НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.457

«До захисту допущено»
Завідувач кафедри
_____ І.А. Дичка
«___»_____2018 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Програмна модель та метод підвищення ефективності функціонування відмовостійкої потокової системи»

Виконав:
студент VI курсу, групи КП-71мп
Лукашов Борис Олегович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н.,
Жабіна В.В. _____

Рецензент:
Доцент кафедри ОТ факультета ФІОТ, к.т.н., доцент,
Верба О.А. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Граф алгоритм – орієнтовний граф, що складається з вершин, відповідних до операцій алгоритму та направлених дуг, що відповідають передачі даних.

Крупнозернистий граф алгоритм – граф алгоритм в якому кожна вершина являє собою набір інших операцій та розкривається у граф.

Дрібнозернистий граф алгоритм – граф алгоритм в якому кожна вершина виступає окремою одиничною операцією з операндами.

СПД – системи, в основі якої лежить керування потоками даних, наборів сигналів з різних джерел, з метою направлення кожного набору сигналів на відповідні блоки системи.

Паралелізм – процес направлення та подальшого виконання різних операцій з операндами одночасно на декілька обчислювальних блоків.

ПЛІС – програмована логічна інтегральна схема.

WPF – Windows Presentation Foundation графічна (презентаційна) підсистема, яка починаючи з .NET Framework 3.0 в складі цієї платформи.

.NET Framework – програмна технологія, запропонована фірмою Microsoft як платформа для створення як звичайних програм, так і веб-застосунків.

MVVM – Model-View-ViewModel це шаблон проектування, що застосовується під час проектування архітектури застосунків (додатків).

ВАД – вільний адресний доступ, схема адресації пам'яті комп'ютера, при якій пам'ять для запущеної програми реалізується однорідним масивом.

Quartus II – це програмований логічний дизайн пристрій програмного забезпечення вироблений компанією Altera. Quartus II дозволяє аналізувати і синтезувати HDL конструкції, що дозволяє розробнику складати свої проекти.

Xlinix – середа розробки, що дозволяє створювати власні архітектури та тестувати їх швидкодію, надійність тощо, вироблена американською компанією Xlinix, реалізовує та використовую граматику HDL.

HDL – Мова опису апаратури спеціалізована формальна комп'ютерна мова, що використовується для проектування структури, дизайну та роботи електронної мікросхеми та її моделювання.

ОС – в контексті даної роботи слід вважати, як обчислювальна система.

ОМ – обчислювальний модуль, процесор.

СФК – система формування команд.

ВСТУП

В наш час, в світі постійно зростає кількість інформації, яку необхідно зберігати та оброблювати, процеси обробки великих об’ємів даних дуже ресурсо-витратні та займають велику кількість часу тому подібні рішення відбуваються паралельно тобто обробка інформаційних даних в декілька потоків, для цього було створено системи, що керують саме потоками даних, для контролю роботи системи, дані так звані СПД (система керування потоками даних) відповідають за те, щоб дані як умога оптимальніше та швидше оброблювались.

Для систем керування технологічними процесами і об'єктами в важливими характеристиками є швидкодія і надійність. У подібних системах часто виникає необхідність реалізації алгоритмів з дрібнозернистою структурою, які являють собою графи. До них відносяться, наприклад, алгоритми інтерполяції функцій, розрахунку траєкторії об'єктів, перетворення координат в багатовимірному просторі тощо. Як зазначалося вище, досягти прискорення обробки подібних алгоритмів можна шляхом розпаралелюванням обчислень на рівні операцій. Один з варіантів оптимізації роботи подібних систем – це використання так званого статичного розпаралелювання, воно є оптимальним, якщо система створюється лише під одну конкретну задачу і всі шляхи рішення відомі програмісту заздалегідь, в такому випадку він може описати сценарій роботи системи та використати статичне розпаралелювання на рівні коду при реалізації. Проте у випадку якщо система може виконувати багато задач, або є дуже великою в такому випадку реалізація статичного розпаралелювання є нераціональною, для цього оптимально використовувати архітектурне рішення та роботу з граф алгоритмом задачі і використовувати ресурси системи по мірі надходження задач, динамічне ропаралелювання.

Отже відповідно до того, шо оптимальним є використання динамічного розпаралелювання, постає питання надійності, швидкодії та тестування подібного підходу до реалізації багатопоточності.

В даній роботі розроблено та досліджено метод підвищення ефективності роботи подібних систем, завдяки використанню ВАД, вільного адресного простору, приведено відповідні обрахунки, а також підвищення надійності роботи системи при відмові якихось обчислювальних блоків системи, реалізовано динамічну реконфігурацію. Відповідно до цього було розроблено програмними засобами прогармне забезпечення, що дозволяю емулювати розроблену системи, що працює на базі досліджуваного методу, завдяки емулятору можна впевнитись, що досліджуваний метод є оптимальним при роботі з багатопроцесорними системами, та збільшує показники надійності системи.

Слід зазначити, що розроблений емулятор є кросплатформеним, а також дозволяє емулювати роботу системи під яку був розроблений і не вимагає додаткових знань у сфері проектування, на відміну від існуючих важких аналогів з власною мовою. Також для подачі даних для формування графу було розроблено просту власну граматику, завдяки якій можна сформувати для системи задачу для рішення.

Отже реалізація емулятору відмовостійкої системи керування потоками даних на базі вільного адресного доступу (ВАД) є актуальною задачею, адже дозволяю просто та швидко без додаткових витрат часу протестувати та проаналізувати роботу системи з заданими параметрами та заданою архітектурою.

1. ПРОБЛЕМА ВІДМОВОСТІЙКОСТІ , ІСНУЮЧІ ЗАСОБИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМ КЕРУВАННЯ ПОТОКАМИ ДАНИХ ТА ЗАСОБИ ПРОЕКТУВАННЯ

1.1. Принцип роботи СПД

В основі системи керування потоками даних, лежить архітектурне рішення, яке дозволяє реалізувати обчислення згідно дрбінозернистому граф алгоритму. Сама по собі СПД складається за набору логічних елементів, запам’ятовуючих пристроїв, пристроїв вводу, пристроїв виводу тощо. Процес роботи – це процес спілкування між даними елементами системи, зо дозволяє системі виконувати поставлені задачі згідно граф алгоритму, що прийшов на вхід, паралельно на доступних процесорах. Даний вид багатопоточності відноситься до динамічного розпаралелвюання та не вимагає втручання програміста чи користувача, завдяки реалі званій архітектурі та програмним налаштуванням система сама розпоряджаться набором виданих їй ресурсів на протягом усього часу роботи[1].

Граф, що подається на вхід, описує поведінку роботи, саме в графі описана послідовність дій, алгоритми обходу графу дозволяють системі розуміти, які з операцій можуть бути виконані на даному етапу роботи системи[2].

Відмічаються наступні переваги СПД:

- досягнення продуктивності спеціалізованих машин при збереженні можливостей універсальних машин за рахунок організації обчислень адекватно структурі алгоритму розв'язуваної задачі;
- спрощення складання і верифікації паралельних програм, а також можливість широкого застосування функціональних мов

програмування, що підвищує продуктивність праці програміста;

- автоматичне розпаралелювання алгоритмів в динамічному режимі, що потенційно може підвищити продуктивність;
- можливість широкого використання каналів зв’язку в зв'язку з регулярністю структури даних графу;
- модульність і гнучкість схем зв'язку дозволяють ефективно нарощувати обчислювальну систему (ОС) для збільшення продуктивності;
- підвищена надійність, що дозволяє відключати несправні ОУ з продовженням обчислювального процесу без зміни алгоритму;
- орієнтація на роботу в реальному масштабі часу, так як використання потокового принципу управління забезпечує швидку реакцію на появу нових даних.

Серед недоліків СПД виділяють наступні:

- велика довжина команд, але його вплив стає менш значним з розвитком технології канальної передачі даних;
- великий обсяг пересилань;
- необхідність використання асоціативної пам'яті (відповідно дану проблему і вирішує метод розроблений в даній роботі).

Майже усі сучасні машини є наслідками фон-неймановської архітектури, є декілька важливих пунктів, що запровадив даний паттерн реалізації обчислювальних машин:

- Машина сприймає задачу, як набір команд, що дозволяє відійти від концепції, так званого «калькулятору», коли машина виконую напряму задачу, а не набір команд. Це рішення дозволяє використовувати обчислювальну машину для рішення різних задач, описаної для неї набором команд.

- Використання спільного адресного простору, для збереження операндів та операцій.

Обчислювальні системи, що формують системи керування потоками даних не є виключенням, працюють на основі вхідних даних. Програма для СПД представляє собою список акторів(керуючих слів) та даних з якими вони оперують, дані позиції розставляються у відповідному порядку у графі по ярусному принципу. Акторам відповідають вершини графу, а дуги – передачі даних у системі. Відповідно – це формує граф алгоритм задачі. Граф алгоритм – кінцевий стан орієнтованого графу з заданим шляхом обчислень, де на вході стартові дані на виході результат роботи усіх акторів, граф алгоритм при обході є синхронним, що означає, що доки не виконались верхній ярус, нижній не може почати виконання, про асинхронний на одному ярусі, непов'язані актори можуть виконуватись паралельно. Приклад дрібнозернистого граф алгоритму зображено на рисунку 1.1 [2].

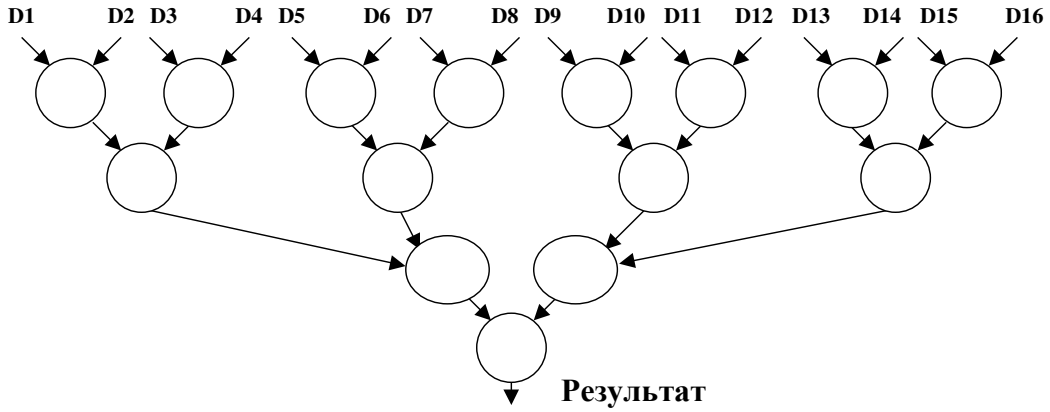


Рис.1.1. Приклад дрібнозернистого граф алгоритму

Як видно з рисунку 1.1, кожна вершина оброблює лише 2 операнди та передає дані результату далі по дугах графу, це означає, що граф дрібнозернистий, адже кожна вершина має в собі 2 операнди та 1 операцію

на виконання, у випадку крупнозернистого граф алгоритму вершина розкривається ще у відповідний граф, за таким самим принципом, аналог графу з рисунку 1.1 тільки крупнозернистого формату зображено на рисунку 1.2.

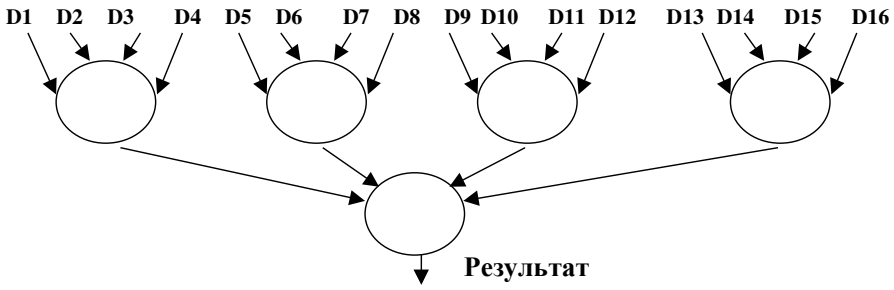


Рис.1.2. Приклад крупнозернистого граф алгоритму

Як видно з рисунку 1.2, тепер кожна вершина графу приймає по 4 параметри на відміну від попереднього прикладу з рисунку 1.1, проте це не означає, що актор тепер став працювати з чотирьома операндами, даний графу розкривається у свій дрібнозернистий аналог, в цьому випадку лише зменшується кількість переходів даних між вершинами, проте зменшується степінь паралелізму, що може сповільнити роботу системи в цілому [3].

Відповідно до фон-неймівської архітектури системи, мають по факту два основі елементи, це СФК(система формування команд) та обчислювальні модулі системи(процесори). Основним модулем СФК є сітка пам’яті, є декілька варіантів адресації:

- Вільний адресний простір – найпростіший в реалізації та найшвидший, завжди вистроєний у ПЛІС;
- Асоціативна пам'ять – більш гнучка та багатофункціональна ніж підхід вільної адресації, проте працює значно повільніше;
- Буферна пам'ять – покращена система адресації вільного простору, об’єднує кластери паняті у окремі масиви для

збереження відповідних даних, наприклад розділення операцій та операндів.

Більшість сучасних подібних систем використовують асоціативну пам'ять, адже вона дозволяє оперативно реконфігурувати системи, адже зберігає усі дані, що циркулюють у системі за весь час роботи. Загальний принцип роботи та основні блоки показані на рисунку 1.3 [4].

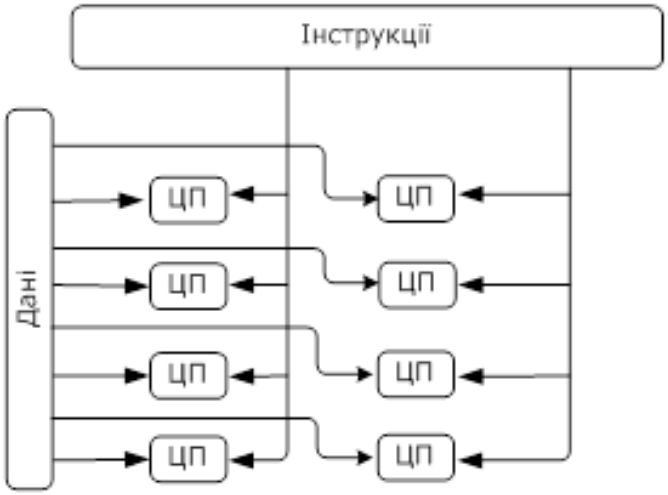


Рис.1.3. Загальна схема роботи СПД

На рисунку 1.3 изображено загальний підхід до реалізації сучасних СПД, маємо набор обчислювальних блоків, та набір інструкцій на відповідній мові, яка призначена для даної системи, а також дані, на входи та виходи, дана схема описує лише входи та виходи, відображає елементи, що оперують у роботі системи. На рисунку 1.4 наведено структуру простішої обчислювальної системи.

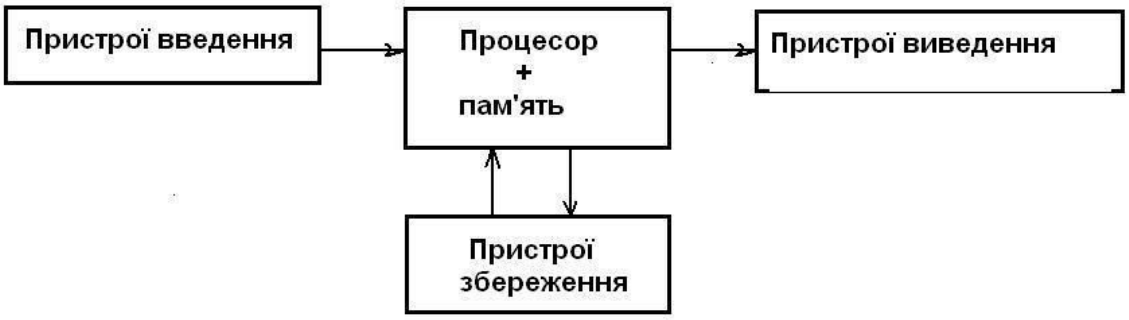


Рис.1.4. Найпростіша потокова система

Як видно з рисунку 1.4, найпростіша потокова система включає в себе пристрої введення виведення, обчислювальні блоки та блоки пам’яті, пристрої збереження.

1.2. Відмовостійкість СПД та існуючі методи підвищення ефективності роботи СПД

Відмовостійкість – це можливість системи продовжувати роботу при поступовому відключенні обчислювальних модулів, без втрати по продуктивності. У більшості систем відмовстійкість реалізована шляхом динамічної реконфігурації системи після віднови обчислювального модуля або відновлення його роботи [5]. Динамічна реконфігурація - це перерозподіл операції, що виконувалась або мала виконуватись на заданому процесорі, проте даний модуль вийшов із строю, тому дана команда має відправлятися на наступний вільний модуль, процес переналаштування системи на роботу без модуля , що вийшов зі строю та неврахування його у наступних діях системи, або навпаки при додавання блоку до системи після його ремонту, система має враховувати його наявність при розподілі задач.

Даний показник є дуже важливим при конфігурації системи, він має бути на достатньо високому рівні. Даний показник є адитивним та рахується за формулою 1.1.

$$A = \sum \lambda * N_i \qquad (1.1)$$

Де: A – загальний показник відмовостійкості;

λ – відсоток відмовостійкості окремого модуля;

N – кількість модулів даного типу.

В різних системах процес реконфігурації описано по різному, але в будь якому випадку, операція, що не виконалась через несправність процесору, має буде збережена та направлена пріоритетно на виконання на вільному процесорі [6].

У потокових системах підготовка завдання здійснюється без урахування кількості обчислювальних модулів. Це створює передумови в разі відмови ОМ продовжувати обчислення до тих пір, поки в системі не залишиться хоча б один працездатний ОМ. З огляду на, що методи забезпечення відмовостійкості пам'яті розроблені досить добре, основна увага приділяється апаратних засобів автоматичної реконфігурації системи при відмові ОМ. При цьому основною проблемою є відновлення інформації про команду, загубленої в зв'язку з відмовою ОМ [6].

Отже, як сказано вище формування відмовостійкості відбувається відповідно до архітектури побудованої завчасно для системи. Так, як в даній роботі проектування системи відбувається з вільним адресним простором тому і аналоги для підвищення відмовостійкості будемо розглядати на прикладі систем з вільним адресним простором. В архітектурі наведеній на рисунку 1.5, використовується вільний адресний простір, та завдяки особливості побудови надає збільшенні ефективності роботи системи.

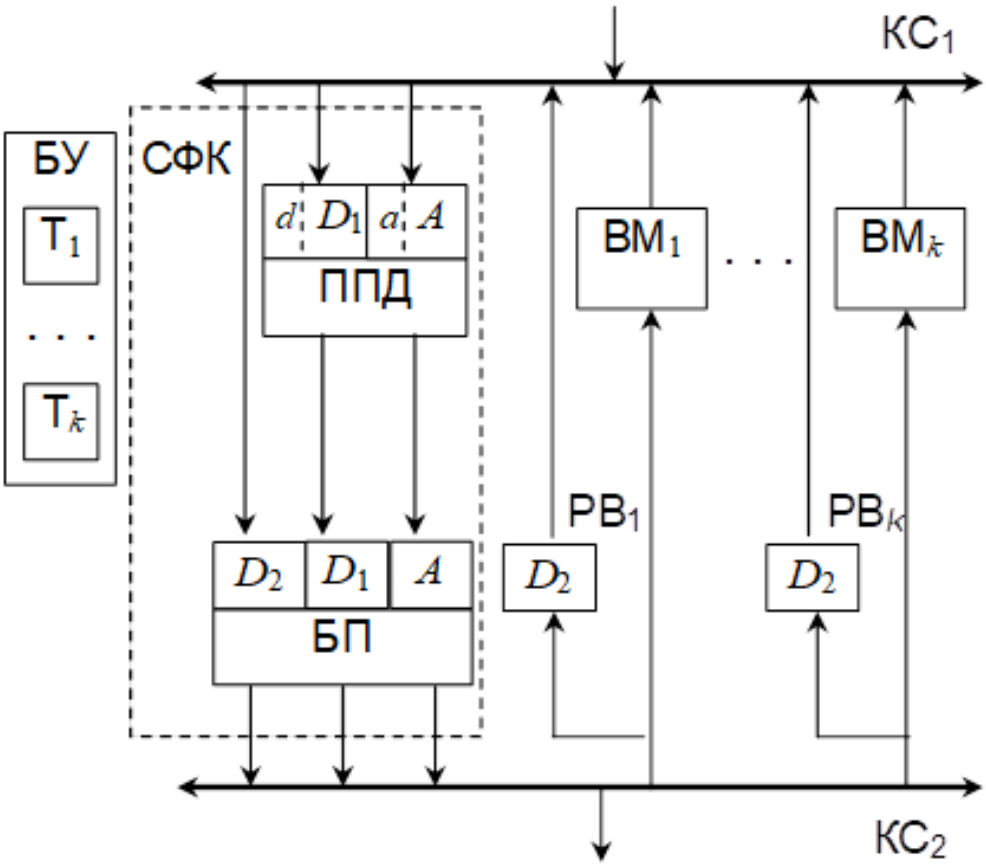


Рис. 1.5. Архітектура з вільним адресним доступом першого варіанту

Розглянемо особливості активації команд для такої системи. Команда починає формуватися в комірках адресного простору, два розряду яких (d і a) є ознаками, що показують наявність в ВАД одного операнда D1 і актора. Запис зазначених об'єктів для і-го команди виконується за адресою I. При надходженні з КС1 другого операнда D2 для даної команди, він записується безпосередньо в відповідними розряди регістра буферної пам'яті (БП) типу FIFO, куди одночасно з ВАД переписуються A і D1.

Таким чином, в БП формується готова для виконання команда, яка потім через КС2 передається у вільний ВМ. При цьому вміст відповідної комірки ВАД зберігається для забезпечення повторного формування команди в разі відмови ОМ. Одночасно з надходженням команди в ОМ її

частина, яка не з-яке береже в ВАД, записується в регістр тимчасового зберігання на вхідний шині відповідного обчислювача. При успішному виконанні операції, тобто коли час її виконання не перевищило ліміт часу виконання команди в ОМ встановлений відповідним таймером Т блоку управління БУ, результат операції надходить через КС1 в ВАД за адресою, а вміст відповідної комірки ВАД за адресою обнуляється.

Якщо при виконанні операції відбулася відмова ОМ, тобто ліміт часу виконання операції, заданий відповідним Т, був перевищений, ОМ вважається несправним і блокується (відключається від КС1 і КС2), а значення з РВ знову передається в СФК. Оскільки значення A і D загублені були (відповідна комірка ВАД була змінена), то команда повторно записується в БП, що дає можливість реалізувати наступну успішну спробу виконання команди в справному ОМ.

До недоліків слід віднести необхідність зберігання компонентів команди в осередках ВАД до повного завершення виконання даної команди, в тому числі, з урахуванням часу реконфігурації системи при відмові ОМ і повторного виконання команди. Це уповільнює обчислення, якщо відповідну комірку пам'яті необхідно використовувати для інших команд, наприклад, для чергової ітерації або черговий реалізації алгоритму в конвеєрному режимі [7, 8 , 9].

Розглянемо другий варіант системи з вільним адресним доступом, архітектура зображена на рисунку 1.6.

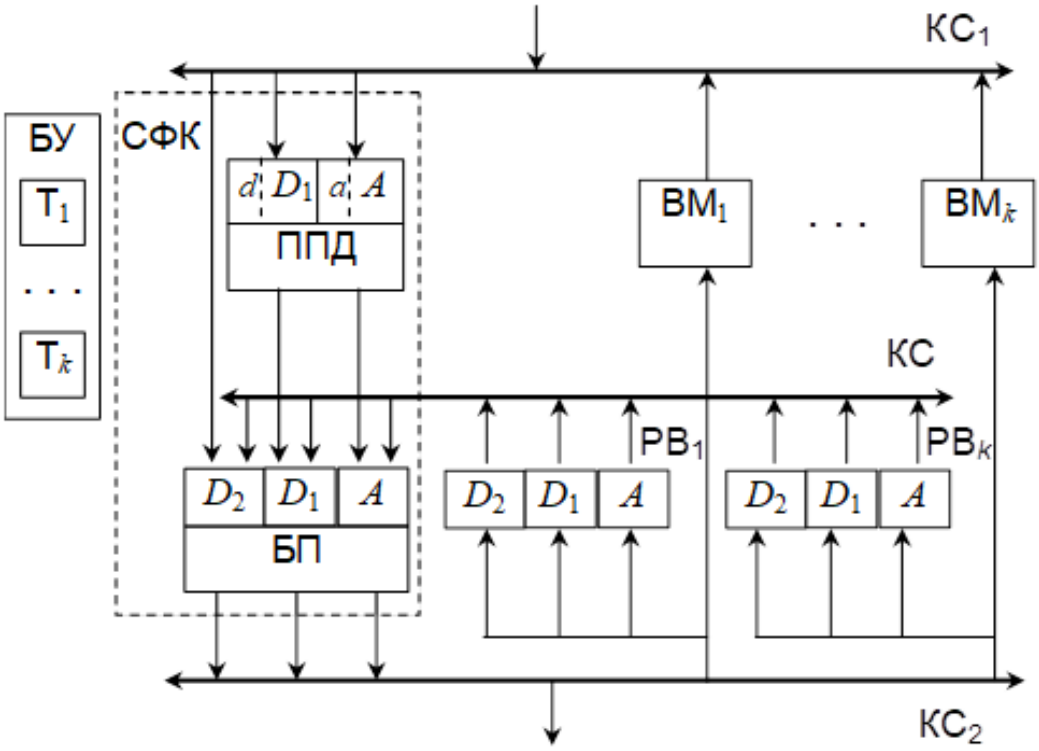


Рис. 1.6. Архітектура з вільним адресним доступом другого варіанту

Для усунення зазначеного недоліку системи з ВАД першого варіанту пропонується в РВ записувати не тільки операнд, а всю команду. При цьому принцип формування команди залишається незмінним. Після запису даних в БП звільняється місце в ВАД для запису таких даних. За аналогією з попереднім варіантом, при відмові ОМ він буде блокований, а дані з РВ будуть передані назад безпосередньо в БК, що дасть можливість реалізувати наступну спробу виконання команди [8 , 9, 10].

Таким чином, при такому підході ресурси системи використовуються більш ефективно. Немає необхідності зберігати фрагменти комірок в СФК. Комірки ВАД можуть використовуватись для інших алгоритмів або інших ітерацій. Це в свою чергу дозволяє поєднувати обчислювальні процеси не тільки на рівні команд, а й на рівні алгоритмів, що забезпечує скорочення часу перетворення інформації [10, 11, 12].

Проте, з точки зору витрат часу на реконфігурацію системи, при відмові ОМ даний підхід, як і розглянуті варіанти, є недостатньо ефективним. Дійсно, час очікування спрацьовування таймера може бути набагато більше, ніж час, реально необхідне для виконання команд. Це вносить додаткову часову затримку при реконфігурації системи.

З розглянутих підходів видно, що перша архітектура є оптимальним рішенням, адже показано, що запропонований підхід забезпечує підвищення швидкості реконфігурації системи при відмові обчислювальних модулів. Це пов’язано із більш ефективною методикою контролю часових інтервалів. Для кожної команди визначається індивідуальний інтервал контролю. У відомих системах використовують максимальний інтервал для всіх команд.

Реконфігурація системи при відмові обчислювального модулю реалізується в основному на апаратному рівні і зводиться до відключення модулю, що відмовив.

Таким чином, у порівнянні з відомими потоковими системами запропонований підхід прискорює час розв’язання задач, що є важливим чинником для обробки даних в реальному часі.

1.3. Сучасні методи проектування обчислювальних систем

У сучасному світі, проектування та тестування подібних систем об’єктивно проводити на емуляторах подібних систем керування потоками даних, тому існують програмні засоби які дозволяють це робити та проводити подальші тести. Дані програмні засоби моделювання систем працюють на певній мові, завдяки якій задаються системні модулі та описують поведінку системи, а потім оброблюють вхідні дані залежно від того, як це буде описано всередині.

Умовна мова, яка використовується для опису подібних модулів називається HDL. Мова опису апаратури.

У комп'ютерній інженерії мова опису апаратного забезпечення (HDL) - це спеціалізована комп'ютерна мова, що використовується для опису структури та поведінки електронних схем, а також, найчастіше, цифрових логічних схем.

Мова опису апаратного забезпечення дає точний, формальний опис електронної схеми, що дозволяє автоматизувати аналіз та моделювання електронної схеми. Це також дозволяє синтезувати опис HDL в netlist (специфікація фізичних електронних компонентів і як вони з'єднані разом), які потім можуть бути розміщені та спрямовані для створення набору масок, що використовуються для створення інтегральної схеми.

Мова опису апаратного забезпечення виглядає так само, як мова програмування, такий як C; це текстовий опис, що складається з виразів, тверджень і керуючих структур. Однією з найважливіших відмінностей між більшістю мов програмування є те, що HDL явно включає поняття часу. Приклад лістингу коду наведено у лістингу 1.1.

Лістинг 1.1. Лістинг приклад коду на мові Verilog

```
library iEEE;
use iEEE.STD_LOGIC_1164.ALL;
use iEEE.STD_NUMERIC_STD.ALL;
entity not1 is
  port(a:in STD_LOGIC;
       b:out STD_logic);
end not1;
architecture behavioral of not1 is
begin
  b <= not a;
end behavioral;
```

У першій строчці лістингу 1.1 «library iEEE» відбувається підключення набору логічних елементів до робочого простору, далі підключення «use iEEE.STD_LOGIC_1164.ALL» підключення конкретних блоків та плат. Далі йде використання та подання сигналів на відповідні входи плат.

Найкращі IDE для розробки та емулювання роботи обчислювальних систем на даний момент Quartus II та Xlinix Verilog[13,14].

1.Quartus II Altera

Програмне забезпечення Altera Quartus II забезпечує повний, багатоплановий дизайн середовища, який легко адаптується до ваших специфічних дизайнерських потреб. Це комплексне середовище для системи-на-програмуванні-чипа (SOPC) . Програмне забезпечення Quartus II включає рішення для всіх етапів FPGA. FPGA – напівпровідниковий пристрій, що може бути налаштований виробником або розробником після виготовлення. Структура Quartus II зображена на рисунку 1.7.

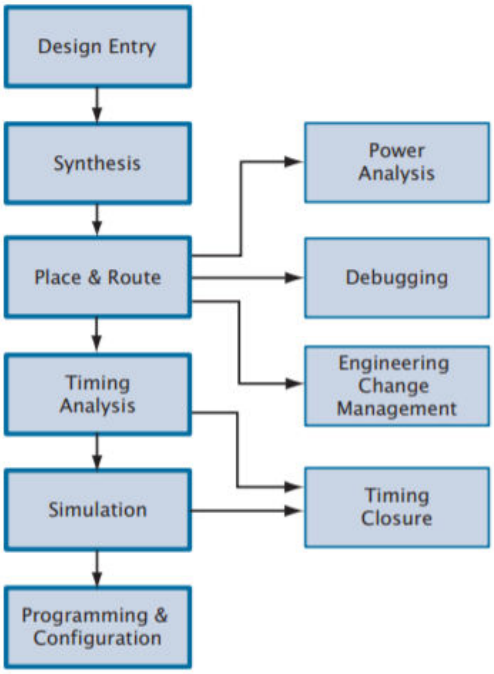


Рис.1.7. Дизайн структура Quartus II

Крім того, програмне забезпечення Quartus II дозволяє використовувати Quartus II. Графічний інтерфейс користувача та інтерфейс командного рядка для кожної фази дизайн потоку Ви можете використовувати один із цих інтерфейсів для всього потоку, або ви може використовувати різні варіанти на різних етапах. На рисунку 1.8 зображено робочий спейс Quartus II [13].

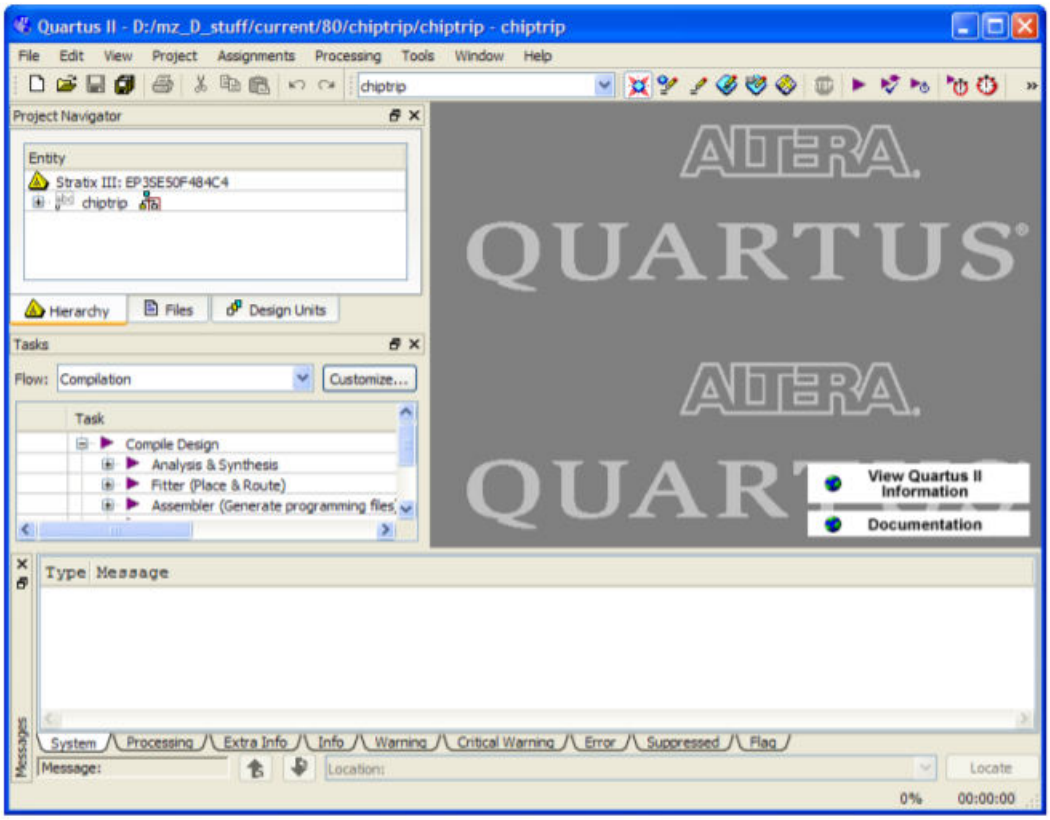


Рис.1.8. Quartus II

2.Xilinx Verilog

Написання коду на Verilog дозволяє нам сфокусуватися на поведінці апаратури з точки зору високого рівня замість того, щоб описувати схему на низькому рівні за допомогою низькорівневих елементів логіки. Існує безліч якісної літератури з програмування на HDL-мовах Verilog і VHDL[14]. Однак, на жаль, часто такі книжки великі за обсягом і досить важкі для освоєння і розуміння. Розробка цифрової схеми на Verilog

зазвичай включає в себе два основних процеси - тест на основі симуляції і реалізація робочого коду. На рисунку 1.9 та 1.10 зображено робочий спейс Xlinix.

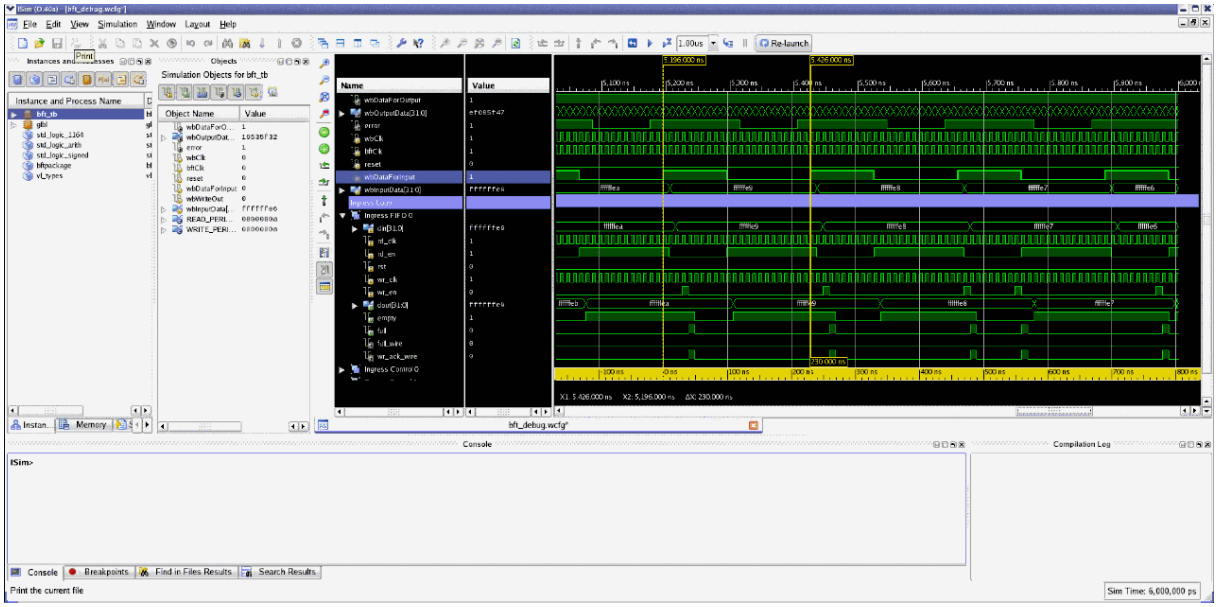


Рис.1.9. Xlinix таблиця результатів емуляції схеми

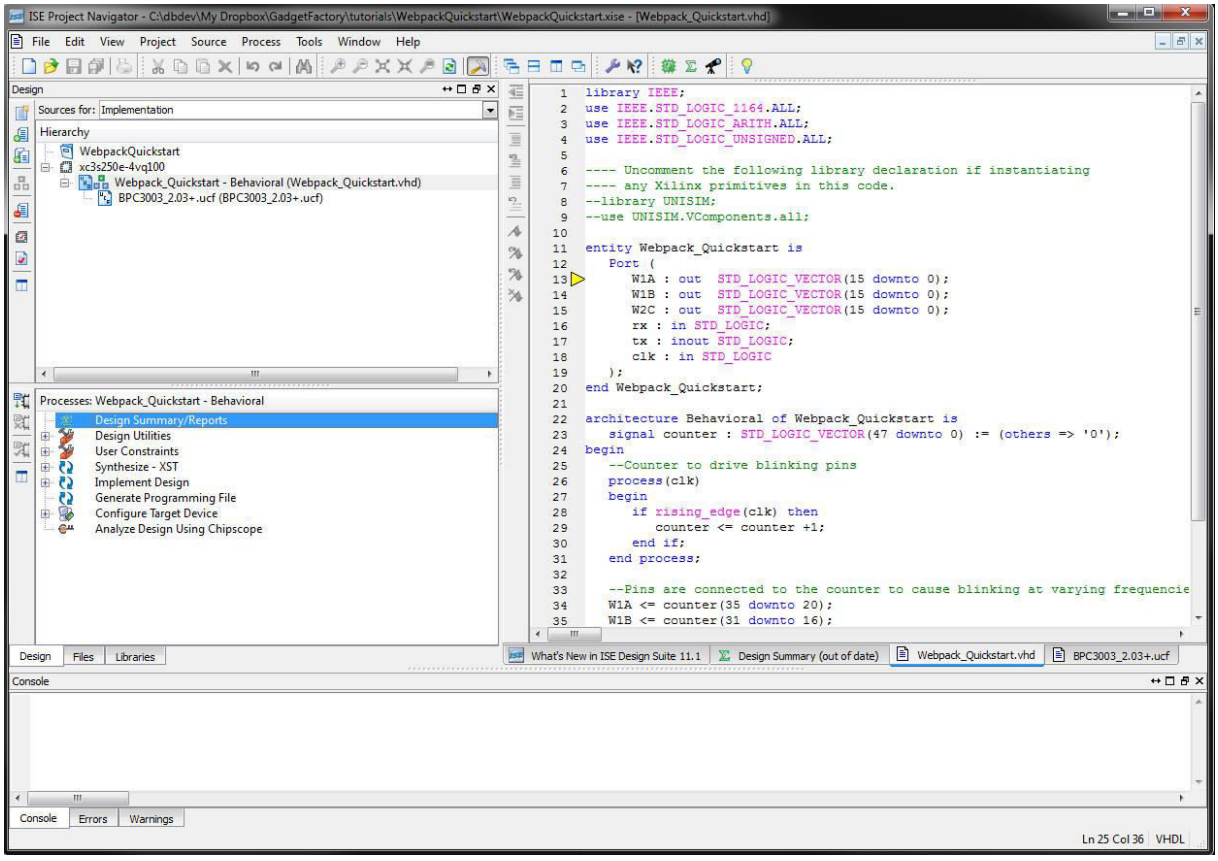


Рис.1.9. Xlinix написання програмного коду мовою Verilog

Обидва програмні рішення містять в собі великий набір логічних елементів та дуже широкий набір інструментів для створення вискорівневих та низькорівневих емуляцій обчсилювальних систем, проте мають перелік недоліків:

- Негнучкі, вимагають встановлення додаткових бібліотек та знань програмного коду;
- Створення кожної емулятивної моделі з нуля;
- Необхідність створення кожної окремої архітектури.

1.4. Висновки

В даному розділі був проведений аналіз існуючих методів підвищення ефективності роботи систем, що керуються потоками даних, також були наведені недоліки існуючих методів. Було описано загальний підхід до створення систем керування потоками даних, також були наведені засоби створення моделей обчислювальних систем та наведені їх недоліки.

З огляду на проведений аналіз було виділено набір вимог та параметрів, яким має відповідати система, що буде описана в подальшому у наступних частинах роботи, також було сформовано вимоги до емулятору, основною з яких стало спрощення роботи з емулятором та тестування ОС. Також було сформовано метод, за допомогою якого модна спростити недоліки існуючих систем з вільним адресним простором, зберігши при цьому їх швидкодію порівняно з асоціативною пам’ять.

2. МЕТОД ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМ, ЩО КЕРУЮТЬСЯ ПОТОКАМИ ДАНИХ ТА РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ

2.1. Архітектура системи та окремі елементи системи

Як вже зазначалося вище, системи керування потоками даних мають в своїй основі дві основні складові – це СФК та обчислювальні блоки. Кількість обчислювальних блоків необмежена починаючи з 1го. СФК – система формування команд, в її основу входять усі основні елементи системи включаючи блоки пам’яті, та відповідно до того, як сформовано дане середовище і буде працювати уся ОС. Можна сказати, що СФК визначає архітектуру [15].

З аналізу проведеного у першому розділі, одна проблем СПД в тому, що вони використовують надлаштування у вигляді асоціативної пам’яті, для реалізації зручного алгоритму та динамічної реконфігурації системи. Проте вільний адресний простір працює швидше за асоціативну пам'ять, до того ж блок адресного простору завжди присутній у ПЛІС, тоді, як асоціативну пам'ять необхідно додавати окремо. Ще її називають, пам'ять адаптована до контенту вона більш зручна для формування окремих комірок для запам’ятовування.

Пам'ять, адаптована до контенту – це спеціальний тип пам'яті комп'ютера, що використовується в деяких дуже швидкісних пошукових програмах. Він також відомий як асоціативна пам'ять або асоціативне сховище і порівнює вхідні дані пошуку (тег) з таблицею збережених даних і повертає адресу відповідних даних (або у випадку асоціативної пам'яті, відповідних даних). На рисунку 2.1 схематично зображено принцип роботи асоціативної пам’яті.

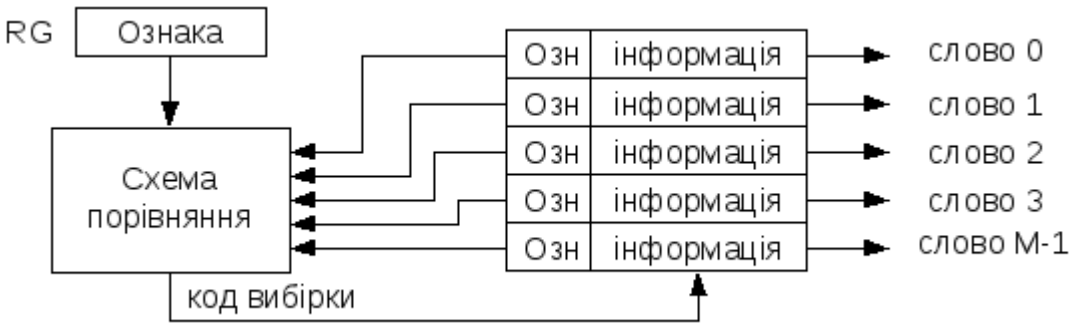


Рис.2.1. Принцип роботи асоціативної пам’яті

Адресний простір – сукупність всіх допустимих адрес будь-яких об'єктів обчислювальної системи – осередків пам'яті, секторів диска, вузлів мережі тощо, які можуть бути використані для доступу до цих об'єктів при певному режимі роботи (стан системи). На відміну від асоціативної, просто являє собою матрицю з комірок з ідентифікаторами, що значно поскладнює роботу з нею у випадку, коли необхідну комірку треба знайти повторно, і коли невідомо, яка з комірок буде необхідна для повторного пошуку, у кожний момент часу необхідно знати адресу. По факту – це матриця [16].

Як зазначалося у першій частині використання вільного адресного доступу є більш оптимальним, тому при проектуванні архітектурі будемо використовувати саме блоки вільного адресного простору, проблему запам’ятовування останніх значень системи, наприклад операндів, буде вирішено додаванням додаткових регістрів. Далі опис процесу формування даних у системі, нічим не відрізняється від систем прототипів.

Як вже зазначалося вище, система керування працює відповідно до граф алгоритму та визначає в собі та оперує 2 типами даних:

- Операнд;
- Актор;

Дані слова довжини залежно від типу системи 32х 64х бітні і так далі, в основі роботи потоків даних лежать саме ці дані, вони починають

формуватися з моменту зчитування вхідних сигналів, формат керуючого слова зображено на рисунку 2.2. Заголовки полів та їх розміри у бітах.

Т	Операція	Номер	Код	Наступна операція	Ввод	Порядок	К
1	16	1	10	16	3	16	1

Рис.2.2. Формат керуючого слова актора

З рисунку 2.2, заголовки полів слова:

- «Т» – тип слова, одиниця для керуючого слова;
- «Операція» – визначає адресу в пам’яті керуючого слова та пам’яті операндів, дане значення визначається на етапі формування і не може бути зміненим для даної комірки, залежно від актора на граф алгоритмі системи;
- «Номер» – номер обчислюваного операцією операнда, дане поле необхідно для тих операцій в яких важливий порядок слідування операндів наприклад ділення чи віднімання, команда обов’язково містить один операнд зі значенням «0» та один операнд зі значенням «1».
- «Код» – дане поле відповідає за кодовий ідентифікатор, з обмеженим кількістю значень кодування.
- «Наступна операція» – визначає адресу комірки в якій знаходиться дані для наступної операції, що стає доступної після виконання ОМ даної.
- «Ввод» – значення, що визначає з яких позицій може вводитися значення до системи для цієї операції, дане поле необхідне для обмеження зчитувань паралельних операцій з одно входу, адже тоді це буде заважати розпаралелюванню.

- «Порядок» – значення визначає пріоритети для пристроїв вводу в систему, дане значення необхідне для зменшення затримок зчитування, тобто виключення варіанту простою значень на входах системи, виключає можливість ввести два операнди без слова керування.
- «К» – поле константи, може приймати значення «0» або «1», визначає чи є операнд операції константою, дане значення важливе, адже у випадку якщо значення встановлене в «1», значення не очиститься після відпрацювання обчислювальним блоком даної операції, відповідно команда може приймати лише 1 операнд зі значенням константи.

Формат слова операнда зображено на рисунку 2.3.

Т	Операція	Номер	Операнд	Ввод	Порядок	К
1	16	1	26	3	16	1

Рис.2.3. Формат слова операнда

З рисунку 2.3, заголовки полів слова:

- «Т» – тип слова, нуль для операнда;
- «Операція» – визначає адресу в пам’яті керуючого слова та пам’яті операндів, дане значення визначається на етапі формування і не може бути зміненим для даної комірки, залежно від актора на граф алгоритмі системи;
- «Номер» – номер обчислюваного операцією операнда, дане поле необхідно для тих операцій в яких важливий порядок слідування операндів наприклад ділення чи віднімання, команда обов’язково містить один операнд зі значенням «0» та один операнд зі значенням «1».

- «Операнд» – визначає безпосередньо операнд над яким буде виконуватись операція.
- «Ввод» – значення, що визначає з яких позицій може вводитися значення до системи для цієї операції, дане поле необхідне для обмеження зчитувань паралельних операцій з одно входу, адже тоді це буде заважати розпаралелюванню.
- «Порядок» – значення визначає пріоритети для пристроїв вводу в систему, дане значення необхідне для зменшення затримок зчитування, тобто виключення варіанту простою значень на входах системи, виключає можливість ввести 2 операнди без слова керування.
- «К» – поле константи, може приймати значення «0» або «1», визначає чи є операнд операції константою, дане значення важливе, адже у випадку якщо значення встановлене в «1», значення не очиститься після відпрацювання обчислювальним блоком даної операції, відповідно команда може приймати лише 1 операнд зі значенням константи.

Таким чином наступним кроком при створенні архітектури є формування СФК.

Метод підвищення ефективності полягає у проектування СФК системи, з урахуванням наступнх пунктів, які виправляють недоліки систем, що були описані у першому розділі. Серед нововведень можна виділити наступні основні частини:

- Використання ВАД замість асоціативної пам’яті;
- Додавання додаткових регістрів для збереження актуальних операндів та операцій;
- Додавання додаткових комірок для збереження мета-даних операцій, до мета даних належать:

- Код операції;
- Час виконання операції.

- Додавання окремого блоку з таймером для кожного обчислювального блоку системи;

Дані введення дозволять системі працювати оптимальніше через оптимізацію відмовостійкості, а також прискорення роботи через більш швидку пам’ять. На рисунку 2.3 зображено повну архітектурну схему роботи розробленої обчислювальної системи, що керується потоками даних на базі досліджуваного методу [17].

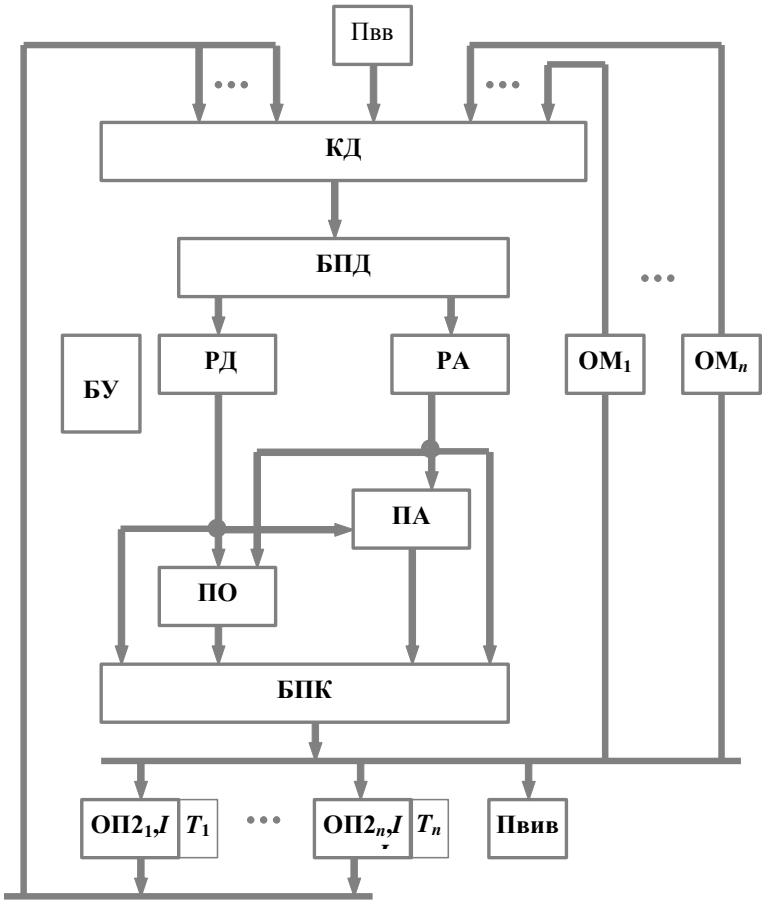


Рис.2.3. Розроблена архітектура

Структура системи показана на рис. 2.2, дана структура є розвитком манчестерської архітектури , але є простішою і більш швидкодіючої, що

пов’язано з використанням запам'ятовуючих пристроїв з адресним доступом замість асоціативної пам'яті.

СПД містить пристрої введення (Пвв) та виведення (Пвив), обчислювальні модулі ОМ, блок управління (БУ). Буферна пам'ять даних (БПД) і команд (БПК), а також пам'ять операндів (ПО), акторів (ПА), регістри адреси (РА) і даних (РД) утворюють СФК. Два додаткових регістра призначені для зберігання другого операнда із номером команди та кодом таймера (ОП2i, I; Ti).

Розглянемо роботу системи, коли відсутні відмови ОМ. Актори і дані через Пвв і КД записуються в буфер даних БПД. Імена I_i об’єктів (1) та (2) використовуються у якості адрес модулів пам’яті ПО та ПА. Через РА адресується комірка пам’яті для всіх складових команди. Актор через РД записується в ПА. Дане, що потрапляє в систему першим через РД записується в ПО. Друге дане разом з актором та першим операндом потрапляє відразу в БПК, де накопичуються готові для виконання команди.

Вільний ОМ зчитує команду з буфера БПК та виконує її відповідно зі значенням актора. Результат потрапляє знов в СФК, де приймає участь в формуванні нової команди. Через Пвив дані можуть видаватися із системи.

В системі відбувається контроль правильності виконання команди методом контролю часових інтервалів. Це відбувається наступним чином.

В склад актора (1) додається поле T_i для ініціалізації таймера. Для кожної команди призначається інтервал часу, який гарантує виконання команди, якщо ОМ працює вірно. Відомо, що час виконання команд може суттєво відрізнятися. Наприклад, множення без прискорення майже в m разів виконується довше ніж додавання або логічні операції (m – розрядність операндів). Це вигідно відрізняє підхід до контролю по відношенню до системи прототипу, де інтервал завжди дорівнює одному максимальному значенню.

Водночас із записом команди в ОМj для її виконання в додатковий регістр з таким самим номером j записуються поля: другий операнд, що прийшов в систему останнім, номер комірки пам'яті в ПА та ПО і код ініціалізації таймера, в якості якого використовується частина додаткового регістру. Таймер запускається з початком виконання команди в ОМ. Після одержання результату операції ОМ скидає свій таймер в додатковому регістрі. Якщо операція не закінчилась у визначений проміжок часу, спрацьовує таймер. В цьому випадку другий операнд разом із своєю адресою по сигналу таймера потрапляє в БПД і знов формує команду, бо перший операнд і актор зберігаються за даною адресою в СФК.

ОМ, що відмовив, блокується. Команда природним чином потрапляє в інший ОМ, де виконується. Дані для системи готуються без врахування кількості ОМ в системі. Тому обчислення можуть продовжуватись поки є ОМ, що працюють. Відновлення системи не потребує комутації модулів, бо всі вони об'єднані одним комунікаційним середовищем.

2.2. Відмовостійкість розробленої системи

У попередньому розділі було визначено архітектуру, що є більш оптимальною та швидкою ніж аналоги з вільним адресним доступом. Таким чином в попередньому підрозділі на рис 2.3 зображена архітектура з описом її роботи у двох станах, при усіх справних процесорах та при відключенні обчислювального блоку.

Отже при ситуації з відмовою обчислювального блоку, відбувається динамічна реконфігурація системи. Більш детально цей процес було вже описано у попередньому пункті роботи, слід зазначити, що при додаванні додаткових регістрів зберігання даних, мі вирішили проблему вільного адресного доступу, що не міг зберігати невиконані операції.

Роздивимось розроблену з точки зору відмовостійкості, з підвищеною надійністю. У подібній системі інтенсивність відмов

підсистеми, не буде просто сумою відмов окремих елементів обчислювальною системи, через процес рекокнфігурації самої системи.

Відомо, що будь яка система буде працювати та виконувати свої функції, доки хоча б один з блоків обчислень функціонує справно, також є вірогідність відновлення роботи блоку після перенавантаження, на основі цього можна зробити висновок, що система являє собою цілісну машину станів, з єдиним фінальним варіантом у даний момент часу. Кожний блок має певну затримку при реконфігурації, тобто це час що система витратить на відправлення не виконаної команди на повторне коло обробки, якщо прийняти час відмови за t, час зчитування команди з резервного регістру за t1, а час подання та формування команди за t2, логічним буде зробити висновок, що час реконфігурація системи для одного блоку, що вийшов з ладу буде рахуватися за формулою:

$$T_{затримки} = t1 + t2 + t3 \qquad (2.1)$$

Аналогічно для блоку, що відновив роботу. Порівняно з системою в якій використовується асоціативна пам'ять даний процес займає більше часу, проте час формування команді з готових операндів, який і є додатковими витратами менший за час доступу до пам’яті в цілому, адже до пам’яті система звертається кожного такту своєї роботи, тоді як затримка при реконфігурації системи присутня лише у ситуаціях зупинки або відновлення роботи системи [18].

Блок схема поведінки системи при відключенні процесору під час виконання певної операції та подальша обробка даного виключення зображена на рисунку 2.4.

Слід зазначити, що у випадку якщо один з операндів був константою, яка не затирається при відпрацюванні команд та за умови, що дана константа була використана, команду можна зразу формувати повторно не звертаючись до запам’ятовуючого регістру, адже, як було описано вище у розділі 2.1 слова операнди позначені, як константи не

затираються після відпрацювання команди, а залишаються у пам’яті з самого початку роботи системи до її повного завершення, також система буде простоювати у випадку, якщо жоден з процесорів не відповідає на сигнали блоку управління, тоді остання команда буде намагатися запустити процесор до тих пір доки хоч один не запрацює [19].

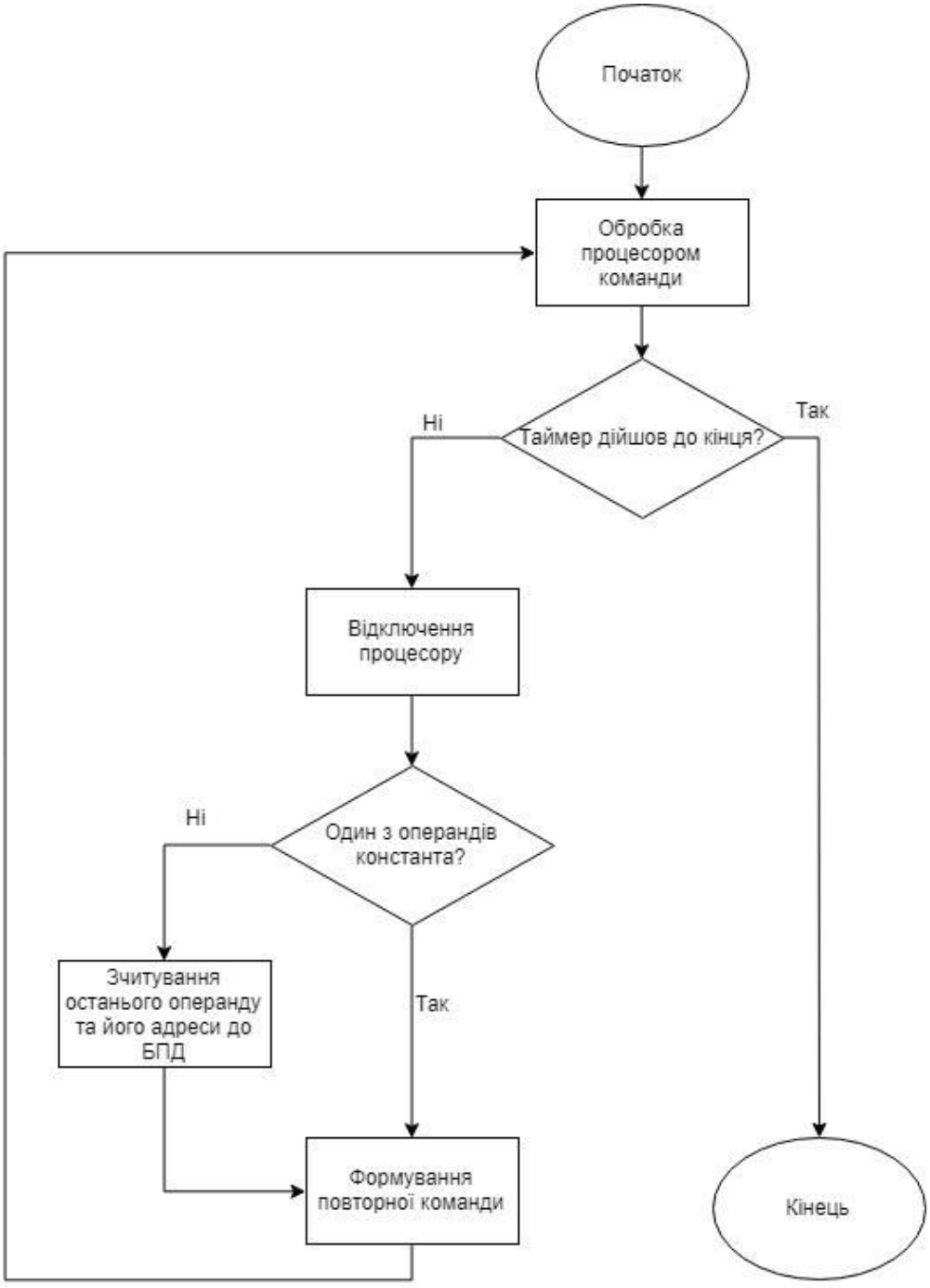


Рис.2.4. Блок схема поведінки системи при відключенні процесора

2.3. Програмний емулятор розробленої архітектури

З огляду на розроблену архітектуру, яка базується на методі підвищення ефективності роботи систем, що керуються потоками даних за рахунок використання вільного адресного простору та підвищення відмовостійкості системи шляхом додавання таймерів.

Отже після проектування алгоритму необхідно провести набір тестів та замірів, для чого необхідно створити зручний та гнучкий емулятор роботи системи. Для цього необхідно розробити наступні модулі:

- Внутрішній компілятор команд, з надлшатування для формування граф алгоритму. Компілятор має будувати дерево з вхідного файлу та формувати актори та дані, це дозволить системі вподальшому формувати керуючі слова;
- Розробити власну граматику та відповідно мову, для подачі на вхід системі файлу з командами, для рішення тієї чи іншої задачі;
- Розробити машину станів, яка повність емулює поведінку багатопроцесорної системи, а також емуляцію роботи пам’яті;
- Зробити зручний та зрозумілий для користувача інтерфейс системи, тобто відображення;
- Механізм формування результатів з тестами.

Слід зазначити, що даний підхід до емуляції дозволяє користувачу працювати та тестувати систему без попереднього налаштування її та не вимагає окремих знань з проектування у таких системах, як Quartus II, чи Xlinix, які є громіздкими та вимагають фундамментальних знань, емулятор дозволяє задати користувачу лише набір вхідних команд для рішення його задачу на даній архітектурі.

Для реалізації даного емулятору було обрано наступні інструменти та технології:

- Мова програмування C#;
- Технологія WPF (Windows Presentation Foundation);
- Devexpress, набір візуальних елементів для WPF;
- Паттерн MVVM;
- DiagramFlowFormer, зручна бібліотека для формування діаграм.

2.3.1 Огляд мови програмування C#

C# – об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research (при фірмі Microsoft).

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. Перейнявши багато що від своїх попередників – мов C++, Delphi, Модула і Smalltalk – C#, спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад множинне спадкування класів (на відміну від C++).

До числа принципово важливих рішень, які реалізовані корпорацією Microsoft у мові програмування C#, можна віднести наступні:

- компонентно-орієнтований підхід до програмування (який характерний і для ідеології Microsoft. NET в цілому);
- властивості як засіб інкапсуляції даних (характерно також в цілому для ООП);
- обробка подій (маються розширення, в тому числі в частині обробки виключень, зокрема, оператор try);

- делегати (delegate – розвиток покажчика на функцію в мовах C і C++);
- індексатори (indexer – оператори індексу для звернення до елементів класу-контейнера);
- перевантажені оператори (розвиток ООП);
- оператор foreach;
- механізми boxing і unboxing для перетворення типів;
- прямокутні масиви (набір елементів з доступом за номером індексу і однаковою кількістю стовпців і рядків).

Для створення десктопних додатків на мові C#, існує підсистема для побудови графічних інтерфейсів WPF(Windows Presentation Foundation). WPF – це перше реальне оновлення технологічного середовища призначеного для користувача інтерфейсу з часу випуску Windows 95. Воно включає нове ядро, яке повинне замінити GDI і GDI+, використовувані на нинішній Windows-платформі. WPF є високорівневим об'єктно-орієнтованим функціональним шаром, що дозволяє створювати двовимірні та тривимірні інтерфейси [20].

WPF інтерфейси описуються за допомогою розширюваної мови розмітки XAML(Extensible Application Markup Language). XAML є мовою розмітки, яку використовують для створення екземплярів об'єктів .NET. Хоча мова XAML — це технологія, що може бути застосовна до багатьох різних предметних областей, її головне призначення — конструювання інтерфейсів користувачів WPF.

Основні переваги, які дає розроблювачам десктоп додатків використання WPF:

- веб подібна модель компоновки об’єктів, що дозволяє створити гнучкий для адаптації графічний інтерфейс;

- насичена модель малювання, що дозволяє працювати вже з готовими примітивами, фігурами, блоками та іншими графічними елементами;
- розвинена текстова модель, відображення стилізованого тексту, незалежно від типу стилізації;
- анімація, не треба задавати таймери та умови, для того щоб, форма себе перемалювала, анімація в WPF рреалізована на рівні частини платформи;
- підтримка відео та аудіо, попередники такі, як Windows Forms, були сильно обмежені у роботі з мультимедіа, однак WPF включає в себе підтримку відтворення будь-якого аудіо або відео файлу(який підтримується Windows Media), дозволяючи відтворювати більше одного медіа файлу одночасно;
- стилі та шаблони, стилі дозволяють стандартизувати форматування та багатократно використовувати його по всьому додатку, шаблони дають можливість змінити спосіб відображення, навіть тих, що були задані, як кнопки;
- команди, не має значення звідки буде ініційована команда “Open”, через меню або через панель інструментів, кінцевий результат один і той самий, ця абстракція доступна коду, тобто можна визначити команди додатку в одному місці, а прив’язати, їх до різних елементів по всьому проекту;
- додаток на основі сторінок, використовуючи WPF, можна будувати браузер-подібні додатки, які дозволять рухатися по колекції сторінок, за допомогою навігації, наприклад кнопок вперед-назад, подібний проект можна відкрити у браузері не дивлячись на те, що він призначений для десктопного використання;

- декларативний користувацький інтерфейс, хоча можна конструювати вікно WPF в коді, є інший підхід, наповнення вікна серіалізується у вигляд XML-дискрипторів у документ XAML. Тобто графічний інтерфейс користувача повністю відділено від коду і дизайнери можуть використовувати професійні інструменти для редагування XAML;
- WPF має перевагу у швидкості в порівнянні з іншими технологіями, для побудови графічних інтерфейсів, саме WPF замінив свого попередника Window Forms.

Недоліками WPF є:

- необхідність наявності ліцензійного ПЗ для розгортування проекту.

2.3.2 Огляд паттерну проектування MVVM

Model-View-VeiwModel (MVVM) — це шаблон проектування, що застосовується під час проектування архітектури застосунків (додатків). Публічно вперше був представлений Джоном Госсманом (John Gossman) у 2005 році як модифікація шаблону Presentation Model (Windowss Forms). MVVM орієнтований на такі сучасні платформи розробки, як Windows Presentation Foundation та Silverlight від компанії Microsoft.

MVVM полегшує відокремлення розробки графічного інтерфейсу від розробки бізнес логіки (бек-енд логіки), відомої як модель (можна також сказати, що це відокремлення представлення від моделі). Модель представлення є частиною, яка відповідає за перетворення даних для їх подальшої підтримки і використання. З цієї точки зору, модель представлення більше схожа на модель, ніж на представлення і оброблює більшість, якщо не всю, логіку відображення даних. Модель представлення може також реалізовувати патерн медіатор, організовуючи доступ до бек-

енд логіки навколо множини правил використання, які підтримуються представленням.

MVVM використовується для відокремлення моделі та її відображення. Необхідністю цього є надання можливості змінювати їх незалежно одну від одної. Наприклад, розробник працює над логікою роботи з даними, а дизайнер — з користувацьким інтерфейсом.

MVVM була створена з метою поділу праці дизайнера і програміста, що є неможливим, коли Java-розробник намагається побудувати GUI в Swing або розробник на Visual C++ намагається створити користувальницький інтерфейс в MFC. Розробники — кмітливі хлопці і мають безліч навичок, але створення зручних і привабливих інтерфейсів вимагає абсолютно інших талантів, ніж ті, якими вони володіють. Ця робота більше підходить для дизайнерів інтерфейсів. Хороші дизайнери інтерфейсів краще знають, чого бажають користувачі, ніж експерти в області проектування і написання коду. Зрозуміло, буде краще, якщо дизайнер інтерфейсів створить інтерфейс, а розробник напише код, який реалізує логіку цього інтерфейсу, але технології типу Swing або MFC просто-напросто не дозволяють чинити таким чином.

MVVM зручно використовувати замість класичного MVC та йому подібних у тих випадках, коли на платформі, де ведеться розробка, присутнє «зв'язування даних». У таких технологіях, як WPF та Silverlight, присутня концепція «зв'язування даних», що дозволяє зв'язувати дані із візуальними елементами в обидві сторони.

Поділ “відповідальностей” архітектури MVVM:

- розробка користувацького інтерфейсу здійснюється дизайнером інтерфейсів за допомогою технології, більш-менш природної для такої роботи (XML);
- логіка користувацького інтерфейсу реалізується розробником як компонент ViewModel;

- Функціональні зв'язки між користувацьким інтерфейсом та ViewModel реалізуються через біндинги (bindings), які, по суті, є правилами типу «якщо кнопка А була натиснута, повинен бути викликаний метод onButtonAClick() з ViewModel». Біндинги можуть бути написані в коді.

Архітектура MVVM використовується в тому чи іншому вигляді усіма сучасними технологіями, наприклад Microsoft WPF і Silverlight, Oracle JavaFX, Adobe Flex, AJAX.

Шаблон MVVM ділиться на три частини:

- модель (Model), як і в класичному шаблоні MVC, Модель являє собою фундаментальні дані, що необхідні для роботи застосунку;
- вид/(Вигляд) (View) як і в класичному шаблоні MVC, Вигляд — це графічний інтерфейс, тобто вікно, кнопки тощо;
- модель вигляду (ViewModel, що означає «Model of View») з одного боку є абстракцією Вигляду, а з іншого надає обгортку даних з Моделі, які мають зв'язуватись. Тобто вона містить Модель, яка перетворена до Вигляду, а також містить у собі команди, якими може скористатися Вигляд для впливу на Модель. Фактично ViewModel призначена для того, щоб здійснювати зв'язок між моделлю та вікном, відслідковувати зміни в даних, що зроблені користувачем, відпрацьовувати логіку роботи View.

Схема, як описані три частини взаємодіють між собою наведена на рисунку 2.5.

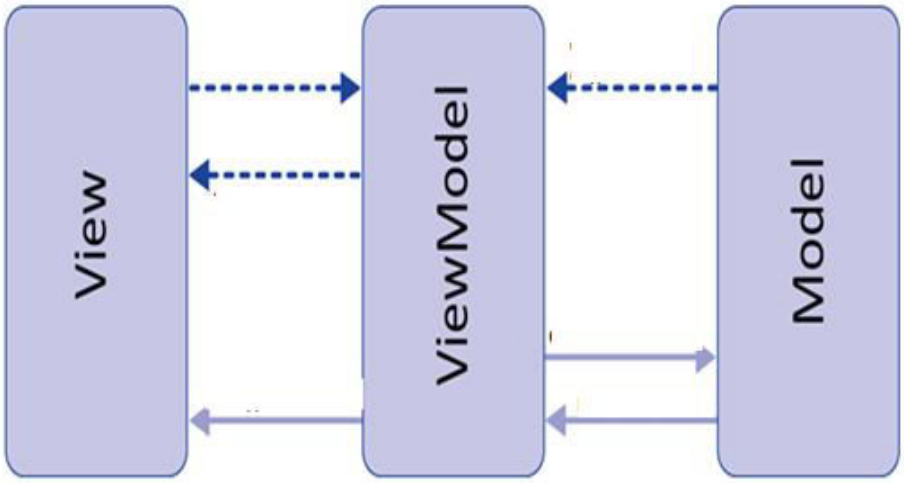


Рис. 2.5. Взаємодія компонентів шаблону MVVM.

Даний підхід дозволяє пов’язувати елементи уявлення з властивостями і подіями View-моделі. Можна стверджувати, що кожен шар цього шаблону не знає про існування іншого шару.

Ознаки View-моделі:

- двостороння комунікація з представленням;
- view-модель це абстракція представлення. Зазвичай це означає, що властивості представлення співпадають з властивостями View-моделі / моделі;
- View-модель не має посилання на інтерфейс представлення (IView). Зміна стану View-моделі автоматично змінює представлення та навпаки, оскільки використовується механізм прив’язок даних(Binding);
- один екземпляр View-моделі зв’язаний з одним відображенням.

Особливості реалізації шаблону MVVM полягає в тому, що немає необхідності представленню реалізовувати інтерфейси, як наприклад IView. Представлення повинно мати посилання на джерело даних (DataContex), яким є View-модель. Елементи представленні у вигляді

зав’язків з відповідними параметрами, властивостями та подіями View-моделі.

У свою чергу, View-модель реалізує спеціальний інтерфейс, який використовується для автоматичного оновлення елементів уявлення. Прикладом такого інтерфейсу в WPF може бути INotifyPropertyChanged.

2.4. Висновки

У даному розділі було досліджено метод підвищення ефективності роботи систем керування потоками даних у багатопроцесорних ОС, також було розроблено архітектуру, яка покращаю роботу подібних систем. Було розроблено структуру програмного засобу емуляції та обрано інстурменти для реалізації даного програмного засобу.

Приведено блок схему алгоритму роботи системи у випадку відключення процесору та описано принцип виграшу у порівнянні з системами іншого типу. Проведено аналіз у порівнянні з системами на основі вільного адресного доступу з першого розділу.

Таким чином, отримані результати підтверджують ефективність даної архітектури у порівнянні з аналогами на основі такої самої адресації та дають виграш у порівнянні з системами на основі асоціативної пам’яті.

3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЕМУЛЯЦІЇ РОБОТИ АРХІТЕКТУРИ НА БАЗІ МЕТОДУ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОБОТИ СПД

2.1 Архітектура програмного забезпечення

Отже для тестування розробленої архітектури, необхідно створити программний засіб для емуляціхї роботи системи за заданими параметрами, у другому розділі даної роботи. У другому розділі було визначено мову та технології з якими буде виконуватись ПЗ.

Отже до програмного забезпечення будуть входити такі модулі:

- Компілятор;
- Формувач вхідних даних;
- Машина станів;
- Модель відображення;

На рисунку 3.1 зображена взаємодія усіх модулів описаних вище.

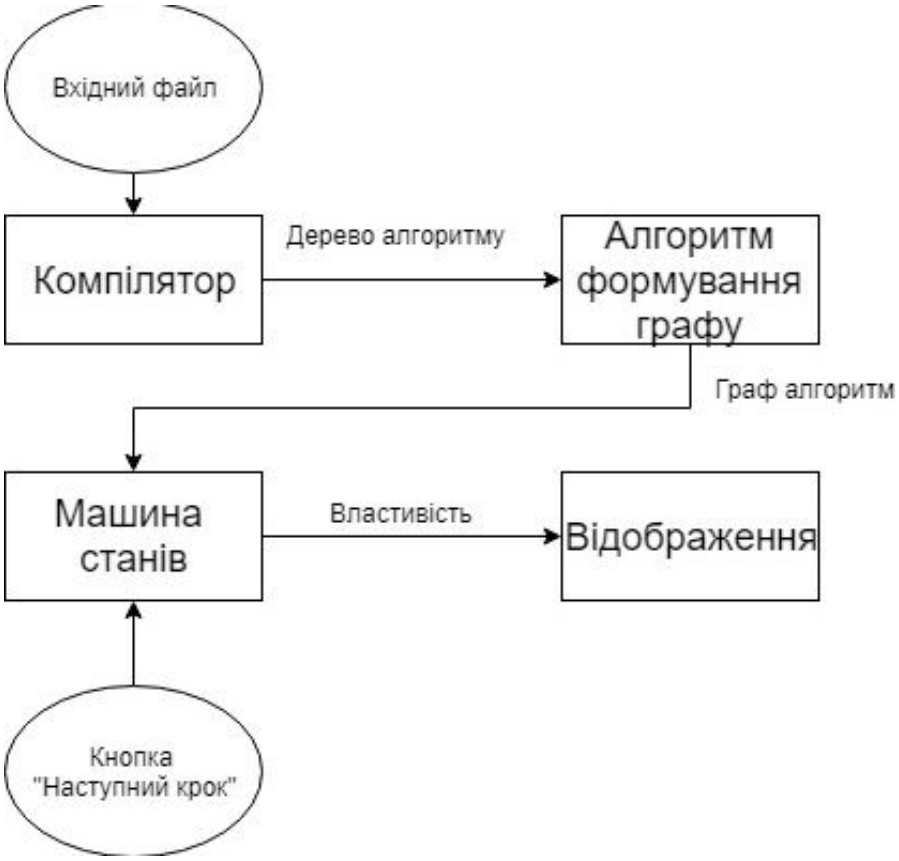


Рис.3.1. Схема роботи програмного емулятору

2.2 Компілятор та граматика для вхідного файлу

Компілятор - це комп'ютерна програма, яка перетворює комп'ютерний код, написаний на одній мові програмування (мовою оригіналу), на іншу мову програмування (мову-ціль). Компілятори - це тип перекладача, який підтримує цифрові пристрої, перш за все, комп'ютери. Компілятор імен перш за все використовується для програм, які перекладають вихідний код з мови програмування високого рівня на мову нижчого рівня (наприклад, мову асемблера, об'єктного коду або машинного коду) для створення виконуваної програми [23].

В даному випадку компілятор працює перетворювачем з формату вхідної мови розробленої спеціально для емулятору, приклад лістинг 3.1.

Лістинг 3.2 Приклад мови вхідного файлу.

```
1: cmd1 obj1, obj2,  2[0]
 2: cmd2    _, obj3, 13[0]
 3: cmd1 obj4, obj2,  4[0]
 4: cmd2    _, obj3, 14[0]
 5: cmd1 obj5, obj6,  6[1]
 6: cmd3 obj7,    _,  9[1]
 7: cmd1 obj8, obj6,  8[1]
 8: cmd3 obj7,    _,  9[0]
 9: cmd1    _,    _, 10[0]
10: cmd2    _, obj3, 12[0]
11: cmd4 obj7, obj3, 12[1]
12: cmd5    _,    _, 13[1]
13: cmd6    _,    _, 14[1]
14: cmd3    _,    _, 15[0]
15: ret     _
```

Формат мови розберемо на прикладі 1 строки коду, першої.

Складається з набору параметрів:

- Перший параметр номер рівня коду для орієнтації по коду;
- Другий параметр назва команди;

- Третій параметр перший операнд
- Четвертий параметр другий операд;
- Пятий параметр кскладається з 2х частин:
    - Номер строки в яку піде результат;
    - Номер операнду.

На лістингу 3.3 показано процес розбору даного файлу лексиром.

Лістинг 3.3 Class Reader.cs.

```
class Reader
    {
        public List<LineCSV> lines = new List<LineCSV>();
        string path = @"code.csv";
        public Reader()
        {

        }

        public void GetStrokes()
        {
            var strokes = File.ReadAllLines(path);
            foreach (var c in strokes)
            {
                string[] splited = c.Split(',');
                string w = splited[6].Substring(0,
                lines.Add(line);
            }
      }
```

2.3 Формування графу

Після формування дерева команд, наступним кроком йду формування графу для роботи системи. За даний процес відповідає набір класів:

- LineCSV.cs;
- EvolveToGraph.cs;

Класс LineCSV відповідає за формування та збереження даних та виступає нодою у дереві, лістинг 3.4.

Лістинг 3.4 класс LineCSV.cs.

```
public class LineCSV
    {
        public int level;
        public int num;
        public string name;
        public int time;
        public int operand1;
        public int operand2;
        public int next;
        public string param_pos;
        public LineCSV(int l, int n, string nam, int t,
int o1, int o2, int nex, string pp)
        {
            level = l;
            num = n;
            name = nam;
            time = t;
            operand1 = o1;
            operand2 = o2;
            next = nex;
            param_pos = pp;
        }
    }
```

Поля обету типу LineCSV відповідають формату згідно строфи описаної у пункті 3.2.

Класс EvolveToGraph дозволяє перетворити на дані з якими має працювати система, тобто виділяє актори та операнди.

2.4 Машина станів

Основою роботи даного емулятор є машина станів, за допомогою її переключення відбуваються дії у системі, для цього було окремо реалізовано логіку кожного елементу системи, що приймають участь у роботі схеми зображеної на рисунку 2.3. Отже список елементів з власною логікою, які приймають участь у роботі системи та класи які їм відповідають:

- ActorMemoryVisualElement – відповідає памяті акторів сисеми.
- BufferCommandMemomoryVisaulElement – відповідає БПК, буфер памяті команд системи.

- BufferMemoryDataVisualElement – відповідає буферній памяті;
- ComutatorVisualElement – відповідає комутатору даних;
- InputVisualElement – відповідає елементу введення;
- OutputVisaulElement – відповідає елементу виведення;
- OperandVisualElement – відповідає елементу памяті операндів;
- ProcessorVisaulElement - відповідає елементу обчислювального блоку;
- RAVisaulElement – відповідає регістру RA;
- RDVisaulElement – відповідає регістру RD;
- TimerVisualElement – відповідає елементу таймер.

Скелет проекту з даними класами зображено на рисунку 2.5.

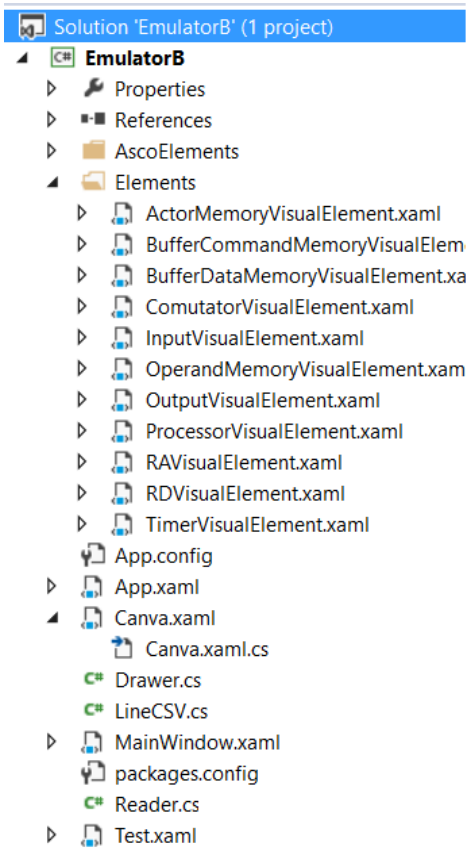


Рис.2.5 Скелет проекту

Завдяки написаному алгоритму та реалізованим окремо кожного елементу системи, машина станів перемикає положення елементів та заповнює їх відповідними значеннями, переходячи від одного стану до іншого так наприклад на рисунку 2.6, зображено передача даних до обчислювального модуля.

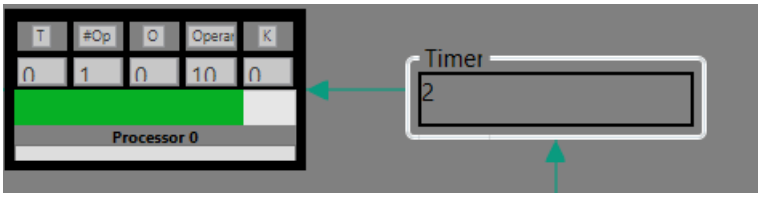


Рис.2.6. Обчислювальний модуль емулятору

Як видно на малюнку 2.6, до обчислювального блоку прикріплено таймер, який містить в собі час виконання команди.

Цьому процесу слід уділити окрему увагу, адже емулятор не знає реального часу виконання тому для його справної роботи додається окремий файл з важелями системи, скільки яка операція займає часу на виконання. Лістинг даного набору даних подається у форматі csv, його лістинг 2.7 [24].

Лістинг 2.7 Важіль команд

```
cmd1 0.9
cmd2 0.01
cmd3 0.2
cmd4 0.1
cmd5 0.2
cmd6 0.5
```

Так на приклад команда cmd1 займає 0.9 умовний одиниць на виконання.

Далі слід приділити окрему увагу реалізації взаємодії машини станів з іншими елементами системи, яка сама по собі виступає блоком управління у реальній системі. На Лістингу 2.8 зображено основний модуль машини станів .

Лістинг 2.8 Метод класу Machine MachineTakt

```
private void MachineTackt()
{
      Clear(_)
      checkProcessors();
       switch (state)
       {
           case 0: checkInputs();
           case 1: setBuffer();
           case 2: SetRaRd();
                   SetActorOperand();
           case 3: SetBufferMmeory();
           case 4: setTimers();
                   SetProc();

           case 5: takt++;
           default: break;
       }

          globaltakt++;
 }
```

В лістингу 2.8 зображено основну конструкцію switch, що перемикає стани системи перенаправляючи дію на даному такті на певну функцію, які виконують відповідні дії на елементами системи. Алгоритм дій машини станів зображено на рисунку 2.9.

Рис 2.9. Алгоритм роботи машини станів

Описання методів, класу Machine:

- checkInputs() – метод, що забезпечую зчитування даних з входів у об’єкти даних;
- setBuffer() – метод відповідає за встановлення буферної пам’яті з актульними даними;
- SetActorOperand() – метод відповідає за встановлення пам’яті акторів та операндів;
- SetBufferMemory() – метод відповідає за встановлення буферної пам’яті команд;
- SetTimer() – метод відповідає за встановлення таймеру з отриманих даних;
- SetProc() – метод відповідає за запуск процесору;
- Clear() – очищає комірки.

На рисунку 2.10 зображена загальна діаграма класів проекту.

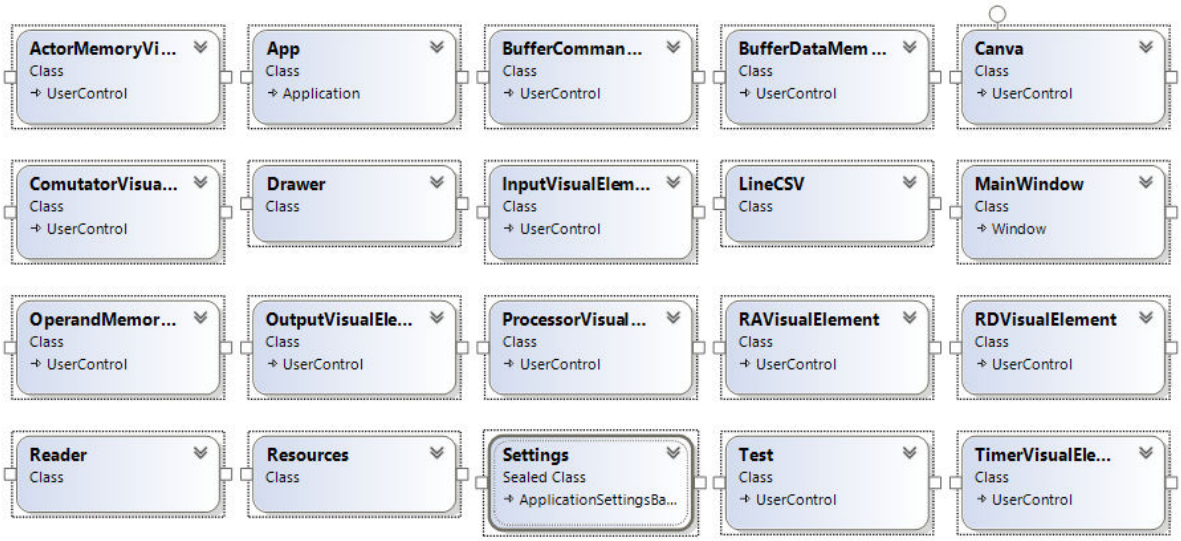


Рис.2.10 Діаграма класів проекту

2.5 Модель відображення та інтерфейс користувача

Наступним етапом реалізації є створення відображення роботи емулятору, вона повинна відображати схему роботи архітектуру та

надавати користувачу можливість переключати такти системи, а також відмикати та вмикати обчислювальні блоки системи для проведення відповідних аналізів та тестів. А також виводити фінальний результат відповідного до вхідного файлу. Повне відображення користувача зображено на рисунку 2.10.

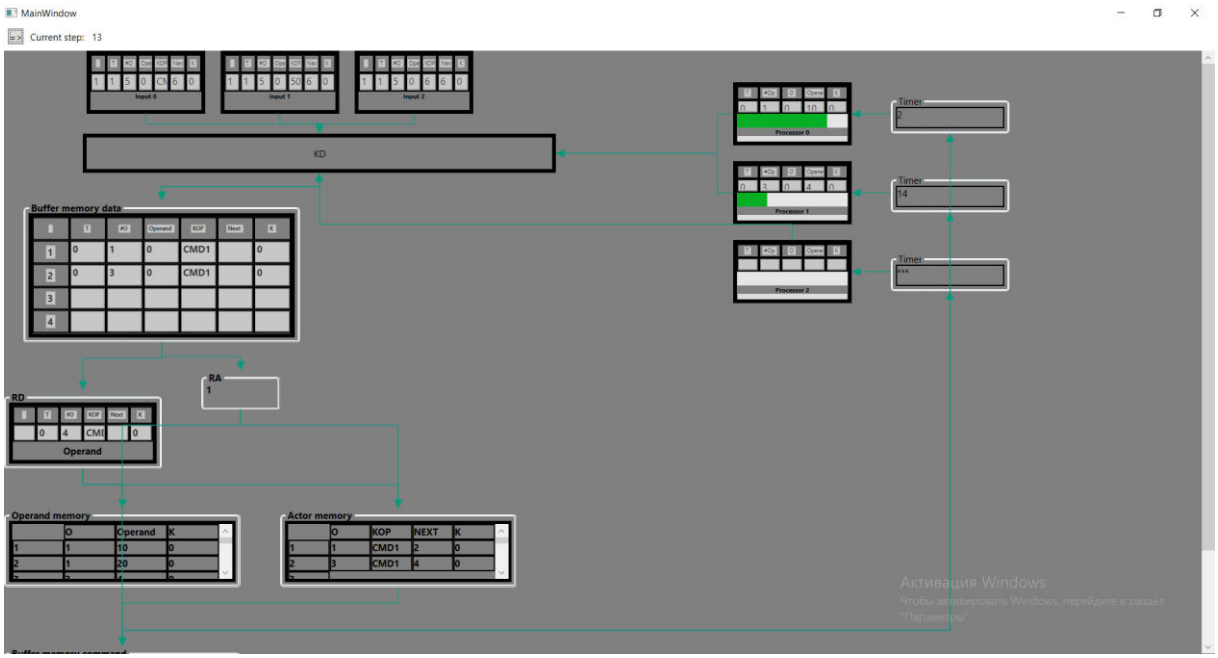


Рис.2.10. Емулятор роботи архітектури

Для керування системою користувач має натискати кнопку руху системи, також біля неї він може спостерігати на якому такті роботи знаходиться.

Також користувач може спостерігати за рухом процесорів, завантажень , прикріплених до кожного обчислювального блоку, також вони мають порядкові номера. Приклад зображено на рисунку 2.11.

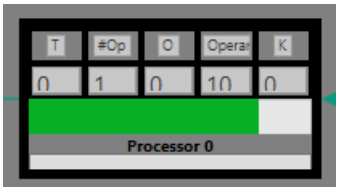


Рис.2.11 Ввідображення обчислювального модуля системи

Також користувач може вручну задати кількість процесорів системи, а також кількість входів та виходів.

Отже після виконання необхідної кількості тактів користувач отримає на віповідні виходи результат його вхідного файлу. Приклад зображено на рисунку 2.12.

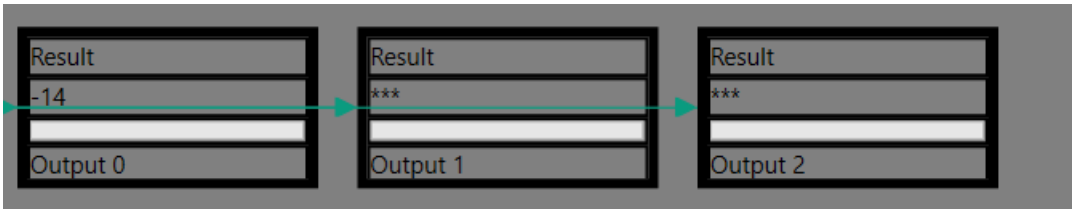


Рис.2.12 Відображення результатів

2.5. Висновки

В даному розділі були повністю описані деталі процесу розробки програмного забезпечення, що дозволяє емулювату роботу архітектурі, що була розглянута у попередніх розділах.

Відповідно було розроблено програмне забезпечення, що дозволить подальшому протестувати роботу архітектури порівняно з іншими архітектурами.

Також було наведено процес роботи і описано процес створення ПЗ, описано користувацький інтерфейс, а також систему подачі команд системі керування потоками даних. Було наведено структуру проекту та розписано процес роботи додатку у випадках відмови. ПЗ має можливість емулювати відключення ОМ та подальше відновлення роботи, ця функція корисна при створенні навантажувальних тестів на систему.

Отже можна зробити висновок, що створене програмне забезпечення повністю емулює роботу архітектури, що була описана у попередніх розділах також на відміну від сучасних аналогів проектування швидко та зручно провести тестування системи на будь-яких вхідних даних не поглиблюючись у знання апаратної мови.

4. ПОРІВНЯЛЬНИЙ АНАЛІЗ НА ОСНОВІ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЕМУЛЯЦІЇ РОБОТИ СИСТЕМ ДЛЯ ЗНАХОДЖЕННЯ КОРЕНЯ N-ГО СТУПЕНЮ НА ОСНОВІ ШВИДКОСХОДЯЩОЇСЯ ІТЕРАЦІЙНОЇ ФОРМУЛИ З ЗАДАНОЮ ТОЧНІСТЮ

4.1 Оцінка ефективності розробленої архітектури за допомогою розробленого програмного забезпечення

Для оцінки ефективності було проведено моделювання на основі використання розробленої імітаційної моделі, емулятор виконував алгоритм знаходження кореня n-го ступеню на основі ітераційної формули з задано точністю. Таким чином, моделювання дає можливість нам порівняти швидкодію наступних архітектур обчислювальних систем:

- Розроблена архітектура з використанням вільного адресного простору (рис. 4.8);
- Обчислювальна система керування потоками даних з ВАД (вільний адресний доступ) першого варіанту (рис. 4.9);
- Обчислювальна система керування потоками даних з ВАД (вільний адресний доступ) другого варіанту (рис. 4.10).

Моделювання роботи потокової системи проводиться декілька разів, при різних умовах:

- Усі процесори працюють справно;
- Поступове відключення всіх процесорів без подальшого відновлення їх роботи;
- Поступове відключення усіх процесорів з подальшим відновленням їх роботи.

Відповідно формула ітераційна формула для знаходження кореня n-го порядку, що буде виконуватися емулятором:

$$x_{k+1} = \frac{1}{2} * (x_k + \frac{A}{x_k}) \tag{4.1}$$

Як зазначалось у попередніх розділах, в основі роботи системи лежить граф алгоритм у якості вершин якого виступають операції, вузли дерева – це операції. Так як для достовірної симуляції, необхідно врахувати різні операції та ввести для них так звані важілі, важіль – це умовна одиниця за яку обчислювальний модуль виконую операції, що відноситься до вершини даного важіля. В алгоритмі знаходження кореню n-го порядку на основі ітераційної формули з задано точністю, використовуються операції додавання, множення ділення. Розподіл на довгі та короткі операції, до коротких відноситься операція додавання, а до довгих операцій множення та ділення [26]. Для цього візьмемо, що обчислювальний модуль симуляційної моделі виконує короткі операції за 10 умовних тактів, а довгі операції за 20 умовних тактів. Для симуляції візьмемо частковий випадок даного алгоритму, а саме методу Ньютона, метод для знаходження нулів функції або так званий метод дотичних:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{4.2}$$

Отже в заданій симуляції будемо шукати нулі функції яка має вигляд:

$$f(x) = x^n - A \tag{4.3}$$

А відповідно її похідна:

$$f'(x) = nx^{n-1} \tag{4.4}$$

Таким чином рішення зводиться до класичного знаходження кореня n-го порядку:

$$x^n = A \tag{4.5}$$

Для емулятору було прийнято, що:

- Підрадикальне число A = 100;
- Ступінь n = 5;
- Точність 0.001.

Для таких даних кількість операцій множення 20, кількість операцій ділення 10, кількість операцій суми 5. Граф алгоритм для даного виразу зображено на рисунку 4.14.

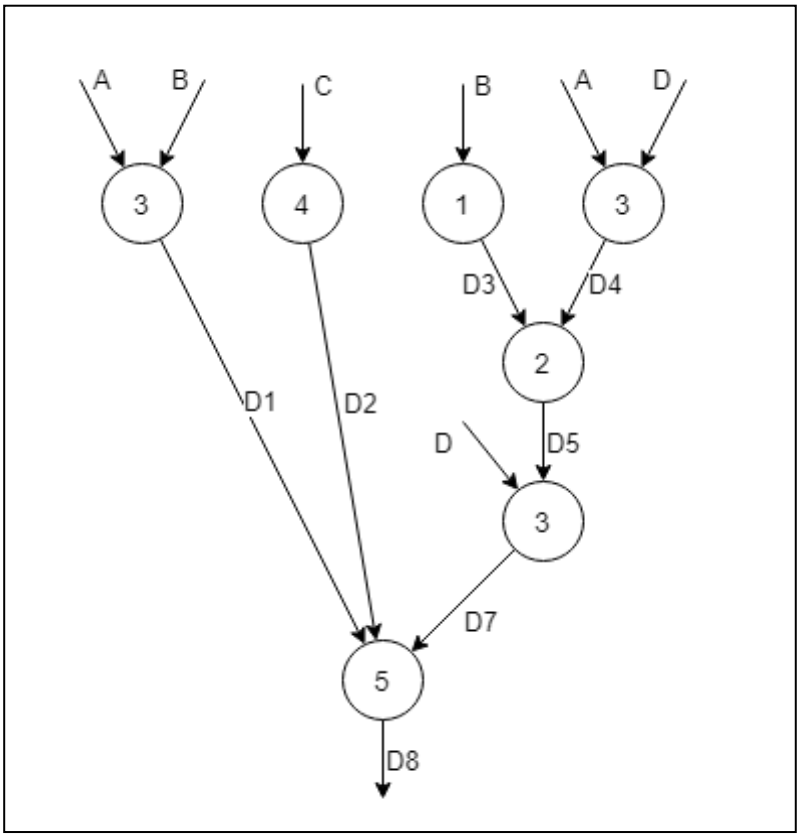


Рисунок 4.14. Граф алгоритм

Для емулятору було виставлено кількість процесорів від одного до п’ятнадцяти.

Результати роботи емулятору занесені до таблиць 4.1, 4.2, 4.3.

Таблиця 4.1 – Результати роботи программного емулятору, при умові справності усіх процесорів.

Варіант архітектури	Процесори	Кількість умовних тактів
Розроблена архітектура	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1541 756 351 344 300 291 290 287 255 251 241 239 236 231 202
Архітектура з ВАД варіант перший	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1641 856 451 450 440 398 390 387 355 351 341 339 336 331 302
Архітектура з ВАД варіант другий	1 2 3 4 5 6 7 8 9	1600 816 541 490 460 440 390 387 365

Варіант архітектури	Процесори	Кількість умовних тактів
	10 11 12 13 14 15	361 331 329 326 291 251

Проаналізувавши дані емулятору з таблиці 4.1, можна зробити висновок, що система другого варіанту архітектури та система першого варіанту архітектури, дають схожі результати, проте система з другим варіантом є більш успішно, найкращі показники має з невеликим відривом від другого типу розроблена архітектура, отже можна зробити висновок, що спроектована архітектура для рішення поставленої задачі підходить за умов справності всіх процесорів. Також можна побачити, різке спадання продуктивності при збільшенні процесорів після 3-5, це повязано, з набором операцій в певний момент, тобто на деякому етапі рішення задачі, кількість процесорів стає занадто великою і така кількість не приносить додаткової користі, тому що усі процесори і так задіяні в роботі системи.

В таблиці 4.2 наведені результати роботи емулятор для рішення тієї самої задачі, про те в даному випадку, процесори не додавалися поступово, а навпаки відключалися після деякого часу роботи системи, це дає змогу прослідити за поведінкою системи та прослідкувати на скільки саме умовних тактів обрахунок сповільниться за умов відключення деякої кількості процесорів, на різних архітектурах. Відключення відбувається поступово в ході роботи. Слід зауважити, що процесори відключаються кожні 20 тактів роботи системи.

Таблиця 4.2 – Результати роботи програмного емулятору за умов поступового відключення процесорів.

Варіант архітектури	Кількість відімкнутих процесорів	Кількість умовних тактів
Розроблена архітектура	1	231
	2	291
	3	350
	4	370
	5	391
	6	402
	7	498
	8	551
	9	591
	10	670
	11	801
	12	891
	13	1002
	14	1541
Архітектура з ВАД перший варіант	1	315
	2	320
	3	351
	4	380
	5	391
	6	412
	7	468
	8	521
	9	601
	10	680
	11	811
	12	901
	13	1102
	14	1641
Архітектура з ВАД другий варіант	1	251
	2	300
	3	321
	4	330
	5	361
	6	452
	7	468
	8	531
	9	601
	10	680
	11	711
	12	801
	13	1202
	14	1591

Як видно з табличних даних 4.2, при робота системи при поступовій відмові процесорів не відповідає відповідним показникам при додаванні процесорів, наприклад при роботі 3 процесорів у спокійному режимі з таблиці 4.1 ми бачимо 351 такт тоді, як в системі при поступовій відмові таблиця 4.2 350, дана різниця охарактеризована тим, що на віднову роботи системи необхідно витратити певний набір тактів для формування нової команди і відправлення її на новий обчислювальний модуль, як виявилося система є надійною і при відмові поступовій відмові процесорів продовжувала роботу з невеликими затримками порівняно з роботою у стані справності.

У таблицю 4.3 занесено дані при умовах, що частина процесорів відновлювалася після їх вимикання таким чином емулятор дозволяє проаналізувати ситуацію з відновленням роботи системи. Слід зазначити, процесор включались та виключались з інтервалом 20 тактів. Також для даного тесту з початку всі процесори були відімкнуті.

Таблиця 4.3 – Результати роботи программного емулятору за умов поступового відновлення роботи процесорів.

Варіант архітектури	Кількість відімкнутих процесорів	Кількість процесорів, відновили роботу	Кількість умовних тактів
Розроблена архітектура	0	15	211
	2	13	351
	3	12	251
	5	10	551
	10	5	871
	11	4	900
	12	3	931
	13	2	1014
	2	13	356
	3	12	771
	0	15	217
	1	14	232
	4	11	351
	7	8	651
	1	14	311

Варіант архітектури	Кількість відімкнутих процесорів	Кількість процесорів, відновили роботу	Кількість умовних тактів
Архітектура з ВАД перший варіант	0	15	251
	2	13	331
	3	12	261
	5	10	651
	10	5	811
	11	4	910
	12	3	921
	13	2	1114
	2	13	366
	3	12	871
	0	15	317
	1	14	222
	4	11	321
	7	8	631
	1	14	301
Архітектура з ВАД другий варіант	0	15	311
	2	13	321
	3	12	251
	5	10	561
	10	5	871
	11	4	910
	12	3	941
	13	2	1214
	2	13	346
	3	12	771
	0	15	317
	1	14	222
	4	11	351
	7	8	611
	1	14	411

Результати емулятору було зібрано у таблиці 4.4 для аналізу у відсотках, для порівняння різних архітектури у різних ситуаціях з розробленою архітектурою.

Таблиця 4.4 – Порівняльний аналіз роботи програмного емулятору для різних систем з розробленою архітектурою.

Умови роботи системи	Порівняння з архітектурою ВАД перший (%)	Порівняння з архітектурою ВАД другий(%)
Справна робота	0.3 0.2 1 1.2 0.1 0.2 0.7 0.12 0.98 1.2 0.13 0.17 0.67 0.9 0.87	0.2 0.12 0.3 1 0.11 0.12 0.17 0.12 0.78 1.02 0.23 0.17 0.77 0.19 0.37
Робота системи з поступовим відключенням обчислювальних блоків	1.13 1.2 0.13 0.2 1.1 0.2 0.17 0.32 0.18 1.12 0.33 0.77 0.17 0.19 1.01	0.3 0.2 1 1.2 0.1 0.2 0.17 0.22 1.08 1.2 0.1 0.7 0.6 0.3 0.47
Робота системи з поступовим відновленням роботи	1.3 2.2 1.2 1.2 1.1 2.2 2.7 2.12 1.98 2.2 1.13	1.3 1.2 2.1 3.2 4.1 2.2 1.7 1.12 2.98 3.2 1.13

Умови роботи системи	Порівняння з архітектурою ВАД перший (%)	Порівняння з архітектурою ВАД другий(%)
	2.17 1.67 3.9 1.87	2.17 3.67 1.9 2.87

Усі результати роботи емулятора були зібрані у таблицях 4.1, 4.2, 4.3, 4.4. З огляду на отримані результати можна зробити висновок, що розроблена архітектура дає значний виграш у ситуаціях з відмова процесорів, незначний у випадку справності роботи системи, а також незначний порівняно з другим варіантом системи у випадку відновлення роботи процесорів, так значний виграш порівняно з першим варіантом архітектури. Для зручності, збірка виведено на рисунках 4.5, 4.6, 4.7, 4.8.

На рисунку 4.5 – зображена діаграма результатів роботи програмного емулятору за умов справності всіх процесорів для трьох архітектур.

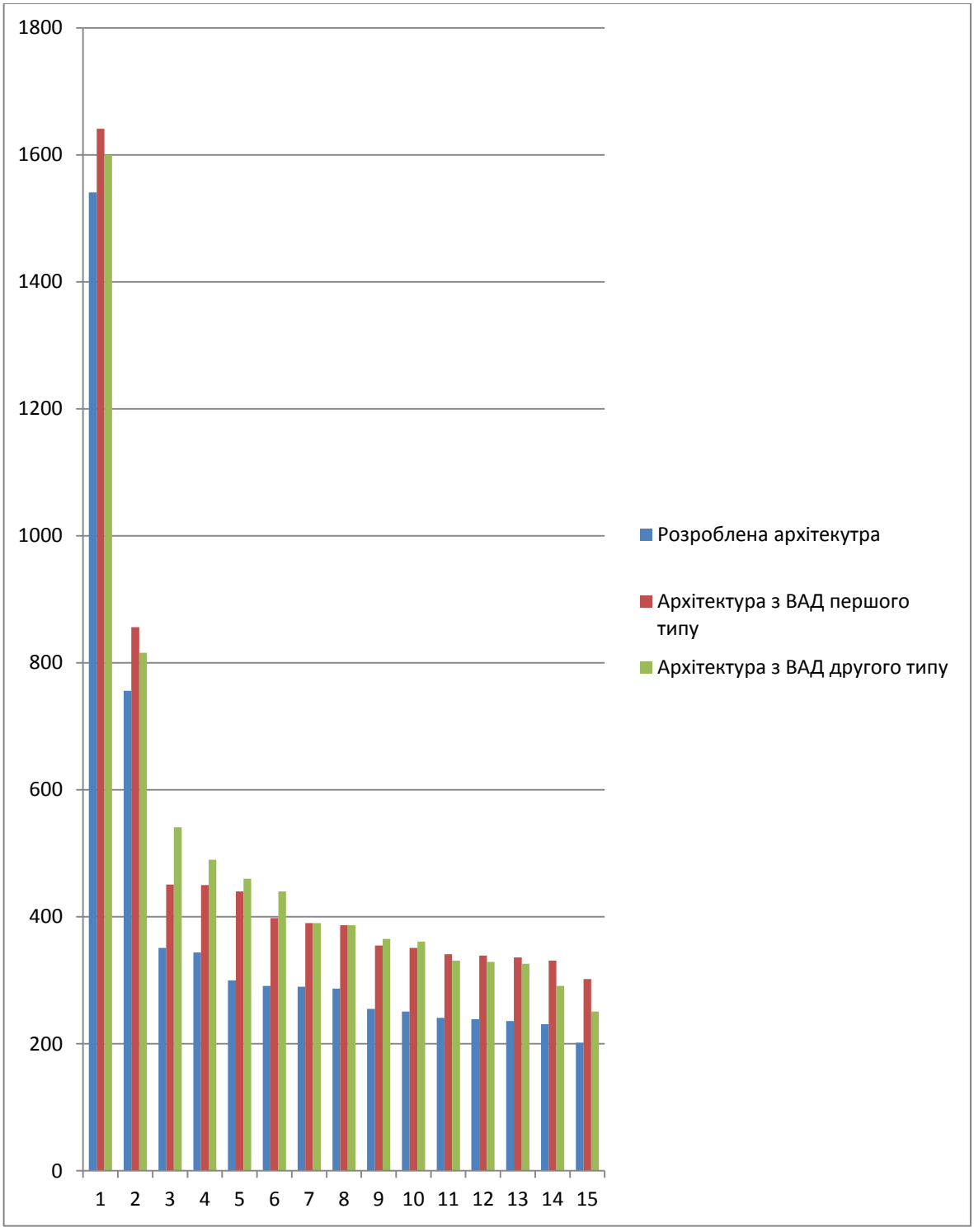


Рисунок 4.5. Діаграма результатів програмного емулятору при справності всіх процесорів

Рисунок 4.6 – Діаграма результатів роботи програмного емулятору за умов поступової відмови обчислювальних модулів для трьох архітектур.

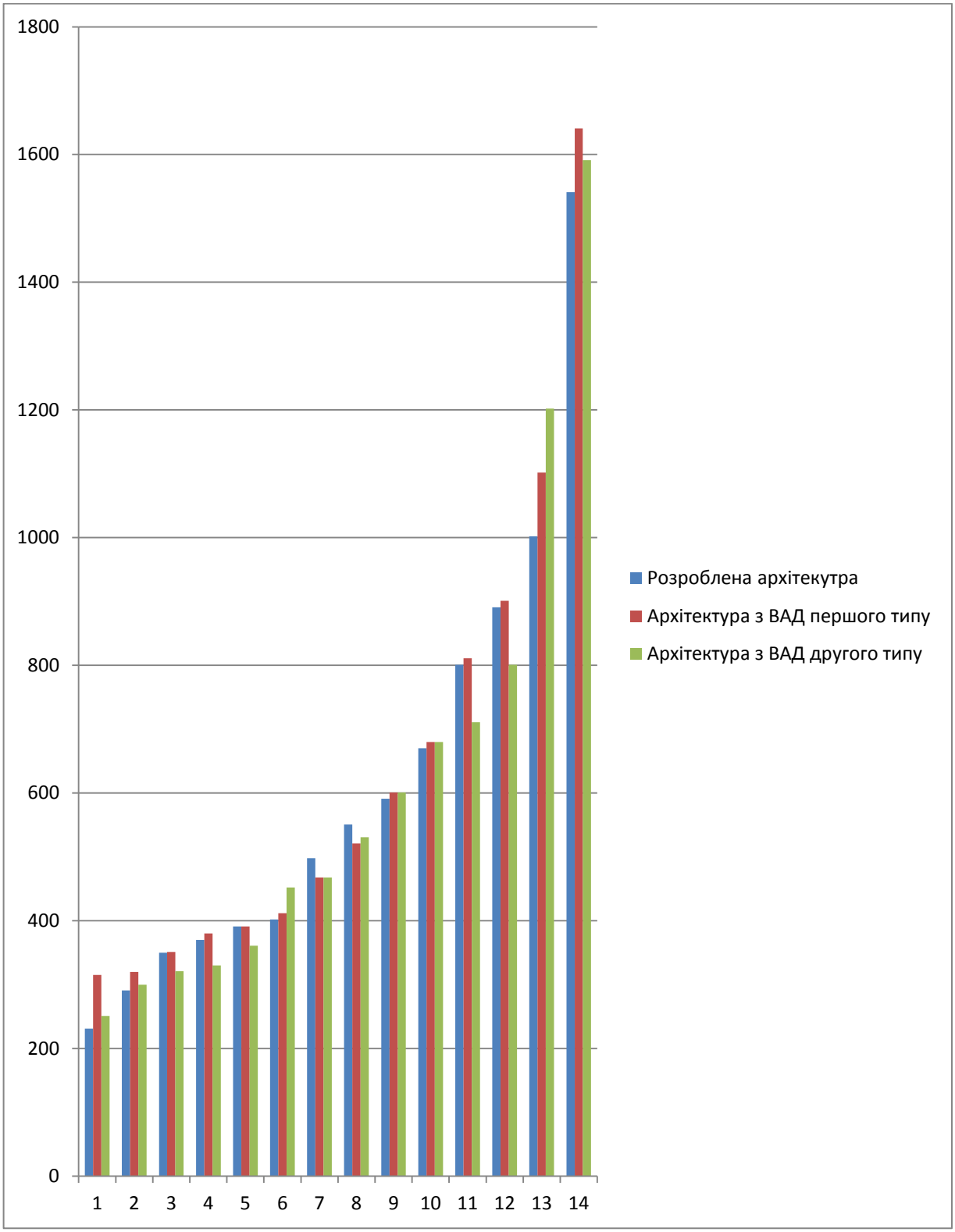


Рисунок 4.6 – Діаграма результатів програмного емулятору при відмові процесорів

Рисунок 4.7 – Діаграма результатів роботи програмного емулятору за умов поступової відмови обчислювальних модулів та подальшої віднови роботи для трьох архітектур.

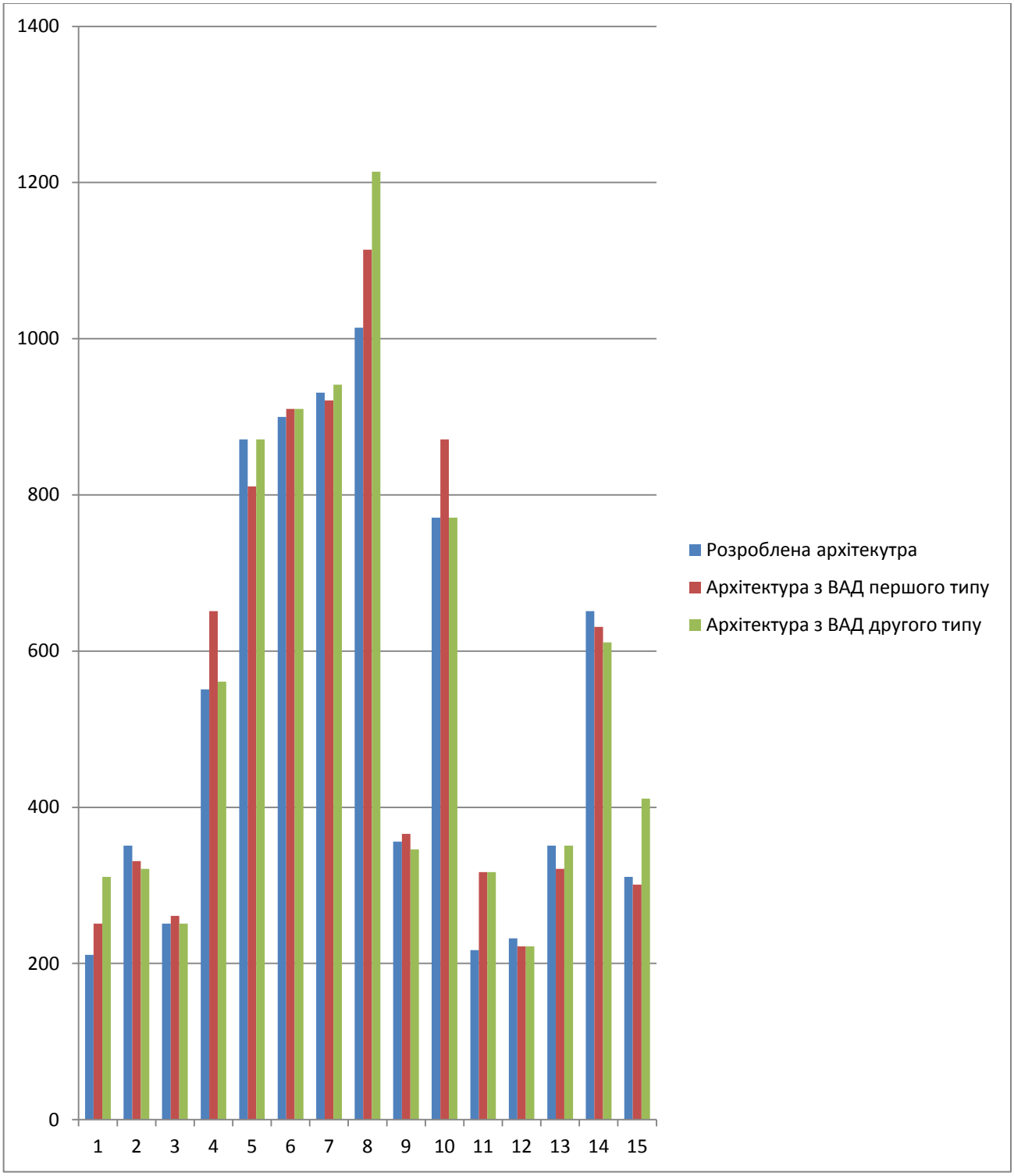


Рисунок 4.7 – Діаграма результатів програмного емулятору при відмові процесорів та подальшій віднові роботи

З наведених вище діаграм можна зробити перший висновок про ефективність і продуктивності розробленої системи. Так як моделювання алгоритмів проводилося в умовних одиницях часу, які не враховують особливості реалізації окремих вузлів тієї чи іншої системи, то ми можемо лише констатувати той факт, що при збільшенні числа обчислювальних блоків у всіх системах час виконання зменшується, але лише до певної межі насичення. Значення цієї межі насичення залежить в першу чергу від алгоритму програми, яка виконується в системі, а саме від ступеня паралелізму цієї програми. Найбільш істотний стрибок продуктивності, як видно з результатів моделювання, відбувається при переході від одно до двопроцесорним системам.

Також можна зробити висновок, що розроблена система ефективно оброблює ситуації виходу зі строю певних обчислювальних модулів, а також подальшого її відновлення, як видно з таблиці 4.4, найкраща продуктивність при відновленні роботи декількох обчислювальних модулів, в такому випадку час витрачений на обробку операцій, що мали виконуватись на несправних модулях є найкращою у порівняння з іншими системами з ВАД.

4.2 Аналіз ефективності розробленої відмовостійкої архітектури з урахування особливостей роботи її окремих компонентів

Як зазначалося раніше, система керування потоками даних складається з набору елементів в основі майже кожного лежить процес запам’ятовування даних та подальшого запису або зчитування з них, в даному випадку ми маємо систему формування команд та обчислювальні елементи, при кожному такті проходять операції передачі даних тощо. В емуляторі в якості часу використовуються умовні одиниці, такти, усі дії системи працюють згідно даним тактам, в першу чергу це спілкування різним блоків систем один з одним. Такими вузлами в першу чергу є блоки

пам’яті. Як відомо пам'ять може бути представлена у вигляді звичайного адресного простору, буферної пам’яті або асоціативної пам’яті, їх різниця полягає у різних алгоритмах запису або зачитування, а також використовується у різних архітектурних рішеннях, слід нагадати, що в розробленій системі використовується вільний адресний простір, також слід зазначити, що звичайні ПЛІС включають в себе завжди блоки вільного адресного простору, через їх просту реалізацію, коли асоціативну та буферну пам'ять необхідно налаштовувати додатково. В даному розділі буде проведено аналіз архітектури, що були розглянуті у попередньому розділі, а також аналіз окремих елементів таких як блоки пам’яті, та проаналізовано процес роботи системи з запам’ятовуючими пристроями [27].

На рисунках 4.5, 4.6 и 4.7 наведені схеми з якими було порівняння, а також розроблена схема.

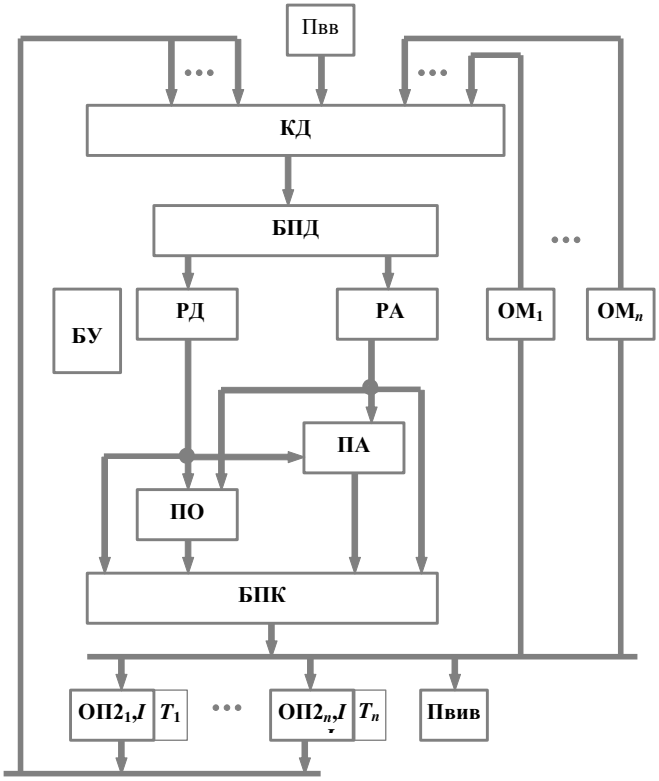


Рисунок 4.8 – Розроблена структурна схема відмовостійкої системи керування потоками даних з використанням вільного адресного простору

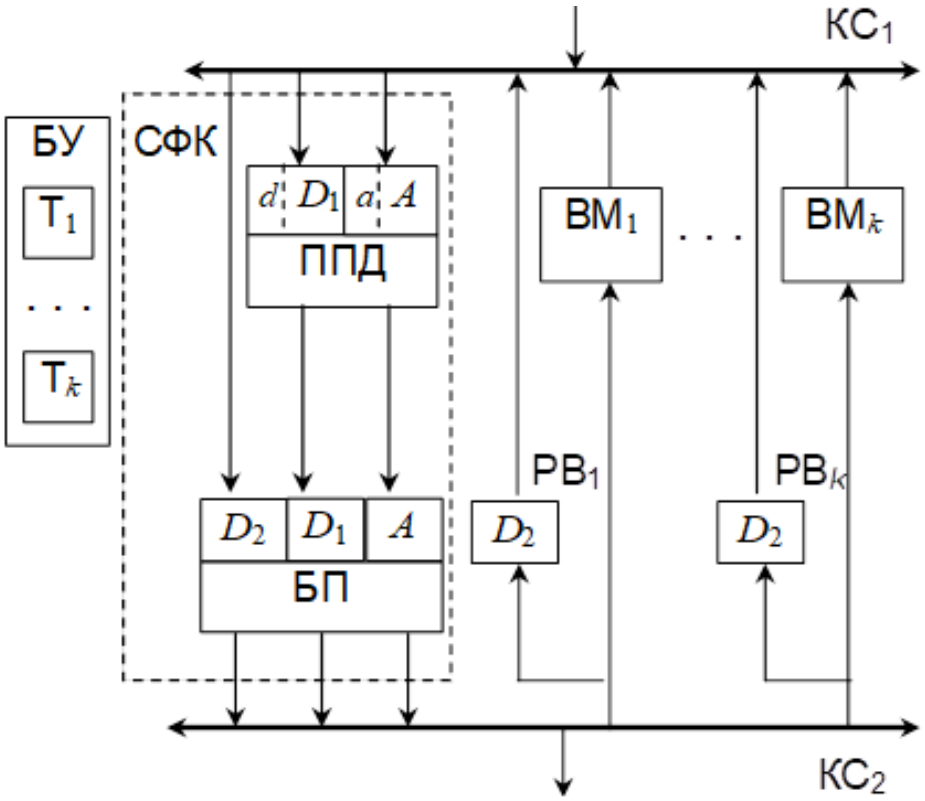


Рисунок 4.9 – Структурна схема системи керування потоками даних з ВАД (вільний адресний доступ) першого варіанту

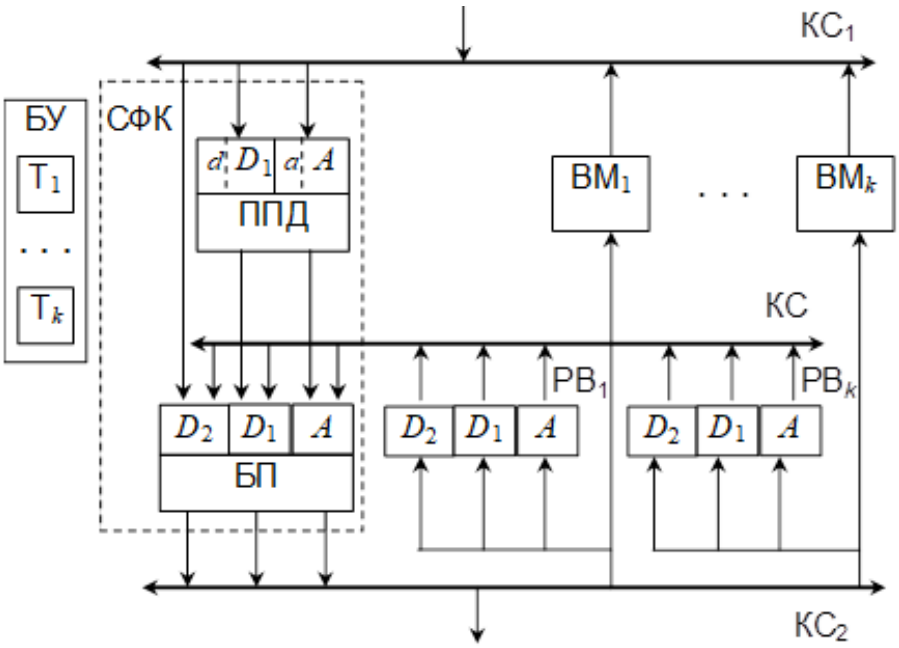


Рисунок 4.10. Структурнаа схема системи керування потоками даних з ВАД (вільний адресний доступ) другого варіанту

Розглянемо окремо усі три варіанти архітектури з вільним адресним доступом.

1.Розроблена архітектура рис 4.8.

СПД містить пристрої введення (Пвв) та виведення (Пвив), обчислювальні модулі ОМ, блок управління (БУ). Буферна пам'ять даних (БПД) і команд (БПК), а також пам'ять операндів (ПО), акторів (ПА), регістри адреси (РА) і даних (РД) утворюють СФК. Два додаткових регістра призначені для зберігання другого операнда із номером команди та кодом таймера (ОП2i, I; Ti).

Розглянемо роботу системи, коли відсутні відмови ОМ. Актори і дані через Пвв і КД аписуються в буфер даних БПД. Імена I_i об’єктів (1) та (2) використовуються у якості адрес модулів пам’яті ПО та ПА. Через РА адресується комірка пам’яті для всіх складових команди. Актор через РД записується в ПА. Дане, що потрапляє в систему першим через РД записується в ПО. Друге дане разом з актором та першим операндом потрапляє відразу в БПК, де накопичуються готові для виконання команди.

Вільний ОМ зчитує команду з буфера БПК та виконує її відповідно зі значенням актора. Результат потрапляє знов в СФК, де приймає участь в формуванні нової команди. Через Пвив дані можуть видаватися із системи.

В системі відбувається контроль правильності виконання команди методом контролю часових інтервалів. Це відбувається наступним чином.

В склад актора (1) додається поле T_i для ініціалізації таймера. Для кожної команди призначається інтервал часу, який гарантує виконання команди, якщо ОМ працює вірно. Відомо, що час виконання команд може суттєво відрізнятися. Наприклад, множення без прискорення майже в m разів виконується довше ніж додавання або логічні операції (m – розрядність операндів). Це вигідно відрізняє підхід до контролю по

відношенню до системи прототипу, де інтервал завжди дорівнює одному максимальному значенню.

Водночас із записом команди в ОМj для її виконання в додатковий регістр з таким самим номером j записуються поля: другий операнд, що прийшов в систему останнім, номер комірки пам’яті в ПА та ПО і код ініціалізації таймера, в якості якого використовується частина додаткового регістру. Таймер запускається з початком виконання команди в ОМ. Після одержання результату операції ОМ скидає свій таймер в додатковому регістрі. Якщо операція не закінчилась у визначений проміжок часу, спрацьовує таймер. В цьому випадку другий операнд разом із своєю адресою по сигналу таймера потрапляє в БПД і знов формує команду, бо перший операнд і актор зберігаються за даною адресою в СФК.

ОМ, що відмовив, блокується. Команда природним чином потрапляє в інший ОМ, де виконується. Дані для системи готуються без врахування кількості ОМ в системі. Тому обчислення можуть продовжуватись поки є ОМ, що працюють. Відновлення системи не потребує комутації модулів, бо всі вони об’єднані одним комунікаційним середовищем.

2.Архітектура з ВАД першого варіанту рис 4.9.

Розглянемо особливості активації команд для такої системи. Команда починає формуватися в комірках адресного простору, два розряду яких (d і a) є ознаками, що показують наявність в ВАД одного операнда D1 і актора. Запис зазначених об'єктів для і-го команди виконується за адресою I. При надходженні з КС1 другого операнда D2 для даної команди, він записується безпосередньо в відповідними розряди регістра буферної пам'яті (БП) типу FIFO, куди одночасно з ВАД переписуються A і D1 [29, 30].

Таким чином, в БП формується готова для виконання команда, яка потім через КС2 передається у вільний ВМ. При цьому вміст відповідної комірки ВАД зберігається для забезпечення повторного формування

команди в разі відмови ОМ. Одночасно з надходженням команди в ОМ її частина, яка не з-яке береже в ВАД, записується в регістр тимчасового зберігання на вхідний шині відповідного обчислювача. При успішному виконанні операції, тобто коли час її виконання не перевищило ліміт часу виконання команди в ОМ встановлений відповідним таймером Т блоку управління БУ, результат операції надходить через КС1 в ВАД за адресою, а вміст відповідної комірки ВАД за адресою обнуляється[30].

Якщо при виконанні операції відбулася відмова ОМ, тобто ліміт часу виконання операції, заданий відповідним Т, був перевищений, ОМ вважається несправним і блокується (відключається від КС1 і КС2), а значення з РВ знову передається в СФК. Оскільки значення А і D загублені були (відповідна комірка ВАД була змінена), то команда повторно записується в БП, що дає можливість реалізувати наступну успішну спробу виконання команди в справному ОМ [31].

До недоліків слід віднести необхідність зберігання компонентів команди в осередках ВАД до повного завершення виконання даної команди, в тому числі, з урахуванням часу реконфігурації системи при відмові ОМ і повторного виконання команди. Це уповільнює обчислення, якщо відповідну комірку пам'яті необхідно використовувати для інших команд, наприклад, для чергової ітерації або черговий реалізації алгоритму в конвеєрному режимі.

3.Архітеутра ВАД другого варіанту рис 4.10.

Для усунення зазначеного недоліку системи з ВАД першого варіанту пропонується в РВ записувати не тільки операнд, а всю команду. При цьому принцип формування команди залишається незмінним. Після запису даних в БП звільняється місце в ВАД для запису таких даних. За аналогією з попереднім варіантом, при відмові ОМ він буде блокований, а дані з РВ будуть передані назад безпосередньо в БП, що дасть можливість реалізувати наступну спробу виконання команди [32, 33].

Таким чином, при такому підході ресурси системи використовуються більш ефективно. Немає необхідності зберігати фрагменти комірок в СФК. Комірки ВАД можуть використовуватись для інших алгоритмів або інших ітерацій. Це в свою чергу дозволяє поєднувати обчислювальні процеси не тільки на рівні команд, а й на рівні алгоритмів, що забезпечує скорочення часу перетворення інформації.

Проте, з точки зору витрат часу на реконфігурацію системи, при відмові ОМ даний підхід, як і розглянуті варіанти, є недостатньо ефективним. Дійсно, час очікування спрацьовування таймера може бути набагато більше, ніж час, реально необхідне для виконання команд. Це вносить додаткову часову затримку при реконфігурації системи [33].

З розглянутих підходів видно, що перша архітектура є оптимальним рішенням, адже показано, що запропонований підхід забезпечує підвищення швидкості реконфігурації системи при відмові обчислювальних модулів. Це пов’язано із більш ефективною методикою контролю часових інтервалів. Для кожної команди визначається індивідуальний інтервал контролю. У відомих системах використовують максимальний інтервал для всіх команд[34].

Реконфігурація системи при відмові обчислювального модулю реалізується в основному на апаратному рівні і зводиться до відключення модулю, що відмовив.

Таким чином, у порівнянні з відомими потоковими системами запропонований підхід прискорює час розв’язання задач, що є важливим чинником для обробки даних в реальному часі.

Наступним кроком аналізу є порівняння систем на основі асоціативної пам’яті та систем з вільним адресним доступом. На рис 4.11 зображено логічну структуру звичайного запам’ятовуючого пристрою, на рис 4.12 зображено логічну структуру асоціативного запам’ятовуючого пристрою.

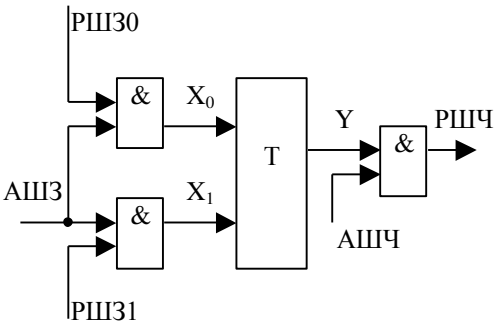


Рисунок 4.11 – Логічна структура звичайного запам’ятовуючого елементу.

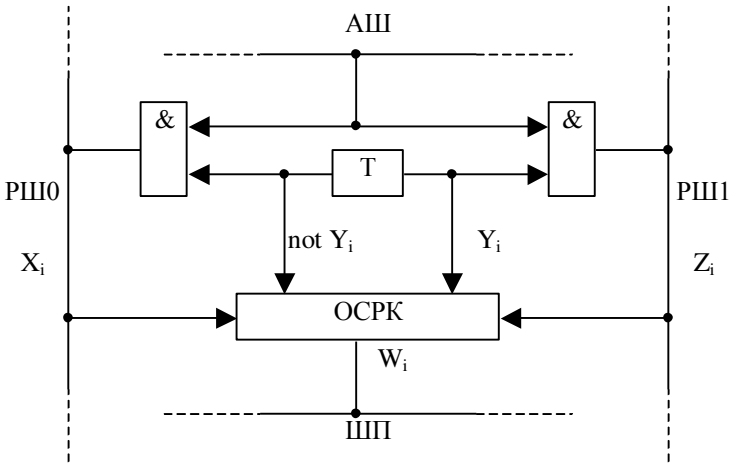


Рисунок 4.12 – Логічна структура запам’ятовуючого елементу з асоціативною пам’яттю.

З рисунків 4.11 та 4.12 видно, що пам'ять асоціативна сама по собі влаштована складніше та вимагає прямого підключення до шин комутації даних, що сповільнює її роботу порівняно з вільною адресацією. Організація буферної пам’яті базується на основі вільної адресації тому можна прийняти, що системи з буферної пам’яттю та системи з ВАД схожі між собою, за типом запису та зчитування даних, також слід зазначити, що ПЛІС завжди мають вистроєні набори вільного адресного простору, тоді як асоціативна пам'ять вимагає додаткових налаштувань.

Базуючись на алгоритмах роботи пам’яті можем зробити заміри робити 2х систем з асоціативною та вільної пам’яттю, приймаємо, що

буферна та вільна пам'ять мають схожі результати. Далі будемо оперувати такими поняттями, як затримка при зверненні до того, чи іншого елементу, яку будемо вимірювати в умовних одиницях.

Таким чином для звернення до комірки адресного простору (буферна пам’ять) ми витратимо час, що рахується за формулою:

$$t = 3 * t1 + t2 + t3$$

Де:

- *t* – загальна затримка;
- *t1*–час роботи тригеру;
- *t2*–час роботи дешифратора;
- *t3* –час роботи логічного елементу &.

Для звернення до комірки асоціативної пам’яті даних:

$$t = 5 * t1 + t2 + t3 + 4 * t4$$

Де:

- *t* – загальна затримка;
- *t1*– час роботи тригеру;
- *t2*– час роботи дешифратора;
- *t3* – час роботи шифратору;
- *t4*– час роботи логічного елементу |.

Вже з наведених формул звернення до різних типів видно, що процес роботи з вільним адресовим простором значно простіший ніж алгоритм звернення до асоціативної пам’яті. Для порівняльного аналізу необхідно спростити формули обрахунку та привести їх до спільного вигляду та спільних змінних, позначимо умовно одиницю затримки *Tz* [35].

Для початку приймаємо, що робота з різними типами логічних елементів займає однаково малий час, тому різницею можна нехтувати, отже *t4 = t3*.

Тоді приймаємо, що:

*t1 = Tz *2;*

*t2 = Tz * k;*

Де: k – складність шифратору.

t3 = Tz * k;

Де: k – складність дешифратору.

Тоді маємо:

*T асоціативної памяті = Tz * (15 + k + 2 *k1 + 6*k2).*

*T вільного адресного простору або буферної памяті = Tz * (7 + k).*

Для прикладу візьмемо адресну сітку розміром 10x10, тобто 100 комірок і врахуємо, що кожен буфер займає 10 комірок тоді маємо, що час звернення для вільного адресного або буферного доступу дорівнює приблизно 12 * *Tz*, в той час як до асоціативної пам’яті 61 * *Tz*.

Як видно з отриманих співвідношень час звернення до асоціативної пам'яті значно вище часу звернення до двох інших видів запам'ятовуючих пристроїв - адресне і буферного. Це ще раз підтверджує наші зауваження з приводу того, що при оцінці продуктивності систем, здійсненої в попередньому підрозділі, слід враховувати різну величину швидкодії різних типів пам'яті. Отже, з урахуванням всього вищесказаного і проведеного моделювання можна констатувати той факт, що обчислювальна система з асоціативної пам'яттю значно поступається по продуктивності системам, де використовуються буферні та адресні пристрої, що запам'ятовують. Таким чином дане порівняння ще раз доводить доцільність використання вільного адресного простору на відміну від асоціативної пам’яті.

4.3 Аналіз надійності обчислювальної системи

Надійність - властивість об'єкта зберігати в часі у встановлених межах значення всіх параметрів, що характеризують здатність виконувати

необхідні функції в заданих режимах і умовах застосування, технічного обслуговування, ремонту, зберігання і транспортування.

Так як розроблена система є відновлюваної, то надійність її компонент характеризується інтенсивністю відмов λ і коефіцієнтом готовності Кг. Також можна порівняти наскільки ефективні виявилися методи підвищення надійності системи, описані в розділі 4 даної роботи.

Розглянемо обчислювальну систему прототип (рис. 4.9).

Інтенсивність відмов величина адитивна і визначається для кожного компонента як:

$$\lambda = \Sigma \lambda i * Ni$$

Де:

- λi - інтенсивність відмов модулів;
- Ni - кількість відповідних компонент.

Для конкретності визначимо на яких елементах і інтегральних схемах може бути побудована розглянута система.

В якості комутатора 2 можна використовувати набір деяких інтегральних схем.

Припустимо для даної схеми маємо наступні характеристики:

$\lambda 0$ = X, тоді сумарна інтенсивність відмов буде дорівнювати $\lambda i * Ni$.

Тоді затримка при відмовах буде дорівнювати $1 / \lambda$.

Тоді коефіцієнт відновлення $1 / (1 + \lambda * (Тз + Тп / 2))$.

Де:

- Тз - час реконфігурації.
- Тп - час перевірки обючилювального модуля.

Як видно з наведених вище формул, чим більший час затримки та час відновлення роботи тим менше значення коефіцієнту відновлення, що є поганим знаком для оцінки надійності системи. Тому задача при проектування архітектури збільшити значення даного показника, що й

було зробоено у розробленій архітектурі (рис 4.8), через зменшення часу реконфігурації загальне значення коєфіцієнту зросте, що напряму показує збільшення надійності системи порівняно з іншими екземплярами [36].

Таким чином, можна зробити висновок, що метод підвищення надійності, обраний нами, дав позитивні результати. І хоча деякі модулі в системі (в даному випадку апаратура підвищення надійності) сама по собі знижує відмовостійкість системи (використатння вільного адресного доступу), але завдяки їй првишается відмовостійкість підсистеми обчислювальних блоків, інтенсивність відмов яких є однією з найбільших у всій системі автоматичного. Звідси також можна зробити висновок, що найбільш відчутний ефект від впровадження такої апаратури буде спостерігатися при використанні в системі блоків пам'яті з найменшою можливою інтенсивністю відмов. Також це видно з роботи емулятору, який демонстурє різке зменшення тактів роботи системи при відмові модулів та подальшому відновленні їх роботи. Емулятор системи дає можливість повністю проаналізувати та порівняти надійності різних систем (таблиця 4.1, 4.2, 4.3).

4.4 Висновки

Отже з огляду на проведений порівняльний аналіз декількох архітектур систем керування потоками даних та різних видів памяті, можна зробити висновок, що відмовостійка розроблена архітектура є отптимальною в чому нам дає переконатися емулятор роботи системи, результати якого демонструють найкращі показники в випадках відмови процесорів та коли всі процесори справно працюють. Відсоток вийграшу в умовних тактах наведено у таблиця 4.1, 4.2, 4.3. Як ми бачимо з розрахунків з пункту 4.2 роботи, вільний адресний простір працюю швидше за асоціативну пам'ять, що також дає виграш у часі роботи. Також розроблений емулятор дозволяє проаналізувати ситуацію при

реконфігурації обчислювальних модулів, як відомо дана система продовжує коректну роботи згідно граф алгоритму після відмови обчислювального блоку, а також після відновлення обчислювального модуля, дану ситуацію емулятор також надає можливість проаналізувати. Емулятор використовує вільну адресацію, адже це більш вигідно і вона працює простіше за асоціативну пам'ять. Через те, що в архітектурі, що реалізовує емулятор рис 4.5, передбачені окремі «блоки» для таймерів це дозволяє зберігати у вигляді окремих мета-даних час для кожної операції і виконуваті її згідно з даним часом, що дозволяє системі адекватно та точно оцінювати відмову обчислювального блоку, яку б операцію він не виконував. Можно зробити висновок, що емулятор дозволяю змоделювати будь-яку ситуацію в роботі системи, з відказами та ремонтом блоків обчислення, та перевіцрити час роботи системи з заданими параметрами.

5. СТАРТАП СКЛАДОВА ЧАСТИНА ПРОЕКТУ

5.1. Опис проблеми та дерево проблем

5.1.1. Анотація проекту.

Проект спрямований на розробку та тестування методу підвищення ефективності роботи паралельних кластерних систем з динамічним розподілом задач. Розробка передбачає винайдення методу ефективного розподілу задач між вузлами кластерної системи. Тестування передбачає створення симуляції кластерної системи, програмними засобами для подальшого використання розробленого методу в межах даної системи.

5.1.2. Опис проблеми, на розв'язання якої спрямований проект.

Розробка методу підвищення ефективності роботи паралельних кластерних систем з динамічним розподілом задач обумовлена тим, що кількість інформації з кожним днем росте, інформація оброблюється циркулює перезаписується, зберігається і все це виконують великі системи, наприклад, як дата-центри, для збереження, чи дата-сайнс корпорації для обробки та виведення статистичних даних, таким чином з'явилася необхідність прискорення подібних процесів взаємодії з даними, так з'явилися паралельні обчислювальні системи, а згодом і проблеми прискорення роботи цих систем, методом динамічної генерації задач для системи. Так як кожна велика система має «пул операцій» які вона може виконувати з даними та певні набори вхідних даних, на виході отримуємо різні результати і на даний момент у сфері прискорення роботи подібних систем є частина з прискоренням процесу підбору таких операцій під вхідні дані, що б результат був відповідний до вимог і дані операції мають підбиратися динамічно за допомогою окремо розроблених модулів, які по суті і є рішенням проблеми. На даний час подібні рішення досягаються динамічно-доповнюючими себе графами. Проте, як вже було сказано вище методів, що надають абсолютного прискорення роботи подібних систем немає, адже для цього необхідно враховувати типи систем їх

особливості та об’єми інформації з якою відбуваються дії. Розроблені на даний момент засоби використовують завчасно спланований так званий «щоденик» задач, за допомогою якого будуть вибрані подальші дії системою, це не є оптимальним, адже стадія генерації подібних «щоденників» займає певний час на початку роботи системи і відповідно чим більший об’єм та складність задач ти довше часу буде на це витрачено. Також є варіант статичного розпаралелювання задач на стадії розробки, що виводить систему на максимальний рівень роботи на найоптимальнішу швидкість, про те це можливо лише у випадку якщо розробник знає та описую у програмному коді всі можливі варіанти розвитку подій за допомогою потоків програмних засобів.

Основна спрямованість проекту – винайдення методу, що дозволить прискорити розподіл та генерацію задач для кластерних систем, прискорення має відбуватися за рахунок генерації наступної задачі на етапі виконання поточної (попередньої). Використання даного методу прискорить роботу усіх маніпуляцій з інформацією та її хранилищами.

5.1.3 Мета та завдання проекту відповідно до проблеми.

Метою проекту є винайдення та тестування системи для прискорення роботи кластерних систем з динамічним розподілом задач. Для цього необхідно, виділити вже існуючі методи підвищення ефективності виявити їх недоліки та переваги один над одними обрати шлях розвитку проекту відповідно до виявлених переваг та недоліків. Прискорити роботу дата-сайнс центрів та дата-центр центрів, прискорити обробку інформації, що дозволить надати проекту з винайдення методу подальшу його оптимізацію. Також ціль створити симулятор для отримання статистичної інформації для демонстрування та порівняння з аналогами винайденого методу. Відповідно презентувати отриманні

результати та знаходження інвестицій для встановлення або переустановлення подібних систем чи заміна старіших і менш оптимальних. Залучити сторонніх розробників методом подальшого випуску вільного пакету програмних засобів, що дозволять розробникам створювати власні методи чи ПЗ на основі розробленого методу, що дасть можливість використовувати та популізувати метод серед розробників.

В основу методу буду покладено засіб заміни «щоденика» та виключення статичного розпаралелювання. «Щоденик» системи буде замінено на оргграф на вузлах якого знаходяться операції системи а ребра графу це послідовність виконання, тобто на початку робити системи ми маємо лише набір вхідних вузлів так званих стартерів, а далі при виконання кожного вузла генерується наступний до якого по графу ребру переходить «поточний стан» і відбувається виконання вже наступного вузла з генерацією наступного.

5.1.4 Дерево проблем.

Дерево проблем проекту зображено на рисунку 5.1.

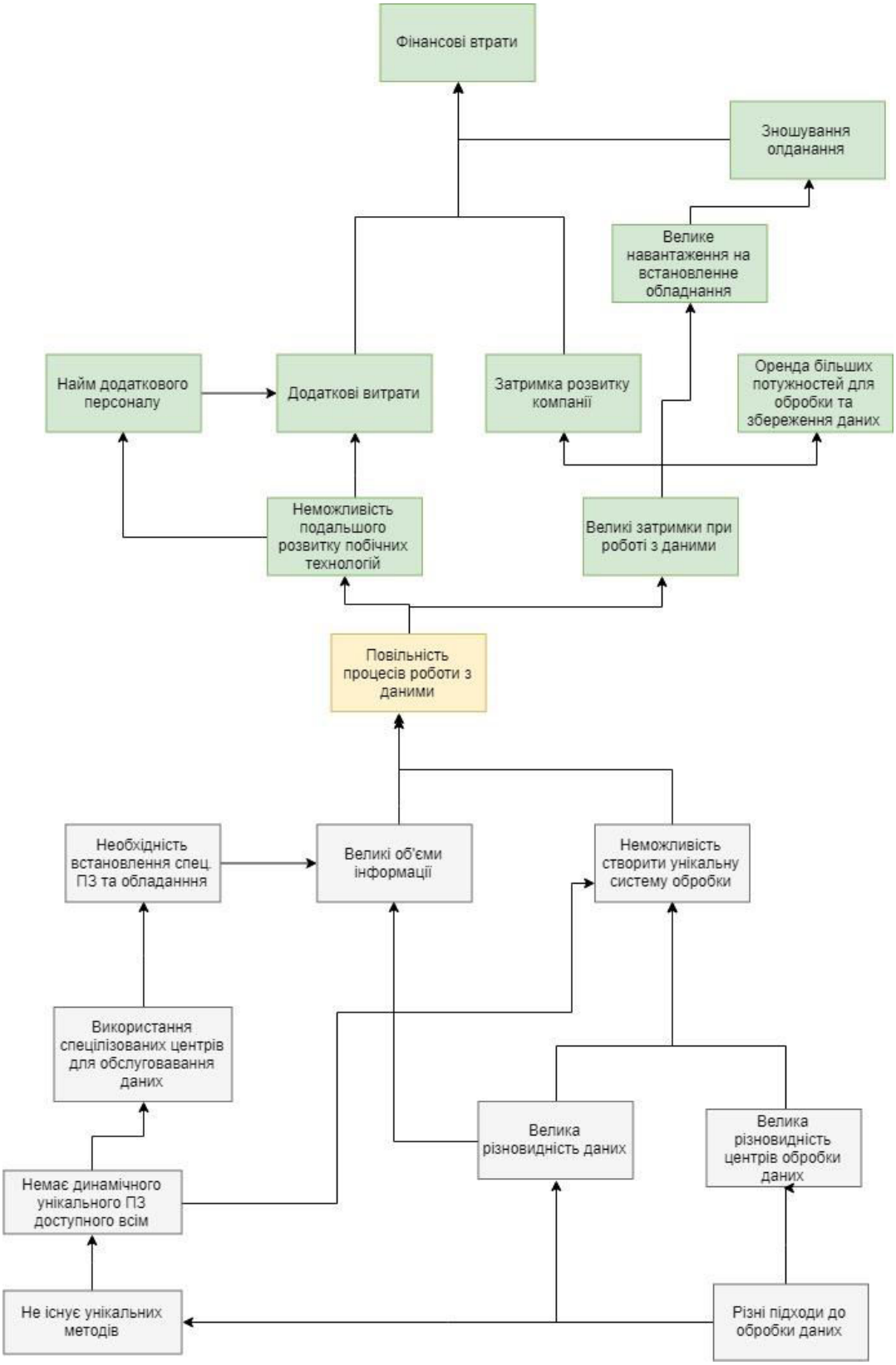


Рис. 5.1. Дерево проблем проекту

5.2. Аналіз зацікавлених сторін проекту

5.2.1 Зацікавлені сторони проекту.

Зацікавлені сторони проекту та оцінка їх важливості та зацікавленості зібрані у таблиці 5.2.1.

Таблиця 5.2.1.

Група зацікавлених осіб	Інтереси групи в проекті	Умови довгого співробітництва з проектом	Важливість	Зацікавленість
Внутрішні зацікавлені сторони проекту				
Розробник проекту	Виконання проекту; Досягнення цільових показників проекту	Подальший розвиток проекту; Приток інвестицій	Висока (10)	Висока (10)
Команда керування проектом	Досягнення цільових показників проекту; Подальший розвиток технологій проекту	Подальший розвиток проекту; Приток інвестицій; Збільшення особистого доходу	Висока (9)	Висока (9)

Інвестори проекту	Отримання зазначених доходів від участі в проекті; Подальший розвиток проекту; Досягнення цілей	Збільшення прибутку від інвестицій; Вигідніші умови підтримки проекту	Висока (9)	Висока (9)
Внутрішньо – корпоративні зацікавлені сторони проекту				
Менеджмент проекту	Розвиток проекту; Збереження притоку робочих місць;	Приріст робочих місць; Можливість кар’єрного росту; Реклама	Висока (9)	Середня (6)
Акціонери	Приріст доходності; Ріст загальної вартості проекту	Збільшення вартості проекту	Висока (9)	Висока (9)

Побічні співробітники	Кар'єрний ріст	Кар'єрний ріст	Середня (6)	Висока (9)
Зовнішні зацікавлені сторони проекту				
Дата центри	Використання продукту проекту для прискорення роботи центру; Якість власного продукту; Власний технологічний розвиток	Розвиток технологічного фону	Нижче середнього (4)	Висока (9)
Підприємства по автоматизації процесів	Використання продукту проекту для збільшення якості власного продукту;	Розвиток проекту	Нижче середнього (4)	Висока (9)
Заводські підприємства з автоматизованим виробництвом	Використання продукту для покращення власного технологічного фону	Покращення старих технологій методами внесення змін у проект	Нижче середнього (4)	Середня (6)

Наукові дослідницькі центри / заклади	Прискорення процесу досліджень	Підтримка продукту	Нижче середнього (4)	Висока (9)
Заклади з великим оборотом інформації	Прискорення процесу передачі інформації; Автоматизація; Спрощення процесу збереження	Підтримка проекту	Середня (5)	Висока (9)
Побічні зацікавлені сторони проекту				
Розробники ПО	Використання потокових технологій	Оновлення технологій	Низька (3)	Низька (3)

5.2.2 Аналіз зацікавлених сторін проекту.

Як видно з таблиці 5.2.1 розділу 5.2.1, зацікавлені сторони були розділені на 4 основні групи.

1.Внутрішні зацікавлені сторони проекту.

Тут зібрані особи, що напряму зацікавлені у проекті та його подальшому розвитку та встановленню на підприємствах закладах тощо, серед яких інвестори, для яких важливо мати прибуток з проекту, розробник проекту, який безпосередньо є керівником усього, що відбувається, як всередині команди так і зовні, команда проекту, що слідкує за виконанням проекту.

2. Внутрішньо – корпоративні зацікавлені сторони проекту

Ця група має менш серйозне відношення до проекту, проте без них його існування та подальший розвиток неможливі, серед них майбутні

власники акцій, до яких входять і особи з першої групи, тому їх важливість та зацікавленість тісно перетинається з учасниками першої групи, а також до акціонерів належать побічні особи, зацікавленість яких може бути менша.

Побічні співробітники, їх важливість не так висока особливо на перших етапах розвитку, по перше їх кількість невелика і по друге вони не несуть такої сильної важливості адже не пов’язані напряму з запуском чи застосунком розроблюваного методу у використання, проте якщо проект отримає подальший розвиток, їх кількість зросте, а співробітники, що були присутні на старті отримують більші доходи.

3. Зовнішні зацікавлені особи.

Це всі хто зацікавлений саме у отриманні доступу до використання методу, тобто «цільова аудиторія» методу, до неї входять усі фірми, підприємства, корпорації, заклади, що працюють з інформацією у тому, чи іншому вигляді, а також можуть бути носіями статистичних даних, або тестовими платформами методу.

4. Побічні зацікавлені сторони.

У супереч тому, що дана група має низьку зацікавленість та невисоку важливість, є потужним рушієм для популяризації методу, адже незалежні програмісти, можуть використовувати метод для створення власних продуктів у майбутньому, що принесе його у постійне використання.

5.3 Опис наукового проекту та технології

Науковим продуктом у рамках наукової дисертації є метод підвищення ефективності роботи розподілених кластерних систем з динамічним розподілом задач, що і є науковою назвою продукту [37].

Даний метод призначений для використання будь-якими закладами, які працюють з інформацією, чи зберігають її. Даний метод спростить

закладам процес роботи з інформацією, шляхом використання технології ациклічного направленого орграфу (DAG). В якому дуги – це шляхи направлення, а вузли це операції які необхідно виконати на даному етапі та згенерувати наступну операцію, таким чином відпадає необхідність завчасного планування дій системи, чи статичного програмування на стадії написання програмного коду системи.

Ациклічний направлений граф — випадок орієнтованого графа, в якому відсутні орієнтовані цикли, тобто шляхи, що починаються і закінчуються в одній і тій самій вершині. Орієнтований ациклічний граф є узагальненням дерева.

На ранніх етапах розробки тестування подібного методу буде проводитись методом збору статистичних даних відносно швидкодії , за допомого симулятору роботи дата-центру з поточною обробкою даних, наприклад конвертацією чи кодуванням, даний симулятор буде створено власноруч засобами написання програмного коду. В наступних етапах застосунку методу на підприємствах тощо, дані підприємства будуть слугувати так званою платформою для тестування та збору інформації для подальшого вдосконалення методу, таким чином розвиток методу ніколи не припиниться.

Для реалізації методу на абстрактному рівні виділено декілька стадій:

- Аналіз різних підходів для прискорення роботи динамічних систем та виділення їх переваг та недоліків;
- Створення систем на основі орграфу;
- Тестування написаної системи порівняно з існуючими методами;
- Написання тестової платформи (симулятору дата-центру);

- Збір статистичних даних та порівняння їх з реальною статистикою;
- Усунення недоліків, якщо це можливо, базуючись на отриманих даних з тестової платформи.

5.4 Бізнес рішення та основні характеристики бізнес - продукту

5.4.1 Резюме продукту

Результатом науково – дослідницької роботи буде ПО та документація до нього, дане ПО допоможе прискорити роботу алгоритмів, або прискорити обмін інформацією. Далі буде розглянуто принцип розробки, користь продукту, способи підтримки та подальший розвиток, що дасть абстрактне розуміння про продукт, що буде отриманий в результаті дослідницько роботи. Також далі будуть розглянуті можливості та шляхи вводу ПО на виробництва різних типів, а також можливості отримання ПО та подальшого використання у «домашніх умовах», також будуть описані необхідні вимоги для встановлення ПО.

5.4.2 Опис продукту

Результатом дослідницької роботи буде ПО, але слід зазначити, що ПО далеко не основний виробничий продукт, в основі ПО лежить метод розподілення операцій на кластерних системах, тобто результатом роботи буде саме метод, який можна встановлювати або інтегрувати в різні системи виробництва, що працюють з великими об’ємами інформації.

Основним плюсом при використанні продукту методу є прискорення обробки інформації та збірок різноманітних програмних продуктів, такий метод може знадобитися: дата центрам, закладам з великим обігом інформації, автоматизованим підприємствам, розробникам забезпечення.

Дата – центри зможуть прискорити процес обробки даних та подальшого їх збереження.

Заклади з великим обігом інформації використовуючи дані технології зможуть в рази швидше сортувати зберігати та перенаправляти необхідну інформацію.

Розробники ПО, за допомогою даного методу зможуть писати багатопоточні додатки не відслідковуючи та координуючи потоки, а також зручно тестувати.

Основною особливістю розробки є простота в інтегруванні, тобто даний метод просто реалізований у вигляді додатку або бібліотеки, що підключається до побічного проекту або інтегрується в процес, що дозволяє розподіляти навантаження методом поточних обчислень.

Процес «продажу» методу буде відбуватися методом інтегрування методу в окремий проект або цілком на виробництво або підприємство, та подальша його підтримка за оплату послуг по підтримці, щомісяця та за використання технології, у випадку з розробниками власних проектів їм буде надана скорочена користувацька бібліотека, що просто в інтеграції проте не підходить для великих проектів. Для отримання доступу до технології необхідно замовити інтегрування в проект та обов'язково подальшу підтримку, до якої буде прив'язаний окремий спеціаліст.

Продукт знаходиться на стадії постійного розвитку, що дасть можливість постійного доповнення методу та пристосування його до різних типів даних, що є однією з проблем оптимізації роботи кластерних систем, адже різні типи інформації та різні системи виконують обробку та зберігають по різному, бізнес рішенням є постійна підтримка та постійній розвиток саме методу, адже ПЗ буде встановлюватись саме на момент дійсної версії методу, тому за оновлення та підтримку необхідна додаткова фінансації команди підтримки та безпосередньо за отримання нової версії продукту. Таким чином оскільки результатом роботи окрім безпосереднього прискорення є статистичні дані кожного власника системи, система обробки інформації буде використовувати власний

спосіб зберігання и прискорення, тому затримок у оновленні даних буде менше, що дасть можливість своєчасно та оперативно помічати будь-які зміни у облуговуємих системах.

В майбутньому метод перейде на інтегрування на великі серверні частини, що значно прискорить обмін інформації, але основним розвитком є перетворення на платформу для запуску додатків, що будуть працювати швидше за допомогою даного методу, а також випуск повноцінної бібліотеки для розробників.

5.4.3. Конкурентні переваги рішення.

Серед основних критеріїв у конкурентноздатності є:

- Патентоздатність (новизна проекту);
- Раціональність виробничої організаційної структури схеми;
- Конкуренто спроможність персоналу;
- Прогресивність технології;
- Прогресивність технологічних процесів та обладнання;
- Науковий рівень розвитку розробників;
- Науковий рівень системи.

Технологічні переваги методу(проекту):

- Відсутність так званого «щоденнику» програмного паралельного забезпечення. «Щоденник» програмного забезпечення це програмний модуль, що стартує до роботи самого додатку роботи з даними та програмує послідовність виконання операцій програмою у відповідних потоках, в процесі виконання коду програми, «щоденник» динамічно надає програмі дані про те що може виконуватися паралельно, а що має чекати виконання попередніх дій. Суть методу у реалізації динамічного розпаралелювання, що надає змогу

позбавитися від «щоденнику» та часу який виділяється на його роботу та часу звернення до нього самою програмою.

- Позбавлення від проблеми статичного розпаралелювання, тобто програміст не має описувати безпосередньо у коді програми потоки даних, що спрощує роботу розробника в подальшому.
- Реалізація у вигляді динамічної бібліотеки з тестовим пакетом, що надасть змогу до імплементації методу на виробництво перевірити результати роботи методу на реальних даних.
- Зручний пакет розробника, що надасть додаткову мотивацію до використання методу «вільним» та побічним розробникам.
- Кроссплатформеність, даний метод, а саме його програмна реалізація не потребує підключення до мережі інтернет, тобто працює без підключення до мережі, навіть локальної.
- Відповідно по пункту 5, програмна реалізація методу має збільшену відказостійкість, при обриванні з мережею інтернет відключаються лише побічні функції ПЗ, а сам метод розпаралелювання продовжує працювати.
- Система моніторингу у реальному часі.

Загальні переваги проекту:

- Так як проект та в майбутньому команда(фірма, компанія, тощо) не використовує підхід кредитування тому може дозволити зручний та сталий формат формування цінової політики;
- Весь персонал, що буде займатися обслуговуванням та оновленням систем та обладнання у замовників, проходять

постійні курси підготовки та розвитку, приймають участь у конференціях;

- Оптимізація потужностей, через те що метод прискоренням систем постійно розвивається та знаходиться у постійному процесі збору статистичних даних, це дає можливість оновлювати ПО та обладнання у найкоротші строки;
- Швидка динаміка у оновленнях та підтримці ПО, відповідно до нових статистичних даних зібраних включаючи інші заклади, відповідно чим більше замовників тим більша швидкість оновлення та доповнення методу.
- Відсутність некваліфікованих співробітників;
- Встановлення та оновлення системи прискорення відбувається напряму без посередників;
- Онлайн тестування методу на прикладах.

Також слід приділити особливу увагу пункту про тестування, адже при впровадженні подібних методів на виробництво, замовник хоче бути впевненим у тому, що дана система буде давати необхідні результати, це проблема вирішується методом реалізації окремої повноцінної, самостійної користувацької бібліотеки тестування, яка надасть змогу окрім симуляції роботи системи на реальних даних та моніторингу, побудувати систему покращення роботи підприємства, а саме програмного забезпечення на яке вона встановлюється. Даний аспект надає суттєву перевагу, адже перед встановленням метод може бути повність протестований, а результати покращення проаналізовані замовником.

Відсутність «щоденника» (пункт 1. Технологічних переваг) надає можливості реалізації динамічного ациклічного орграфу, так званого DAG файлу, приклад данного графу зображений на рисунку 5.2, його перевага над іншими реалізаціями у тому, що набір операцій додатку зберігається

окремо, а самі процеси виконання окремо, тобто дуги графу є напрямки дії програми, а вузли є операціями, що достаються із загального пулу операцій.

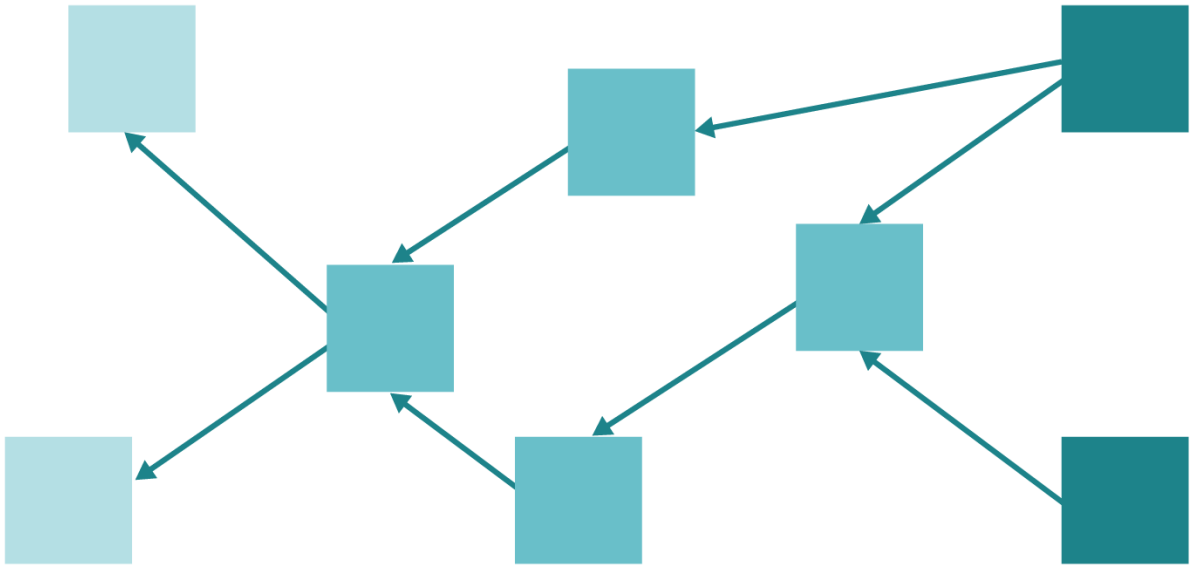


Рис. 5.2. DAG.

Основними рисами моделі паралельного програмування є більш висока продуктивність програм, застосування спеціальних прийомів програмування та, як наслідок, більш висока трудомісткість програмування, проблеми з перенесенням програм. Паралельна модель не володіє властивістю унікальності. В паралельної моделі програмування з'являються проблеми, незвичні для програміста, який звик займатися послідовним програмуванням. Серед них: керування роботою безлічі процесорів, організація міжпроцессорных пересилок даних і т. д. Можна сформулювати чотири фундаментальних переваги паралельних програм:

- Паралелізм;
- Універсальність;
- Локальність;
- Модульність.

5.4.4 Клієнти та сигменти ринку споживачів

Як вже було зазначено у главі 5.2, аналіз зацікавлених сторін, основними клієнтами та споживачами даного методу є підприємства та заклади з великим обігом даних. Детальна збірка інформації про клієнтів зібрана у таблиці 5.4.1.

Таблиця 5.4.1.

Тип закладу	Опис проблеми закладу, яку вирішує метод	Рейтинг споживача
Дата центри	Подібні заклади щосекунди оброблюють велику кількість інформацію, яку при цьому треба зберігати на оброблювати, інформації з кожним днем стає все більше і більше, а методи обробки, збереження та надсилання розвиваються занадто повільно, що веде до втрати швидкості та продуктивності роботи закладу та втрати точності. Метод підвищення ефективності роботи кластерних систем з динамічним розподілом задач, частково вирішує дану проблему, адже це і є його суть, прискорити процеси роботи з даними, за допомого методу дані, що циркулюють в системі будуть в рази швидше проходити обробку	10

Тип закладу	Опис проблеми закладу, яку вирішує метод	Рейтинг споживача
	відповідно більша кількість в одиницю часу, що призведе до збільшення ефективності роботи дата – центрів та збільшить їх потенціал.	
Гос. Структури	Зазвичай подібні заклади мають великий обіг інформації, що робить процес обробки та збереження затягнутим та громіздким, а також важливим у таких закладах є момент фільтрації даних та налаштування фільтрів пошуку інформації та складання її за певними тегами, що зрозуміло відбувається динамічно у процесі надходження інформації, запропонований метод динамічного розподілення задач для кластерних систем значно прискорить процес фільтрації та пошуку даних методами налаштування ациклічних орграфових конструкцій, що розгрузить навантаження на систему, адже обладнання устаріле.	6

Тип закладу	Опис проблеми закладу, яку вирішує метод	Рейтинг споживача
Виробничі підприємства	Зазвичай подібні компанії оперують статистичними даними та великими базами даних, наприклад бази з ціновими політками, чи списками співробітників, чи набори товарів та інше, для роботи база повинна опрацьовувати набори запитів, що краще роботи асинхронно у окремих потоках, зазвичай подібні підприємства мають стабільне кластерне обладнання, що дозволяє без проблем на його основі встановити програмне забезпечення, що реалізує метод підвищення ефективності запитів до БД, подібний підхід дозволить позбавитись від черг запитів, чи перенавантажених конкотованих запитів, також так як програмне забезпечення з реалізацією методу дозволяю зручно моніторити та збирати статистичні дані до окремої БД, з результатами роботи додатку.	8

Як видно з таблиці, що наведено вище основними клієнтами є великі компанії, що оперують з інформаційними потоками, перевагою даного сигменту є обмий інвестицій та привзаність до кожного великого клієнта, проте мінусом є великі ризики та складність у знаходженні нових клієнтів. Поле «Рейтинг споживачів» таблиці 5.4.1, вказує на сумарний коефіцієнт на необхідність конкретному типу закладів у відношенні до того чи готові вони впроваджувати подібні технології та чи потрібні вони їм взагалі, наприклад для гос. Закладів подібний метод впровадження було б корисно, через прискорення роботи та розгруження персоналу, проте готовність встановлювати та оновлювати подібне ПЗ менша ніж на приватних підприємствах.

Як відомо сегментація ринку є 3х рівнів:

- Стратегічне сегментування (макросегментування) воно забезпечую типізацію товарів ринку і опис загальних характеристик товару на ринку, проект базується на розробці методу, тобто є науковою новизною.
- Товарне сегметування ринку (мікросегментвуання) воно базується на врахуванні потреб потенційних покупців, в даному випадку це є різновидності обробки та типи прискорень роботи систем.
- Конкурентне сегментування воно задає принцип конкуренції у заданому сегменті ринку, основою методу є швидкість роботи, тобто швидкодія є рішучим фактором конкуренції.

5.5 Унікальна цінність пропозиції (наукового продукту)

Як вже згадувалося раніше подібних рішень існує цілий набір, але вони всі або статичні, або користуються так званим «щоденником» планування, в основі методу лежить децентралізований ациклічний орграф, який і є так званою новизною рішення, що дозволить не витрачати час на планування процесів.

Оптимізацію роботи можна досягати на різних рівнях за допомогою подібних графів, розглянемо деякі з них:

- Рівень процедур - на цьому рівні різні розділи однієї і тієї самої програми мають виконуватися паралельно. Ці розділи називаються процесами і відповідають приблизно послідовним процедурам. Проблеми поділяються на суттєво незалежні частини так, щоб по можливості рідше виконувати операції обміну даними між процесами, які потребують відносно великих витрат часу. В різних галузях застосування стає ясно, що цей рівень паралельності ні в якому разі не обмежується розпаралелюванням послідовних програм. існує великий ряд проблем, які потребують паралельних структур цього типу навіть тоді, коли так само, як і на програмному рівні, у користувача є тільки один процесор. Застосування (основне) - загальне паралельне оброблення інформації, де застосовується поділ вирішуваної проблеми на паралельні задачі - частини, які вирішуються багатьма процесорами з метою підвищення обчислювальної продуктивності
- Рівень арифметичних виразів - арифметичні вирази виконуються паралельно покомпонентно, причому в суттєво простіших синхронних методах. Якщо, наприклад, йдеться про арифметичний вираз складання матриць, то він синхронно розпаралелюється дуже просто тому, що кожному процесорові

підпорядковується один елемент матриці. При застосуванні n*n процесорних елементів можна одержати суму двох матриць порядку n*n за час виконання однієї операції складання (за винятком часу, потрібного на зчитування та запис даних). Цьому рівню притаманні засоби векторизації та так званої паралельності даних. Останнє поняття пов'язане з детальністю розпаралелювання, а саме - з його поширенням на оброблювані дані. Майже кожному елементу даних тут підпорядковується свій процесор, завдяки чому ті дані, що в машині фон Неймана були пасивними. перетворюються на “активні обчислювальні пристрої.

- Рівень двійкових розрядів - на цьому рівні відбувається паралельне виконання бітових операцій в межах одного слова . Паралельність на рівні бітів можна знайти в будь-якому працюючому мікропроцесорі. Наприклад, у 8-розрядному арифметико-логічному пристрої побітова обробка виконується паралельними апаратними засобами.
- Паралелізм на рівні потоків - Одне з рішень даної проблеми пов'язане з реалізацією концепції паралелізму на рівні потоків (Паралелізм завдань - TLP). Якщо запущене на виконання завдання не в змозі завантажити роботою всі функціональні пристрої, то можна дозволити процесору виконувати більш ніж одну задачу, створюючи тим самим другий потік, щоб він завантажив простоюючи пристрій. Тут є аналогія з багатозадачною операційною системою(ОС): щоб процесор не простоював, коли завдання перебувають у стані очікування (наприклад, завершення введення-виведення), ОС перемикається на виконання іншої задачі. Найбільш ефективною на сьогодні стала архітектура з одночасним

виконанням потоків (Simultaneous Multi-Threading). У такій ситуації на кожному новому такті на виконання в який-небудь виконавчий пристрій може направлятися команда будь-якого потоку - залежно від обчислювального алгоритму .

Розпаралелювання на рівні потоків TLP, на відміну від ILP, управляється програмно. Віртуальна багатопоточність створюється в результаті виділення в одному фізичному процесорі двох або більше логічних процесорів. Класичним прикладом такого підходу стала технологія Hyper-Threading (HT) компанії Intel. Внаслідок того, що протягом такту, як правило, не всі виконавчі модулі процесора задіяні, їх можна завантажити паралельним потоком завдань. Зрозуміло, що вдвічі продуктивність не збільшиться, оскільки паралельні потоки використовують загальні пам'ять, кеш і т.д. (до того ж виникають втрати через синхронізації і розпаралелювання інструкцій), але вона в цілому зростає на 35-50%. Мінусом використання технології TLP є виникнення конфліктів, коли одному потоку потрібні результати виконання іншого договору, що призводить до очікування і зростанню кількості тактів, необхідних для виконання інструкцій.

5.6 Доходи і витрати

5.6.1 Витрати.

Для прийняття рішення щодо інвестиційного проекту всі витрати, пов'язані з його здійсненням, необхідно розподілити на інвестиційні та виробничі.

Загальна сума витрат на здійснення проекту включає:

- Витрати на формування основного капіталу містять початкові і поточні інвестиції;
- Витрати на формування оборотного капіталу;
- Виробничі витрати.

Всі інвестиційні потреби підприємства можна поділити на три групи:

- Прямі інвестиції - безпосередньо необхідні для реалізації інвестиційного проекту;
- Супутні інвестиції - вкладення в об'єкти, безпосередньо технологічно які пов'язані із забезпеченням нормальної експлуатації;
- Інвестування виконання науково-дослідних робіт.

  До складу початкових інвестицій відносяться:

- Витрати на передінвестиційні дослідження, проведення дослідницьких і конструкторських робіт, на розробку проектних матеріалів, на робоче проектування і прив'язку проекту;
- Витрати на придбання та оренду потужностей, включаючи вартість підготовки до освоєння;
- Витрати на придбання та доставку машин, обладнання, інструменту та інвентарю, в тому числі імпортних;
- Витрати на приймально-здавальні випробування;
- Витрати на пусконалагоджувальні роботи, комплексне освоєння проектних потужностей і досягнення проектних техніко-економічних показників;
- Витрати на придбання патентів, ліцензій, ноу-хау, технологій та інших амортизаційних нематеріальних активів;
- Витрати на підготовку кадрів для об'єктів, що вводяться в дію;

- Одноразові виплати, зокрема гарантує і страховим організаціям;
- Витрати, що виникають при створенні та реєстрації фірми (оплата юридичних послуг зі складання установчих документів, витрати на реєстрацію фірми і оформлення прав власності;
- Витрати на підготовчі дослідження не враховано в кошторисної вартості об'єкта;
- Витрати, пов'язані з діяльністю персоналу в період підготовки виробництва (оплата праці, витрати, утримання приміщень, автомобілів, комп'ютерів та іншого обладнання), не враховані в кошторисної вартості об'єкт.

Висновки по витратам:

Вартість проекту = ціна проектування + ціна розробки + активна підтримка і доопрацювання після впровадження + плата за компоненти та сервіси + реклама і просування. На рисунку 5.6.1 схематично зображено 2 етапи витрат.

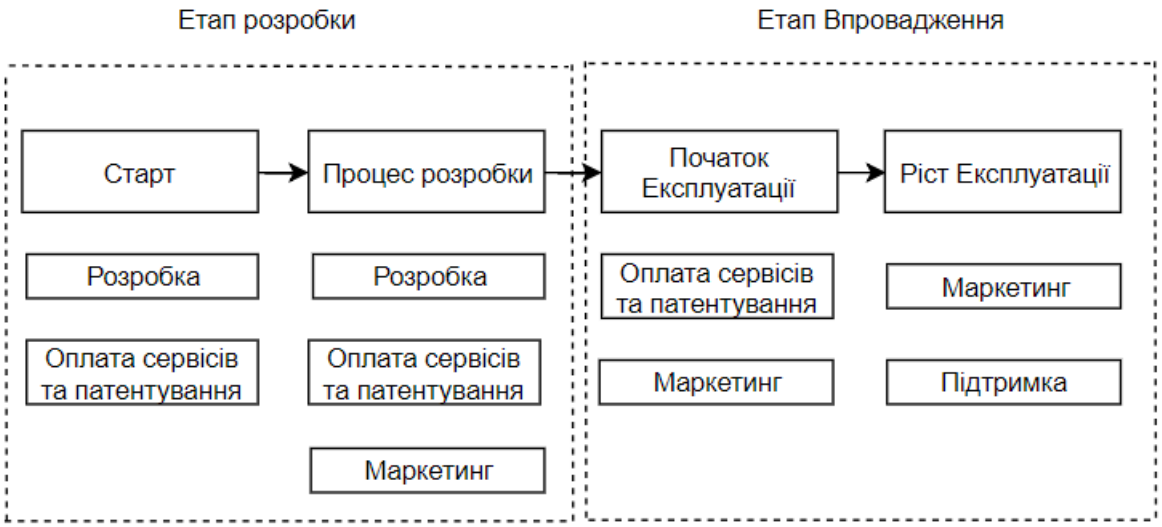


Рис. 5.6.1. Види витрат на проект на різних етапах

5.6.2 Доходи.

Як зрозуміло з попередніх пунктів та ринку споживачів, товаром є метод для підвищення ефективності роботи паралельних кластерних систем з динамічним розподілом задач. Основними джерелом доходу будуть кошти з експлуатації даного методу. Загальну дохідність проекту в майбутньому буде прийнято рахувати метриками доходності, такими, як:

- Bookings (замовлення)**;**
- Revenue (дохід);
- TCV — Total Contract Value (загальна вартість контракту);
- LTV — Life Time Value (тотальна цінність);
- Unearned or Deferred Revenue (майбутні доходи).

Bookings - це оцінка вартості контракту між компанією і клієнтом. Ця метрика відбиває зобов'язання клієнта заплатити компанії зазначені в договорі гроші. Так як основою експлуатації та впровадження на виробництво методу є контракт дана метрика, грає основну роль в обрахунках доходності проекту, сумарна вартість контракту на один заклад.

Про прибуток (Revenue) можна говорити тоді, коли послуга вже надана або буде надаватися регулярно протягом зазначеного в договорі терміну підписки. Є вихідним фактором першої метрики, рахується по закінченню контракту, тобто при вдалому завершенні.

TCV – вихідна вартість контракту, рахується у випадку, якщо у контракті описані пункти підтримки, доповнення або перерахування вартості чогось, наприклад оплата подальшої підтримки виробництва. TCV може з часом збільшуватися або зменшуватися. Переконайтеся, що TCV враховує також одноразові витрати, оплату спеціальних послуг і повторювані платежі.

LTV - Довічна цінність - це поточна оцінка майбутньої чистого прибутку від клієнта протягом усього періоду його відносин з компанією. Вона допомагає визначити довгострокову цінність клієнта, а також чистий прибуток в розрахунку на одного клієнта з урахуванням витрат на його залучення (CAC). Місячна маржа від кожного клієнта = дохід від клієнта мінус змінні витрати, пов'язані з клієнтом. Змінні витрати включають в себе всі адміністративні та операційні витрати, пов'язані з обслуговуванням клієнта.

UDR - это те деньги, которые вы собираете при заказе продукта (услуги), то есть до его получения (реализации). Компании признают только выручку в течение срока предоставления услуги — даже если клиент оплачивает большую часть услуги до совершения сделки. Поэтому в большинстве случаев эти деньги учитываются в балансе в строке так называемого отложенного дохода.

1.6.3 Таблиця з витратами / доходами.

Таблиця 5.6.1 описує стадії розвитку проекту, розділи витрат та сумарний дохід. Слід врахувати, що перший та другий етапи є разовими і є препроектими, третій етап розраховується помісячно, після закінчення перших двух етапів.

Таблиця 5.6.1.

Етап проекту	Статті витрат	Шляхи доходу	Витрати	Дохід	Сумм
Етап розробки	Розробка, оплата сервісів, патентування, маркетинг		1.000 у.о	0	-1.000у.о
Етап впровадження	Маркетинг, підтримка, реалізація, впровадження	Впровадження у тестовому режимі на виробництво	2.000 у.о	1.000 у.о	-1.000 у.о
Етап експлуатації	Підтримка, обслуговування, розвиток проекту, оплата персоналу	Замовлення, оплата послуг замовниками, підтримка, підписка, оплата тестових режимів, оплата послуг моніторингу закладів, в інформації	2.000 у.о	5.000 у.о	+3.000 у.о

5.7 Бізнес - модель

На рисунку 5.7.1 зображена загальна схема бізнес моделі.

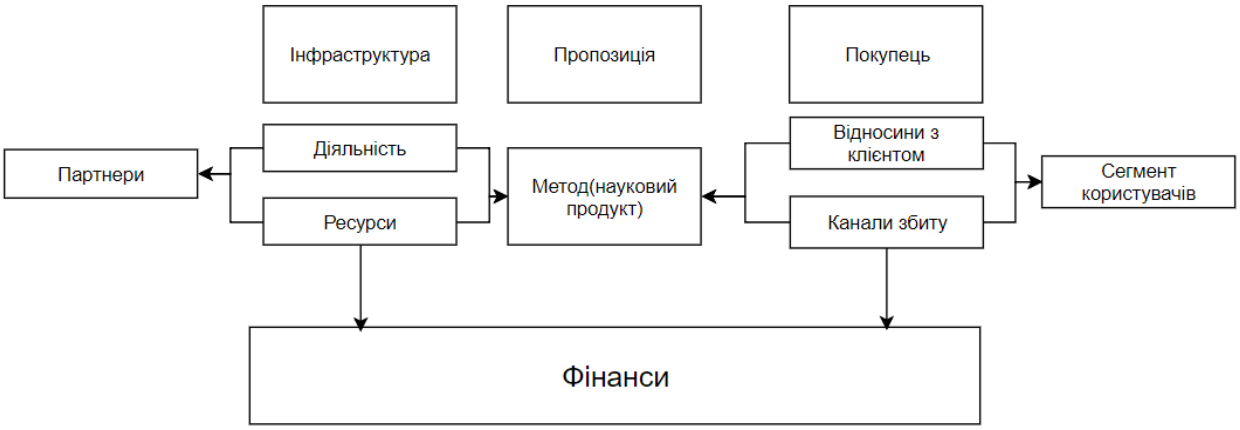


Рис. 5.7.1 Загальна бізнес модель.

Сегмент користувачів:

Для кого? – Заклади з великим обігом інформації та даних.

Найважливіші клієнти? – Великі заклади та корпорації з укладанням контракту на подальшу підтримку та оновлення.

Пропозиція:

Яку цінність від методу отримують клієнти? – Позбавлення планувальника, що наддасть змогу швидше оброблювати зберігати дані, що наддасть прибуток та спростить навантаження на обладнання.

Яку проблему користувачів ми вирішуємо? – Затримки обробки даних, скорочення витрат на оптимізацію процесів роботи з інформацією.

Яку необхідність користувачів ми задовольняємо? – Користувачам необхідно оновлювати, зберігати оброблювати інформацію та дані швидше від поточних показників, метод оптимізації дозволяє це зробити.

Які набори продуктів ми пропонуємо? – Метод пришвидшення, платформа для тестування та збору статистики, також підтримки оновлення програмного забезпечення та безкоштовний відкритий код.

Канали розповсюдження:

Через які канали ми хочемо досягти кожен сегмент наших споживачів? – реклама та створення вільної бібліотеки, розповсюдження її у всіх можливих пакетах для різних мов програмування, наприклад NuGet або MavenGet.

Який з каналів розповсюдження є найпривабливішим? – можливість розповсюдження методу, через вільні бібліотеки програмного коду.

Потоки доходу:

За що готові платити користувачі? – за використання методу для прискорення роботи систем роботи з інформацією, що надає можливість поліпшити стан системи, за статистичну інформацію та моніторинг, а також за оновлення та технічне обслуговування.

Яким чином вони можуть платити? – так, як основними клієнтами є заклади, а не цільові одиночні клієнти, то більшість оплата буде укладатися за контрактами в яких буде описано умови підтримки, тестування та оновлення, контракт є можливість продовжувати, або заключати, як підписку на певний статичний час.

Альтернативні шляхи? – дохід також може бути з платних пакетів для одиночних розробників, що надають змогу кастимізвувати та покращувати локальні проекти.

Основні ресурси :

Необхідні ресурси? – серед необхідних ресурсів, платформи для тестування, та статистичні дані закладів по обробці даних. Наукові досягнення та патенти на подібні методи та фінансові ресурси.

Шляхи знаходження необхідних ресурсів? – безкоштовно тестові пакети для закладів, від чого виграє проект і заклади.

Відносини з клієнтами? – замовник підписуючи один з видів контракту, отримує набір послуг, окрім впровадження технології на

виробництво, це може бути обслуговування оновлення, встановлення систем поточного моніторингу та оповіщення.

Потоки доходів? – як вже декілька разів сказано вище, основний дохід йду від закладів, що працюють з інформацією, а невелика частина з користувацьких додатків на основі бібліотеки. Заклади оформлюють підписку у вигляді контракту за деяким пакетом послуг, які коштують по різному і містять різні набори послуг окрім безпосереднього впровадження методу.

ВИСНОВКИ

Метою даної магістерської дисертації була розробка програмного забезпечення для емуляції роботи системи керування потоками даних, на базі архітектури, що була спроектована на основі розробленого та досліджуваного методу підвищення ефективності роботи подібних систем та збільшення відмовостійкості. У ході даної роботи було виконано наступні кроки:

- Аналіз існуючих підходів до проектування систем керування потоками даних показав, що використання деяких окремих елементів можна спростити, а недоліки, яке це спричиняє можна ліквідувати шляхом реконфігурації системи в цілому. А засобі моделювання є негнучкими та важкими для користувачів, у випадку якщо необхідно тестування тощо;
- Проведено аналіз існуючих програмних засобів для проектування та тестування різних СПД, зроблено висновки, оцінено недоліки даних засобів;
- Була досліджена можливість підвищення ефективності роботи систем керування потоками даних, шляхом використання вільного адресного простору замість асоціативної пам’яті, а проблеми, що виникала внаслідок цього, було вирішено шляхом додавання додаткових регістрів для збереження операцій на операндів. Також це збільшило відмовостійкість системи. На основі цього дослідження було спроектовано систему;
- За необхідністю створення програмного емулятору роботи системи, для тестування та роботи з системою, для моніторингу та відображення поведінки системи. За аналізом технологій реалізації роботи даної системи у вигляді

програмного забезпечення було обрано мову C#, реалізовано у вигляді десктопного додатку з використанням технології WPF, а також розроблено спеціальну мову для подачі сигналів до системи.

- Було розроблено програмний продукт для емуляції роботи системи за розробленим методом підвищення ефективності систем, що керуються потоками даних. Програмний продукт для емуляції дозволяє користувачу:
    - Створити задачу для рішення у вигляді псевдокоду та подати на вхід системі для рішення;
    - Проаналізувати кількість тактів системи;
    - Покроково оцінити роботу багатопроцесорної системи;
    - Зимітувати відключення процесору для перевірки поведінки системи;
    - Відновити роботу обчислювального модуля.
- Було запропоноване бізнес модель, яка б надала можливість впровадження даного методу а також тестування подібних систем на підприємствах різних типів, а також як програмна бібліотека. Було створено дерево проблем, оцінено переваги та недоліки проекту, виділено цінність проекту та розробки. Проведено аналіз зацікавлених сторін, також зведено доходи та витрати на початку роботи проекту та впровадження.
- За результатами роботи програмного емулятору, що був розроблений в рамках даної роботи, можна зробити висновок, що розроблена архітектура на основі розробленого методу підвищення ефективності роботи СПД, завдяки використанню ВАД, є більш оптимальною та менш апаратно затратною, а також працює швидше за аналогічні системи.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Kapoor, Harsh, et al. "Systems and methods for processing data flows." U.S. Patent Application No. 11/610,296.
2. Belser, David, et al. "Aggregation of mac data flows through pre-established path between ingress and egress switch to reduce number of number connections." U.S. Patent No. 6,151,324. 21 Nov. 2000.
3. Kapoor, Harsh, et al. "Systems and methods for processing data flows." U.S. Patent No. 8,135,657. 13 Mar. 2012.
4. Грекул, Владимир Иванович, Галина Николаевна Денищенко, and Нина Леонидовна Коровкина. "Проектирование информационных систем." М.: Интернет-университет информационных технологий–ИНТУИТ. ру(2005).
5. Афонин, Сергей Иванович, et al. "МЕТОДЫ РЕШЕНИЯ ЗАДАЧ РАСПРЕДЕЛЕНИЯ ИНФОРМАЦИОННЫХ ПОТОКОВ В СЕТИ ПЕРЕДАЧИ ДАННЫХ ПРЕДПРИЯТИЯ НА ОСНОВЕ РЕЗЕРВИРОВАНИЯ РЕСУРСОВ." Информационные системы и технологии 1 (2012): 78-84.
6. Дунець, Р. Б., and Д. Я. Тиханський. "Дослідження часткової реконфігурації ПЛІС." Радіоелектронні і комп'ютерні системи 6 (2009): 240-244.
7. Таненбаум Э. Современные операционные системы / Таненбаум Э., Бос Х : пер. с англ — СПб.: БХВ-Петербург, 2015. – 123 с.
8. Джейн, Анил К., Жианчанг Мао, and К. М. Моиуддин. "Введение в искусственные нейронные сети." Открытые си стемы 4.97 (1997): 16-24.
9. Мартышкин, А. И. "Разработка аппаратного буферного устройства памяти многопроцессорной системы." Фундаментальные исследования 12-3 (2015): 485-489.
10. Зотов, Валерий. "Организация памяти микропроцессорного ядра MicroBlaze." Компоненты и технологии 40 (2004).
11. Синельникова, А. С., Е. А. Чентаева, and И. И. Кищенко. "МНОГОПРОЦЕССОРНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ." П 33 ПРОБЛЕМЫ И ПЕРСПЕКТИВЫ ТЕХНИЧЕСКИХ НАУК (2015): 36.
12. Четверушкин, Б. Н. "МНОГОПРОЦЕССОРНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ." ВЕСТНИК РОССИЙСКОЙ АКАДЕМИИ НАУК 72.9 (2002): 2.
13. «Xilinx». [Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/Xilinx — Дата доступу : Листопад 2017.
14. «Altera». [Електронний ресурс]. — Режим доступу: https://ru.wikipedia.org/wiki/Altera — Дата доступу : Листопад 2017.

15. «Altera». [Електронний ресурс]. — Режим доступу: https://www.altera.com/ — Дата доступу : Листопад 2017.
16. Каляев, Анатолий Васильевич, and Илья Израилевич Левин. Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений. Янус-К, 2003.
17. Николаев, А. Б., and Виктор Сергеевич Подлазов. "Отказоустойчивое расширение системных сетей многопроцессорных вычислительных систем." Автоматика и телемеханика 1 (2008): 162-170.
18. Каравай, Михаил Федорович, and Виктор Сергеевич Подлазов. "Расширенный обобщенный гиперкуб как отказоустойчивая системная сеть для многопроцессорных систем." Управление большими системами: сборник трудов45 (2013).
19. Подлазов, В. С., and В. В. Соколов. "Метод однородного расширения сетей связи многопроцессорных вычислительных систем." Проблемы управления 2 (2007).
20. Гуров, В.О. Прискорення динамічного розподілу завдань в мультипроцесорних система [Текст] / В.О. Гуров, В.В. Жабіна // VIII Міжнародній конференції студентів і молодих учених "Сучасні Інформаційні Технології 2018" — Одеса, 2018 — С. 1-5.
21. Гуров, В.О. Модель реалізації крупнозернистого паралелізму в мультипроцесорних система [Текст] / В.О. Гуров, В.В. Жабіна // Наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 — Київ, 2018. — С. 1-5.
22. Романкевич, А. М., et al. "Об одном подходе к расчету надежности отказоустойчивых многопроцессорных систем." Автоматизированные системы управления и приборы автоматики 119 (2002): 54-58.
23. Ильюшенко, Н. В., А. В. Уланович, and В. А. Селезнев. "Объемное моделирование и прототипирование в литейном производстве." Современные наукоемкие технологии 8-2 (2013): 198-200.
24. Кальдин, Дмитрий Александрович, et al. "Оптимизация алгоритмов компьютерного моделирования трехмерного физического пространства." Современные информационные технологии и ИТ-образование 8 (2012).
25. Тарасик В.П. Математическое моделирование технических систем [Текст] / В.П.Тарасик. — Мн.: Дизайн ПРО, 2004. — 640 с.
26. Вержбицкий, Валентин Михайлович. Численные методы (математический анализ и обыкновенные дифференциальные уравнения). Издательский дом" ОНИКС 21

век", 2005.Cheng, Guo-Xiong, and Shi-Qing Hu. "System architecture and pattern research of RIA based on silverlight [J]." Computer Engineering and Design 8.31 (2010): 1706-1709.

27. Kishore, MVVM Satya, et al. "The rock salt oxide Li2MgTiO4: Type I dielectric and ionic conductor." Materials research bulletin41.7 (2006): 1378-1384.
28. Богатырев, В. А. "Отказоустойчивость распределенных вычислительных систем динамического распределения запросов и размещение функциональных ресурсов." Наука и образование: научное издание МГТУ им. НЭ Баумана 1 (2006).
29. Бутковский, Анатолий Григорьевич. Методы управления системами с распределенными параметрами/АГ Бутковский. Наука, 1975.
30. Земляков, Станислав Данилович, Владислав Юльевич Рутковский, and Андрей Владимирович Силаев. "Реконфигурация систем управления летательными аппаратами при отказах." Автоматика и телемеханика 1 (1996): 3-20.
31. Каляев, Игорь Анатольевич, and Эдуард Всеволодович Мельник. "Децентрализованные системы компьютерного управления." (2011).
32. Николайчук, Я. М., О. І. Волинський, and С. В. Кулина. "Швидкодіючий алгоритм та процесор порівняння чисел у системі залишкових класів базису Крестенсона." (2008).
33. Coleman, Nicholas, et al. "Immunological events in regressing genital warts." American journal of clinical pathology 102.6 (1994): 768-774.
34. Palnitkar, Samir. Verilog HDL: a guide to digital design and synthesis. Vol. 1. Prentice Hall Professional, 2003.
35. Nikhil, Rishiyur. "Bluespec System Verilog: efficient, correct RTL from high level specifications." Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on. IEEE, 2004.
36. Клименко, И. А., and В. В. Жабина. "Обеспечение отказоустойчивости потоковых систем на однотипных вычислительных модулях." (2009).
37. Репин, Владимир, and Виталий Елиферов. Процессный подход к управлению. Моделирование бизнес-процессов. Манн, Иванов и Фербер, 2004.

ДОДАТКИ

Додаток 1

Лістинги машини станів

```
using Syncfusion.UI.Xaml.Diagram;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using EmulatorB.Elements;
using System.ComponentModel;

namespace EmulatorB
{


    public partial class Canva : UserControl, INotifyPropertyChanged
    {

        private string STEP_;
        public string STEP
        {
            get { return STEP_; }
            set { STEP_ = value; OnPropertyChanged("STEP"); }
        }
        List<ProcessorVisualElement> Processors = new
List<ProcessorVisualElement>();
        List<InputVisualElement> Inputs = new List<InputVisualElement>();
        List<TimerVisualElement> Timers = new List<TimerVisualElement>();
        BufferDataMemoryVisualElement BufferData = new
BufferDataMemoryVisualElement();
        RDVisualElement RD = new RDVisualElement();
        RAVisualElement RA = new RAVisualElement();
        ActorMemoryVisualElement Actors = new ActorMemoryVisualElement();
        OperandMemoryVisualElement Operands = new OperandMemoryVisualElement();
        BufferCommandMemoryVisualElement BufferCommand = new
BufferCommandMemoryVisualElement();
        List<OutputVisualElement> Outputs = new List<OutputVisualElement>();
        List<LineCSV> lines;
        int last;
        int processors;
        int inputs = 3;
        int outputs = 3;
        public Canva(int counter, List<LineCSV> lines)
        {
            DataContext = this;
            this.lines = lines;
            last = lines.Last().num;
            processors = counter;
            InitializeComponent();

            InitProcessors();
            InitComutator();
            InitConnectionsComutatorProcessors();
            InitInputs();
            InitConnectionsInputsComutator();
            InitBufferMemoryData();
            InitConnectionsComutatorBufferMemoryData();
```

```
            InitRD();
            InitRA();
            InitConnectionsBufferMemoryDataRD();
            InitConnectionsBufferMemoryDataRA();
            InitActorMemory();
            InitOperandMemory();
            InitConnectionsRDACtors();
            InitConnectionsRAOperands();
            InitConnectionsRAActors();
            InitConnectionsRDOperands();
            InitBufferCommandMemory();
            InitConnectionsRABuffferComamnd();
            InitConnectionsRDBufferCommand();
            InitConnectionsActorsMemoryCommand();
            InitConnectionsOperandsMemoryCommand();
            InitTimers();
            InitConnectionsTimersProcessors();
            InitConnectionsMemoryCommandTimers();
            InitOutputs();
            InitConnectionsMemoryCommandOutputs();

        }


        public void InitProcessors()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel Proc = new NodeViewModel();
                Proc.OffsetX = 1000;
                Proc.OffsetY = 50 + 100 * i;
                Proc.ID = "proc" + i;

                ProcessorVisualElement PVE = new
ProcessorVisualElement("Processor " + i);
                Proc.Content = PVE;
                NodeCollection.Add(Proc);
                Processors.Add(PVE);
            }
        }

        public void InitComutator()
        {
            NodeViewModel Comutator = new NodeViewModel();
            Comutator.OffsetX = 400;
            Comutator.OffsetY = 100;
            Comutator.ID = "Comutator";
            Comutator.Content = new ComutatorVisualElement();
            NodeCollection.Add(Comutator);
        }

        public void InitInputs()
        {
            for (int i = 0; i < inputs; i++)
            {
                NodeViewModel Input = new NodeViewModel();
                Input.OffsetX = 180 + 170 * i;
                Input.OffsetY = 10;
                Input.ID = "input" + i;
                InputVisualElement IVE = new InputVisualElement("Input " + i);
                Inputs.Add(IVE);
                Input.Content = IVE;
                NodeCollection.Add(Input);
            }
```

```
        }

        public void InitTimers()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel timer = new NodeViewModel();
                timer.OffsetX = 1200;
                timer.OffsetY = 50 + 100 * i;
                timer.ID = "timer" + i;
                TimerVisualElement IVE = new TimerVisualElement();
                Timers.Add(IVE);
                timer.Content = IVE;
                NodeCollection.Add(timer);
            }
        }

        public void InitOutputs()
        {
            for (int i = 0; i < outputs; i++)
            {
                NodeViewModel output = new NodeViewModel();
                output.OffsetX = 400 + i * 170;
                output.OffsetY = 800;
                output.ID = "output" + i;
                OutputVisualElement IVE = new OutputVisualElement("Output " +
i);

                Outputs.Add(IVE);
                output.Content = IVE;
                NodeCollection.Add(output);
            }
        }

        public void InitBufferMemoryData()
        {
            NodeViewModel Buffer = new NodeViewModel();
            Buffer.OffsetX = 200;
            Buffer.OffsetY = 250;
            Buffer.ID = "BufferMemoryData";
            Buffer.Content = BufferData;
            NodeCollection.Add(Buffer);
        }

        public void InitRD()
        {
            NodeViewModel RD = new NodeViewModel();
            RD.OffsetX = 100;
            RD.OffsetY = 450;
            RD.ID = "RD";
            RD.Content = this.RD;
            NodeCollection.Add(RD);
        }

        public void InitRA()
        {
            NodeViewModel RA = new NodeViewModel();
            RA.OffsetX = 300;
            RA.OffsetY = 400;
            RA.ID = "RA";
            RA.Content = this.RA;
            NodeCollection.Add(RA);
        }

        public void InitActorMemory()
```

```
        {
            NodeViewModel ActorMemory = new NodeViewModel();
            ActorMemory.OffsetX = 500;
            ActorMemory.OffsetY = 600;
            ActorMemory.ID = "Actors";
            ActorMemory.Content = Actors;
            NodeCollection.Add(ActorMemory);
        }

        public void InitOperandMemory()
        {
            NodeViewModel OperandMemory = new NodeViewModel();
            OperandMemory.OffsetX = 150;
            OperandMemory.OffsetY = 600;
            OperandMemory.ID = "Operands";
            OperandMemory.Content = Operands;
            NodeCollection.Add(OperandMemory);
        }

        public void InitBufferCommandMemory()
        {
            NodeViewModel CommandMemory = new NodeViewModel();
            CommandMemory.OffsetX = 150;
            CommandMemory.OffsetY = 800;
            CommandMemory.ID = "BufferMemoryCommand";
            CommandMemory.Content = BufferCommand;
            NodeCollection.Add(CommandMemory);
        }


        void InitConnectionsComutatorProcessors()
        {
            for (int i = 0; i < processors; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "proc" + i;
                CVM.TargetNodeID = "Comutator";
                CVM.ConnectorGeometryStyle   =   this.Resources["ConnectorStyle"]
as Style;
                ConectorCollection.Add(CVM);
            }
        }

        void InitConnectionsInputsComutator()
        {
            for (int i = 0; i < inputs; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "input" + i;
                CVM.TargetNodeID = "Comutator";
                CVM.ConnectorGeometryStyle                                    =
this.FindResource("ConnectorStyle") as Style;

                ConectorCollection.Add(CVM);
            }
        }

        void InitConnectionsComutatorBufferMemoryData()
        {
            for (int i = 0; i < inputs; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "Comutator";
                CVM.TargetNodeID = "BufferMemoryData";
```

```
                    CVM.ConnectorGeometryStyle                                  =
this.FindResource("ConnectorStyle") as Style;
                    ConectorCollection.Add(CVM);
                }
            }

            void InitConnectionsBufferMemoryDataRA()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "BufferMemoryData";
                CVM.TargetNodeID = "RA";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsBufferMemoryDataRD()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "BufferMemoryData";
                CVM.TargetNodeID = "RD";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRDACtors()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RD";
                CVM.TargetNodeID = "Actors";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRDOperands()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RD";
                CVM.TargetNodeID = "Operands";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRAActors()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RA";
                CVM.TargetNodeID = "Actors";


                var st = FindResource("ConnectorStyle") as Style;
                CVM.ConnectorGeometryStyle  =  this.Resources["ConnectorStyle"]  as
Style;
                ConectorCollection.Add(CVM);

            }
```

```
                void InitConnectionsRAOperands()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RA";
                    CVM.TargetNodeID = "Operands";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsOperandsMemoryCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "Operands";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsActorsMemoryCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "Actors";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsRDBufferCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RD";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsTimersProcessors()
                {
                    for (int i = 0; i < processors; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "timer" + i;
                        CVM.TargetNodeID = "proc" + i;
                        CVM.ConnectorGeometryStyle                                    =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsMemoryCommandTimers()
                {
```

```
                    for (int i = 0; i < processors; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "BufferMemoryCommand";
                        CVM.TargetNodeID = "timer" + i;
                        CVM.ConnectorGeometryStyle                                          =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsMemoryCommandOutputs()
                {
                    for (int i = 0; i < outputs; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "BufferMemoryCommand";
                        CVM.TargetNodeID = "output" + i;
                        CVM.ConnectorGeometryStyle                                          =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsRABuffferComamnd()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RA";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                int takt = 0;
                int globaltakt = 0;
                int state = 0;
                LineCSV line;
                private void Button_Click(object sender, RoutedEventArgs e)
                {
                    MachineTackt();
                }

                private bool Fool()
                {
                    foreach (var c in Processors)
                    {
                        if (c.isfree)
                            return true;
                    }
                    return false;
                }
                int lastline = 100;
                private void MachineTackt()
                {
                    if (Outputs[0].rdy)
                    {
                        MessageBox.Show("END");
                        return;
                    }
```

```
if (takt == last)
{
    foreach (var c in Processors)
    {
        c.Progress.Value = 0;
        c.Clear();

    }
    BufferData.Clear1();
    BufferData.Clear2();
    BufferCommand.Clear1();
    BufferCommand.Clear2();
    RD.Clear();
    WriteOutPut();
    globaltakt++;
    STEP = globaltakt.ToString();
    return;
}
line = lines[takt];

checkProcessors();
checkTimers();
if (!Fool())
{
    globaltakt++;
    return;
}

switch (state)
{
    case 0:
        {
            if (takt >= 6)
            {
                clearInputs();
            }
            else
                checkInputs();
            state = 1;
            break;
        }
    case 1:
        {
            setBuffer();
            state = 2;
            break;
        }
    case 2:
        {
            SetRaRd();
            SetActorOperand();
            state = 3;
            break;
        }
    case 3:
        {
            SetBufferMmeory();
            //  SetActorOperand();
            state = 4;
            break;
        }
    case 4:
        {
```

```
                            if (takt == last)
                            {

                            }
                            else if (lastline != takt)
                            {
                                setTimers();
                                SetProc();
                                lastline = takt;
                            }
                            state = 5;
                            break;
                        }
                    case 5:
                        {
                            takt++;
                            state = 0;
                            break;
                        }
                    default:
                        {

                            break;
                        }
                }

                globaltakt++;
                STEP = globaltakt.ToString();
            }

            private void clearInputs()
            {
                Inputs[0].WRITE("", "", "0", "", "", "0");
                Inputs[1].WRITE("", "", "0", "", "", "0");
                Inputs[2].WRITE("", "", "0", "", "", "0");
            }
            private void checkInputs()
            {
                Inputs[0].WRITE("1",            line.num.ToString(),            "0",
line.name.ToString(), line.next.ToString(), "0");
                Inputs[1].WRITE("1",            line.num.ToString(),            "0",
line.operand1.ToString(), line.next.ToString(), "0");
                Inputs[2].WRITE("1",            line.num.ToString(),            "0",
line.operand2.ToString(), line.next.ToString(), "0");
            }


            void setBuffer()
            {
                Random rand = new Random();
                int a = rand.Next(0, 1);
                if (BufferData.is1empty)
                {
                    if (a == 0)
                        BufferData.Set1(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), "", "0");
                    else
                        BufferData.Set1(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), line.next.ToString(), "0");
                }
                else if (BufferData.is2empty)
                {
                    if (a == 0)
```

```
                        BufferData.Set2(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), "", "0");
                    else
                        BufferData.Set2(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), line.next.ToString(), "0");
                }
                else
                {
                    BufferData.Clear2();
                    BufferData.Clear1();
                }
            }

            void SetRaRd()
            {
                Random rand = new Random();
                int a = rand.Next(0, 2);
                RA.SETRA(takt.ToString());
                if (a == 0)
                {
                    RD.SetRD(a.ToString(),                line.operand1.ToString(),
line.name.ToString(), "", "0", "Operand");

                }
                else if (a == 1)
                {
                    RD.SetRD(a.ToString(),                    line.num.ToString(),
line.name.ToString(), line.next.ToString(), "0", "Actor");
                }
                else if (a == 2)
                {
                    RD.SetRD("", "", "", "", "", "Free");
                }


            }

            public void WriteOutPut()
            {
                Outputs[0].MoveProgress(lines.Last().operand1.ToString());
            }

            public void SetActorOperand()
            {
                Actors.AddStroke(line.num.ToString(),          line.name.ToString(),
line.next.ToString(), "0");
                Operands.AddStroke(line.num.ToString(),    line.operand1.ToString(),
"0");
                Operands.AddStroke(line.num.ToString(),    line.operand2.ToString(),
"0");


            }

            int lastTime;
            public void SetBufferMmeory()
            {
                Random rand = new Random();
                int t = rand.Next(10, 20);
                lastTime = t;
                BufferCommand.Set1(line.name.ToString(),   line.operand1.ToString(),
line.operand2.ToString(), line.next.ToString(), "0", t.ToString());
            }
```

```
                int currentProcessor = 0;
                void setTimers()
                {
                    if (currentProcessor >= Processors.Count)
                    {
                        currentProcessor = 0;
                        return;
                    }
                    if (!Processors[currentProcessor].isWork)
                    {
                        Timers[currentProcessor].SetTime("***");
                        return;
                    }
                    Timers[currentProcessor].SetTime(lastTime.ToString());
                }
                void SetProc()
                {

                    if (currentProcessor >= Processors.Count)
                    {
                        currentProcessor = 0;
                        return;
                    }
                    if (!Processors[currentProcessor].isWork)
                    {
                        currentProcessor++;
                        return;
                    }
                    Processors[currentProcessor].SetProc("0", line.num.ToString(), "0",
line.operand1.ToString(), "0");
                    Processors[currentProcessor].MoveProgress();


                    currentProcessor++;
                }


                void checkProcessors()
                {
                    foreach (var c in Processors)
                    {
                        if (!c.isfree && c.isWork)
                        {
                            c.MoveProgress();
                        }
                    }
                    //  return false;
                }
                void checkTimers()
                {
                     for(int i =0; i < processors; i ++)
                     {
                         Timers[i].Decrement();
                     }
                }

                public event PropertyChangedEventHandler PropertyChanged;
                public void OnPropertyChanged(string prop = "")
                {
                    if (PropertyChanged != null)
                        PropertyChanged(this, new PropertyChangedEventArgs(prop));
                }
            }
        }
```

```
using Syncfusion.UI.Xaml.Diagram;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using EmulatorB.Elements;
using System.ComponentModel;

namespace EmulatorB
{


    public partial class Canva : UserControl, INotifyPropertyChanged
    {

        private string STEP_;
        public string STEP
        {
            get { return STEP_; }
            set { STEP_ = value; OnPropertyChanged("STEP"); }
        }
        List<ProcessorVisualElement> Processors = new
List<ProcessorVisualElement>();
        List<InputVisualElement> Inputs = new List<InputVisualElement>();
        List<TimerVisualElement> Timers = new List<TimerVisualElement>();
        BufferDataMemoryVisualElement BufferData = new
BufferDataMemoryVisualElement();
        RDVisualElement RD = new RDVisualElement();
        RAVisualElement RA = new RAVisualElement();
        ActorMemoryVisualElement Actors = new ActorMemoryVisualElement();
        OperandMemoryVisualElement Operands = new OperandMemoryVisualElement();
        BufferCommandMemoryVisualElement BufferCommand = new
BufferCommandMemoryVisualElement();
        List<OutputVisualElement> Outputs = new List<OutputVisualElement>();
        List<LineCSV> lines;
        int last;
        int processors;
        int inputs = 3;
        int outputs = 3;
        public Canva(int counter, List<LineCSV> lines)
        {
            DataContext = this;
            this.lines = lines;
            last = lines.Last().num;
            processors = counter;
            InitializeComponent();

            InitProcessors();
            InitComutator();
            InitConnectionsComutatorProcessors();
            InitInputs();
            InitConnectionsInputsComutator();
            InitBufferMemoryData();
            InitConnectionsComutatorBufferMemoryData();
```

```
            InitRD();
            InitRA();
            InitConnectionsBufferMemoryDataRD();
            InitConnectionsBufferMemoryDataRA();
            InitActorMemory();
            InitOperandMemory();
            InitConnectionsRDACtors();
            InitConnectionsRAOperands();
            InitConnectionsRAActors();
            InitConnectionsRDOperands();
            InitBufferCommandMemory();
            InitConnectionsRABuffferComamnd();
            InitConnectionsRDBufferCommand();
            InitConnectionsActorsMemoryCommand();
            InitConnectionsOperandsMemoryCommand();
            InitTimers();
            InitConnectionsTimersProcessors();
            InitConnectionsMemoryCommandTimers();
            InitOutputs();
            InitConnectionsMemoryCommandOutputs();

        }


        public void InitProcessors()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel Proc = new NodeViewModel();
                Proc.OffsetX = 1000;
                Proc.OffsetY = 50 + 100 * i;
                Proc.ID = "proc" + i;

                ProcessorVisualElement PVE = new
ProcessorVisualElement("Processor " + i);
                Proc.Content = PVE;
                NodeCollection.Add(Proc);
                Processors.Add(PVE);
            }
        }

        public void InitComutator()
        {
            NodeViewModel Comutator = new NodeViewModel();
            Comutator.OffsetX = 400;
            Comutator.OffsetY = 100;
            Comutator.ID = "Comutator";
            Comutator.Content = new ComutatorVisualElement();
            NodeCollection.Add(Comutator);
        }

        public void InitInputs()
        {
            for (int i = 0; i < inputs; i++)
            {
                NodeViewModel Input = new NodeViewModel();
                Input.OffsetX = 180 + 170 * i;
                Input.OffsetY = 10;
                Input.ID = "input" + i;
                InputVisualElement IVE = new InputVisualElement("Input " + i);
                Inputs.Add(IVE);
                Input.Content = IVE;
                NodeCollection.Add(Input);
            }
```

```
        }

        public void InitTimers()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel timer = new NodeViewModel();
                timer.OffsetX = 1200;
                timer.OffsetY = 50 + 100 * i;
                timer.ID = "timer" + i;
                TimerVisualElement IVE = new TimerVisualElement();
                Timers.Add(IVE);
                timer.Content = IVE;
                NodeCollection.Add(timer);
            }
        }

        public void InitOutputs()
        {
            for (int i = 0; i < outputs; i++)
            {
                NodeViewModel output = new NodeViewModel();
                output.OffsetX = 400 + i * 170;
                output.OffsetY = 800;
                output.ID = "output" + i;
                OutputVisualElement IVE = new OutputVisualElement("Output " +
i);

                Outputs.Add(IVE);
                output.Content = IVE;
                NodeCollection.Add(output);
            }
        }

        public void InitBufferMemoryData()
        {
            NodeViewModel Buffer = new NodeViewModel();
            Buffer.OffsetX = 200;
            Buffer.OffsetY = 250;
            Buffer.ID = "BufferMemoryData";
            Buffer.Content = BufferData;
            NodeCollection.Add(Buffer);
        }

        public void InitRD()
        {
            NodeViewModel RD = new NodeViewModel();
            RD.OffsetX = 100;
            RD.OffsetY = 450;
            RD.ID = "RD";
            RD.Content = this.RD;
            NodeCollection.Add(RD);
        }

        public void InitRA()
        {
            NodeViewModel RA = new NodeViewModel();
            RA.OffsetX = 300;
            RA.OffsetY = 400;
            RA.ID = "RA";
            RA.Content = this.RA;
            NodeCollection.Add(RA);
        }

        public void InitActorMemory()
```

```
        {
            NodeViewModel ActorMemory = new NodeViewModel();
            ActorMemory.OffsetX = 500;
            ActorMemory.OffsetY = 600;
            ActorMemory.ID = "Actors";
            ActorMemory.Content = Actors;
            NodeCollection.Add(ActorMemory);
        }

        public void InitOperandMemory()
        {
            NodeViewModel OperandMemory = new NodeViewModel();
            OperandMemory.OffsetX = 150;
            OperandMemory.OffsetY = 600;
            OperandMemory.ID = "Operands";
            OperandMemory.Content = Operands;
            NodeCollection.Add(OperandMemory);
        }

        public void InitBufferCommandMemory()
        {
            NodeViewModel CommandMemory = new NodeViewModel();
            CommandMemory.OffsetX = 150;
            CommandMemory.OffsetY = 800;
            CommandMemory.ID = "BufferMemoryCommand";
            CommandMemory.Content = BufferCommand;
            NodeCollection.Add(CommandMemory);
        }


        void InitConnectionsComutatorProcessors()
        {
            for (int i = 0; i < processors; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "proc" + i;
                CVM.TargetNodeID = "Comutator";
                CVM.ConnectorGeometryStyle   =   this.Resources["ConnectorStyle"]
as Style;
                ConectorCollection.Add(CVM);
            }
        }

        void InitConnectionsInputsComutator()
        {
            for (int i = 0; i < inputs; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "input" + i;
                CVM.TargetNodeID = "Comutator";
                CVM.ConnectorGeometryStyle                                    =
this.FindResource("ConnectorStyle") as Style;

                ConectorCollection.Add(CVM);
            }
        }

        void InitConnectionsComutatorBufferMemoryData()
        {
            for (int i = 0; i < inputs; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "Comutator";
                CVM.TargetNodeID = "BufferMemoryData";
```

```
                    CVM.ConnectorGeometryStyle                                          =
this.FindResource("ConnectorStyle") as Style;
                    ConectorCollection.Add(CVM);
                }
            }

            void InitConnectionsBufferMemoryDataRA()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "BufferMemoryData";
                CVM.TargetNodeID = "RA";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsBufferMemoryDataRD()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "BufferMemoryData";
                CVM.TargetNodeID = "RD";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRDACtors()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RD";
                CVM.TargetNodeID = "Actors";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRDOperands()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RD";
                CVM.TargetNodeID = "Operands";
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsRAActors()
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "RA";
                CVM.TargetNodeID = "Actors";


                var st = FindResource("ConnectorStyle") as Style;
                CVM.ConnectorGeometryStyle  =  this.Resources["ConnectorStyle"]  as
Style;
                ConectorCollection.Add(CVM);

            }
```

```
                void InitConnectionsRAOperands()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RA";
                    CVM.TargetNodeID = "Operands";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsOperandsMemoryCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "Operands";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsActorsMemoryCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "Actors";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsRDBufferCommand()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RD";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                void InitConnectionsTimersProcessors()
                {
                    for (int i = 0; i < processors; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "timer" + i;
                        CVM.TargetNodeID = "proc" + i;
                        CVM.ConnectorGeometryStyle                                    =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsMemoryCommandTimers()
                {
```

```
                    for (int i = 0; i < processors; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "BufferMemoryCommand";
                        CVM.TargetNodeID = "timer" + i;
                        CVM.ConnectorGeometryStyle                                          =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsMemoryCommandOutputs()
                {
                    for (int i = 0; i < outputs; i++)
                    {
                        ConnectorViewModel CVM = new ConnectorViewModel();
                        CVM.SourceNodeID = "BufferMemoryCommand";
                        CVM.TargetNodeID = "output" + i;
                        CVM.ConnectorGeometryStyle                                          =
this.FindResource("ConnectorStyle") as Style;

                        ConectorCollection.Add(CVM);
                    }
                }

                void InitConnectionsRABuffferComamnd()
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "RA";
                    CVM.TargetNodeID = "BufferMemoryCommand";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);

                }

                int takt = 0;
                int globaltakt = 0;
                int state = 0;
                LineCSV line;
                private void Button_Click(object sender, RoutedEventArgs e)
                {
                    MachineTackt();
                }

                private bool Fool()
                {
                    foreach (var c in Processors)
                    {
                        if (c.isfree)
                            return true;
                    }
                    return false;
                }
                int lastline = 100;
                private void MachineTackt()
                {
                    if (Outputs[0].rdy)
                    {
                        MessageBox.Show("END");
                        return;
                    }
```

```
if (takt == last)
{
    foreach (var c in Processors)
    {
        c.Progress.Value = 0;
        c.Clear();

    }
    BufferData.Clear1();
    BufferData.Clear2();
    BufferCommand.Clear1();
    BufferCommand.Clear2();
    RD.Clear();
    WriteOutPut();
    globaltakt++;
    STEP = globaltakt.ToString();
    return;
}
line = lines[takt];

checkProcessors();
checkTimers();
if (!Fool())
{
    globaltakt++;
    return;
}

switch (state)
{
    case 0:
        {
            if (takt >= 6)
            {
                clearInputs();
            }
            else
                checkInputs();
            state = 1;
            break;
        }
    case 1:
        {
            setBuffer();
            state = 2;
            break;
        }
    case 2:
        {
            SetRaRd();
            SetActorOperand();
            state = 3;
            break;
        }
    case 3:
        {
            SetBufferMmeory();
            //  SetActorOperand();
            state = 4;
            break;
        }
    case 4:
        {
```

```
                            if (takt == last)
                            {

                            }
                            else if (lastline != takt)
                            {
                                setTimers();
                                SetProc();
                                lastline = takt;
                            }
                            state = 5;
                            break;
                        }
                    case 5:
                        {
                            takt++;
                            state = 0;
                            break;
                        }
                    default:
                        {

                            break;
                        }
                }

                globaltakt++;
                STEP = globaltakt.ToString();
            }

            private void clearInputs()
            {
                Inputs[0].WRITE("", "", "0", "", "", "0");
                Inputs[1].WRITE("", "", "0", "", "", "0");
                Inputs[2].WRITE("", "", "0", "", "", "0");
            }
            private void checkInputs()
            {
                Inputs[0].WRITE("1",            line.num.ToString(),            "0",
line.name.ToString(), line.next.ToString(), "0");
                Inputs[1].WRITE("1",            line.num.ToString(),            "0",
line.operand1.ToString(), line.next.ToString(), "0");
                Inputs[2].WRITE("1",            line.num.ToString(),            "0",
line.operand2.ToString(), line.next.ToString(), "0");
            }


            void setBuffer()
            {
                Random rand = new Random();
                int a = rand.Next(0, 1);
                if (BufferData.is1empty)
                {
                    if (a == 0)
                        BufferData.Set1(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), "", "0");
                    else
                        BufferData.Set1(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), line.next.ToString(), "0");
                }
                else if (BufferData.is2empty)
                {
                    if (a == 0)
```

```
                        BufferData.Set2(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), "", "0");
                    else
                        BufferData.Set2(a.ToString(),           line.num.ToString(),
line.level.ToString(), line.name.ToString(), line.next.ToString(), "0");
                }
                else
                {
                    BufferData.Clear2();
                    BufferData.Clear1();
                }
            }

            void SetRaRd()
            {
                Random rand = new Random();
                int a = rand.Next(0, 2);
                RA.SETRA(takt.ToString());
                if (a == 0)
                {
                    RD.SetRD(a.ToString(),                line.operand1.ToString(),
line.name.ToString(), "", "0", "Operand");

                }
                else if (a == 1)
                {
                    RD.SetRD(a.ToString(),                    line.num.ToString(),
line.name.ToString(), line.next.ToString(), "0", "Actor");
                }
                else if (a == 2)
                {
                    RD.SetRD("", "", "", "", "", "Free");
                }


            }

            public void WriteOutPut()
            {
                Outputs[0].MoveProgress(lines.Last().operand1.ToString());
            }

            public void SetActorOperand()
            {
                Actors.AddStroke(line.num.ToString(),          line.name.ToString(),
line.next.ToString(), "0");
                Operands.AddStroke(line.num.ToString(),    line.operand1.ToString(),
"0");
                Operands.AddStroke(line.num.ToString(),    line.operand2.ToString(),
"0");


            }

            int lastTime;
            public void SetBufferMmeory()
            {
                Random rand = new Random();
                int t = rand.Next(10, 20);
                lastTime = t;
                BufferCommand.Set1(line.name.ToString(),   line.operand1.ToString(),
line.operand2.ToString(), line.next.ToString(), "0", t.ToString());
            }
```

```
        int currentProcessor = 0;
        void setTimers()
        {
            if (currentProcessor >= Processors.Count)
            {
                currentProcessor = 0;
                return;
            }
            if (!Processors[currentProcessor].isWork)
            {
                Timers[currentProcessor].SetTime("***");
                return;
            }
            Timers[currentProcessor].SetTime(lastTime.ToString());
        }
        void SetProc()
        {

            if (currentProcessor >= Processors.Count)
            {
                currentProcessor = 0;
                return;
            }
            if (!Processors[currentProcessor].isWork)
            {
                currentProcessor++;
                return;
            }
            Processors[currentProcessor].SetProc("0", line.num.ToString(), "0",
line.operand1.ToString(), "0");
            Processors[currentProcessor].MoveProgress();


            currentProcessor++;
        }


        void checkProcessors()
        {
            foreach (var c in Processors)
            {
                if (!c.isfree && c.isWork)
                {
                    c.MoveProgress();
                }
            }
            //  return false;
        }
        void checkTimers()
        {
             for(int i =0; i < processors; i ++)
             {
                 Timers[i].Decrement();
             }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged(string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Додаток 2

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

ПРОГРАМНА МОДЕЛЬ ТА МЕТОД ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ФУНКЦІОНУВАННЯ ВІДМОВОСТІЙКОЇ ПОТОКОВОЇ СИСТЕМИ

Виконав: Лукашов Борис Олегович

Науковий керівник: доцент, к.т.н. Жабіна Валентина Валеріївна

Київ – 2018

Актуальність

- Ефективним підходом до проблеми підвищення надійності систем та достовірності результатів обробки інформації є забезпечення відмовостійкості систем. Найбільш економічним з точки зору витрат апаратури є програмні засоби підвищення відмово стійкості, однак для прискорення відновлення обчислень при відмові обладнання створено методи динамічної реконфігурації систем. Але такі методи потребують подальшого дослідження, особливо для потокових систем. Саме тому розробка системи емуляції для тестування відмово стійкості є актуальною задачею.

Об'єкт та предмет досліджень

- **Об’єктом дослідження** в даній роботі є обчислювальні системи, що керуються потоком даних.
- **Предметом дослідження** є методи підвищення ефективності роботи системах, що керуються потоком даних.

Мета

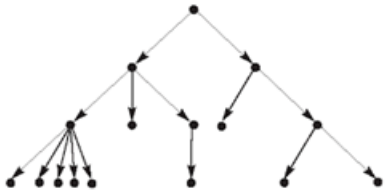
- Метою дослідження є реалізація програмного емулятору відмовостійкої обчислювальної системи, що керуються потоком даних на основі розробленої архітектури та досліджуваного методу.

Задачі

- Аналіз існуючих архітектур СПД та виявлення їх недоліків та переваг;
- Аналіз існуючих підходів до створення програмних моделей СПД;
- Розробка відмовостійкої архітектури СПД на основі досліджуваного методу з використанням вільного адресного доступу;
- Розробка програмного забезпечення для моделювання та подальшого тестування системи;
- Тестування та порівняльний аналіз різних архітектур за допомого розробленого ПЗ.

Вступ

- Системи автоматичного керування та моделювання в реальному часі реалізовують алгоритми дрібнозернистої та крупнозернистої структури.

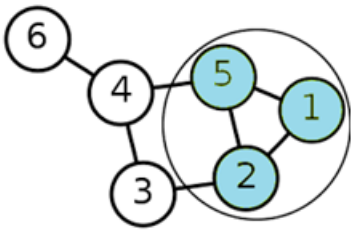


Системи, що керуються потоком даних

- СПД містить кілька обчислювальних модулів (ОМ) та одне середовище формування команд (СФК), зв'язаних комунікаційними засобами. Для реалізації даної моделі обчислень можуть бути використані ПЛІС, які містять обчислювальні ядра, модулі пам'яті і засоби комунікації.

Принцип роботи системи, що керуються потоком даних

- Робота СПД визначається орієнтованим графом потоку даних (ГПД), в якому вершини відповідають операціям, а дуги визначають залежність за даними між операціями.

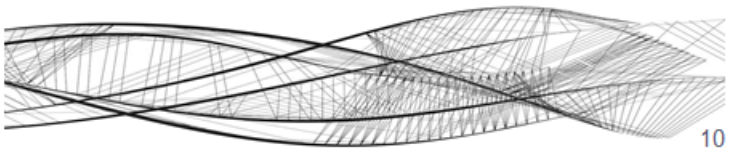


Граф потоку даних

- У графі потоку управління кожен вузол (вершина) графа відповідає базовому блоку - прямолінійній ділянці коду, який не містить в собі ні операцій передачі управління, ні точок, на які управління передається з інших частин програми. Кожній вершині призначається операція (актор), а кожній дузі – дані.

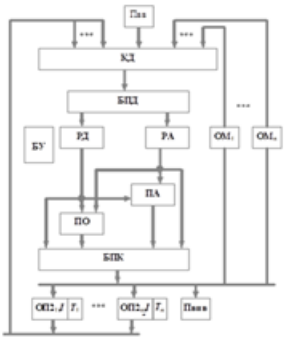
Динамічний розподіл задач

Існує декілька варіантів розпаралелювання, наприклад статичний та динамічний розподіл задач, у випадку статичного, займатися розподілом має розробник, тому актуальним є використання динамічного розподілу задач в системі. Саме принцип динамічного розпаралелювання лежить в основі розроблено архітектури.



Опис архітектури та моделі потокової системи

- СПД містить пристрої введення (Пвв) та виведення (Пвив), обчислювальні модулі ОМ, блок управління (БУ). Буферна пам'ять даних (БПД) і команд (БПК), а також пам'ять операндів (ПО), акторів (ПА), регістри адреси (РА) і даних (РД) утворюють СФК.



Особливості архітектури та новизна

- Використання вільного адресного простору у якості запам'ятовуючого механізму замість асоціативної пам'яті.
- Використання окремих комірок для зберігання акторів та операндів.
- Використання таймерів для засікання роботи обчислювальних блоків.
- Зберігання мета-даних кожної операції.
- Розроблено програмний емулятор роботи системи.
- Розроблено граматику мови для подачі команд СПД.

Існуючі інструменти та емулятори

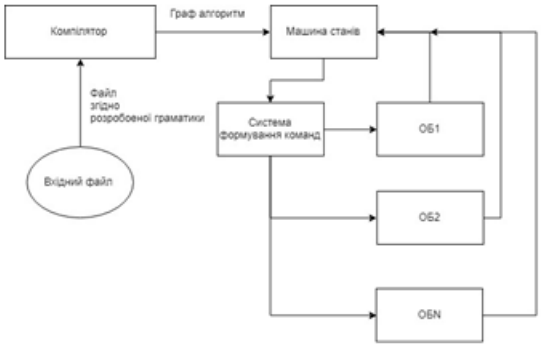
Існуючі IDE:

- Quartus II
- Xlinix

«Мова апаратури»:

- *HDL*

Схема програмного емулятору



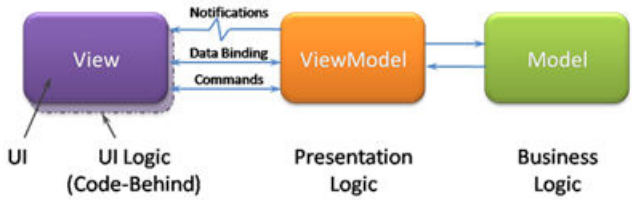
Інструменти

- C#
- Visual Studio
- Devexpress
- DiagramBuilder

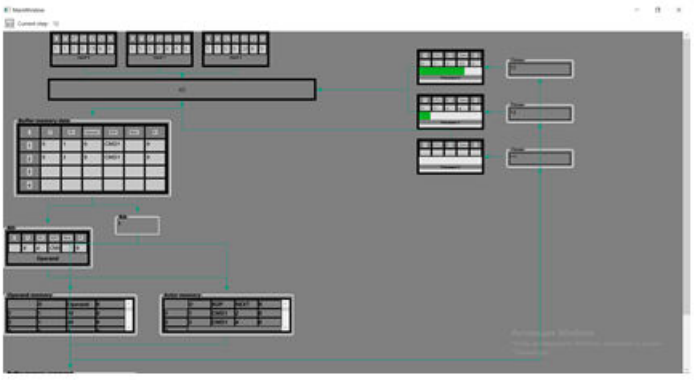
Технології реалізації

- WPF (Windows Presentation Foundation)
- Паттерн MVVM

Model-View-ViewModel

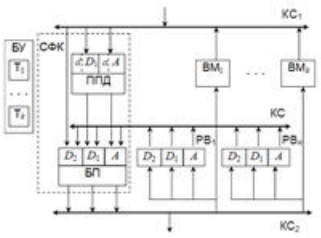


Програмний емулятор роботи системи

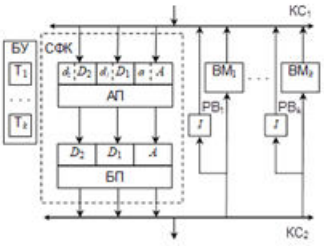


Архітектури аналоги

Архітектура з вільним адресним доступом



Архітектура з асоціативною пам'яттю



Розрахунок надійності архітектури

- $\lambda = \Sigma \lambda i * Ni$

  Де:

  λi - інтенсивність відмов модулів;

  Ni - кількість відповідних компонент.

- Коефіцієнт відновлення $1 / (1 + \lambda * (Tз + Tп / 2))$

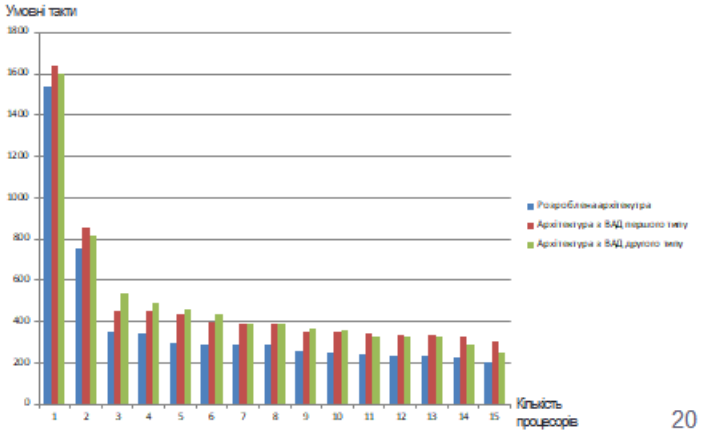
  Де:

  Tз - час реконфігурації.

  Tп - час перевірки обчислювального модуля.

Для розробленої архітектури - 0,99974789

Результати емуляції



Практична цінність роботи

- **Практична цінність** отриманих в роботі результатів полягає в тому, що запропонований метод за ефективністю не поступається існуючим, а в більшості випадків навіть перевершує. Розроблений емулятор дозволяє в повному обсязі протестувати та порівняти результати з аналогами.

Бізнес модель

Доходи і витрати

Висновки

- Було проведено аналіз існуючих підходів до проектування СПД та виявлено їх недоліки та переваги;
- Було проведено аналіз існуючих програмних засобів для емуляції СПД;
- Було розроблено архітектуру на основі методу, що використовую вільний адресний простір та підвищує відмовостійкість системи;
- Розроблено програмне забезпечення для емуляції роботи розробленої архітектури;
- Було проведено порівняльний аналіз існуючих варіантів та розробленої СПД.

- Проведений порівняльний аналіз за допомогою розробленого програмного забезпечення показав, що розроблена архітектура порівняно з аналогами працює та реконфігурується швидше. Результати тестування наведені в таблиці нижче.

	Виграш при справній роботі ОМ (%)	Виграш при поступовому відключенні ОМ (%)	Виграш при поступовому відновленні ОМ (%)
Архітектура з асоц. пам'яттю	0.3	3.1	4.7
Архітектура з вільним адресним доступом	0.07	3.9	5.1

Апробування отриманих результатів

- XI наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» (ПМК-2018-2).

Дякую за увагу!