переповнення слугує прапорцем для блоку контроля операцій про проходження k_{vect} тактів. А для повернення з відлічення квадратної підматриці додаються буфери, що зберігають стан лічильника рядків до відлічення квадратної підматриці. Більшою мірою формування квадратних підматриць застосовується в операції транспонування задля ефективного використання пам’яті [1].

Лічильник для рядків. Задача даного лічильника, згідно з назвою, перебір значень стовпчиків. Тепер лічильник рядків є здвоєним, проте він фактично є лічильником рядків попередньої версії, розбитий на дві частини. Більша частина відповідає за адресацію по векторам, а менша за зсув векторів. Під час розрахунку адреси, значення лічильника зсуву конкатенується зі значенням результуючої адреси, при чому значення лічильника зсуву займає молодші біти.

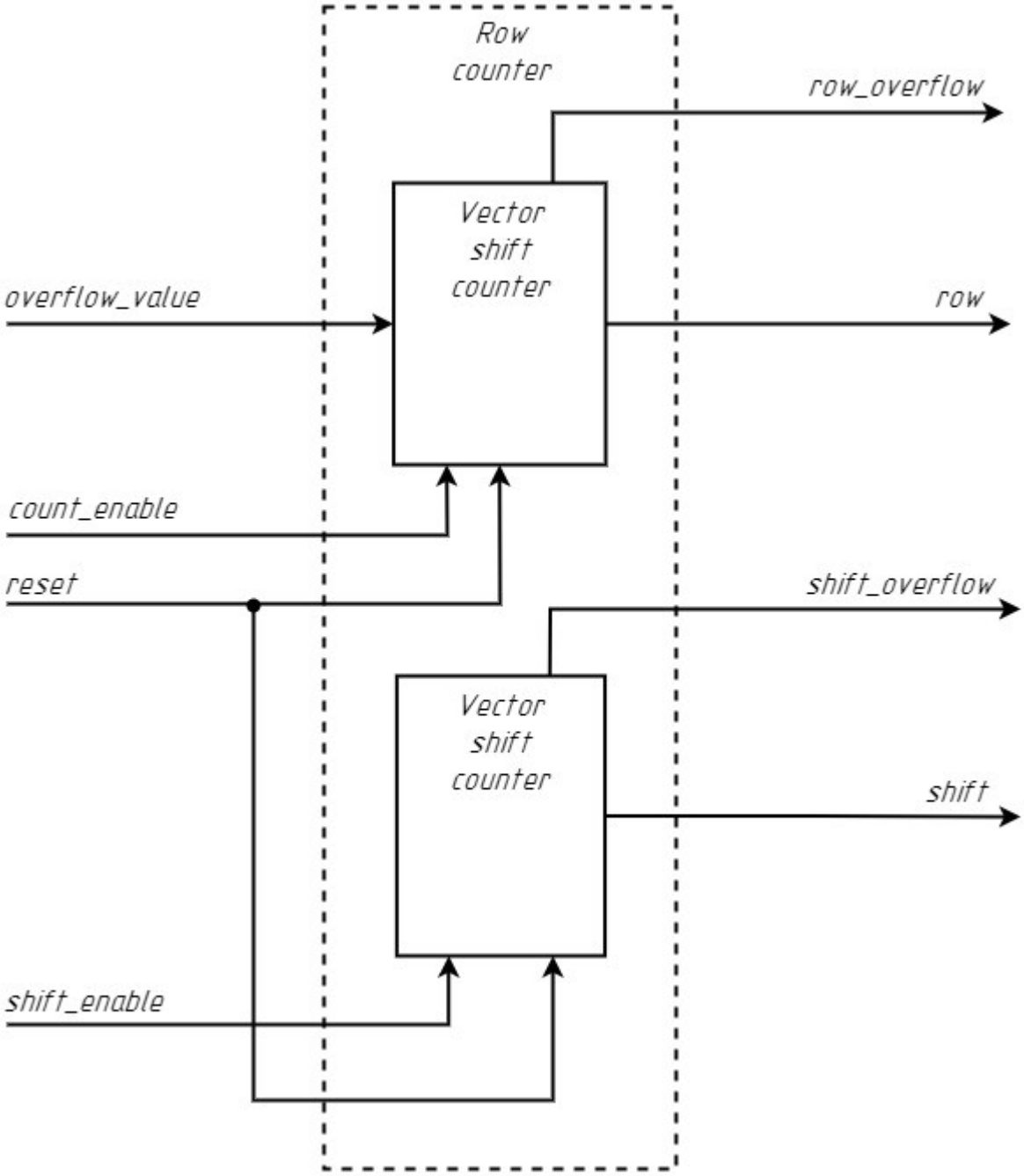


Рисунок 3.22 – Схематичне зображення лічильника для рядків

На вході лічильника з рис 3.22 можна задавати значення переповнення *overflow_value*, *count_enable* та *shift_enable* для дозволу ліченення лічильників вектору та зсуву відповідно. При досягненні значення переповнення лічильник скидається в нуль, а вхід reset одразу скидає обидва лічильники в нуль. На виході ж самі значення лічильників *row* та *shift,* та їхні прапорці переповнення. Ці прапорці дозволяють блоку керування операціями відстежувати стан лічильників для необхідної обробки сигналів.

Лічильник для стовпчиків. Оскільки матриця в пам’яті витягується у довгий вектор, то для переходу на наступний стовпчик необхідно відразу «перескочити» усі елементи поточного. Є 2 варіанти реалізації такого проходження: множити значення лічильника на кількість рядків, або відразу робити крок, рівний цій кількості. Серед суттєвих недоліків першого варіанту є те, що обчислення адреси потребує забагато часу (одна операція множення та дві додавання). До суттєвих недоліків другого варіанту належить досить складне обчислення значення переповнення $(m-1)n, | A_{m\times n}$, що потребує множення та віднімання. Навіть якщо кодувати це значення безпосередньо в іменній моделі, тоді, у випадку 16 бітів на адресу та по 8 для розмірностей, сама іменна модель буде витрачати на 8 бітів більше через необхідність зберігання результату множення, або на 25% більше. Таке збільшення вкрай негативно сказиться на ефективному використанню пам’яті. Проте об’єднавши ці 2 рішення можна отримати нове, а саме використання двох лічильників: один слугує для лічення адреси, а другий для лічення до переповнення. Таким чином схема множення не потрібна взагалі, а також зберігається нативність лічення.

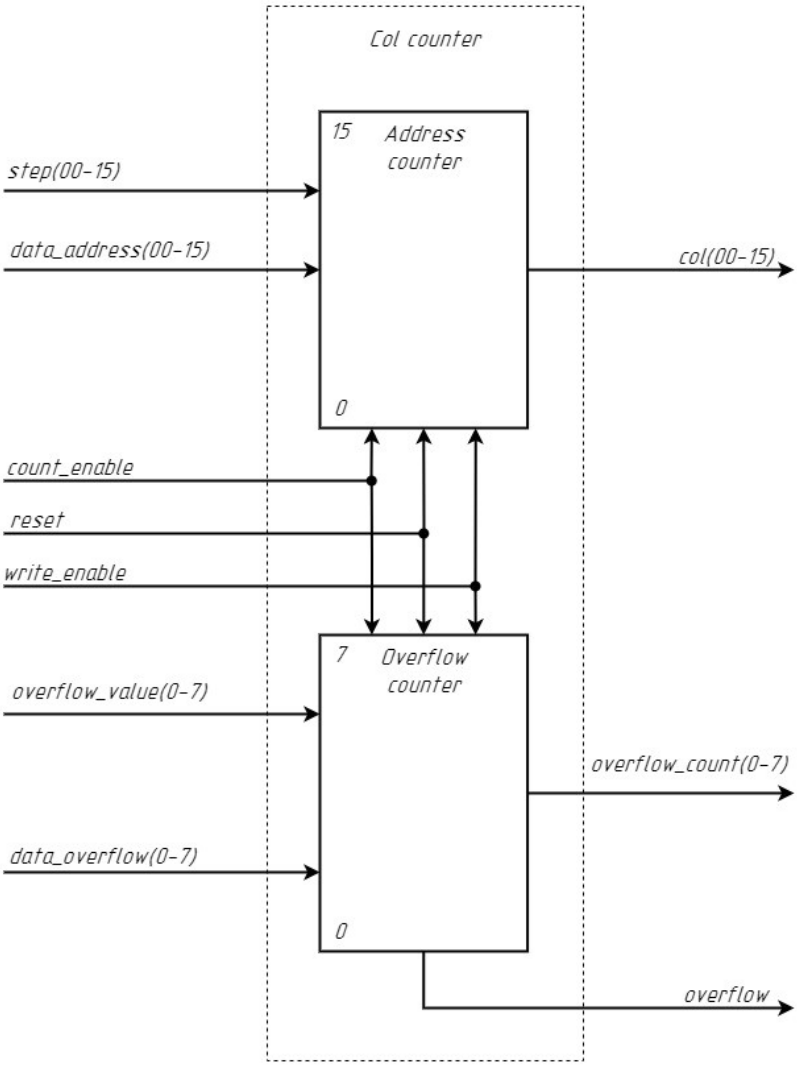


Рисунок 3.23 – Схематичне зображення лічильника для стовпчиків

На рисунку 3.23 представлений лічильник для іменної множини де, адреса займає 16-біт, а кількість стовпчиків 8. Як видно, лічильник складається з двох окремих під лічильників: адреси (*Address counter*) та переповнення (*Overflow counter*). Лічильник переповнення відповідає лише за прапорець *overflow* переповнення, а його вміст з виходу *overflow_count* зчитується лише буфером, у якому тимчасово зберігається стан всього лічильника стовпчиків. Значення переповнення задається на вході *overflow_value* а перезаписати його вміст можна за допомогою входу *data_overflow*. Хоч *overflow_value* називається значенням переповнення, проте на ділі це значення, в яке переходить лічильник після переповнення, адже він є інверсним, тобто рахує від *overflow_value* до нуля. Це дозволяє трохи зекономити ресурсів на перевірці значення переповнення та запису

даних. В лічильнику адрес наявне налаштування кроку лічення через відповідний вхід *step*. Кожного разу, додаючи до вмісту лічильника адрес значення кількості рядків, з кожним кроком лічення значення лічильник переводить адресу на наступний стовпчик. Перезапис вмісту лічильника відбувається через вхід *data_address* при подачі активного сигналу на *write_enable*. А за дозвіл на лічення відповідає вхід *count_enable*. Подачею активного сигналу на вхід *reset* можна скинути лічильники в нуль. На виході лічильник адрес має своє значення *col*.

На вхід *step* подається кількість рядків, щоб таким чином з кожним тактом лічильник вказував на перший рядок наступного стовпчика.

Лічильник тактових сигналів. В деяких ситуаціях необхідно відстежувати проходження *k* тактів. Саме для цього використовується лічильник тактів.

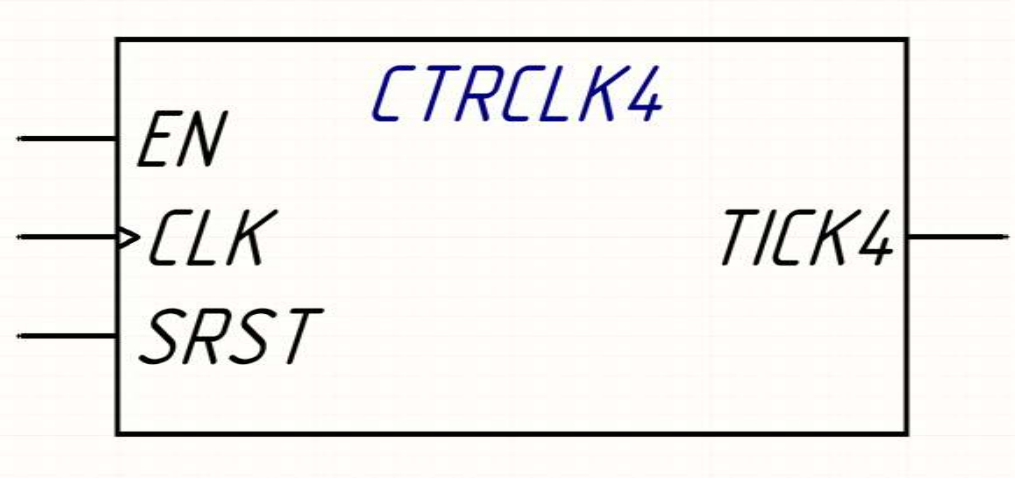


Рисунок 3.24 – Схематичне зображення лічильника тактових сигналів

Даний лічильник має лише один вихід *TICK4*, який призначений для статусу переповнення. З входів наявні лише вхід для тактової частоти *CLK*, синхронне зкидування *SRST* та дозвіл на лічення *EN*.

Лічильники саме для реляцій трохи відрізняються, що обумовлено типовою обробкою реляцій, а саме основна робота відбувається з атрибутами. Так, у реляціях часто потрібно виділити певний атрибут, замість усього запису. Також у реляціях нема сенсу виділяти квадратні підматриці, як і взагалі немає особливої потреби в її транспонуванні. Тому цей механізм

вирізано з реалізації для реляцій. Так, у реляціях замість лічильників стовпчиків та рядків представлені лічильники по атрибутам та записам.

Лічильник по атрибутам фактично заміняє лічильник рядків. Так, цей лічильник є подвійним, та може проходитись як по коміркам, так і по елементам завдяки лічильнику зміщення вектора. Проте, на відміну від лічильника по рядкам, він має механізм перезапису свого вмісту та супутниковий буфер для тимчасового зберігання свого значення. Перше обумовлено необхідністю відразу зчитати атрибут з певним позиційним номером, а друге необхідно для зчитування групи атрибутів, які в тому числі можуть утворювати складний атрибут по типу класу чи символьної строки.

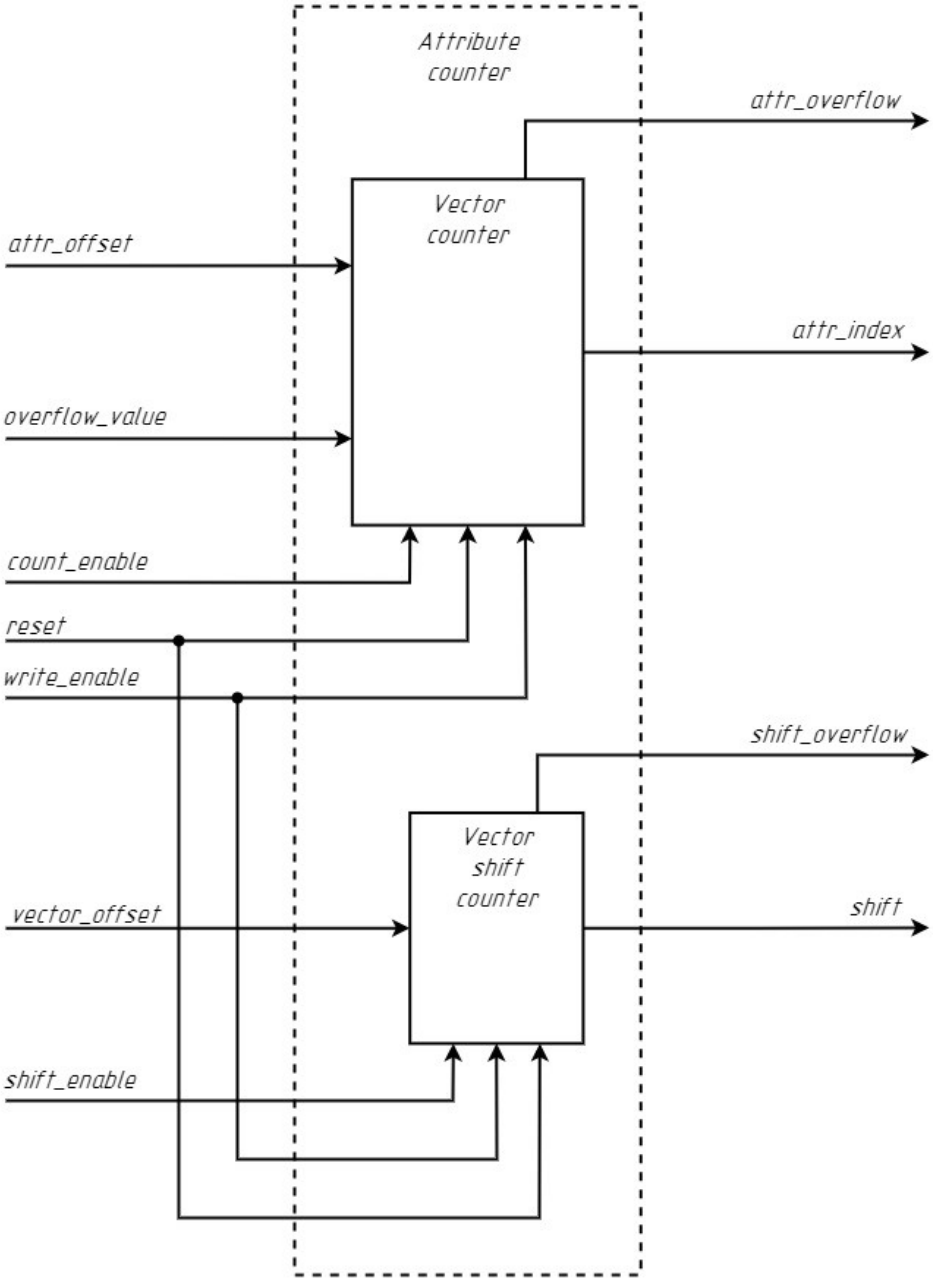


Рисунок 3.25 — Схематичне зображення лічильника атрибутів

Лічильник на рисунку 3.25 має наступні входи: запис значення в лічильник векторів *attr_offset* та дозвіл на запис *write_enable*, значення переповнення лічильника по векторам *overflow_value*, дозвіл на лічення лічильнику по векторам *count_enable* та лічильнику зсуву вектора *shift_enable*, і скидання лічильнику *reset*. На виході схема має індекс вектора атрибутів запису *attr_index*, зсув вектору *shift*, та сигнали переповнення лічильника атрибутів *attr_overflow* та зсуву *shift_overflow*. Входи дозволу та

виходи сигналізації переповнення дозволяють блоку контролю операцій мати над лічильником повний контроль, не вникаючи в конкретику з самими даними на вході та виході.

Лічильник же записів (або кортежів), на відміну від лічильника стовпчиків матриць, є дещо спрощенним та не має механізму перезапису свого змісту. Єдине, що їх об’єднує, це можливість задавати крок лічення та збільшена розрядність до розміру адреси.

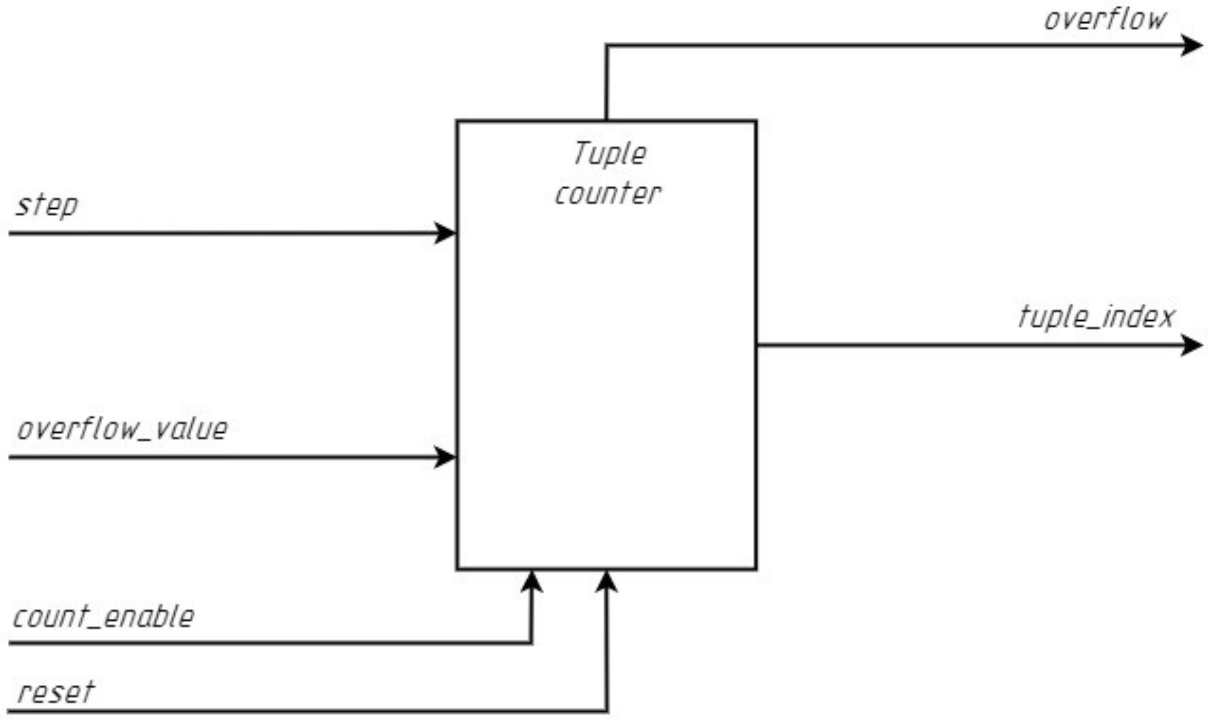


Рисунок 3.26 — схематичне зображення лічильника записів (кортежів)

На вході лічильник з рисунку 3.26 приймає на вхід значення кроку *step*, переповнення *overflow_value*, дозвіл на лічення *count_enable*б та зкидання *reset*. На виході ж він має індекс кортежу *tuple_index* та прапорець переповнення *overflow*. Як і лічильник атрибутів, вхід дозволу лічення та вихід прапорця переповнення дозволяє повністю дозволяє блоку контроля операцій керувати цим лічильником. А крок завжди рівний розміру кортежу, що дозволить відразу перейти до того ж атрибуту наступного запису.

Також до схеми додається звичайний інверсний лічильник із перезаписом змісту. Такий лічильник слугує для того, щоб відміряти

необхідний діапазон атрибутів, з якими працює обчислювач. Інверснивність пояснюється дещо спрощеною схемотехнікою таких лічильників для переповнення.

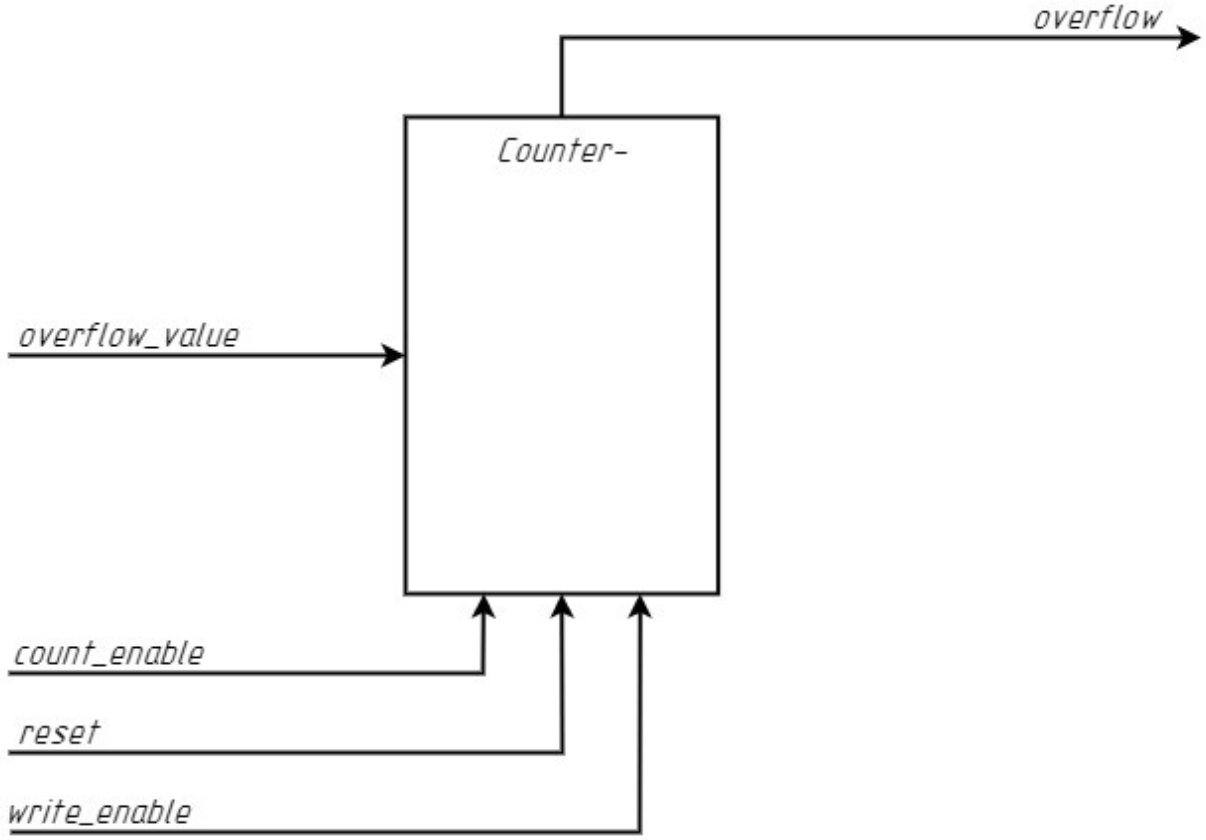


Рисунок 3.27 – схематичне зображення інверсного лічильника

На вході лічильника з рисунку 3.27 наявні: значення переповнення *overflow_value*, дозвіл на лічення *count_enable*, вхід скидання *reset*, та дозвіл перезапису *write_enable*. На виході лише прапорець переповнення *overflow*, що сигналізує про проходження по всіх необхідних атрибутах.

З усіх цих лічильників складається блок обчислення адрес. Як вже було зазначено, на кожен операнд приходиться по 2 лічильника: атрибутів та кортежів; їхні значення потім додаються до адреси першого елемента, таким чином вказуючи на адресу поточних елементів, з якими працює обчислювач. Враховуючи інверсний лічильник, усього блок налічує 7 лічильників.

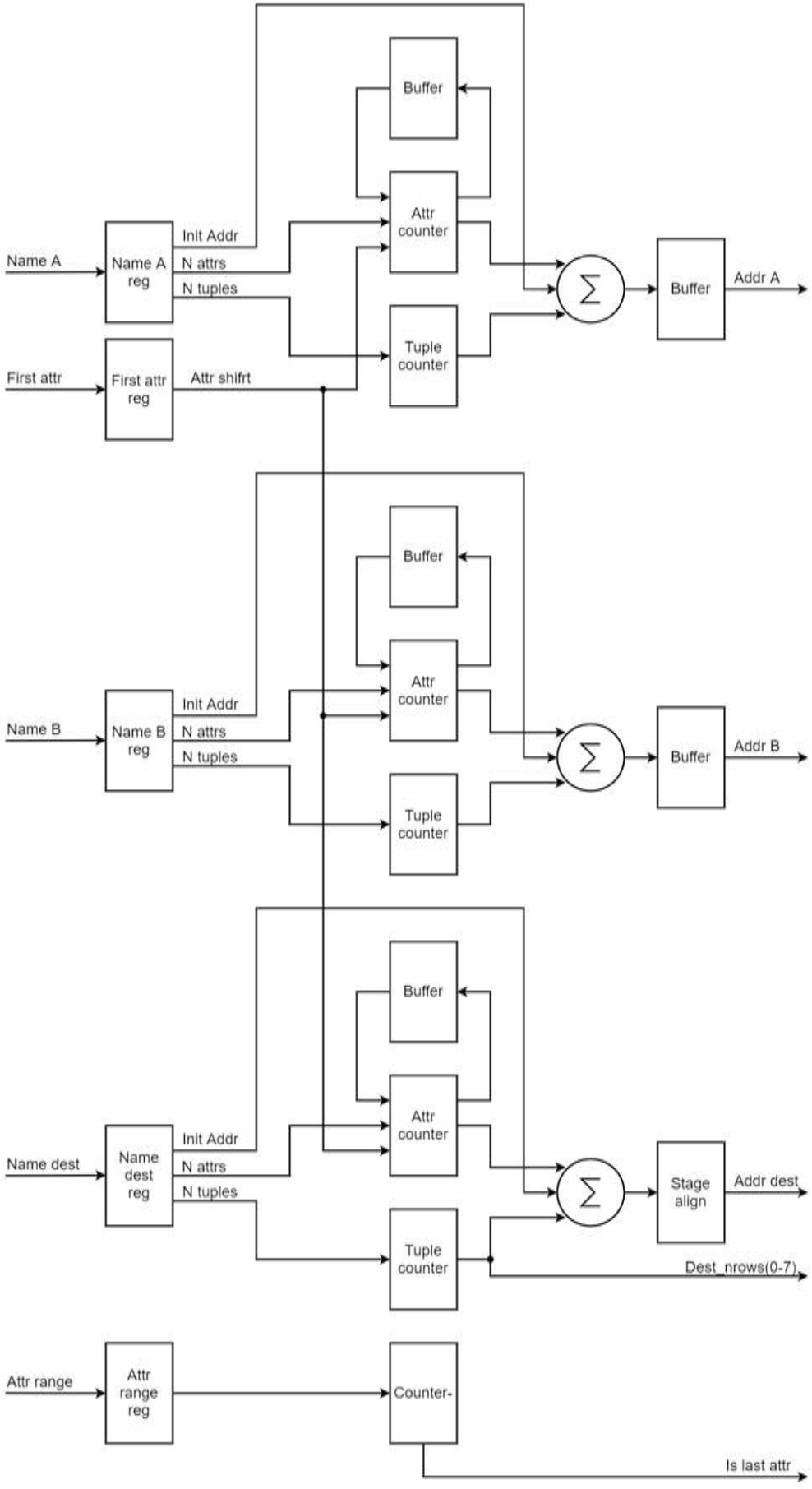


Рисунок 3.28 – Структура блока обчислення адрес

На вході блоку з рисунку 3.28 присутні імена трьох операндів: входу A (*name A*), B (*name B*), та виходу dest (*name dest*); позиція першого атрибуту в кортежі, з якого починається перевірка (*First attr*), та загальна кількість атрибутів(*Attr range*), по яким треба «пройтися». Імена таблиць розбиваються на початкову адресу (*Init Addr*), кількість атрибутів(*N attrs*), кількість кортежів (*N tuples*). На виході ж адреси для кожного імені (*Addr A, Addr B, Addr dest*), прапорець, що вказує на те, чи долічено до останнього атрибуту (*Is last attr*), а також вихід, де вказано кінцеву кількість строк для реляції результату (*Dest_nrows*). Прапорець *Is last attr* є прапорцем переповнення інверсного лічильника, перейменованим в рамках схеми для зрозумілості контексту. Вихід *Dest_nrows* необхідний, бо кінцевий розмір деяких реляцій після маніпуляцій над ними не відомий. Блок *Stage align*, це спеціальна схема, що складається з послідовно з'єднаних регістрів. Її задача вирівняти значення адреси запису результату зі стадіями конвеєру виконавчого ядра. З 4-х стадій, а саме **prefetch → fetch → execute → store**, значення адреси запису обчислюється на першій стадії (**prefetch**), а необхідно воно на останній стадії (**store**). Таким чином виникає затримка у 2 стадії (тобто 2 такти), яку необхідно компенсувати. У блоці *Stage align* якраз використовується 2 додаткових регістри, і загалом виходить 3 регістра. А отже адреси будуть «рухатись» по конвеєру згідно своєї стадії. **Зауваження:** у схемі випущено керуючі сигнали, заради спрощення схеми. Повна схема блоку обчислення адрес представлена схемі електричній принциповій, арк. 4.

Розберемо трохи детальніше роль входів *First attr* та *Attr range.* За їхньою допомогою можна відразу зчитати необхідні атрибути у заданому діапазоні. Тобто фактично можна вилучити частину кортежу $T(Tuple\ counter(First\ attr:\ First\ attr + Attr\ range))$, при цьому не тягнучи інші атрибути, якщо вони не потрібні. Це дозволить як зекономити час виконання, так і зробити зручнішим маніпулювання з таблицями.

3.5 Кінцевий автомат

Кінцевим автоматом у цифровій схемотехніці називається послідовна схема зі зворотнім зв'язком. Вихід такої схеми в наступний такт сигналу синхронізації залежить від її внутрішнього стану. Внутрішній стан фактично діє як пам'ять, що запам'ятовує стан автомату, а завдяки зворотньому зв'язку можна заздалегідь задати чому буде рівний внутрішній стан в наступний момент часу. Існує два типи кінцевих автоматів: Мілі та Мура. Вихід автомату Мілі залежить не лише від внутрішього стану, а й від вхідних сигналів. Тоді як вихід автомату Мура залежить лише від його внутрішнього стану. Є 2 класичні варіанти представлення автоматів: за допомогою логічних формул та діаграм переходів. Логічні формули напряму описують як розряди наступного стану залежать від попередніх. Діаграми переходів є більш розвиненим представленням автоматів, адже вони показують зв'язки між станами, тому при їхньому використанні проектування кінцевого автомату стає більш інтуїтивним.

Сама діаграма переходів зображується як орієнтований граф. Вершини V такого графу відповідають за його стан, а ребра E вказують у який стан перейде автомат у наступний момент часу. Вершина V визначається єдиними параметром, а саме значеннямs, що записане у внутрішньому стані (регістрі) $V = \{s\}$. Тепер допустимо, що всі значення внутрішнього стану визначені на множині $S, s \in S$. В автоматі Мілі з однієї вершини може виходити декілька ребер, перехід до яких визначається предикатом p. Таким чином можна визначати ребро як множину з пари вершин (у випадку з орієнтованим це вхідна та вихідна) та предикату $E = \{(V_{in}, V_{out}), p\}$. При цьому, очевидно, що для всіх можливих комбінацій входу та внутрішнього стану завжди є один, і лише один предикат, що рівний істинні.Також, для кожної пари вершин (з врахуванням направленості) може бути лише одне ребро, оскільки декілька ребер з однаковими парами можна об'єднати в одне, застосовуючи операцію

логічного “АБО” для предикатів паралельних ребер, $E_{res} = \{(V_j, V_k), p_{res}\}, p_{res} = \vee p_i, \forall p_i \in E_i = \{(V_j, V_k), p_i\}$. В автоматі Мура же з однієї вершини виходить лише один вузол, який не містить умов ($p = True$).

Також, у такому графі обов’язково можна виділити підграфи, що будуть замкнені; якщо таких немає, тоді у графі повинна бути петля (вузол, що вказує на одну й ту саму вершину) з безумовним переходом. Така особливість забезпечує визначенність внутрішнього стану для кожного відліку часу від початку роботи автомату, навіть якщо кількість вершин в графі скінчена.

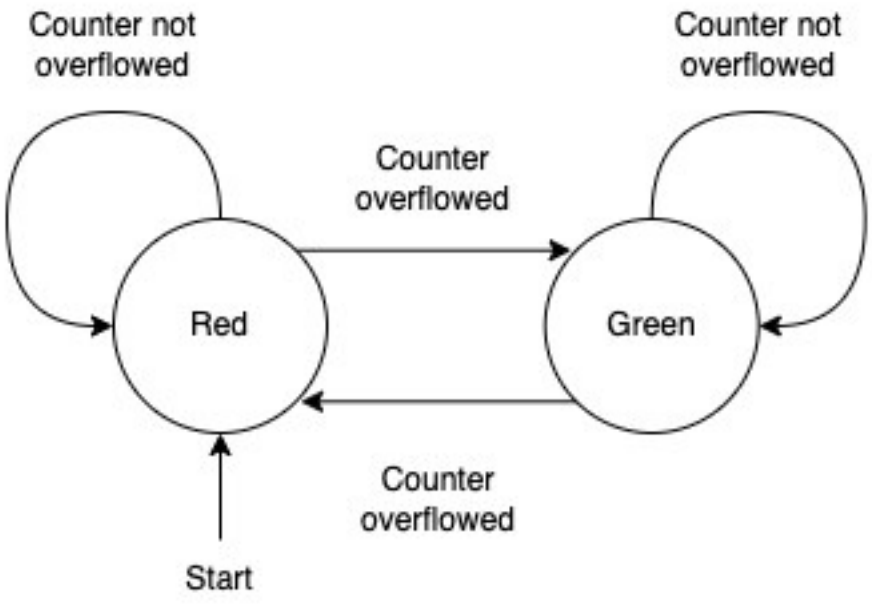


Рисунок 3.29 — Діаграма переходів для автомату двосигнального світлофора.

На рисунку 3.29 наведено приклад автомату Мілі з двома станами. До входу автомату під’єданий сигнал переповнення лічильника, і коли цей сигнал набуває істинного значення, світло міняється. А до переповнення автомат буде знаходитись в тому ж стані завдяки петлі. А замкненність графу говорить про те, що цей автомат міняє свій стан періодично. Таким чином

стан автомату можна визначити для будь якого моменту часу, знаючи період переповнення лічильника.

Далі візьмемо для прикладу автомат з 4-бітною коміркою для внутрішнього стану. З наступним тактом вміст комірки буде зсуватися на один розряд вліво, починаючи зі значення “1”. Такі автомати ще називають лічильниками зсуву.

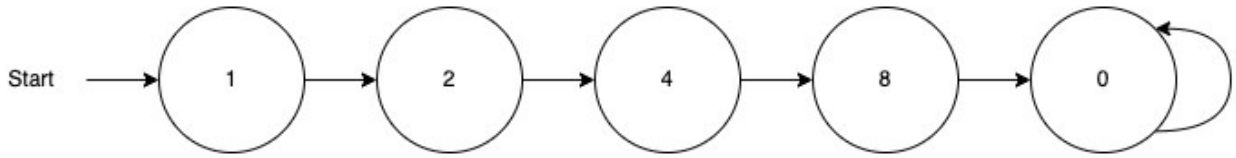


Рисунок 3.30 — лічильник зсуву.

Як видно з рисунка 3.30, автомат має лише 5 станів, після чого попадає в нескінчену петлю з вершини “0”. Такий автомат не є замкненим, проте петля забезпечує його визначенність для будь якого відрізку часу.

3.6 Функціональні графи

Тепер візьмемо автомат, що керує виконанням операцій у виконавчому ядрі. Якщо абстрагуватись від всієї схеми, то його виходи є просто набором логічних та цифрових значень. Проте в контексті саме всієї системи ці набори логічних та цифрових значень керують виконанням дій блоку обчислення адрес та блоку обчислення даних. Тому, в контексті виконавчого ядра, вони стають певними операціями, за які вони відповідають. З рисунку 3.13 видно, що виходи блоку керування, тобто автомату, у більшій мірі приєднані до блоку обробки адрес та блоку обробки даних. Також наявні виходи дозволу запису до пам’яті, що дозволяє перезаписувати лише певні елементи вектору, або акумулювати результат у буфері блоку обробки даних. І звісно присутні виводи статусу для керуючого ядра, щоб чітко проходитись по інструкціям програми. Тобто кожен стан автомату (вузол графу) вказує операцію над елементами, на які вказує блок обчислення адрес. А оскільки

даний кінцевий автомат керує даним блококом, тому можна стверджувати, що кожен стан повноцінно описує операцію над елементами матриці (чи багатомірного типу даних).

$$M_{res}[addr_{out}] = f(M_A[addr_A], M_B[addr_B], opcode),$$

де $M[addr]$ — селекторна функція, яка вибирає kелементів з матриці Mза адресою $addr$;

f — функція, яка приймає на вхід два підвектора розміром k та код операції;

$opcode$ — код операції.

Базисні операції.

Як відомо, адреса елементу реляції чи матриці обчислюється як сума початкової адреси, індексу рядка та стовпчика. Тому, зробимо деяке спрощення записів для кращого розуміння графів. Так, елемент -го рядка (підвектору) j-гостовпчика будемо записувати як $R[i,j]$. Для відокремлення індексів різних реляцій будемо використовувати нижні індекси. Так, вихідна реляція позначатиметься як $R_o[i_o, j_o]$; а реляції А та В відповідно $R_A[i_A, j_A]$, $R_B[i_B, j_B]$. Також введемо операцію інкрементаціїї індексу, що й робить блок обчислення адрес. Так, інкрементацію індексу запишемо як $++i$, що фактично є запозиченням з мови програмування С, де така операція позначає, що індекс спочатку інкрементується, а потім повертається на вихід $++i = i := i + 1, returni$.Тобто запис $R_A[++i_A, j_A]$ витягує наступний рядок після i_A-го рядка та j_A-йстовпчик, при цьому сам індекс i_Aзалишається збільшеним на одиницю. Також згадаємо про нову систему адресації, де можна задавати зсув підвектору, тобто по суті наявний поелементний зсув стовпчиків. Позначимо цей зсув як окремим індексом $j_S \in [1, k]$ (S від англійського слова single, один чи єдиний). Таким чином, повна адресація виглядає наступним чином $R[i, j + j_S]$.

Тепер розберемо типи функцій, які доступні для побудови функціональних графів. Очевидно, що ці функції напряму визначаються

блоком обчислення даних. Так, у доступі наявні всі функції з таблиці 3.9. Беручи до уваги нову систему адресації, а також можливість параметрично задавати індекси підвектору, які будуть опрацьовуватись, введемо параметричні операції зі всіх операцій з таблиці 3.9, записуючи параметр нижнім індексом. Так, наприклад, параметрична операція суми для додавання усіх елементів виглядає наступним чином $(a_1, \dots, a_k) +_{\overrightarrow{1,k}} (b_1, \dots, b_k) = (a_1 + b_1, \dots, a_k + b_k)$, а для випадку операції лише над першим елементом вектору таким чином $(a_1, \dots, a_k) +_1 (b_1, \dots, b_k) = (a_1 + b_1)$; тобто вихідний підвектор складається лише з одного елементу, який буде записано у відповідний елемент вихідної реляції. А також окрім виходу результуючого вектору, блок обчислення даних має виходи NZCV прапорців для кожного елемента підвектору $(a_1, \dots, a_k)$. Ці виходи використовуються для обчислення булевого значення предикатів порівняння. І призначені вони саме для керуючого блоку, тобто кінцевого автомату, для керування переходами між вузлами. Очевидно, що предикати рівності $=$ та нерівності $\neq$ є природніми як для натуральних чисел, так і для векторів. Проте складні предикати нерівності $<, >, \leq, \geq$ для векторів не визначені, тому тут наявна певна свобода в їхньому визначені. Визначити їх необхідно таким чином, щоб вони працювали як для випадку з одним елементом, так і з багатьма, тобто розширити операції на вектори. У такому випадку природнім є порівняння відповідних елементів, і якщо всі відповідні елементи відповідають умові, тоді оператор повертає істину, і хибу інакше. Продемонструємо це на прикладі операції менше або дорівнює $(a_1, \dots, a_k) \wedge_{\overrightarrow{1,k}} (b_1, \dots, b_k) = \bigwedge_{i=1}^{k} (a_i \leq b_i)$. Порівняємо результат роботи даної функції для першого елементу вектору $(a_1, \dots, a_k) \wedge_1 (b_1, \dots, b_k) = (a_1 \leq b_1)$, як видно, результат повністю відповідає порівнянню двох натуральних чисел.

3.7 Підтримка ППА та повнота функціональних графів

Після введення базисних операцій, необхідно ввести операції для маніпулювання ними, тобто сигнатурні операції. Як і раніше, відштовхнемося від ППА, через її потужність. Дана алгебра має 3 сигнатурні операції: суперпозиція $S^{m+1}(f^m, f_1^n, \dots, f_m^n)$, галуження $\diamond(p, f_1, f_2)$ та циклування $*(p^n, f_1^n, \dots, f_m^n)$, де n, m — арності відповідних функцій. Дані операції наявні в усіх сучасних мовах програмування, і тому якогось дискомфорту при побудові функціональних графів не викликають. Тож виразимо сигнатурні операції ППА через графи.

Суперпозиція. Підтримка даної сигнатурної операції найбільше обмежується апаратною частиною, оскільки саме ця операція найменше залежить від графів. Як було визначено в минулому розділі, операції суперпозиції потрібна пам'ять, щоб зберігати проміжні результати. Саме це і створює основні апаратні обмеження на використання суперпозиції. Розіб'ємо функції, над якими виконується суперпозиція, на 2 класи: обчислювальні та предикативні. В обчислювальних функціях об'єктом для проміжного зберігання є вектори та числові значення. У предикативних об'єктом є булеві значення. В обчислювальних функціях для зберігання проміжного результату апаратно використовується регістр накопичення, який дозволяє реалізувати акумулятивні операції. З основної пам'яті легальним шляхом вилучити попередньо обчисленні дані не можна. Хоч накопичувальний регістр і дозволяє реалізувати суперпозицію над даними, проте він же їх сильно обмежує. Так, функції над даними, які можуть працювати з суперпозицією (окрім першої), є лише ті, що використовують накопичення (таблиця 3.9). На першу ж функцію зі списку обмеження не накладаються, оскільки вміст регістру накопичення до початку виконання операції (програмного рівня) вважається нулем. Позначимо множину всіх можливих операцій блоку обчислення даних як F, тоді для виразу

$S^{m+1}(f^m, f_1^n, \dots, f_m^n)$ справедлим є $f_1^n \in F$, $f_1^n \in F, f_2^n, \dots, f_m^n \in F_{acc} \subset F$, де F_{acc} є підмножиною F, функції якої в аргументах обов'язково мають попередній результат. Тобто

$$F_{acc} = \{(a_n) + c_{-1},$$
$$(a_n) - c_{-1},$$
$$(a_n) *_n (b_n) + c_{-1},$$
$$(a_n)/_n(b_n) + c_{-1},$$
$$(a_n) *_n (b_n) - c_{-1},$$
$$(a_n)/_n(b_n) - c_{-1}\}$$

де c_{-1}— попередній результат, $*_n$ — поелементне множення векторів, $/_n$ — поелементне ділення. Як бачимо, F_{acc} складається всього з 6 функцій з 10 в F. Тобто суперпозиція явно обмежується апаратно. Проте, цього потенціалу повинно вистачати для підтримки базових функцій процесора, а більше від функціонального графу і не треба.

Візьмемо для прикладу функцію:

$$g(a, b, c, d, e, f, h) = a + b + c \cdot d - \frac{e}{l} - h$$

І спробуємо відтворити її за допомогою суперпозиції. Для початку, визначимо функції $f_1^n, \dots, f_m^n$. Так $f_1^n = (a_n) - c_{-1}$, $f_2^n = (a_n)/_n(b_n) - c_{-1}$, $f_3^n = (a_n) *_n (b_n) - c_{-1}$, $f_4^n = (a_n) + c_{-1}$, $f_5^n = (a_n) + c_{-1}$. Тепер визначимо функцію f^m, $f^m = f_5(a, f_4(b, f_3(c, d, f_2(e, l, f_1(h, c_0)))))$, де $c_0 \triangleq (0_n)$ — початковий стан накопичувального регістра, що за замовчуванням дорівнює нулю. Побудуємо функціональний граф, що відповідає даній функції.

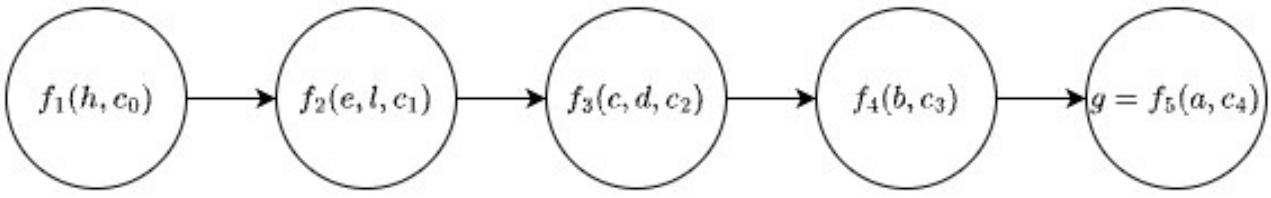


Рисунок 3.31 — суперпозиція даних у графі

Зображений граф на рисунку 3.31 представляє поетапне обчислення функції із застосуванням суперпозиції та накопичувального регістру. В кінці

граф присвоює змінній g в пам'яті значення, що було отримано в результаті усіх обчислень. Таким чином реалізується суперпозиція для даних.

Тепер розглянемо відмінності для суперпозиції над предикатами. Результат обчислення предикатів є завжди булевим значенням; а як було сказано вище, сама діаграма переходів (тобто функціональний граф, у даному контексті) для переходів з одного стану в інший використовує булеві значення. Тобто самі булеві функції $f_1^n, \dots, f_m^n$ можуть управляти графом. Розглянемо на прикладі застосування такого принципу. Нехай, існує наступна функція $f_b: N^{4^2} \to \mathbb{B}$:

$$f_b(< a_1, \dots, a_4 >, < b_1, \dots, b_4 >) = a_1 > b_1 \wedge a_2 < b_2 \wedge a_3 \geq b_3 \wedge a_4 \leq b_4$$

Реалізуємо її, використовуючи суперпозицію. Так, $f_1^2 = a > b$, $f_2^2 = a < b$, $f_3^2 = a \geq b$, $f_4^2 = a \leq b$. А функція $f^m, m = 4$ виражається як m-арна логічна функція "І", $f^m = \bigwedge_{i=1}^{m} f_i^2(a_i, b_i)$. При цьому, принцип роботи m-арного логічнаого "І" є таким, що всі m булевих значень повинні бути істинними, щоб на виході теж була істинна. Тобто можна послідовно перевіряти значення, і якщо якесь виявиться хибним, повернути хибу.

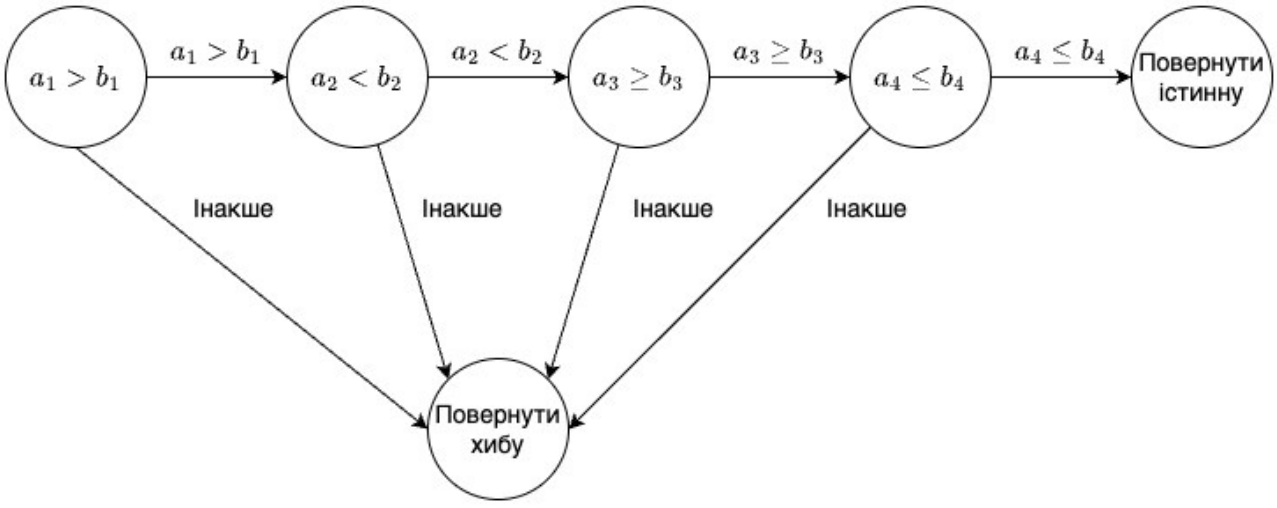


Рисунок 3.32 — суперпозиція логічна І у графі

Як видно, починаючи з початкової функції, діаграма переходів поступово "наближається" до вузлу "Повернути істинну" кожного разу, як

виконується умова порівняння. Проте варто хоч одному предикату бути хибним — повернеться хиба. Даний механізм повністю відповідає опису багатомісної операції логічного “І”.

Для логічного “АБО”, граф має подібну структуру, лише з тим урахуванням, що для багатомісного логічного “АБО” необхіден хоча б один істинний аргумент, щоб повернути істинну. Так, видозмінемо функцію $f_b: N^{4^2} \to \mathbb{B}$ у нову $f_b': N^{4^2} \to \mathbb{B}$.

$$f_b'(< a_1, \dots, a_4 >, < b_1, \dots, b_4 >) = a_1 > b_1 \vee a_2 < b_2 \vee a_3 \geq b_3 \vee a_4 \leq b_4$$

Логічно, що для f_b' змінилась лише функція $f^{m\prime} = \bigvee_{i=1}^{m} f_i^2(a_i, b_i)$, а функціональний граф виглядає наступним чином.

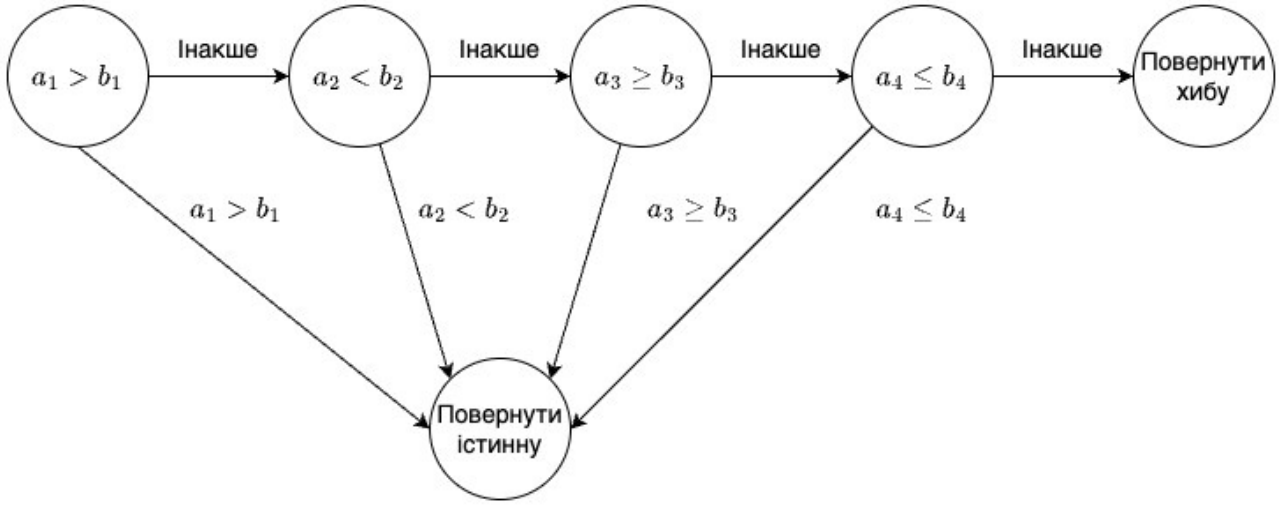


Рисунок 3.33 — суперпозиція логічнаАБО у графі

Як видно з рисунків 3.32 та 3.33, структура графа для обох багатомісних логічних функцій ідентична. Тільки у випадку з логічним “АБО” функція навпаки наближається до хиби з кожним хибним предикатом. Операції “І” та “АБО” є дуже важливими, адже разом з операцією заперечення вони утворюють набір примітивних функцій над булевою множиною {¬,∨,∧}. А одже для предикатів операція суперпозиції є більш потужнішою і менше залежить від апаратної реалізації.

І хоча для повноцінної реалізації у графах суперпозиції найкраще підійде пам’ять типу стеку або черги; бо така пам’ять є багатовмісна, а також дозволяє абстрагуватися від адресації при її логічному використанні. Проте, наявного апаратного потенціалу повністю вистачає для реалізації базових операцій реляційного процесору, а більше від нього і не треба.

Галуження. Операція галуження для фукнціональних графів є досить природньою, та на відміну від суперпозиції, повністю підтримується не лише для базисних функцій. Та навіть функціональні графи підтримують мультигалуження $\diamond^{2m}(p_1^n, \dots, p_m^n, f_1^n, \dots, f_m^n)$. Підтримка мультригалуження грунтується наступною структурою.

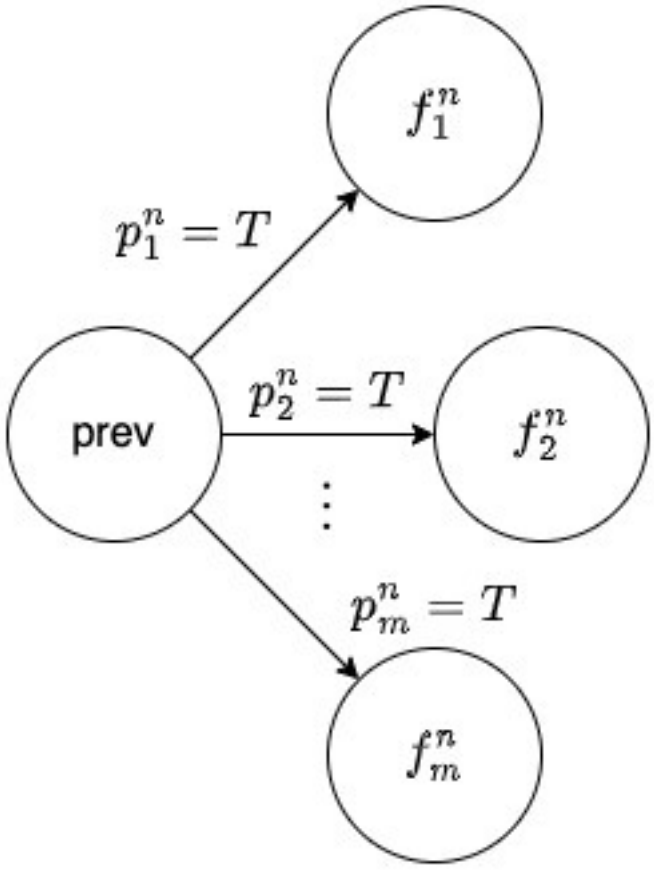


Рисунок 3.34 — мультигалуження в графі

Граф на рисунку 3.34 доводить можливість галуження в графах та демонструє принцип його реалізації. Проте, варто повернутись до

вищезазначених обмежень такої конструкції. Так, для кожного можливого входу предикатів, лише один предикат має бути істинним, коли як всі інші мають бути хибними. Формально кажучи, $\forall(a_n) \in N^*, \exists l \in [1,m]$ для якої $p_l^n(a_n) = T, p_i^n(a_n) = F, i \in [1,m] \backslash \{l\}$. Для запобігання від можливих невизначеностей введемо обмеження, що хоча б один з предикатів є інверсією іншого $\exists i,j \in [1,m], i \neq j$ для яких $p_i^n(a_n) = \neg p_j^n(a_n)$. Легко перевірити, що для випадку бінарного гілкування завжди буде один предикат, що є рівним істинні на всіх можливих значеннях. $\diamond^2 (p_1^n, p_2^n, f_1^n, f_2^n), p_2^n = \neg p_1^n \Longrightarrow \diamond^2 (p_1^n, p_2^n, f_1^n, f_2^n) \equiv \diamond (p^n, f_1^n, f_2^n)$. А отже підтримка гілкування у функціональних графах можлива.

Циклування. В графах циклування реалізується за допомогою замкнення чи петлі. Петлею реалізуються прості циклування, що складають з однієї примітивної функції (в сенсі що вона реалізована апаратно). Замкненим графом же представляється складна функція циклування, що вже складається з декількох примітивних функцій. Її кінцевий вузол має перехід на перший вузол комплексної функції, що змушує цикл повторюватись до тих пір, доки предикат не буде хибним.

Розберемо для прикладу функцію порівняння на рівність двох строк реляцій. Дана функція покроково порівнює на рівність компоненти двох строк, і якщо хоча б одна пара відповідних елементів не рівна — функція повертає хибу. Реалізуємо дану функцію за допомогою циклування $*(p^n, f_1^n, \dots, f_m^n)$. Для початку визначимо функції $f_1^n, \dots, f_m^n$, дані функції відповідають за адресацію, порівняння та повернення хибного результату, коли як предикат p^n включає в себе порівняння. Наведемо запис функції порівняння, реалізованої апаратно, дана функція приймає на вхід 2 підвектори та предикат, по якому вони порівнюються $cmp: N^* \times N^* \times P \to \mathbb{B}$. Так, запишемо функцію порівняння строк за допомогою циклування.

$$p = cmp(a[i_a = 0, j_a], b[i_b = 0, j_b], =)$$
$$(j_a \neq m_a \vee j_b \neq m_b) * ($$
$$(\neg p) \diamond (\textit{Повернути хибу}),$$
$$p = cmp(a[i_A, + + j_A], b[i_B, + + j_B], =),$$
$$)$$
$$\textit{Повернути істинну}$$

У графах дана операція виглядає наступним чином.

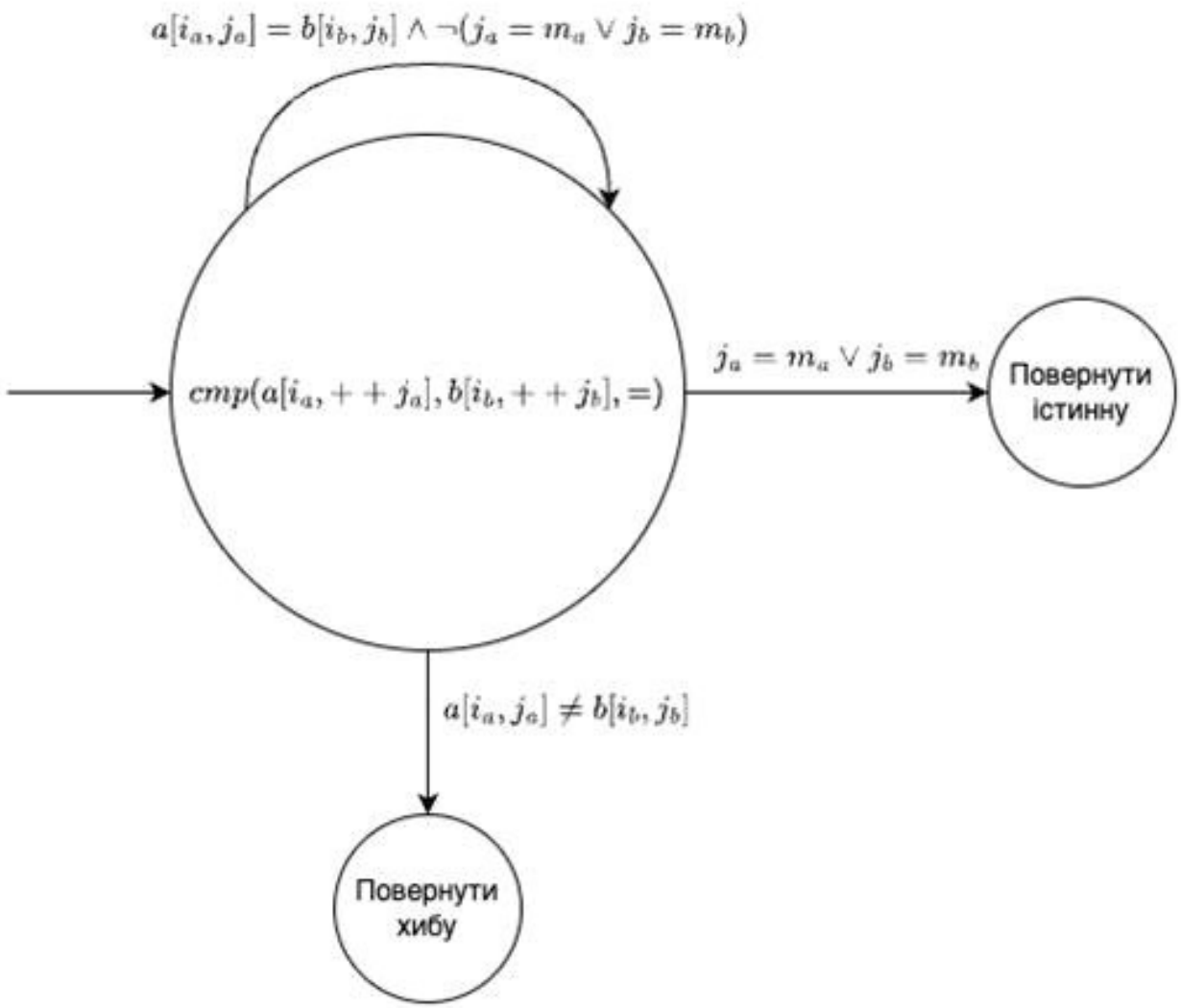


Рисунок 3.35 — граф порівняння двох строк реляції

У найбільшому вузлі (функція циклування) графа на рисунку 3.35 індекси стовпчиківзбільшуються на одиницю, а самі нові підвектори порівнюються між собою функцією cmp. Якщо результат порівняння є істинним, функція продовжує циклування; а в інакшому випадку поветрає хибу. Коли функція доходить до кінця строки $j_a = m_a \vee j_b = m_b$, функція

повертає істинну. Як видно з умов переходу, вони обрані таким чином, щоб для всіх можливих варіантів був лише один можливий перехід.

Розглянемо простіший приклад з операцією множення з накопиченням стовпчиків двох реляцій. У введених термах дана операція виглядає наступним чином $(i_a \neq n_a \vee i_b \neq n_b) *_i (c_i = a[++i_a, j_a] *_k b[++i_b, j_b] + c_{i-1}), g = c_i$, де c_i — поточний стан регістру накопичення, c_{i-1} — попередній стан регістру накопичення, g — змінна, куди записується результат. Проілюструємо тепер рівняння множення з накопиченням у графі.

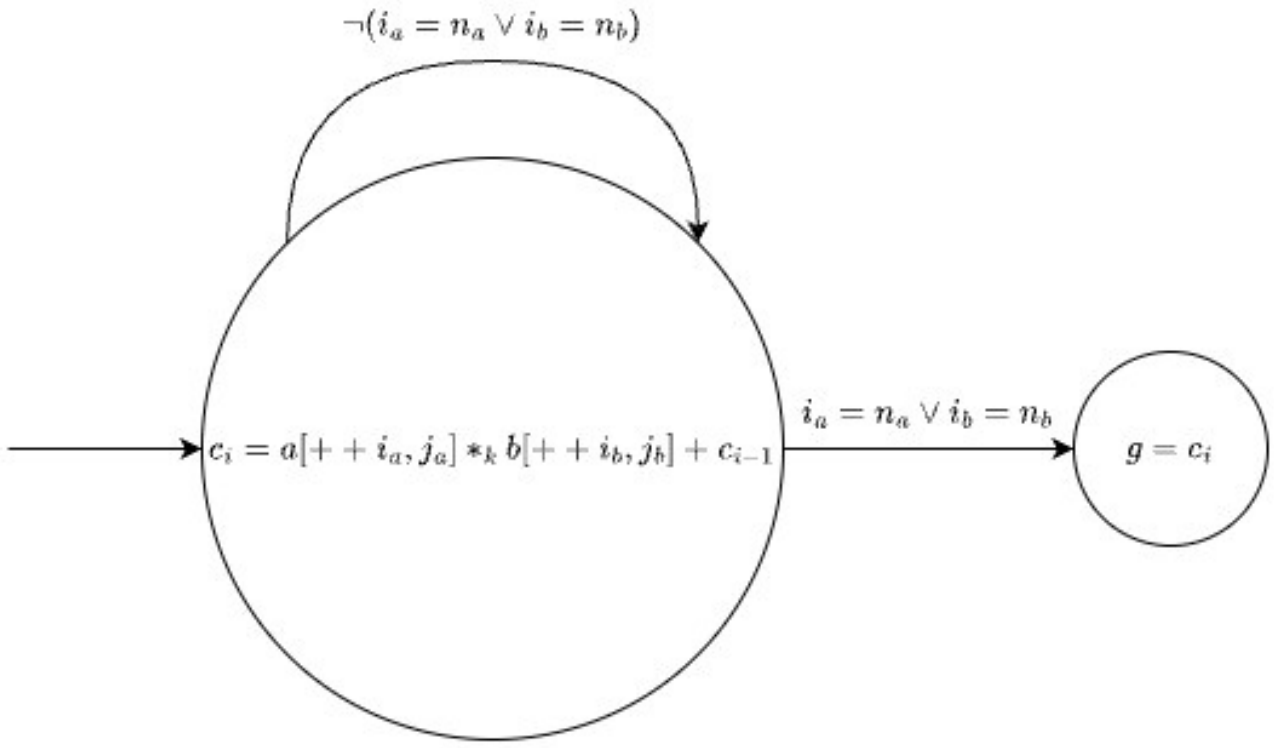


Рисунок 3.36 — граф множення з накопиченням стовпчиків реляції

Вузол циклу (найбільший) на рисунку 3.36 виконує наступні дії: інкрементує індекси стопвчиків, перемножає відповідні значення i-го рядка та j-го стовпчика елементів реляцій a, b, та додає до регістру накопичення. Після проходження всіх елементів матриці граф записує результат до g. Таким чином була доказана та продемонстрована підтримка циклування у функціональних графах.

Висновок. Отриманий результат доказує, що функціональні графи підтримують сигнатурні операції ППА, хоч і не в повному обсязі. А також було продемонстровано імплементацію цих сигнатурних операцій в графах на прикладах. За результатом, обмеженої повноти підтримки ППА функціональними графами вистачає щоб реалізувати на них базисні операцій реляційного обчислювача, тобто його атомарні інструкції.

3.8 Блок керування операціями

Очевидно що даному блоку, який є функціональним графом, необхідні сигнали, що дозволяють відстежати стан системи та задавати їй команди. Зараз ми розглянемо входи та виходи між блоком керування операціями — блоком обчислення адрес та блоком керування операціями — блоком обчислення даних. Також розглянемо деякі входи та виходи з пам’яттю та керуючим ядром, і спеціальні регістри для даних з інструкцій.

Зв’язок з блоком обчислення адрес. Лінії між цим блоком дозволяють керувати лічильниками та зчитувати їхні стани. Під станами для керуючого блоку розуміється чи не переповнений лічильник. Під керуванням лічильників мається на увазі дозвіл на лічення, перезапис вмісту лічильника, та джерела перезапису, його скидання. Як вже було сказано, блок обчислення адрес обчислює 3 адреси. Для обчислення кожної адреси використовується пара лічильників з лічильника атрирбутів (стовпчиків) та лічильника строк (рядків), при цьому лічильник атрибутів є здвоєним, тобто місить, всередині 2 лічильника зі своїми сигналами переповнення, і додатково лічильник кількості атрибутів. Таким чином блок керування операціями на вхід приймає 10 сигналів переповнення.

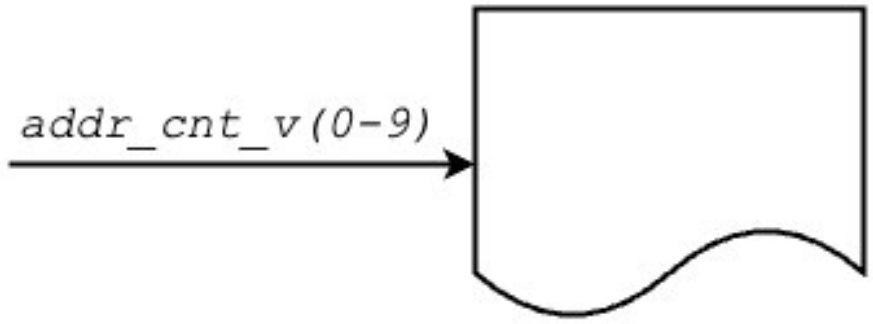


Рисунок 3.37 — вхідна шина переповнень лічильників

Шина, яка збирає усі лініїї переповнення, називається *addr_cnt_v*. Відповідність бітів шини до лічильника наведена в таблиці 3.12.

Таблиця 3.12 — відповідність бітів шин *addr_cnt_v* та *addr_cnt_en addr_cnt_reset* до лічильників

Номер біта	Належність до лічильника
0	Зсув вектору реляції A
1	Атрибутів реляції A
2	Строк реляції A
3	Зсув вектору реляції B
4	Атрибутів реляції B
5	Строк реляції B
6	Зсув вектору реляції Dest
7	Атрибутів реляції Dest
8	Строк реляції Dest
9	Кількість атрибутів

Таким чином, спочатку групуються сигнали переповнення лічильників, які відносяться до певної реляції, за порядком: 1 — лічильник зсуву вектора, 2 — лічильник атрибутів, 3 — лічильник строк. Спочатку в шині йде група

для реляції A, потім для реляції B, і реляції Dest, і в кінці біт переповнення лічильника кількості атрибутів. Завдяки даній таблиці, абстрагуємось від позиційного номеру біту.

Виходи ж до блоку обчислення адрес є більш обширними. До нього направлена 10-бітна шина дозволу лічення, 4-бітна шина дозволу перезапису вмісту лічильників, 3-бітна шина вибору джерела перезапису та скидання блоку.

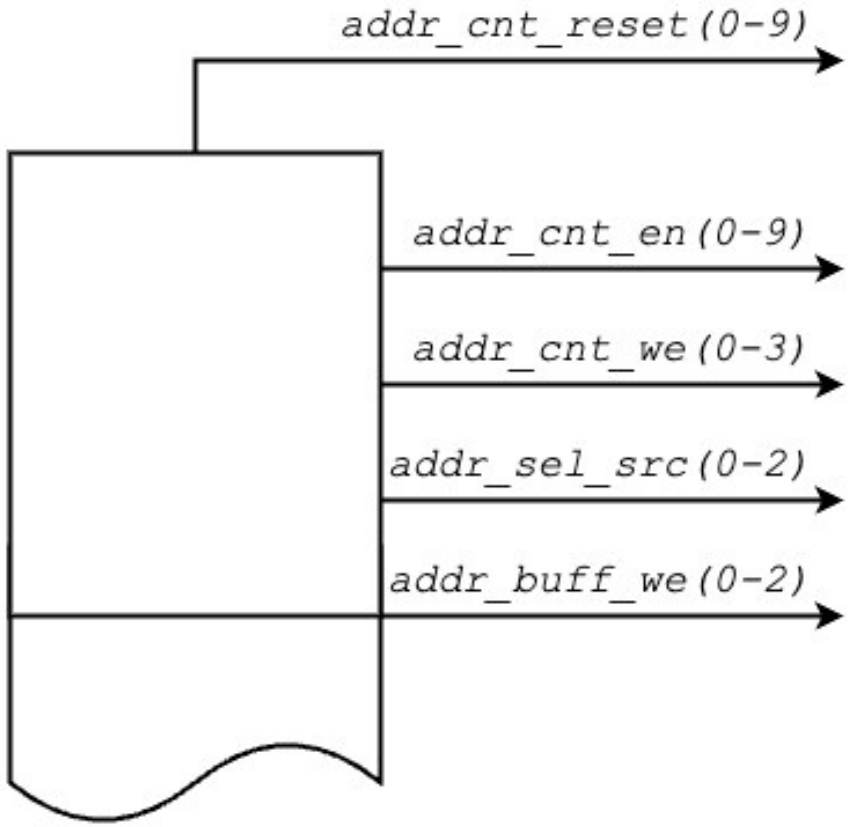


Рисунок 3.38 – Вихідні шини до блоку обчислення адрес

Так, шина дозволу лічення називається *addr_cnt_en*, її біти визначені в таблиці 3.12. Шина дозволу перезапису вмісту лічильників називається *addr_cnt_we*, а шина вибору джерела перезапису *addr_sel_src*. Також, за допомогою виходу *addr_buff_we* можна задавати дозвіл на запис буферів атрибутів. А *addr_cnt_reset* зкидає відповідні до таблиці 3.12 лічильники.

Таблиця 3.13 — відповідність бітів шини *addr_cnt_we* до лічильників

Номер біта	Належність до лічильника
0	Атрибутів реляції A
1	Атрибутів реляції B
2	Атрибутів реляції Dest
3	Лічильника кількості атрибутів

Таблиця 3.14 — відповідність бітів шин *addr_sel_src* та *addr_buff_we* до лічильників

Номер біта	Належність до лічильника/буфера
0	Атрибутів реляції A
1	Атрибутів реляції B
2	Атрибутів реляції Dest

Повернемось до позначень операцій та його відношення до керуючих бітів. Як вже зазнчалось, адреса елементу реляції визначається виразом $R[i, j + j_S]$, де i — строка (рядок) реляції, j — підвектор атрибутів (стовпчиків) реляції, j_S — зсув вектору атрибутів. Записом $++i$ позначалося збільшення значення індексу стовпчиків на одиницю. Співставимо це з керуючими бітами. Таким чином запис $++i_A$означає, що прапорець дозволу лічення лічильника строк реляції A виставлено в активний сигнал. Запис $j_A + j_{SA} = attr_{init}$ означає, що лічильники атрибутів та зсуву вектору перезаписується з регістру, у якому зберігається значення початкового атрибуту; а записом $buffer_{attrA}$ позначимо регістр тимчасового зберігання індексу атрибуту реляції A (цей регістр також зберігає й значення зсуву вектора). А перезапис тимчасового регістру позначимо як

$buffer_{attrA} = j_A + j_{SA}$. Таким чином, запис $R_A[i_A, buffer_{attrA} = + + j_A + j_S]$позначає наступне: біт дозволу лічення лічильника строк реляції А виставлено в “0”, біт дозволу лічення лічильника атрибутів реляції А виставлено в “1”, біт дозволу лічення лічильника зсуву вектору реляції А виставлено в “0”, біт перезапису лічильника атрибутів реляції А виставлено в “0”, біт перезапису тимчасового буферу атрибутів реляції А виставлено в “1”. Подібний вид запису розповсюджується на всі виходи.

Зв’язок з пам’яттю реляцій. До пам’яті реляцій від блоку керування операцій надходить kліній дозволу запису даних. Дані лінії дозволяють виборочно перезаписувати дані в підвекторах реляцій, а також взагалі блокувати запис під час перевірки реляцій на рівність або нерівність.

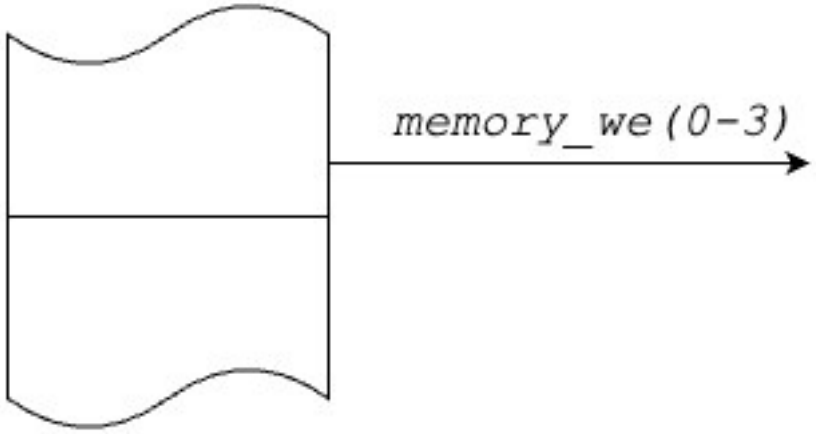


Рисунок 3.39 — вихід дозволу запису даних до пам’яті блоку керування операцій

Шина дозволу запису даних називається *memory_we*. Значення такої шини у функціональних графах задається як параметр та записується верхнім індексом. Наприклад, $R_D^{\overrightarrow{1,k}}[i_D, j_D + j_{SD}] = R_A[i_A, j_A + j_{SA}] + R_B[i_B, j_B + j_{SB}]$означає запис усіх елементів вектору до пам’яті, а наступний запис $R_D^1[i_D, j_D + j_{SD}] = R_A[i_A, j_A + j_{SA}] + R_B[i_B, j_B + j_{SB}]$ записує до пам’яті лише перший елемент вихідного вектора.

Зв’язок з керуючим ядром. Оскільки керуюче ядро передає інструкції, їхній взаємозв’язок є дуже важливим. Очевидно, що керуюче ядро передає

виконавчому операцію, яку останній повинен виконати; а також усі її параметри, як от арифметична операція чи предикат порівняння. Все це оброблюється саме блоком керування операціями. Також керуюче ядро надсилає сигнал дозволу роботи саме даному блоку.

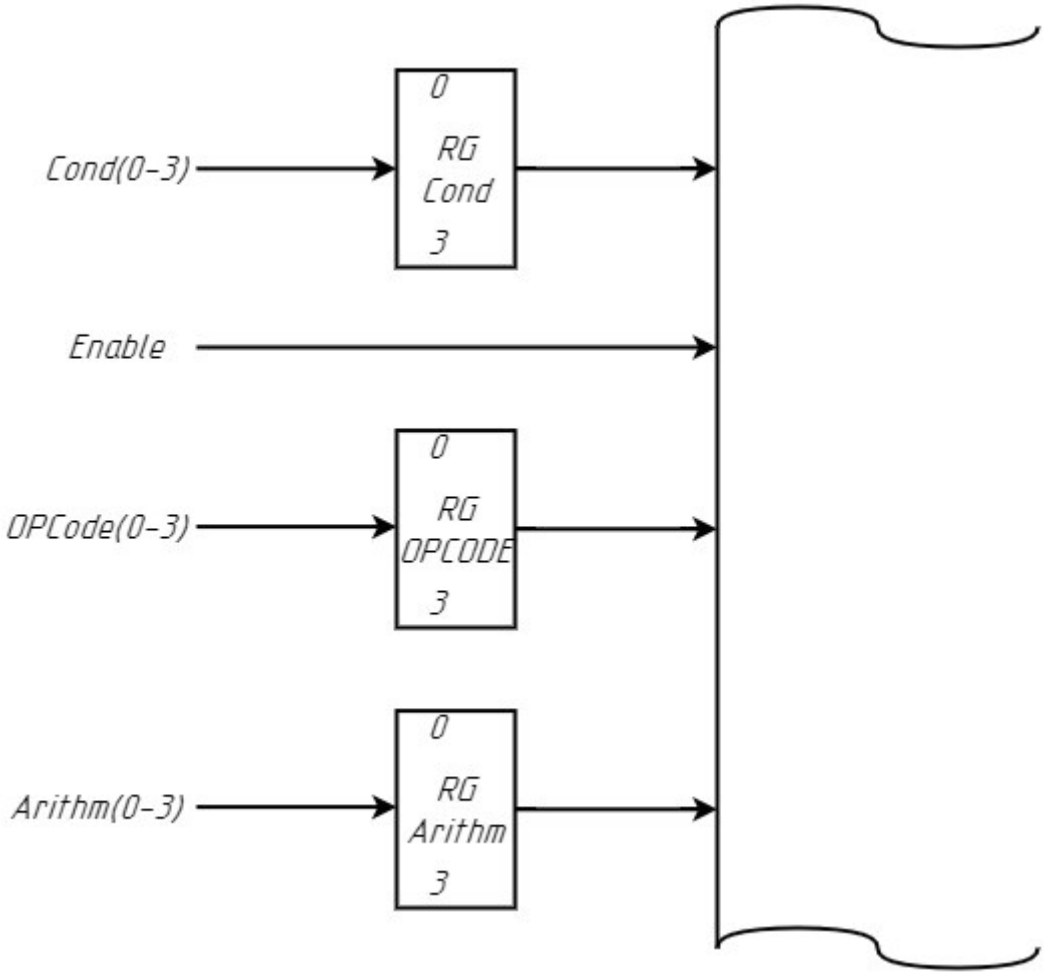


Рисунок 3.40 — Вхідні сигнали від керуючого ядра до блоку керування операціями

Як видно, операція *OPCode*, код арифметичної дії *Arithm* та код предикату *Cond* зберігаються в спеціально відведених регістрах, попередньо розширюючись для і вже ці регістри подаються на вхід блоку керування операціями. Їхнє використання дозволить застосовувати одну й ту саму логіку, не залежно від самого контексту операції.

На виході блок надсилає керуючому ядру 2 біти статусу.

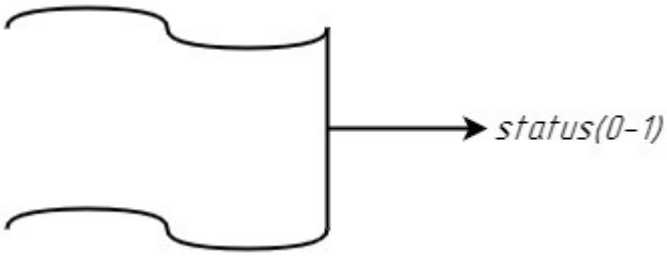


Рисунок 3.41 — шина статусу до керуючого ядра

Нульовий біт шини *status* (рисунок 3.41) відповідає за зайнятість виконавчого ядра. А перший біт відповідає за статус перевірки предикату.

Зв'язок з блоком обчислення операцій. З даним блоком керуючий автомат пов'язують вихід з командою та дозволу для АЛУ, та вхід на перевірку порівняння. При чому, через використання векторизації, таких шин налічується для кожного елементу підвектора (тобто значення з одної комірки). Таким чином на вхід блок приймає статус обчислення, а на виході дає команду та дозвіл на зберігання результату (фактично на виконання).

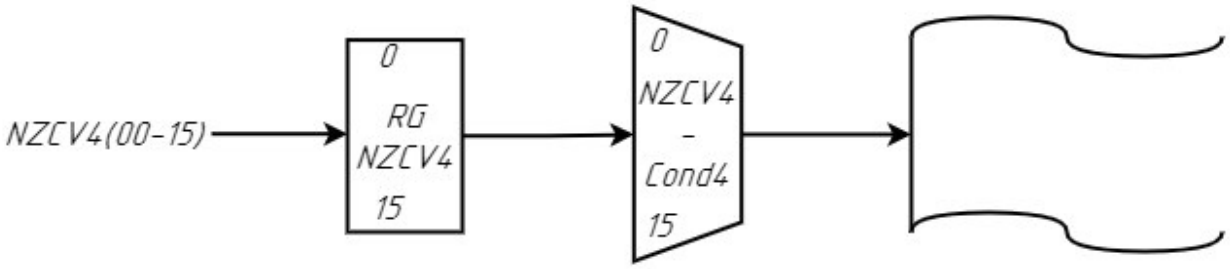


Рисунок 3.42 – вхідні дані від арифметичного блоку

Як видно з рисунку 3.42, статус обчислення представлений вже зазначеними прапорцями NZCV, при чому на вході представлені дані прапорці від всіх АЛУ, тобто *NZCV4* має займати 16 бітів. Ці дані зберігаються в регістр, оскільки в самих АЛУ вони не зберігаються. На вхід самому блоку подається декодоване значення з NZCV у код умов з інструкцій (таблиця 3.3) Cond дешифратором *NZCV4-Cond4*. Таким чином, блок обчислення адрес порівнює дані від АЛУ та інструкцій напряму.

На виході блок керування операціями передає АЛУ сигнали про операцію, яку необхідно виконати, а також дозвіл на запис до

накопичувального регістру; а оскільки з регістру зчитуються дані до пам’яті, то фактично дозвіл на запис до регістру є дозволом на виконання операцій.

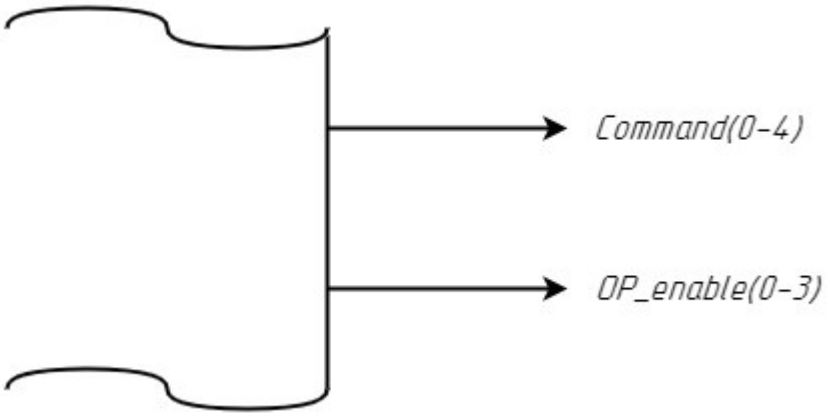


Рисунок 3.43 – вихідні дані до арифметичного блоку

По шині *Command* з рисунку 3.43 блок керування операціями передає команди 4-м АЛУ. З таблиці 3.8 видно, що команда для АЛУ займає 5 бітів, звідти отримується розмір шини *Command*. Усі 4 АЛУ (рисунок 3.14) мають спільну шину команд. Тобто шиною *Command* задається одна й та сама операція усім АЛУ. А шина *OP_enable* відповідає за дозвіл запису до регістру накопичення, кожен біт під’єднано до відповідного по нумерації АЛУ (рисунок 3.14).

3.9 Функції-підграфи

Принцип розділення та володарювання є давно відомим, таким же чином функціональний граф розділяється на підграф-функції. Введемо поняття підграф-функції. Це такий підграф, що виділяється за контекстом виконання дій. При цьому, виділемо саме такі підграфи-функції, які можуть реалізувати базові операції. Даний підграф має єдиний “вхідний” вузол, та єдиний “вихідний”. Це обумовлено тим, що базові операції вважаються атомарними, а після їх завершення граф повертається в стан очікування. При цьому “вихідний” вузол розміщується за підграфом, проте в самому підграфі на цей вузол посилаються всі “вихідні ребра”, тобто ті, що посилаються на вихідний вузол. А сам “вхідний” вузол розміщується всередині підграфа.

Побудуємо керуючий функціональний граф. Почнемо з початкового вузла, тобто стан, в який кінцевий автомат переходить при підключенні живлення. З цього стану кінцевий автомат очікуватиме на послідуючі операції та дозвіл виконання. Назвемо даний стан “Stand by” (англ. Очікування), у ньому всі операції заборонені (дозволяючі лінії встановлені в “0”). З цього стану автомат переходить до виконання підграфу-функції. Після виконання вся система повинна перейти назад у стан очікування. Назвемо даний стан “Reset core”, тобто при ньому все переводиться до початкового стану.

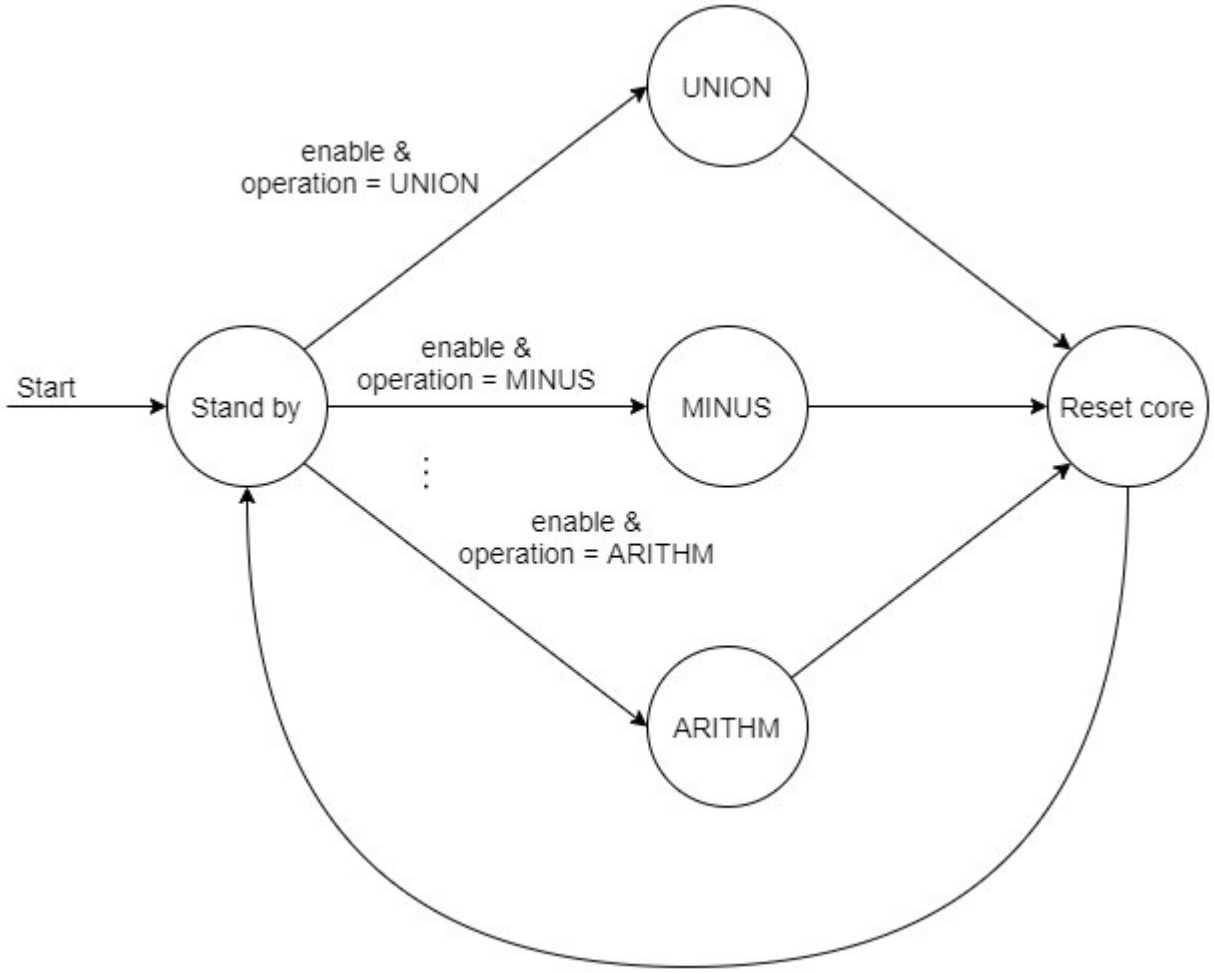


Рисунок 3.44 – структура функціонального графу блоку керування операціями

Граф на рисунку 3.44 представляє найвищий рівень абстракції кінцевого автомату блоку керування операціями. Так, він представляє основну логіку – на початку роботи кінцевий автомат знаходиться у стані

очікування на операцію та прапорця дозволу; коли надходить сигнал дозволу — автомат обирає одну з функцій-підграфів, в залежності від поля *OPCode*; після виконання функції автомат переходить у стан скидання виконавчого ядра, після чого знову переходить до стану очікування. У графі представлено усі операції з таблиці 3.1, а також для усіх 3-х типів арифметичних операцій окремо виділено операції з накопиченням. На рисунку 3.44 є лише частина, для графічної демонстрації.

Розглянемо як працюють операції з накопиченням. Дані арифметичні операції працюють зі стовпчиками реляцій. Наприклад, операція множення з накопиченням є аналогом зваженої суми усіх елементів стовпчика реляції. Визначається дана операція таким чином:

$$d_i = \sum_{j=1}^{m} a_i^j \cdot b_i^j,$$

де a_i^j — j-та строка i-го стовпчика (атрибуту) реляції a, аналогічно для реляції b;m — кількість строк реляції; d_i — i-й стовпчик однострочної реляції. Реляція d_i не обов'язково має бути однострочною, проте результат буде записано саме до першої строки. Аналогічним чином працюють й інші операції накопичення.

Спрощення позначення. У минулому підрозділі були визначені зв'язки між математичними позначеннями та вихідними сигналами. Тепер ще більше спрощемо позначення, щоб було зручніше записувати дії до вузолу графа. Перше спрощення направлене на позбавлення від індексів. Так запис $R_A[i_A, ++j_A + j_S]$ стає A[i, ++j + js]. Так, належність лічильника до реляції визначається не індексом, а позначенням реляції, що стоїть перед квадратними дужками. Це також є справедливим і для буферу A[i, buffer = j + js]. Друге спрощення полягає в тому, що запис логічних операторів замінюється з їхнього математичного символу на позначення, що можуть бути відтворені у стандартному 8-бітному кодуванні ASCII. Так, знак заперечення ¬ стає ~. Символ логічного І ∧ замінюється на простий

англійський напис з маленької букви and. А символ логічного АБО аналогічно замінюється на англійський напис з маленької букви or. Також вводяться спеціальні макро заміни, або ж макроси. Дані макрозаміни являють собою прості словосполучення, що призвані замінити на графі складні логічні операції. Приймемо, що макроси записуються великими літерами. Четверте спрощення полягає у тому, що якщо у ребра реляції не вказана умова переходу, то перехід є безумовним, тобто при будь-яких значеннях. І врахуємо те, що на самому початку функції-підграфу усі лічильники та регістри скинуті в нуль.

Допоміжні функції-підграфи. Деякі базові функції інструкцій можуть мати складну логіку, яка є складно сприйманою людиною. Тому, саме для таких функцій введемо деякі допоміжні функції, що допоможуть зручніше побудувати функції інструкцій та легше сприймати їхню логіку. Розглянемо функцію порівняння двох атрибутів строк двух реляцій.

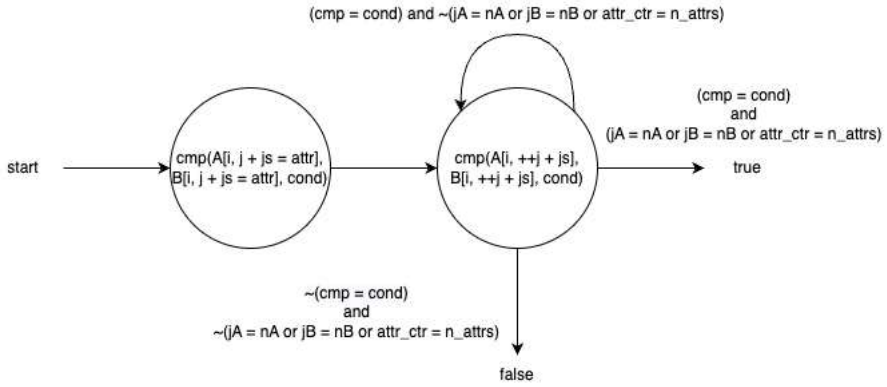


Рисунок 3.45 — функція-підграф порівняння певних атрибутів двух реляцій

Як бачимо з рисунка 3.45, дана функцій-підграф має всього 2 вузли (стани кінцевого автомату). У першому порівнюються початкові атрибути i-х строк реляцій (під i-ю розуміється лічильник строк, який у кожної реляції свій власний, проте позначається в межах квадратних дужок однаково). У наступному вузлі порівнюються між собою вже наступні атрибути. Тобто порівнюються елементи за інкрементованими адресами. Спочатку блок обчислення адрес інкрементує адреси, потім вилучається значення з комірки

за розрахованою адресою, і потім це значення порівнюється. Існує 2 різних варіанта для виходу з функції:

- true — якщо всі значення строк дійсно рівні;
- false — якщо знайшлися двоє нерівних відповідні підвектори.

З назви даної функції-підграфу очевидно, що дана функція є предикатом та повинна повернути булеве значення, що якрах і відповідає дійсності. Як вже було зазначено раніше, щоб предикат порівняння повернув істинну необхідно щоб усі відповідні елементи відповідали умові порівняння.

Тепер введемо параметричніть допоміжних функцій підграфів. Вона необхідна, щоб можна параметрами можна було замінити елементи, для яких логіка буде однаковою, чи несуттєво відрізнятись. Так, наприклад, у вищезазначеному прикладі в якості умови порівняння береться зовнішня, що зберігається в регістрі *Cond*. Тепер, щоб її використати необхідно її прописати в параметрі функції. Сама ж функція-підграф позначається кружечком, всередині якого написаної ім’я функції. Після імені у квадратних дужках вказуються параметри. Для параметрів різного типу відводяться свої дужки. Так, щоб вказати що ми хочемо порівнювати саме з умовою в Cond, необхідно це прописати після імені функції так: [Cond]. Окрім цього, для спрощення обходу значень реляцій, можна вказати в параметрах, що функція буде проходитись по наступним строкам записом [iA + 1, iB + 1]. Даний запис означає, що у першому стані лічильник строк реляцій A та B інкрементуються (лише один раз).

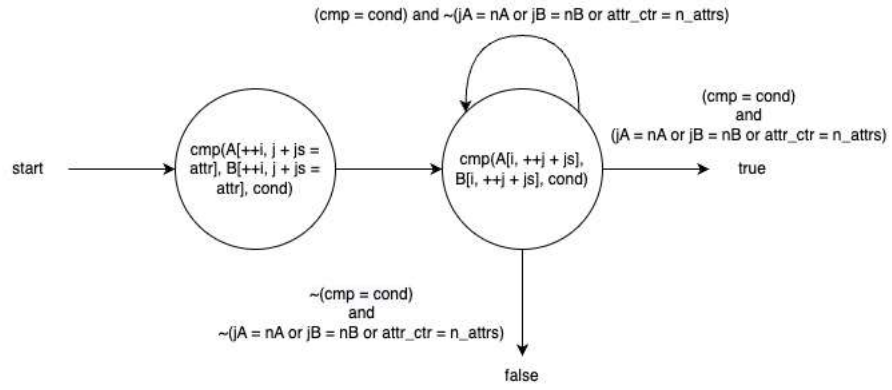


Рисунок 3.46 — функція-підграф порівняння певних атрибутів наступної строки двох реляцій

Дана функція-підграф майже не відрізняється від аналогічної на рисунку 3.45, окрім як інкрементації лічильників строк реляцій A та B.

Введемо ще декілька допоміжних функцій-підграфів.

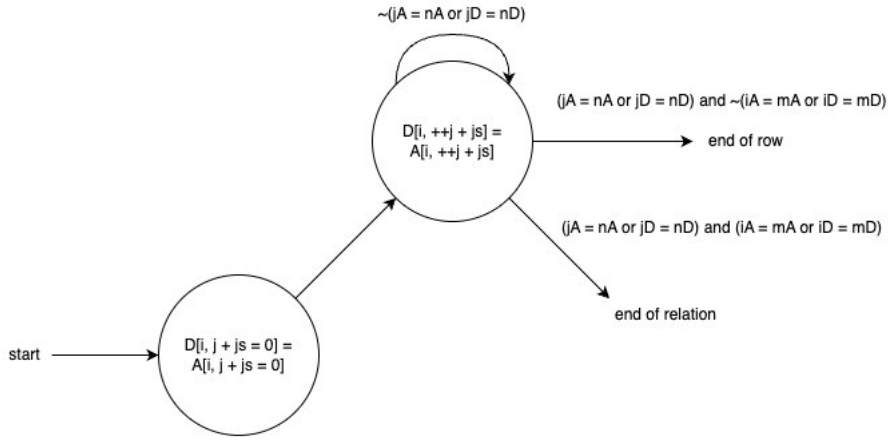


Рисунок 3.47 — функція-підграф копіювання строки реляції

Для легшої побудови більш абстрактної логіки, дана функція-підграф з рисунку 3.47 має 2 виходи: end of row призначений для випадку, коли лічильники не долічили до кінця реляцій; а end of relations навпаки, коли функція-підграф долічила до кінця хоча б одної з реляцій. Для вказання ресурсу для копіювання (реляція A чи B), необхідно задати параметр [D = A] чи [D = B] відповідно. А також розглянемо дану функцію з параметром наступної строки.

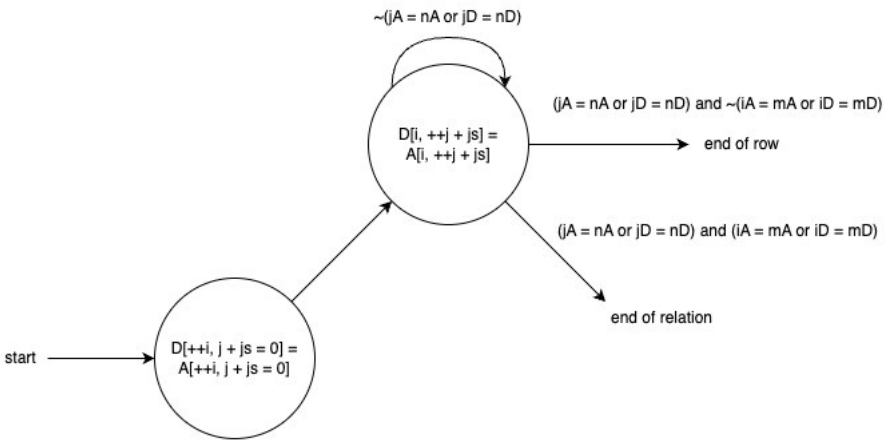


Рисунок 3.48 — функція-підграф копіювання наступних строк реляцій

Як видно, ситуація з графами з рисунків 3.48 та 3.47 аналогічна до 3.45 та 3.46. Проте потрібно ввести **зауваження** щодо використання одночасно двох однакових функцій-підграфів з різними параметрами. Так, їхні спільні вузли та ребра можуть бути як спільними при переході на менш низький рівень абстракції, так і відрізнятись. Це залежить саме від того, як апаратно реалізовані кінцеві автомати.

Наступна функція є похідною від функції порівняння атрибутів, вона порівнює 2 строки на рівність.

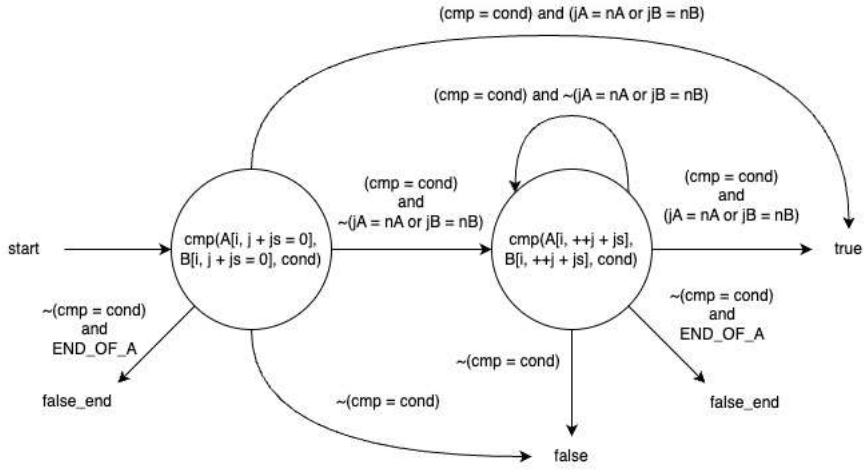


Рисунок 3.49 — функція-підграф порівняння двох строк

Як бачимо, дана функція-підграф є аналогічною до порівняння атрибутів з рисунку 3.45, за вийнятком що тепер для переходу не використовується предикат лічильника атрибутів та лічильники на початку скидуються в нуль.

Наступна допоміжна функція-підграф є вкрай цінною для операцій над множинами, а саме перевірка чи на наявність строки в реляції. Так, доступні 2 різновиди даної функції-підграфа:

- row A is in B;
- row B is in A.

Логіка їхньої роботи нічим не відрізняється, тому параметр даної функціївказуєтьсявідразу вімені. Першою є реляція, стовпчик якої перевіряється на його наявність у B.

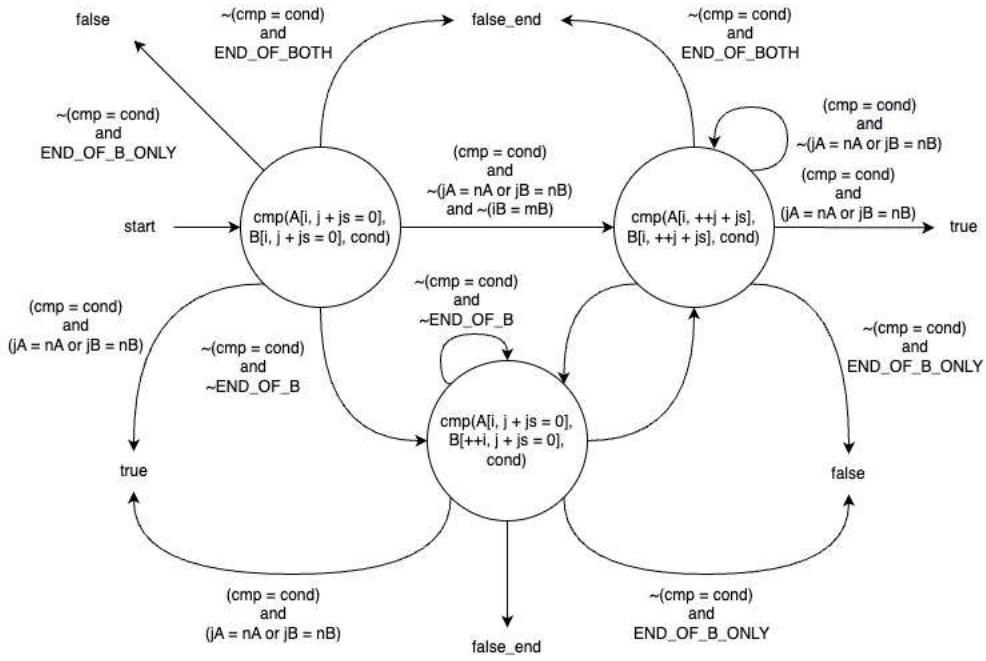


Рисунок 3.50 — функція-підграф наявності строки в реляції

З рисунку 3.50 видно, що дана функція-підграф у циклі обходить усю реляцію B у пошуку відповідної строки A. При цьому, якщо перші атрибут и однакових стовпчиків A та B не рівні, то функція відразу переходить до наступного. Дана функція вона має 3 виходи:

- true — якщо знайшлась відповідна строка;
- false — якщо даної строки не знайшлося;
- false end — якщо даної строки не знайшлося, та вона є останньою в A.

Останній вихід необхідний, щоб можна було відстежувати завершення реляції A у більш абстрактних графах. При цьому, кожен вузол з рисунку 3.50 може перейти до будь-якого виходу. Це забезпечує працездатність як реляцій усіх розмірів (одноелементні, одностовпчикові, та багаторозмірні.
Також є ще одна реалізація функції-підграфа наявності строки в реляції, зі спрощенною в порівнянні з рисунком 3.50 логікою.

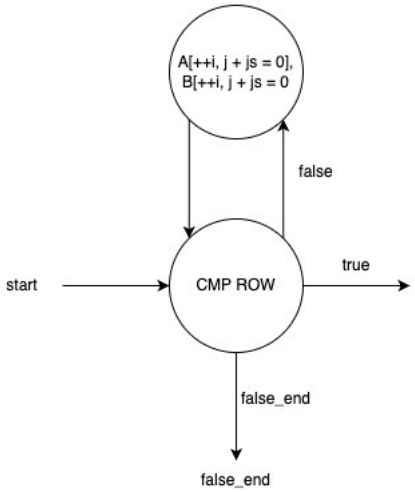


Рисунок 3.51 — спрощена функція-підграф наявності строки в реляції

Даний граф не має стільки зв’язків, як аналогічний на рисунку 3.50. Проте його недолік в тому, що при переході на новий стовпчик у відповідному вузлі не порівнюються значення; коли як у попередній порівняння відбувається завжди. Різниця в продуктивності між функцією-підграфом з рисунку 3.50 та 3.51 напряму залежить від кількості атрибутів реляції за формулою $\frac{n}{n+1}$, де n позначає розмір строки. Значення в чисельнику та знаменнику показує скільки тактів витрачається на порівняння однієї строки для комплексної схеми та спрощенної відповідно. Так, для реляцій з одним стовпчиком продуктивність впаде вдвічі. Проте надалі буде значно менше впливати.

Останньою допоміжною функцією-підграфом є функція копіювання однієї реляції в іншу. Вона потрібна лише для операції об’єднання реляцій.

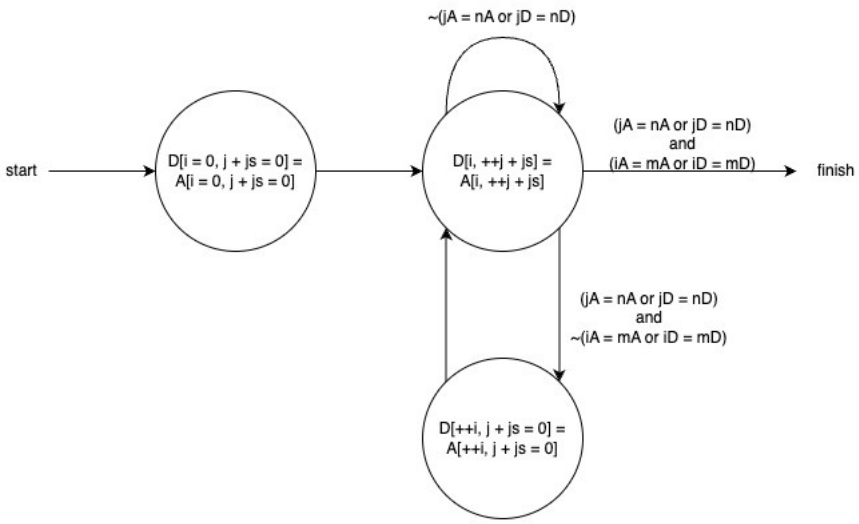


Рисунок 3.52 — функція-підграф копіювання реляції

Як видно з рисунку 3.52, для зупинки копіювання достаньо лише, щоб лічильники долічили до кінця хоча б однієї з реляцій. Саме це і необхідно в операції об’єднання. А дана допоміжна функця підграф також є параметричною. Тобто можна вказати джерело копіювання [D=A], або [D=B].

У даній роботі наводиться лише 9 з 18 функцій-підграфів, що реалізують базові функції реляційного процесора. Причина тому брак місця, а для опису усіх функцій-підграфів потрібно багато простору.

Minus. Операція віднімання множин $A \backslash B$ у своїй суті дуже проста, до нової множини входять ті елементи A, яких немає в B. При множинних операціях над реляціями необхідно врахувати, що реляція є множино строк.

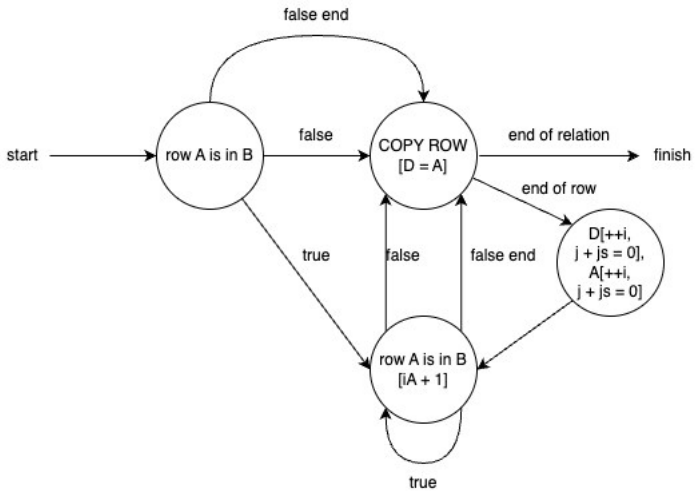


Рисунок 3.53 — функція-підграф віднімання реляцій

З рисунку 3.53 видно, що в даній функції використовуються як допоміжні функції-підграфи, так і окремий вузол. Спочатку порівнюються перша строка реляції Aна її налічення в B, оскільки, як було, на початку виконання лічильники скинуті в нуль. Якщо першої строки A нема в B, тоді вона копіюється до вихідної реляції. У наступному вузлі лічильники строкреляцій A та D інкрементуються. І так далі для інших строк реляції A, аж поки лічильники не дійдуть до її кінця.

Intersect. Операція пересічення дуже схожа на попередню, а саме копіюванням тих строк A, які є в B. Так, принцип дії даних функцій однаковий, різниця лише в тому, що в пересічення інверсна умова для копіювання строки до реляції результату до подібної в операції віднімання множин.

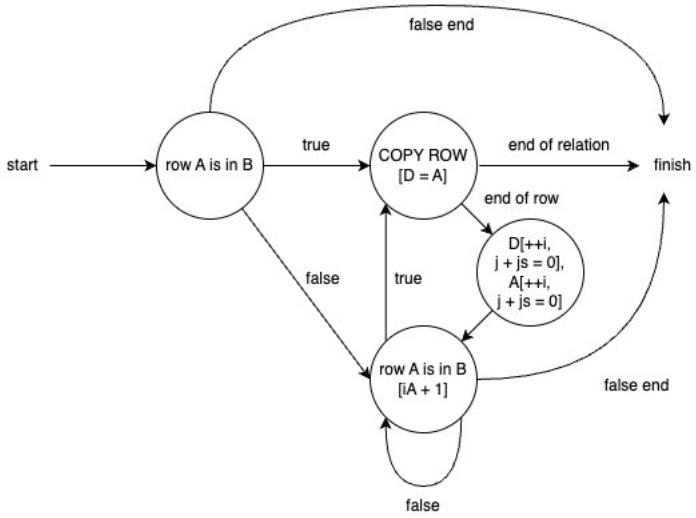


Рисунок 3.54 — функція-підграф пересічення реляцій

Як видно з рисунку 3.54, функція-підграф пересічення є подібною до віднімання з рисунку 3.53. Різниця лише в умові копіювання строки. Також, якщо у функції-підграфі пересічення буде задетектовано кінець реляції A, то функція завершує роботу.

Union. Операція об’єднання, що слідує з її назви, об’єднує дві реляції як множини строк. При операціях з множинами варто пам’ятати, що в множині не може бути однакових елементів. Саме тому банальна стовпчикова конкатенація не спрацює тут. Одним з ефективних методів

реалізації об’єднання є вертикальна конкатенація реляції A та різниці множини A та B, $A \cup B = A \circ_R^V B \backslash A$, де $\circ_R^V$ позначає вертикальну конкатенацію. Допоміжна функція-підграф виведена спеціально для того, щоб на початку копіювати всю реляцію A до D. Після чого реляція A використовується для перевірки належності кожної строки реляції B до неї.

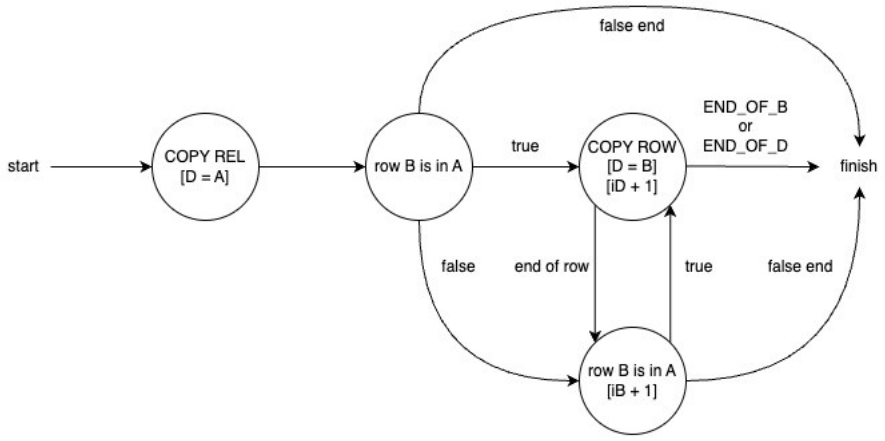


Рисунок 3.55 — функція-підграф об’єднання реляцій

Так, функція-підграф з рисунку 3.55 спочатку копіює всю реляцію A до D. Наступним кроком вона перевіряє чи є перша строка реляції B в A. Якщо є, тоді перша строка B копіюється в D, при цьому лічильник реляції D інкрементується, щоб перейти від останньої строки A до нової порожньої. І так далі для всіх інших строк реляції B. І якщо функція-підграф помічає кінець реляції B, тоді вона завершує свою роботу.

Project. Операція проекції є простим копіюванням певних атрибутів з реляції A до D. Так, функція-підграф використовує значення з полей *first attribute* та *attributes range* для копіювання відповідного діапазону атрибутів.

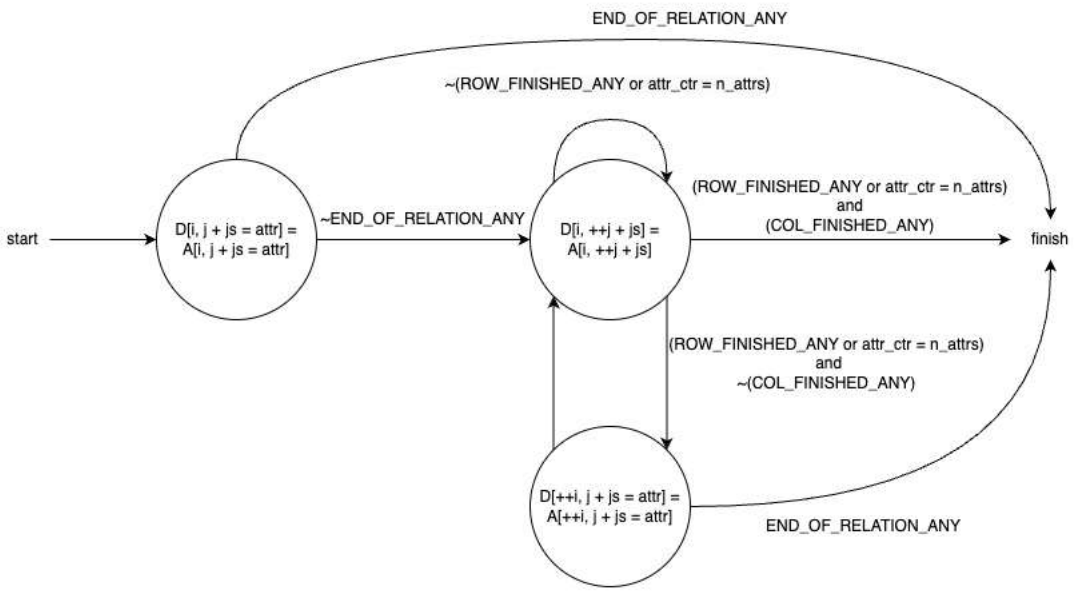


Рисунок 3.56 функція-підграф проекції реляції

Функція-підграф на рисунку 3.56 починає з копіювання першої строки початкового атрибуту (підвектора, перший елемент в якому є початковим атрибутом) реляції. Далі вона переходить у стан копіювання атрибутів однієї строки, аж доки не дістається до кінцевого атрибуту чи до кінця строки будь-якої реляції. Якщо у цьому стані функція не досягла кінця будь-якої реляції, то в наступному стані (вузлі) вона інкрементує лічильники строк, переводить лічильники атрибутів до початкового атрибуту, і копіює підвектор згідно нових адрес. Якщо ж у будь-якому стані функція досягає кінця будь-якої реляції, то вона миттєво завершує свою роботу.Також у даній функції-підграфі представлені наступні макроси:

- END_OF_RELATION_ANY — лічильники досягли кінця будь-якої реляції;
- ROW_FINISHED_ANY — лічильники досягли кінцястрочки будь-якої реляції;
- COL_FINISHED_ANY — лічильники досягли кінцястовпчика будь-якої реляції;

Дані макровизначення замінюють комплексні формули перевірки умов для будь-яких реляцій (тобто перевіряються лічильники всіх активних реляцій), і вони надалі будуть використовуватись для спрощення схем.

Select. Як вже було зазначено, параметрична селекторна функція вибирає з вхідної реляції лише ті строки, які відповідають параметричній умові. Для перевірки конкретних атрибутів вхідної реляції A були введені такі поля в інструкції SELECT як *first attribute* та *attributes range.* Для порівняння використовується "референсна" реляція B. Причому, в залежності від інструкції, референсна реляція може мати:

- такий же самий розмір, як реляція A;
- обмежену кількість атрибутів, які необхідні лише для порівняння;
- однорозмірна, якщо усі атрибути потрібно перевірити на відповідність одному елементу.

У даній роботі демонструється операція селекції першого типу, тобто де реляції A та B однорозмірні. Для інших операцій логіка суттєво не відрізняється.

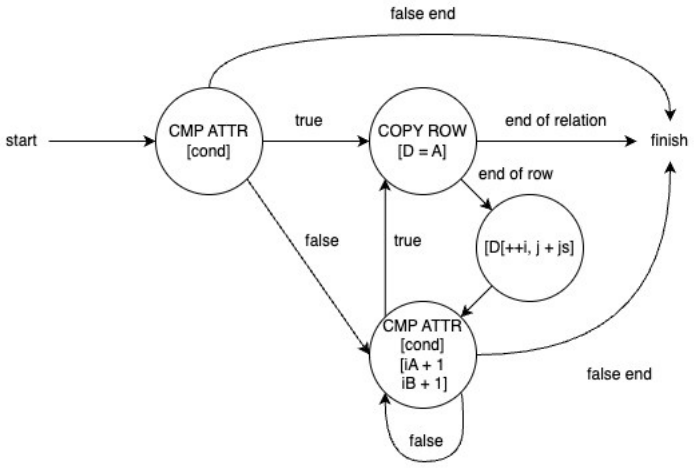


Рисунок 3.57 — функція-підграф селекторної функції

Спочатку функція-підграф на рисунку 3.57 порівнює перших дві строки реляцій A та B на відповідністьатрибутів умові *Cond.* При співпадінні функція копіює перший рядок A до першого рядку D, після чого наступний вузол інкрементує лічильник строк реляції D. І далі таким же чином перевіряються й інші рядки реляцій A та B. Коли функція-підграф досягне кінця будь-якої реляції, то робота завершиться.

Arithmetic. Арифметичні операції, як і селектроні, поділяються на 3 типи в залежності від розмірності другого операнду:

- Операції над реляціями ($R_{m\times n} \times R_{m\times n}$);
- Операції над реляціями ($R_{m\times n} \times R_{1\times n}$);
- Операції над реляціями ($R_{m\times n} \times R_{1\times 1}$);

де $R_{m\times n}$ — реляція з m кортежів розміру n. Хоча вище було вказано повні розмірності, проте в реальності дані операції обмежені по атрибутам значеннями *first attribute* та *attributes range.* Всі вищезазначені типи функцій-підграфів мають ідентичну структуру. Тому розглянемо для прикладу функцію-підграф арифметичної операції над реляціями ($R_{m\times n} \times R_{m\times n}$).

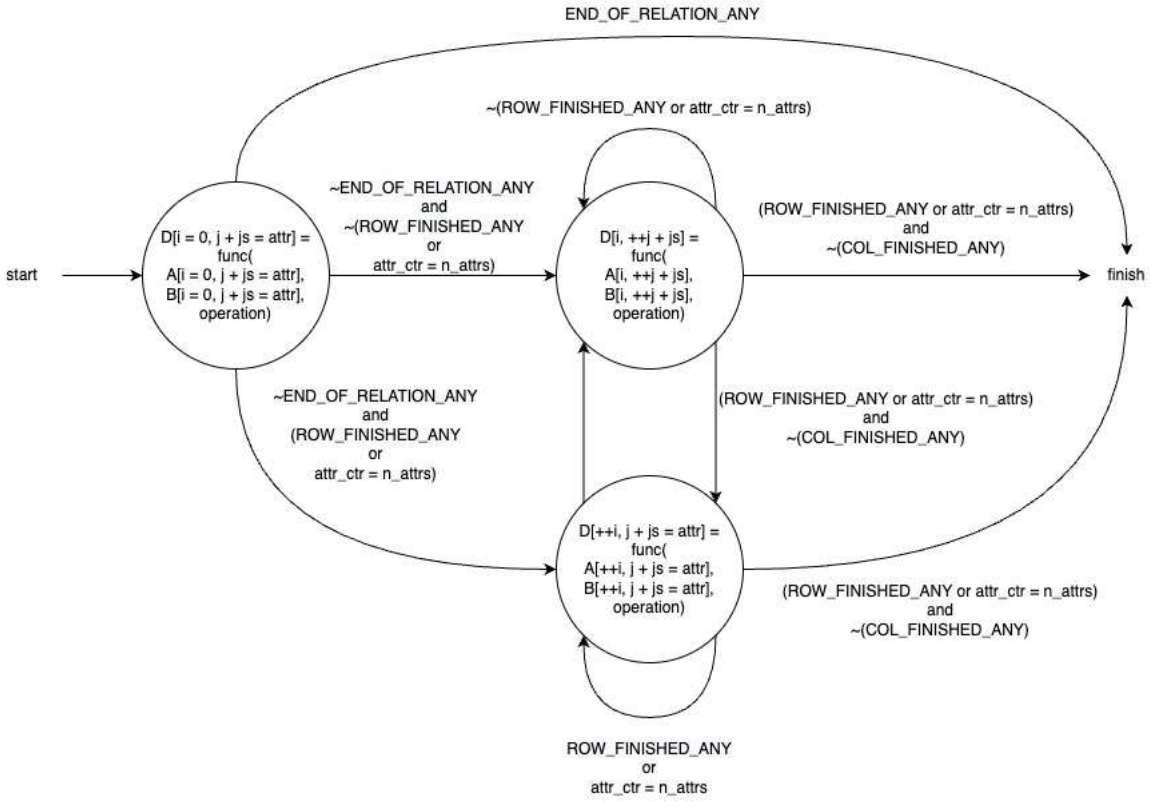


Рисунок 3.58 — функція-підграф арифметичної функції

Як видно з рисунку 3.58, функція-підграф починається з вузлу, який встановлює лічильники строк в нуль, а лічильники атрибутів у початковий індекс, після чого арифметичний блок виконує над даними задану операцію та зберігає у пам’яті. Якщо у першому вузлі не було досягнуто ані кінця діапазону атрибутів, ані кінця реляції, тоді відбувається перехід до лічення

по рядкам з виконанням відповідної функції. Після досягнення кінця строки функція переходить до вузлу інкрементації лічильників реляції та переведення лічильників атрибутів на початкову позицію, відповідно з виконанням операції. І так до кінця будь-якої реляції. Якщо ж кінець будь-якої реляції детектується на будь-якому вузлі, тоді функція-підграф завершує роботу. Також передбачена ситуація, коли потрібно опрацювати лише один стовпчик. Тоді граф відразу після першого вузлу переходить до вузлу інкрементації лічильників строк. Параметр операції *operation* береться з регістру *Arithm*.

Accumulate. Арифметичні операції з накопиченням відокремлені від основних, оскільки для них логіка вже відрізняється. Як було сказано вище, дані операції опрацьовуються саме для стовпчиків реляції з послідовним накопиченням результату. Проте, вони теж поділяються на 3 види за розміром:

- Операції над реляціями ($R_{m\times n} \times R_{m\times n}$);
- Операції над реляціями ($R_{m\times n} \times R_{1\times n}$);
- Операції над реляціями ($R_{m\times n} \times R_{1\times 1}$);

Різниця між цими операціями аналогічна як і у звичайних арифметичних операцій. Тож як і зі звичайними арифметичними операціями, для прикладу використаємо операцію з реляціями ($R_{m\times n} \times R_{m\times n}$).

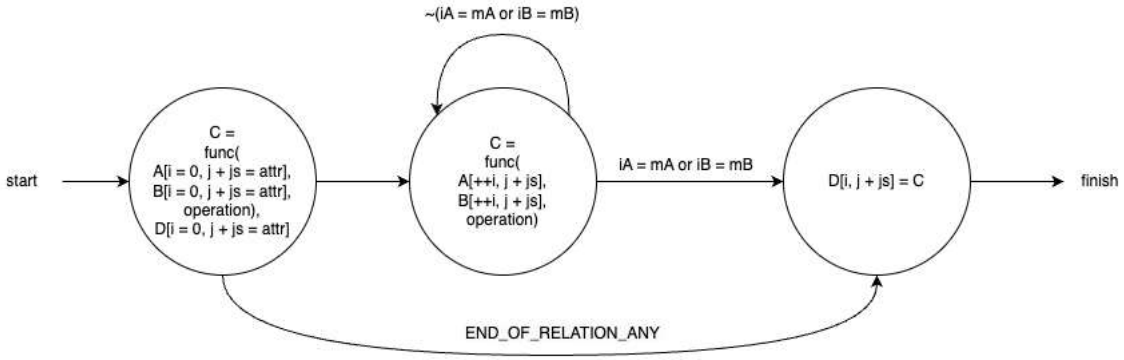


Рисунок 3.59 — функція-підграф арифметичної функції

Як видно з рисунку 3.59, дана функція-підграф є меншою за звичайні арифметичні з рисунку 3.58, оскільки вони працюють тільки зі стовпчиками.

Так, всі лічильники у початковому вузлі встановлюються у початкову позицію, а в регістр накопичення новий результат. У наступному стані відбувається циклічна інкрементація лічильників строк за допомогою петлі. І під кінець реляцій A чи B, переходить до вузлу запису вмісту регістра накопичення до пам'яті. Проте якщо кінець цих реляцій буде детектовано ще на початковому вузлі, то наступим вузлом відразу стає запис вмісту регістру накопичення. Цим функція-підграф накопичувальної арифметики відрізняється від інших, тут обов'язково результат обичслень повинен бути записаний до пам'яті в окремому вузлі після завершення всіх дій.

Join. Операція, так званої, натуральної конкатенації реляцій дуже просто реалізується. Так, в ній конкатенуються відповідні строки реляцій $\{< a_n^m >\} join \{< b_l^m >\} = \{< a_1^i, \dots, a_n^i, b_1^i, \dots, b_l^i > | \ i = \overrightarrow{1, n \cdot l}\}$. Реалізуємо даний принцип у графі.

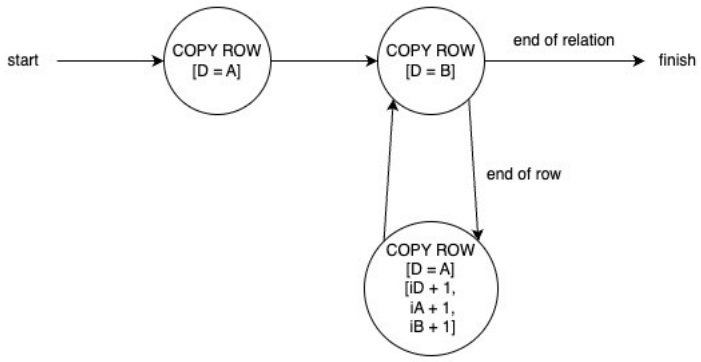


Рисунок 3.60 — функція-підграф з'єднання реляцій

З рисунку 3.60 видно, що спочатку йде допоміжна функція-підграф копіювання строки A до першої строки D, при цьому, ніякого переходу на іншу строку не відбувається. Далі граф безумовно переходить до функції копіювання строчки B. Якщо після цього було досягнуто кінця реляції B чи D, то функція завершує виконання. В іншому випадку вона переходить до функції копіювання рядка A до D, при цьому інкрементуються лічильники строк всіх реляцій.

Times. Функція декартового добутку реляцій є одною з найскладніших. З визначення отримуємо, що одна строка реляції A почергово конкатенується

з усіма строками реляції B, і так для всіх строк з A. Дана операція в програмуванні реалізуєься через подвійний цикл. Приклад у псевдокоді:

$$
\begin{aligned}
&for i = \overrightarrow{1, n}\{ \\
&\quad for t = \overrightarrow{1, l}\{ \\
&\qquad d^{i \cdot l + j} = a^i \circ_R b^t \\
&\quad \} \\
&\}
\end{aligned}
$$

Так $i \cdot l + j$-та строка реляції D дорівнює конкатенації i-ї строки реляції A та t-ї строки реляції B. Реалізуємо даний принцип у графі.

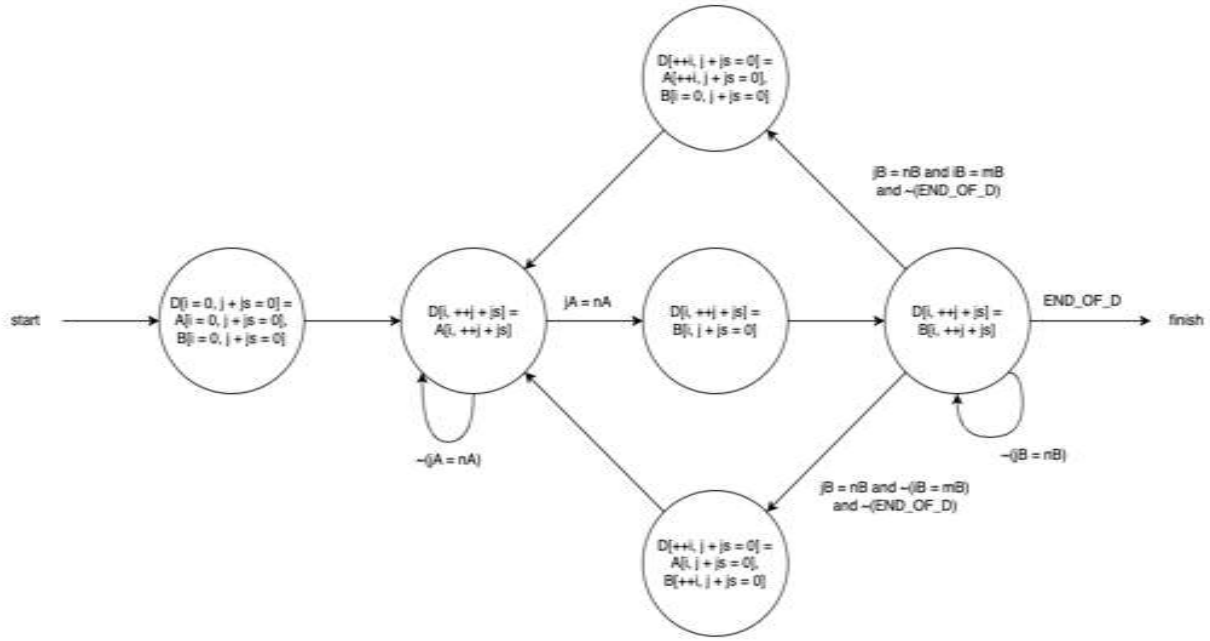


Рисунок 3.61 — функція-підграф декартового добутку реляцій

Як видно з рисунку 3.61, подвійний цикл у графах реалізується у вигляді двох замкнутих підграфах зі спільним “основним тілом” циклу. На початку функції-підграфу усі лічильники усіх реляцій скидаються в нуль. Далі граф переходить до початку циклу, тобто вузлу копіювання строки A до D, наступним кроком до тієї ж самої строки реляції D записується перший атрибут строки B. Після цього граф переходить до циклічного копіювання строки реляції B.По завершенню циклічного копіювання наявні 3 шляхи:

- Кінець реляції B не досягнуто;
- Кінець реляції B досягнуто;

• Кінець реляції D досягнуто;

У першому випадку граф переходить до вузлу, який інкрементує лічильники строк лише дял реляцій B та D; лічильники атрибутів усіх реляцій скидуються в нуль; а до наступної строки першого атрибуту реляції D копіюється перший елемент поточної строки реляції A. І далі функція знову переходить до початку циклу. Другий випадок відрізняється від першого тим, що після копіювання строки B до D, наступний вузол інкрементує лічильник реляції A; і також робить все те ж саме, що й відповідний стан для першого випадку. Після чого, як і в першому випадку, повертається до основного тіла циклу. У третьому випадку робота функції-підграфа завершується, оскільки саме реляція D має максимальний розмір, оскільки повинна вмістити декартовий добуток A та B.

Отже в даному підрозділі було виведено основну половину функцій-підграфів для блоку керування операціями. Виведені графи демонструють потужність імплементації кінцевого автомату у якості засобу керування та задавання операцій. А також вони демонструють гнучкість системи та здатність її реконфігурувати. Таким чином, можна реалізувати будь-яку базову функцію процесору, задавши її функцію-підграф, звісно що в межах даної архітектури.

3.10 Висновок до розділу 3

Отже в даному розділі було описано реляційний процесор, що базується на матричному обичслювачі. Так, були модифіковані ядра керування та виконання, пам’яті імен та реляцій. При чому керуюче ядро зазнало не значних модифікацій, коли як виконавче ядро було майже повністю реструктуровано. Всі покращення були введенні задля покращення швидкодії, покращення програмування та реконфігурації, а також задля адаптації до реляцій.

З керуючого ядра було видалено блок перевірки розмірностей, адже його цілком можна замінити перевіркою розмірностей на етапі компіляції, не відбираючи ресурсів кристалу. Впливовим нововведенням стало створення механізму стеку даних, і підримка інструкціій швидкого стекування. Стекування використовується здебільшого при переході до виконання іншої функції. Таким чином до стеку зберігаються дані з певних регістрів та точка повернення до виконання минулої функції з моменту після переходу.

Виконавче ядро зазнало сильної реорганізації. Так, блок підтримки ізоморфізму був розбитий на 2 окремі блоки — блок обчислення адрес та блок керування операціями. З блоку обробки даних було прибрано транспонування через його непотрібність. Також було замінено регістр послідовного/паралельного читання на звичайний, а сам АЛУ тепер підтримує широкий спектр операцій, у тому числі з накопиченням. Серйозно вплинуло на виконавче ядро й нова пам’ять реляцій. Головною зміною є механізм адресації даних, а саме зменшення мінімального кроку з розміру підвектора (параметру векторизації) до одного елементу. Все це досягнуто за допомогою механізму “зсуву підвектора”. Його суть полягає в тому, що він може витягнути відповідні елементи підвектору з наступної адреси, а потім отримане значення циклічно зсунути для отримання правильного порядку слідування. Такий принцип застосовуєтсья при читанні та запису даних. Перевірка даної пам’яті в САПР Quartus 21.1 показала, що даний механізм працює як задумувалось.

Блок обчислення адрес тепер вміщує в собі всі необхідні елементи для розрахунку адрес та лічення по строкам та атрибутам. Тепер, для кожної реляції виділено свій власний лічильник строк та атрибутів. Лічильник атрибутів, через нову пам’ять реляцій, став сдвоєним. У ньому присутній лічильник для обходу самих комірок, а також окремий для зсуву вектора. Лічильник строк також став сдвоєним. Викликано це тим, що з одної сторони йому необхідно кожного разу збільшуватись на кількість атрибутів, а з

іншого відстежувати переповнення саме по параметру кількості стовпчиків. Відтепер, один лічильник слугує для лічення по адресам, а інший для відлічення переповнення. Також введено інструментарій для лічення вздовж однієї строки в межах певного діапазону атрибутів. На кінець, тепер через можливу невизначенність кількості строк вихідної реляції, блок передає дані з лічильника стовпчиків до керуючого ядра.

Блок керування операціями представляє з себе кінцевий автомат, що було описано функціональним графом. Основні операції задаються спеціальними функціями-підграфами, що розміщуються між станом очікування та скиданням. Було доведено підтримку функціональними графами сигнатурних операції ППА, що свідчить про їх повноту в межах архітектури. А також були реалізовані 9 з 18 основних операцій, а саме: віднімання множин, пересічення множин, об’єднання множин, проекції, селекції, арифметичної, арифметичної з накопиченням, натуральної конкатенації та декартового добутку.

Отже вищевикладені нововедення покращили процесор та дозволили йому підтримувати тип даних реляцій. А отже завдання розділу виконано!

Розділ 4 Розроблення стартап-проекту

Від 13 жовтня 2016 року, відповідно до рішення Методради Національного Технічного Університету України “Київський Політехнічний Інститут” від 15 вересня 2016 року, вступило в силу розпорядження по університету, згідно з яким студенти програми магістра-професіонала мають зробити зі своїх магістерських дисертацій стартап-проекти. Метою такого рішення є залучення більшої кількості учасників у конкурсі Sikorsky Challenge. Саме через це виключно негативне рішення виникла потреба в даному розділі.

Мета цього розділу розробити життєздатний стартап проект для подальшоно переформування у повноцінне підприємство. У цьому розділі досліджуються можливості адаптації реляційного процесору для його подальшого просування на ринку. Для досягнення мети ставляться опрацьовуються наступні питання:

- Дослідження потенційної цільової аудиторії;
- Розробка мінімального життєздатного продукту на основі реляційного процесору, що задовільнить потреби потенційної цільової аудиторії;
- Бізнес-модель стартапу;
- Оцінювання ринку стартапу;
- Стадії стартапу та фінансування;
- Рекламування стартап проекту;
- Подальше масштабування стартап проекту.

4.1 Дослідження потенційної цільової аудиторії

Очевидно, що серед потенційної цільової аудиторії є ті, що використовують комп’ютерні сервери, особливо для хмарних баз даних баз даних великого навантаження. Насамперед нас цікавлять бізнеси та інформаційно-комунікаційні підприємства, що використовують реляційні бази даних (БД). Згідно статистикиз [36, 37], з 10-ки найбільш популярних систем управління базами даних (СУБД) більшість з них підтримує реляційні

бази даних, дві з яких підтримують виключно реляційні БД. А якщо глянути у топ-100, то виходить що чимала кількість з них створювалась лише для реляційних БД. З останього розуміємо що реляційна БД на сьогоднішній день є вкрай популярною.

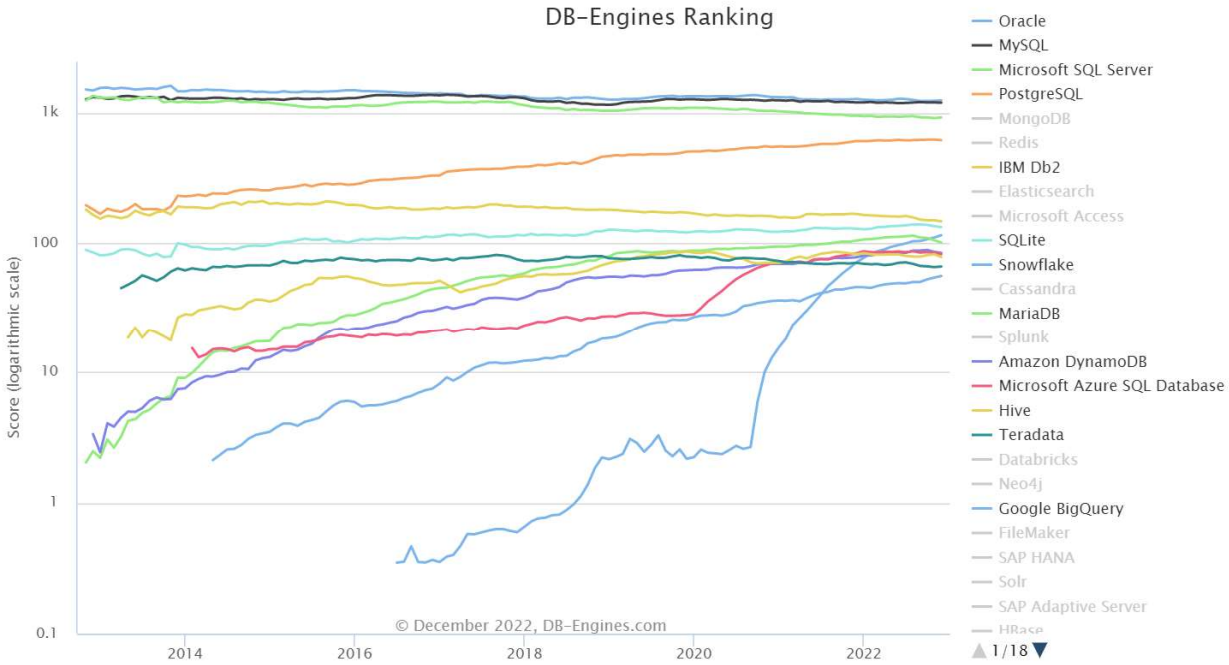


Рисунок 4.1 — графік популярності СУБД, що підтримують реляційні БД

На сьогоднішній день БД набули дуже широкого використання. Їх використовують як великі корпорації по типу Meta, Apple, Netflix, Google, Amazon (MANGA), більш відомі як велика п’ятірка найбільших IT корпорацій світу, та Microsoft; так і звичайні користувачі для своїх особистих цілей. При цьому самі сервери для баз даних можуть досягати площею декілька футбольних полів.

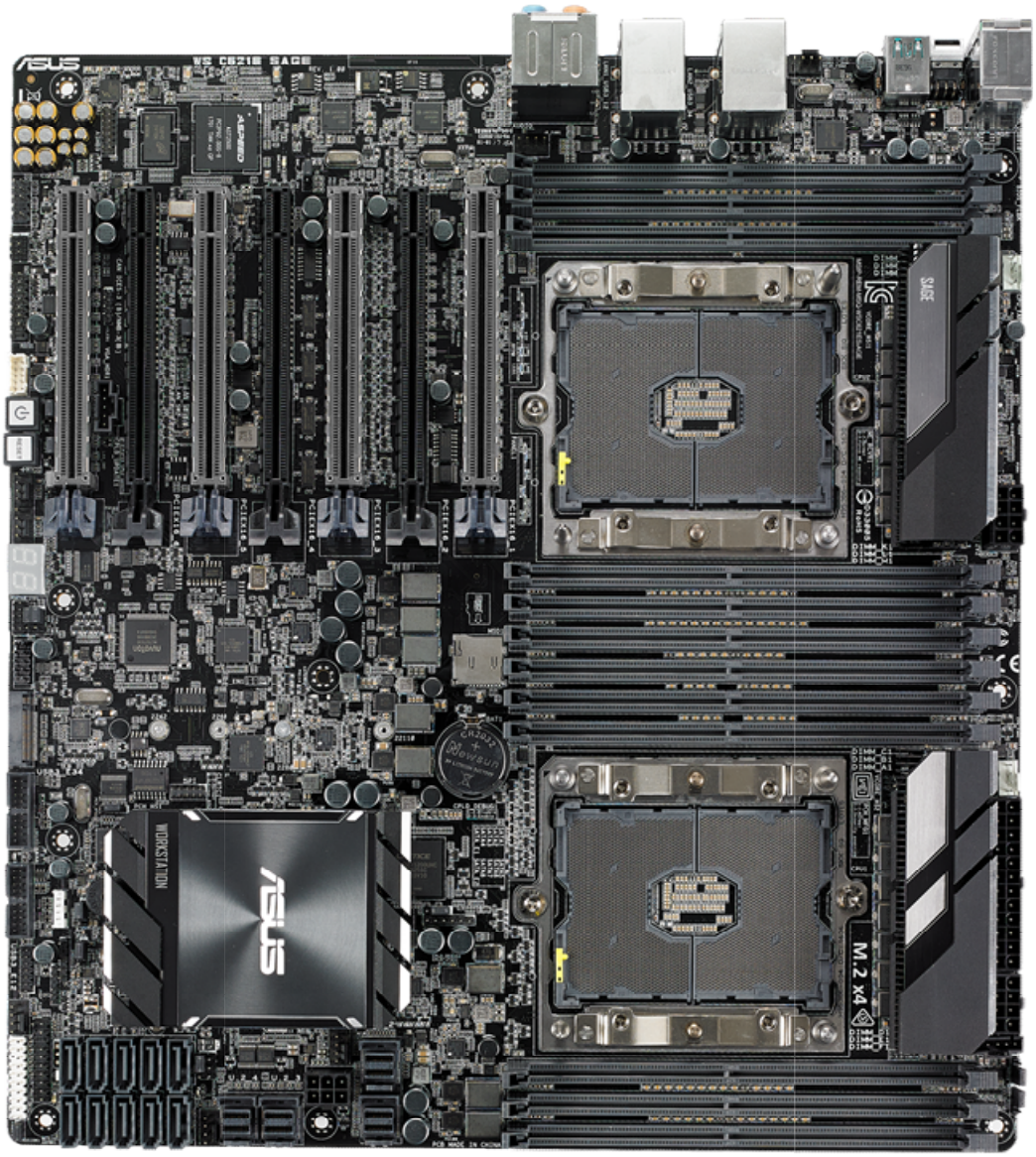
Найбільш апаратні засоби для менеджменту БД використовують або процесори загального застосування, або ж адаптують для обробки великих масивів даних прискорювачі, що не призначені для роботи з БД. Таким чином спеціальний процесор для реляційних БД може зробити їх обробку в рази ефективнішою.

Підсумок. Основною цільовою аудиторією стартапу є бізнес клієнти, що активно використовують бази даних великих об’ємів, у тому числі MANGA.

4.2 Мінімальний життєздатний продукт

Ключовим фактором для стартап проекту є потреби потенційної цільової аудиторії, тож при розробці мінімального життєздатного продукту (minimal viable product, MVP) орієнтуємось на те, в якій формі найзручніше використовувати продукт потенційним клієнтам. При цьому однією з ознак успішного стартапу є продумання масштабування на етапі розробки MVP.

Фізичний пристрій. Для визначення найзручнішої форми використання продукту потрібно розглянути яке обладнання використовується для запуску на ньому сервера з реяційною базою даних. Так, найбільш розповсюдженним засобом є спеціальні комп'ютерні сервери. Комп'ютерні сервери представляють собою великі кластери зі спеціальних серверних комп'ютерів. Серверні комп'ютери, в свою чергу, відрізняються від звичайних домашніх комп'ютерів тим, що перші підтримують більший обсяг обчислювальної техніки як центральні процесори (Central Processing Unit, CPU), периферійні пристрої; а також декілька високопропускних мережевих каналів. При цьому, серверні материнські плати (Extended ATX, EATX) мають ті ж самі роз'єми, що й звичайні домашні (Advanced Technology Extended, ATX).



a)

б)

Рисунок 4.2 а — серверна материнська плата, 4.2 б — ATX материнська плата

З рисунків 4.2 а та 4.2 б видно, що серверні та користувацькі плати мають однакові роз’єми, окрім сокету процесора, оскільки серверні процесори представляються в іншому модельному ряді, та мають більшу кількість контактів. Проте, нас цікавить конкретно роз’єми (слоти) для підключення периферійних карт. В обох випадках використовуються PCI-E

(Peripheral Component Interconnect Express), що є стандартом для комп’ютерної периферії.

Далі необхідно визначитись до якого класу обчислювальної техніки належить прилад. Реляційний процесор підпадає до периферійних пристроїв, адже повністю заминити CPU він не може, і направлений на обробку конкретного типу даних. А як вже було розібрано, у периферійних пристроїв для з’єднання з материнською платою використовуються PCI-E.

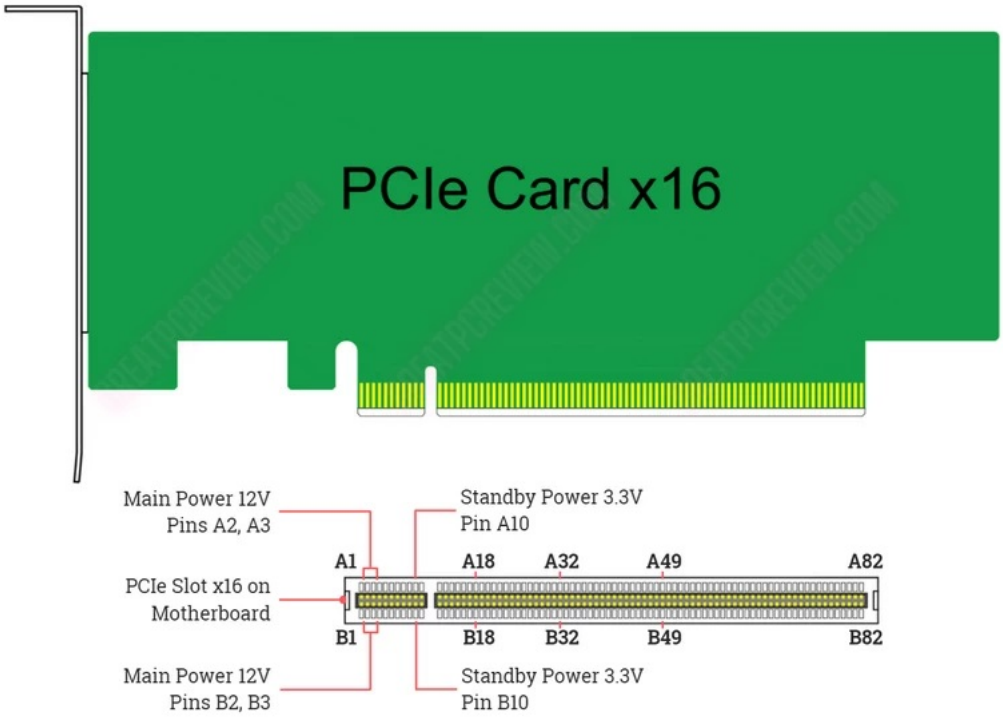


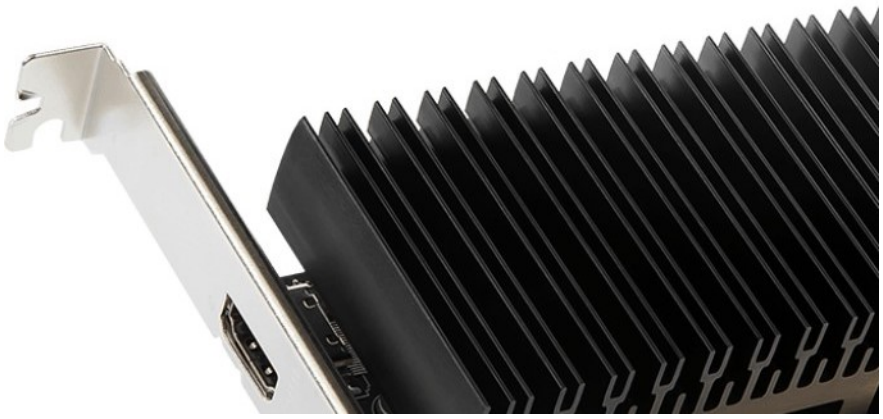
Рисунок 4.3 — роз’єм PCI-E x16

На рисунку 4.3 зображена модифікація PCI-E на 16 ліній, тобто x16. Такий стандарт є типовим для потужних периферійних пристроїв, адже забезпечує максимальну пропускну здатність каналу між CPU та периферійним пристроєм. Тому в якості роз’єму обираємо PCI-E x16.

Найновішим стандартом, що використовується в індустрії, є PCI-E 4.0. Проте, впродовж довгого часу виробники використовували стандарт 3.0. Для збереження сумісності продукт забезпечує підтримку як 4.0, так і 3.0

Типовим для потужних периферійних пристроїв є використання системи охолодження. Зачасту корпус процесора, як і корпуси компоненттів живлення та пам’яті, не здатні самостійно відвести тепло, тому включення

охолодження є необхідною мірою. Саме ж охолодження поділяється на активне та пасивне, При цьому активне охолодження поділяється на повітряне та рідкісне.



а)

б)

в)

Рисунок 4.4 а — плата з пасивною системою охолодження, 4.? б — плата з активною повітрянною системою охолодження, 4.? в — плата з активним рідким охолодженням.

Як видно з рисунку 4.4 а, у пасивному охолодженні використовується лише радіатор, що розсіює тепло вздовж своєї поверхні. Такий тип охолодження є найдешевшим та використовується у малопотужних приладах. Активне повітряне охолодження використовує у парі з радіатором вентилятори, що відводять тепле повітря. Такий тип охолодження є оптимальних за співвідношенням ціни та якості, й використовується у більшості потужних професійних та домашніх периферійних пристроях. Активне рідке охолодження найкраще відводить тепло від компонентів, оскільки теплопровідність рідин більша за теплопровідність повітря. Такий тип охолодження є найдорожчим, при цьому найдешевша система рідкого охолодження (і з сумнівною якістю) вартує як хороша система активного повітряного охолодження; хоча різниця в продуктивності мала, і та на корист повітряного. Така система використовується вже для роботи в екстремальних умовах з підвищенням тактової частотои (OC, overclocking англ.). Прилад для MVP розрахований на використання при нормальних умовах (закрите приміщення з діапазоном температур 10-35 градусів цельсію, без OC), апо

потужності співставне з відеокартами середньої-високої потужності. Тому для охолодження приладу береться активне повітряне охолодження. З естетичних та захистних поглядів охолодження зазвичай накривають захистним кожухом, що й робиться для цього пристрою.

Отже, фізичний пристрій являє собою периферійну плату з PCI-E роз'ємом, що дозволить його використовувати як у професійних серверах, так і в домашніх. Пристрій підтримує останню версію PCI-E 4.0, і для сумісності 3.0. Для охолодження електричних компонент використовується активне повітряне охолодження, що є оптимальним по відношенню ціни та якості. Також для естетичності та захисту використовується зовнішній кожух.

Архітектура процесору пристроя. У сучасній комп'ютерній архітектурі використовується принцип багатоядерної архітектури. Вона дозволяє як досягти значного бусту обробки за рахунок розпаралелювання операцій. А також багатоядерну архітектуру легко масштабувати у плані потужності. Таким чином іде збільшення кола потенційних клієнтів за рахунок розширення асортименту для різних бюджетів. Серед найновіших методів виготовлення напівпровідникових виробів є використання для одного кристалу декількох підкладинок з їх наступним з'єднанням. Такий метод дозволяє збільшити розміри кристалу, при цьому без збільшення відсотку браку.

Для архітектури найвищого рівня (top-level architecture) було обрано багатоядерну. Разом з декількома ядрами використовуютсья кеші різного рівня, де на одне ядро виділяється кеш найнижчого рівня. Ще в прискорювачі матричних операцій було зроблено базис для імплементації багатоядерної архітектури. У реляційному процесорі було розглянуто декілька способів по впровадженню багатоядерної архітектури, а на даний робота над багатоядерністю продовжується (дослідження по багатоядерності до магістерської дисертації не ввійшли через "сирість" матеріалу). При цьому,

окрім багатоядерності необхідно додати спеціальні блоки для роботи з зовнішньою памяттю. В якості зовнішньої пам’яті було обрано пам’ять стандарту GDDR6. Це сучасний стандарт пам’яті, що призначений для роботи з великими кластерами пам’яті. Саме такий стандарт використовується у сучасних відеокартах. Контролери внутрішньої пам’яті вже входять до багатоядерної архітектури, тому окремо про них говорити нема сенсу. Також необхідно модифікувати блок I/O для підтримки стандартів PCI-E 4.0, 3.0.

Отже, архітектура найвищого рівня реляційного процесору була обрана багатоядерною. При цьому вона має бути параметрично масштабованою та підтримувати кеш-пам’ять різних рівнів. До процесору також входять блоки, що працюють із зовнішньою пам’ятю типу GDDR6. А блок I/O перероблюється таким чином, щоб підтримувати стандарти PCI-E 4.0, 3.0.

Програмне забезпечення. Останнім етапом в розробці периферійного пристрою є написання його програмного забезпечення. У даному випадку під програмним забезпеченням мається на увазі:

- підтримка зв’язку периферійного пристрою з CPU (драйвери);
- програмний інтерфейс (Application Programming Interface, API) для роботи з реляційним процесором (посередних між драйверами та високорівневими мовами програмування);
- Компілятор високорівневого коду в асемблерні інструкції реляційного процесору.

Саме такий набір програмного забезпечення можна вважати повним для периферійного пристрою.

З розробкою драйверів є один неприємний нюанс. Драйвери пишуться для кожного ядра операційної системи (ОС) окремо. Тобто не вдасться запустити один і той же драйвер на всіх операційних системах. На сьогоднішній день є 3 найбільш популярні ОС: GNU/Linux, Windows, MacOS. Серед них найчастіше для серверів використовується саме GNU/Linux, в особливості через її ядро Linux. Windows найбільше використовується у

домашніх (найчастіше ігрових) та корпоративних комп'ютерах. MacOS найчастіше використовують для професіональних потреб (здельшого креативних), та є ексклюзивом комп'ютерів компанії Apple пд назвою Mac. Очевидно що найбільш сприятлива ОС для написання драйверів є GNU/Linux; особливо враховуючи що всі компоненти даної ОС є відкритими, тобто їхній код лежить у відкритому доступі.

Сьогодні наявно безліч високорівневих мов програмування. Проте серед них особливо виділяється мова програмування С. Так, ця мова серед усіх високорівневих, є найбільш використованою для написання драйверів, примітивних функцій ОС, а також що інші мови програмування часто використовуються в якості проміжного звена між залізом саме мову програмування С. Тому логічно буде реалізувати API саме під мову С.

Коли мова програмування, до якої реалізується API обрано, залишилось реалізувати компілятор з обраної мови в асемблерні інструкції. Для мови програмування С існує декілька компіляторів, проте серед них компілятор GCC є стандартним для ОС GNU/Linux. Більше того, більшість проектів, пов'язаних із залізом використовують саме його. Також даний компілятор є відкритим, а виробники мікроконтролерів та мікро комп'ютерів найчастіше модифікують саме GCC для забезпечення інструменту компіляції розробників.

Таким чином у якості ОС для написання драйверів було обрано GNU/Linux. API реалізується для мови програмування С. А в якості компілятору буде виступати модифікація GCC під реляційний процесор. Також не варто забувати про SQL, оскільки інструкції були розроблені з огляду на підтримку команд SQL один до одного.

Вартість приладу. Цей показник не тільки визначає трати та потенційні прибутки від них, а також визначають цільову аудиторію згідно її бюджету. Найбільшою цільовою аудиторією на ринку продаж вважається з середнім бюджетом.

Оскільки прямих конкурентів поки немає, тому відштовхнемося від найбільшого суміжного ринку, а саме ринку відеокарт. При цьому варто зробити значущу **примітку**: хоча компанія NVidia й має в асортименті професійні відеокарти, проте їхня фінансова модель сильно відрізняється від ігрових і є великою мірою штучною.Так, за одну й ту саму продуктивність гравець та професіонал віддають зовсім різні гроші, єдина різниця в драйверах та доступах до певних технологій, що використовуються у професійних програмах та САПР.

У відеокартах середнім бюджетом вважається ціна 399-499$. Ціна в 500$ вважається за психологічний бар'єр, більше якої пристрій вважається вже високобюджетним. Одже орієнтуємось саме на середньоціновий діапазон цін. Продукти для цієї ланки відзначаються найбільшим співвідношенням ціни та якості, а також співвідношенням продуктивності та енергоспоживання.

Підсумки. Одже MVP приладом є периферійна карта з роз'ємом PCI-E, активним повітряним охолодженням а захистним кожухом. Архітектура найвищого рівня реляційного процесору є багатоядерною та легкомасштабованою. На етапі MVP підтримується лише ОС GNU/Linux з її стандартними інструментами компіляції коду мови програмування C, при цьому є варіант трансляції команд мови SQL в асемблерний код. Прилад знаходиться у середньобюджетній ланці. Така ланка є оптимально по відношенню ціни та якості.

4.3 Бізнес-модель

Оскільки основною цільовою аудиторією було обрано підприємства та корпорації, тому цей стартап працює за моделлю business-to-business (B2B).

Для отримання повноцінного "паспорту" бізнес моделі найкращим способом є побудова так званого шаблону Canvas. Для стартапів, а також для більшого розуміння бізнес-моделі для самого підприємця, найкраще всього

використовувати “ощадливий” шаблон Lean Canvas. Ощадливий стартап (англ. Lean Startup) — це концепція “ощадливого” запуску і розвитку компаній, в її основі лежить максимально оптимальне витрачення ресурсів, засноване на науковому підході до впровадження будь-якого нового продукту, послуги або ідеї: масштабування ідеї у разі успіху, визначеного тестуванням та оцінюванням результатів за метриками [36].

Таблиця 4.1 — Lean Canvas для стартап проекту

<table>
<tr>
<td rowspan="2">Ключові проблеми споживачів

1. Незадовільна швидкість роботи СУБД
2. Надмірне споживання електроенергії
3. Завелика кількість обладнання

Існуючі рішення

1. Використання багатоядерних CPU
2. Адаптація графічного прискорювача для роботи СУБД</td>
<td>Рішення

Створення спеціального апаратного прискорювача для СУБД, що буде ефективно оброблювати запити як по швидкості, так і по споживаній потужності</td>
<td rowspan="2">Унікальна ціннісна пропозиція

Спеціальний процесор, що прискорює роботу СУБД з можливістю задавати нові інструкцій та прямий переклад мови SQL в асемблерний код

Концепція високого рівня

Швидше та ефективніше</td>
<td>Несправ

1. Уніка
розро
науков
2. Нови
ринку о
пр</td>
</tr>
<tr>
<td>Ключові метрики

1. Дохід
2. Впізнаваність на ринку
3. Налагодженні зв’язки з виробником процесорів
4. Можливість робити систематичні наукові дослідження</td>
<td>Канал
1. Нау
виставк
жур
2. Да
техноб
сфері д
3. Сторін
соціал</td>
</tr>
</table>

Побудована Lean Canvas для стартап проекту надає повну інформацію про його бізнес-модель. Так, тут є пункти про клієнтський сегмент, їхні проблеми, потоки доходів, рішення для вирішення проблеми, унікальну ціннісну пропозицію, канали просування, ключові метрики успіху, структуру витрат та несправедливі переваги. Даний перелік повністю характеризує як сам стартап, так і його бізнес-модель. Проте в Lean Canvas випускається одна вкрай важлива деталь саме для бізнесу, що займається електронікою. Так, для цих компаній вижливою інформацією в моделі є бізнес-модель виготовлення продукції. А саме визначається взаємодія між компанією-замовником та компанією-виробником. Даний стартап є замовником у оригінального виробника обладнання (original equipment manufacturer, OEM). OEM компанія виготовлює деталі та обладнання на замовлення іншої компанії. OEM-

контракт під собою розуміє, що виробник виробляє та складає виріб за кресленнями та документами замовника. Отже з цим уточненням бізнес-модель побудовано.

4.4 Оцінювання ринку стартап-проекту

Розглянемо коротко сучасний стан ринку напівпровідникових процесорів. На ринку CPU основні позиції займають компанії, що були там майже з самого початку. Мова йде про Intel, AMD. Так, вони є єдиними виробниками процесорів, що є сумісними з X86 архітектурою. Дана архітектура до нещодавна була чи не єдиною архітектурою для «великих» комп'ютерів. Проте в 2020 році в боротьбу втрутилась компанія Apple зі своїм чіпом M1. Даний чіп набагато підняв продажі ноутбуків компанії MacBook. І відтак саме ноутбуки компанії Apple вважаються на сьогодні найбільш оптимальними. Також невелику долю ринку займають процесори ARM, хоча на свому ринку мікроконтролерів та мобільних ядер ця компанія є найпотужнішою.

На ринку виготовників напівпровідникових процесорів наявна сильна дуополія, при чому в одного виготовника вдвічі більша перевага по відсотку ринку. Черги на замовлення даних корпорацій є надзвичай великими, тому вони вибирають по пріоритетності крупніших замовників. Враховуючи дефіцит та штучну інфляцію, спричинену німецькими виробниками автомобілів, дані корпорації надають перевагу крупним гравцям по типу Apple, NVidia, AMD, Intel. Таким чином наразі вибитися в лідери досить проблематично. Проте існують і менші компанії та стартами, що уживаються поруч з вищезазначеними гігантами.

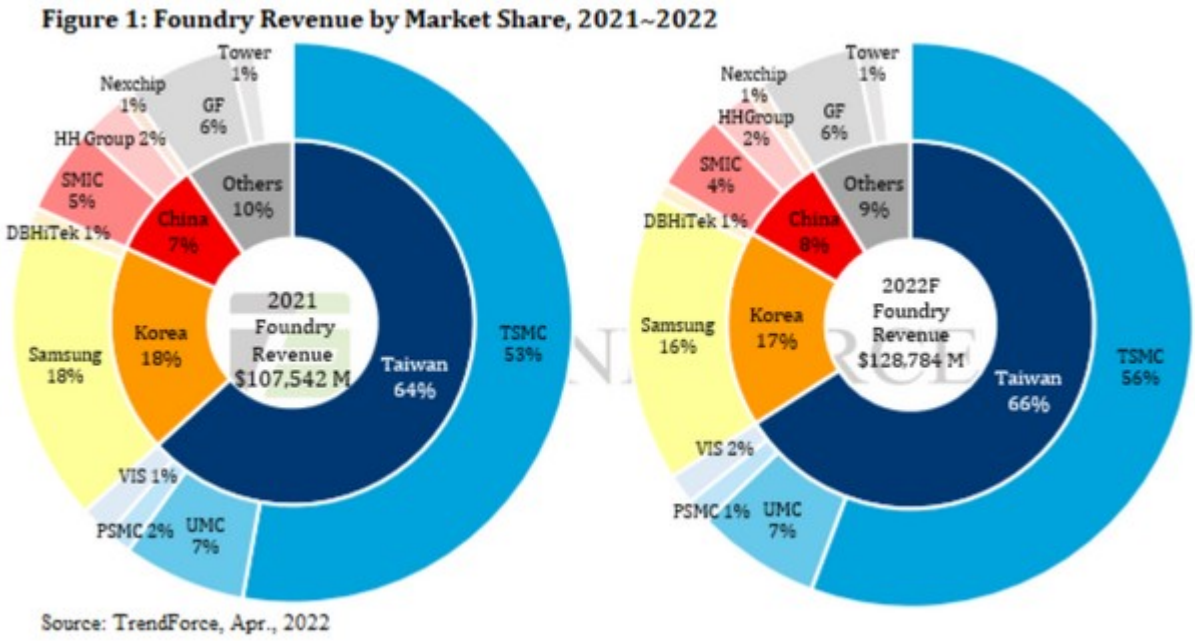


Рисунок 4.5 – долі виробників напівпровідникових мікросхем за 2021-2022

Незважаючи на такі здавалося б несприятливі умови для початку бізнесу в цій галузі, існує чимала кількість стартапів, що функціонують на даному ринку. Ось одні з найуспішніших стартапів в даній області: Celebras Systems, SambaNova Systems, Hailo, GrAI Matter Labs, Nervana Systems, Mythic, Lightmatter, Syntiant, Flex Logic. При тому, всі вони засновані між 2010 та 2020 роками. Вони всі спеціалізуються на створенні процесорів, що спеціалізуються на обчисленнях нейроних мереж, так звані нейронні процесори [48]. Отже, на даному ринку є місце тим, хто вносить до нього інновації.

Далі заповнюються усі необхідні таблиці для аналізу ринку.

Таблиця 1.2 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	4
2	Загальний обсяг продаж, грн/ум.од	259,38 млрд $
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Дефіцит на ринку виготовлення напівпровідникових процессорів, перевага надається крупним гравцям
5	Специфічні вимоги до стандартизації та сертифікації	Стандартні вимоги до виготовлення напівпровідникової обчислювальної техніки
6	Середня норма рентабельності в галузі (або по ринку), %	21.3%

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають. Фактори в таблиці подавані в порядку зменшення значущості.

Таблиця 4.3 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Розробка одним з лідерів ринку аналогу	Пряма конкуренція з набагато потужнішим конкурентом	Використати перевагу в першості та збільшити відрив в освоєнні потенційним конкурентом ринку

Продовження таблиці 4.3

2	Неосвоєнність ринком України напівпровідникової галузі, зменшення фінансування стартапів через повномасштабну війну	Потенційно мале фінансування та інвестиції в стартап від вітчизняних фондів та бізнес-янголів, малий інтерес з боку українського бізнесу	Залучення до проекту іноземних інвестицій від зацікавлених компаній, переконання у необхідності у стартапі шляхом порівняльного тесту
3	Не рентабельність	Не рентабельність зачасту означає припинення існування стартап-проекту та марну трату грошей та зусиль	1. Перед переходом до посівної стадії (фаза перших витрат) впевнетися в рентабельності проекту завдяки тестуванням та порівнянням 2. Введення гнучкого принципу розробки дозволяє швидко підлаштуватися під потреби клієнтів, що постійно уточнюються

Таблиця 4.4 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Інновація на ринку напівпровідникових процесорів	Статус піонеру дає можливість швидко стати лідером у вибраному напрямку	Активно використовувати цей статус та не втрачати хватки, стрімкий розвиток до виходу на на рівень лідера у напрямку
2	Співпраця з лідерами ринку	Заповнення технологічного та досвідного пробілу, значне збільшення фінансування	Погодитися на взаємовигідних умовах зі збереженням стартапом самостійності
3	Розвинення ринку напівпровідникових процесорів України	Стати безумовним лідером в органному напрямку на ринку України	Подальший розвиток ринку напівпровідникових процесорів України

Таблиця 4.5 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - монополія	Перший на ринку прискорювачів та процесорів реляційний процесор	Сталий розвиток задля забезпечення лідируючих позицій
2. За рівнем конкурентної боротьби - світовий	При ведені бізнесу варто враховувати фактори та процеси, що протікають у світовій напівпровідниковій галузі	Бути завжди в тренді останніх розробок, новин, та кроків непрямих конкурентів
3. За галузевою ознакою - міжгалузева	Необхідність працювати як з комп'ютерною архітектурою, так і з математичними методами, алгоритмами, програмним забезпеченням, тощо	Необхідність на розділення діяльності за напрямками
4. Конкуренція за видами товарів: - товарно-родова	Бізнес-клієнтам важлива функціональність, за яку вони платять	Демонстрація переваги продукту над аналогами
5. За характером конкурентних переваг - цінова	Бізнес-клієнт завжди підраховує прибутки та ризики	Виготовляти оптимальний продукт для своєї ніші
6. За інтенсивністю - не марочна	Бізнес-клієнти оцінюють не по бренду, а по продукту	Виготовляти якісну продукцію

Таблиця 4.6 – Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
Складові аналізу	Немає	Для технологічних гігантів не так важко розробити аналог	Високі	Високі	Не значні

Продовження таблиці 4.6

Висновки:	Поки немає прямих конкурентів у стартапу є шанс вирватися вперед	Приблизний цикл розробки процесорів, як правило, займає 2-3 роки	Підвищений попит на виготовлення напівпровідникових процесорів під час дефіциту сильно впливає на стартап	Бізнес-клієнти є головними оцінювачами продукту, від їхніх потреб може змінитися весь ринок	Дана ніша поки не зайнята, тому розробка рішення, що буде краще за аналоги дозволить зайняти ринок

Таблиця 4.7 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Ціна товару	Обрана ціна у середньому сегменті є оптимальною для спроби продукту клієнтом на власному досвіді
2	Інформаційне забезпечення	Наявність для сопроцесору відлагодженної бібліотеки та API є безпосередньою перевагою, адже визначає насамперед комфорт у програмуванні
3	Застосування інноваційних технологій	Підтримка певних новаторських технологій дає перевагу компанії над іншими на короткий час
4	Ефективність виконання операцій	Ефективне виконання операцій дозволяє як збільшити обсяг інформації що оброблюється, так і зекономити електроенергію.
5	Естетичний вигляд	У світі сучасного комп’ютерного ринку естетичний вигляд продукту забезпечує позитивну динаміку росту продажів
6	Підтримка загально признаних стандартів	Підтримка загальних стандартів обміну інформацією, розмірів корпусу, та програмного забезпечення гарантує сумісність із іншими комп’ютерними електронними приладами

Таблиця 4.8 – Порівняльний аналіз сильних та слабких сторін «назва проекту»

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів						
			–3	–2	–1	0	+1	+2	+3
1	Ціна товару	20				*			
2	Інформаційне забезпечення	15							*
3	Застосування інноваційних технологій	18				*			
4	Ефективність виконання операцій	19		*					
5	Естетичний вигляд	16							*
6	Підтримка загально признаних стандартів	20				*			

Таблиця 4.9 – SWOT-аналіз стартап-проекту

Сильні сторони: • Унікальна пропозиція • Започаткування нової підніші • Реконфігурація • Зручне програмування • Забезпечення повноти	Слабкі сторони: • Запуск на слабкому локальному ринку • Відсутність досвіду власника стартапу в бізнесі • Необхідність верифікації технології
Можливості: • Співпраця з клієнтами • Співпраця з лідерами суміжних підніш • Отримати належне фінансування	Загрози: • Входження ло підніші мастодонтів з суміжних підніш • Несприйняття ринком проекту • Нерентабельність

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації з огляду на потенційні проекти конкурентів, що можуть бути виведені на ринок. Визначені альтернативи аналізуються з точки зору строків та ймовірності отримання ресурсів.

Таблиця 4.10 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Участь у тематичних науково-технічних конференціях, виставках, публікація у відповідних журналах	Висока	1 рік
2	Рекламна компанія, SMM	Від середньої до високої, в залежності від реалізації	6 місяців
3	Пряме звернення з пропозицією до клієнту	Від середньої до високої в залежності від реалізації	1 рік

В якості початкової альтернативи входу на ринок візьмемо участь у виставках та конференціях. Проте дане рішення є гнучким, та інші варіанти не відкидаються. Тому планується застосувати всі альтернативи у тій чи іншій пропорційності.

4.5 Розроблення ринкової стратегії проекту

Таблиця 4.11. – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Великі корпорації, що використовують сервери	Висока	Високий	Немає	Легка, з точки зору економіки
2	Великі компанії, що використовують для адміністрування та керування БД	Середня	Невідомо	Немає	Середня
3	Приватні особи, що використовують власні сервери з БД	Невідомо	Низький	Немає	Важка
Які цільові групи обрано: Великі корпорації, що використовують сервери					

Таблиця 4.12 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
	Участь у конференціях, виставках	Стратегія товарної спеціалізації	Розробка продукту, що ефективно вирішує конкретні потреби та задачі	Стратегія спеціалізації

Таблиця 4.13 – Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
	Так	Так	Ні	Стратегія лідера

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та до продукту, а також в залежності від обраної базової стратегії розвитку та стратегії конкурентної поведінки розробляється стратегія позиціонування, що полягає у формуванні ринкової позиції

(комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 4.14 – Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспро-можні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Ефективне виконання вузького кола задач	Стратегія спеціалі-зації	Ефективна робота з реляційними СУБД	Зосередженість саме на обробці БД дозволяє ефективно побудувати архітектурні засоби для роботи конкретно для БД
2	Швидкість обробки	Стратегія спеціалі-зації	Спеціальне апаратне прискорення операцій з реляційними БД	При розробці архітектури реляційного процесору головними аспектами було його максимальне прискорення зі збереженням повноти та зручності у програмуванні
3	Сумісність з наявними комп'ютерни ми компонентами	Стратегія спеціалі-зації	Прилад розроблений з оглядкою на підтримку сучасних комп'ютерів	При розробці приладу основна увага приділялась саме підтримці сучасних стандартів для комп'ютерної електроніки

Отже в якості стратегії охоплення ринку була обрана стратегія спеціалізації. А в якості стратегії конкурентної поведінки була обрана стратегія лідера, адже продукт стартап-проекту є першим у своїй стязі.

4.6 Розроблення маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 4.15 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Швидка та ефективна обробка БД	Спеціальний проскорювач для реляційних БД	Створення спеціального процесору для реляційних БД дозволяє підвищити ефективність СУБД

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар (остаточне визначення ціни відбувається під час фінансово-економічного аналізу проекту), яке передбачає аналіз ціни на товари-аналоги або товари субститути, а також аналіз рівня доходів цільової групи споживачів. Аналіз проводиться експертним методом.

Таблиця 4.16 – Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники, $	Рівень цін на товари-аналоги, $	Рівень доходів цільової групи споживачів, $	Верхня та нижня межі встановлення ціни на товар/послугу, $
1	399-≥799*	399-499	≥ 1 млдр	399-499

* від адаптації відео карт середнього класу до використання серверних CPU мінімального порогу входження

Дана ціна, як вже було сказано, відноситься до середнього сегменту суміжного ринку відеокарт. Така ціна дозволить клієнту відчути приріст швидкодії, не віддавши надто багато грошей (500$ є психологічним бар'єром).

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення:

- проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту);

- вибір та обґрунтування оптимальної глибини каналу збуту;
- вибір та обґрунтування виду посередників.

Таблиця 4.17 – Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
	Оптова	Доставка, гарантійна та програмна підтримка продукту	Нульового рівня	Прямий, через замовлення

Останньою складовою маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів.

Таблиця 4.18 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуютьс я цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Підвищення швидкодії серверів	Наукові конференції та виставки, e-mail, соціальні мережі, партнерство, рекомендації	Ефективне та швидке виконання операцій з реляційними БД, підтримка сучасних стандартів монтажу та зв’язку з іншими електронними компонентами комп’ютерів	Наше рішення дозволить пришвидшити роботу серверів, а також зекономити гроші на споживанні енергії за рахунок вузької направленості архітектури	Опис принципу роботи, теоретичне обґрунтування повноти даного процесору для типу даних реляції, демонстрація графіків швидкодії та споживання
2	Зменшення споживання енергії заради зменшення витрат на охолодження та енергію				

Отже, для даного стартап-проекту було обґрунтовано ціну в 499$, що відповідає середньому ціновому сегменту пристроїв периферії комп’ютера. В якості оптимальної системи збуту було обрано пряму, або через замовлення. А також було пояснено концепцію маркетингових комунікацій.

4.7 Стадії стартапу та фінансування

Кожен стартап проходить певні стадії перед тим, як стати повноцінним бізнесом. У класичному представленні розділяють 5 стадій стартапу, розглянемо ж їх.

Рисунок 4.8 — Діаграма стадій стартапу

На посівній стадії формується команда, робиться дослідження ринку та генерується ідея для стартапу. На стадії запуску здійснюється пошук потенційних інвесторів, а також створюється та тестується прототип продукту. Стадія зростання передбачає значного зростання залучення інвестицій, при цьому проводяться альфа та закритий бета тести продукту з метою виявлення помилок та його покращення; рекламування проекту. На стадій розширення вже проводиться відкритий бета-тест, залучується більше інвестицій для розширення продукту та його реклами. На стадії виходу стартап або викупається, або перетворюється на бізнес.

У нашому ж випадку було насильно-примусово прикручено стартап до магістерської дисертації, яка не мала бути стартапом та не може класифікуватись як стартап. Отже для адаптації дисертаційної роботи

необхідно змінити структуру стадій. Стадії даного стартапу представлені в таблиці 4.19.

Таблиця 4.19 — Стадії стартап проекту

Назва стадії	Передпосівна (pre-seed)	Запуск (startup)	Посівна стадія (seed)	Стадія зростання	Стадія розширення	Стадія виходу (exit)
План дій	Розробка та тестування реляційного процесору, виявленя та виправлення помилок	Пошук команди, розробка прототипу приладу	Розробка альфа-версії, рекламна компанія по залученню потенційних клієнітв, вивчення потреб клієнтів	Розробка MVP, закритий бета тест, активна реклама продукту, коригування продукту до потреб клієнтів, установка зв'язків з клієнтами	Відкритий бета тест MVP, активна рекламна компанія, остання корекція продукту до потреб клієнтів	Запуск масового випуску продукту, перехід від стартапу до бізнесу, росширення штату
Джерело доходу	Немає	3F, гранти, бізнес-янголи	Венчурні фонди, бізнес-янголи	Венчурні фонди	Венчурні фонди, фонди прямих інвестицій, банківські кредити	Стратегічні та портфельні інвестори
Основні витрати	Немає/мінімальні	Заробітні плати команді, замовлення прототипу	Реклама, замовлення партії для альфа-тесту, заробітні плати команді	Замовлення великої партії для бета-тесту, реклама	Замовлення ще більшої партії, реклама, обробка даних по бета-тесту	Договір про випуск продукту компанією виробником (OEM-контракт)
Приблизний бюджет, $	0	до 50 000	до 200 000	до 5 000 000	до 10 000 000	від 10 000 000

Як видно з таблиці 4.19, до оригінальної схеми стадій стартапу додалась передпосівна стадія. Її часто виділяють поза самим стартапом, оскільки це зазвичай стадія до активних дій. Проте в даному випадку передпосівна стадія слугує для розробки та перевірки реляційного процесору. Така стадія необхідна, щоб можна було впевнено почати стартап; а при випадку недопрацювання доробити несправні елементи та додати необхідного функціоналу для початку розробки плати для процесору. Таким

чином передпосівна стадія, в даному випадку, слугує запобіжником від очевидного провалу. Абревіатура 3F позначає family, friends, fools (родина, друзі, безглузді з англ.). Дана категорія інвесторів, зазвичай, є дуже зговорчивою, та іноді не просить повернути витрачені їхні гроші. Також саме вони готові проінвестувати ідею, котра знаходиться лише на стадії зародження.

Окрему уваго варто приділити тому, що у трьох стадіях до виходу проводяться тестування та опитування клієнтів. Даний принцип базується на тому факту, що стартапу варто підтримувати постійний контакт з потенційними покупцями. Таким чином корекція продукту “уточнює вірний шлях” для продукту стартапа. Це є мірою запобігання випуску нерентабельного продукту.

На перших активних стадіях стартапу в якості інвесторів виступають бізнес-янголи та венчурні фонди. Бізнес-янголом називається приватний венчурний інвестор, що надає підтримку та забезпечує фінансуванням стартапи на ранніх стадіях розвитку. Венчурним фондом є інститут спільного інвестування недиверсифікованого виду, закритого типу. Саме венчурне інвестування є ризикованим вкладенням коштів в обмін на певну частку в компанії, яку потім можна продати дорожче. Проте саме такі венчурні інвестори витягують стартапи з так званої “долини смерті”.

Рисунок 4.6 — графік фінансування стартапу від часу, “крива J”

Графіком на рисунку 4.6 економісти кличуть крив”ою J” через її форму. Область, що помічена червоним, називається долиною смерті. Це такий проміжок часу, коли стартапи сильно потребують фінансування, проте не отримують його і банкротують. За даними Oxford Review of Financial Services, у США лише 3% всіх стартапів отримують інвестиції. Очевидно що інші 97% через брак фінансування банкротують та вважаються “помершими”.

Отже, було виведено стадії стартап-проекту. Так, у даному проекті наявні 6 стадій: передпосівна, посівна, зростання, розширення та виходу. Передпосівна стадія введена як протидія запобіжник від провалу стартапа. На цій стадії реалізується робота з даної дисертації.

4.8 Подальше масштабування стартап проекту

Після запуску одного успішного продукту очевидно що в бізнесу виникає бажання подальшого масштабування. Дослідження питання масштабування вкрай важливе для перетворення стартапу на бізнес, що стало розвививається.

Розглянемо можливі напрямки масштабування в межах даного стартапу. На початку стартап продає периферійний пристрій, що пришвидшує операції над реляціями. Надалі напрямки пошуки напрямків для розвитку відштовхуються від первинного продукту. Так, наявно 2 напрямки для розвитку.

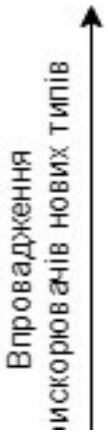


Рисунок 4.9 — координати можливих напрямків розвитку

Перший напрямок, сегментування продукції на різні цінові категорії, є класичним для ринку напівпровідникових. Так, усі виробники в цій галузі розбивають свою продукцію на декілька класів:

1. Надбюджетний;
2. Бюджетний;
3. Середній;
4. Середньо-високий;
5. Високий;
6. Надвисокий;

Дані класи є типовими для споживчого ринку, проте для бізнес ринку проміжні класи та надбюджетний відсутні чи не настільки ярко виразні. Відтак, сегментація відбувається по наступній схемі.

Таблиця 4.20 – Сегментація продукції на цінові категорії

Сегмент	Бюджетний	Середній	Високий
Ціна, $	≤ 500	≤ 1000	> 1000

Виходить, що перший продукт, згідно даної класифікації знаходиться у бюджетному сегменті. Типовим параметром, який відрізняється між сегментами, є потужність процесору. Тому процесори для вищих сегментів матимуть більший обсяг ядер та обчислювальної потужності.

Другим напрямком є впровадження нової продукції, а саме прискорювачів інших типів даних. До прискорювачів, що наразі найбільш стрімко розвиваються, відносять нейронні процесори. Як вже було зазначено з [48], даний напрямок вже є висококонкурентним (від 10-ти потужних конкурентів), тому для адекватного входження на цей ринок необхідна чимала підготовка та наявність значущої переваги. Втім, початок саме у новітній напрямок надає певну “подушку безпеки”, і таким чином бізнес не так сильно ризикує, як стартап.

Для кожного напрямку масштабування є найбільш придатний час. Так, наприклад, більше освоєння ринку, на який стартап вийшов, є більш пріорітетним, аніж вхід до нового. Враховуючи вищесказане, побудуємо графіки залежності пріорітетності напрямків від часу.

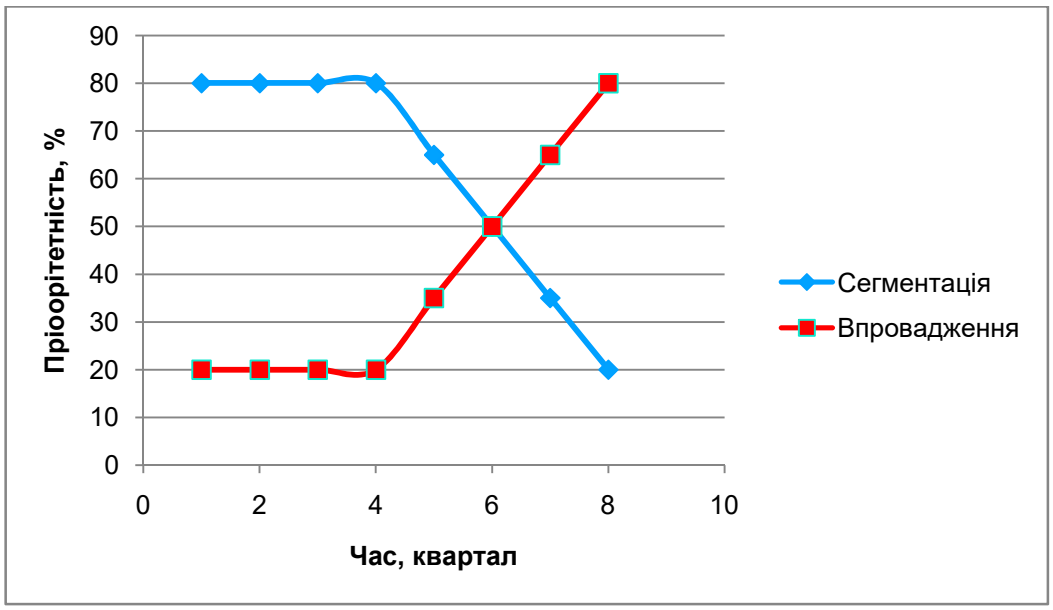


Рисунок 4.11 — графік пріорітетності сегментації та впровадження нових рішень від часу

Так, за рисунком 4.11 а, спочатку на сегментацію продукції виділяється 80% пріорітету. Після його достатнього освоєння (4 квартал) рівень пріорітетності спадає. При цьому рівень пріорітетності впровадження нових рішень зростає з базових 20%. І поступово рівні пріорітетності міняються місцями. Відтак, на освоєння нових рішень відводиться 80% пріорітету, а на сегментацію 20%. Принцип розподілу пріорітетів основується на принципі Парето. А точку, де пріорітети рівні між собою (6 квартал), ми назвали точкою початку освоєння нового ринку.

Отже, отримавши план подальшого розвитку та масштабування, стартап може перейти на стадію пост-стартапу.

4.9 Висновок

Отже, в даному розділі перевірялась та виводилась можливість реалізації дисертаційної роботи у стартап-проект. Так, було визначено основних клієнтів, сформовано концепцію продукту, сформовано бізнес-план, перевірено конкурентоспроможність проекту на ринку, опрацьовані

основні канали зв’язку, розроблено стадії стартапу та їхнє відповідне фінансування, а також була розроблена стратегія для етапу пост-стартапу.

Головним клієнтом стартапу є бізнес, що активно використовує сервери та БД. Вони найбільше потребують пришвидшення роботи з БД.

Продукт являє собою периферійну плату з роз’ємом PCI-E x16 з підтримкою стандартів 3.0 та 4.0. В якості системи охолодження було обрано активну повітряну, оскільки вона є оптимальною в плані відношення ціни та якості. Продукт накривається захистним кожухом по захистним та естетичним причинам. Найвищий рівень архітектури процесору є багатоядерним, з підтримкою зовнішньої пам’яті типу GDDR6. Працює він на стадії MVP під ОС GNU/Linux з мовами програмування C та SQL.

Було створено Lean Canvas стартапу, що фактично є паспортом його бізнес-моделі. Також було уточнено, що стартап працює з виробником компонентів та апаратури за OEM-контрактом.

Було оцінено ринок стартапу, порівняно проект з непрямими конкурентами, та зроблено SWOT-аналіз. В якості початкової альтернативи входу на ринок візьмемо участь у виставках та конференціях. Проте дане рішення є гнучким. Планується застосувати всі альтернативи у тій чи іншій пропорційності.

Було розроблено ринкову стратегію стартап-проекту. Було обґрунтовано ціну в 499$, що відповідає середньому ціновому сегменту пристроїв периферії комп’ютера. В якості оптимальної системи збуту було обрано пряму, або через замовлення. А також було пояснено концепцію маркетингових комунікацій.

Виведено основні стадії стартап-проекту та їхнє фінансування. До класичної схеми стадій було додано передпосівну стадію, що є запобіжником початку провальної ідеї.

Було розроблено стратегію для стадії пост-стартапу. Так, спочатку відбувається сегментація продукції по ціновим діапазонам, а потім освоюються нові напрямки ринку.

В результаті, даний розділ показав, що даний стартап проект може бути рентабельним. Гнучка розробка та введення бізнесу дозволяють пристосуватись до більшості неочікуваних змін. Проте, автор даного проекту сильно не любить стартапи, а тому даний стартап-проект загинув ще на передпосівній стадії.

Висновок до магістерської дисертації

Отже, під час роботи над магістерською дисертацією були зроблені наступні кроки та вирішені наступні задачі на магістерську дисертацію.

Розділ 1. В даному розділі були розглянуті та порівнянні універсальні та спеціалізовані комп'ютерні архітектури. Було розглянуто як класичні рішення, так і сучасні. Серед класичних універсальних були RISC у вигляді процесору MIPS та CISC у вигляді процесору i8086. А сучасні рішення були представлені мікроархітектурою Golden Cove. Спеціалізовані архітектури були представленими нейронним процесором від компанії Google TPU та типовою для обчислювачів структурних даних архітектурою SM. Було виділено основні принципи та підходи між ними, та вибрано ті, що застосуються для реляційного процесору.

Розділ 2. В даному розділі було виведено алгебраїчну характеристику реляційних чр-функцій та чр-предикатів для ППА методом ізоморфного відображення на векторну та матричну алгебри. Було доведено потужність даної алгебраїчної характеристики шлях моделювання операцій з алгебри Кодда за допомогою базисних операцій виведеної алгебри. Тип реляціх та її алгебраїчну характеристику було уточнено для іменної моделі даних, а також була представлена можливість апаратної підтримки складних типів даних методом нормалізації даних та таблиць.

Розділ 3. В даному розділі було описано реляційний процесор, що базується на матричному обичслювачі. Так, були модифіковані ядра керування та виконання, пам'яті імен та реляцій. Керуюче ядро зазнало не значних модифікацій. Виконавче ядро було майже повністю реструктуровано заміною блоку підтримки ізоморфізму на блок обчислення адрес та блок керування операціями. Було введено механізм стекування даних керуючого ядра. Було модифіковано пам'ять реляцій для можливості по елементного кроку адресації при векторному читанні. Блок обчислення даних тепер підтримує широкий спектр накопичувальних операцій. Блок обчислення

адрес тепер використовує на кожну реляцію по парі лічильників з лічильника атрибутів та лічильника строк; а також був модифікований під читання конкретного діапазону атрибутів. Блок керування операціями базується на скінченних автоматів, що в даній роботі були представленими у вигляді функціональних графів. Для них було доказано підтримку ППА в необхідному обсязі та виведено 9 з 18 базових операцій. Також було представлено зручний перехід від зрозумілих людині записів до бітових значень на керуючих шинах.

Розділ 4. В даному розділі перевірялась та виводилась можливість реалізації дисертаційної роботи у стартап-проект. Так, було визначено основних клієнтів у вигляді бізнес сегменту, що активно використовує БД. Сформовано концепцію продукту у вигляді периферійної карти для комп’ютера. Сформовано бізнес-план, перевірено конкурентоспроможність проекту на ринку, та опрацьовані основні канали зв’язку. Розроблено стадії стартапу та їхнє відповідне фінансування з впровадженням додаткової стадії для запобігання випуску на ринок нерентабельного продукту. А також була розроблена стратегія для стадії пост-стартапу, де спочатку йде сегментація продукції по ціновим діапазонам, а потім освоюються нові напрямки ринку. В результаті, даний розділ показав, що даний стартап проект може бути рентабельним.

Отже, враховуючи вищезазначене, можна стверджувати, що всі завдання на магістерську дисертацію були виконані!

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Кудлай С. В. Прискорювач матричних операцій. Київ: КПІ, 2021

2. Редько В.Н. Основания композиционного программирования // Программирование. – 1979. - № 3. С.3-13.

3. Басараб И.А., Никитченко Н.С., Редько В.Н. Композиционные базы данных// К.:Либідь, 1992.-192 с.

4. Буй Д.Б., Редько И.В.Примитивные программные алгебры вычислимых функций //Кибернетика.–1987.– №3.– С.68-74

5. Мальцев А.И. Алгоритмы и рекурсивные функции//М.:Наука.–1965.– 391 с.

6. Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л. Алгебра. Языки. Программ ирование Изд. 3, перераб. и доп. 1989. 376 с. ISBN 5-12-000499-7

7. Буй Д.Б., Редько В.Н.Примитивные программные алгебры. I, II//Кибернетика.–1984.– №5.–С.1-7;– 1985.–№1.–С.28-33

8. SWEBOK Pierre Bourque, Robert Dupuis; executive editors, Alain Abran, James W. Moore, eds. (2004).

9. Ершов А.П. Вычислимость в произвольных областях и базисах // Семиотика и информатика.–1982.–Вып. 19.–С.3-58

10. Горєлов А.В., Редько І.В., Яганов П.О. Композиційні засади програмістської діяльності//Вісник КНУТД.–№3.–2015.–С.3-12

11. Редько І.В., Снігур Н.М. Примітивна програмна алгебра обчислюваних функцій над графами//Наукові вісті НТУУ “КПІ”. - 2011. - № 4. - С. 75– 80

12. Т. Л. Захарченко, І. В. Редько, Д. І. Редько та П. О. Яганов, «Примітивна програмна алгебра обчислюваних функцій над записами,» Наукові вісті НТУУ «КПІ», № 2, pp. 29-40, 2015.

13. D. I. Redko, I. V. Redko, P. O. Yahanov and T. L. Zakharchenko, "Compositional basis in programmer activity," Системні дослідження та інформаційні технології, no. 4, pp. 83-96, 2015

14. 13. Т. Л. Захарченко та І. В. Редько, «Проблема повноти в класі функцій над записами, які зберігають денотати,» Наукові вісті НТУУ «КПІ», № 5, pp. 23-30, 2015

15. І. В. Редько, Д. І. Редько та Т. Л. Захарченко, Концептологічні основи проектування, Київ: Компринт, 2016

16. Редько В.Н. Семантические структуры программ. // Программирование. – 1981. - № 1. С.3-19.

17. Дэвид М. Харрис, Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера, второе издание. 2013. ISBN 978-0-12-394424-5

18. A. M. TURING. Intelligent Machinery, A Heretical Theory. *Philosophia Mathematica*, Volume 4, Issue 3, September 1996, Pages 256–260

19. A. M. TURING. On Computable Numbers, with an Application to the Entscheidungsproblem. 1937. https://doi.org/10.1112/plms/s2-42.1.230

20. David A. Patterson, John L. Hennessy. Computer Organization and Design – The Hardware / Software Interface, 4th.Edition. Morgan Kaufmann, Elsevier, 2009.

21. Intel. 8086 Family User's Manual. Mnemonics, Intel Corp., October 1979.

22. clamchowder. Popping the Hood on Golden Cove. Chips and cheese, December 2, 2021 : веб-сайт. URL: https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove (дата звернення 22.12.2022)

23. Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson. A Domain-Specific Architecture for Deep Neural Networks. Communications of the ACM, September 2018, Vol. 61 No. 9, Pages 50-59 10.1145/3154484

24. Scott Rixner. Stream Processor Architecture. Rice University, 2002

25. Intel Core 12th Gen Alder Lake Preview : Веб-сайт. URL: https://www.techpowerup.com/review/intel-core-12th-gen-alder-lake-preview/3.html (дата звернення 22.12.2022)

26. Intel Architecture Day 2021 Golden Cove Front End : Веб-сайт. URL: https://www.servethehome.com/intel-golden-cove-performance-x86-core-detailed/intel-architecture-day-2021-golden-cove-front-end/ (дата звернення 22.12.2022)

27. Sunny Cove, Willow Cove и Golden Cove - новые архитектуры CPU : Веб-сайт. URL: https://www.hardwareluxx.ru/index.php/news/hardware/prozessoren/46104-sunny-cove-willow-cove-golden-cove-cpu.html (дата звернення 22.12.2022)

28. NVIDIA. NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V2.1. NVIDIA Corporation, 2021

29. E. F. CODD. A Relational Model of Data for Large Shared Data Banks. IBM Research Laboratory, San Jose, California, Volume 13, Number 6, June 1970.

30. C.J.Date. An Introduction to Database Systems, 8-th edition. Pearson Education, Inc., 2004. ISBN 0-321-19784-4

31. С. Кудлай, Н. Бондаренко, В. Бондаренко. Побудова та верифікація моделі цифрового еквалайзера. // Вісник Хмельницького національного університету. – 2022 – № 5 (313). – С. 178-184. DOI: 10.31891/2307-5732-2022-313-5-178-184

32. Joseph Yiu. The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, third edition. 2014. 675 c. ISBN–13: 978-0-12-408082-9

33. Короткий Є. В. Методичні матеріали з предмету Схемотехніка, Київ : КПІ, 2018.

34. Варфоломеєв А. Ю. Комбінаційні пристрої. Методичні матеріали з курсу Функціонально-логічне проектування, Київ : КПІ, 2018.

35. Глушков В.М. Синтез цифровых автоматов.−М.: Физматгиз, 1962. − 476 с.

36. Гавриш О. Я., Бояринова К. О., Кравченко М. О., Копішинська К. О. Дисципліна МЕНЕДЖМЕНТ СТАРТАП-ПРОЄКТІВ, опорний конспект лекцій, Київ : КПІ, 2020

37. DB-Engines Ranking : Веб-сайт : URL : https://db-engines.com/en/ranking (дата звернення 23.12.22)

38. Ranking of the most popular relational database management systems worldwide, as of January 2022 : Веб-сайт : URL : https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/ (дата звернення 23.12.22)

39. ASUS WS C621E SAGE Motherboard specifications : Веб-сайт : URL : https://www.asus.com/ua-ua/Commercial-Servers-Workstations/WS-C621E-SAGE/ (дата звернення 23.12.22)

40. ASUS PRIME H510M-A Motherboard specifications : Веб-сайт : URL : https://www.asus.com/ua-ua/motherboards-components/motherboards/prime/prime-h510m-a/ (дата звернення 23.12.22)
41. Tung Lee. Where Does PCIe Cables Go – A Guide To Powering Graphics Cards. : Веб-сайт : URL : https://greatpcreview.com/guides/where-does-pcie-cables-go/ (дата звернення 23.12.22)
42. MSI GT1030 Low Profile GPU specifications : Веб-сайт : URL : https://www.msi.com/Graphics-Card/GeForce-GT-1030-2G-LP/Specification (дата звернення 23.12.22)
43. MSI GeForce RTX™ 3080 GAMING X TRIO 10G Specifications : Веб-сайт : URL : https://ua.msi.com/Graphics-Card/GeForce-RTX-3080-GAMING-X-TRIO-10G (дата звернення 23.12.22)
44. AORUS GeForce RTX™ 3080 XTREME WATERFORCE WB 10G Specifications : Веб-сайт : https://www.gigabyte.com/Graphics-Card/GV-N3080AORUSX-WB-10GD-rev-10 (дата звернення 23.12.22)
45. Joanne Chiao. Localization of Chip Manufacturing Rising. Taiwan to Control 48% of Global Foundry Capacity in 2022. TrendForce, 25 April 2022.
46. Павел Котов. AMD захватила максимальную долю рынка x86-процессоров за всю историю — помог рост поставок серверных чипов. 3DNews, 10.02.2022
47. Advanced Micro Devices. Condensed consolidated statements of operations, Q3 2022. Advanced Micro Devices, inc, dec 2022.
48. Top AI Processors Startups : Веб-сайт : https://tracxn.com/d/trending-themes/Startups-in-AI-Processors (дата звернення 23.12.22).