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

Кваліфікаційна наукова
праця на правах рукопису

ОНАЙ МИКОЛА ВОЛОДИМИРОВИЧ

УДК 004.31:004.27

ДИСЕРТАЦІЯ
МЕТОДИ ТА ЗАСОБИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РЕАЛІЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ ОПЕРАЦІЙ У СКІНЧЕННИХ ПОЛЯХ

Спеціальність 05.13.05 – Комп’ютерні системи та компоненти
Галузь знань – Інформаційні технології

Подається на здобуття наукового ступеня кандидата технічних наук

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

М.В. Онай

Науковий керівник Дичка Іван Андрійович,
доктор технічних наук, професор

Київ – 2017

АНОТАЦІЯ

Онай М.В. Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.05 – Комп'ютерні системи та компоненти (Інформаційні технології). – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, 2017.

Зміст анотації

Дисертаційна робота присвячена проблемі підвищення ефективності обчислень у скінченних полях. Запропоноване вирішення цієї задачі базується на удосконаленні існуючих та розробленні нових методів виконання операцій додавання, віднімання, піднесення до степеня, обчислення мультиплікативно оберненого елемента та ділення у скінченних полях виду $GF(p)$ та $GF(2^m)$; отриманні архітектурних рішень для реалізації запропонованих методів та моделюванні процесів виконання операцій в скінченних полях.

Проведено аналіз сучасного стану розвитку методів виконання операцій у скінченних полях та виділено пріоритетні ознаки, за якими доцільно проводити їх класифікацію. На основі виділених ознак виконано класифікацію методів виконання найбільш обчислювально витратних операцій у скінченних полях, що дало можливість провести ґрунтовне дослідження та сформувати напрямки розвитку зазначених методів.

Операції в скінченних полях можна реалізовувати як програмно, так і апаратно, проте в останні роки особлива увага приділяється задачам прикладного характеру, де обробка інформації відбувається в реальному

часі, що можливо забезпечити за рахунок апаратної реалізації обчислень у скінченних полях. Прикладом таких задач є задачі криптографії з відкритим ключем, цифрової обробки сигналів та завадостійкого кодування даних. У перелічених задачах дуже важливе значення має час виконання операції та секретність передачі даних, що належним чином не можна забезпечити при програмній реалізації. Через специфіку обчислень в полях Галуа їх програмна реалізація з використанням універсальних комп'ютерів та мов програмування високого рівня не завжди забезпечує потрібну швидкість отримання результату.

Встановлено, що для забезпечення високої ефективності обчислень у скінченних полях необхідно застосовувати спеціалізовані обчислювальні засоби. У наш час з розвитком інтегральної схемотехніки, зокрема ПЛІС, з'являються нові можливості для реалізації обчислень у скінченних полях з потрібною швидкодією, досягти якої можливо лише за рахунок апаратної реалізації операцій.

Докладно проаналізовано методи обчислення мультиплікативно оберненого елемента, що ґрунтуються на модулярному піднесенні до степеня, та методи, що ґрунтуються на знаходженні найбільшого спільного дільника (НСД) двох чисел. Встановлено, що менш обчислювально витратними є методи, що ґрунтуються на знаходженні НСД двох чисел.

Запропоновано метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та відповідні структури апаратних засобів для його реалізації. До адитивних операцій належать додавання та віднімання; до мультиплікативних – множення, піднесення до степеня, обчислення мультиплікативно оберненого елемента та ділення.

Внаслідок вивчення стану проблеми високоефективних обчислень у скінченних полях виду $GF(2^m)$ встановлено, що скінченним полям властивий ізоморфізм – елементи поля допускають 4 подання: цілочислове, степеневе, векторне, многочленне. Всі ці подання елементів

поля є еквівалентними, і кожне з них є зручним для реалізації відповідних операцій над елементами.

Операцію додавання елементів поля та обчислення протилежного елемента зручно виконувати над векторним (многочленним) поданням, що у випадку $p = 2$ співпадає з двійковими кодами елементів поля. А операцію множення, обчислення мультиплікативно оберненого елемента, ділення та піднесення до степеня зручно виконувати над степеневим поданням елементів поля, оскільки в цьому випадку необхідно виконувати операції лише над показниками степеня. Однак, степеневе подання елементів поля $GF(2^m)$ має істотний недолік – воно не дозволяє отримувати нульовий елемент поля. Тому під час виконання операцій в $GF(2^m)$ необхідно динамічно, залежно від характеру операції, переходити від однієї форми подання елементів до іншої, і навпаки, тобто оперативно, на апаратному рівні забезпечувати ізоморфізм поля.

Дослідження показали, що за рахунок табличного зберігання елементів поля $GF(2^m)$ у многочленному та степеневому їх поданні забезпечується максимальна швидкодія та універсальність арифметико-логічного пристрою. При використанні операндів великої розрядності запропоновано розріджене формування таблиці елементів поля, що дозволяє у кілька разів скоротити витрати пам’яті для її зберігання. Побудовано алгоритми перетворення степеневого подання у многочленне та многочленного подання у степеневе з використанням розрідженої таблиці. Розроблений метод забезпечує зростання швидкодії на 15% порівняно з існуючим методом.

Запропоновано спосіб отримання верхньої оцінки кількості елементарних однотактних операцій для методів обчислення мультипілкативно оберненого елемента та операції піднесення до степеня в полі $GF(2^m)$. Встановлено, що метод з використанням розрідженої таблиці елементів поля характеризується слабкою залежністю кількості операцій зсуву і порозрядного додавання для виконання операції

обчислення мультиплікативно оберненого елемента та ділення від параметра m поля. Для операції обчислення мультиплікативно оберненого елемента для значення $m = 20$ запропонований метод дає приріст швидкодії приблизно в 4 рази, а для операції піднесення до степеня дає приріст швидкодії майже у 25 разів.

Проведено експериментальні дослідження для визначення оптимального ступеня розрідження таблиці елементів поля $GF(2^m)$. Встановлено, що для значення $m < 21$ оптимальним є ступінь розрідження рівний 8. Для виконання операції піднесення до степеня доцільним є збільшення ступеня розрідження до значення 16.

Досліджено методи піднесення до степеня у скінченних полях, а саме: методи, що ґрунтуються на поданні показника степеня у двійковій системі числення та аналізі бітів показника степеня як по одному, так і по кілька бітів за одну ітерацію роботи алгоритму; методи, що ґрунтуються на поданні показника степеня у симетричній трійковій системі числення (*Non Adjacent Form*), та методи, що грунтуються на поданні показника степеня в системі числення з мультиосновою.

Запропоновано модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном. Відмінність від існуючих методів полягає в тому, що при формуванні таблиці передобчислень використовуються показники степеня, що є простими числами, та при аналізі двійкового подання показника степеня виділяються блоки бітів, що утворюють просте число. Для досягнення високої швидкодії при побудові таблиці передобчислень рекомендовано використовувати заздалегідь обчислені адитивні ланцюжки з метою мінімізації кількості операцій множення, що дозволяє отримувати кожен наступний елемент таблиці за одну-дві операції модулярного множення. Модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном можна застосовувати в еліптичній криптографії для поліпшення часових характеристик операції скалярного множення точки еліптичної кривої на число.

Розроблено модель обчислювального процесу виконання операцій у скінченних полях, яка дозволяє на основі заданих наборів вхідних даних виконувати порівняння методів та здійснювати оптимальний вибір параметрів і форм подання операндів, що забезпечує зростання швидкодії при реалізації обчислювальних операцій на ПЛІС. На основі запропонованої моделі розроблено методики дослідження нових способів апаратної реалізації обчислень у полях Галуа. Моделювання у середовищі розробки *Xilinx ISE* та за допомогою програми *Mentor Graphics Precision* показало, що розроблені структури апаратних засобів характеризуються мінімальною апаратною складністю та високою швидкодією.

Створено генератор *Verilog*-коду для синтезу на ПЛІС елемента *ROM*, що містить розріджену таблицю елементів поля $GF(2^m)$ у многочленному та степеневому їх поданні, який автоматично генерує *Verilog*-код за заданим незвідним многочленом та ступенем розрідження таблиці.

Розроблено архітектуру та систему команд спеціалізованого процесора Галуа, орієнтованого на виконання операцій у скінченних полях. Запропонована система команд дозволяє створювати програми довільної складності на мові Асемблера процесора Галуа з подальшим виконання цих програм функціональними блоками процесора. Система команд включає: арифметичні команди (*ADD*, *MULT*, *DIV*, *POW*, *INVM*, *CDP*, *CPD*, *INVA*, *SUB*, *INC*, *DEC*), команди пересилання даних (*MOV*, *LOAD*, *OUT*) та команди передачі керування (*JMP*, *LOOP*).

Процесор Галуа можна використовувати в універсальній обчислювальній системі як співпроцесор, доповнюючи систему команд центрального процесора, або як спецобчислювач на основі ПЛІС, який порівняно з універсальними обчислювальними засобами підвищує продуктивність обробки інформації в реальному часі. Особливістю архітектури розробленого процесора є те, що не змінюючи інтерфейса процесора, шляхом зміни арифметико-логічного пристрою можна

здійснити перехід до поля Галуа $GF(p)$ або до $GF(2^m)$. Проведено дослідження розробленого процесора Галуа, яке показало, що даний процесор забезпечує зростання продуктивності обчислень на 27% порівняно з універсальними обчислювальними засобами.

Запропоновано програмістську модель процесора Галуа, яка дозволяє створювати програмне забезпечення мовою Асемблера процесора Галуа.

Отже, у дисертаційній роботі вирішено актуальну науково-прикладну задачу – підвищення продуктивності систем цифрової обробки даних, забезпечення завадостійкості зберігання і передачі даних та криптографічних перетворень за рахунок створення ефективних технічних засобів для виконання обчислень у скінченних полях шляхом структурно-логічної оптимізації процесів виконання операцій у полях Галуа.

Ключові слова: поле Галуа, скінченне поле, мультиплікативно обернений елемент, піднесення до степеня, незвідний многочлен, Асемблер Галуа, процесор Галуа, ПЛІС.

Список публікацій здобувача

1. Zhengbing Hu The Analysis and Investigation of Multiplicative Inverse Searching Methods in the Ring of Integers Modulo M [Text] / Zhengbing Hu, I. A. Dychka, Onai Mykola, Bartkoviak Andrii // International Journal of Intelligent Systems and Applications (IJISA), 2016. – Vol. 8, No. 11. – P. 9-18 (Входить до міжнародної наукометричної бази даних SCOPUS).
2. Дичка, І.А. Модифікований віконний метод однократного множення точки еліптичної кривої на скаляр у полі $GF(p)$ [Текст] / І.А. Дичка, М.В. Онай, Т.П. Дрозда // Радіоелектроніка, інформатика, управління. – Запоріжжя. – 2016. – №2. – С. 95-102 (Входить до міжнародної наукометричної бази даних Web of Science).

3. Дичка, І.А. Апаратна реалізація обчислень у скінченних полях характеристики два [Текст] / І.А. Дичка, М.В. Онай, Ю.В. Бухтіяров // Наукові вісті НТУУ “КПІ”. – 2013. – №6. – С. 20-27 (Входить до міжнародної наукометричної бази даних EBSCO).
4. Дичка, І.А. Апаратна реалізація процедур множення і ділення многочленів у скінченних полях [Текст] / І.А. Дичка, В.І. Голуб, М.В. Онай // Наукові вісті НТУУ “КПІ”. – 2012. – №5. – С. 61-66 (Входить до міжнародної наукометричної бази даних EBSCO).
5. Дичка, І.А. Апаратна реалізація операторів та функцій в полях Галуа [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Хмельницького національного університету. – 2012. – Вип. 5. – С. 234-240 (Входить до міжнародної наукометричної бази даних Index Copernicus).
6. Дичка, І.А. Архітектура проблемно-орієнтованого процесора для реалізації арифметики скінченних полів [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №12 (183), ч.2. – С. 99-106.
7. Дичка, І.А. Організація спеціалізованих комп’ютерних систем для реалізації обчислень у скінченних полях [Текст] / І.А. Дичка, В.І. Голуб, М.В. Онай // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №6 (177). – С. 268-278.
8. Дичка, І.А. Подання інформації у графічно-кодованому вигляді: технологія кодування та декодування [Текст] / І.А. Дичка, М.В. Онай, М.В. Новосад // Реєстрація, зберігання і обробка даних (Data Recording, Storage & Processing). – 2010. – Т. 12, №2. – С. 69-80.
9. Державний патент України №111351, МПК G06F 7/00, G06F 7/50. Схема для пошуку мультиплікативно оберненого елемента за довільним модулем [Текст] / Дичка І.А., Онай М.В., Приходько Е.В.; заявник та патентовласник Дичка І.А., Онай М.В., Приходько Е.В. – № u201604179; дата подання заявки 15.04.2016; дата публ. 10.11.2016, бюл. №21, 2016 р. – 7 с.

10. Державний патент України №73309, МПК G06F 7/50. Пристрій для виконання обчислень в полі $GF(2^n)$ [Текст] / Дичка І.А., Онай М.В. ; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201115679; дата подання заявки 30.12.2011; дата публ. 25.09.2012, бюл. №18, 2012 р. – 10 с.
11. Державний патент України №57281, МПК G06F 7/48. Суматор елементів поля $GF(p^m)$ [Текст] / Дичка І.А., Онай М.В.; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201004903; дата подання заявки 23.04.2010; дата публ. 25.02.2011, бюл. №4, 2011 р. – 17 с.
12. Державний патент України №54637, МПК G06F 7/50. Суматор за модулем простого числа [Текст] / Дичка І.А., Онай М.В.; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201001074; дата подання заявки 02.02.2010; дата публ. 25.11.2010, бюл. №22, 2010 р. – 14 с.
13. Онай, М.В. Знаходження мультиплікативно оберненого елемента у кільці лишків за довільним модулем методом Джої-Пейє [Текст] / М.В. Онай, А.Ю. Бартков’як // Міжнародна науково-практична конференція “Проблеми інформатики та комп’ютерної техніки”. Праці конференції. – Чернівці : Видавничий дім “Родовід”, 2015. – С. 91-93.
14. Онай, М.В. Пошук мультиплікативно оберненого елементу у кільці лишків за довільним модулем методами, що грунтуються на модулярному піднесенні до степеня [Текст] / М.В. Онай, А.Ю. Бартков’як // Шістнадцята міжнародна наукова конференція імені академіка Михайла Кравчука, 14-15 травня, 2015 р., Київ : Матеріали конф. Т. 2. Алгебра. Геометрія. Математичний аналіз. – К. : НТУУ “КПІ”, 2015. – С. 139-141.

15. Дичка, І.А. Застосування *k*-арного методу Евкліда для пошуку мультплікативно оберненого елемента у кільці лишків за модулем *m* [Текст] / І.А. Дичка, М.В. Онай, А.Ю. Бартков'як // Матеріали статей П'ятої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Івано-Франківськ : п. Голіней О.М., 2015. – С. 151-153.
16. Онай, М.В. Спосіб прискорення алгоритмів множення точки еліптичної кривої на число в полі $GF(p)$ [Текст] / М.В. Онай, О.С. Князькіна // Прикладна математика та комп'ютинг. ПМК, 2014 : шоста наук. конф. магістрантів та аспірантів, Київ, 16-18 квітня 2014 р. : зб. тез доп. / [ред кол.: Дичка І.А. та ін.]. – К. : Просвіта, 2014. – С. 148-154.
17. Онай, М.В. Спосіб перетворення многочленного подання елементів поля $GF(p^m)$ у степеневе [Текст] / М.В. Онай, Ю.В. Вальчук // Шістнадцята всеукраїнська (одинадцята міжнародна) студентська наукова конференція з прикладної математики та інформатики СНКПМІ-2013: Тези доповідей, 11-12 квітня, 2013 р. – Львів : ЛНУ 2013. – С. 46-47.
18. Дичка, І.А. Організація проблемно-орієнтованого процесора для реалізації операцій в полях Галуа виду $GF(2^m)$ [Текст] / І.А. Дичка, М.В. Онай // Тези доповідей Четвертої Міжнародної науково-практичної конференції "Методи та засоби кодування, захисту й ущільнення інформації" м. Вінниця, 23-25 квітня 2013 року. – Вінниця: ПП "ТД "Едельвейс і К", 2013. – С. 270-273.
19. Дичка, І.А. Спосіб зберігання в пам'яті ЕОМ різних форм подання елементів скінченного поля характеристики 2 [Текст] / І.А. Дичка, М.В. Онай // Комп'ютерні інтелектуальні системи та мережі. Матеріали VI Всеукраїнської WEB-конференції аспірантів, студентів та молодих вчених (19-21 березня 2013 р.). – Кривий Ріг : Криворізький національний університет, 2013. – С. 56-59.

20. Дичка, І.А. Організація системи команд співпроцесора Галуа [Текст] / І.А. Дичка, М.В. Онай // Міжнародна науково-технічна конференція “Радіотехнічні поля, сигнали, апарати та системи”. Київ, 11-15 березня 2013 р.: матеріали конференції – Київ : 2013. – С. 212-213.
21. Дичка, І.А. Способи знаходження мультиплікативного оберненого елемента в скінченних полях [Текст] / І.А. Дичка, М.В. Онай // Друга наукова конференція магістрантів та аспірантів присвячена 20-річчю факультету прикладної математики «Прикладна математика та комп’ютинг ПМК-2010» : Київ, 14-16 квіт. 2010 р. : зб. тез доп. / ред кол. : Дичка І.А. [та ін.] – К. : Просвіта, 2010. – С. 313-317.

ABSTRACT

Onai M.V. “Methods and Means of Implementation Efficiency Increasing for Computational Operations in Finite Fields”. – Qualifying scientific work on the rights of the manuscript.

Thesis for a PhD degree by specialty 05.13.05 – Computer Systems and Components (Information Technologies). – National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, 2017.

Abstract Content

The thesis is devoted to the problem of increasing the efficiency of computations in finite fields. The proposed solution of this problem is based on improving existing methods and developing new methods for the following: performing addition, subtraction, exponentiation, calculating the multiplicative inverse element and division in the finite fields of the form $GF(p)$ and $GF(2^m)$; obtaining architectural solutions for the implementation of the proposed methods and modeling the processes of performing operations in finite fields.

The analysis of the current state of the development of methods of operations in finite fields is carried out and priority points are highlighted. It is best to classify them on the basis of the distinguished features. The classification of the methods of performing the most computational expensive operations in the finite fields was performed. It enables to conduct thorough research and form the directions of development for these methods.

Operations in finite fields can be implemented both in software and in hardware. In recent years, though, particular attention has been paid to tasks of the applied nature, in which the processing of information takes place in real time. This can then be provided by the hardware implementation of computations in finite fields. Examples of such tasks include public key

cryptography, digital signal processing, and error-correction coding. In the listed tasks, it is crucial to have the operation time and the secrecy of the data transfer, which can not be properly ensured during software implementation. Due to the specifics of computations in the Galois fields, its software implementation with the use of universal computers and high-level programming languages does not always provide the speed required to obtain a result.

In the thesis, it is discovered that in order to ensure the high efficiency of calculations in finite fields it is necessary to use specialized computing means. In our time, with the development of integrated circuitry (in particular, the *FPGA*), there are new possibilities for the implementation of computations in finite fields with the required speed. Note that this, can only be achieved through the hardware implementation.

The methods for calculating a multiplicative inverse element based on modular exponentiation, and methods based on calculating the greatest common divisor (*GCD*) of two integers are analyzed in detail. It has been discovered that methods that are based on the *GCD* algorithm are less computationally expensive.

The method of high-speed implementation of additive and multiplicative operations on elements of $GF(2^m)$ and corresponding hardware structures for its implementation are proposed. Additive operations include addition and subtraction. Multiplicative operations include multiplication, exponentiation, multiplicative inverse element calculation, and division.

Through research, it is found that the state of the problem of highly effective computations in finite fields of the form $GF(2^m)$ are characterized by isomorphism and the field elements allow for four representations: an integer, a power, a vector, and a polynomial. All these representations of the field elements are equivalent, and each of them is can be used to implement the corresponding operations on the elements.

The operation of adding field elements and calculating the contrary element is easy performed over a vector (polynomial) representation, which in

the case of $p = 2$ coincides with the binary codes of the field elements. It is easy to perform the multiplication operation and calculate the multiplicative inverse element, division, and exponentiation on the power representation of the field elements since in this case it is necessary to perform operations only over the exponents. The power representation of elements of the $GF(2^m)$ has a significant disadvantage, however, as it cannot receive a zero element of the field. Therefore, when performing operations in $GF(2^m)$, it is necessary to dynamically to move from one form of elements representation to another one depending on the type of the operation and vice versa, that is, operatively, at the hardware level, to ensure the isomorphism of the field.

The research has shown that table storage of elements of the $GF(2^m)$ in their polynomial and power representation ensures the maximum speed and versatility of the arithmetic logic unit. For the use of long integer operands, the sparse formation of the table of field elements is proposed. It enables reducing the memory consumption for its storage in several times. The algorithms for converting power representation into polynomial one and polynomial representation into power one with the use of a sparse table are constructed. The developed method provides a 15% increase in performance comparing with the existing method.

The method for obtaining an upper estimate of the number of elementary one-clock operations for methods of calculating a multiplicative inverse element and exponentiation in the $GF(2^m)$ is proposed. It is found out that the method with the use of a sparse table of field elements is characterized by a weak dependence of the number of shift operations and bitwise addition for the calculating the multiplicative inverse element and the division from the field parameter m. The proposed method for the operation of calculating the multiplicative inverse element gives an increase in the speed of approximately 4 times for the value $m = 20$, and for the exponentiation increase in speed almost 25 times.

Experimental researches were performed to determine the best sparse ratio of the elements table of the $GF(2^m)$. It has been discovered that for a value of $m < 21$, the best sparse ratio is equal to 8. For the exponentiation, it is desirable to increase the sparse ratio to 16.

The methods of exponentiation in finite fields are investigated, namely: methods based on the representation of the exponent in the binary numerical system and the analysis of the exponent bits both one, and several bits for one iteration of the algorithm's operation; methods based on the representation of the exponent in a non-adjacent form, and methods based on representation of the exponent in a multi-base numerical system.

The modification of the method of exponentiation elements $GF(p)$ with sliding window is proposed. The difference from the existing methods is that when forming a precomputation table, the exponents are prime numbers, and when analyzing the binary representation of the exponent, blocks of bits forming a prime number are allocated. To achieve high performance, when constructing a precomputation table, it is recommended to use pre-calculated additive chains to minimize the number of multiplication operations, which allows each of the following table items to be obtained in one or two modular multiplication operations. The modification of the exponentiation method of elements $GF(p)$ with a sliding window can be used in elliptic-curve cryptography to improve the time characteristics of the scalar multiplication on elliptic curve.

The model of the computational process of operations execution in finite fields is developed, which enables comparison of methods on the basis of given sets of input data and choose the optimal parameters values and forms of operands representation, which ensure an increase in speed for the implementation of computing operations on the *FPGA*. The research methodologies of new methods hardware implementation of calculations in Galois fields are developed on the basis of the proposed model. Simulation in the development environment of *Xilinx ISE* and using the *Mentor Graphics*

Precision software showed that the developed hardware structures are characterized by minimal hardware complexity and high performance.

With the help of the developed computational model, it has been found out that the proposed modification of the exponentiation method of elements $GF(p)$ with a sliding window provides an increase in speed by 7-9%.

A *Verilog* code generator for *FPGA* synthesis of a *ROM* element containing a sparse table of $GF(2^m)$ field elements in a polynomial and power representation is created which automatically generates a *Verilog* code for a given irreducible polynomial and a sparse ratio of the table.

The architecture and the command system of the specialized Galois processor, focused on operations in finite fields, have been developed. The proposed command system allows one to create software of arbitrary complexity in the Assembler of the Galois processor, followed by execution of these software by the functional blocks of the processor. The command system includes: arithmetic instructions (*ADD*, *MULT*, *DIV*, *POW*, *INVM*, *CDP*, *CPD*, *INVA*, *SUB*, *INC*, *DEC*), data transfer instructions (*MOV*, *LOAD*, *OUT*) and control instructions (*JMP*, *LOOP*).

The Galois processor can be used in the universal computing system as a coprocessor, complementing the command system of the central processor unit, or as a special device based on the *FPGA*, which, in comparison with the universal computing means, increases the efficiency of processing information in real time. The architecture feature of the developed processor is that the user can change the arithmetic logic unit to make the transition to the $GF(p)$ or $GF(2^m)$ without changing the processor interface. The research of the developed Galois processor has been carried out, which showed that this processor provides an increase in the productivity of computing by 27% compared with the universal computing means.

A program model of the Galois processor is constructed, which allows the user to create software in Assembler of the Galois processor.

Thus, in the PhD thesis the actual scientific-applied task is solved: to increase the productivity of systems of digital data processing, error-correction coding of storage and data transmission, and cryptographic transformations at the expense of creation of effective technical means for computing in finite fields by means of structurally-logical optimization of operations in Galois fields.

Keywords: Galois field, finite field, multiplicative inverse element, exponentiation, irreducible polynomial, Galois Assembler, Galois processor, *FPGA*.

PhD candidate's publication list

1. Zhengbing Hu The Analysis and Investigation of Multiplicative Inverse Searching Methods in the Ring of Integers Modulo M [Text] / Zhengbing Hu, I. A. Dychka, Onai Mykola, Bartkoviak Andrii // International Journal of Intelligent Systems and Applications (IJISA), 2016. – Vol. 8, No. 11. – P. 9-18 (Входить до міжнародної наукометричної бази даних SCOPUS).
2. Дичка, І.А. Модифікований віконний метод однократного множення точки еліптичної кривої на скаляр у полі *GF*(*p*) [Текст] / І.А. Дичка, М.В. Онай, Т.П. Дрозда // Радіоелектроніка, інформатика, управління. – Запоріжжя. – 2016. – №2. – С. 95-102 (Входить до міжнародної наукометричної бази даних Web of Science).
3. Дичка, І.А. Апаратна реалізація обчислень у скінченних полях характеристики два [Текст] / І.А. Дичка, М.В. Онай, Ю.В. Бухтіяров // Наукові вісті НТУУ “КПІ”. – 2013. – №6. – С. 20-27 (Входить до міжнародної наукометричної бази даних EBSCO).

4. Дичка, І.А. Апаратна реалізація процедур множення і ділення многочленів у скінченних полях [Текст] / І.А. Дичка, В.І. Голуб, М.В. Онай // Наукові вісті НТУУ “КПІ”. – 2012. – №5. – С. 61-66 (Входить до міжнародної наукометричної бази даних EBSCO).
5. Дичка, І.А. Апаратна реалізація операторів та функцій в полях Галуа [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Хмельницького національного університету. – 2012. – Вип. 5. – С. 234-240 (Входить до міжнародної наукометричної бази даних Index Copernicus).
6. Дичка, І.А. Архітектура проблемно-орієнтованого процесора для реалізації арифметики скінченних полів [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №12 (183), ч.2. – С. 99-106.
7. Дичка, І.А. Організація спеціалізованих комп’ютерних систем для реалізації обчислень у скінченних полях [Текст] / І.А. Дичка, В.І. Голуб, М.В. Онай // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №6 (177). – С. 268-278.
8. Дичка, І.А. Подання інформації у графічно-кодованому вигляді: технологія кодування та декодування [Текст] / І.А. Дичка, М.В. Онай, М.В. Новосад // Реєстрація, зберігання і обробка даних (Data Recording, Storage & Processing). – 2010. – Т. 12, №2. – С. 69-80.
9. Державний патент України №111351, МПК G06F 7/00, G06F 7/50. Схема для пошуку мультиплікативно оберненого елемента за довільним модулем [Текст] / Дичка І.А., Онай М.В., Приходько Е.В.; заявник та патентовласник Дичка І.А., Онай М.В., Приходько Е.В. – № u201604179; дата подання заявки 15.04.2016; дата публ. 10.11.2016, бюл. №21, 2016 р. – 7 с.

10. Державний патент України №73309, МПК G06F 7/50. Пристрій для виконання обчислень в полі $GF(2^n)$ [Текст] / Дичка І.А., Онай М.В. ; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201115679; дата подання заявки 30.12.2011; дата публ. 25.09.2012, бюл. №18, 2012 р. – 10 с.
11. Державний патент України №57281, МПК G06F 7/48. Суматор елементів поля $GF(p^m)$ [Текст] / Дичка І.А., Онай М.В.; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201004903; дата подання заявки 23.04.2010; дата публ. 25.02.2011, бюл. №4, 2011 р. – 17 с.
12. Державний патент України №54637, МПК G06F 7/50. Суматор за модулем простого числа [Текст] / Дичка І.А., Онай М.В.; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201001074; дата подання заявки 02.02.2010; дата публ. 25.11.2010, бюл. №22, 2010 р. – 14 с.
13. Онай, М.В. Знаходження мультиплікативно оберненого елемента у кільці лишків за довільним модулем методом Джої-Пейє [Текст] / М.В. Онай, А.Ю. Бартков’як // Міжнародна науково-практична конференція “Проблеми інформатики та комп’ютерної техніки”. Праці конференції. – Чернівці : Видавничий дім “Родовід”, 2015. – С. 91-93.
14. Онай, М.В. Пошук мультиплікативно оберненого елементу у кільці лишків за довільним модулем методами, що грунтуються на модулярному піднесенні до степеня [Текст] / М.В. Онай, А.Ю. Бартков’як // Шістнадцята міжнародна наукова конференція імені академіка Михайла Кравчука, 14-15 травня, 2015 р., Київ : Матеріали конф. Т. 2. Алгебра. Геометрія. Математичний аналіз. – К. : НТУУ “КПІ”, 2015. – С. 139-141.

15. Дичка, І.А. Застосування *k*-арного методу Евкліда для пошуку мультплікативно оберненого елемента у кільці лишків за модулем *m* [Текст] / І.А. Дичка, М.В. Онай, А.Ю. Бартков'як // Матеріали статей П'ятої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Івано-Франківськ : п. Голіней О.М., 2015. – С. 151-153.
16. Онай, М.В. Спосіб прискорення алгоритмів множення точки еліптичної кривої на число в полі $GF(p)$ [Текст] / М.В. Онай, О.С. Князькіна // Прикладна математика та комп'ютинг. ПМК, 2014 : шоста наук. конф. магістрантів та аспірантів, Київ, 16-18 квітня 2014 р. : зб. тез доп. / [ред кол.: Дичка І.А. та ін.]. – К. : Просвіта, 2014. – С. 148-154.
17. Онай, М.В. Спосіб перетворення многочленного подання елементів поля $GF(p^m)$ у степеневе [Текст] / М.В. Онай, Ю.В. Вальчук // Шістнадцята всеукраїнська (одинадцята міжнародна) студентська наукова конференція з прикладної математики та інформатики СНКПМІ-2013: Тези доповідей, 11-12 квітня, 2013 р. – Львів : ЛНУ 2013. – С. 46-47.
18. Дичка, І.А. Організація проблемно-орієнтованого процесора для реалізації операцій в полях Галуа виду $GF(2^m)$ [Текст] / І.А. Дичка, М.В. Онай // Тези доповідей Четвертої Міжнародної науково-практичної конференції "Методи та засоби кодування, захисту й ущільнення інформації" м. Вінниця, 23-25 квітня 2013 року. – Вінниця: ПП "ТД "Едельвейс і К", 2013. – С. 270-273.
19. Дичка, І.А. Спосіб зберігання в пам'яті ЕОМ різних форм подання елементів скінченного поля характеристики 2 [Текст] / І.А. Дичка, М.В. Онай // Комп'ютерні інтелектуальні системи та мережі. Матеріали VI Всеукраїнської WEB-конференції аспірантів, студентів та молодих вчених (19-21 березня 2013 р.). – Кривий Ріг : Криворізький національний університет, 2013. – С. 56-59.

20. Дичка, І.А. Організація системи команд співпроцесора Галуа [Текст] / І.А. Дичка, М.В. Онай // Міжнародна науково-технічна конференція “Радіотехнічні поля, сигнали, апарати та системи”. Київ, 11-15 березня 2013 р.: матеріали конференції – Київ : 2013. – С. 212-213.
21. Дичка, І.А. Способи знаходження мультиплікативного оберненого елемента в скінченних полях [Текст] / І.А. Дичка, М.В. Онай // Друга наукова конференція магістрантів та аспірантів присвячена 20-річчю факультету прикладної математики «Прикладна математика та комп’ютинг ПМК-2010» : Київ, 14-16 квіт. 2010 р. : зб. тез доп. / ред кол. : Дичка І.А. [та ін.] – К. : Просвіта, 2010. – С. 313-317.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

КА – керуючий автомат;

КП – контролер переривань;

НСД – найбільший спільний дільник;

ОА – операційний автомат;

ПД – пам’ять даних;

ПЗП – постійний запам’ятовувальний пристрій;

ПК-д – пам’ять команд;

СхМн – схема двійкового множення;

СЦД – схема цілочислового ділення;

ША – шина адрес;

ШД – шина даних;

DBNS – *DoubleBase Number System*, система числення з подвійною основою;

GF(*p*) – *Galois Field*, поле Галуа або скінченне поле, де *p* – просте число, що є кількістю елементів поля;

$GF(p^m)$ – *Extended Galois Field*, розширення поля Галуа над *GF*(*p*), де *p* – просте число, що є характеристикою поля, а p^m – кількість елементів поля;

LR – *Left-to-Right*, аналіз показника степеня зліва направо;

MBNS – *MultyBase Number System*, система числення з мультиосновою;

NAF – *Non-Adjacent Form* (несуміжна форма), подання у симетричній трійковій системі числення;

RAM – *Random Access Memory* (пам'ять з довільним доступом);

RL – *Right-to-Left*, аналіз показника степеня справа наліво;

ROM – *Read Only Memory* – енергонезалежна пам'ять, з якої може проводитись тільки зчитування даних;

wNAF – *window Non-Adjacent Form* (віконна несуміжна форма).

ВСТУП

При реалізації обчислювальних операцій в комп'ютерних системах важливе значення мають поля, що утворені скінченною множиною елементів. Особливо значний внесок у розвиток теорії скінченних полів належить Еварісту Галуа. Тому скінченні поля часто називають полями Галуа. Основи систематизованої теорії були сформовані в дослідженнях видатних математиків кінця XIX століття – Муром та Діксоном. Зокрема американський математик Е. Мур у 1893 році вперше довів, що будь-яке скінченне поле є полем Галуа, а Л. Діксон виконав перший систематичний виклад теорії скінченних полів [74, 93].

До останньої чверті 20-го століття теорія скінченних полів розвивалась як галузь класичної математики [12, 15]. Але у зв'язку з потребами цифрової обробки сигналів, завадостійкого кодування та бурхливим розвитком криптографії в наш час активно розвиваються прикладні аспекти теорії скінченних полів. Систематизація та узагальнення теоретичних основ арифметики скінченних полів кінця 20-го століття належить вченим Лідлу та Нідеррайтеру [114].

Сучасний виклад теорії скінченних полів з орієнтацію на прикладні сфери застосування з'явився у працях Ніла Кобліца, Брюса Шнайєра, Дарела Хенкерсона, Альфреда Менезеса, Жан-П'єр Дешама, Річарда Крендалла, Карла Померанса, Венбо Мао, Четін Кая Коч, Гордона Бредлі, А.О. Болотова, С.Б. Гашкова, О.Б. Фролова, А.О. Часовських, О.Б. Маховенко, О.Н. Василенка, О.В. Черемушкіна, В.П. Боюна, В.М. Опанасенка, В.П. Тарасенка, Я.М. Николайчука, І.А. Жукова, В.В. Яцківа та інших [2, 9, 10, 11, 13, 18, 38, 40, 45, 49, 64, 66, 69, 83, 90, 91, 92, 103, 111, 135, 142].

Скінченні поля використовуються для побудови більшості відомих завадостійких кодів, цифрових фільтрів, генераторів псевдовипадкових чисел та криптографічних алгоритмів. Зокрема операції над елементами полів Галуа виконуються у *CRC*-кодах, кодах *BCH* (Боуза-Чоудхурі-

Хоквінгема) та кодах Ріда-Соломона [85, 106], які знайшли застосування у турбо-кодах, що використовуються у телевізійних приймачах з супутниковими антенами; цифрових фільтрах як зі скінченною імпульсною характеристикою, так і з безкінечною, вейвлетних фільтрах третього порядку над простим полем Галуа та вейвлет-перетвореннях в базисі Хаара над полем Галуа; у лінійному конгруентному генераторі псевдовипадкових чисел та його модифікаціях [110]; у найбільш поширених симетричних криптографічних алгоритмах, таких як *AES*, *IDEA*, *RC*5 [67, 89, 118, 135], у асиметричних криптографічних шифрах, таких як *RSA*, *DSA*, протоколі Діффі-Хелмана [94], схемі Ель-Гамаля, потокових шифрах та алгоритмах еліптичної криптографії.

Актуальність роботи. На даний момент з теорії скінченних полів та її застосування в комп'ютерних системах опубліковано низку наукових статей, але здебільшого висвітлюються математичні аспекти виконання операцій у скінченних алгебраїчних структурах та не приділяється достатньої уваги архітектурі обчислювальних засобів для виконання операцій в скінченних полях [39, 41, 74, 78, 109, 113, 114, 119, 127, 137].

Операції в скінченних полях можна реалізовувати як програмно, так і апаратно, проте в останні роки особлива увага приділяється задачам прикладного характеру, де обробка інформації відбувається в реальному часі, що можливо забезпечити за рахунок апаратної реалізації обчислень у скінченних полях. Прикладом таких задач є задачі криптографії з відкритим ключем, цифрова обробка сигналів та завадостійке кодування даних [6-8, 36, 37, 42, 43, 54, 56, 60, 69, 71, 76, 109, 127, 129, 131]. У перелічених задачах дуже важливе значення має час виконання операції та секретність передачі даних, що належним чином не можна забезпечити при програмній реалізації. Через специфіку обчислень в полях Галуа їх програмна реалізація з використанням універсальних комп'ютерів та мов програмування високого рівня не завжди забезпечує потрібну швидкість отримання результату [109, 113].

Для забезпечення високої ефективності обчислень у скінченних полях необхідно застосовувати спеціалізовані обчислювальні засоби. У наш час з розвитком інтегральної схемотехніки, зокрема ПЛІС, з'являються нові можливості для реалізації обчислень у скінченних полях з потрібною швидкодією, досягти якої можливо лише за рахунок апаратної реалізації операцій [112, 128, 141].

Тому, останнім часом дедалі більша увага приділяється питанням апаратної реалізації арифметики скінченних полів, оскільки при цьому істотно зростає швидкість обчислень, що має важливе значення для задач криптографії з відкритим ключем, завадостійкого кодування даних та цифрової обробки сигналів [33, 38, 44, 58, 62, 77, 79-81, 113, 121, 134, 139].

У працях [78, 109, 113, 120, 127] розглядаються лише питання апаратної реалізації однієї з операцій в скінченних полях або реалізація операцій в скінченних полях, що орієнтована тільки на криптографічні додатки, в той час як існує необхідність побудови універсальних апаратних засобів для реалізації операцій в полях Галуа, що будуть орієнтовані на задачі криптографії, завадостійкого кодування та цифрової обробки сигналів.

Відомо, що спеціалізація та проблемна орієнтація на клас розв'язуваних задач є одним з основних шляхів істотного поліпшення технічних характеристик обчислювальних засобів [3, 5, 11, 70, 72].

Під спеціалізацією та проблемною орієнтацією як правило розуміють орієнтацію структури процесора і систем команд та принципів організації обчислень на розв'язування певного класу задач.

Спеціалізація та проблемна орієнтація дозволяє забезпечити високу продуктивність та оперативність обробки інформації в реальному часі за значно менших порівняно з універсальними засобами апаратурних витратах і вартості або за однакових апаратурних витрат значно підвищити продуктивність і точність обробки інформації. І в першому, і в другому випадках істотно підвищується ефективність обробки інформації, тобто відношення продуктивності обчислювальних засобів до їх вартості [84].

Урахування специфіки задач та виду оброблюваної інформації дозволяє орієнтувати або спеціалізувати технічні засоби і математичне забезпечення на них і за рахунок цього підвищити ефективність систем [75].

Отже, аналізуючи клас задач, який має місце в галузі криптографічного захисту інформації [50], при завадостійкому кодуванні та цифровій обробці сигналів можливі два підходи до створення обчислювальних засобів для реалізації обчислень в полях Галуа: 1) побудова спеціалізованих комп'ютерних систем (СКС) на основі ПЛІС [30, 112]; 2) побудова проблемно-орієнтованих процесорів [25].

Таким чином, існує важлива науково-технічна задача створення методів та архітектур апаратних засобів для реалізації високопродуктивних обчислень на основі арифметики скінченних полів.

Зв'язок роботи з науковими програмами, планами, темами. Розроблення основних положень дисертаційного дослідження здійснювалась відповідно до планів НДР, програм і договорів, що виконувались в КПІ ім. Ігоря Сікорського:

- НДР "Розроблення та дослідження високоефективних архітектур спеціалізованих комп'ютерних систем для реалізації обчислень у скінченних полях" (номер державної реєстрації 0115U000319);
- НДР "Методи та засоби інформаційного забезпечення систем автоматизованого імпорту об'єктів на основі графічного кодування даних" (номер державної реєстрації 0112U003175).

Мета і задачі дослідження. Метою дисертаційної роботи є підвищення швидкості обчислень в скінченних полях за рахунок структурно-логічної оптимізації архітектур апаратних засобів, що реалізують процеси виконання операцій у скінченних полях.

Досягення зазначеної мети передбачає вирішення таких задач:

- розробити методи високошвидкісного виконання операцій в полях Галуа виду $GF(p)$: додавання, віднімання, множення, ділення, піднесення до степеня, обчислення мультиплікативно оберненого елемента;

- синтезувати структури апаратних засобів для реалізації операцій арифметики скінченних полів, які б забезпечували більш високу швидкість обчислень порівняно з існуючими рішеннями;
- розробити методи апаратної реалізації операцій в полях Галуа виду $GF(2^m)$;
- розробити архітектуру та систему команд проблемно-орієнтованого процесора Галуа для виконання операцій у скінченних полях;
- розробити модель обчислювального процесу, що має місце при реалізації операцій у скінченних полях;
- створити засоби для моделювання та дослідження ефективності методів виконання операцій в полях Галуа.

Об'єктом дослідження є процеси обробки даних у скінченних полях.

Предметом дослідження є методи обчислень та засоби апаратної реалізації операцій у скінченних полях.

Методи дослідження базуються на використанні теорії чисел, вищої алгебри, дискретної математики, теорії обчислювальної складності алгоритмів, теорії скінченних алгебраїчних структур, комп'ютерного моделювання та схемотехнічного проектування.

Наукова новизна одержаних результатів полягає в розвитку методів та удосконаленні архітектур апаратних засобів для виконання операцій у скінченних полях: додавання, віднімання, множення, піднесення до степеня, обчислення мультиплікативно оберненого елемента та ділення.

1. Вперше запропоновано метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$, характерною особливістю якого, на відміну від існуючих, є застосування табличного зберігання елементів поля у многочленному та степеневому їх поданні. Даний метод передбачає можливість розрідженого формування таблиці елементів поля, що зменшує витрати пам'яті для її зберігання.

Метод забезпечує зростання швидкодії на 15% порівняно з існуючим рішенням.

2. Запропоновано модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном, яка полягає в тому, що при формуванні таблиці передобчислень використовуються показники степеня, що є простими числами. Такі показники степеня дозволяють отримувати кожен наступний елемент таблиці передобчислень за одну-дві операції модулярного множення та забезпечують зменшення кількості операцій множення – наслідком чого є приріст швидкодії на 7-9 %.
3. Розроблено модель обчислювального процесу, що має місце при реалізації операцій у скінченних полях, яка дозволяє виконувати порівняння методів за заданими показниками та здійснювати вибір параметрів і форм подання операндів, що забезпечують максимальну швидкодію при реалізації обчислювальних операцій.

Практичне значення одержаних результатів полягає в розробці засобів підвищення швидкості виконання обчислювальних операцій арифметики скінченних полів. Зокрема, практичну цінність мають наступні результати.

1. Спроектовано на ПЛІС фірми *Xilinx* процесор Галуа, що орієнтований на виконання операцій у скінченних полях виду $GF(p)$ та $GF(2^m)$. Процесор Галуа можна застосувати для вирішення задач завадостійкого кодування даних, цифрової обробки сигналів та захисту інформації.
2. Побудовано програмістську модель процесора Галуа, яка дозволяє створювати програмне забезпечення довільної складності мовою Асемблера процесора Галуа.
3. Розроблено структурні та схемотехнічні рішення блоків виконання обчислювальних операцій у скінченних полях виду $GF(p)$ та $GF(2^m)$, які характеризуються низькою апаратною складністю та високою швидкістю обробки даних.

4. Розроблено програмний модуль для імітаційного та функціонального моделювання обчислювального процесу в процесорі Галуа, що дозволяє проводити дослідження розроблених методів виконання операцій у скінченних полях.
5. Створено генератор *Verilog*-коду для синтезу на ПЛІС елемента *ROM*, що містить розріджену таблицю елементів поля $GF(2^m)$ у многочленному та степеневому їх поданні, який автоматично генерує *Verilog*-код за заданим незвідним многочленом та ступенем розрідження таблиці.
6. Сформовано методичні рекомендації щодо використання новорозроблених методів виконання операцій у скінченних полях для їх застосування в галузі обробки сигналів, криптографічних перетворень та завадостійкого кодування.
7. Розроблено програмний інструментарій для моделювання та дослідження обчислювальних процесів у полях Галуа, що дозволяє обирати оптимальні параметри для кожного методу залежно від класу вхідних даних.

Теоретичні та практичні результати дисертаційної роботи використано і впроваджено:

- на Українському державному підприємстві поштового зв'язку "Укрпошта" при розробці технології цифрових поштових марок;
- на підприємстві ТОВ «Відео Інтернет Технології» при розробці системи обробки та аналізу зображень для розпізнавання реєстраційних номерів транспортних засобів;
- у навчальному процесі кафедри програмного забезпечення комп'ютерних систем КПІ ім. Ігоря Сікорського при проведенні лекційних та лабораторних занять з дисциплін "Теорія інформації та кодування", "Архітектура комп'ютера" та "Цифрова обробка сигналів та зображень", а також при

підготовці магістерських робіт зі спеціальності “Інженерія програмного забезпечення”.

Особистий внесок здобувача полягає в теоретичному обґрунтуванні одержаних результатів, їх експериментальній перевірці та дослідженні. Основні результати дисертаційного дослідження автором отримано самостійно. У публікаціях, написаних у співавторстві, здобувачеві належить: [26, 35, 53, 57, 143] – класифікація методів обчислення мультиплікативно оберненого елемента, удосконалення модифікації Бредлі розширеного алгоритму Евкліда та методика проведення експериментального дослідження; [27] – модифікований метод піднесення до степеня в адитивній групі; [22, 34, 61] – метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та спосіб перетворення числового подання елементів поля $GF(2^m)$ у степеневе і навпаки, що орієнтований на апаратну реалізацію на ПЛІС; [24] – спосіб ділення многочленів з остачею у скінченному полі; [23, 33] – визначення місця теорії скінченних полів при графічному кодуванні даних та архітектура апаратних засобів для реалізації обчислень в полях Галуа; [25] – система команд спеціалізованого процесора, що орієнтований на арифметику скінченних полів; [30] – схеми функціональних блоків спеціалізованої комп’ютерної системи для високошвидкісного виконання операцій у скінченних полях; [17] – архітектура апаратних засобів для реалізації операції обчислення мультиплікативно оберненого елемента в основному скінченному полі; [21, 28, 29] – архітектура пристрою для виконання обчислень у полях виду $GF(2^m)$, орієнтована на ПЛІС; [20] – спосіб виконання підсумовування елементів поля $GF(p^m)$; [19] – функціональна схема суматора за модулем простого числа; [62] – модифікація розширеного алгоритму Лемера обчислення мультиплікативно оберненого елемента в полі $GF(p)$.

Апробація результатів дисертації. Основні результати дисертації доповідались, обговорювались та отримали позитивну оцінку на наступних конференціях: Міжнародна науково-практична конференція “Інформаційна

безпека та комп'ютерні технології”, м. Кіровоград, 2016; Міжнародна науково-практична конференція “Методи та засоби кодування, захисту й ущільнення інформації”, м. Вінниця, 2013, 2016; Наукова конференція магістрантів та аспірантів Прикладна математика та комп’ютинг “ПМК”, м. Київ, 2009, 2010, 2014, 2016; Міжнародна науково-практична конференція “Проблеми інформатики та комп’ютерної техніки”, м.Чернівці, 2013, 2014, 2015; Міжнародна наукова конференція імені академіка Михайла Кравчука, м. Київ, 2012, 2014, 2015; Міжнародна науково-практична конференція “Інформаційні технології та комп’ютерна інженерія”, м. Івано-Франківськ, 2014, 2015; Всеукраїнська науково-практична конференція молодих учених та студентів, м. Хмельницький, 2015; Міжнародна студентська наукова конференція з прикладної математики та інформатики СНКПМІ, м. Львів, 2012, 2013; Всеукраїнська WEB-конференції аспірантів, студентів та молодих вчених, м. Кривий Ріг, 2013; Міжнародна науково-технічна конференція “Радіотехнічні поля, сигнали, апарати та системи”, м. Київ, 2013; Всеукраїнська науково-практична конференція “Інформатика та системні науки (ІСН-2013)”, м. Полтава, 2013.

Публікації. За матеріалами дисертації опубліковано 21 друковану працю, серед яких 8 статей (з них одна стаття у закордонному науковому виданні, що входить до наукометричної бази даних Scopus, одна стаття у фаховому науковому виданні України, що входить до наукометричної бази даних Web of Science, одна стаття у фаховому науковому виданні України, що входить до наукометричної бази даних Index Copernicus, дві статті у фаховому науковому виданні України, що входить до наукометричної бази даних EBSCO), 4 державні патенти України на корисну модель, а також 9 тез доповідей на наукових конференціях.

РОЗДІЛ 1
АНАЛІЗ ТА КЛАСИФІКАЦІЯ АЛГОРИТМІВ ВИКОНАННЯ ОПЕРАЦІЙ У СКІНЧЕННИХ ПОЛЯХ

1.1. Редукція значень величин у скінченних полях

Операція зведення цілого числа за модулем є надзвичайно важливою в теорії скінченних полів, оскільки використовується в алгоритмах множення та піднесення до степеня елементів скінченного поля. Якщо доповнити алгоритм виконання редукції таким чином, щоб він давав також значення частки від ділення, то такий алгоритм є необхідним при обчисленні мультиплікативно оберненого елемента в скінченному полі.

Отримати значення $z = x \bmod m$ можливо шляхом виконання цілочисельного ділення значення x на m, де x та m натуральні числа.

Одним з найперших алгоритмів виконання ділення з остачею є алгоритм поданий у [16]. Згідно цього алгоритму необхідно виконувати ділення з остачею шляхом послідовного віднімання дільника від діленого поки отримана часткова остача є більшою за ділене. Таким чином, частка буде дорівнювати кількості виконаних віднімань, а остача – останній частковій остачі.

Існують два широко розповсюджених алгоритми цілочисельного ділення, а саме: з відновленням остачі та без відновлення остачі [40, 73, 105, 125]. Показано, що більш ефективним є алгоритм без відновлення остачі [40, 73, 105, 125].

Зрозуміло, що для натурального y та $s \in [-2y; 2y)$ рівняння

$$s = qy + r \tag{1.1}$$

має принаймні один розв'язок для $q \in \{-1; 0; 1\}$ та $r \in [-y; y)$.

У [105] визначається функція $quotient(s; y)$, що дає один з розв'язків рівняння (1.1). На основі даної функції побудовано алгоритм А.1 редукції (додаток А), де $\tilde{n}$ – кількість двійкових розрядів діленого, $\tilde{k}$ – кількість двійкових розрядів модуля.

Другий крок алгоритму А.1 редукції потрібен для вирівнювання розрядності модуля, тобто зсув модуля вліво на таку кількість двійкових розрядів, щоб модифікований модуль *y* займав таку саму кількість розрядів як і ділене *x*, а саме, $\tilde{n}$ розрядів.

У відповідності до діаграми Робертсона [91] легко бачити, що можливо обмежитись лише двома значення q, а саме $q=-1$, якщо $s\in[-2y;0)$ та $q=1$, $s\in[0;2y)$. Цьому випадку відповідає класичний алгоритм цілочисельного ділення без відновлення остачі.

Аналізуючи алгоритм А.1 редукції можна дійти висновку, що при апаратній реалізації розрядність n регістра діленого має задовольняти нерівність $n\geq\tilde{n}$, а розрядність k регістра модуля – нерівність $k\geq\tilde{k}$. Для реалізації алгоритма А.1 редукції необхідно додати обчислення фактичної розрядності діленого та модуля. Це можна зробити за допомогою зсувів. Але аналіз даного алгоритму показує, що в тому випадку, коли на другому кроці буде виконано зсув модуля на кількість розрядів, що перевищує значення $\tilde{n}-\tilde{k}$, на результат це не вплине (якщо на меншу кількість розрядів ніж $\tilde{n}-\tilde{k}$, то результат буде спотворено). Цей факт дає можливість виконувати зсув модуля вліво на $n-k$ розрядів за умови, що $n-k\geq\tilde{n}-\tilde{k}$. Оскільки значення $n-k$ є фіксованим при апаратній реалізації, то зсув на таку кількість розрядів можна виконувати шляхом відповідного з'єднання шин, що дозволить прискорити процес обчислень. Залишається тільки забезпечити виконання умови $n-k\geq\tilde{n}-\tilde{k}$. Для модуля $m\geq 2$ потрібно $\tilde{k}\geq 2$ розряди. Отже, величина $\tilde{n}-\tilde{k}$ досягає свого максимального значення при $\tilde{k}=2$. Якщо розрядність регістра діленого дорівнює n, то максимально можливе значення розрядності діленого $\tilde{n}=n$, тому для

фіксованої розрядності (n та k) діленого та модуля відповідно максимально можливе значення величини $\tilde{n}-\tilde{k}=n-2$. Таким чином, вимога $n-k\geq\tilde{n}-\tilde{k}$ перетворюється на вимогу $n-k\geq n-2$, і це означає, що максимальна кількість розрядів, на яку може знадобитись виконувати зсув, дорівнює $n-2$. Тоді розрядність регістра для збереження проміжного результату (часткової остачі) має дорівнювати $n+k-1$ з врахуванням одного знакового розряду. У [91] пропонується взяти $k=n$ виходячи з того, що максимально можлива розрядність модуля дорівнює розрядності діленого, і тоді розрядність регістра для збереження проміжного результату буде дорівнювати $2n-1$, а зсув на другому та четвертому кроці алгоритма А.1 редукції завжди буде виконуватись на $n-2$ розряди. Така модифікація алгоритма А.1 редукції подана як алгоритм А.2 редукції.

Схемотехнічно алгоритм А.2 редукції реалізується як показано на рис. 1.1 та 1.2.

Результатом роботи розглянутого алгоритму є отримання остачі від ділення двох цілих чисел, але при реалізації алгоритмів виконання операцій у скінченних полях виникає необхідність виконувати цілочисельне ділення з отриманням частки [91].

На кожній ітерації алгоритму А.2 редукції отримують один біт частки поданої у двійковій системі числення зі знаком, де допустимими значеннями розрядів є $\{-1;1\}$, а саме 1, коли $s\geq 0$, та мінус 1 – в іншому випадку. Число, подане в такій системі числення, пов’язане зі звичайною двійковою системою числення співвідношенням $q_i=(q_i'+1)/2$, де q_i – розряд частки, поданий у звичайній двійковій системі числення, q_i' –розряд частки, поданий у двійковій системі числення зі знаком. За допомогою алгоритму А.2 редукції можна отримати $n-1$ розряд частки, що подана у двійковій системі числення зі знаком. При перетворенні до звичайної двійкової системи числення за алгоритмом А.3 редукції буде отримано n розрядів.

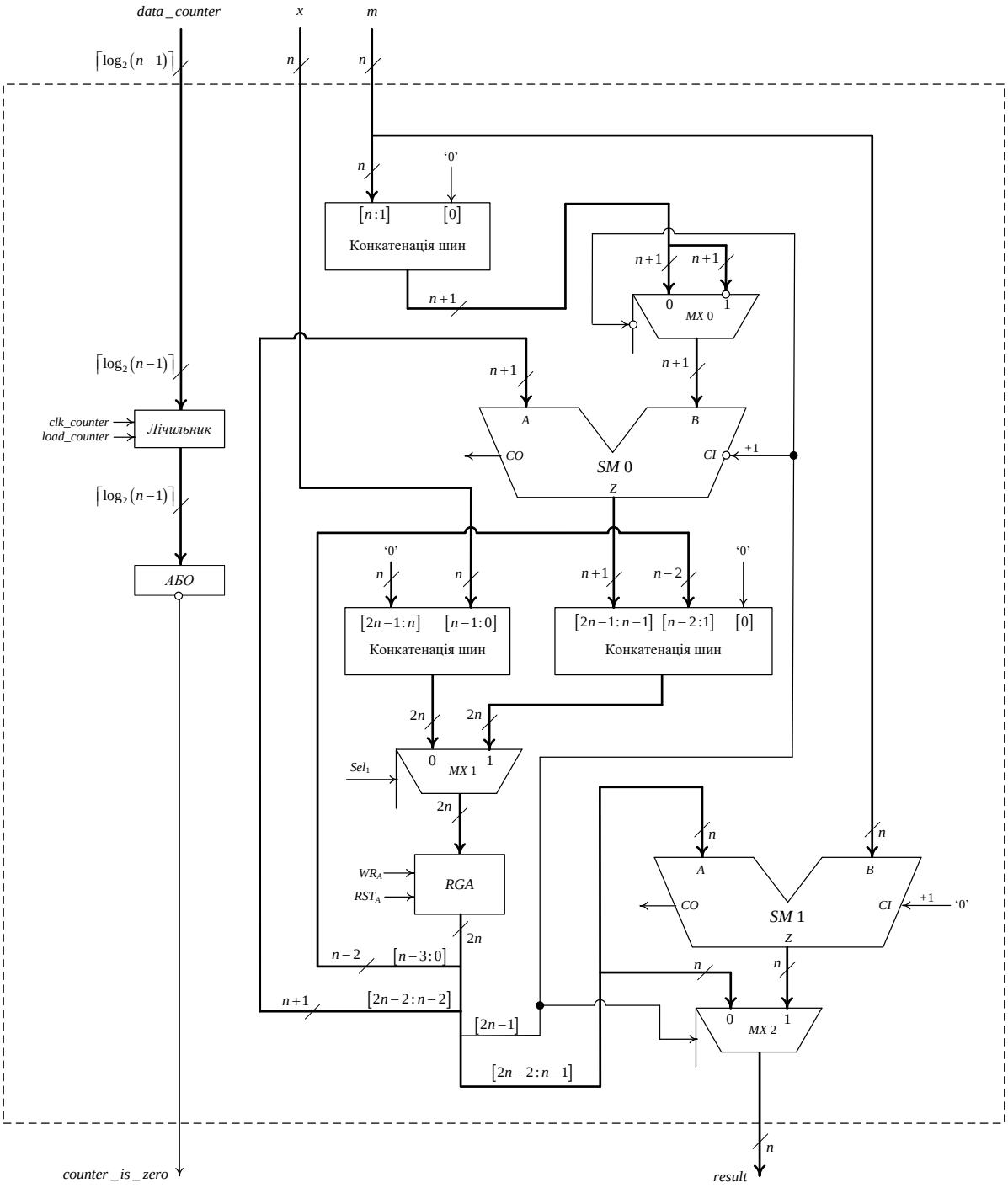


Рис. 1.1. Операційний автомат схеми, що реалізує алгоритм А.2 редукції

Використовуючи алгоритм А.3 редукції можна модифікувати алгоритм А.2 редукції [105] таким чином, щоб на виході отримувати остачу від ділення на m та частку у звичайній двійковій системі числення (алгоритм А.4 редукції).

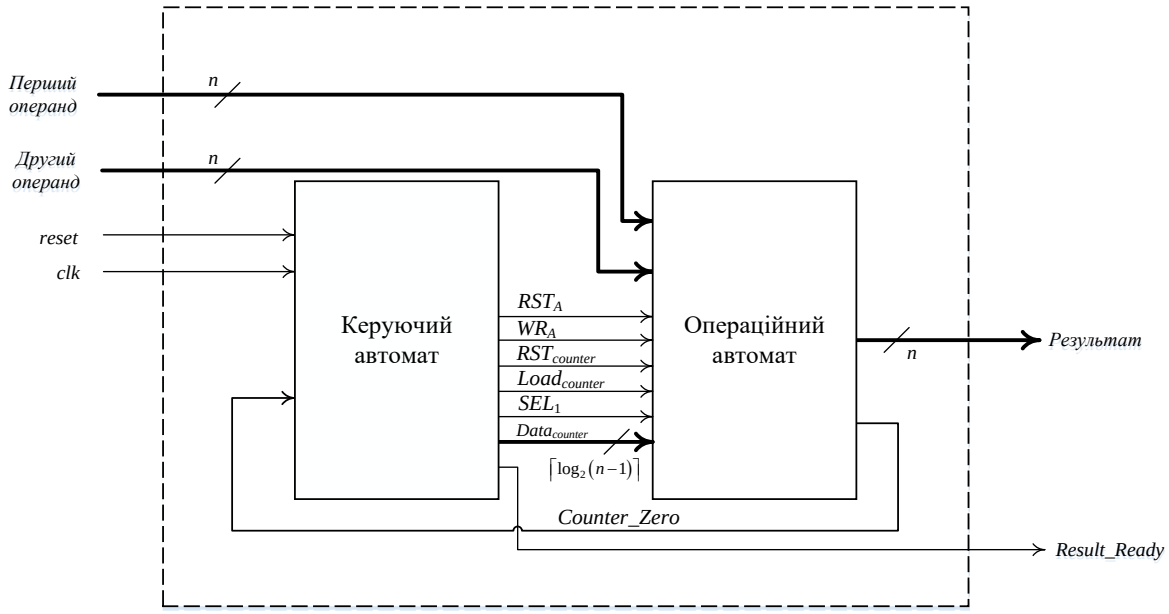


Рис. 1.2. З’єднання операційного та керуючого автомата схеми, що реалізує алгоритм А.2 редукції

Схемотехнічно операційний автомат, що реалізує алгоритм А.4 редукції, виглядає як показано на рис. 1.3.

У [91] з посиланням на [92, 97, 125] та в [87, 132, 138] зазначається, що можна досягти скорочення часу роботи схеми цілочисельного ділення, якщо для проведення ітераційного процесу використовувати суматор із збереженням переносу. Скорочення часу роботи схеми досягається за рахунок скорочення часу кожного підсумовування під час роботи схеми, оскільки підсумовування кожного біта виконується паралельно. Тоді, наприклад, *n*-розрядний результат підсумовування s буде поданий за допомогою двох *n*-розрядних чисел s_s (часткова сума, тобто сума без врахування переносу) та s_c (значення переносу для кожного розряду), а для отримання значення s необхідно буде додати s_s та s_c. Такий алгоритм цілочисельного ділення дістав назву *SRT*-алгоритм – на честь розробників цього алгоритму (Sweeney, Robertson, Tocher).

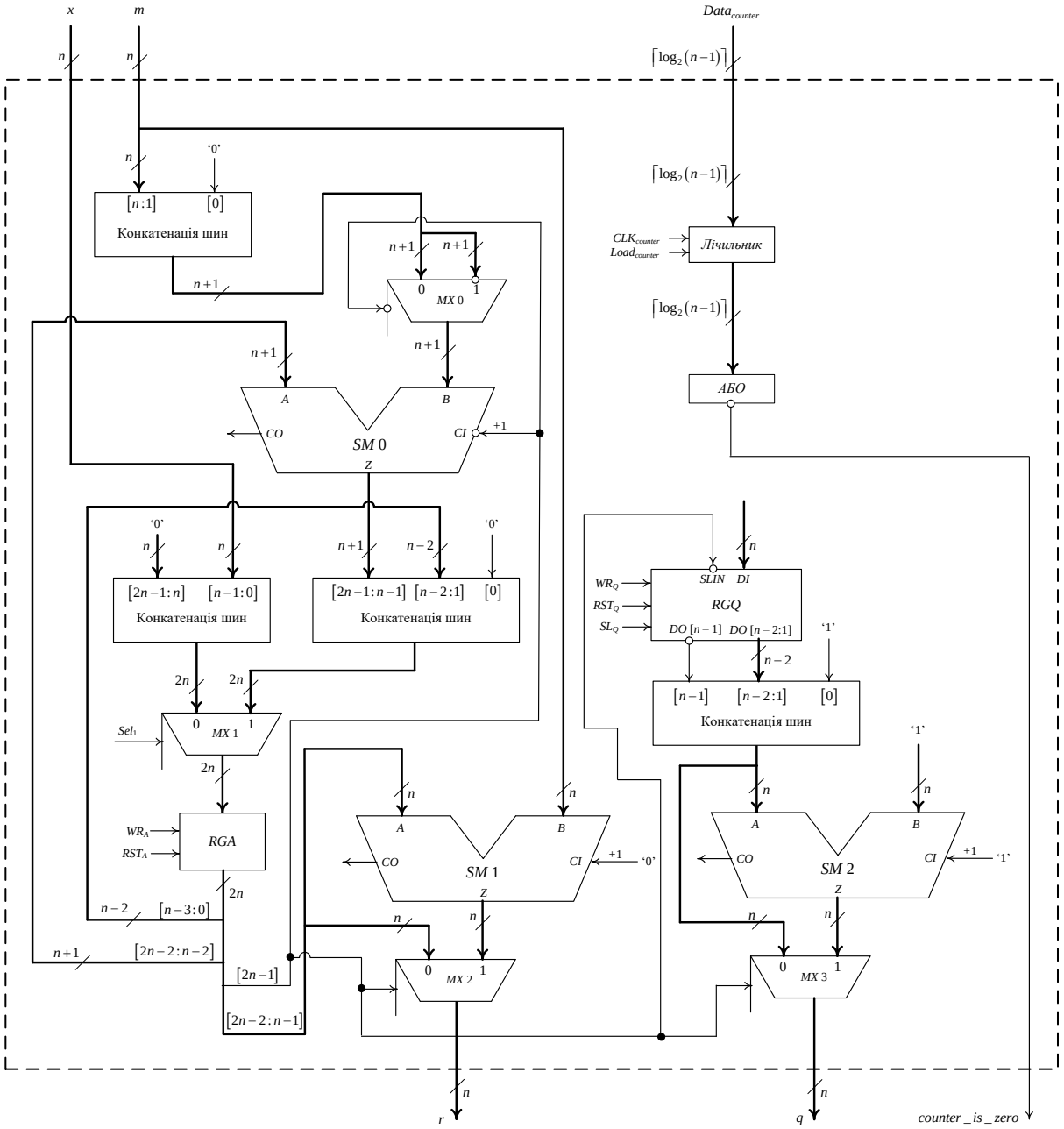


Рис. 1.3. Операційний автомат схеми, що реалізує алгоритм А.4 редукції

Побудуємо *SRT*-алгоритм на основі алгоритму А.1 редукції. На кожній ітерації алгоритму А.1 виконується одна з трьох операцій $r \leftarrow w + y$, $r \leftarrow w$ або $r \leftarrow w - y$. Узагальнюючи можна записати, що виконується дія $r \leftarrow w + h$, де h приймає значення y, 0 або $-y$. Тоді значення суми та переносу для i-го розряду буде обчислено наступним чином:

$$r_s(i) = \left(s_s(i) + s_c(i) + h(i)\right) \bmod 2, \tag{1.2}$$

$$r_c(i) = \left(s_c(i-1) + s_c(i-1) + h(i-1)\right) \operatorname{div} 2, \tag{1.3}$$

причому $r_c(i)$ – це перенос з $(i-1)$-го у *i*-й розряд.

Єдиною перепоною для апаратної реалізації алгоритму А.1 з використанням суматора зі збереженням переносу є необхідність визначати значення функції *quotient* за значенням часткової суми та переносу. У [91] з посиланням на [92] зазначається, що для визначення значення функції *quotient* достатньо лише чотирьох старших бітів w_s та w_c, а саме, значення функції *quotient* необхідно обчислювати таким чином:

$$t = w_s' + w_c' \bmod 16, \tag{1.4}$$

$$q_1 = t_3 \oplus t_2 t_1 t_0, \quad q_0 = \overline{t_2 t_1 t_0}, \tag{1.5}$$

де w_s' та w_c' – старші чотири біта величин w_s та w_c відповідно, а q_1 є знаковим розрядом величини q.

Апаратна реалізація *SRT*-алгоритму наведена на рис. 1.4.

Даний алгоритм було розроблено для операцій з плаваючою крапкою, тому він ґрунтується на тому, що мантиса операндів є нормалізованою. При застосуванні цього алгоритму для цілих чисел обов’язковою є вимога, щоб старший розряд дільника (модуля) дорівнював одиниці. Отже, розрядність регістра модуля має дорівнювати фактичній розрядності модуля, тобто обов’язковим є виконання умови $k = \tilde{k}$, що є недоліком даного алгоритму.

При виконанні ітераційного процесу використовується суматор *SM*0, тому саме цей суматор було реалізовано як суматор зі збереженням переносу.

Розглянемо докладніше схему обчислення *quotient*. Згідно формул (1.4) та (1.5) для реалізації даної схеми необхідний чотирирозрядний суматор та логічні елементи (рис. 1.5).

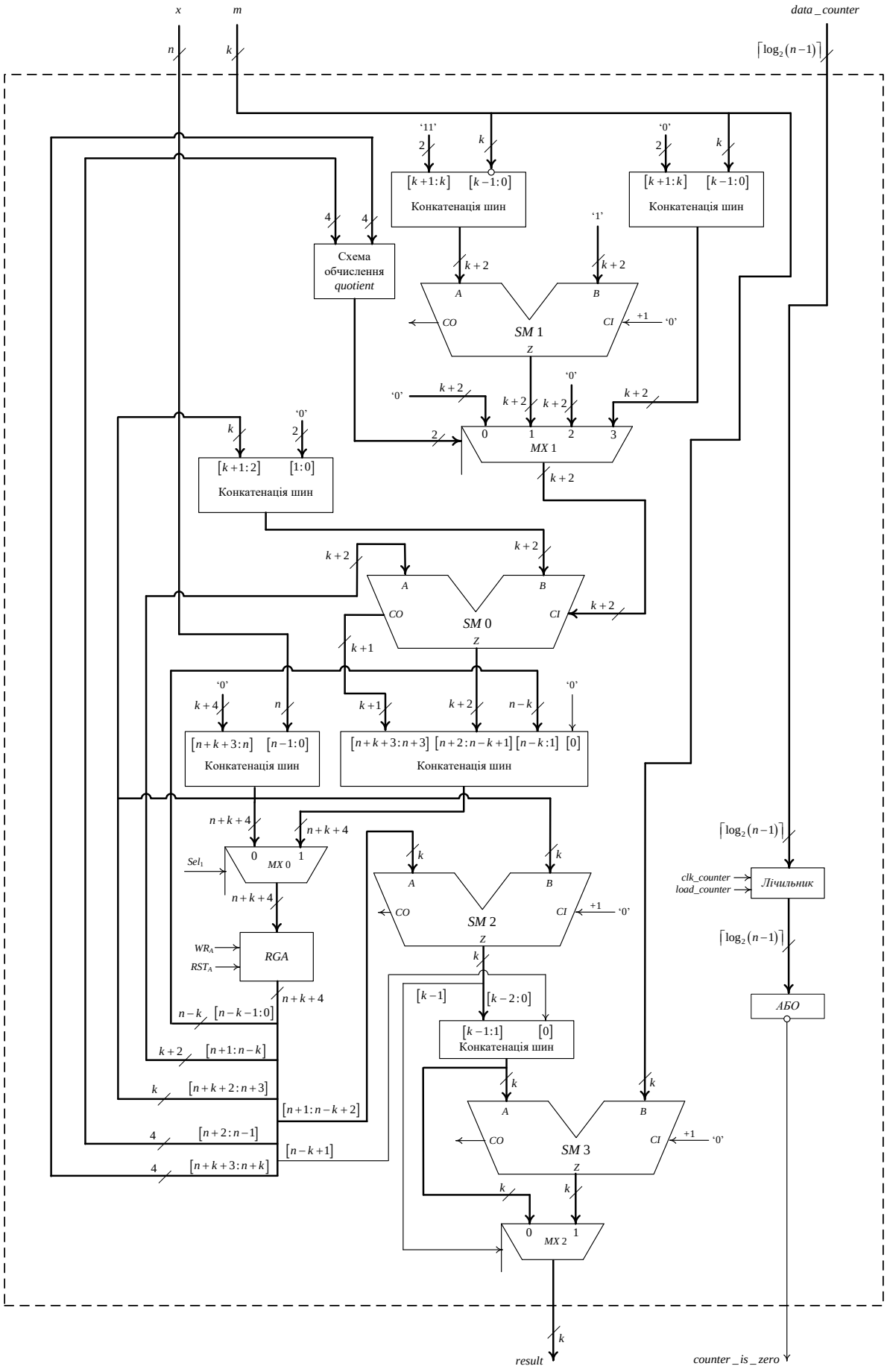


Рис. 1.4. Апаратна реалізація *SRT*-алгоритму

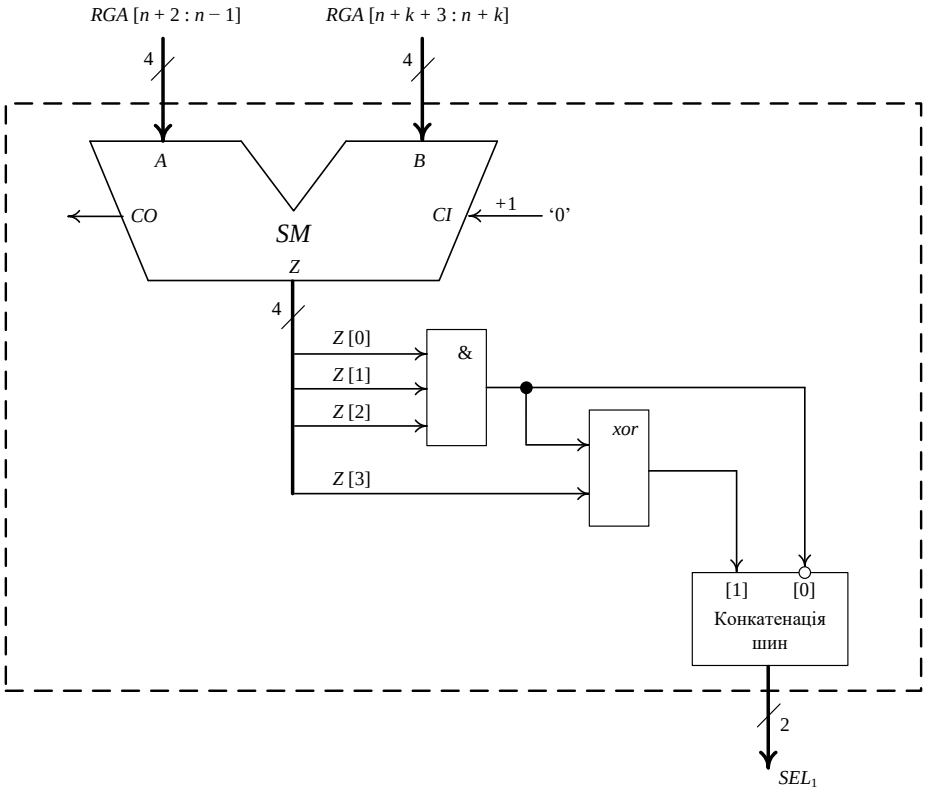


Рис. 1.5. Схема обчислення *quotient*

1.2. Додавання, віднімання та множення елементів скінченного поля

Нехай $x, y \in GF(p)$, $m = p$. Результат виконання операції додавання або віднімання елементів скінченного поля $GF(p)$ позначимо Z.

Додавання та віднімання елементів скінченного поля $GF(p)$ можна реалізувати у вигляді комбінаційної схеми [19, 91], яка складається з двох суматорів, $2n$ елементів *XOR*, двох інтверторів, двох елементів *OR*, двох елементів *AND* та одного двовходового мультиплексора (рис. 1.6).

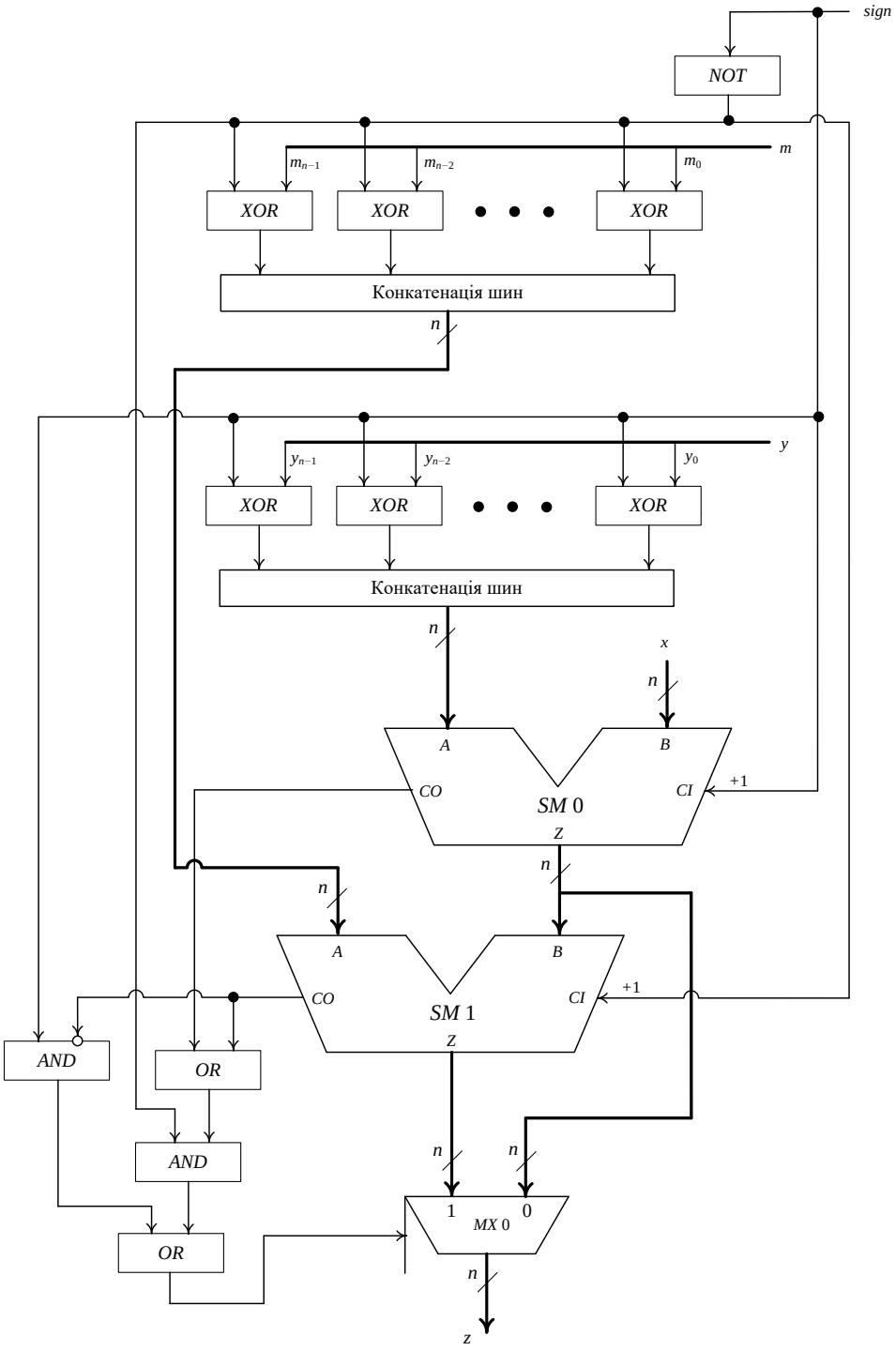


Рис. 1.6. Комбінаційна схема додавання та віднімання елементів поля $GF(p)$

Позначення *sign* вказує на знак другого операнда, тобто на код операції, що виконується, а саме: якщо $sign = 0$, то виконується додавання, якщо $sign = 1$, то віднімання. Два блоки з n елементів *XOR* відіграють роль інверторів для величин y та m, що керуються кодом операції, а саме: якщо виконується операція додавання, то блок елементів *XOR* не змінює

значення y, але інвертує значення m (при додаванні може виникнути потреба відняти m від результату), якщо виконується операція віднімання, то блок елементів *XOR* не інвертує значення y, але не змінює значення m (при відніманні може виникнути потреба додати m до отриманого результату).

Розглянемо операцію множення елементів скінченного поля $GF(p)$.

Позначимо операнди x та y. Значення модуля $m = p$. Нехай модуль є n-розрядною величиною, тоді величини x та y є не більше ніж n-розрядними. Найбільш простим, зі схемотехнічної точки зору, способом реалізації множення елементів поля $GF(p)$ є використання довільного помножувача для множення n-розрядних величин та зведення отриманого $2n$-розрядного результату за модулем m (рис. 1.7). Редукцію можна виконати за допомогою будь-якої схеми цілочисельного ділення.

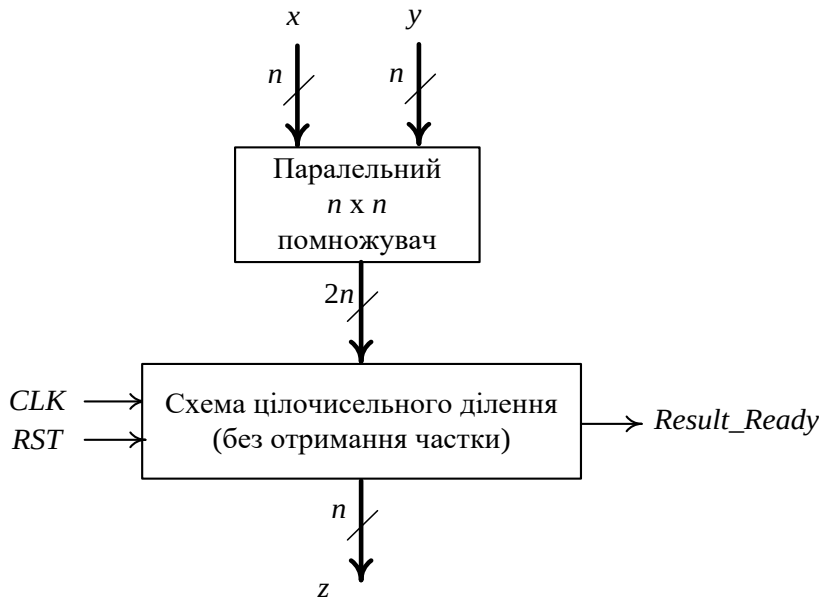


Рис. 1.7. Схема множення двох елементів поля $GF(p)$

1.3. Знаходження мультиплікативно оберненого елемента у скінченному полі

Модулярна арифметика є базовою при реалізації більшості криптографічних алгоритмів, зокрема алгоритмів асиметричної криптографії [45-48, 66, 67]. Однією з найбільш обчислювально-витратних

операцій модулярної арифметики є обчислення мультиплікативно оберненого елемента у кільці лишків за модулем m, де $m \geq 2$ і m є цілим числом. Ця операція багаторазово використовується при виконанні множення точки еліптичної кривої на число у афінних координатах над полем $GF(p)$, у методі обміну ключами Діффі-Хелмана, алгоритмі *RSA* та багатьох інших алгоритмах, що реалізують методи криптографії з відкритим ключем [90, 102, 103]. Окрім цього, при завадостійкому кодуванні даних та у деяких алгоритмах генерування псевдовипадкових чисел також виникає необхідність обчислення мультиплікативно оберненого елемента [81, 110]. Тому є актуальною задача пошуку та дослідження ефективних методів знаходження мультиплікативно оберненого елемента у кільці лишків за модулем m за критерієм мінімальності обчислювальної та часової складності.

Обчислення мультиплікативно оберненого елемента у полі $GF(p)$ є частковим випадком обчислення мультиплікативно оберненого елемента у кільці лишків за довільним модулем m коли значення m є простим числом. Тому розглянемо задачу обчислення мультиплікативно оберненого елемента в кільці лишків за довільним модулем m.

Класифікація методів обчислення мультиплікативно оберненого елемента у кільці лишків за модулем m

Мультиплікативно оберненим елементом до числа b у модулярній арифметиці є таке число y, для якого виконується рівність:

$$b \cdot y = 1 \left(\mathrm{mod}\, m \right).$$

Умовою існування мультиплікативно оберненого елемента є $\mathrm{НСД}\left(b; m\right) = 1$. Якщо ця умова не виконується, то мультиплікативно обернений елемент за модулем m до числа b не існує [57, 65, 68, 88, 110, 115, 124, 126, 133, 140].

Методи обчислення мультиплікативно оберненого елемента можна розділити на два великих класи [143]: методи, що є похідними від методів

знаходження НСД, та методи, що ґрунтуються на модулярному піднесенні до степеня (рис. 1.8).

Рис. 1.8. Класифікація методів знаходження мультиплікативно оберненого елемента

Методи знаходження мультиплікативно обереного елемента, що ґрунтуються на модулярному піднесенні до степеня

Методи Ейлера, Кармайкла та Аразі

Згідно теореми Ейлера, якщо $НСД(b;m)=1$, то:

$$b^{\varphi(m)}=1(\bmod m), \tag{1.6}$$

де $\varphi(m)-$ функція Ейлера.

Помноживши рівність (1.6) на b^{-1} отримаємо:

$$b^{-1}=b^{\varphi(m)-1}(\bmod m). \tag{1.7}$$

В той же час, згідно визначення функції Кармайкла [35], для будь-якого b, що задовольняє умову $НСД(m;b)=1$, є справедливою рівність:

$$b^{\lambda(m)}=1(\bmod m), \tag{1.8}$$

де $\lambda(m)-$ функція Кармайкла.

З рівності (1.8) отримаємо метод обчислення мультиплікативно оберненого елемента у кільці лишків за модулем m, що заснований на функції Кармайкла (назвемо його методом Кармайкла):

$$b^{-1} = b^{\lambda(m)-1} \pmod m.$$

Таким чином, для знаходження мультиплікативно оберненого елемента необхідно знайти значення функції Ейлера $\varphi(m)$ або функції Кармайкла $\lambda(m)$ та виконати модулярне піднесення до степеня.

Якщо модуль m є простим числом, то $\varphi(p) = \lambda(p) = p - 1$ та $b^{-1} = b^{p-2} \pmod p$.

Для випадку коли мультиплікативно обернений елемент до m за модулем b знайти простіше, ніж мультиплікативно обернений елемент до b за модулем m, використовується формула Аразі [100, 107]:

$$b^{-1} \pmod m = \frac{1 + m \cdot \left(-m^{-1} \bmod b\right)}{b}.$$

Коли число b є простим, формула Аразі спрощується до вигляду:

$$b^{-1} \pmod m = \frac{1 + m \cdot \left(-m^{b-2} \bmod b\right)}{b}.$$

Якщо b та m є складеними числами та для них виконується умова $НСД(m; b) = 1$, то для обчислення $b^{-1} \pmod m$ можна використати таку формулу:

$$b^{-1} \pmod m = \frac{1 + m \cdot \left(-m^{\lambda(b)-1} \bmod b\right)}{b}, \qquad (1.9)$$

яка дозволяє швидше знайти $b^{-1} \pmod m$, ніж формула (1.8), оскільки значення $\lambda(b)$ обчислити простіше, ніж $\lambda(m)$, за рахунок того, що $b < m$. Такий метод знаходження мультиплікативно оберненого елемента назвемо методом Аразі.

Метод Джої-Пейє

Якщо m та b є складеними числами, то при знаходженні $b^{-1}\left(\mathrm{mod}\, m\right)$ виникає необхідність обчислювати функцію Ейлера або Кармайкла. Щоб уникнути цього Джої та Пейє запропонували [53, 107] знаходити $b' = b + Cm,$ де $C -$ довільна ціла константа, а $b' -$ просте число.

Очевидним є той факт, що

$$b^{-1}\left(\mathrm{mod}\, m\right) = \left(b'\right)^{-1}\left(\mathrm{mod}\, m\right),$$

тому перехід від аргумента b до b' дозволяє уникнути обчислення функції Ейлера або Кармайкла.

Знаходження необхідної константи C можна здійснювати у різний спосіб. З метою дослідження розглянемо чотири алгоритми знаходження значень константи C, де найпростіший з яких це перебір C починаючи з 1 поки b' не буде простим числом (назвемо такий підхід до реалізації методу Джої-Пейє алгоритмом №0).

Інший підхід до вибору константи C полягає у тому, що початкове значення константи C приймається рівним $\left[1 - b^{\lambda(T)}\right] \mathrm{mod}\, T,$ де T є добутком певного набору простих чисел, а потім значення b' збільшується на величину $m \cdot T$ поки не буде отримано просте b'. Позначимо таку реалізацію як алгоритм №1 реалізації методу Джої-Пейє.

Наступний підхід до вибору C ґрунтується на зведенні модуля m до простого числа m' шляхом додавання величини пропорційної $b \cdot T$ та виконанні обчислень за наступними формулами:

$$u = b^{m'-2}\left(\mathrm{mod}\, m'\right),\ v = m^{-1}\left(\mathrm{mod}\, b\right) = \frac{1 + b \cdot \left(m' - u\right)}{m'}, \qquad (1.10)$$

$$b^{-1}\left(\mathrm{mod}\, m\right) = \frac{1 + m \cdot \left(b - v\right)}{b}. \qquad (1.11)$$

Формули (1.10) та (1.11) можна об'єднати в одну:

$$b^{-1}\left(\mathrm{mod}\, m\right) = \frac{m' + m\left(b \cdot u - 1\right)}{b \cdot m'}.$$

Назвемо таку реалізацію методу Джої-Пейє алгоритмом №2.

Якщо модуль m є дуже великим числом порівняно з аргументом b, то доцільно виконати редукцію $m' = m(\mathrm{mod}\, b)$ з подальшим зведенням m' до простого числа шляхом збільшення на величину пропорційну $b \cdot T$ та провести обчислення за формулами:

$$u = b^{m'-2}(\mathrm{mod}\, m'),$$

$$b^{-1}(\mathrm{mod}\, m) = \frac{m \cdot u - \left\lfloor \frac{m}{b} \right\rfloor + C}{m'}.$$

Алгоритм, що реалізує такий підхід, назвемо алгоритмом №3 реалізації методу Джої-Пейє.

Замість залучення цілочисельного ділення, для непарного дільника рекомендують виконувати всі операції за модулем $2^{|m|}$, де $|m|$ – бітова довжина числа m [107]. При виконанні операцій за модулем $2^{|m|}$ цілочисельне ділення зводиться до ділення за модулем $2^{|m|}$, яке швидко виконується за допомогою методу Ньютона.

Методи обчислення мультиплікативно оберненого елемента, що ґрунтуються на знаходженні НСД

Доведено, що з обчислювальної точки зору найбільш ефективними методами знаходження НСД є методи, засновані на алгоритмі Евкліда [88, 98, 110].

Ідею обчислення НСД можна пристосувати до потреб знаходження мультиплікативно оберненого елемента. Для цього застосовують розширений алгоритм Евкліда, який дозволяє знайти НСД двох чисел *m* та *b* і коефіцієнти *x* та *y* таких, що виконується рівність [17, 35, 82, 88, 110, 126]:

$$x \cdot m + y \cdot b = d, \tag{1.12}$$

де *d* – найбільший спільний дільник чисел m та *b*.

Якщо рівність (1.12) звести за модулем *m*, то отримаємо:

$$y \cdot b = d \left(\mathrm{mod}\, m \right).$$

Для випадку $d = 1$ маємо, що елемент *y* є мультиплікативно оберненим елементом до *b* за модулем *m*. В той же час рівність одиниці НСД $\left(b; m \right)$ є необхідною умовою існування мультиплікативно оберненого елемента. Таким чином, можна зробити висновок: якщо мультиплікативно обернений елемент існує, то його можна знайти за допомогою розширеного алгоритму Евкліда знаходження НСД.

При побудові розширеного алгоритму Евкліда для знаходження НСД необхідно підтримувати виконання двох рівностей:

$$u = A \cdot m + B \cdot b; \tag{1.13}$$

$$v = C \cdot m + D \cdot b, \tag{1.14}$$

причому на початку роботи алгоритму $u = m$, $v = b$.

Для того, щоб забезпечити виконання рівностей (1.13) та (1.14), на початку роботи алгоритму встановлюється $A = 1, B = 0, C = 0, D = 1$. Значення *u* та *v* в процесі роботи алгоритму змінюється так само як і при роботі звичайного алгоритму Евкліда знаходження НСД. Значення *A, B, C* та *D* змінюється таким чином, щоб підтримувати виконання рівностей (1.13) та (1.14).

Методи, що ґрунтуються на знаходженні НСД, можна розділити на два підкласи:

1) методи, що засновані на алгоритмі Евкліда;
2) методи, що засновані на бінарному алгоритмі Евкліда.

Розширений алгоритм Евкліда та його модифікації

Перший підклас (рис. 1.8) розширених алгоритмів – це розширені алгоритми Евкліда, основні ідеї яких були започатковані в Індії в V ст. [98]. Ці алгоритми ґрунтуються на тотожності $НСД\left(m; b \right) = НСД\left(b; m \bmod b \right)$, а протягом роботи алгоритмів підтримуються рівності виду (1.13) та (1.14).

Гордоном Бредлі було показано [110, 126], що під час ітераційного процесу достатньо підтримувати рівності виду:

$$A \cdot m = u \text{ та } C \cdot m = v,$$

а в кінці ітераційного процесу знайти значення другого коефіцієнта за формулою $B = \dfrac{u - A \cdot m}{b}$.

Очевидно, що з метою обчислення $b^{-1} \pmod m$ немає необхідності шукати обидва коефіцієнти рівності (1.13), а достатньо лише знайти коефіцієнт при b. Враховуючи цей факт, та використовуючи ідею Бредлі, нами запропономано модифікувати розширений алгоритм Евкліда так, щоб він знаходив лише коефіцієнт B, тобто в процесі роботи алгоритм підтримував тільки рівності виду $B \cdot b = u$ та $D \cdot b = v$. Окрім цього, пропонується зупинку ітераційного процесу виконувати за значенням $v = 1$, а не $v = 0$, як у базовому алгоритмі. Назвемо такий алгоритм удосконаленою модифікацією Бредлі розширеного алгоритму Евкліда.

Французьким вченим Лемером запропоновано модифікацію алгоритму Евкліда для знаходження *НСД* двох чисел великої розрядності [55, 110]. Його ідея полягає в тому, що для великих чисел частка $\dfrac{m}{b}$ не зміниться, якщо в числах m та b відкинути певну кількість молодших розрядів. Використовуючи цю ідею можна побудувати розширений алгоритм Лемера. Недоліком цього алгоритму є фіксована кількість молодших розрядів, яка відкидається, в той час як операнди можуть мати різну довжину. Тому доцільним є знаходження оптимальної кількості розрядів, що відкидається для заданих довжин операндів.

Бінарний розширений алгоритм Евкліда та його модифікації

Другий підклас (рис. 1.8) розширених алгоритмів – це бінарні розширені алгоритми Евкліда.

Ці алгоритми також в процесі роботи підтримують виконання рівностей (1.13) та (1.14), але на кожній ітерації відбувається ділення на 2 даних рівностей, звідки і походить назва цього підкласу алгоритмів. При цьому, на кожній ітерації виконується перевірка коефіцієнтів A та B на парність. Якщо обидва коефіцієнти парні, то виконується ділення на 2, інакше – додається b до A, а від B віднімається m. Таким чином, отримують парні числа, які далі знову ділять на 2.

Можна довести, що в тому випадку, коли m – непарне, b – парне, і не виконується умова, що A та B парні, то це завжди буде свідчити про те, що A – парне і B – непарне. Для випадку, коли m та b – непарні, парність A та B співпадає. Таким чином, достатньо перевіряти на парність лише B.

Отже, для непарного модуля в бінарних алгоритмах достатньо підтримувати лише рівність виду $B \cdot b = u$. Якщо ж модуль є парним, то для коректної роботи алгоритму під час ітераційного процесу достатньо підтримувати лише рівність виду $A \cdot m = u$, а в кінці ітераційного процесу значення B знайти за формулою $B = \dfrac{u - A \cdot m}{b}$.

Таким чином, на початку роботи бінарного алгоритму необхідно виконувати перевірку на парність модуля, а потім підтримувати рівності виду $A \cdot m = u$ або $B \cdot b = u$. Лише за рахунок цього можливо зменшити кількість операцій приблизно вдвічі.

Існують два фундаментальні підходи для побудови алгоритмів, що реалізують бінарні методи [1, 4, 13, 26, 136]: *Right-Shift (RS)* та *Left-Shift (LS)*.

Перший підхід *(RS)* ґрунтується на таких тотожностях:

1) якщо *u* та *v* парні, то $НСД(u; v) = 2 \cdot НСД\left(\dfrac{u}{2}; \dfrac{v}{2}\right)$;

2) якщо *u* парне, а *v* непарне, то $НСД(u; v) = НСД\left(\dfrac{u}{2}; v\right)$;

3) якщо *u* та *v* непарні, то $НСД(u; v) = НСД\left(\dfrac{|u - v|}{2}; v\right)$;

4) якщо $v = 0$, то $НСД(u; 0) = u$.

Згідно бінарного *RS*-алгоритму Евкліда для знаходження НСД двох чисел значення аргументів зменшується шляхом ділення на 2, що еквівалентно зсуву вправо на один двійковий розряд.

Узагальненням бінарного алгоритму є *k*-арний алгоритм [26]. При побудові даного алгоритму порівняно з бінарним термін «непарність» замінюються на «взаємнопросте з *k*» і при реалізації ітераційного процесу виконуються ділення на *k*.

Другий підхід *(LS)* ґрунтується на таких тотожностях [14, 16]:

1) $НСД(u; v) = НСД(|u + t \cdot v|; v)$, де t – будь-яке ціле число;

2) $НСД(u; 0) = u$.

Для *k*-арного алгоритму t обирається як степінь числа *k*. Згідно даного підходу під час обчислення t отримують від'ємне число та таке, що є степенем 2, тоді величину $t \cdot v$ можна знайти шляхом зсуву v на деяку кількість двійкових розрядів вліво.

Таким чином, виникає необхідність дослідження впливу значення *k* на ефективність *k*-арного методу обчислення мультиплікативно оберненого елемента у кільці лишків за модулем *m*.

Водночас існують дві модифікації бінарного *RS*-алгоритму [82], які позиціонуються як алгоритми для апаратної реалізації. Особливістю цих алгоритмів є те, що вони на кожній ітерації оперують лише двома молодшими бітами операндів. Ці алгоритми дістали назву розширених плюс-мінус алгоритмів [82].

1.4. Піднесення до степеня елементів скінченного поля

Всі методи піднесення до степеня відрізняються способами обробки показника степеня і не залежать від основи, тому розглянемо методи піднесення до степеня для поля $GF(p)$, а для поля $GF(2^m)$ ці методи також

будуть справедливими з точністю до заміни арифметики чисел на арифметику многочленів [5, 39, 51, 89, 96, 99, 101, 104, 108, 112].

Загалом, всі алгоритми піднесення до степеня ґрунтуються на поданні показника степеня у певній системі числення та розгляді розрядів цього подання зліва направо, тобто від старшого до молодшого – *LR* і справа наліво, від молодшого до старшого – *RL*. Ці методи можна розділити на наступні категорії (рис. 1.9).

Бінарні методи

У комп'ютерній техніці доцільніше використовувати двійкову системою числення. Відповідно алгоритми, що ґрунтуються на такому поданні показника степеня дістали назву бінарних. Вперше ця ідея була використана Дональдом Кнутом [110] для мультиплікативної групи, тобто для піднесення до степеня відносно операції множення. Пізніше Хенкерсон та Менезес [103, 116, 117] використали цю ідею для побудови алгоритмів піднесення до степеня в адитивній групі. Але незалежно від виду групи ідея алгоритму залишається незмінною з точністю до заміни знаку додавання на множення. Згідно даного методу, коли число представлене в двійковому вигляді, при порозрядному аналізі даного подання, якщо черговий розряд дорівнює 1 – виконується піднесення до квадрата та домноження накопиченого значення добутку, а якщо 0, то тільки піднесення до квадрата. Алгоритми А.5 та А.6 ілюструють бінарний *RL*-алгоритм та *LR*-алгоритм виконання операції піднесення до степеня. Алгоритм А.5 піднесення до степеня розглядає біти числа k справа наліво, в той час як алгоритм А.6 піднесення до степеня – зліва направо. Приклади роботи даних алгоритмів наведено у Додатку Б.

Інші бінарні алгоритми, що розглядаються в [130] – це *RL*-алгоритм Джої та *LR*-алгоритм Монтогомері. У праці [130] алгоритми побудовані для адитивної групи точок еліптичної кривої над скінченним полем, але їх можна застосувати для мультипілкативної групи кільця лишків за

модулем m шляхом заміни операції додавання на множення (алгоритм А.7 та А.8 піднесення до степеня у додатку А). Дані алгоритми базуються на інваріантах $R_0 \cdot R_1$ та $\frac{R_1}{R_0}$ відповідно, що змінюються в циклі. Метод Джої на кожній i-тій ітерації підтримує виконання рівності: $R_0 \cdot R_1 = a^{2^i}$, а метод Монтгомері в кінці кожної ітерації циклу задовольняє наступну рівність: $\frac{R_1}{R_0} = a$, де a – це число, яке потрібно піднести до степеня (основа), R_0 – це змінюване на кожній ітерації значення основи, а R_1 – це накопичуваний результат.

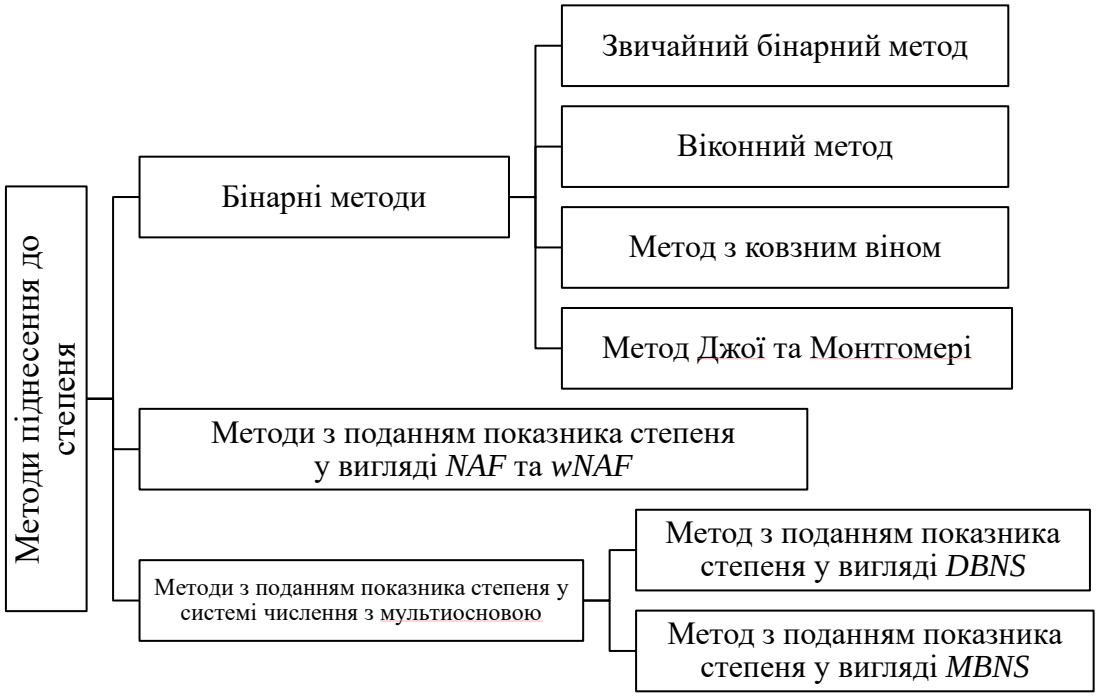


Рис. 1.9. Класифікація методів піднесення до степеня

Одним з напрямків модифікації звичайних бінарних методів є побудова таблиць, що містять наперед обчислені значення степеня для маленьких показників степеня та даної основи, які надалі використовуються при реалізації алгоритму піднесення до степеня.

У [9, 51, 86, 90, 103, 122, 123, 130] розглядаються різні модифікації бінарних віконних алгоритмів піднесення до степеня в адитивній групі, які завдяки побудові таблиць передобчислень (*precomputation table*) на початкових стадіях алгоритмів дають прийнятні показники швидкодії та можуть бути застосовані для мультиплікативної групи шляхом зміни операції додавання на множення.

Віконні методи отримали таку назву через те, що в них розглядається не по одному двійковому розряду показника степеня, а по w розрядів, де w – довжина вікна. У [130] Метью Рівайн розглядає два види віконних алгоритмів: *RL*- та *LR* -алгоритми (алгоритм А.9 та А.10 піднесення до степеня у додатку А).

Нехай k – деякий скаляр, який є показником степеня, тоді його подання у системі числення за основою 2^w матиме вигляд

$$k = \sum_{i=0}^{l-1} k_i 2^{iw}, \tag{1.15}$$

де $k_i \in \{0,1,...2^w - 1\}$, $k_{l-1} \neq 0$, $l = \left\lceil \frac{n}{w} \right\rceil$ та n – бітова довжина скаляра k.

Віконний *RL*-алгоритм передбачає виконання обчислень за такою формулою:

$$a^k = \prod_{i=0}^{l-1} \left(a^{k_i}\right)^{2^{iw}}. \tag{1.16}$$

Віконний *LR*-алгоритм полягає у виконанні обчислень за наступними формулами:

$$T_{l-1} = a^{k_{l-1}} \tag{1.17}$$

$$T_i = \left(T_{i+1}\right)^{2^w} + a^{k_i},\ i = l-2..0. \tag{1.18}$$

Після проведення обчислень за формулами (1.17) - (1.18) значення a^k буде знаходитися у T_0.

Для обох алгоритмів на початковій стадії будується таблиця передобчислень, яка складається з $2^w - 1$ елементів.

При аналізі алгоритмів А.9 та А.10 піднесення до степеня стає зрозумілим, що *RL*-алгоритм має більшу обчислювальну складність ніж *LR*-алгоритм, оскільки у *RL*-алгоритмі на кожній ітерації циклу виконується переобчислення всіх значень, що записані у таблицю, тому далі ми будемо розглядати лише *LR*-алгоритми реалізації віконних методів.

Наступним методом піднесення до степеня, що розглядається в роботі [103], є метод з ковзним вікном (алгоритм А.11 піднесення до степеня у додатку А). Даний метод дістав таку назву через те, що згідно цього методу необхідно виділяти вікно завдовжки *w* біт тільки якщо старший біт дорівнює 1, в іншому випадку виконують дії, передбачені відповідним звичайним бінарним алгоритмом.

При розробленні віконного методу з ковзним вікном ставилось за мету досягти компромісу між кількістю множень та піднесень до квадрата. На початковій стадії цього методу будується таблиця передобчислень, що містить елементи $a^t \bmod m$ для $t = \{2^{w-1}, 2^{w-1}+1, ..., 2^w - 1\}$, за рахунок чого досягається зменшення використання оперативної пам'яті вдвічі порівняно з бінарним віконним методом.

Методи з поданням показника степеня у вигляді NAF (Non-Adjacent Form)

Одним з напрямків прискорення методів піднесення до степеня є перетворення показника степеня у *NAF*-подання [103]. *NAF*-подання числа є нічим іншим як поданням числа у симетричній трійковій системі числення [40]. Подання показника степеня *k* у формі *NAF* виражається формулою $k = \sum_{i=0}^{l-1} k_i 2^i$, де $k_i \in \{0; \pm 1\}$ та $k_{l-1} \neq 0$.

NAF подання числа *k* є унікальним і позначається *NAF*(*k*).

Властивості *NAF*-форми: *NAF*(*k*) має найменшу кількість ненульових цифр порівняно з іншими системами числення зі знаком; довжина *NAF*(*k*) перевищує довжину бінарного подання числа *k* не більше ніж на один

розряд; якщо довжина *NAF*(*k*) дорівнює *l*, тоді $\frac{2^l}{3} < k < \frac{2^{l+1}}{3}$; ненульові цифри не можуть бути суміжними при такому поданні числа, що пояснює його назву (*non-adjacent* – не суміжна); середня щільність ненульових цифр серед всіх *NAF*-форм завдовжки *l* приблизно дорівнює $\frac{1}{3}$.

Ефективно знайти *NAF*(*k*) можна використовуючи алгоритм А.12 перетворення додатного цілого числа у *NAF*-подання (додаток А). Двійкові розряди *NAF*(*k*) генеруються повторенням ділення числа *k* на 2, допустимими остачами є 0 або ±1. Якщо *k* є непарним, тоді остача *r* дорівнює –1 або 1, а якщо частка – парна, то наступна цифра *NAF*-подання буде дорівнювати 0.

Маючи алгоритм отримання *k* у вигляді *NAF*, піднесення до степеня *k* виконують за допомогою алгоритма А.13 або А.14 (додаток А).

Метод з поданням показника степеня у вигляді *NAF*, є ефективнішим (при наявності *NAF*-розкладання показника степеня), ніж звичайні бінарні методи через те, що два сусідні розряди не можуть бути одночасно ненульовими, а це зменшує кількість операцій множення та ділення.

Віконна реалізація даного методу дістала назву метод з поданням показника степеня у вигляді *wNAF*. Подання показника степеня у *wNAF*-формі виражається формулою (1.19), де кожне ненульове t_i є непарним та таким, що $t_i \in \left[-2^{w-1}; 2^{w-1}-1\right]$, $t_{n-1} \neq 0$:

$$k = \sum_{i=0}^{n-1} t_i 2^i . \qquad (1.19)$$

Окрім цього *wNAF* подання має властивість, що серед *w* послідовних розрядів цього подання знаходиться не більше одного ненульового розряду, за рахунок чого зменшується загальна кількість множень та ділень при використанні цього подання в алгоритмах піднесення до степеня.

На початковій стадії реалізації *LR*-алгоритму цього методу піднесення до степеня (алгоритм А.16 методу піднесення у додатку А) потрібно крім побудови таблиці передобчислень перетворити скаляр у *wNAF*-подання використовуючи алгоритм А.15 перетворення додатного цілого числа у *wNAF* подання (додаток А).

Таким чином, метод з поданням показника степеня у вигляді *wNAF* є аналогом віконного методу, але оперуючи знаком *NAF*-подання, тільки за рахунок цього, дозволяє скоротити обсяг необхідної для зберігання таблиці передобчислень пам'яті вдвічі порівняно з класичним віконним методом. Враховуючи особливості *wNAF*-подання цілих додатних чисел таблиця передобчислень буде складатися з елементів $a^i \pmod m$, де $i = 1,3,...,2^{w-1}-1$ (використовуються тільки непарні показники степеня), тобто порівняно з бінарним віконним методом отримуємо скорочення обсягу таблиці передобчислень у чотири рази.

Логічним вдосконаленням методу з поданням показника степеня у вигляді *wNAF* є метод з вікном змінної довжини та поданням показника степеня у вигляді *NAF*. На відміну від попереднього алгоритму, де необхідно перетворити число k у *wNAF*-подання, у даному алгоритмі потрібно перетворити його у *NAF*-форму, використовуючи алгоритм А.12 піднесення до степеня (додаток А), а потім виконувати піднесення до степеня за алгоритмом А.17 піднесення до степеня (додаток А).

Таблиця передобчислень для цього методу буде містити такі елементи $a^i \pmod m$, де $i = 1,3,...,\frac{2(2^w-(-1)^w)}{3}-1$.

Як уже зазначалося вище, перевага цього методу полягає у тому, що два сусідніх k_i не можуть бути одночасно ненульовими, тобто буде менша кількість множень та ділень порівняно з бінарними методами, але в свою чергу потрібно попередньо перетворити скаляр k у *NAF*-подання.

Методи з поданням показника степеня у системі числення з мультиосновою (MBNS)

Найбільш широко використовується частковий випадок *MBNS* (*MultyBase Number System*)-подання, а саме *DBNS* (*DoubleBase Number System*). Також зустрічається в науковій літературі синонімічні абревіатури до абревіатур *MBNS* та *DBNS*, а саме *MBNR* (*MultyBase Number Representation*) та *DBNR* (*DoubleBase Number Representation*).

DBNS – це схема подання цілого числа [95, 96], в якій кожне додатне ціле число k можна подати у системі числення з подвійною основою, зазвичай в якості основи обираються числа 2 та 3, наступним чином:

$$k = \sum_{i=1}^{l} s_i 2^{x_i} 3^{y_i}, \text{ де } s_i \in \{-1,1\}, \text{ і } x_i, y_i \geq 0. \tag{1.20}$$

Існує багато методів перетворення цілого числа k у системи числення з подвійною основою, і всі модифікації методу піднесення до степеня з використанням такої системи числення відрізняються способом отримання *DBNS* подання показника степеня. Васіл Дімітров показав [95], що якщо розглядати *DBNS* тільки для значення $s_i = 1$, то, наприклад число 10_{10} має п’ять різних *DBNS* представлень, 100_{10} має 402 різні *DBNS* подання, а 1000_{10} має 1 295 576 різні *DBNS*-подання.

Деякі з цих представлень викликають особливий інтерес. Найбільш цікавим є подання, що вимагає мінімальної кількості розрядів у системі числення з основою $\{2,3\}$. Наприклад, цілі числа можуть бути подані за допомогою l розрядів цієї системи числення, але не може бути представлене за допомогою $l-1$ або менше розрядів. Таке поданням називається канонічним поданням та зустрічається дуже рідко серед множини всіх можливих *DBNS*-розкладань.

Ефективне знаходження одного з канонічних *DBNS*-представлень, особливо для дуже великих цілих чисел є непростою задачею. Для того, щоб знайти одне з розкладань, можна застосувати жадібний алгоритм [95], що

полягає у знаходженні на кожному кроці найкращого наближення до поточного числа використовуючи $\{2,3\}$ -основу, тобто отримати ціле число виду 2^x3^y. Потім обчислити різницю між поточним числом та 2^x3^y і повторити процес для отриманої різниці. У загальному випадку цей алгоритм дає неканонічне подання числа у *DBNS*, однак в більшості випадків результатом його роботи є канонічне розкладання.

У роботі [95] вводиться поняття розкладання у системі числення з основою $\{2,3\}$ (*DB*-розкладання). Довжина *DB*-розкладання дорівнює кількості $\{2,3\}$ -основ в $k=\sum_{i=1}^{l} s_i 2^{x_i}3^{y_i}$, де $s_i \in \{-1,1\}$, і $x_i, y_i \geq 0$ використану для подання *k*. На основі поняття *DB*-розкладання побудований алгоритм А.18 перетворення цілого числа у *DBNS* (додаток А).

Приклад *DB*-розкладання для показника степеня $k=841232$:

$$841232 = 2^7 3^8 + 1424$$

$$1424 = 2^1 3^6 - 34$$

$$34 = 3^3 + 7$$

$$7 = 3^2 - 2$$

$$2 = 3^1 - 1$$

В результаті отримуємо наступне *RL*-розкладання:

$$a^{841232} = a^{2^7 3^8 + 2^1 3^6 - 3^3 - 3^2 + 3^1 - 1} =$$

$$= a^{2^7 3^8} \cdot a^{2^1 3^6} \cdot \left(a^{-1}\right)^{3^3} \cdot \left(a^{-1}\right)^{3^2} \cdot a^{3^1} \cdot a^{-1}.$$

LR-розкладання матиме вигляд:

$$a^{841232} = a^{3\cdot(3\cdot(3\cdot(2^1 3^3\cdot(2^6 3^2+1)-1)-1)+1)-1} =$$

$$= \left(\left(\left(\left(a^{2^6 3^2}\cdot a\right)^{2^1 3^3}\cdot a^{-1}\right)^{3^1}\cdot a^{-1}\right)^{3^1}\cdot a\right)^{3^1}\cdot a^{-1}.$$

Існують два підходи реалізації кроку 2.1 алгоритму А.18 перетворення цілого числа у *DBNS* (додаток А):

- використання функції

$$y = -x\log_3 2 + \log_3 k, \qquad (1.21)$$

- використання табличних значень $3^0; 3^1; ...; 3^m$.

Перший підхід вимагає мінімальне використання додаткової пам'яті, але водночас потребує застосуванням арифметики дійсних чисел.

Згідно першого підходу необхідно шляхом перебору точок $(x; y)$, що знаходиться поблизу прямої (1.21), знайти цілі невід'ємні значення x та y, такі що $2^x \cdot 3^y$ дає найкраще наближення до k.

Перший підхід пропонується реалізувати шляхом перебору всіх можливих значень показника степеня 2 в діапазоні від 1 до x_{max}, де x_{max} задане максимально можливе значення показника степеня 2, та обчислення для кожного з них нев'язки $\lceil y \rceil - y$. Пара $\{x; y\}$ з найменшим значенням нев’язки обирається за шукані показники степеня 2 та 3.

Другий підхід передбачає побудову списку двійкового подання чисел $3^0; 3^1; ...; 3^{y_{max}}$ та його впорядкування у лексикографічному порядку. Потім шукають два найближчі елементи цього списку до двійкового подання числа k, тобто відшукуються два такі елементи у впорядкованому списку, що вставка між ними двійкового подання числа k не порушить впорядкованість списку. На наступному кроці другого алгоритму, для показників степеня 3 (y_1 та y_2), що відповідають обраним елементам, обчислюють показник степеня 2 (x_1 та x_2), який забезпечує найкращу апроксимацію числа k добутком $2^{x_1} \cdot 3^{y_1}$ та $2^{x_2} \cdot 3^{y_2}$ відповідно. Далі обирається найкраща пара показників степеня.

Маючи подання показника степеня у *DBNS* піднесення до степеня виконується за алгоритмом А.19 піднесення до степеня (додаток А). Для прискорення роботи алгоритму А.19 піднесення до степеня доцільно перед

початком роботи алгоритму побудувати таблицю, що буде містити значення a у степені кожного елемента множини S за модулем m.

Наступний спосіб отримання *DBNS*-подання показника степеня є реалізація алгоритму, що базується на побудові дерева [95]. Структуру дерева задають три параметри: *B, S* та *D*. Розглянемо побудову дерева для таких значень параметрів $B=2$, $S=\{-1;1\}$, $D=\{2;3\}$. Процес побудови відбувається наступим чином. Виконується цілочисельне ділення числа *k* на 2 (перший елемент множини *D*) поки остача не стане рівною нулю, потім на 3 (другий елемент множини *D*). При діленні накопичуються показники степеня. До отриманого числа додається 1 (перший елемент множини *S*), і це буде лівий нащадок даного вузла дерева, та віднімається 1 (другий елемент множини *S*) – правий нащадок. На кожному кроці алгоритму вибирається *B* мінімальних нащадків поточного рівня і продовжується робота з ними. Алгоритм триватиме поки один з нащадків не стане рівним 1.

RL та *LR* алгоритми подання показника степеня у вигляді *DBNS*, що базуються на побудові дерева, відрізняються способом обходу дерева: зверху вниз – *RL*-алгоритм, знизу вгору – *LR*-алгоритм.

Приклад *RL*-розкладання показника степеня, що базуються на побудові дерева:

$$841232 = 2^{18}3^1 + 2^{14}3^1 + 2^{11}3^1 - 2^9 + 2^4. \qquad (1.22)$$

Якщо розбити дерево на умовні рівні, то вийде 5 рівнів, до того часу як ми отримаємо 1 на останньому з них. У зв'язку з цим приклад *RL*-розкладання має 5 доданків. Як зазначалось вище, у *RL*-розкладанні ми проходимо дерево згори вниз, тобто перший рівень буде верхній на рис. 1.10. Показник степеня двійки та трійки у першому доданку розкладання (1.22) – це сума степенів двійок і трійок на всіх 5-ти рівнях відповідно. Другий доданок – це така сама сума, але уже на перших чотирьох рівнях, третій – на перших трьох, і т.д. Останній доданок – це показник степеня двійки на першому рівні. Знак першого доданка завжди додатний, далі знак перед кожним доданком визначається в залежності від

того який нащадок входить до ланцюжка, що утворює розкладання: якщо правий, то – мінус, лівий – плюс. Наприклад, перед четвертим доданком стоїть мінус, тому що елемент ланцюжка на третьому рівні є правим нащадком елемента з попереднього рівня.

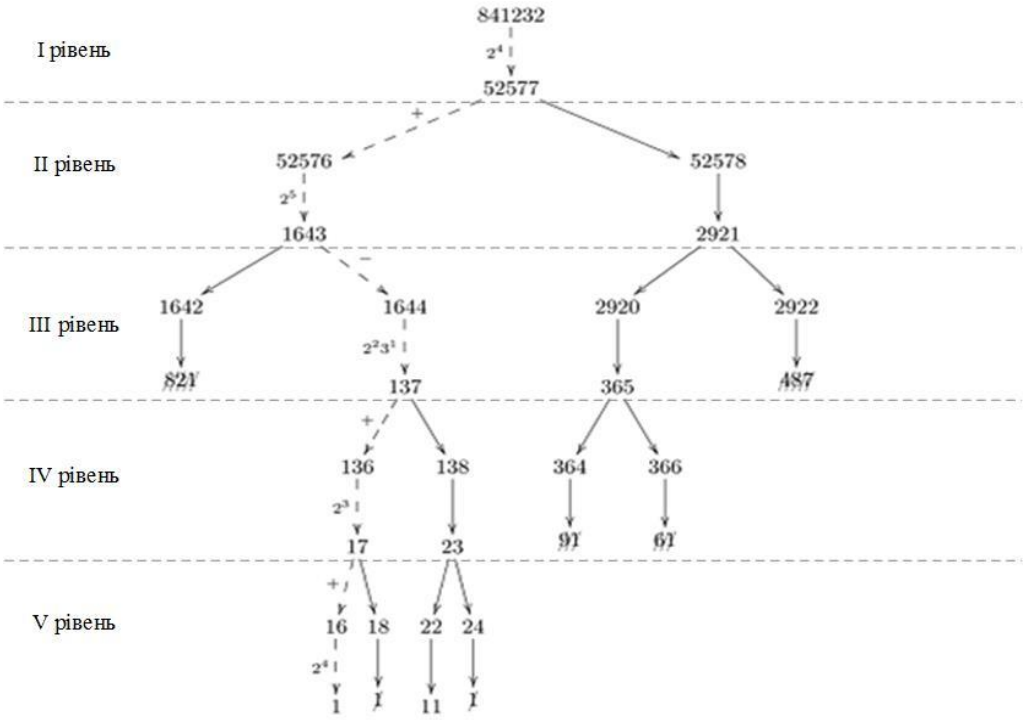


Рис. 1.10. Приклад побудови дерева для *DBNS*-подання числа *k*, де $B=2$

Приклад *LR*-розкладання показника степеня, що базується на побудові дерева:

$$841232=2^4\left(2^5\left(2^2 3^1\left(2^3\left(2^4+1\right)+1\right)-1\right)+1\right).$$

У *LR*-розкладанні обхід дерева здійснюється навпаки – знизу вгору. Розглянутий приклад *LR*-розкладання складається з таких груп: 2^4+1, $2^3\left(2^4+1\right)+1$, $2^2 3^1\left(2^3\left(2^4+1\right)+1\right)-1$, $2^5\left(2^2 3^1\left(2^3\left(2^4+1\right)+1\right)-1\right)+1$ та $2^4\left(2^5\left(2^2 3^1\left(2^3\left(2^4+1\right)+1\right)-1\right)+1\right)$. Тут також 5 груп, які відповідають 5-ти рівням, але нумерація рівнів починається знизу дерева на рис 1.10.

1.5. Висновки до розділу 1

Аналіз методів виконання операцій у скінченних полях показав необхідність проведення комплексного дослідження, спрямованого на підвищення швидкості обчислень в скінченних полях за рахунок структурно-логічної оптимізації процесів виконання операцій. Поставленої мети дослідження можна досягти шляхом вирішення таких задач.

1. Розроблення методів високошвидкісного виконання операцій в полях Галуа виду $GF(p)$: додавання, віднімання, множення, ділення, піднесення до степеня, обчислення мультиплікативно оберненого елемента. Найбільш перспективними для поліпшення часових характеристик є віконні методи піднесення до степеня та методи з поданням показника степеня у системі числення з мультиосновою.
2. Синтез структур апаратних засобів для реалізації операцій арифметики скінченних полів, які б забезпечували більш високу швидкість обчислень порівняно з існуючими рішеннями.
3. Розроблення методів апаратної реалізації операцій в полях Галуа виду $GF(2^m)$.
4. Розроблення архітектури та системи команд проблемно-орієнтованого процесора Галуа для виконання операцій у скінченних полях.
5. Розроблення моделі обчислювального процесу, що має місце при реалізації операцій у скінченних полях.
6. Створення засобів для моделювання та дослідження ефективності методів виконання операцій в полях Галуа.

РОЗДІЛ 2
АПАРАТНА РЕАЛІЗАЦІЯ ОПЕРАЦІЙ В СКІНЧЕННИХ ПОЛЯ ВИДУ *GF*(*P*)

2.1. Модифіковані алгоритми редукції значень величин у скінченному полі *GF*(*p*)

Розглянемо модифікацію алгоритму А.2 редукції (додаток А).

В тому випадку, коли на вхід надходить значення $x \le m$, можна не виконувати обчислення, передбачені алгоритмом А.2, а одразу видавати результат на вихід схеми. Коли $x < m$, то на вихід видається значення x як результат; коли $x = m$, то на вихід видається значення 0, в інших випадках необхідно виконувати обчислення, передбачені алгоритмом цілочисельного ділення.

Для певного набору вхідних даних можна прискорити алгоритм А.2 редукції (додаток А) шляхом зменшення кількості ітерацій. Цього можна досягти, якщо замість виконання зсуву значення m на $n-2$ розрядів вліво виконувати зсув значення модуля на мінімально необхідну кількість розрядів. Математично критерій зупинки зсувів вліво значення модуля можна записати наступним чином: виконуємо зсув вліво значення модуля на один розряд поки не виконується умова

$$\left(\left(x \text{ xor } m\right) > x\right) \text{and } m > x, \tag{2.1}$$

а потім виконуємо один зсув вправо.

Розглянемо докладніше умову (2.1). Вона складається з двох умова

$$\left(x \text{ xor } m\right) > x \tag{2.2}$$

та

$$m > x. \tag{2.3}$$

Умови (2.2) та (2.3) є справедливими у двох випадках, один з яких є хибним для критерія зупинки зсувів, але коли хибно спрацьовує умова (2.2), то умова (2.3) сигналізує, що критерій зупинки зсувів не виконується, і навпаки, якщо хибно спрацьовує умова (2.3), то умова (2.2) сигналізує, що критерій зупинки зсувів не виконується. Отже, зсув вліво значення модуля необхідно виконувати поки не стане справедливою умова (2.1), де and – логічна операція, а xor – побітова.

В алгоритмі А.2 редукції (додаток А) незалежно від фактичної розрядності операндів виконується $n-1$ ітерація основної частини алгоритму. У запропонованій модифікації кількість ітерацій основної частини алгоритму буде дорівнювати кількості виконаних зсувів вліво модуля (алгоритм А.20 редукції).

При такому підході кількість зсувів буде дорівнювати $\tilde{n}-\tilde{k}+2$ на другому та третьому кроках алгоритму 2.1 та $\tilde{n}-\tilde{k}$ на четвертому кроці. Загальна кількість зсувів $2\left(\tilde{n}-\tilde{k}+1\right)$.

При підході, що розглянуто в [23] зсуви на другому та четвертому кроці алгоритму А.2 редукції (додаток А) виконуються шляхом відповідного з'єднання шин та не потребують тактів процесора, але під час виконання алгоритму буде виконуватись $n-2-\left(\tilde{n}-\tilde{k}+1\right)$ зайвих ітерацій ("холостих ітерацій"). Одна ітерація виконується за один такт процесора, за рахунок того, що зсув на один розряд вліво виконується шляхом відповідного з'єднання шин. Тому запропонована модифікація алгоритму буде ефективною коли $n-2-\left(\tilde{n}-\tilde{k}+1\right)>2\left(\tilde{n}-\tilde{k}+1\right)$. Спрощуючи останню нерівність отримаємо $n+3\tilde{k}-3\tilde{n}-3>0$. Легко бачити, що ця нерівність буде виконуватись коли розрядність регістра діленого буде великою, а фактична розрядність операндів малою. Наприклад, якщо $n=100$, $\tilde{n}=35$, $\tilde{k}=7$, то

$$n+3\tilde{k}-3\tilde{n}-3=100+21-105-3=13>0.$$

При цьому доцільно ввести додатковий параметр s. Даний параметр буде вказувати мінімально можливу різницю бітової довжини операндів, тобто s буде значенням, меншою за яке не може бути різниця бітової довжини операндів. Тоді можна виконувати зсув на s розрядів шляхом з'єднання шин, що скоротить загальну кількість тактів роботи схеми, а саме: алгоритм А.20 редукції (додаток А) буде працювати швидше за алгоритм А.2 редукції (додаток А), коли

$$n + 2s + 3\tilde{k} - 3\tilde{n} - 3 > 0. \qquad (2.4)$$

Апаратна реалізація алгоритму А.20 редукції наведена на рис. 2.1. У даній схемі використовується два лічильники. Нульовий лічильник ініціалізується значенням s, перший – нульовим значенням. Нульовий та перший лічильник працюють в режимі інкременту під час зсувів значення модуля вліво. Після завершення зсувів значення модуля вліво у нульовому лічильнику буде знаходитись значення, що відповідає необхідній кількості ітерацій алгоритму цілочисельного ділення, а у першому лічильнику буде знаходитись значення необхідної кількості зсувів результату обчислення вправо на завершальному етапі роботи алгоритму. Далі під час проведення ітераційного процесу нульовий лічильник працює в режимі декременту та виконується зупинка ітераційного процесу за нульовим значенням у нульовому лічильнику. На завершальному етапі роботи алгоритму (зсув вправо результату) перший лічильник працює в режимі декременту та зсув результату вправо зупиняється при досягненні нульового значення у першому лічильнику.

Алгоритм А.20 редукції можна доповнити таким чином, щоб результатом його роботи, окрім остачі від ділення, також була частка (алгоритм А.21 редукції).

Порівняно з апаратною реалізацією алгоритма А.20 редукції (рис. 2.1) необхідно додати блок обчислення значення q, який використовується у схемі, що наведена на рис. 1.3.

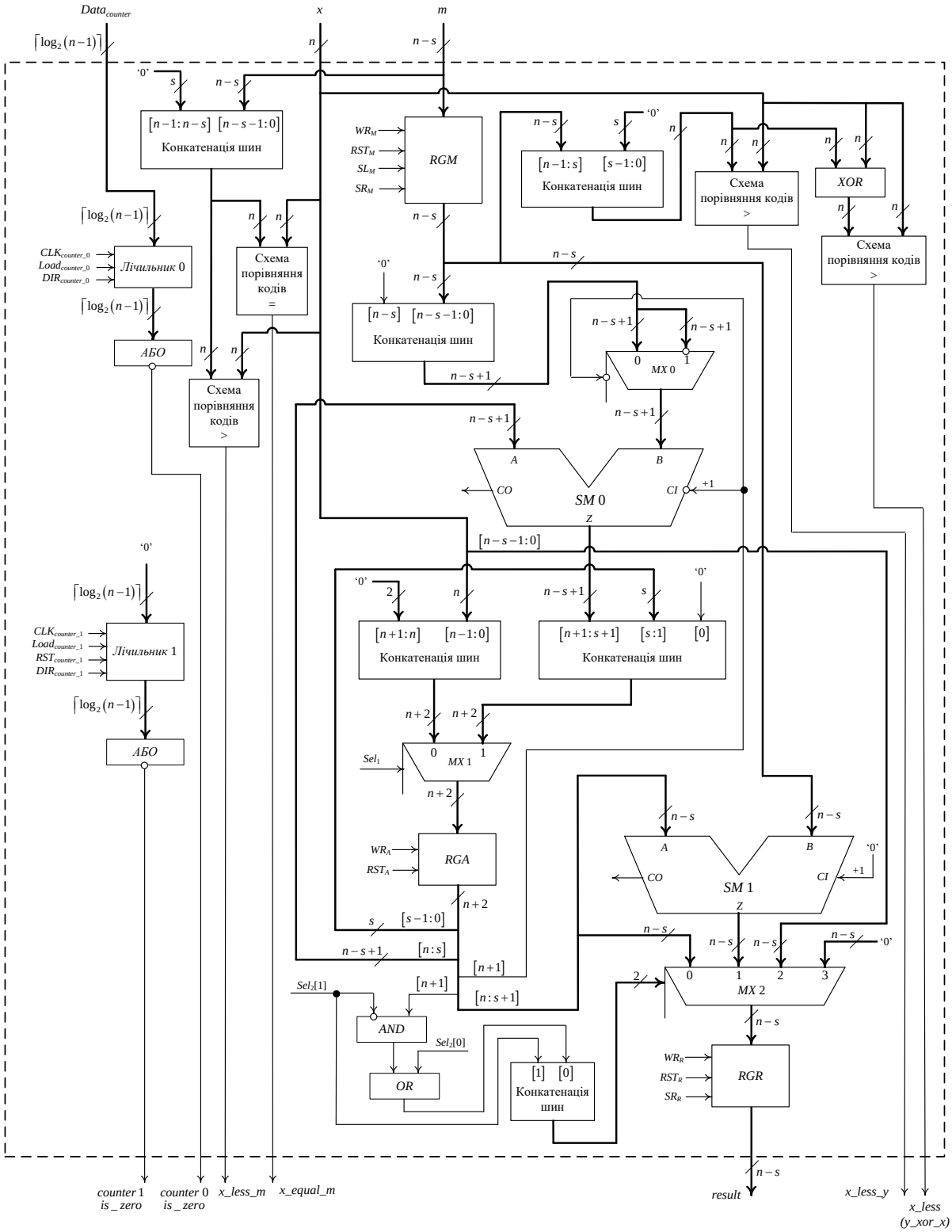


Рис. 2.1. Операційний автомат схеми, що реалізує алгоритм А.20 редукції

Розглянемо блок обчислення значення q докладніше (рис. 2.2).

При аналізі даної схемної реалізації (рис. 2.2) було встановлено, що мультиплексор та суматор, які використовується на виході частки q, не

потрібні. Суматор виконує віднімання 1 від результату, в якому молодший розряд завжди дорівнює 1, згідно алгоритму. Таким чином, при відніманні 1 від результату інвертується молодший розряд та інших дій не відбувається. Отже, суматор впливає тільки на молодший розряд результату. Результат з суматора видається на вихід схеми тільки якщо знаковий розряд регістра *A* буде дорівнювати 1 (за це відповідає мультиплексор *MX* 3), в іншому випадку на вихід видається не кориговане значення. На основі цього можна зробити висновок, що коли знаковий розряд регістра *A* дорівнює 1, то молодший біт результату буде дорівнювати 0, в іншому випадку 1. Тому суматор і мультиплексор можна замінити на однорозрядний інвертор знакового розрядку регістра *A*.

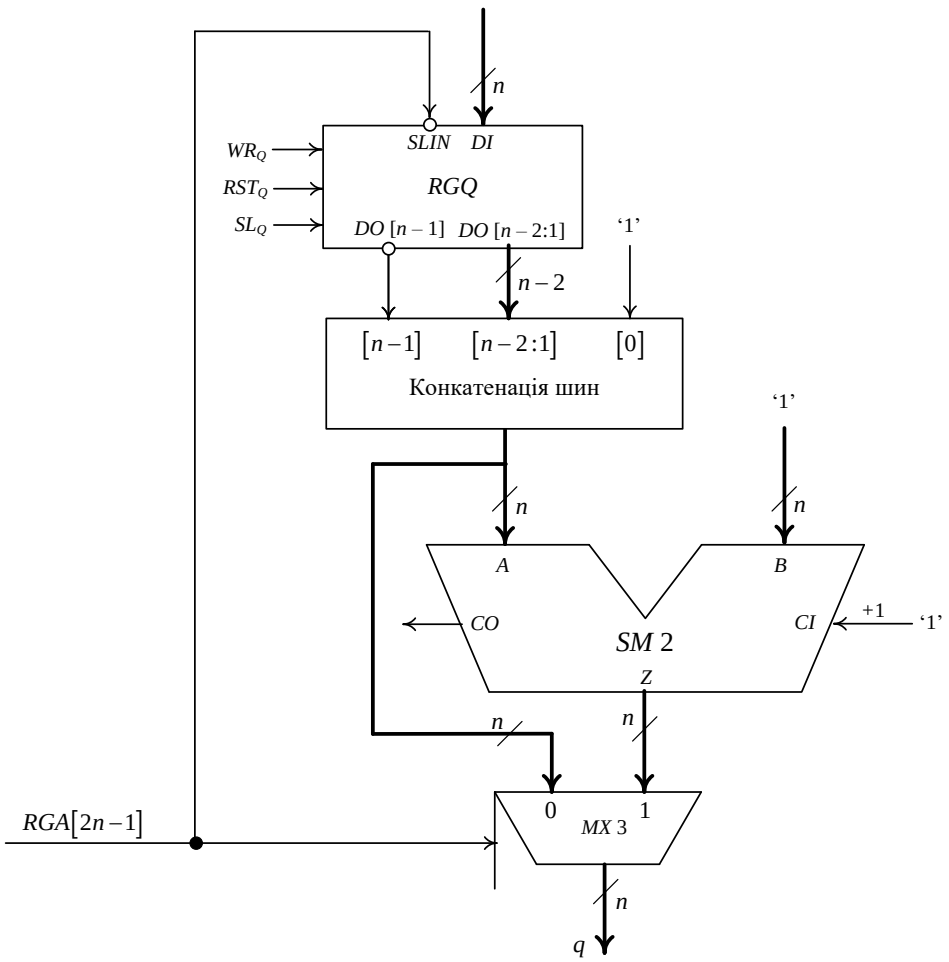


Рис. 2.2. Блок обчислення частки при апаратній реалізації алгоритму А.4 редукції

Окрім цього, потрібно модифікувати інвертування старшого розряду регістра частки, оскільки в базовому алгоритмі (алгоритм А.4 редукції) завжди виконується $n-1$ зсув, а у модифікованому алгоритмі виконується $\tilde{n}-\tilde{k}+1$ і позиція старшого розряду частки, який потрібно інвертувати буде змінюватись залежно від значень операндів. Таким чином, у регістр Q має записуватись одразу коректне значення старшого розряду. Це можна реалізувати шляхом додавання однорозрядного мультиплексора, який буде видавати на вхід заповнення молодшого розряду регістра Q пряме значення знакового розряду регістра A або інвертоване. Отже, на першій ітерації (під час формування старшого розряду частки) мультиплексор має видавати на вихід пряме значення знакового розряду регістра A, а на всіх інших ітераціях – інвертоване.

Схемотехнічно модифікований блок обчислення частки буде виглядати як на рис. 2.3.

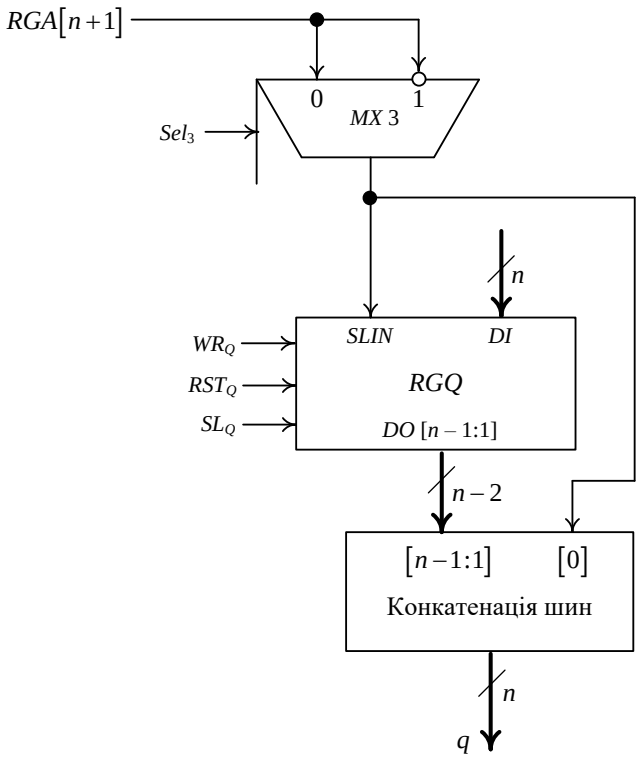


Рис. 2.3. Апаратна реалізація модифікованого блока обчислення частки

Як зазначалось в розділі 1 недоліком *SRT*-алгоритму є той факт, що розрядність регістру модуля має дорівнювати фактичній розрядності модуля, тобто обов'язковим є виконання умови $k = \tilde{k}$. Тому актуальним є побудувати модифікований *SRT*-алгоритм для апаратної реалізації, який буде дозволяти виконувати операції над всіма аргументами, що задовольняють нерівність $\tilde{n} \le n$ та $\tilde{k} \le k$.

Можна запропонувати два шляхи модифікації *SRT*-алгоритму для цілих чисел.

1. До початку ітераційного процесу привести обидва аргументи до вигляду, коли у старшому розряді відповідного регістра міститься 1 (нормалізований вигляд). При такому підході кількість ітерацій основної частини алгоритму буде мінімально можливою і дорівнюватиме $\tilde{n} - \tilde{k} + 1$. Водночас недоліком цього підходу є те, що необхідна надлишкова кількість тактів процесора для встановлення обох аргументів у нормалізований вигляд.
2. До початку ітераційного процесу привести лише модуль до нормалізованого вигляду, що є необхідною умовою для коректної роботи цього алгоритму. При такому підході під час нормалізації відсутні надлишкові операції, але це спричинює необхідність виконувати надлишкові ітерації під час ітераційного процесу через неможливість встановити мінімально необхідну кількість ітерацій та через той факт, що після нормалізації модуля та виконання зсуву модуля на $n - k$ розрядів (цей зсув виконується шляхом відповідного з'єднання шин та не потребує тактів процесора) значення нормалізованого модуля буде більшим за значення діленого за умови $n - \tilde{n} < k - \tilde{k},$ тобто коли різниця між розрядністю регістра модуля та фактичною розрядністю модуля буде більше ніж

різниця між розрядністю регістра діленого та фактичною розрядністю діленого.

Розглянемо ці способи докладніше. Для виконання нормалізації вхідних даних необхідно використати два лічильники. Після виконання нормалізації у першому лічильнику $C1$ буде знаходитись необхідна кількість ітерацій основної частини алгоритму, другий лічильник $C2$ буде містити необхідну кількість зсувів вправо результату після виконання ітераційного процесу. При цьому перший лічильник ініціалізується значенням $n-k+1$, а другий – нульовим значенням.

Згідно першого способу необхідно наступне.

1. Виконувати зсув модуля та діленого вліво на один розряд поки один з аргументів не стане нормалізованим. Під час паралельного зсуву діленого та модуля значення в першому лічильнику залишається незмінним, оскільки операнди збільшуються в однакову кількість разів, і це не вплине на остаточний результат. Водночас другий лічильник буде працювати в режимі інкременту.
2. Подальший процес залежить від того, який з операндів першим став нормалізованим:
    - якщо першим став нормалізованим модуль, то зсув модуля та інкремент лічильника $C2$ зупиняється, і далі виконується зсув вліво тільки діленого, при цьому лічильник $C1$ починає працювати в режимі декременту поки ділене не досягне нормалізованого стану;
    - якщо першим стало нормалізованим ділене, то зсув вліво діленого зупиняється, але другий лічильник продовжує працювати в режимі інкременту та перший лічильник починає працювати в режимі

інкременту, поки модуль не досягне нормалізованого стану.

Для виконання нормалізації необхідно $n-\tilde{n}+k-\tilde{k}$ тактів. Після нормалізації виконується основний ітераційний процес за $\tilde{n}-\tilde{k}+1$ ітерацій (одна ітерація виконується за один такт процесора) та фінальна корекція результату (зсув вправо результату) за $k-\tilde{k}$ такти. Отже загалом необхідно $n+2k-3\tilde{k}+1$ тактів.

Згідно другого способу необхідно:

1. Виконувати зсув модуля та діленого вліво на один розряд, поки один з аргументів не стане нормалізованим. Під час паралельного зсуву діленого та модуля значення в першому лічильнику залишається незмінним, оскільки операнди збільшуються в однакову кількість разів, і це не вплине на остаточний результат. Водночас другий лічильник буде працювати в режимі інкременту.
2. Подальший процес залежить від того, який з операндів першим став нормалізованим:
    - якщо першим став нормалізованим модуль, то на цьому етап нормалізації зупиняється;
    - якщо першим стало нормалізованим ділене, то зсув вліво діленого зупиняється, але другий лічильник продовжує працювати в режимі інкременту та перший лічильник починає працювати в режимі інкременту поки модуль не досягне нормалізованого стану.

Для виконання нормалізації необхідно $k-\tilde{k}$ тактів. Після нормалізації виконується основний ітераційний процес за $n-k+1+\delta\tau$ ітерацій (одна ітерація виконується за один такт процесора), де $\delta=0$ якщо $\tau\geq 0$ та $\delta=-1$ якщо $\tau<0$, причому $\tau=n-\tilde{n}-\left(k-\tilde{k}\right)$. Фінальна

корекція результату (зсув вправо результату) виконується за $k-\tilde{k}$ тактів. Отже, загалом необхідно $n+k-2\tilde{k}+1+\delta\tau$ тактів. Причому найбільша кількість тактів буде, коли $k-\tilde{k}>n-\tilde{n}$, тоді $\delta\tau=k-\tilde{k}-n-\tilde{n}$ та загальна кількість тактів дорівнює $\tilde{n}+2k-3\tilde{k}+1$, найменша кількість тактів буде коли $k-\tilde{k}\le n-\tilde{n}$, тоді $\delta\tau=0$ та загальна кількість тактів дорівнює $n+k-2\tilde{k}+1$.

Таким чином, згідно другого підходу до нормалізації аргументів:

якщо $k-\tilde{k}\ge n-\tilde{n}$, то кількість тактів дорівнює $\tilde{n}+2k-3\tilde{k}+1$;

якщо $k-\tilde{k}<n-\tilde{n}$, то кількість тактів дорівнює $n+k-2\tilde{k}+1$.

Порівняємо кількість тактів процесора необхідну для виконання редукції при застосуванні першого та другого підходу до нормалізації аргументів. Очевидним є те, що при виконанні нерівностей

$$\begin{cases} n\ge\tilde{n} \\ k\ge\tilde{k} \end{cases} \tag{2.5}$$

автоматично буде виконуватись нерівність

$$n+2k-3\tilde{k}+1\ge\max\left\{\tilde{n}+2k-3k+1;\, n+k-2\tilde{k}+1\right\}. \tag{2.6}$$

Нерівність (2.5) буде виконуватись завжди, оскільки неможливо проводити обчислення, якщо розрядність операндів перевищує розрядність регістрів для їх зберігання. Виконання нерівності (2.6) показує той факт, що при другому підході до нормалізації необхідна кількість тактів буде не більшою, ніж при першому підході. На основі цього твердження робимо висновок, що в середньому кращі часові показники буде давати другий підхід.

2.2. Узагальнений метод піднесення до степеня в полі *GF*(*p*) з поданням показника степеня в системі числення з мультиосновою

Подання показника степеня у системі числення з мультиосновою записується наступним чином [63]:

$$k = \sum_{i=1}^{l} s_i \prod_{j=1}^{n} d_j^{c_{ij}}, \tag{2.7}$$

де $d_j \in D$, c_{ij} – цілі додатні числа, $s_i \in S$. Допустимими елементами множини D є цілі додатні числа, а множини S цілі числа без обмеження на знак.

Залежно від значень параметрів для перетворення показника степеня у подання (2.7) можна застосовувати різні алгоритми. Наприклад, якщо $S = \{-1;1\}$ та $D = \{2,3\}$, то доцільно застосовувати «жадібний» алгоритм [45]. В той же час зі збільшенням потужності множини D з практичної точки зору стає недоцільно застосовувати цей алгоритм, тому виникає потреба у побудові узагальненого алгоритму, який дозволить, при потребі, легко варіювати параметрами. Окрім цього, загальним недоліком всіх алгоритмів, орієнтованих на маленьку потужність множини D, зазвичай $D = |2|$, є те, що вони не забезпечують пошук оптимального розкладання показника степеня.

Зазначені недоліки нівельовані у підході, що ґрунтується на побудові дерева. Згідно цього підходу для $S = \{-1;1\}$ та $D = \{2,3\}$ пропонується у базовому числі (число, яке підлягає розкладанню) виділити множники, що є степенями 2 та 3, а потім з отриманого результату знайти два числа шляхом додавання та віднімання 1 (два нащадки у дереві). Далі продовжити цей процес з кожним нащадком. Після отримання нащадка зі значенням 1, необхідно лише зібрати з дерева всі елементи розкладання. Таким чином можна отримати оптимальне розкладання.

Оскільки для великих чисел дерево, побудоване таким чином, містить багато елементів, то з метою зменшення обсягу оперативної пам'яті, що використовується, та з метою збільшення швидкодії на кожному рівні дерева обирається B найменших елементів для продовження обчислень. Зазвичай значення B приймається рівним 2.

На основі підходу, що ґрунтується на побудові дерева, можна побудувати метод подання показника степеня у системі числення з мультиосновою [58, 63].

1. Виділити у числі, що підлягає розгалуженню, всі множники, що належать множині D, тобто ділити дане число почергово на елементи множини D поки воно не стане взаємно простим з кожним елементом множини D.
2. Отримати $2 \cdot |S|$ нащадків шляхом додавання та віднімання елементів множини S.
3. На поточному рівні дерева обрати B найменших елементів для подальшого розгалуження. Виконати дії з п. 1-2 для кожного обраного елемента. Процес побудови дерева зупинити при отриманні елемента зі значенням 1.

Наприклад, для числа $k = 52638$ та параметрів $B = 2$, $S = \{1,3,5\}$, $D = \{2,3,5\}$ отримаємо дерево наведене на рис. 2.4.

Після отримання дерева розкладання показника степеня необхідно коректно зібрати елементи розкладення для запуску піднесення до степеня. Як зазначалося вище, розрізняють два різновиди алгоритмів піднесення до степеня: *LR* (*left-to-right*) та *RL* (*right-to-left*) алгоритми. Префікс *LR*- та *RL*- вказує в якому напрямку зліва направо чи справа наліво розглядати розкладання показника степеня.

Для нашого прикладу маємо такі послідовності для кожного з алгоритмів:

LR: $2^1 3^1 \cdot (2^6 \cdot (2^2 5^1 \cdot (2^3 - 1) - 3) + 5) = 52638;$

RL: $5 \cdot 2^1 3^1 - 3 \cdot 2^6 \cdot 2^1 3^1 - 1 \cdot 2^2 5^1 \cdot 2^6 \cdot 2^1 3^1 + 2^3 \cdot 2^2 5^1 \cdot 2^6 \cdot 2^1 3^1 = 52638.$

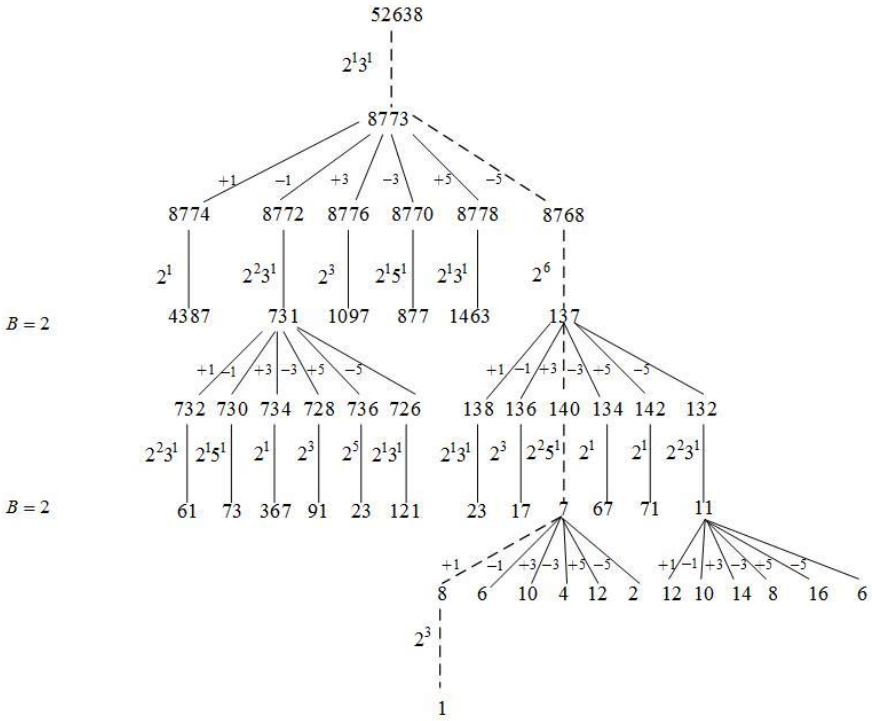


Рис. 2.4. Дерево розкладання числа $k = 52638$

Отже, узагальнений метод піднесення до степеня з поданням показника степеня у системі числення з мультиосновою полягає у виконанні наступних кроків [52, 63].

1. Побудувати дерево розкладання показника степеня згідно запропонованого методу.
2. Зібрати необхідні елементи розкладання. Для *LR*-алгоритму ці дії виконуються таким чином: дерево збирається знизу вверх та елементи множини S беруться з протилежним знаком. Для *RL*-алгоритму дерево збирається аналогічно зверху вниз.

3. Використовуючи дані отримані у п. 2 виконати множення згідно алгоритмів А.22 або А.23 піднесення до степеня (додаток А).

В алгоритмах А.22 та А.23 піднесення до степеня використовується список *decomposition* в якому зберігаються *DecompositionItem*, що складається з масиву *pows*, в якому зберігаються степені (c_{ij}) елементів множини D (d_j) та значення s ($i.s$) в якому зберігаються поточний елемент s_i множини S.

2.3. Модифікований метод піднесення до степеня елементів скінченного поля з ковзним вікном

У бінарних віконних методах часові витрати зменшуються завдяки побудові на першій стадії їх роботи таблиць передобчислень, що дозволяє скоротити загальну кількість множень під час виконання операції піднесення до степеня. Таблиці передобчислень будуються для значень, наведених у таблиці 2.1.

Розглянемо процес побудови модифікованого методу піднесення до степеня елементів скінченного поля з ковзним вікном на прикладі елементів поля $GF(p)$. Для розширення основного поля Галуа модифікований метод можна застосовувати замінивши елементи основного поля на елементи розширення поля Галуа.

Під час виконання бінарного методу піднесення до степеня показник степеня подається у двійковій системі числення та почергово аналізується один біт. Якщо черговий біт дорівнює нулю, то виконується тільки операція піднесення до квадрата, якщо одиниці, то виконується піднесення до квадрата та множення. При використанні віконних методів кількість піднесень до квадрата не скорочується і залишається такою самою як в

бінарному методі, а саме – дорівнює кількості бітів показника степеня. Віконні методи дозволяють скоротити кількість множень.

Таблиця 2.1

Наперед обчислені значення для віконних методів піднесення до степеня

Методи	Значення показника степеня k	Кількість наперед обчислених значень
Класичний віконний метод	$\{1, 2, ..., 2^w - 1\}$	$2^w - 1$
Метод з ковзним вікном	$\{2^{w-1}, 2^{w-1} + 1, ..., 2^w - 1\}$	2^{w-1}
Метод з поданням показника степеня у вигляді *wNAF*	$\{1, 3, 5, ..., 2^{w-1} - 1\}$	2^{w-2}
Метод з ковзним вікном та поданням показника степеня у вигляді *NAF*	$\left\{1, 3, ..., \frac{2\left(2^w - (-1)^w\right)}{3} - 1\right\}$	$\frac{1}{3}\left(2^w - (-1)^w\right)$

Розглянемо це докладніше.

Наприклад, якщо $w = 3$ для класичного віконного методу і при аналізі показника степеня зустрічається блок 101_2, це означає, що при реалізації класичного віконного методу буде виконано одне множення замість двох при реалізації бінарного методу. Скорочення кількості операцій множення дорівнює вазі Хеммінга бітового блоку показника степеня, що аналізується, мінус 1.

Для показників степеня з бітовою довжиною до $n = 25$ біт можна легко програмним шляхом підрахувати точне значення кількості скорочень множення при піднесенні до степеня за допомогою віконного методу. Алгоритм обчислення середнього значення кількості скорочень множення

при піднесенні до степеня за допомогою віконного методу порівняно з бінарним методом полягає у реалізації наступних кроків.

1. Бітове подання кожного числа, що має довжину менше або рівну n біт проаналізувати зліва направо (LR-алгоритм). Під час аналізу виділяти бітові блоки, що містяться в таблиці передобчислень та виконувати інкремент кількості блоків відповідного типу. Таким чином буде знайдено кількість бітових блоків кожного типу, що представлені у таблиці передобчислень, у всіх числах довжиною менше або рівній n біт.
2. Поділивши отримані статистичні значення на кількість чисел, що досліджувались, тобто на 2^n, отримаємо середню кількість разів, яку відповідний блок зустрічається в одному слові довжиною не більше n біт.
3. Помноживши отримані середні значення на коефіцієнт, що дорівнює кількості множень, на яку відбувається скорочення (для класичного віконного алгоритма – це вага Хеммінга бітового блоку мінус 1), коли зустрічається відповідний бітовий блок при аналізі показника степеня. Таким чином, буде отримано значення, які показують в середньому, на скільки операцій множення дозволяє скоротити кожен бітовий блок при застосуванні даного віконного алгоритму порівняно з бінарним методом для чисел довжиною не більше n біт.
4. Додавши отримані значення з третього пункту та віднявши кількість операцій, що необхідна для побудови таблиці передобчислень, отримаємо значення, що показує в середньому на скільки операцій множення відбувається зменшення при застосуванні даного віконного алгоритму порівняно з бінарним методом для чисел довжиною не більше n біт.

Розглянемо приклад. Нехай за допомогою класичного віконного алгоритму до степеня необхідно піднести число a за модулем m, при

цьому $w = 3$ та довжина показників степеня не перевищує $n = 7$ біт. Таблиця передобчислень буде містити значення a у степені 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2 за модулем m. Знайдемо середню кількість скорочень операцій множення порівняно з бінарним алгоритмом (табл. 2.2).

Таблиця 2.2

Аналіз класичного віконного методу (показник степеня довжиною 7 біт)

Показник степеня для якого збережене значення у таблиці передобчислень	Кількість разів, яку зустрічається відповідний бітовий блок при аналізі всіх чисел заданої довжини	Середня кількість бітових блоків відповідного типу, що зустрічається у одному числі заданої довжини	Середня кількість операцій множення, на яку виконується зменшення порівняно з бінарним алгоритмом, у одному числі заданої довжини
001_2	12	0.09375	0
010_2	12	0.09375	0
011_2	12	0.09375	0.09375
100_2	43	0.33594	0
101_2	43	0.33594	0.33594
110_2	43	0.33594	0.33594
111_2	43	0.33594	0.67188

З табл. 2.2 отримуємо (підсумуємо елементи останнього стовпця), що середнє зменшення кількості операцій множення в одному числі довжиною 7 біт при застосуванні класичного віконного алгоритму дорівнює 1.43751.

Порівняємо скорочення кількості операцій множення, що забезпечує класичний віконний метод, та віконний метод з ковзним вікном. З цією метою знайдемо різницю середнього зменшення кількості операцій множення класичного віконного методу та методу з ковзним вікном для різних значень n та різної кількості елементів у таблиці

передобчислень (табл. 2.3). Дослідження проводились на сформованому для кожного значення *n* наборі з 1000 випадкових чисел.

З табл. 2.3 бачимо, що при однаковій кількості елементів таблиці передобчислень кращі результати показує віконний метод з ковзним вікном. У третьому та четвертому методі з таблиці 2.1 досить обчислювально витратною є операція перетворення показника степеня у *NAF* або *wNAF*-вигляд та це спричинює погіршення часових характеристик. Таким чином, доцільно модифікувати класичний віконний метод та віконний метод з ковзним вікном, оскільки вони не вимагають попереднього перетворення показника степеня у спеціальну форму. У зв'язку з цим доцільно будувати віконний метод, що буде ґрунтуватись на двійковому поданні показника степеня.

Таблиця 2.3

Порівняння
класичного віконного методу та методу з ковзним вікном

		Кількість елементів таблиці передобчислень				
		3	7	31	63	127
Довжина числа, біт	32	0.7	3.0	13.4	15.7	17.1
	64	4.2	3.9	15.8	19.5	21.1
	128	11.9	6.6	19.5	25.3	34.0
	256	29.8	17.5	19.1	29.7	35.0
	384	43.9	31.2	21.7	31.4	38.2
	512	59.9	44.4	26	37.9	44.1
	1024	124.6	89.6	48.6	48.7	88.0
	2048	250.7	181.7	99.2	73.3	91.4
	4096	502.0	366.9	203.2	150.7	115.9
	8192	1027.8	743.5	415.1	357.6	257.8

З проведеного аналізу очевидним є те, що чим більше одиниць містить бітовий блок таблиці передобчислень, тим більше операцій множення він економить, коли зустрічається при аналізі показника степеня. Водночас важливим показником є те, скільки разів в середньому зустрічається кожен бітовий блок у числах заданої довжини, оскільки, наприклад, два блоки 1011 зекономлять чотири операції множення, а один блок 1111 – лише три операції множення.

Таким чином, доцільно побудувати віконний метод, в якому кожен бітовий блок, що представлений у таблиці передобчислень, буде містити більше однієї одиниці [27].

У зв'язку з цим доцільно побудувати віконний метод піднесення до степеня з вікном змінної довжини, який би відрізнявся від існуючих способом подання таблиць передобчислень, завдяки чому буде забезпечуватись приріст швидкодії [27].

Отже, на першій стадії запропонованого методу будуються три таблиці передобчислень *Table*1, *Table*2, *Table*3. В залежності від розрядів показника степеня k вибирається одна з цих таблиць. Потім виконується піднесення до степеня, що відповідає виділеним розрядам. Ці дії повторюються, поки не будуть розглянуті всі розряди числа k. Блок-схема запропонованого віконного методу піднесення до степеня з ковзним вікном зображена на рис. 2.5.

LR-алгоритми запропонованої модифікації віконного методу з різними таблицями передобчислень

Для запропонованого віконного методу піднесення до степеня з вікном змінної довжини побудовано три *LR*-алгоритми його реалізації (*RL*-алгоритми не розглядалися, оскільки при аналітичному аналізі було встановлено, що *LR*-алгоритми реалізації існуючих віконних методів дають кращі показники швидкодії ніж *RL*-алгоритми).

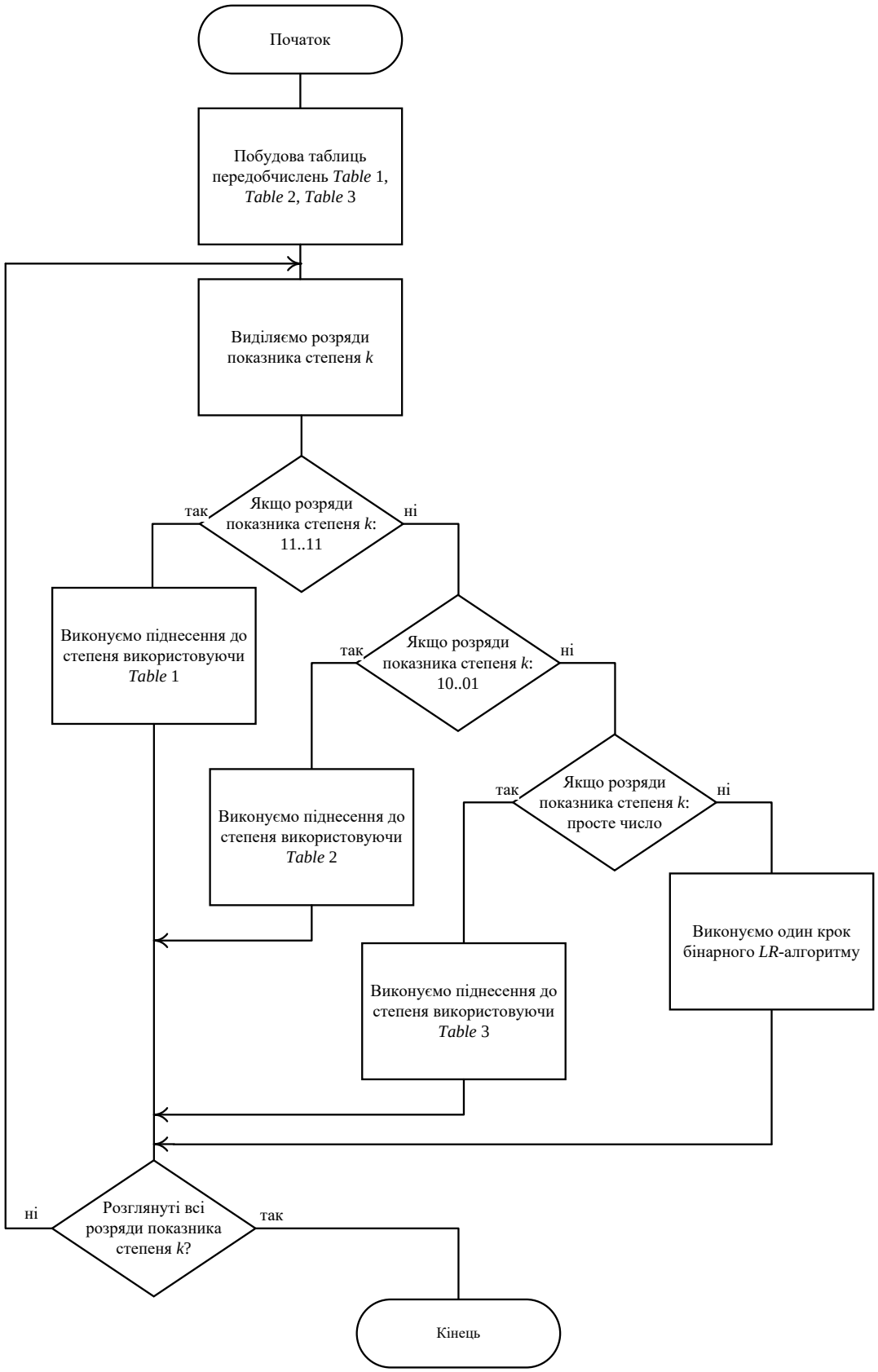


Рис. 2.5. Запропонована модифікація методу піднесення до степеня з ковзним вікном змінної довжини

У табл. 2.4 наведені способи побудови таблиць передобчислень для двох *LR*-алгоритмів реалізації запропонованої модифікації віконного методу. Кожна з них в деякому частковому випадку буде збільшувати швидкодію алгоритму реалізації віконного методу піднесення до степеня.

Таблиця 2.4

Наперед обчислені значення для *LR*-алгоритмів реалізації запропонованої модифікації методу піднесення до степеня з ковзним вікном

Алгоритм	Значення	Кількість наперед обчислених значень
№1	$2^i - 1$, де $i \in [1; w]$	w
№2	1 та $2^{i-1} + 1$, де $i \in [2; w]$	w

Статистично показано, що двійкові подання чисел довжиною понад 100 біт містять довгі послідовності одиниць, тому запропоновано алгоритм реалізації віконного методу, згідно якого буде виділятися вікно з одиничними бітами, довжина вікна буде варіюватись в межах від 1 до w біт. Даний алгоритм назвемо алгоритмом №1. Показник степеня k таблиці передобчислень в даному алгоритмі буде рівним $2^i - 1$, де $i = 1, 2, .., w$.

Алгоритм 2.1. *LR*-алгоритм №1 реалізації модифікованого методу піднесення до степеня з ковзним вікном

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. *for* $i = 1$ *to* w

   1.1. $Table[i] \leftarrow a^{2^i - 1} \pmod m$

2. *end for*

3. $b \leftarrow 1; i \leftarrow \lceil \log_2 k \rceil$

4. *while* $i > 0$ *do*

4.1. $b = b^2 \pmod m$

4.2. *if* $k_i = 1$ *then*

4.2.1. знайти $\max(t) \le w$ таке, що $k_{i-1} = k_{i-2} = \ldots = k_{i-t+1} = 1$

4.2.2. $b \leftarrow b^{2^{t-1}} \pmod m$

4.2.3. $b \leftarrow b \cdot Table[t] \pmod m$

4.2.4. $i \leftarrow i - t + 1$

4.3. $i \leftarrow i - 1$

5. *end while*

6. *return b*

Приклад роботи *LR*-алгоритму №1 для $a = 5$, $k = 239_{(10)}$ та $w = 4$.

1. Показник степеня *k* в двійковому вигляді: $k = \underbrace{1110}_{7}\,\underbrace{1111}_{15}$.

2. Таблиця передобчислень для $w = 4$ матиме вигляд:

№ елементу	Показник степеня	Значення
1	$2^1 - 1$	$5^{1_2} = 5^1$
2	$2^2 - 1$	$5^{11_2} = 5^3$
3	$2^3 - 1$	$5^{111_2} = 5^7$
4	$2^4 - 1$	$5^{1111_2} = 5^{15}$

3. Ініціалізація: $b = 1,\ i = 8$.

4. Поки $i > 0$ виконуються наступні кроки:

$$k_8 = 1 \quad \Rightarrow \quad \begin{aligned} & b = b^2 = 1^2 = 1 \\ & \max(t) = 3 \\ & b = b^{2^{t-1}} = 1^{2^2} = 1 \\ & b = b \cdot Table[t] = b \cdot Table[3] = 1 \cdot 5^7 = 5^7 \\ & i = i - t = 8 - 3 + 1 = 6 \\ & i = i - 1 = 6 - 1 = 5 \end{aligned}$$

$$k_5 = 0 \quad \Rightarrow \quad b = b^2 = \left(5^7\right)^2 = 5^{14}$$
$$i = i - 1 = 5 - 1 = 4$$

$$b = b^2 = \left(5^{14}\right)^2 = 5^{28}$$
$$\max(t) = 4$$
$$k_4 = 1 \quad \Rightarrow \quad b = b^{2^{t-1}} = \left(5^{28}\right)^{2^3} = \left(5^{28}\right)^8 = 5^{224}$$
$$b = b \cdot Table[t] = b \cdot Table[4] = 5^{15} \cdot 5^{224} = 5^{239}$$
$$i = i - t + 1 = 4 - 4 + 1 = 1$$
$$i = i - 1 = 1 - 1 = 0$$

5. Результат піднесення до степеня знаходиться у змінній $b = 5^{290}$.

Побудуємо алгоритм в якому з розрядів числа *k* виділяється частина біт, що починається і закінчується одиницею, а між ними міститиметься певна кількість нулів (алгоритм №2). Показник степеня *k* у таблиці передобчислень в алгоритмі №2 буде рівним $2^{i-1}+1$, де $i = 2,3,..,w$.

Алгоритм 2.2. *LR*-алгоритм №2 реалізації модифікованого методу піднесення до степеня з ковзним вікном

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $Table[1] \leftarrow a$
2. *for* $i = 2$ *to* w
   2.1. $Table[i] \leftarrow a^{2^{i-1}+1} \pmod m$
3. *end for*
4. $b \leftarrow 1; i \leftarrow \lceil \log_2 k \rceil$
5. *while* $i > 0$ *do*
   5.1. $b = b^2 \pmod m$
   5.2. *if* $k_i = 1$ *then*

5.2.1. *if* $k_{i-1} = 0$ *then*

5.2.1.1. знайти $\max(t) \le w$ таке,

що $k_i = k_{i-t+1} = 1$ та $k_{i-1} = k_{i-2} = ... = k_{i-t+2} = 0$

якщо такого t не існує, то $t \leftarrow 0$

5.2.1.2. *if* $t \neq 0$ *then*

a) $b \leftarrow b^{2^{t-1}} \pmod m$

b) $b \leftarrow b \cdot Table[t] \pmod m$

c) $i \leftarrow i - t + 1$

5.2.1.3. *else*

a) $b \leftarrow b \cdot Table[1] \pmod m$

5.2.2. *else*

a) $b = b^2 \pmod m$

b) $b \leftarrow b \cdot Table[2] \pmod m$

c) $i \leftarrow i - 1$

5.3. $i \leftarrow i - 1$

6. *end while*

7. *return b*

Оскільки за допомогою виразу $2^{i-1} + 1$ неможливо отримати 1, то до початку циклу побудови таблиці передобчислень у елемент таблиці з індексом 1, необхідно записати значення основи *a*.

Приклад роботи *LR*-алгоритму №2 для $a = 5$, $k = 567_{(10)}$ та $w = 5$.

1. Показник степеня *k* в двійковому вигляді: $k = \underbrace{1000}_{17}\underbrace{11}_{5}\underbrace{011}_{3}$.
2. Таблиця передобчислень для $w = 5$ матиме вигляд:

№ елементу	Показник степеня	Значення
1	1	$5^{1_2} = 5^1$
2	$2^1 + 1$	$5^{11_2} = 5^3$
3	$2^2 + 1$	$5^{101_2} = 5^5$
4	$2^3 + 1$	$5^{1001_2} = 5^9$
5	$2^4 + 1$	$5^{10001_2} = 5^{17}$

3. Ініціалізація: $b=1, i=10$.
4. Поки $i>0$ виконуються наступні кроки:

$$
\begin{array}{ll}
 & b=b^2=1^2=1 \\
i=10 & \max(t)=5 \\
k_{10}=1 \Rightarrow & b=b^{2^{t-1}}=1^{2^4}=1 \\
k_9=0 & b=b\cdot Table[t]=b\cdot Table[5]=1\cdot 5^{17}=5^{17} \\
 & i=i-t+1=10-5+1=6 \\
 & i=i-1=6-1=5
\end{array}
$$

$$
\begin{array}{ll}
 & b=b^2=\left(5^{17}\right)^2=5^{34} \\
i=5 & \max(t)=3 \\
k_5=1 \Rightarrow & b=b^{2^{t-1}}=\left(5^{34}\right)^{2^2}=\left(5^{34}\right)^4=5^{136} \\
k_4=0 & b=b\cdot Table[t]=b\cdot Table[3]=5^{136}\cdot 5^5=5^{141} \\
 & i=i-t+1=5-3+1=3 \\
 & i=i-1=3-1=2
\end{array}
$$

$$
\begin{array}{ll}
 & b=b^2=\left(5^{141}\right)^2=5^{282} \\
i=2 & \\
k_2=1 \Rightarrow & b=b^2=\left(5^{282}\right)^2=5^{564} \\
k_1=1 & b=b\cdot Table[2]=5^{564}\cdot 5^3=5^{567} \\
 & i=i-1=2-1=1 \\
 & i=i-1=1-1=0
\end{array}
$$

5. Результат піднесення до степеня знаходиться у змінній $b=5^{567}$.

Узагальнений LR-алгоритм запропонованої модифікації віконного методу

Алгоритми № 2.1 - 2.2 дають приріст швидкодії тільки в окремих випадках, тому актуальним є побудувати узагальнений алгоритм реалізації запропонованого віконного методу піднесення до степеня з ковзним вікном.

Алгоритм 2.3. Узагальнений *LR*-алгоритм реалізації модифікованого методу піднесення до степеня з ковзним вікном

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $Table1[1] \leftarrow a,\ Table2[1] \leftarrow a$
2. *for* $i = 1$ *to* $w - 1$ *do*
   - 2.1. $Table1[i+1] = a^{2^{i+1}-1} \pmod m$
   - 2.2. $Table2[i+1] = a^{2^i+1} \pmod m$
3. *end for*
4. $b \leftarrow 1, i \leftarrow \lceil \log_2 k \rceil$
5. *while* $i > 0$ *do*
   - 5.1. *if* $k_i = 1$ *then*
     - 5.1.1. *if* $k_{i-1} = 0$ *then*
       - 5.1.1.1. *find* $\max(t) \le w$ *such that* $k_i = k_{i-t+1} = 1$ *and* $k_{i-1} = k_{i-2} = ... = k_{i-t+2} = 0$ якщо такого t не існує, то $t \leftarrow 0$
       - 5.1.1.2. *if* $t \neq 0$ *then*
         - *a*) $b \leftarrow b^{2^t} \pmod m$
         - *b*) $b \leftarrow b \cdot Table2[t] \pmod m$
         - *c*) $i \leftarrow i - t$
       - 5.1.1.3. *else*
         - *a*) $b \leftarrow b^2 \pmod m$
         - *b*) $b \leftarrow b \cdot Table2[1] \pmod m$
         - *c*) $i \leftarrow i - 1$
     - 5.1.2. *else*
       - 5.1.2.1. *find* $\max(t) \le w$ *such that* $k_{i-1} = k_{i-2} = ... = k_{i-t+1} = 1$
       - 5.1.2.2. $b \leftarrow b^{2^t} \pmod m$
       - 5.1.2.3. $b \leftarrow b \cdot Table1[t] \pmod m$
       - 5.1.2.4. $i \leftarrow i - t$
   - 5.2. *else*
     - 5.2.1. $b = b^2 \pmod m$
     - 5.2.2. $i = i - 1$
6. *end while*
7. *return b*

При реалізації кроку 5.1.1 необхідно передбачити обробку ситуації коли $i=1$, тобто k_i-им є молодший біт показника степеня, і тоді k_{i-1}-го біта не існує. В такому випадку необхідно прийняти k_{i-1}-ий біт рівним нулю, а функція $\max(t)$ на кроці 5.1.1.1 має повернути значення t рівне нулю. На кроках 5.1.1.2.*a* та 5.1.2.2 необхідно виконати *t* разів піднесення до квадрата за модулем *m* та недоцільно викликати загальні методи піднесення до степеня.

Приклад роботи узагальненого *LR*-алгоритму для $a=5$, $k=637_{10}$ та $w=4.$

1. Показник степеня *k* в двійковому вигляді: $k=\underbrace{1001}_{9}\underbrace{1111}_{15}01.$
2. Таблиці передобчислень *Table1*, *Table2* для $w=4$:

*Table*1

№ елементу	Значення
1	$5^{1_2}=5^1$
2	$5^{11_2}=5^3$
3	$5^{111_2}=5^7$
4	$5^{1111_2}=5^{15}$

*Table*2

№ елементу	Значення
1	$5^{1_2}=5^1$
2	$5^{11_2}=5^3$
3	$5^{101_2}=5^5$
4	$5^{1001_2}=5^9$

3. Ініціалізація: $b=1, i=10$.
4. Поки $i>0$ виконуються наступні кроки:

$$
\begin{array}{ll}
i=10 & \max(t)=4 \\
k_{10}=1 \Rightarrow & b=b^{2^t}=(1)^{2^4}=1 \\
k_9=0 & b=b\cdot Table2[t]=b\cdot Table2[4]=1\cdot 5^9=5^9 \\
 & i=i-t=10-4=6
\end{array}
$$

$$
\begin{array}{l}
i = 6 \\
k_6 = 1 \\
k_5 = 1
\end{array}
\quad
\begin{array}{l}
\max(t) = 4 \\
\Rightarrow\ b = b^{2^t} = \left(5^9\right)^{2^4} = \left(5^9\right)^{16} = 5^{144} \\
b = b \cdot Table1[t] = b \cdot Table1[4] = 5^{144} \cdot 5^{15} = 5^{159} \\
i = i - t = 6 - 4 = 2
\end{array}
$$

$$
\begin{array}{l}
i = 2 \\
k_2 = 0
\end{array}
\quad
\begin{array}{l}
\Rightarrow\ b = b^2 = \left(5^{159}\right)^2 = 5^{318} \\
i = i - 1 = 2 - 1 = 1
\end{array}
$$

$$
\begin{array}{l}
i = 1 \\
k_1 = 1
\end{array}
\quad
\begin{array}{l}
\max(t) = 0 \\
\Rightarrow\ b = b^2 = \left(5^{318}\right)^2 = 5^{636} \\
b = b \cdot Table1[1] = 5^{636} \cdot 5^1 = 5^{637} \\
i = i - 1 = 1 - 1 = 0
\end{array}
$$

5. Результат піднесення до степеня знаходиться у змінній $b = 5^{637}$.

Перевагою узагальненого алгоритму реалізації запропонованої модифікації віконного методу є те що при достатній довжині вікна він покриває всі можливі ненульові блоки у двійковому поданні показника степеня. Окрім цього, даний алгоритм характеризується значно меншим використанням оперативної пам’яті порівняно з існуючими віконними алгоритмами.

Ще більшого приросту швидкодії можна досягти, якщо додати до таблиці передобчислень запропонованої модифікації методу піднесення до степеня показники степеня, що є простими числами. Прості числа характеризуються більшою вагою Хеммінга, ніж числа, що належать рівномірному закону розподілу, за рахунок цього можна досягти більшого скорочення кількості операцій множення.

2.4. Висновки до розділу 2

1. Запропоновано модифіковану схему цілочисельного ділення, яка працює швидше за класичну на $n+2s+3\tilde{k}-3\tilde{n}-3$ тактів, коли виконується умова $n+2s+3\tilde{k}-3\tilde{n}-3>0$, де n – розрядність регістра діленого, k – розрядність регістра модуля, $\tilde{n}$ – кількість двійкових розрядів діленого, $\tilde{k}$ – кількість двійкових розрядів модуля, мінімально можливу різницю бітової довжини операндів, s – значення, меншою за яке не може бути різниця бітової довжини операндів.
2. Розроблено узагальнений метод піднесення до степеня у системі числення з мультиосновою, який дає можливість виконувати піднесення до степеня з використанням адитивних ланцюжків, що мають довжину меншу, ніж генерує віконний метод. Реалізовані *RL*- та *LR*- алгоритми даного методу піднесення до степеня.
3. Запропоновано модифікацію методу піднесення до степеня з ковзним вікном, яка відрізняється від існуючого методу способом побудови таблиць передобчислень. Побудовані два *LR*-алгоритми реалізації розробленої модифікації віконного методу з використанням різних таблиць передобчислень та узагальнений *LR*-алгоритм, що полягає у комбінації на початковій стадії роботи алгоритму трьох таблиць передобчислень, а саме таблиць передобчислень з алгоритмів №1, №2 та таблиці, що містить показники степеня, які є простими числами. Обчислювальні експерименти показують, що такий підхід до формування таблиці передобчислень дозволяє скоротити кількість операцій множення в середньому на 10%, коли показник степеня має довжину понад 256 біт, та забезпечує приріст швидкодії на 7-9 %.

РОЗДІЛ 3
ВИКОНАННЯ ОПЕРАЦІЙ В СКІНЧЕННИХ ПОЛЯ ВИДУ $GF(2^m)$ З ВИКОРИСТАННЯМ ТАБЛИЧНОГО ЗБЕРІГАННЯ ЕЛЕМЕНТІВ ПОЛЯ

У різних сферах застосування теорії скінченних полів виникає необхідність виконувати такі операції над елементами поля $GF\left(2^m\right)$: додавання елементів, знаходження адитивно оберненого (протилежного) елемента, віднімання елементів, множення елементів, знаходження мультиплікативно оберненого елемента, ділення елементів, піднесення до степеня елемента, де m – степінь незвідного многочлена $P_m(x)$.

На основі перелічених вище операцій можна реалізовувати інші – складніші операції в полі $GF\left(2^m\right)$.

3.1. Форми подання елементів поля $GF(2^m)$

Існують три форми [22, 59] подання елементів поля $GF\left(2^m\right)$: степеневе (у вигляді степеня примітивного елемента α поля з невід'ємним показником та у вигляді степеня примітивного елемента α поля з від'ємним показником), многочленне, числове.

Примітивним елементом α поля $GF\left(2^m\right)$ є такий елемент поля, при піднесенні якого до степенів від 0 до 2^m-2 буде отримано всі елементи цього поля.

Розглянемо способи отримання різних форм подання елементів поля $GF\left(2^4\right)$ за модулем незвідного многочлена четвертого $(m=4)$

степеня $P_4(x) = x^4 + x + 1$:

- Степеневе подання (у вигляді показника степеня примітивного елемента α поля):
  - ✓ у вигляді степеня примітивного елемента α поля з невід'ємним показником (нульовий елемент поля неможливо подати у вигляді степеня): α^0, α^1, …, α^{14}, $\alpha^0 = 1$, $\alpha^{15} = \alpha^0 = 1$;
  - ✓ у вигляді степеня примітивного елемента α поля з від'ємним показником (нульовий елемент поля неможливо подати у вигляді степеня): α^{-15}, α^{-14}, …, α^{-1}, де $\alpha^{-15} = \alpha^0 = 1$.
- Многочленне подання (у вигляді многочлена від α).

  Для отримання многочленного подання елемента поля необхідно взяти його степеневе подання (у вигляді степеня примітивного елемента α поля з невід'ємним показником) та поділити на незвідний многочлен $P_4(x)$ степеня $m = 4$. Отримана остача й буде многочленним поданням цього елемента поля.

  Наприклад:

$$\begin{array}{ll} \alpha^5 & \mid \underline{\alpha^4 + \alpha + 1} \\ \underline{\alpha^5 + \alpha^2 + \alpha} & \alpha \\ \boldsymbol{\alpha^2 + \alpha} & \end{array}$$

- Числове подання елементів поля отримують шляхом переписування коефіцієнтів многочленного подання у відповідні розряди двійкової системи числення: коефіцієнт при α^0 записують у розряд з вагою 2^0, коефіцієнт при α^1 записують у розряд з вагою 2^1 і так далі. Таким чином, отримуємо числове подання у двійковому вигляді. Щоб

отримати числове подання елементів поля у десятковому вигляді необхідно перетворити отримані числа з двійкової системи числення у десяткову.

Зведемо різні форми подання елементів поля $GF\left(2^4\right)$ до табл. 3.1.

Таблиця 3.1

Подання елементів поля $GF\left(2^4\right)$ за модулем незвідного многочлена $P_4(x)=x^4+x+1$

Степеневе подання		Многочленне подання (у вигляді многочлена від α)	Числове подання	
У вигляді степеня примітивного елемента α поля з невід'ємним показником	У вигляді степеня примітивного елемента α поля з від'ємним показником		Двійкове	Десяткове
–	–	0	0000	0
α^0	α^{-15}	1	0001	1
α^1	α^{-14}	α	0010	2
α^2	α^{-13}	α^2	0100	4
α^3	α^{-12}	α^3	1000	8
α^4	α^{-11}	$\alpha+1$	0011	3
α^5	α^{-10}	$\alpha^2+\alpha$	0110	6
α^6	α^{-9}	$\alpha^3+\alpha^2$	1100	12
α^7	α^{-8}	$\alpha^3+\alpha+1$	1011	11
α^8	α^{-7}	α^2+1	0101	5
α^9	α^{-6}	$\alpha^3+\alpha$	1010	10
α^{10}	α^{-5}	$\alpha^2+\alpha+1$	0111	7
α^{11}	α^{-4}	$\alpha^3+\alpha^2+\alpha$	1110	14
α^{12}	α^{-3}	$\alpha^3+\alpha^2+\alpha+1$	1111	15
α^{13}	α^{-2}	$\alpha^3+\alpha^2+1$	1101	13
α^{14}	α^{-1}	α^3+1	1001	9

Зрозуміло, що всі три способи подання елементів поля $GF\left(2^m\right)$ є тотожними. Якщо елементи поля $GF\left(2^m\right)$ необхідно зберігати в пам'яті, то для заданого елемента поля при степеневому поданні елементів поля в пам'яті достатньо зберігати лише показник степеня примітивного елемента α поля (невід'ємний або від'ємний), а при числовому поданні – двійковий код елемента поля (табл. 3.2).

Таблиця 3.2

Зберігання елементів поля $GF\left(2^4\right)$ в пам'яті ЕОМ (як незвідний вибрано многочлен $P_4\left(x\right)=x^4+x+1$)

Елемент поля	При степеневому поданні		При числовому поданні (двійковий код)
	Невід'ємний показник степеня примітивного елемента α поля	Від'ємний показник степеня примітивного елемента α поля	
0	–	–	0000
1	0	–15	0001
2	1	–14	0010
4	2	–13	0100
8	3	–12	1000
3	4	–11	0011
6	5	–10	0110
12	6	–9	1100
11	7	–8	1011
5	8	–7	0101
10	9	–6	1010
7	10	–5	0111
14	11	–4	1110
15	12	–3	1111
13	13	–2	1101
9	14	–1	1001

3.2. Аналіз операцій над елементами поля $GF(2^m)$

Проаналізуємо перелічені операції над елементами поля $GF\left(2^m\right)$.

Операцію додавання елементів поля та знаходження адитивно оберненого (протилежного) елемента зручно виконувати над числовим поданням у двійковому вигляді. При цьому виконується порозрядне (без переносу) підсумовування двійкових кодів елементів за модулем два, тобто операція XOR.

Оскільки в полі $GF\left(2^m\right)$ операції додавання та віднімання є тотожними (знак "–" замінюють на знак "+"), то $-b = b$ (b – елемент поля $GF\left(2^m\right)$) – протилежний елемент співпадає з прямим елементом.

Операцію множення, знаходження мультиплікативно оберненого елемента, ділення та піднесення до степеня зручно виконувати над степеневим поданням елементів поля, оскільки в цьому випадку необхідно виконувати мікрооперації лише над показниками степеня.

Розглянемо операцію множення елементів поля $GF\left(2^m\right)$.

Наприклад, необхідно помножити два елементи поля $GF\left(2^4\right)$: елемент α^7 на елемент α^{13}. За правилами множення маємо: $\alpha^7 \cdot \alpha^{13} = \alpha^{7+13} = \alpha^{20}$. Оскільки максимально можливий показник степеня примітивного елемента в полі $GF\left(2^4\right)$ дорівнює 14, то отриманий показник степеня необхідно взяти за модулем 15 (в загальному випадку – за модулем $2^m - 1$): $\alpha^{20 \bmod 15} = \alpha^5$.

У загальному випадку множення виконують так: $\alpha^t \cdot \alpha^d = \alpha^{(t+d) \bmod \left(2^m - 1\right)}$.

Тобто виникає необхідність виконання додавання величин за модулем $2^m - 1$.

Розглянемо операцію знаходження мультиплікативно оберненого елемента поля $GF\left(2^m\right)$.

Для обчислення мультиплікативно оберненого елемента поля необхідно показник степеня заданого елемента поля помножити на -1.

Нехай необхідно знайти мультиплікативно обернений елемент до елемента α^7 поля $GF\left(2^4\right)$. Таким елементом буде $\left(\alpha^7\right)^{-1} = \alpha^{-7}$ або у вигляді невід'ємного показника степеня $\alpha^{-7} = \alpha^{15-7} = \alpha^8$ (див. табл. 3.1). У загальному випадку маємо: $\left(\alpha^s\right)^{-1} = \alpha^{-s} = \alpha^{2^m-1-s} = \alpha^{s_{\text{інв}}}$, де $s_{\text{інв}}$ – інверсне значення величини s.

Отже, для знаходження мультиплікативно оберненого елемента до елемента α^s поля необхідно проінвертувати показник степеня s:

$$\left(\alpha^s\right)^{-1} = \alpha^{s_{\text{інв}}}. \tag{3.1}$$

Розглянемо операцію ділення елементів поля $GF\left(2^m\right)$.

Очевидним є те, що при виконанні операції ділення елементів поля $GF\left(2^m\right)$ виникає необхідність виконання віднімання величин за модулем $2^m - 1$:

$$\alpha^w : \alpha^q = \alpha^{(w-q) \bmod \left(2^m-1\right)}. \tag{3.2}$$

Наприклад,

$$\alpha^2 : \alpha^9 = \alpha^{(2-9) \bmod 15} = \alpha^{(-7) \bmod 15} = \alpha^8 \quad \text{(див. табл. 3.1).}$$

Операція піднесення до степеня елементів поля $GF\left(2^m\right)$ реалізується за формулою $\left(\alpha^t\right)^r = \alpha^{(t \cdot r) \bmod \left(2^m-1\right)}$. Тому виникає необхідність у реалізації мікрооперації множення величин за модулем $2^m - 1$.

Узагальнюючи розглянуті приклади можна зробити висновок, що при виконанні операцій над ненульовими елементами поля $GF\left(2^m\right)$ необхідно виконувати зведення показника степеня результату за модулем 2^m-1. Таким чином, виникає необхідність виконувати такі мікрооперації за модулем 2^m-1: додавання, віднімання, множення показників степеня, доповнення показника степеня до 2^m-1 (інвертування).

Таблиця 3.3

Мікрооперації для реалізації операцій над елементами поля $GF(2^m)$

№ п/п	Мікрооперація	Для яких операцій над елементами поля застосовується
1.	Додавання величин за модулем 2^m-1	Множення елементів поля
2.	Віднімання величин за модулем 2^m-1	Ділення елементів поля
3.	Множення величин за модулем 2^m-1	Піднесення елемента поля до степеня
4.	Інвертування величин	Знаходження мультиплікативно оберненого елемента поля

Як бачимо, степеневе подання елементів поля необхідне для реалізації операцій множення, ділення, піднесення до степеня елементів поля, а також для знаходження мультиплікативно оберненого елемента поля.

Однак, степеневе подання елементів поля $GF\left(2^m\right)$ має істотний недолік: воно не дозволяє отримувати нульовий елемент поля. Це означає, що при степеневому поданні неможливо виконати такі операції, як: $b+b=0$ (додавання однакових елементів поля); $b-b=0$ (віднімання однакових елементів поля), а також $0\cdot b$, $0:b$, 0^b, $0\pm b$, тобто коли один з

операндів дорівнює нулю, оскільки результатом цих операцій є нуль (нульовий елемент поля), а подати його у вигляді степеня неможливо.

У загальному випадку, степеневе подання не можна застосовувати при виконанні довільних операцій, оскільки, степеневе подання дає (відтворює) лише ненульові елементи поля.

Компромісом, очевидно, є поєднання числового (многочленного) подання та степеневого подання. Таке поєднання характеризується універсальністю – забезпечується виконання будь-яких операцій *над всіма*, без винятку елементами поля, та дозволяє істотно знизити обчислювальну складність реалізації операцій множення, ділення, піднесення до степеня та знаходження мультиплікативно оберненого елемента.

Найбільш простим способом реалізації зазначених перетворень є табличний спосіб.

Розглянемо перетворення числового подання елементів поля у степеневе (рис. 3.1).

В комірках ПЗП при цьому необхідно зберігати значення невід’ємних показників степеня примітивного елемента поля, а адресами комірок пам’яті мають бути числові подання елементів поля [34].

Таким чином, для перетворення числового подання елемента поля у його степеневе подання необхідно з ПЗП прочитати слово записане за адресою, яка є двійковим кодом числового подання елемента поля. Зчитане слово буде невід’ємним показником степеня примітивного елемента поля.

Перетворення елементів поля зі степеневого подання у числове можна також реалізувати таблично (рис. 3.2). В цьому випадку значення невід’ємного показника степеня примітивного елемента буде використовуватись як адреса, за якою в ПЗП буде розміщено значення елемента поля у числовому поданні.

Для перетворення елемента поля, поданого у вигляді від’ємного показника степеня примітивного елемента, в числове подання необхідно

значущі розряди показника степеня проінвертувати та використати їх як адресу ПЗП.

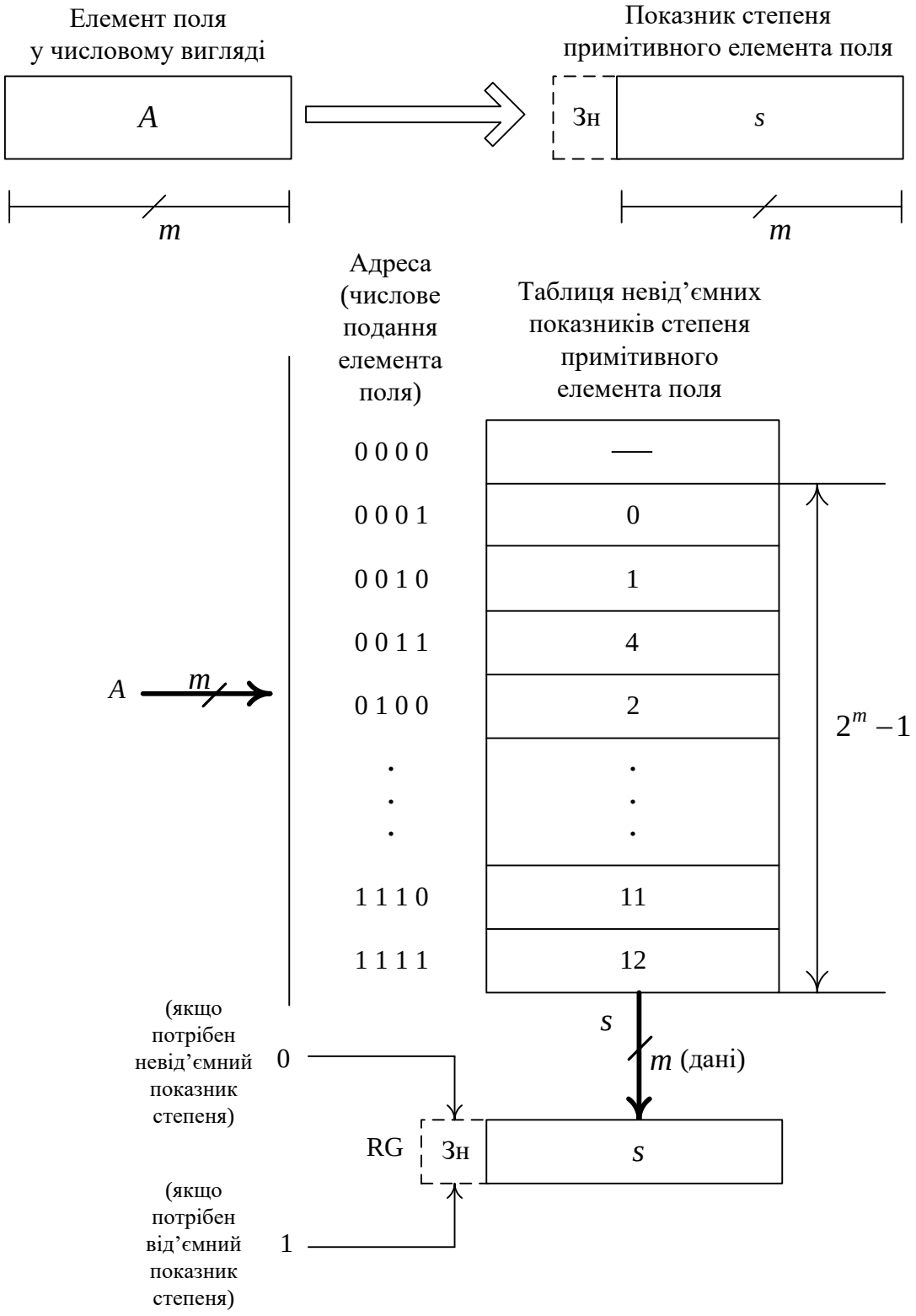


Рис. 3.1. Перетворення числового подання елемента поля у степеневе ($A \to s$ або $A \to -s$)

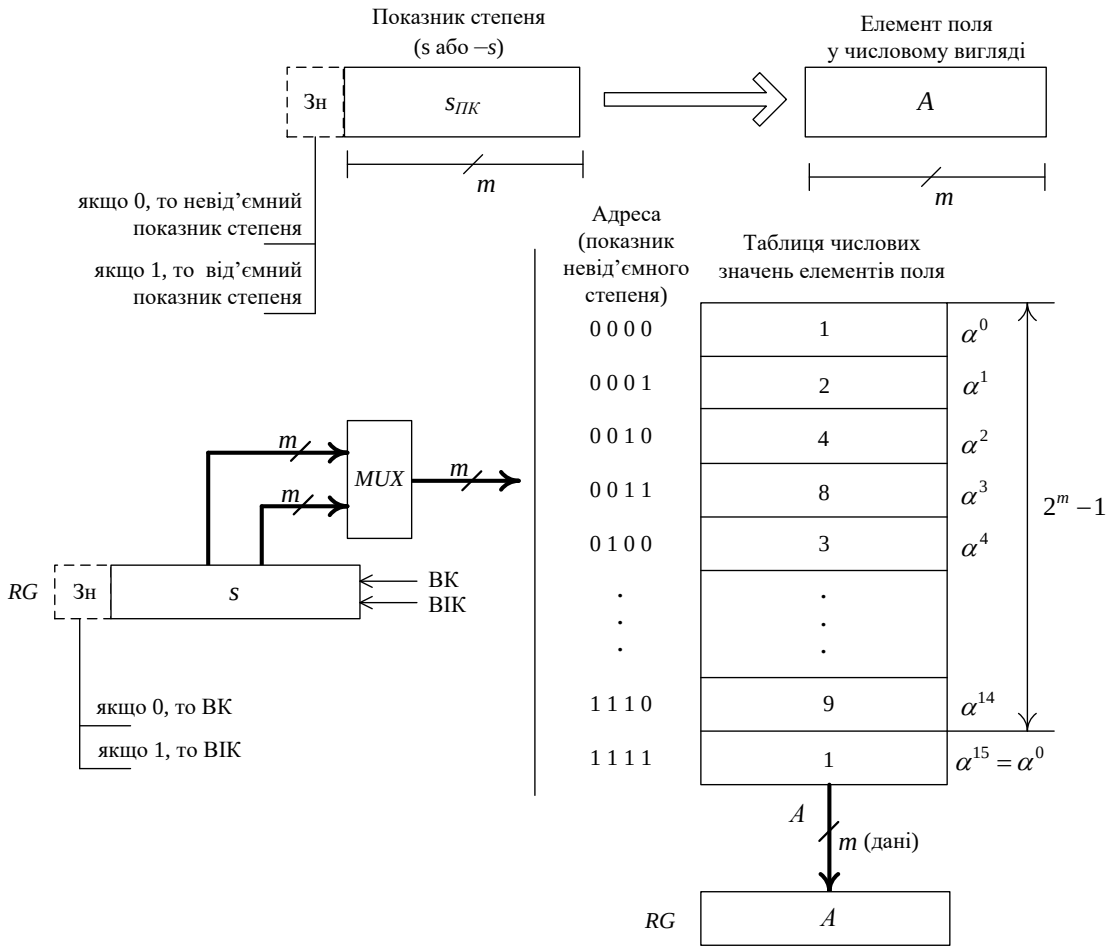


Рис. 3.2. Перетворення елемента поля зі степеневого подання у числове ($s \to A$ або $-s \to A$)

Таблиці на рис. 3.1 та рис. 3.2 можна звести в одну, що матиме таку структуру:

адреса	невід'ємний показник степеня примітивного елемента α поля	числове значення елемента поля

Така структура таблиці забезпечує перетворення як числового подання елементів поля у степеневе, так і степеневого у числове (рис. 3.3). Розмірність таблиці $2^m \times 2m$.

Якщо адресою є числове значення елемента поля, то звертання відбувається до відповідного рядка лівої частини таблиці (права частина таблиці при цьому не використовується). Результатом читання з таблиці є невід'ємний показник степеня примітивного елемента α поля.

Адреса	Невід'ємний показник степеня примітивного елемента α поля	Числове значення елемента поля
0 0 0 0	–	1 (α^0)
0 0 0 1	0	2 (α^1)
0 0 1 0	1	4 (α^2)
0 0 1 1	4	8 (α^3)
0 1 0 0	2	3 (α^4)
0 1 0 1	8	6 (α^5)
0 1 1 0	5	12 (α^6)
0 1 1 1	10	11 (α^7)
1 0 0 0	3	5 (α^8)
1 0 0 1	14	10 (α^9)
1 0 1 0	9	7 (α^{10})
1 0 1 1	7	14 (α^{11})
1 1 0 0	6	15 (α^{12})
1 1 0 1	13	13 (α^{13})
1 1 1 0	11	9 (α^{14})
1 1 1 1	12	1 (α^{15})

$2^m = 16$

m m

Рис. 3.3. Зведена таблиця для поля $GF(2^4)$ та незвідного многочлена $x^4 + x + 1$

Якщо адресою є невід'ємний показник степеня примітивного елемента поля, то звертання відбувається до відповідного рядка правої частини таблиці (ліва частина таблиці при цьому не використовується). Результатом читання з таблиці є числове значення елемента поля.

Отже, до чотирьох мікрооперацій, необхідних для реалізації операцій над елементами поля $GF\left(2^m\right)$, що зазначені в табл. 3.3 та операції *XOR*, необхідно додати ще дві операції: операцію перетворення числового подання елемента поля у степеневе, операцію перетворення степеневого подання елемента поля у числове.

На основі чотирьох мікрооперацій з табл. 3.3, операції *XOR* та операцій перетворення зі степеневого подання елемента поля у многочленне і навпаки, отже, загалом 4 мікрооперації та 3 операції, можна реалізовувати довільні операції над елементами поля $GF\left(2^m\right)$.

3.3. Реалізація мікрооперацій, необхідних для забезпечення операцій над елементами поля $GF(2^m)$

Аналіз можливих операцій над елементами поля $GF(2^m)$ показав, що для забезпечення мультиплікативних операцій, таких як множення, піднесення до степеня, обчислення мультиплікативно оберненого елемента та ділення необхідно чотири мікрооперації [22]. Адитивні операції над елементами поля $GF(2^m)$ реалізуються як операція *XOR* – з використанням 2-входових суматорів за модулем два, кількість яких дорівнює m [20]; перетворення зі степеневого подання елемента поля у многочленне і навпаки реалізуються як мікрооперації читання з ПЗП.

Докладніше розглянемо реалізацію чотирьох мікрооперацій з табл. 3.3 [22]. Цю множину мікрооперацій можна апаратно реалізувати, як систему мікрокоманд мікроасемблера (табл. 3.4).

Поставимо у відповідність кожній мікрооперації певне мнемонічне позначення та значення коду мікрооперації (КмОП).

Реалізація мікрооперації ADM – додавання m-розрядних двійкових величин за модулем $2^m - 1$

Математично мікрооперацію додавання за модулем $2^m - 1$ запишемо так [32]:

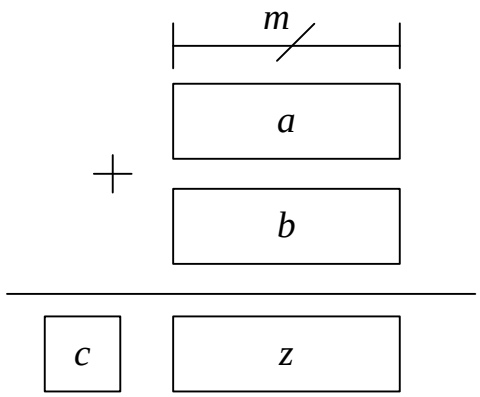
$$\left(a+b\right) \bmod \left(2^m - 1\right) = \begin{cases} a+b, & якщо\ a+b < 2^m - 1; \\ 0, & якщо\ a+b = 2^m - 1; \\ a+b-\left(2^m - 1\right), & якщо\ a+b > 2^m - 1. \end{cases} \quad (3.3)$$

Таблиця 3.4

Система команд Мікроасемблера, необхідних для виконання операцій над елементами поля $GF(2^m)$

КмОП	Мнемоніка	Опис мнемоніки	Призначення мікрооперації
0 0	**adm**	**ad**dition by **m**odulo	додавання m-розрядних двійкових величин за модулем 2^m-1
0 1	**sbm**	**s**u**b**traction by **m**odulo	віднімання m-розрядних двійкових величин за модулем 2^m-1
1 0	**mlm**	**m**u**l**tiplication by **m**odulo	множення m-розрядних двійкових величин за модулем 2^m-1
1 1	**cpl**	**c**om**pl**iment	інвертування m-розрядної двійкової величини

Нехай підсумовування a і b здійснюється за схемою:



де z – значущі розряди результату підсумовування двох операндів на інформаційних входах комбінаційного суматора,

c – значення переносу за межі старшого розряду результату.

Друга рівність в (3.3) означає, що коли $z=\underset{m}{11\ldots1}$, то коректним результатом має бути $z=\underbrace{00\ldots0}_{m}$, тому в цьому випадку отримане значення z потрібно проінвертувати або до отриманого значення z додати 1.

Третю рівність в (3.3) запишемо у вигляді:

$$a+b-\left(2^m-1\right)=a+b+1-2^m=z+1-2^m=z+1. \qquad (3.4)$$

Доданок -2^m не вливає на z, оскільки $2^m \to 1\underbrace{0\ldots0}_{m}$.

Тому (3.3) перепишемо наступним чином [26]:

$$\left(a+b\right)\bmod\left(2^m-1\right)=\begin{cases} z, & \textit{якщо } z<2^m-1\left(c=0\right); \\ z+1, & \textit{якщо } z=2^m-1\left(c=0\right); \\ z+1, & \textit{якщо } z>2^m-1\left(c=1\right). \end{cases} \qquad (3.5)$$

Сформулюємо алгоритм виконання операції додавання за модулем 2^m-1:

1. Обчислити $z:=a+b$; зафіксувати c.
2. Якщо $z='1\ldots1'$, то $z:=z+1$; перейти до п.4.
3. Якщо $c=1$, то $z:=z+1$.
4. Видати z.

Розглянемо приклади додавання за модулем 2^m-1. Нехай $m=4$, тоді $2^m-1=15$.

$a=4_{10}=0100_2$

$b=7_{10}=0111_2$

$z=11_{10}=1011_2$

	0	1	0	0
+	0	1	1	1
0	1	0	1	1

$c=0 \to$ отримано правильний результат

$a = 9_{10} = 1001_2$

$b = 10_{10} = 1010_2$

$z = 4_{10} = 0100_2$

$$\begin{array}{cccc} 1 & 0 & 0 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} \end{array}$$

$c = 1 \rightarrow$ отримано неправильний результат, необхідно виконати корекцію (додати до отриманого результату 1)

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ + & & & & 1 \\ \hline & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0} \end{array}$$

$a = 7_{10} = 0111_2$

$b = 8_{10} = 1000_2$

$z = 0_{10} = 0000_2$

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ + & 1 & 0 & 0 & 0 \\ \hline \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} \end{array}$$

незважаючи на те, що $c = 0$ – отримано неправильний результат, тому необхідно виконати корекцію (додати до отриманого результату 1)

$$1\ 1\ 1\ 1 \xrightarrow{+1} \boxed{0\ 0\ 0\ 0}$$

Реалізація мікрооперації SBM – віднімання m-розрядних двійкових величин за модулем $\mathbf{2^m - 1}$

Мікрооперацію віднімання за модулем $2^m - 1$ математично запишемо так [31]:

$$(a-b) \bmod \left(2^m - 1\right) = \begin{cases} 0, & \textit{якщо } a = b; \\ a - b, & \textit{якщо } a > b; \\ a - b + \left(2^m - 1\right), & \textit{якщо } a < b. \end{cases} \qquad (3.6)$$

Наприклад, $m=4$, $2^m-1=15$:

1. $a=2$, $b=2 \Rightarrow (2-2) \bmod 15 = 0 \bmod 15 = 0$.

2. $a=5$, $b=2 \Rightarrow (5-2) \bmod 15 = 3 \bmod 15 = 3$.

3. $a=2$, $b=5 \Rightarrow (2-5) \bmod 15 = -3 \bmod 15 = -3+15 = 12$.

В ЕОМ операцію віднімання двійкових кодів реалізують шляхом перетворення від'ємника у доповняльний код та його додавання до зменшуваного. З врахуванням цього другу рівність з (3.6) перепишемо таким чином:

$$a-b = a+\bar{b}+1, \tag{3.7}$$

а третю, як

$$a-b+\left(2^m-1\right) = a+\bar{b}+1+2^m-1 = a+\bar{b}+2^m. \tag{3.8}$$

Доданок 2^m не вливає на значущі цифри результату, оскільки $2^m \to 1\underbrace{0\ldots0}_{m}$, тому (3.6) перепишемо у вигляді [25]:

$$(a-b) \bmod \left(2^m-1\right) = \begin{cases} 0, & \textit{якщо } a=b; \\ a+\bar{b}+1, & \textit{якщо } a>b; \\ a+\bar{b}, & \textit{якщо } a<b. \end{cases} \tag{3.9}$$

Значення $a+\bar{b}$ на виході комбінаційного суматора досягається шляхом видачі прямого коду з виходу регістра першого операнда та інверсного – з виходу регістра другого операнда. Таким чином, результатом підсумовування двох величин на входах комбінаційного суматора є $a+\bar{b}$, тому можемо використати позначення $z = a+\bar{b}$.

Розглянемо випадок, коли $a=b$. Тоді

$$z = a+\bar{b} = a+\bar{a} = b+\bar{b} = 2^m-1 = \underbrace{11\ldots1}_{m}, \tag{3.10}$$

а відповідно до першої рівності з (3.9) z має дорівнювати 0. Отже, при $a=b$, значення z необхідно проінвертувати (знайти $\bar{z}$), або додати '1'.

Тому (3.9) можна переписати, як

$$(a-b)\bmod\left(2^m-1\right)=\begin{cases} z+1, & \textit{якщо } a=b\,(c=0);\\ z+1, & \textit{якщо } a>b\,(c=1);\\ z, & \textit{якщо } a<b\,(c=0). \end{cases} \qquad (3.11)$$

Сформулюємо алгоритм виконання операції віднімання за модулем 2^m-1:

1. Обчислити $z:=a+\bar{b}$; зафіксувати c.
2. Якщо $z=$'11…1', то $z:=z+1$; перейти до п.4.
3. Якщо $c=1$, то $z:=z+1$.
4. Видати z.

Розглянемо приклади віднімання за модулем 2^m-1. Нехай $m=4$, тоді $2^m-1=15$.

$a=2_{10}=0010_2$

$b=5_{10}=0101_2$

	0	0	1	0	
+	1	0	1	0	$\leftarrow\bar{b}$
0	1	1	0	0	$\leftarrow z$

$z=12_{10}=1100_2$

$c=0 \rightarrow$ отримано правильний результат

$a=5_{10}=0101_2$

$b=2_{10}=0100_2$

	0	1	0	1	
+	1	1	0	1	$\leftarrow\bar{b}$
1	0	0	1	0	$\leftarrow z$

$z=3_{10}=0011_2$

$c=1 \rightarrow$ отримано неправильний результат, необхідно виконати корекцію (додати до отриманого результату 1)

	0	0	1	0	$\leftarrow z$
+				1	
	0	0	1	1	$\leftarrow z+1$

$a = 6_{10} = 0110_2$

$b = 6_{10} = 0110_2$

$z = 0_{10} = 0000_2$

$$\begin{array}{rcccc} & 0 & 1 & 1 & 0 \\ + & 1 & 0 & 0 & 1 \quad \leftarrow \bar{b} \\ \hline \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} \quad \leftarrow z \end{array}$$

незважаючи на те, що $c = 0$ – отримано неправильний результат, тому необхідно виконати корекцію (проінвертувати отриманий результат або до отриманого результату додати 1)

$$1\ 1\ 1\ 1 \xrightarrow{+1} \boxed{0\ 0\ 0\ 0}$$

При використанні табличного способу зберігання елементів поля, можна врахувати той факт, що мікрооперація $(a+b) \bmod \left(2^m - 1\right)$, а також мікрооперація $(a-b) \bmod \left(2^m - 1\right)$ виконується над показниками степеня примітивного елемента α поля, а результати виконання операцій в $GF\left(2^m\right)$ мають бути у числовому поданні, тому у випадку $z = 2^m - 1$ можна не виконувати корекцію величини z в регістрі RGZ залишаючи її без змін, а натомість в таблиці (ПЗП) елементів поля, за адресою ‘11 … 1’ необхідно записати (зафіксувати) значення, а саме ‘0 … 01’, таке саме як і за адресою ‘00 … 0’ оскільки $\alpha^0 = \alpha^{2^m - 1} = 1$ (рис. 3.4).

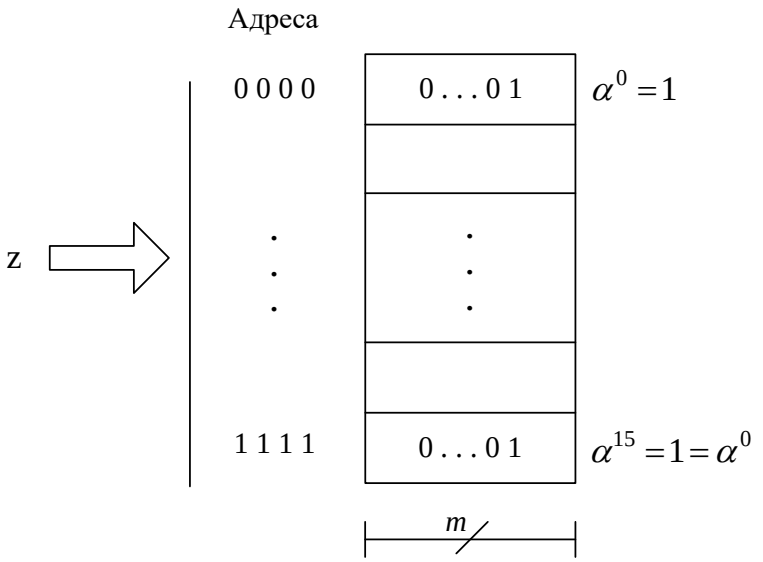


Рис. 3.4. Схема розміщення елементів поля в ПЗП

Тоді алгоритм виконання операції додавання та віднімання за модулем 2^m-1 набуває вигляду:

1. Якщо виконується операція додавання, то обчислити $z := a+b$, інакше $z := a+\overline{b}$; зафіксувати значення c.
2. Якщо $c=1$, то $z := z+1$.
3. Видати z.

Реалізація мікрооперації MLM – множення двійкових величин за модулем 2^m-1

Мікрооперацію множення m-розрядних величин a і b за модулем 2^m-1 можна записати таким чином:

$$
\begin{aligned}
Z &= (a\cdot b)\bmod\left(2^m-1\right)= \\
&= a\cdot\left(b_{m-1}\cdot 2^{m-1}+b_{m-2}\cdot 2^{m-2}+...+b_1\cdot 2+b_0\right)\bmod\left(2^m-1\right)= \\
&= \Bigg(\Bigg(\Bigg(\Bigg(...\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg((0+a\cdot b_{m-1})\bmod\left(2^m-1\right)\Bigg)\cdot 2\Bigg)\bmod\left(2^m-1\right)\Bigg)+a\cdot b_{m-2}\Bigg)\bmod\left(2^m-1\right)\Bigg)\cdot 2\Bigg)\bmod\left(2^m-1\right)+ \\
&\quad +a\cdot b_{m-3}\Bigg)\bmod\left(2^m-1\right)\Bigg)\cdot 2\Bigg)\bmod\left(2^m-1\right)\Bigg)+...+a\cdot b_1\Bigg)\bmod\left(2^m-1\right)\Bigg)\cdot 2\Bigg)\bmod\left(2^m-1\right)+a\cdot b_0\Bigg)\bmod\left(2^m-1\right).
\end{aligned}
$$

Сума часткових добутків за модулем 2^m-1 на j-му кроці, $j=1,2,...,m-1$ обчислюється за рекурентною формулою

$$Z_j=\left(\left(\left(Z_{j-1}+a\cdot b_{m-j}\right)\bmod\left(2^m-1\right)\right)\cdot 2\right)\bmod\left(2^m-1\right) \qquad (3.12)$$

з початковим значенням $j=1$, $Z_0=0$, тобто

$$Z_j=\begin{cases}\left(2\cdot Z_{j-1}\right)\bmod\left(2^m-1\right), \textit{ якщо } b_{m-j}=0; \\ \left(2\cdot\left(\left(Z_{j-1}+a\right)\bmod\left(2^m-1\right)\right)\right)\bmod\left(2^m-1\right), \textit{ якщо } b_{m-j}=1.\end{cases} \qquad (3.13)$$

В останньому такті зсув накопиченої суми часткових добутків $\left(Z_m\right)$ не потрібен:

$$Z_m=\left(Z_{m-1}+a\cdot b_0\right)\bmod\left(2^m-1\right).$$

Результатом є $Z := Z_m$.

Сформулюємо алгоритм виконання мікрооперації множення за модулем $2^m - 1$:

1. Встановити $Z := 0$.
2. Якщо $b_{m-1} = 1$, то $Z := \left(\left(Z + a \right) \cdot 2 \right) \bmod \left(2^m - 1 \right)$ і $b := b \cdot 2$; інакше $Z := \left(Z \cdot 2 \right) \bmod \left(2^m - 1 \right)$ і $b := b \cdot 2$.
3. п.2 повторити $m - 2$ рази.
4. Якщо $b_{m-1} = 1$, то $Z := \left(\left(Z + a \right) \cdot 2 \right) \bmod \left(2^m - 1 \right)$; інакше $Z := \left(Z \cdot 2 \right) \bmod \left(2^m - 1 \right)$.

Відомо, що множення числа на 2 апаратно реалізується як бітовий зсув на один розряд вліво. Для апаратної реалізації алгоритму множення за модулем $2^m - 1$ необхідно реалізувати мікрооперацію зсуву вліво за модулем $2^m - 1$, яка використовується при зсуві суми часткових добутків.

Алгоритм виконання зсуву за модулем $2^m - 1$ полягає в наступному: якщо старший розряд слова, яке зсувається є нульовим, то зсув вліво за модулем $2^m - 1$ еквівалентний логічному зсуву вліво, в іншому випадку необхідно виконати корекцію результату, а саме, додати одиницю до молодшого розряду отриманого результату.

Схемотехнічно алгоритм виконання мікрооперації додавання, віднімання та множення за модулем $2^m - 1$ можна реалізувати як показано на рис. 3.5, де *A, B* – *m*-розрядні операнди, WR_A, WR_B та WR_Z – сигнали прийому коду (*WRite*) у регістри *RGA*, *RGB* та *RGZ* відповідно; RD_A, RD_B та RD_Z – сигнали видачі коду (*ReaD*) з регістрів *RGA*, *RGB* та *RGZ* відповідно; сигнал RDI_B – сигнал видачі інвертованого коду з регістра *RGB*. Вивод *msb_reg_b* (*Most Significant Bit of REGister B*) приєднано до старшого розряду регістру *B*.

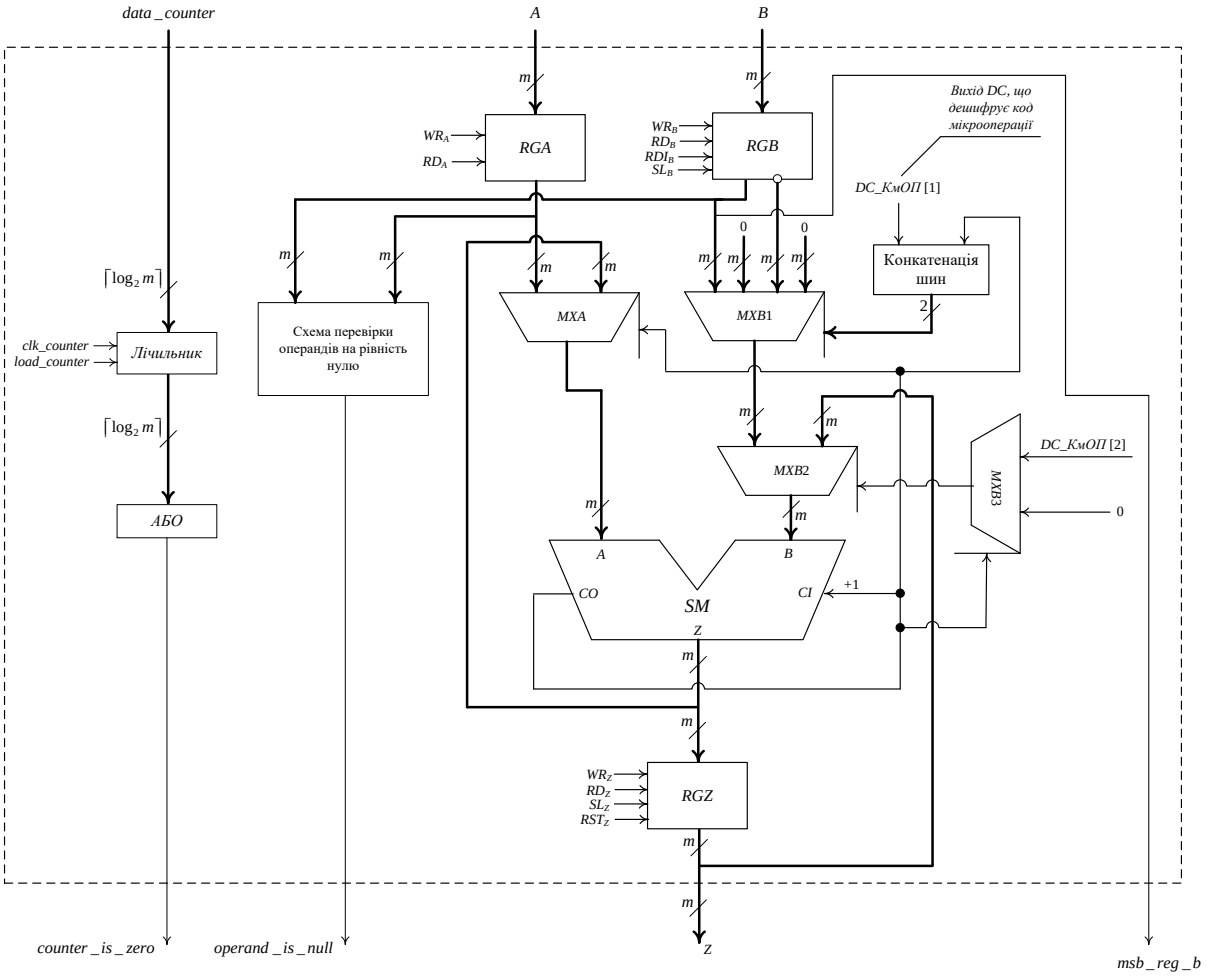


Рис. 3.5. Функціональна схема вузла виконання операції додавання, віднімання та множення за модулем $2^m - 1$

Перед початком виконання мікрооперації множення регістр Z ініціалізується нульовим значенням, а лічильник значенням $m-1$. Встановлення регістра Z в нульове значення відбувається за активним рівнем сигналу $reset_Z$, а завантаження значення з входу *data_counter* у лічильник відбувається за активним значення сигналу *load_counter*. Далі слід перевірити на рівність нулю кожен з операндів. Якщо один з операндів є нульовим, то результат теж буде нульовим і виконання мікрооперації множення необхідно закінчити (в регістрі Z знаходиться нульове значення).

В тому випадку коли обидва операнди ненульові, виконуємо дії передбачені алгоритмом виконання мікрооперації множення за модулем $2^m - 1$.

Реалізація мікрооперації CPL – інвертування двійкових величин

Мікрооперацію інвертування двійкових величин можна реалізувати шляхом додавання на виході схеми, що реалізує мікрооперації додавання, віднімання та множення за модулем $2^m - 1$ мультиплексора, який дозволить обирати що саме виводити на вихід операційного автомата блоку виконання мікрооперацій за модулем $2^m - 1$, а саме значення з регістра Z або інвертоване значення першого операнда.

Блок виконання мікрооперацій за модулем $2^m - 1$ *(додавання, віднімання, множення, інвертування)*

Блок складається з операційного автомата та керуючого автомата. Операційний автомат (рис. 3.6) виконує необхідні операції або мікрооперації, а керуючий автомат забезпечує генерування послідовності сигналів відповідно до алгоритму виконання мікрооперації.

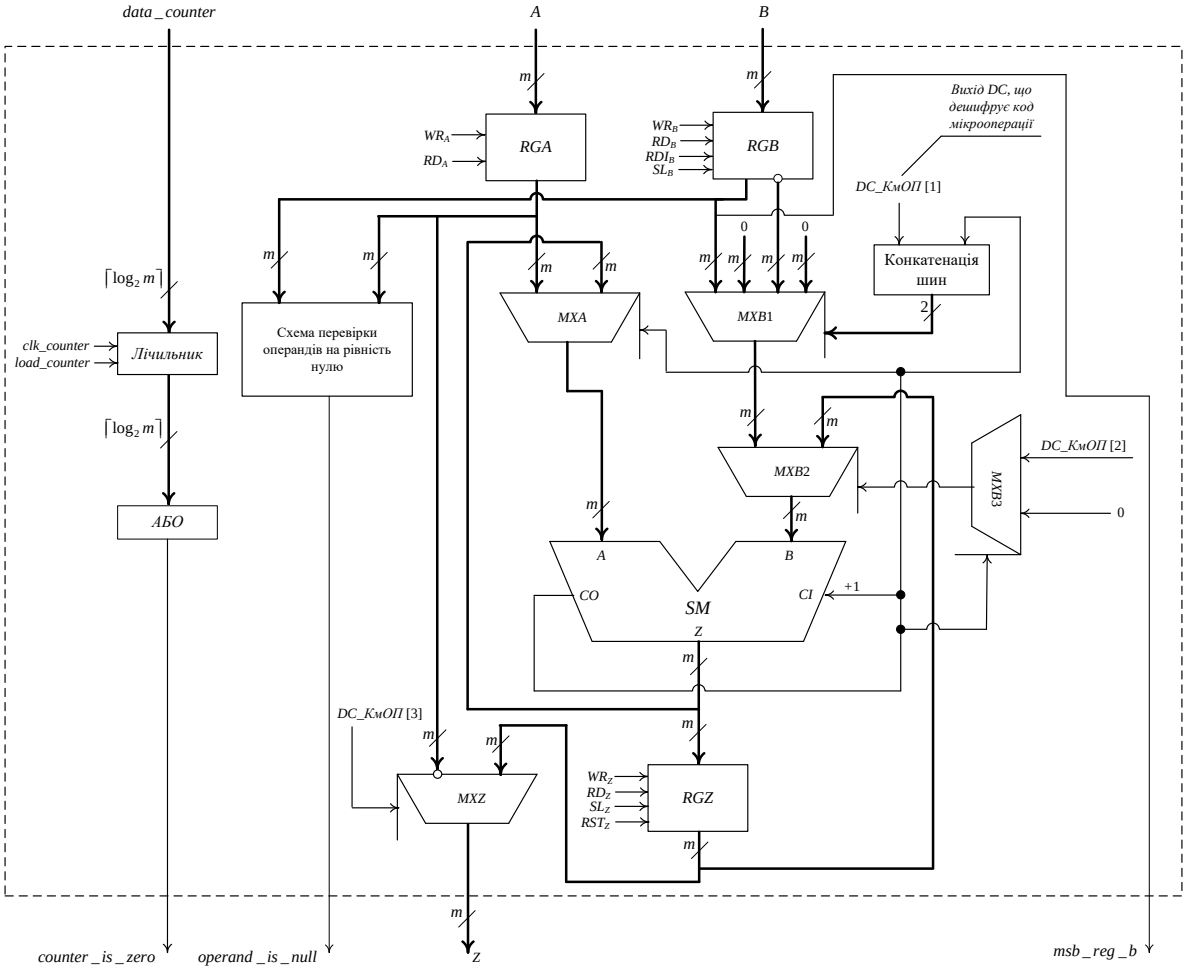


Рис. 3.6. Функціональна схема операційного автомата блока виконання мікрооперацій за модулем $2^m - 1$

Керуючий автомат працює згідно алгоритмів, що наведено на рис. В.1 – В.4 (додаток В). На вхід керуючого автомата надходить код операції, тактовий синхросигнал *clk*, сигнал *reset*, за допомогою якого керуючий автомат встановлюється у початковий стан, а також значення логічних умов з операційного автомату. На вихід генерує керуючі сигнали для операційного автомату та сигнал *result_ready* за активним значенням якого робиться висновок, що на виході даного блоку наявний результат виконання операції.

3.4. Реалізація операцій над елементами поля $GF(2^m)$

При табличному зберіганні елементів поля $GF(2^m)$ операції додавання, знаходження протилежного елемента та віднімання виконуються над многочленним поданням елементів поля, операції множення, знаходження мультиплікативного оберненого елемента, ділення та піднесення до степеня над поданням елементів у вигляді степеня примітивного елемента поля. У зв’язку з тим, що різні операції вимагають подання елементів у різних формах виникає необхідність у операціях конвертування многочленного подання елемента поля $GF(2^m)$ у степеневе і навпаки.

Таким чином, для роботи з елементами поля $GF(2^m)$ блок виконання операцій над елементами поля $GF(2^m)$ має підтримувати виконання набору команд, що наведено у табл. 3.5.

При побудові блоку виконання операцій над елементами поля $GF(2^m)$ реалізуємо його таким чином, що на вхід завжди має надходити многочленне подання елементів поля та отриманий результат на виході блоку виконання операцій над елементами поля $GF(2^m)$ також буде поданий у многочленному поданні [24].

Таблиця 3.5

Система команд блоку виконання операцій над елементами поля $GF(2^m)$

КОП	Мнемоніка	Опис мнемоніки	Призначення операції
0 0 0	ADD_SUB	ADDition and SUBtraction	додавання та віднімання елементів поля $GF(2^m)$
0 0 1	MULT	MULTiplication	множення елементів поля $GF(2^m)$
0 1 0	DIV	DIVision	ділення елементів поля $GF(2^m)$
0 1 1	POW	POWer	піднесення до степеня елементів поля $GF(2^m)$
1 0 0	INVM	INVerse Multiplicative	знаходження мультиплікативно оберненого елемента поля $GF(2^m)$
1 0 1	CDP	Convert Digit to Power	перетворення числового (многочленного) подання елемента поля $GF(2^m)$ у степеневе
1 1 0	CPD	Convert Power to Digit	перетворення степеневого подання елемента поля $GF(2^m)$ у числове (многочленне)

Реалізація операцій ADD_SUB, MULT, DIV, POW, INVM

Операція *ADD_SUB* апаратно реалізується як двовходовий елемент *XOR* з розрядністю операндів *m* та передбачає надходження операндів на вхід у многочленному поданні.

Математично операції *MULT* та *DIV* передбачають надходження операндів у степеневому поданні, але апаратно вони реалізовані таким чином, що значення операндів надходить у многочленному поданні, потім перетворюється до степеневого подання, а далі залучається блок виконання мікрооперацій за модулем $2^m - 1$ для додавання / віднімання операндів за

модулем $2^m - 1$ над значенням степеневого подання і отриманий результат переводиться у многочленне подання.

Операція *POW* передбачає надходження першого операнда у степеневому поданні, другим операндом виступає звичайне число у прямому коді, але аналогічно, до операції *MULT* та *DIV* спочатку перший операнд надходить у многочленному поданні, потім перетворюється у степеневе подання, і далі залучається блок виконання мікрооперацій за модулем $2^m - 1$ для множення операндів за модулем $2^m - 1$, отриманий результат перетворюється у многочленне подання.

Операція *INVM* передбачає надходження операнда у степеневому поданні, але аналогічно до попередніх трьох операцій спочатку операнд надходить у многочленному поданні, потім перетворюється у степеневе подання, і далі залучається блок виконання мікрооперацій за модулем $2^m - 1$ для інвертування значення операнда і отриманий результат перетворюється у многочленне подання.

Реалізація операцій CDP та CPD

Операції *CDP* та *CPD* реалізуються таблично (рис. 3.7), тобто шляхом зберігання в ПЗП всіх елементів поля $GF(2^m)$.

При виконанні операції *CDP* адресою має бути числове подання елемента поля та звертання має відбуватися до відповідного рядка лівої частини таблиці (права частина таблиці при цьому не використовується). Результатом читання з ПЗП буде невід'ємний показник степеня примітивного елемента α поля.

При виконанні операції *CPD* адресою має бути невід'ємний показник степеня примітивного елемента поля та звертання має відбуватися до відповідного рядка правої частини таблиці (ліва частина таблиці при цьому не використовується). Результатом читання з ПЗП буде числове значення елемента поля.

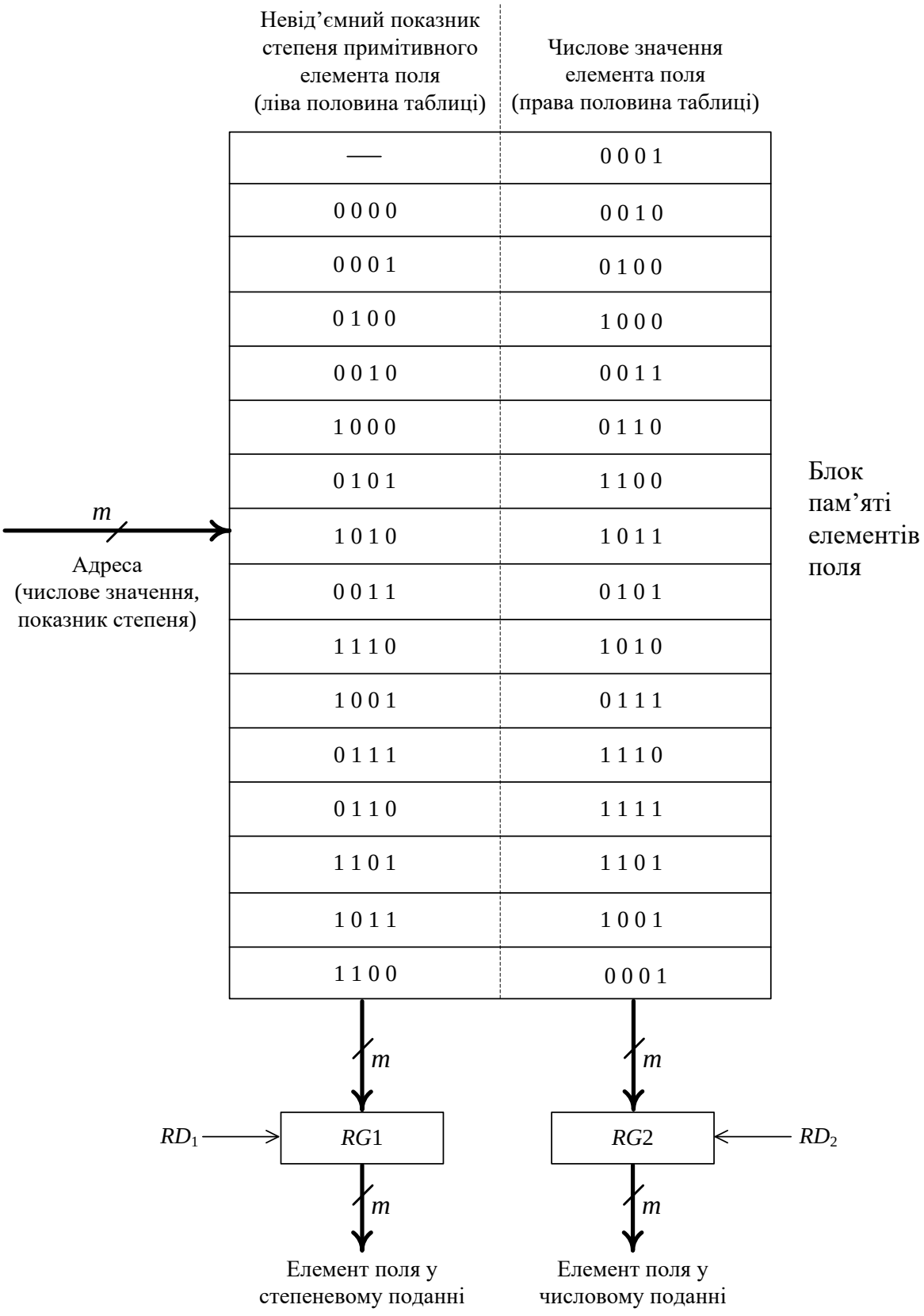


Рис. 3.7. Схема перетворення елемента поля з числового подання у степеневе та з степеневого у числове

Функціональна організація операційного автомата блока виконання операцій в полі $GF(2^m)$ наведена на рис. 3.8 [21]. На даній функціональній схемі *MUX*_1 – мультиплексор адрес, який визначає яке значення подавати на адресний вхід *ROM*, каскад мультиплексорів *MUX*_2 та *MUX*_3 дозволяє обрати, яке значення подавати на вхід регістра *Z* (регістра результату). Було прийняте рішення будувати каскад з двох мультиплексорів з метою економії апаратних ресурсів, оскільки виникла необхідність з'єднати 5 входів та один вихід, то потрібно було залучати один восьмивходовий мультиплексор, а в даному випадку було використано один чотирьохвходовий та один двовходовий мультиплексор, що зменшує апаратні витрати.

Розглянемо докладніше процес роботи блока виконання операцій у полі $GF(2^m)$.

При виконанні команди *ADD_SUB* операнди у многочленному поданні одразу надходять на елемент *XOR*, де виконується підсумовування і результат виконання операції, через каскад мультилпексорів, записується у регістр результату. Керуючий автомат блока виконання операцій у полі $GF(2^m)$ при виконанні операції додавання та віднімання працює згідно алгоритму, що наведено на рис. В.5.

При виконанні команд *MULT* та *DIV*, що потребують перетворення обох операндів у степеневе подання, спочатку на адресний вхід *ROM* подається значення першого операнда, з першого виходу *ROM* степеневе подання першого операнда записується в регістр *A*, далі на вхід *ROM* подається значення другого операнда, з першого виходу *ROM* степеневе подання другого операнда одразу потрапляє на вхід блока виконання мікрооперацій за модулем $2^m - 1$, на інший вхід якого через мультиплексор *MX*4 подається значення з регістра *A*. Отриманий результат з виходу блока виконання мікрооперацій за модулем $2^m - 1$ через *MX*1 надходить на адресний вхід *ROM*, далі з другого виходу *ROM*, через каскад мультиплексорів, результат виконання операції надходить у многочленному поданні на вхід регістра результату. Керуючий автомат блока виконання

операцій у полі $GF(2^m)$ при виконанні операції множення та ділення працює згідно алгоритмів, що наведені на рис. В.6 та В.7.

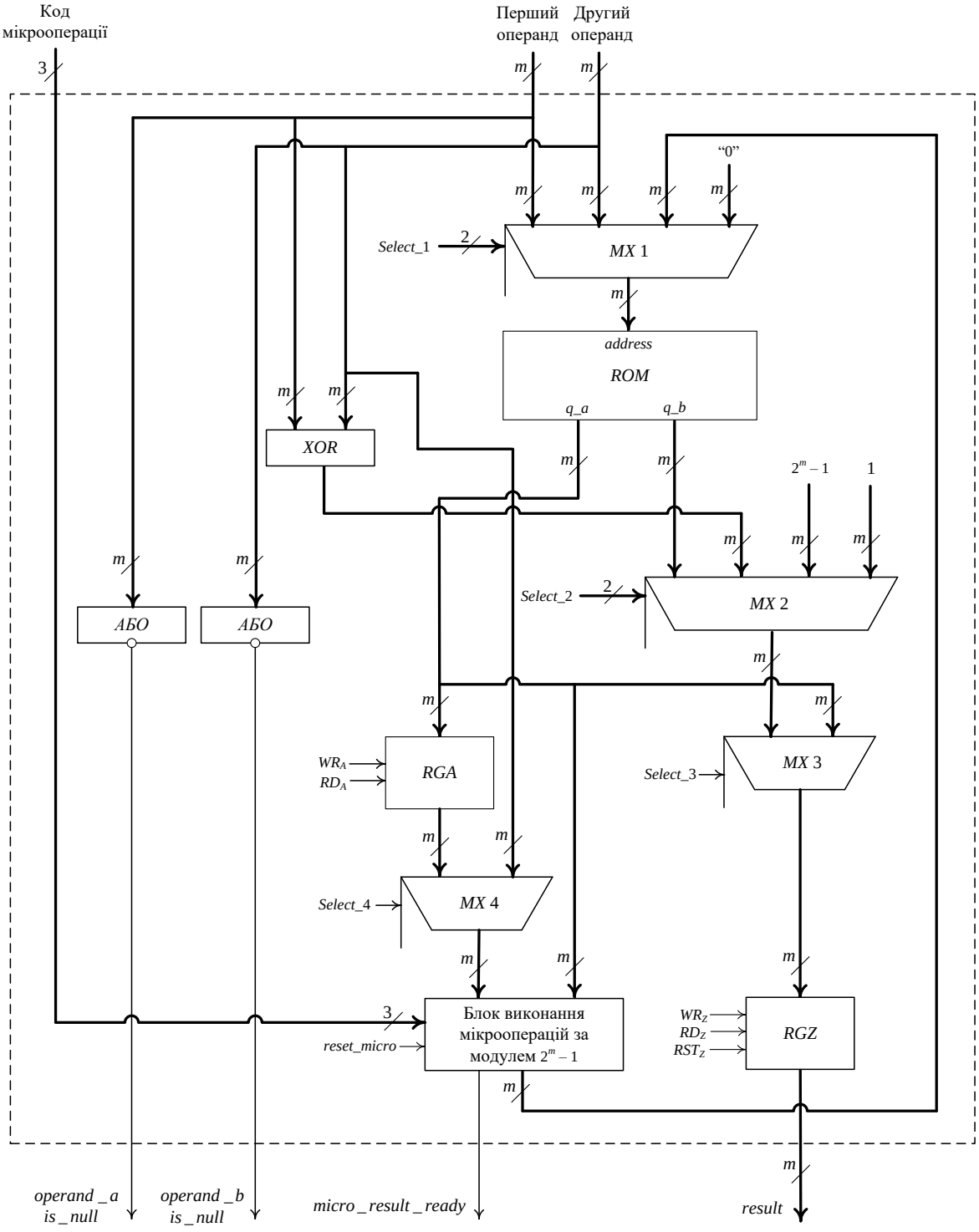


Рис. 3.8. Функціональна схема операційного автомата блока виконання операцій у полі $GF(2^m)$

При виконанні команд *POW* та *INVM*, що потребують перетворення тільки одного операнду у степеневе подання, спочатку на вхід *ROM* подається значення першого операнда, з першого виходу *ROM* степеневе подання першого операнда одразу потрапляє на вхід блока виконання мікрооперацій за модулем $2^m - 1$, на інший вхід якого через мультиплексор *MX*4, надходить значення другого операнда (показник степеня до якого потрібно піднести перший операнд для операції *POW*). Далі, за аналогією, до процесу виконання операцій *MULT* та *DIV*, отриманий результат перетворюється у многочленне подання та записується у регістр результату. Керуючий автомат блока виконання операцій у полі $GF(2^m)$ при виконанні операції піднесення до степеня та знаходження мультиплікативно оберненого елемента працює згідно алгоритмів, що наведені на рис. В.8 та В.9.

При роботі блока виконання операцій у полі $GF(2^m)$ також контролюється надходження нульових операндів та прискорюється отримання результату в такому випадку. Якщо при виконанні операції ділення другий операнд є нульовим, то вихід результату переводиться у третій стан.

Керуючий автомат блока виконання операцій у полі $GF(2^m)$ при виконанні операції конвертування степеневого подання елемента поля у многочленне та навпаки працює згідно алгоритмів, що наведені на рис. В.10 та В.11.

Структурна організація блока виконання операцій у полі $GF(2^m)$

Керуючий автомат (рис. 3.9) на вхід приймає код операції, тактовий синхросигнал *clk*, сигнал *reset*, за допомогою якого керуючий автомат встановлюється у початковий стан, а також значення логічних умов з операційного автомата. На вихід генерує керуючі сигнали для операційного

автомата та сигнал *result_ready* за активним значенням якого робиться висновок, що на виході даного блоку наявний результат виконання операції.

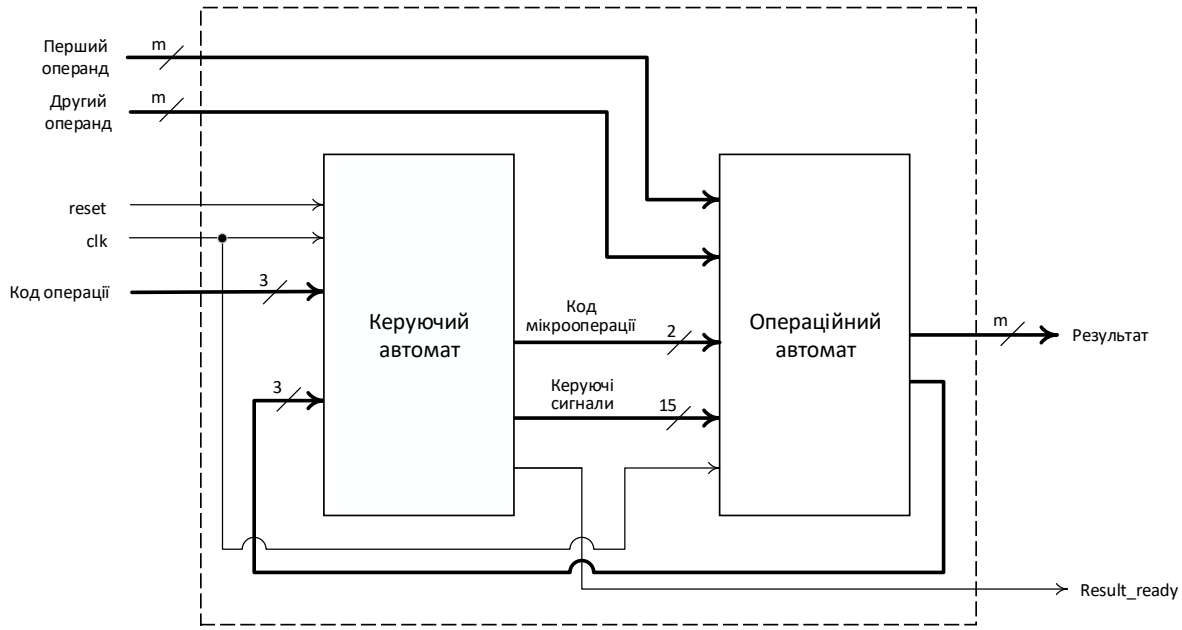


Рис. 3.9. Блок виконання операцій у полі $GF(2^m)$

3.5. Метод виконання операцій над елементами поля $GF(2^m)$ з використанням розрідженої таблиці

Основним недоліком запропонованого методу виконання операцій над елементами поля $GF(2^m)$ є використання великого об’єму пам’яті для зберігання таблиці відповідності між многочленним та степеневим поданням елементів поля [22]. Нівелювати цей недолік можна шляхом зберігання розрідженої таблиці відповідності між многочленним та степеневим поданням елементів поля.

Наприклад, таблиця елементів поля $GF(2^4)$ (рис. 3.3) може бути розріджена, якщо зберігати кожен третій рядок таблиці (рис. 3.10), в такому випадку маємо ступінь розрідженості $k = 3$, тобто об’єм необхідної пам’яті для зберігання таблиці зменшується в 3 рази.

Адреса у новій таблиці	Адреса у початковій таблиці	Невід'ємний показник степеня примітивного елемента α поля	Числове значення елемента поля
0 0 0	0 0 0 0	–	1 (α^0)
0 0 1	0 0 1 1	4	8 (α^3)
0 1 0	0 1 1 0	5	12 (α^6)
0 1 1	1 0 0 1	14	10 (α^9)
1 0 0	1 1 0 0	6	15 (α^{12})
1 0 1	1 1 1 1	12	1 (α^{15})

$2^m / k$

m m

Рис. 3.10. Розріджена таблиця для поля $GF(2^4)$ та незвідного многочлена $x^4 + x + 1$

У розглянутому прикладі використано функцію ділення на 3 (залишаємо в таблиці кожен рядок, номер якого кратний трьом). Загалом, для скорочення таблиці може бути використана будь-яка математична функція, що має обернену. Наприклад, можна використати функцію логарифмування за визначеною основою і залишити в таблиці рядки, номери яких при застосуванні обраної логарифмічної функції дають ціле число. Але такий вибір функції розрідження спричинить нерівномірне розрідження таблиці – чим далі від початку таблиці, тим більш розрідженою вона буде. Тому даний підхід доцільно використовувати, якщо наявні апріорні відомості про закон розподілу вхідних даних, а сам закон відрізняється від рівномірного; в іншому випадку доцільним є використання функції, яка забезпечує рівномірне розрідження таблиці. Такою є функція ділення на число k, де k – ступінь розрідження.

Після вибору функції розрідження необхідно побудувати алгоритм перетворення елементів поля $GF(2^m)$ зі степеневого подання у числове і навпаки.

Повна таблиця відповідності між многочленним та степеневим поданням елементів поля $GF(2^m)$ має наступну властивість: числові подання елементів поля розміщені у порядку зростання відповідних їм значень степеневого подання (це ілюструє другий стовпчик на рис. 3.3). Дана властивість забезпечує можливість знаходження для довільно заданого елемента у степеневому поданні, базового для нього елемента у розрідженій таблиці. Використовуючи знайдений базовий елемент, можна отримати числове подання заданого елемента, для пошуку якого необхідно не більше, ніж $\left\lfloor \frac{k}{2} \right\rfloor$ ітерацій.

Адреса базового елемента α^r у розрідженій таблиці знаходиться шляхом цілочисельного ділення показника степеня заданого елемента α^d, тобто d, на k з округленням до найближчого цілого. Позначимо результат цілочисельного ділення *addr*, значення у другому стовпчику таблиці за адресою *addr* позначимо s. Таким чином, α^r є степеневим поданням базового елемента, а s – числовим. Подальші дії залежать від того, до найменшого чи найбільшого цілого було виконане округлення (алгоритм 3.1).

Якщо округлення було виконане до найменшого цілого ($r < d$), то числове подання, що відповідає елементу α^d, знаходиться шляхом відтворення процесу побудови фрагмента таблиці відповідності числового та степеневого подання елементів поля $GF(2^m)$, починаючи від базового елемента α^r до елемента α^d. Тобто, числове подання елемента α^d знаходиться шляхом зсуву вліво значення s на $\left\{ \frac{d}{k} \right\}$ розрядів та отриманням остачі від ділення результату зсуву на незвідний многочлен.

Якщо округлення було виконане до найбільшого цілого ($r > d$), то числове подання, що відповідає елементу α^d, знаходиться шляхом відтворення процесу, що є зворотним до процесу побудови фрагмента таблиці відповідності числового та степеневого подання елементів

поля $GF(2^m)$, починаючи від елемента α^d до базового елемента α^r. Загальна кількість ітерацій дорівнює $k - \left\{\frac{d}{k}\right\}$. Якщо молодший розряд дорівнює 0, то одна ітерація полягає у зсуві операнда вправо на один розряд, інакше виконується додавання незвідного многочлена та зсув вправо на один розряд.

Алгоритм 3.1. Перетворення степеневого подання у многочленне з використанням розрідженої таблиці

Вхід: $table\left[\frac{2^m}{k} \text{ x } 2\right], ir\,[1 \text{ x } m],\ pow \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $res\,[1 \text{ x } m]$

1. $adr \leftarrow \left\lfloor \frac{pow}{k} \right\rfloor; c \leftarrow \left\{\frac{pow}{k}\right\}$
2. *if* $\left(c < k - c \ \ OR \ \ adr = \frac{2^m}{k}\right)$
   2.1. $res \leftarrow table\,[adr, 2]$ { res – *многочлен* }
   2.2. $j \leftarrow 0$
   2.3. *while* $(j < c)$
      2.3.1. *while* $(\deg(res) < m \ \text{ and } \ j < c)$
         a) $res \leftarrow res \cdot x$ { x – *одночлен* }
         b) $j \leftarrow j + 1$
      2.3.2. *end while*
      2.3.3. *if* $(\deg(res) \geq m)$
         a) $res \leftarrow res + ir$
      2.3.4. *end if*
   2.4. *end while*
3. *else*
   3.1. $adr \leftarrow adr + 1$
   3.2. $c \leftarrow k - c$
   3.3. $res \leftarrow table\,[adr, 2]$ { res – *многочлен* }
   3.4. $j \leftarrow 0$
   3.5. *while* $(j < c)$
      3.5.1. *if* $(res\,[0] == 1)$
         a) $res \leftarrow res + ir$
      3.5.2. *end if*
      3.5.3. *while* $(res\,[0] == 0 \ \text{ and } \ j < c)$
         a) $res \leftarrow res / x$ { x – *одночлен* }
         b) $j \leftarrow j + 1$
      3.5.4. *end while*
   3.6. *end while*
4. *end if*
5. *return res*

Розглянемо приклади.

Для елемента α^7 поля $GF(2^4)$ з незвідним многочленом $x^4 + x + 1$, використовуючи розріджену таблицю (рис. 3.10), знайти його многочленне подання. Ступінь розрідження $k = 3$.

Цілочисельне ділення 7 на 3 з округленням до найближчого цілого дає 2 (адреса базового елемента в розрідженій таблиці), тобто $addr = 2$. За

адресою 2 знаходимо числове подання базового елемента s = 1100. Зсуваємо значення s на $\left\{\frac{d}{k}\right\}=\left\{\frac{7}{3}\right\}=1$ розряд вліво та отримуємо s = 11000. Даний результат ділимо з остачею на незвідний многочлен 10011 (для даного прикладу ділення зводиться до однократного додавання незвідного многочлена) і отримуємо шукане значення, яке дорівнює 1011 (перевірити коректність можна за допомогою рис. 3.3).

Для елемента α^{14} поля $GF(2^4)$ з незвідним многочленом $x^4 + x + 1$, використовуючи розріджену таблицю (рис. 3.10), знайти його многочленне подання. Ступінь розрідження $k=3$.

Цілочисельне ділення 14 на 3 з округленням до найближчого цілого дає 5 (адреса базового елемента в розрідженій таблиці), тобто $addr$ = 5. За адресою 5 знаходимо числове подання базового елемента s = 0001. Загальна кількість ітерацій дорівнює $k-\left\{\frac{d}{k}\right\}=3-\left\{\frac{14}{3}\right\}=1$. Молодший розряд s дорівнює 1, тому виконуємо додавання незвідного многочлена (результат 10010) та зсув на один розряд вправо, отримуємо 1001 (перевірити коректність можна за допомогою рис. 3.3).

Розглянемо підхід до перетворення елементів поля $GF(2^m)$ з многочленного (числового) подання у степеневе [61].

У розрідженій таблиці наявні рядки повної таблиці, адреси яких у повній таблиці кратні k, тому необхідно від заданого числового подання елемента d перейти до числового подання елемента поля, що кратне k, тобто до числового подання, для якого присутнє відповідне степеневе подання у розрідженій таблиці. Таким чином, отримуємо базовий елемент з відомим степеневим поданням. Далі з отриманого степеневого подання базового елемента необхідно отримати степеневе подання шуканого елемента.

Перехід від заданого числового подання елемента d до числового подання елемента поля, що кратне k, здійснюється шляхом відтворення

процесу побудови фрагмента таблиці відповідності числового та степеневого подання елементів поля $GF(2^m)$, або процесу, що є зворотним до нього (алгоритм 3.2 та 3.3).

Алгоритм 3.2. Перетворення многочленного подання у степеневе з пошуком базового елемента у верхній частині розрідженої таблиці

Вхід: $table[\,2^m \text{ x } 2],\ pol, ir\,[1 \text{ x } m],\ k \in \mathbb{N}$

Вихід: $res \in \mathbb{N}$

1. $dec \leftarrow Bin2Dec(pol)$
2. $count \leftarrow 0$
3. *while* $\left(\left\{\frac{dec}{k}\right\} \neq 0\right)$
   - 3.1. *if* $\left(\left\{\frac{dec}{2}\right\} \neq 0\right)$
     - 3.1.1. $pol \leftarrow pol + ir$
     - 3.1.2. $pol \leftarrow pol / x$
     - 3.1.3. $dec \leftarrow Bin2Dec(pol)$
     - 3.1.4. $count \leftarrow count + 1$
   - 3.2. *end if*
   - 3.3. *while* $\left(\left\{\frac{dec}{k}\right\} \neq 0 \text{ and } \left\{\frac{dec}{2}\right\} == 0\right)$
     - 3.3.1. $pol \leftarrow pol / x$
     - 3.3.2. $dec \leftarrow Bin2Dec(pol)$
     - 3.3.3. $count \leftarrow count + 1$
   - 3.4. *end while*
4. *end while*
5. $dec \leftarrow dec / k$
6. *return* $table[dec, 1] + count$

Алгоритм 3.3. Перетворення многочленного подання у степеневе з пошуком базового елемента в нижній частині розрідженої таблиці

Вхід: $table[\,2^m \text{ x } 2],\ pol, ir\,[1 \text{ x } m],\ k \in \mathbb{N}$

Вихід: $res \in \mathbb{N}$

1. $dec \leftarrow Bin2Dec(pol)$
2. $count \leftarrow 0$
3. *while* $\left(\left\{\frac{dec}{k}\right\} \neq 0\right)$
   - 3.1. *while* $\left(\left\{\frac{dec}{k}\right\} \neq 0 \text{ and } \deg(pol) < m\right)$
     - 3.1.1. $pol \leftarrow pol \cdot x$
     - 3.1.2. $dec \leftarrow Bin2Dec(pol)$
     - 3.1.3. $count \leftarrow count + 1$
   - 3.2. *end while*
   - 3.3. *if* $(\deg(pol) = m)$
     - 3.3.1. $pol \leftarrow pol + ir$
     - 3.3.2. $dec \leftarrow Bin2Dec(pol)$
   - 3.4. *end if*
4. *end while*
5. $dec \leftarrow dec / k$
6. *return* $table[dec, 1] - count$

Розглянемо цей процес докладніше.

Пошук базового елемента відбувається ітераційним шляхом, допоки черговий отриманий результат не буде кратним *k*.

Якщо пошук базового елемента виконувати шляхом відтворення процесу, зворотного до процесу побудови фрагмента таблиці відповідності числового та степеневого подання елементів поля $GF(2^m)$, тобто у верхній частині розрідженої таблиці, то кожна ітерація полягає у наступних кроках.

1. Якщо молодший розряд елемента *d* дорівнює 1, то додати до операнда незвідний многочлен. Результат записати в *d*.
2. Зсунути *d* вправо на один розряд, допоки не отримаємо значення, кратне *k*, або допоки молодший розряд *d* не стане одиничним.

Під час виконання другого пункту ітераційного процесу необхідно підраховувати кількість виконаних зсувів. Нехай загальна кількість зсувів вправо дорівнює *count*.

Після отримання адреси базового елемента у повній таблиці, його адреса у розрідженій таблиці отримується шляхом ділення на *k*, таким чином $\frac{d}{k}$. Далі необхідно з розрідженої таблиці за адресою $\frac{d}{k}$ з першого стовпчика взяти степеневе подання та додати до нього значення *count*. Отриманий результат буде шуканим степеневим поданням.

Якщо пошук базового елемента виконувати шляхом відтворення процесу побудови фрагмента таблиці відповідності числового та степеневого подання елементів поля $GF(2^m)$, тобто в нижній частині розрідженої таблиці, то кожна ітерація полягає у наступних кроках.

1. Зсунути *d* вліво на один розряд, допоки не отримаємо значення кратне *k*, або допоки старший розряд *d* (розряд з номером *m* при нумерації з нуля) не стане одиничним.
2. Якщо старший розряд елемента *d* дорівнює 1, то додати до нього незвідний многочлен. Результат записати в *d*.

Під час виконання першого пункту ітераційного процесу необхідно підраховувати кількість виконаних зсувів. Нехай загальна кількість зсувів вправо дорівнює *count*.

Після отримання адреси базового елемента у повній таблиці, його адреса у розрідженій таблиці отримується шляхом ділення на k, таким чином $\frac{d}{k}$. Далі необхідно з розрідженої таблиці за адресою $\frac{d}{k}$, з першого стовпчика взяти степеневе подання та відняти від нього значення *count*. Отриманий результат буде шуканим степеневим поданням.

Розглянемо приклади.

Нехай для елемента 1010_2 поля $GF(2^4)$ з незвідним многочленом $x^4 + x + 1$, використовуючи розріджену таблицю (рис. 3.10), необхідно знайти його степеневе подання. Ступінь розрідження $k = 3$.

Згідно першого підходу виконуємо перевірку вхідного операнда на кратність 3, оскільки 1010_2 не кратне 3, то запускаємо ітераційний процес.

Перша ітерація. Виконуємо перевірку на парність. Вхідний операнд кратний 2, тому переходимо до зсувів та виконуємо один зсув. Отриманий проміжний результат 101_2 (*count* = 1).

Друга ітерація. Додаємо незвідний многочлен, отримуємо 10110_2. Зсуваємо результат вправо, отримуємо 1011_2 (*count* = 2).

Третя ітерація. Додаємо незвідний многочлен, отримуємо 11000_2. Зсуваємо результат вправо, отримуємо 1100_2 (*count* = 3).

Отримали число кратне 3, а саме 12. Ділимо його на 3, отримуємо 4. За адресою 4 у розрідженій таблиці знаходимо значення степеневого подання, що дорівнює 6. До отриманого значення додаємо накопичену кількість зсувів і отримуємо 9.

Згідно другого підходу виконуємо перевірку вхідного операнда на кратність 3, оскільки 1010_2 не кратне 3, то запускаємо ітераційний процес.

Перша ітерація. Виконуємо перевірку на кратність 3 та на рівність одиниці старшого розряду (розряду з номером m при нумерації з нуля).

Обидві умови не виконуються, тому переходимо до зсувів та виконуємо один зсув. Отриманий проміжний результат 10100_2 (*count* = 1). Додаємо незвідний многочлен, отримуємо 111_2.

Друга ітерація. Зсуваємо на два біта ліворуч, отримуємо 11100_2 (*count* = 3). Додаємо незвідний многочлен, отримуємо 1111_2.

Отримали число кратне 3, а саме 15. Ділимо його на 3, отримуємо 5. За адресою 5 у розрідженій таблиці знаходимо значення степеневого подання, що дорівнює 12. Від отриманого значення віднімаємо накопичену кількість зсувів і отримуємо 9.

3.6. Висновки до розділу 3

1. Аналіз показує, що особливістю поля $GF(2^m)$ є те, що різні форми подання елементів поля є ізоморфними, тобто такими, що замінюють одна одну, зокрема використовують степеневе, многочленне та числове подання. Степеневе подання доцільно використовувати для мультиплікативних операцій, таких як множення, ділення та обчислення мультипілкативно оберненого елемента, а многочленне подання для адитивних операцій, таких як додавання та віднімання. Недоліком степеневого подання є те, що воно не дозволяє отримувати нульовий елемент поля.
2. Найбільш раціональним є табличний спосіб виконання операцій над елементами поля $GF(2^m)$, коли зберігається відповідність многочленного та степеневого подання елементів поля. Використовується таблиця, що складається з двох частин: ліва частина таблиці використовуються при перетворенні числового подання у степеневе, а права частина – степеневого у числове, за рахунок чого досягається можливість виконувати операції над всіма елементами поля, в тому числі й над нульовим елементом.

3. Якщо потужність поля велика, то можна не зберігати всі елементи поля в таблиці, а формувати таблицю розрідженою, тобто зберігати лише базові елементи повної таблиці, що дозволяє скоротити витрати пам’яті на її зберігання. Це є можливим за рахунок скінченності поля та властивості циклічності показників степеня степеневого подання елементів поля.
4. Для розрідження таблиці може бути використана будь-яка математична функція, що має обернену. При такому виборі функції завжди можна відтворити будь-який елемент повної таблиці, маючи базовий елемент. В якості функції розрідження доцільно обирати функцію ділення на константу.
5. Запропоновано метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та відповідні структури апаратних засобів для його реалізації, що характеризуються універсальністю. Дослідження показали, що за рахунок розрідженого табличного зберігання елементів поля у многочленному та степеневому їх поданні забезпечується максимальна швидкодія та універсальність арифметико-логічного пристрою. Розроблений метод забезпечує зростання швидкодії в середньому на 15% порівняно з існуючим методом.

РОЗДІЛ 4
АРХІТЕКТУРА ПРОБЛЕМНО-ОРІЄНТОВАНОГО ПРОЦЕСОРА ДЛЯ ВИКОНАННЯ ОБЧИСЛЕНЬ У СКІНЧЕННИХ ПОЛЯХ

4.1. Етапи розроблення процесора Галуа

Розроблення процесора Галуа (*G*-процесора) складається з таких етапів [25, 28]:

1. Проектування системи команд (Асемблера) процесора Галуа.
2. Розроблення компілятора для перетворення асемблерного коду процесора Галуа в машинний код.
3. Синтез апаратної частини процесора Галуа.

На рис. 4.1 наведено етапи виконання програми мовою Асемблера процесора Галуа. Спочатку необхідно написати код мовою Асемблера процесора Галуа, далі компілятор перетворює мнемонічний код у машинний код, і отриманий машинний код записується у пам’ять команд процесора Галуа, виконання програми (машинного коду) передається процесору Галуа. Арифметичні команди процесора Галуа виконуються в арифметико-логічному пристрої, що міститься в складі процесора.

Таким чином, загальна структура апаратної частини процесора Галуа не буде залежати від виду поля, в якому виконуються операції, а змінюватись буде лише блок АЛП, що відповідає за виконання арифметичних операцій.

Процесор орієнтований на виконання операцій в полях Галуа двох видів – $GF(p)$ та $GF(2^m)$.

Рис. 4.1. Етапи виконання програми мовою Асемблера процесора Галуа

4.2. Структурна організація програмного забезпечення процесора Галуа

Розглянемо систему команд [29] процесора Галуа (табл. 4.1).

Таблиця 4.1

Система команд Асемблера процесора Галуа

КОП	Поле $GF(p)$		Довжина команди у байтах	Поле $GF(2^m)$	
	Операція	Мнемоніка		Операція	Мнемоніка
0 0 0 0	Додавання	ADD R1, R2	1	Додавання Віднімання	ADD R1, R2
0 0 0 1	Множення	MUL R1, R2	1	Множення	MUL R1, R2
0 0 1 0	Ділення	DIV R1, R2	1	Ділення	DIV R1, R2
0 0 1 1	Піднесення до степеня	POW R1, R2	1	Піднесення до степеня	POW R1, R2
0 1 0 0	Знаходження мультиплікативно оберненого елемента	INVM R	1	Знаходження мультиплікативно оберненого елемента	INVM R
0 1 0 1	Знаходження протилежного (адитивно оберненого) елемента	INVA R	1	Перетворення многочленного подання у степеневе	CDP R
0 1 1 0	Віднімання	SUB R1, R2	1	Перетворення степеневого подання у многочленне	CPD R
0 1 1 1	Пересилання “регістр-регістр”	MOV R1, R2	1	Пересилання з *reg* в *reg*	MOV R1, R2

Продовження табл. 4.1

КОП	Поле $GF(p)$			Довжина команди у байтах	Поле $GF(2^m)$		
	Операція		Мнемоніка		Операція		Мнемоніка
1 0 0 0	0	Пересилання “пам’ять-регістр”	MOV R, M	2	0	Пересилання “пам’ять-регістр”	MOV R, M
	1	Пересилання “регістр-пам’ять”	MOV M, R	2	1	Пересилання “регістр-пам’ять”	MOV M, R
1 0 0 1	0	Пересилання “масив-регістр”	MOV R, A [AC ± disp]	3	0	Пересилання “масив-регістр”	MOV R, A [AC ± disp]
	1	Пересилання “регістр-масив”	MOV A [AC ± disp], R	3	1	Пересилання “регістр-масив”	MOV A [AC ± disp], R
1 0 1 0	111	Безумовний перехід	JMP Label	2	111	Безумовний перехід	JMP Label
	xx1	Умовний перехід за нульовим значенням в регістрі	JMP R, Label (доступними є регістри R0, R1 та R2)		xx1	Умовний перехід за нульовим значенням в регістрі	JMP R, Label (доступними є регістри R0, R1 та R2)
1 0 1 1	Повторити цикл *val* разів		LOOP *N*, *val* (початок циклу)	2	Повторити цикл *val* разів		LOOP *N*, *val* (початок циклу)
1 1 0 0	Завантаження лічильника адрес значенням з іншого лічильника адрес		LOAD AC1, AC2	1	Завантаження лічильника адрес значенням з іншого лічильника адрес		LOAD AC1, AC2

Продовження табл. 4.1

КОП	Поле $GF(p)$			Довжина команди у байтах	Поле $GF(2^m)$		
	Операція		Мнемоніка		Операція		Мнемоніка
1 1 0 1	Завантаження лічильника адрес		LOAD AC, *val*	2	Завантаження лічильника адрес		LOAD AC, *val*
1 1 1 0	00	Інкремент регістру	INC R	1	00	Інкремент регістру	INC R
	10	Інкремент лічильника адрес	INC AC		10	Інкремент лічильника адрес	INC AC
	01	Декремент регістру	DEC R		01	Декремент регістру	DEC R
	11	Декремент лічильника адрес	DEC AC		11	Декремент лічильника адрес	DEC AC
1 1 1 1	0	Видача на вихід слова з пам’яті даних	OUT M	2	0	Видача на вихід схеми змінної (слова пам’яті даних)	OUT M
	1	Видача на вихід елемента масиву	OUT A [AC ± disp]	3	1	Видача на вихід схеми елемента масиву (слова пам’яті даних)	OUT A [AC ± disp]

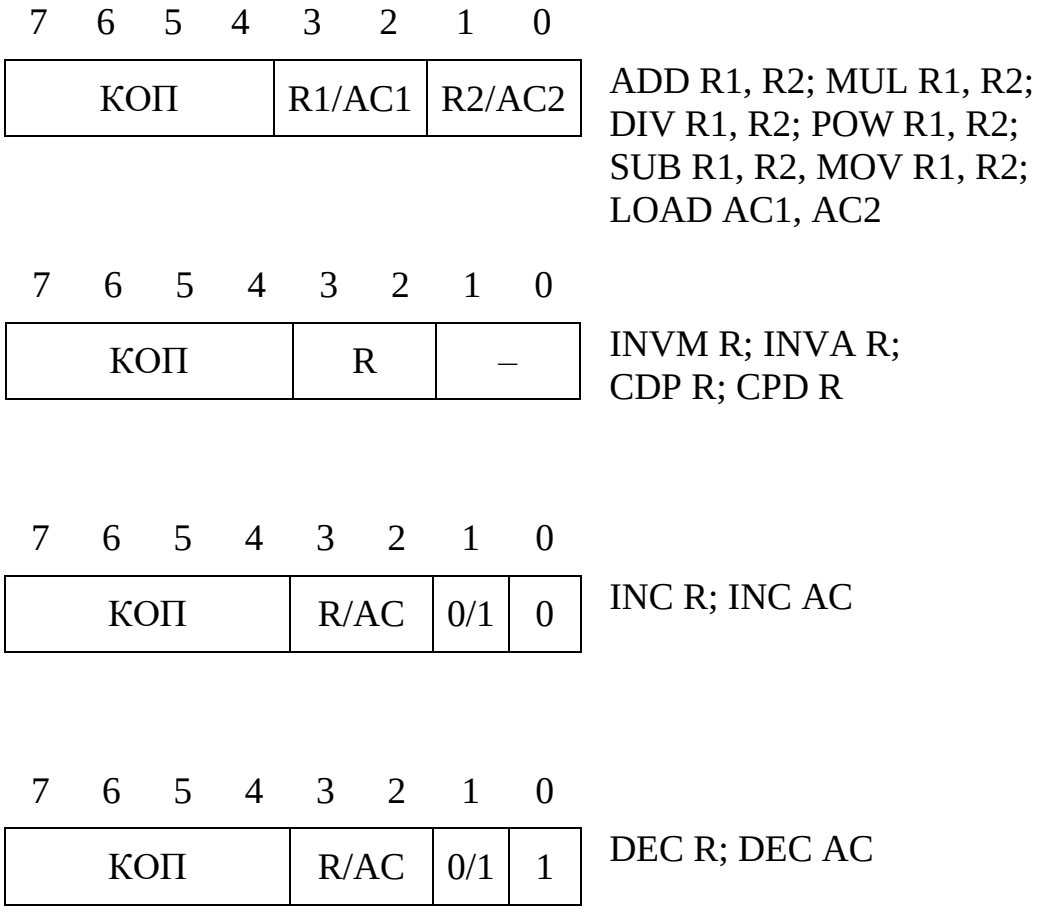
Примітка: xx – будь-яка комбінація двох бітів, окрім 11.

За форматом команди розрізняють *CISC* та *RISC* архітектуру процесора, що є найбільш поширеними. Зазвичай для побудови високопродуктивних систем використовується *RISC*-архітектура, тому саме така архітектура була обрана для процесора Галуа.

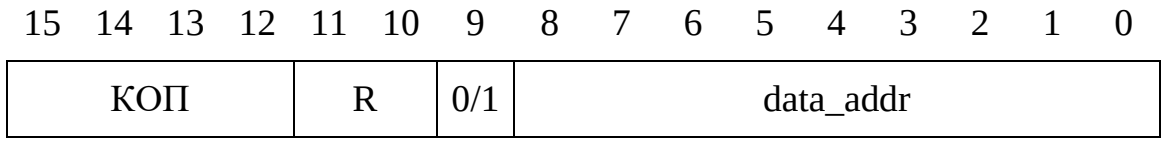
При розробці системи команд враховувались такі принципи *RISC*-архітектури: виконання будь-якої команди повинно потребувати невелику кількість машинних тактів. Щоб спростити декодування команд використовуються команди фіксованої довжини (однобайтні, двобайтні та трибайтні).

Розглянемо формати команд (табл. 4.2).

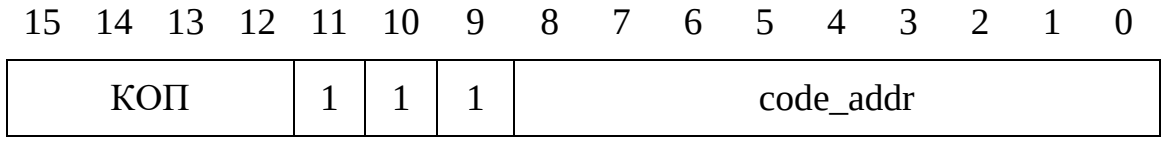
Формати однобайтних команд



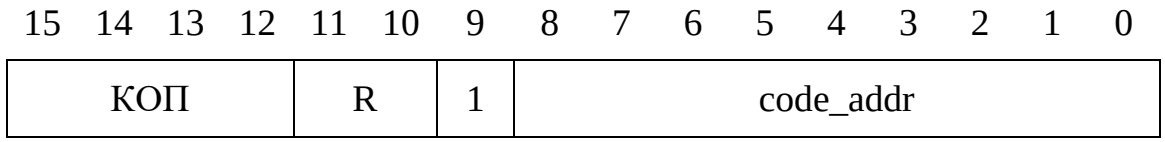
Формати двобайтних команд



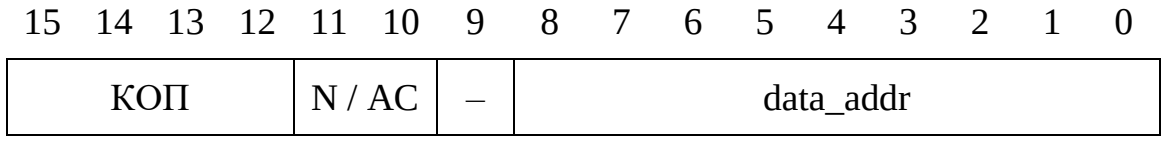
MOV M, R; MOV R, M



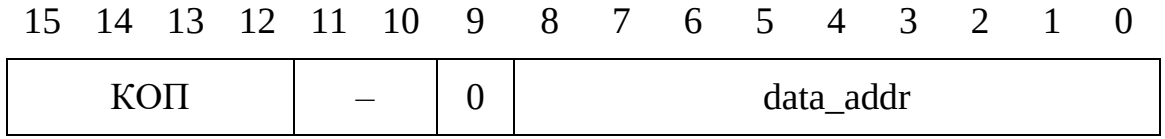
JMP Label



JMP R, Label

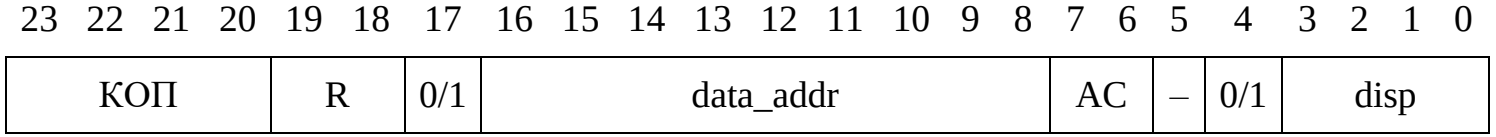


LOOP N, val;
LOAD AC, val



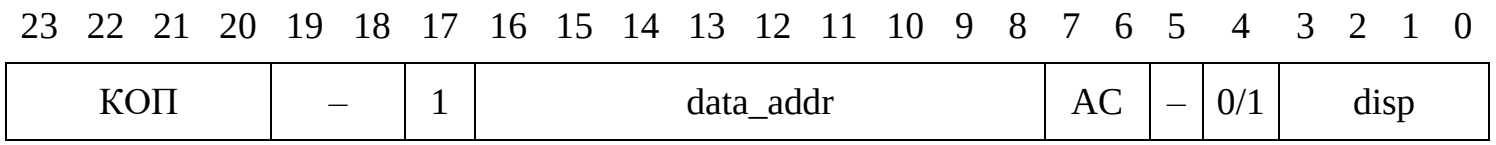
OUT M

Формати трибайтних команд



MOV A [AC ± disp], R

MOV R, A [AC ± disp]



OUT A [AC ± disp]

Таблиця 4.2

Класифікація системи команд процесора Галуа за довжиною команди (в байтах)

Мнемонічні позначення команд		
Однобайтних	Двобайтних	Трибайтних

ADD R1, R2
MUL R1, R2
DIV R1, R2
POW R1, R2
INVM R
INVA R
CDP R
CPD R
SUB R1, R2
MOV R1, R2
LOAD AC1, AC2
INC R
INC AC
DEC R
DEC AC
	MOV M, R
	MOV R, M
	JMP Label
	JMP R, Label
	LOOP N, val
	LOAD AC, val
	OUT M
		MOV A [AC ± disp], R
		MOV R, A [AC ± disp]
		OUT A [AC ± disp]

Приклади використання кожної команди наведено у Додатку Г.

Система команд поділяється на такі групи: арифметичні команди (*ADD*, *MULT*, *DIV*, *POW*, *INVM*, *CDP*, *CPD*, *INVA*, *SUB*, *INC*, *DEC*), команди пересилання даних (*MOV*, *LOAD*, *OUT*) та команди передачі керування (*JMP*, *LOOP*).

При створенні тексту програми програміст використовує, поряд з мнемонічними позначеннями команд Асемблера, службові слова – директиви компілятора: #GF, DATA, CODE, const, Array.

Під час компіляції заповнюється два види пам’яті: пам’ять команд та пам’ять даних. У відповідності до цього у структурі тексту програми виділяється дві секції: секція даних та секція коду.

Директива #GF дозволяє вказувати вид поля, що використовується, а саме #GF(p) та #GF(2^m). Замість значення *p* та *m* потрібно вказувати число в десятковій системі числення.

Якщо вказана директива #GF(p), то розрядність слова пам’яті даних обчислюється як логарифм двійковий від *p* з округленням до найближчого більшого цілого. Число *p* задається за допомогою десяткової константи. Наприклад #GF(13). Компілятором виконується перевірка чи є простим задане число *p*. Якщо число *p* не просте, то видається відповідна помилка (список повідомлень компілятора при трансляції тексту програми у машинний код наведено у Додатку Ж). Число *p* записується за нульовою адресою в пам’ять даних. Додатково компілятором контролюється переповнення розрядної сітки коли значення операндів перевищує величину $p - 1$. Наприклад, якщо $p = 13$, і ми спробуємо задати константу, що дорівнює 27, то отримаємо помилку.

Якщо вказана директива #GF(2^m), то розрядність слова пам’яті даних дорівнює *m*. Число *m* задається за допомогою десяткової константи. Наприклад #GF(2^10). Додатково компілятором контролюється переповнення розрядної сітки, а саме, щоб операнди не перевищували значення $2^m - 1$.

У секції DATA можна об’являти константи, змінні та масиви.

Приклад об’явлення констант:

```
const t = 5
const pow_A = 6
const pow_B = 5
```

Приклад об’явлення змінних:

```
t = 5
pow_A = 6
pow_B = 5
```

Приклад об’явлення масивів:

```
Array A [ pow_A ] = ( 12, 15, 1, 0, 7, 11 )
Array B [ pow_B ] = ( 7, 1, 4, 3, 10 )
Array C [pow_A + pow_B ] = ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
```

В квадратних дужках вказується кількість елементів масиву, в круглих дужках значення елементів. Нумерація елементів масиву виконується з нуля, наприклад довжина останнього масиву дорівнює pow_A + pow_B, тому перший елемент знаходиться за індексом 0, а останній за індексом pow_A + pow_B – 1.

Коли кількість вказаних значень елементів масиву не відповідає задекларованій довжині масиву, то масив автозаповнюється нулями, наприклад:

```
Array C [pow_A + pow_B ] = ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
```

тотожне

```
Array C [pow_A + pow_B ] = ( )
```

Також можна частково вказувати значення елементів:

```
Array B [ 7 ] = ( 15, 1 )
```

тоді нульовий і перший елемент заповнюється значенням 15 та 1 відповідно, а інші елементи автозаповнюються нулями.

При заданні початкових значень констант та змінних в секції DATA можна використовувати арифметичні вирази.

Компілятор підтримує таку форму арифметичного виразу (інструкція з експлуатації розробленого інтегрованого середовища розробки “Асемблер Галуа” наведена у Додатку Є): арифметичні вирази з числами (числовими константами) та константами описаними у секції DATA, що подані у різних системах числення, а саме у десятковій, двійковій та шістнадцятковій. Також в якості виразу може виступати тільки одна змінна без залучення арифметичних операцій над нею та констант. Це обмеження пов’язане з тим, що компілятор додає у відповідну команду, де присутній вираз, адресу в пам’яті даних за якою записує результат обчислення цього виразу, а якщо

там буде присутня змінна, то значення цього виразу буде змінюватись в ході виконання програми, але в пам’яті даних не буде відбуватись змін.

Числа у двійковій системі числення задаються таким чином

b’101 – число 5 у двійковій системі числення.

Числа у шістнадцятковій системі числення задаються таким чином

h’F – 15 у шістнадцятковій системі числення (літери у шістнадцятковій системі числення мають бути великими).

Наприклад, можна писати такі вирази

```
pow_B * r / b'11 + 1 + h'F
```

де pow_B та r константи описані в секції DATA, b'11 – число записане у двійковому вигляді, 1 – число записане у десятковому вигляді, h'F – число записане у шістнадцятковому вигляді. Якщо pow_B = 2 та r = 27, то вираз буде дорівнювати 34. При реалізації операції ділення, якщо отримується дробове число, то береться ціла частина від цього числа, тобто виконується округлення до найменшого цілого. Використання дужок в арифметичних виразах не підтримується.

При заданні кількості елементів масиву в секції DATA також можна використовувати арифметичний вираз, але в якості операндів вираз може містити тільки константи описані в секції DATA та числа у десятковій системі числення.

У команді MOV R, M операнд M має бути константою або змінною, що описана в секції DATA. У команді MOV M, R операнд M має бути змінною, що описана в секції DATA. Арифметичні вирази та задання чисел в явному вигляді в цій команді не підтримується. Запис даних з регістру в константу не підтримується, оскільки константу змінювати не можна.

У командах LOOP та LOAD в якості операнду val може виступати будь-який арифметичний вираз, що підтримується компілятором.

При роботі з масивом у командах MOV та OUT зміщення має задаватись явним чином у вигляді числа в десятковій системі числення. Якщо зміщення не задано, то воно сприймається як нульове.

Наприклад:

```
MOV  C [CA_3], R2
```

інтерпретується компілятором як

```
MOV  C [CA_3 + 0], R2
```

При роботі з командою JMP використовуються мітки в тексті програми, а саме перехід за міткою, що розміщена в тексті програми. Мітки в тексті програми потрібно задавати в окремому рядку, тобто в цьому рядку має бути тільки мітка та дві крапки і нічого більше.

Наприклад,

```
L:
LOAD    CA_2, b'000000000
JMP R0, L
```

Коментарі в Асемблері процесора Галуа підтримуються:

- ✓ однорядкові (за допомогою символів // закоментований текст)
- ✓ багаторядкові (/* закоментований текст */)

Компілятор процесора Галуа надає опціональну можливість задавати незвідний поліном для поля $GF(2^m)$. В тому випадку коли використовується, для цього поля, АЛП з табличним виконанням операцій у полі, то значення незвідного поліному вже враховано при побудові таблиці елементів, яка записана в ROM і задавати явно незвідний многочлен не потрібно. Якщо використовується АЛП з алгоритмічним виконанням операцій у полі, то виникає необхідність виконувати приведення за модулем незвідного многочлена і цей многочлен доцільно задавати в тексті програми. Це можна зробити за допомогою необов'язкової директиви після задання поля, наприклад

```
#GF(2^15)
#число у десятковій або у двійковій, або у шістнадцятковій системі числення.
```

При компіляції, якщо задане значення незвідного поліному, то воно записується за нульовою адресою у пам'ять даних.

Приклади програм мовою Асемблера Галуа наведено у Додатку К.

4.3. Структура апаратних засобів для реалізації системи команд

Після розроблення системи команд процесора необхідно розробити апаратні засоби для реалізації системи команд, яка б забезпечувала прискорення обчислень для класу задач, на які орієнтований даний процесор.

Розрізняють фоннеймановську та гарвардську архітектуру процесорів [22, 27]. Основна відмінність цих архітектур полягає в тому, що у фоннеймановській архітектурі програма та дані знаходяться у спільній пам’яті, доступ до якої здійснюється по одній шині даних і команд, а в гарвардській архітектурі пам’ять програм та даних розділені та мають окремі шину даних і шину команд.

Оскільки розрядність слова команди та слова даних зазвичай є різною (розрядність слова команди не залежить від виду поля Галуа, а розрядність слова даних доцільно обирати мінімально можливою для розміщення довільного елемента відповідного поля Галуа), то для побудови процесора Галуа було обрано гарвардську архітектуру, що дозволить підвищити швидкодію процесора.

Розширення центрального процесора спеціалізованим процесором Галуа має за мету орієнтувати його на виконання операцій в скінченних полях та підвищити продуктивність системи. Процесор при цьому залишається універсальним, однак його продуктивність при виконанні операцій в скінченних полях, на які він орієнтований, значно підвищується.

Два найбільш доцільних способи реалізації процесора Галуа полягають у наступному.

1. Реалізувати на ПЛІС.
2. Реалізувати як спеціалізований співпроцесор.

В першому випадку процесор Галуа можна реалізувати на ПЛІС та використовувати як зовнішній пристрій по відношенню до основного комп’ютера та центрального процесора.

Перші ПЛІС приєднувались до комп’ютера за допомогою порту *RS*-232, у сучасних системах для з’єднання ПЛІС з комп’ютером застосовується два варіанти (рис. 4.2).

1. З використанням кабеля *Ethernet*.
2. З використанням кабеля *USB*.

В обох випадках для реалізації фізичної взаємодії ПЛІС та центрального процесора (*Host*-комп’ютера) використовується, так зване, *software*-ядро (програма, як правило мовою *Verilog* або *VHDL*, яка розміщується (“прошивається”) на ПЛІС).

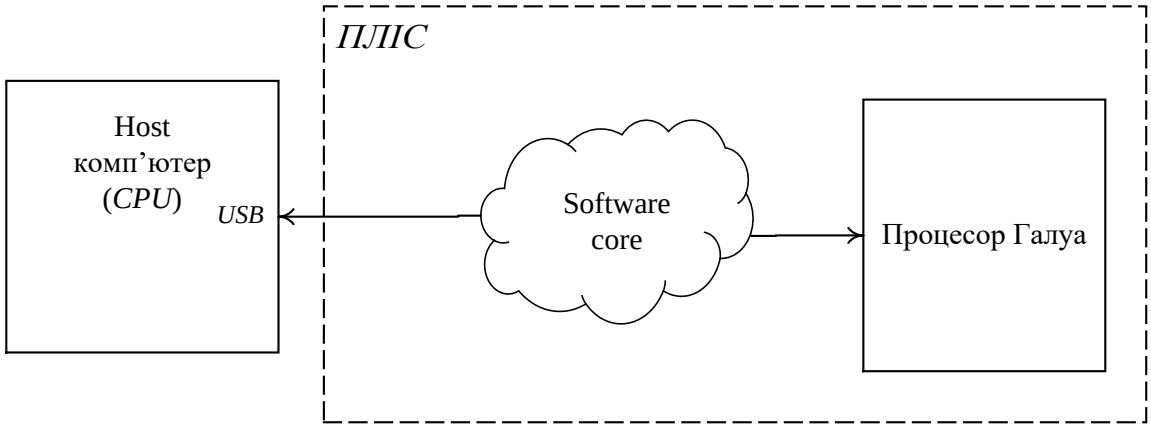


Рис. 4.2. Узагальнена схема фізичного з’єднання ПЛІС та центрального процесора

При реалізації процесора Галуа (рис. 4.3) як співпроцесора по відношенню до *CPU* необхідно в систему команд центрального процесора ввести спеціальні команди для виконання операцій у полі Галуа (табл. 4.3). При отриманні такої команди центральний процесор передає її на виконання співпроцесору Галуа. Наслідком цього має стати зростання продуктивності та ефективності обробки інформації.

Таблиця 4.3

Спеціальні команди центрального процесора

Мнемонічне позначення	Призначення команди
ADD	Додавання елементів поля $GF(p)$ або $GF(2^m)$
MUL	Множення елементів поля $GF(p)$ або $GF(2^m)$
DIV	Ділення елементів поля $GF(p)$ або $GF(2^m)$
POW	Піднесення до степеню елемента поля $GF(p)$ або $GF(2^m)$
INVM	Знаходження мультиплікативно оберненого елемента поля $GF(p)$ або $GF(2^m)$
INVA	Знаходження адитивно оберненого елемента поля $GF(p)$
CDP	Перетворення многочленного подання елемента поля $GF(2^m)$ у степеневе
CPD	Перетворення степеневого подання елемента поля $GF(2^m)$ у многочленне
SUB	Віднімання елементів поля $GF(p)$
MOV	Пересилання даних
LOAD	Завантаження лічильника адрес
INC	Інкремент значення в регістрі або лічильнику адрес
DEC	Декремент значення в регістрі або лічильнику адрес
JMP	Зміна природного порядку виконання програми
LOOP	Цикл з лічильником
OUT	Видача на вихід елемента

Процесор Галуа з’єднується з центральним процесором (*CPU*) за допомогою шини даних (ШД), адреси (ША) та керування (ШК). Для з’єднання може використовуватися також контролер переривань (КП) (рис. 4.3).

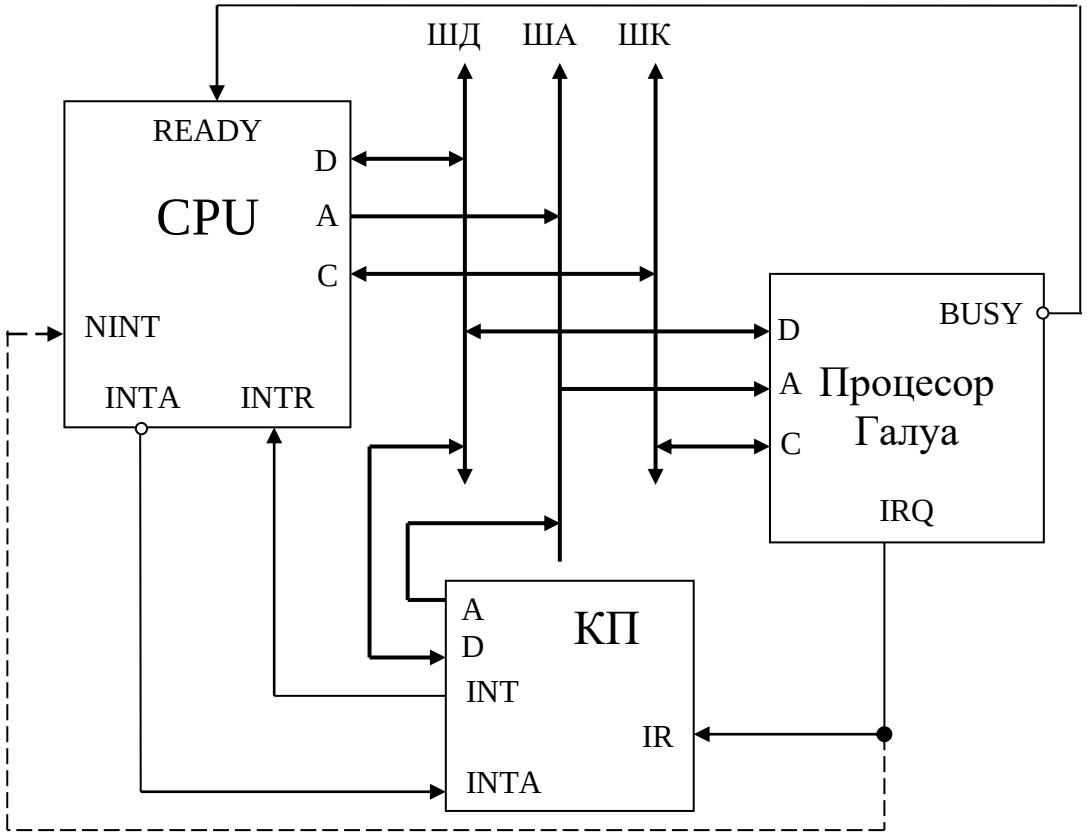


Рис. 4.3. Схема взаємодії *CPU* з процесором Галуа

CPU визначає приналежність команди за першим байтом цієї команди до системи команд процесора Галуа. В кожний момент часу *CPU* бере перший байт команди, співпроцесор так само отримує перший байт цієї ж команди. За першим байтом *CPU* та співпроцесор визначають тип команди, що надійшла. Якщо ця команда належить до команд співпроцесора, то наступні байти команди обробляються співпроцесором, інакше (якщо це звичайна команда *CPU*), то наступні байти цієї команди будуть проігноровані співпроцесором та будуть оброблятися *CPU*.

Як тільки в потоці команд з’являється одна із спецкоманд, співпроцесор змінює стан сигналу *busy* на нульовий і тримає нульовий рівень сигналу *busy* поки не виконається відповідна команда, після чого змінює сигнал *busy* на одиничний і це означає, що співпроцесор готовий прийняти наступну команду на виконання. Коли чергова команда надходить на виконання до процесора Галуа, обидва процесори здійснюють

паралельну роботу. Співпроцесор виконує дії, визначені спецкомандою, а центральний процесор продовжує виконання програми. При цьому шиною керує центральний процесор, і кожного разу, коли співпроцесору необхідно звернутися до пам’яті, він здійснює запит шини. Якщо центральному процесору для подальших обчислень необхідні результати виконання команди співпроцесора, то він переходить в режим очікування та неперервно опитує сигнал *busy* від процесора Галуа.

Процесор Галуа може розглядатися *CPU* як зовнішній пристрій. У такому випадку процесор Галуа формує запит на переривання (з виходу IRQ), який надходить на один з входів фіксації запитів на переривання (IR). Далі КП визначає вектор переривання процесора Галуа та формує вимогу переривання INT, яка надходить на вхід INTR центрального процесора. У відповідь *CPU* формує сигнал підтвердження переривання $\overline{\text{INTA}}$, за яким КП формує команду CALL виклику підпрограми обслуговування переривання (код операції команди видається на ШД, а адреса підпрограми – на ША).

Взаємодія процесора Галуа та *CPU* може відбуватися й без залучення КП. У цьому випадку використовується вхід NINT немаскованого переривання.

Якщо від процесора Галуа надходить запит на переривання на вхід NINT, то він розглядається центральним процесором як невідкладний, тобто такий, що потребує негайної реакції.

Запит NINT має більш високий пріоритет, ніж запит, що надходить на вхід INTR. Цей запит є асинхронним щодо *CPU*.

Вибирати команди з пам’яті програм (на рис. 4.4 не показана) може лише центральний процесор.

Узагальнена структура процесора Галуа (рис. 4.4-4.5) містить пам’ять команд (ПК-д), де зберігається бітовий код програми, пам’ять даних (ПД), де зберігаються вхідні дані, а також операційний (ОА) та керуючий

автомат (КА). Пам’ять команд та даних являють собою запам’ятовувальний пристрій (*RAM*). Для керування потоками даних між структурними елементами процесора Галуа використовується шість мультиплексорів. При побудові процесора Галуа було виконано поєднання гарвардської та *RISC*-архітектури (рис. 4.6).

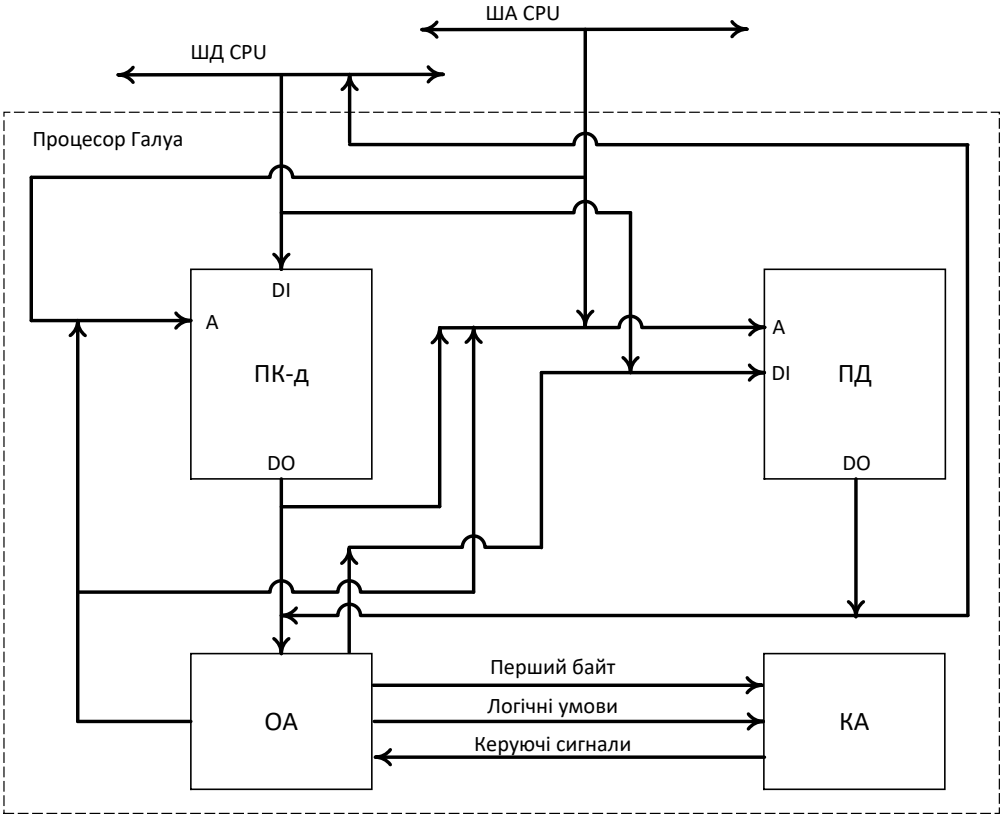


Рис. 4.4. Узагальнена структура процесора Галуа

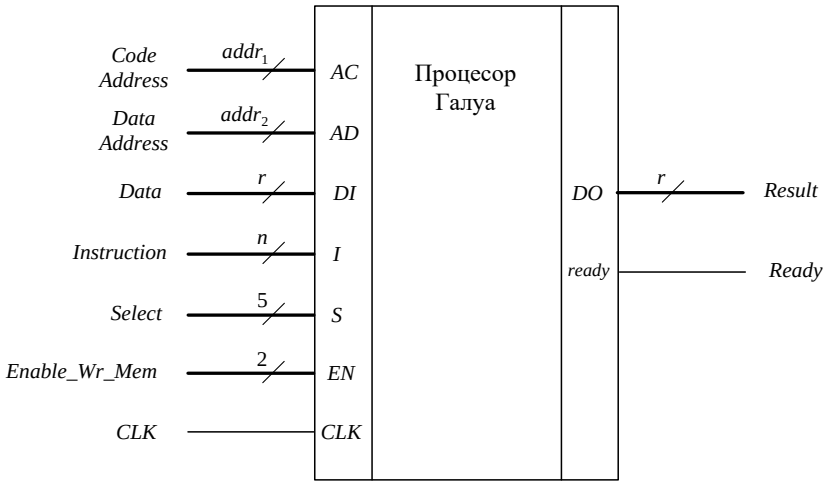


Рис. 4.5. Умовне графічне позначення процесора Галуа

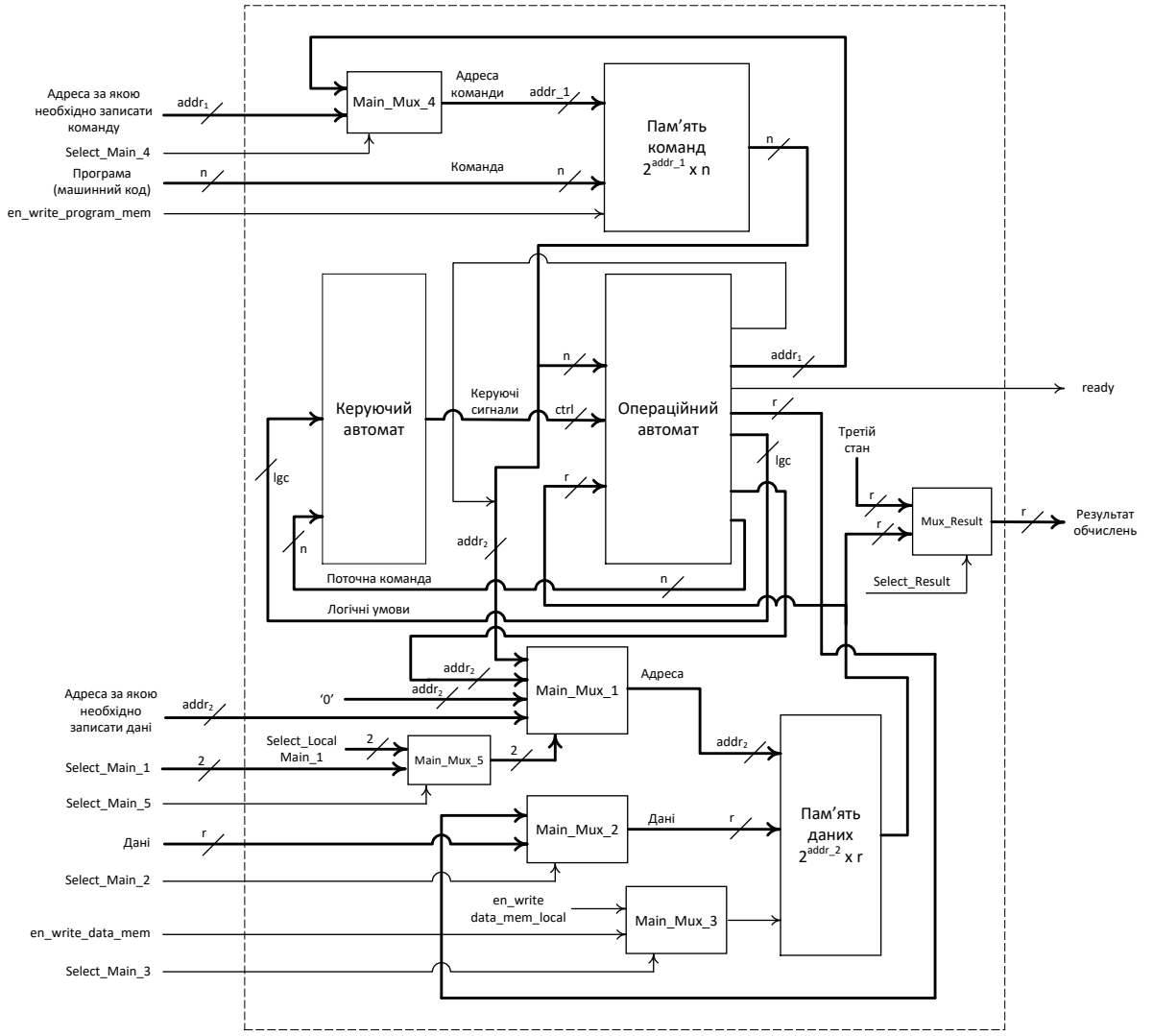


Рис. 4.6. Функціональна схема процесора Галуа

Узагальнено функціонування процесора Галуа полягає в наступному: спочатку програмний код та дані надходять з центрального процесора до пам'яті команд та даних відповідно. Сигнали керування мультиплексорами та сигнали дозволу запису в пам'ять команд та даних формуються керуючим автоматом. Коли команда (програма) та дані записані у пам'ять, починається виконання програми шляхом надходження відповідних керуючих сигналів від блока керування до операційного автомата, далі операційний автомат формує значення логічних умов, які надходять на входи керуючого автомата, аналізуючи які керуючий автомат формує сигнали керування компонентами операційного автомата.

Команда подається в процесор побайтово (починаючи зі старшого байта) залежно від довжини[1].

Для апаратної реалізації процесора Галуа рекомендовано використовувати такі значення розрядності: розрядність адреси пам'яті команд $addr_1 = 9$, розрядність одного слова команди $n = 8$, розрядність адреси пам'яті даних $addr_2 = 9$, розрядність одного слова даних r дорівнює $\lceil \log_2 p \rceil$, якщо обрано поле $GF(p)$ та $r = m$, якщо обрано поле $GF(2^m)$, розрядність шини сигналів керування (*ctrl*) складає 65 біт, розрядність шини логічних умов *lgc* = 9 біт.

Розглянемо докладніше структуру операційного автомата (рис. 4.7). Схемам, що наведені на рис. 4.6 та 4.7, відповідають функціональні схеми (додаток Д), отримані у середовищі розробки *Xilinx ISE* та за допомогою програми *Mentor Graphics Precision*. Тестування процесора Галуа виконувалось за допомогою програми *Mentor Graphics ModelSim*. Часові діаграми наведено у Додатку Л.

Розглянемо функціональне призначення вузлів з яких складається операційний автомат процесора Галуа.

- Мультиплексори регулюють напрями передачі даних, а саме мультиплексор *MUX*_1 визначає джерело даних для блока регістрів загального призначення, мультиплексор *MUX*_2 визначає джерело даних для блока лічильників адрес, мультиплексор *MUX_CA* визначає джерело даних для першого входу суматора адрес.
- Лічильник команд (*Program Counter*) формує адресу наступної команди у пам'яті команд, яка зберігається в регістрі адрес команд.
- Регістр команд (*Instruction Register*) використовується для зберігання команди, що зчитана з пам'яті команд.

[1] Якщо $n \neq 8$, то команда буде подаватись словами, кожне з яких матиме довжину n.

- Регістр модуля використовується для зберігання значення модуля p або значення незвідного многочлена.
- Регістр адрес даних використовується коли виконується трибайтна команда, пов’язана з обробкою даних в масиві, оскільки адреса елемента масиву обчислюється як базова (початкова) адреса масиву + значення у відповідному лічильнику адрес + зміщення (базова індексна адресація), базова адреса масиву формується конкатенацією бітів з першого байту та другого байту команди та записується в регістр адрес даних, а зміщення зчитується безпосередньо з регістра команд.
- Блок лічильників циклів призначений для апаратної реалізації команди *LOOP*, а саме при дешифруванні команди *LOOP*, з відповідної адреси пам’яті даних переписується значення кількості повторень циклу у відповідний лічильник, і далі лічильник працює в декрементному режимі.
- Блок перевірки на рівність нулю значень лічильників кількості повторень циклу, видає на вихід чотирибітове слово, яке сигналізує про рівність нулю відповідного лічильника циклу (*i*-й біт відповідає за *i*-й лічильник).
- Блок лічильників адрес використовується для роботи з масивами, а суматор адрес використовується для обчислення адреси елемента масиву. Адреса елемента масиву обчислюється шляхом додавання трьох компонент: початкової адреси масиву, індекса (значення у відповідному лічильнику адрес циклу) та зміщення зі знаком плюс або мінус (значення зміщення за абсолютною величиною не повинно перевищувати 15).
- Арифметико-логічний пристрій (АЛП) призначений для виконання алгебраїчних операцій.

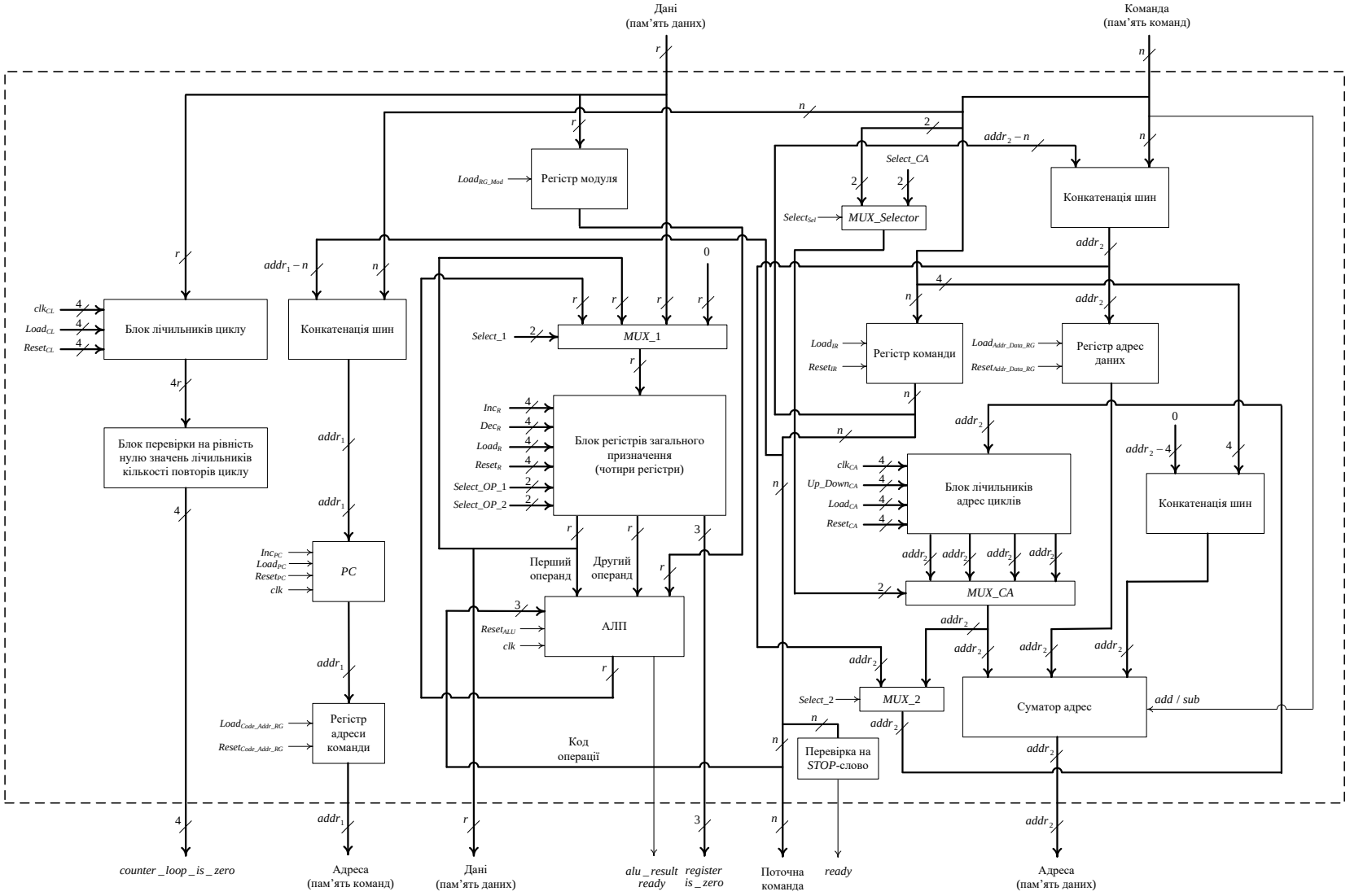


Рис. 4.7. Структурна схема операційного автомата процесора Галуа

В якості арифметико-логічного пристрою (АЛП) може виступати блок виконання операцій у полі Галуа виду $GF(p)$ або $GF(2^m)$. Причому для поля $GF(2^m)$ пропонується два різновиди АЛП, а саме АЛП, що ґрунтується на табличному зберіганні елементів поля $GF(2^m)$ та АЛП, що ґрунтується на алгоритмічному виконанні операцій в полі $GF(2^m)$. Регістр модуля використовується для зберігання значення модуля p, якщо в процесорі використовується АЛП реалізації операцій в полі $GF(p)$, та значення незвідного многочлена, якщо використовується АЛП реалізації операцій в полі $GF(2^m)$ на основі алгоритмічного підходу виконання операцій в цьому полі. При використанні АЛП, що ґрунтується на табличному зберіганні елементів поля $GF(2^m)$, немає потреби зберігати у регістр модуля значення незвідного многочлена, оскільки при побудові таблиці елементів поля враховується значення обраного незвідного многочлена, тому в такому випадку регістр модуля не використовується в обчисленнях. Регістр модуля заповнюється з пам'яті даних. Значення модуля p та незвідного многочлена знаходиться в пам'яті даних за нульовою адресою, для випадку АЛП, що ґрунтується на табличному зберіганні елементів поля $GF(2^m)$ значення незвідного многочлена у пам'ять даних не записується. АЛП, в якості джерела операндів використовує в регістри загального призначення. Результат обчислень АЛП, через мультиплексор *MUX*_1, записується у регістр, що є першим операндом в команді, а далі може бути записаний у пам'ять даних.

Блок регістрів загального призначення (рис. 4.8) складається з чотирьох регістрів, двох мультиплексорів вибору джерела першого та другого операнду відповідно та блоку перевірки операндів на нуль. Блок перевірки операндів на нуль контролює лише перші три регістри та видає на вихід трирозрядний код, рівність 1 кожного розряду сигналізує про рівність

нулю відповідного регістра. Відсутність перевірки рівності нулю четвертого регістра пояснюється тим, що значення 11_2 в полі команд, яке відповідає за номер регістра, значення якого перевіряється на рівність нулю, зарезервоване для інших цілей.

Керування послідовністю дій та синхронізація виконується керуючим автоматом. Керуючий автомат керує станом АЛП та визначає значення сигналів керування функціональними вузлами. Основними функціями керуючого автомата є завантажувати дані в регістр, обирати шлях даних через мультиплексори, керувати процесом запису даних у пам’ять.

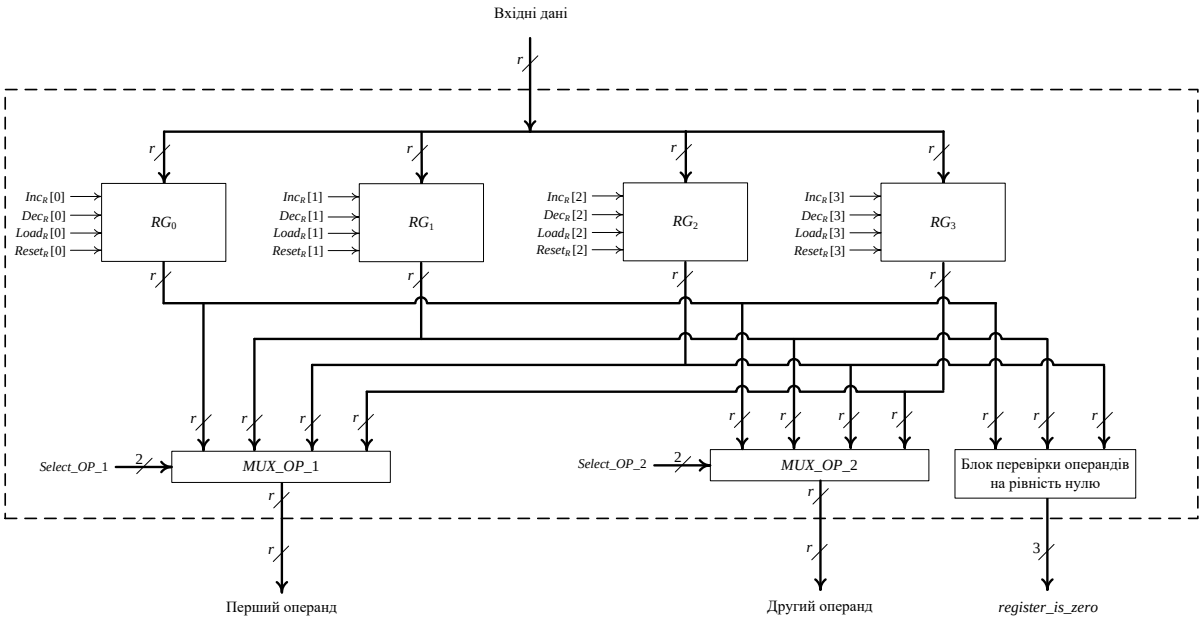


Рис. 4.8. Блок регістрів загального призначення

Вхідні сигнали керуючого автомата – це: синхросигнал, поточна команда, яка знаходиться в регістрі команд та слово значень логічних умов, довжиною *lgc* = 9 біт. Слово значень логічних умов має наступні поля

counter_loop_is_zero	*alu_result_ready*	*register_is_zero*	*ready*
4 біта	1 біт	3 біта	1 біт

Одиничне значення одного з чотирьох бітів поля *counter_loop_is_zero* вказує на нульове значення відповідного лічильника кількості повторень циклу. Поле *alu_result_ready* вказує на те, що поточна команда виконана в АЛП і результат знаходиться на виході АЛП. Одиничне значення одного з

трьох бітів поля *register_is_zero* вказує на нульове значення відповідного регістра загального призначення. Поле *ready* вказує на те, що черговою командою є *STOP*-слово і виконання програми записаної у пам’ять програм завершено.

Керуючі сигнали, що є вихідними сигналами керуючого автомата утворюють слово довжиною 65 біт. Дане слово містить зазначені в табл. 4.4 поля.

Таблиця 4.4

Сигнали керуючого автомата

Призначення поля сигналів	Назва поля сигналу	Кількість сигналів
Скидання АЛП	$Reset_{ALU}$	1
Керування регістрами загального призначення	$Reset_R$	4
	$Load_R$	4
	Inc_R	4
	Dec_R	4
Керування лічильником команд	$Load_{PC}$	1
	Inc_{PC}	1
Керування внутрішніми мультиплексорами	*Select_OP_*1	2
	*Select_OP_*2	2
	*Select_*1	2
	*Select_*2	1
	$Select_{Sel}$	1
	Select_CA	2
Керування регістром команд	$Load_{IR}$	1
	$Reset_{IR}$	1

Продовження табл. 4.4

Призначення поля сигналів	Назва поля сигналу	Кількість сигналів
Керування регістром адрес команд	$Load_{Code\_Addr\_RG}$	1
	$Reset_{Code\_Addr\_RG}$	1
Керування регістром адрес даних	$Load_{Code\_Data\_RG}$	1
	$Reset_{Code\_Data\_RG}$	1
Керування регістром модуля	$Load_{RG\_Mod}$	1
Керування лічильником кількості повторів циклу	clk_{CL}	3
	$Reset_{CL}$	3
	$Load_{CL}$	3
Керування лічильником адрес циклів	clk_{CA}	4
	$Reset_{CA}$	4
	$Load_{CA}$	4
	Up_down_{CA}	4
Сигнали керування зовнішніми мультиплексорами	*Select_Local_Main*_1	2
	Select_Result	1
Дозвіл запису у пам’ять даних	*en_write_data_mem_local*	1

На рис. В.12 – В.21 (додаток В) наведено алгоритми роботи керуючого автомата при виконанні кожної команди.

Розглянемо цикл виконання трибайтної команди *MOV* запису даних з елемента масиву в регістр (рис. В.15).

1. Виконується завантаження адреси з *PC* у регістр адреси команди.
2. Інкремент значення в *PC* та завантаження першого байту команди у регістр команди.

3. Керуючий автомат процесора Галуа аналізує вміст регістра команди та переходить в режим виконання трибайтної команди *MOV*.
4. Виконується завантаження адреси з *PC* у регістр адреси команди.
5. Інкремент значення в *PC* та завантаження базової адреси масиву у регістр адрес даних. Базова адреса масиву складається з молодшого біта значення з регістра команди та другого байту команди, що знаходиться на виході пам’яті команд.
6. Виконується завантаження адреси з *PC* у регістр адреси команди.
7. Інкремент значення в *PC*. Аналіз четвертого біта третього байту команди (третій байт команди знаходиться на виході пам’яті команд та дані аналізуються безпосередньо з виходу пам’яті команд).
8. Залежно від значення четвертого біта третього байта виконується запис з регістра в елемент масиву (з регістру в пам’ять даних), або з елемента масиву в регістр (з пам’яті даних у регістр). Ми розглядаємо випадок запису даних з елемента масиву в регістр. Адреса елемента масиву в пам’яті даних обчислюється як сума значення базової адреси масиву (знаходиться у регістрі адрес даних), значення в лічильнику адрес циклів (номер лічильника адрес циклів вказується в третьому байті команди та зчитується безпосередньо з пам’яті команд) плюс зміщення (чотири розряди в третьому байті команди, зчитується безпосередньо з пам’яті команд). Номер регістра, в який потрібно записати дані з елемента масиву (пам’яті даних), знаходиться в першому байті команди та зчитується з регістра команд (в регістрі команд знаходиться перший байт команди).

4.4. Висновки до розділу 4

1. Розроблено архітектуру спеціалізованого процесора, що орієнтований на виконання обчислень у полях Галуа. Особливістю архітектури є те, що, не змінюючи інтерфейсу процесора, шляхом зміни АЛП, можна здійснити перехід до поля Галуа $GF(p)$ або до поля $GF(2^m)$.
2. Процесор Галуа можна використовувати в універсальній обчислювальній системі як співпроцесор, доповнюючи систему команд центрального процесора, або як спецобчислювач на основі ПЛІС, який порівняно з універсальними обчислювальними засобами підвищує продуктивність обробки інформації в реальному часі.
3. Запропонована система команд дозволяє створювати програми мовою Асемблера процесора Галуа з подальшим виконання цих програм функціональними блоками розробленого процесора.
4. Розроблено програмістську модель процесора Галуа, яка дозволяє створювати програмне забезпечення довільної складності мовою Асемблера процесора Галуа.
5. Комп'ютерне моделювання обчислювальних процесів, що мають місце при реалізації операцій у скінченних полях, показало, що продуктивність системи на основі співпроцесора Галуа порівняно з універсальною обчислювальною системою зростає на 27%, залежно від команд, що використовуються в тексті програми мовою Асемблера процесора Галуа.

РОЗДІЛ 5
АНАЛІЗ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ МЕТОДІВ РЕАЛІЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ ОПЕРАЦІЙ У СКІНЧЕННИХ ПОЛЯХ

5.1. Дослідження запропонованої модифікації методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном

Запропоновану модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном доцільно порівнювати з класичним методом за кількістю операцій множення. Аналізуючи зменшення кількості операцій множення (табл. 5.1), яке забезпечує запропонована модифікація методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном, порівняно з класичним методом, бачимо, що для довжини показника степеня до 128 біт оптимальною кількістю елементів таблиці передобчислень запропонованого методу є 7, для довжини показника степеня від 256 біт до 1024 біт оптимальним значенням є 31 елемент таблиці передобчислень, для довжини показника степеня від 2048 біт до 8192 біт оптимальним значенням є 63 елементи таблиці передобчислень.

Під час дослідження методів піднесення до степеня методи було розділено на дві групи: методи, що розглядають по одному розряду показника степеня (табл. 5.2) у двійковій і симетричній трійковій системі числення, та методи, що розглядають по кілька розрядів (табл. 5.3) за одну ітерацію (віконні методи).

З метою проведення експериментальних досліджень (заміру часових показників) розроблено програмний продукт на мові програмування C# у середовищі розробки “Visual Studio 2015”. Експериментальне дослідження проводилося на віртуальній машині з операційною системою Windows 10 Education, об’ємом оперативної пам’яті 8 Gb, яка встановлена на сервері з

такими технічними характеристиками: процесор Intel Core i7-3930K CPU 3.20GHz (Processor Sockets 1, Processor Cores per Socket 6, Logical Processors 12), об’єм оперативної пам’яті 32 Gb.

Таблиця 5.1

Порівняння
модифікованого методу піднесення до степеня
та класичного методу

Довжина показника степеня, біт	Кількість елементів таблиці передобчислень				
	3	7	31	63	127
32	1.0	3.0	1.0	1.0	0.5
64	1.2	3.7	2.0	2.0	1.0
128	1.3	1.5	1.4	0.7	0.5
256	1.2	1.4	1.8	1.4	0.8
384	1.1	1.7	1.5	0.7	0.9
512	1.1	2.3	5.1	3.6	3.0
1024	1.2	6.1	10.4	6.9	8.3
2048	1.1	10.6	18.8	24.9	22.6
4096	1.0	25.2	42.8	57.9	40.3
8192	1.2	51.9	85.5	113.7	95.6

Для проведення експериментального дослідження сформовано три множини чисел довжиною 2048 біт: множина основ, множина показників степеня та множина модулів. Потужність кожної множини становить 10 елементів. Декартовий добуток елементів трьох сформованих множин дає 1000 трійок різних чисел довжиною 2048 біт, що є набором операндів для проведення заміру часових характеристик методів. Після виконання

обчислень отримані часові результати були усереднені, щоб уникнути впливу операндів з особливою структурою на результат експерименту (табл. 5.2-5.3).

Таблиця 5.2

Порівняння методів піднесення до степеня елемента поля $GF(p)$ для значень p довжиною 2048 біт за часовими показниками

Метод	Час, мс
Бінарний *RL*	128,588
Бінарний *LR*	128,976
NAF бінарний *RL*	535,926
NAF бінарний *LR*	111,132
Додавань-віднімань *RL*	539,815
Додавань-віднімань *LR*	114,489
Джої *RL*	164,562
Монтгомері *LR*	164,568

Отримані експериментальні результати (табл. 5.2-5.3) показують, що для довжини показника степеня 2048 біт запропонована модифікація показує приріст швидкодії на 8% порівняно з найкращим з існуючих методів.

Проведені дослідження для довжин операндів від 128 до 8192 біт показали, що запропонована модифікація методу забезпечує приріст швидкодії на 7-9 %.

Таблиця 5.3

Порівняння віконних методів піднесення до степеня елемента поля *GF*(*p*) для значень *p* довжиною 2048 біт за часовими показниками, мс

Метод	Максимально допустима довжина вікна *w*, біт									
	2	3	4	5	6	7	8	9	10	11
Віконний *RL*	116,065	108,201	103,710	101,168	100,490	101,553	106,075	117,321	141,419	191,978
Віконний *LR*	113,938	107,812	103,226	100,700	99,845	100,564	104,044	114,188	135,340	180,147
Віконний *RL* з ковзним вікном	111,066	104,292	100,212	98,578	98,213	99,369	103,768	113,787	135,475	179,125
Віконний *LR* з ковзним вікном	111,349	104,598	100,775	98,031	96,960	96,745	98,233	102,878	113,002	134,943
NAF віконний *RL* з ковзним вікном	111,564	103,072	101,074	98,682	98,512	101,152	106,806	119,475	145,193	196,757
NAF віконний *LR* з ковзним вікном	111,832	105,459	105,180	110,052	121,404	148,757	206,151	312,902	532,572	984,993
wNAF віконний *RL*	112,838	106,103	101,784	99,506	99,003	99,930	103,977	113,009	132,206	172,994
wNAF віконний *LR*	112,200	107,745	105,215	107,579	115,886	134,755	175,227	257,418	422,178	752,088
Запропонована модифікація методу з ковзним вікном	113,501	108,157	97,151	90,469	89,438	91,211	92,910	97,567	109,471	121,705

Кожен з розглянутих методів піднесення до степеня у полі *GF*(*p*) можна реалізувати із застосуванням арифметики Монтгомері (табл. 5.4-5.5). Отримані експериментальні результати показують, що застосування арифметики Монтгомері не дає приросту швидкодії.

Таблиця 5.4

Порівняння методів піднесення до степеня елемента поля $GF(p)$ для значень p довжиною 2048 біт за часовими показниками з використанням арифметики Монтгомері

Метод	Час, мс
Бінарний *RL*	146,591
Бінарний *LR*	146,641
NAF бінарний *RL*	572,062
NAF бінарний *LR*	134,941
Додавань-віднімань *RL*	574,822
Додавань-віднімань *LR*	138,206
Джої *RL*	203,292
Монтгомері *LR*	203,825

Таблиця 5.5

Порівняння віконних методів піднесення до степеня елемента поля $GF(p)$ для значень p довжиною 2048 біт за часовими показниками з використанням арифметики Монтгомері, мс

Метод	Максимально допустима довжина вікна w, біт									
	2	3	4	5	6	7	8	9	10	11
Віконний *RL*	134,563	127,102	122,649	120,648	120,742	124,822	135,791	159,787	209,984	314,341
Віконний *LR*	134,144	126,638	121,580	118,735	117,558	118,156	122,893	134,613	158,374	209,039
Віконний *RL* з ковзним вікном	130,553	122,746	118,800	116,838	116,960	120,046	127,710	145,045	181,669	256,114
Віконний *LR* з ковзним вікном	130,569	122,101	118,087	115,637	114,109	113,906	115,888	121,253	135,194	158,240
NAF віконний *RL* з ковзним вікном	132,735	121,775	119,205	117,618	118,779	122,661	133,194	155,822	201,779	295,808
NAF віконний *LR* з ковзним вікном	131,607	123,730	123,571	128,329	139,870	167,996	223,171	337,366	576,234	1026,205
wNAF віконний *RL*	131,789	124,090	121,101	118,237	118,090	120,641	128,144	144,883	179,268	248,721
wNAF віконний *LR*	131,745	125,603	123,189	125,246	133,841	153,550	194,955	278,454	450,030	792,315
Запропонована модифікація методу з ковзним вікном	129,915	117,116	110,153	105,178	103,995	103,055	107,051	112,291	124,187	147,159

5.2. Дослідження операції обчислення мультиплікативно оберненого елемента у скінченному полі $GF(p)$

Для проведення дослідження швидкості роботи методів обчислення мультиплікативно оберненого елемента, що ґрунтуються на модулярному піднесенні до степеня (табл. 5.6) було сформовано 2 множини з парами випадкових чисел m та b. Кожна множина містить 100 непарних модулів m певної довжини (8, 16 біт) та 250 чисел b для кожного m. У сформованих множинах числа m та b є взаємнопростими, оскільки це є умовою існування мультиплікативно оберненого елемента.

Отримані експериментальні результати показують, що серед методів обчислення мультиплікативно оберненого елемента, які ґрунтуються на модулярному піднесенні до степеня, найкращі часові показники дають методи, що базуються на формулі Кармайкла.

Таблиця 5.6

Порівняння алгоритмів обчислення мультиплікативно оберненого елемента, що ґрунтуються на модулярному піднесенні до степеня за часовими показниками, мс

Назва методу	Довжина модуля, біт	
	16	24
Метод Ейлера	10,925	3119,554
Метод Кармайкла	3,747	442,817
Метод Аразі (на основі формули Ейлера)	10,919	3131,802
Метод Аразі (на основі формули Кармайкла)	3,729	441,753

З метою проведення дослідження швидкості роботи методів обчислення мультиплікативно оберненого елемента, що ґрунтуються на пошуку НСД двох чисел за алгоритмом Евкліда, було сформовано 3 групи по 7 множин з парами випадкових чисел m та b. Перша група містить довільні модулі (табл. 5.7), друга – модулі, що є непарними числами (табл. 5.8), третя – модулі, що є простими числами (табл. 5.9).

Кожна множина містить 50 модулів m фіксованої довжини (128, 256, 512, 1024, 2048, 4096, 8192 біт) та 100 чисел b для яких існує мультиплікативно обернений елемент за модулем m.

Таблиця 5.7

Порівняння часу роботи методів, що ґрунтуються на пошуку НСД для довільного модуля, мс

Назва методу	Довжина модуля, біт						
	128	256	512	1024	2048	4096	8192
Розширений алгоритм Евкліда	0,047	0,108	0,291	0,609	1,626	4,669	15,507
Модифікація Бредлі розширеного алгоритму Евкліда	0,036	0,082	0,223	0,478	1,289	3,673	12,708
Удосконалена модифікація Бредлі розширеного алгоритму Евкліда	0,035	0,079	0,219	0,468	1,248	3,532	12,163
Розширений *RS*-бінарний алгоритм Евкліда	0,172	0,422	1,299	3,442	11,479	39,619	155,181
Розширений плюс-мінус алгоритм	0,293	0,711	2,018	5,487	18,074	61,606	239,408
Модифікований розширений плюс-мінус алгоритм	0,267	0,671	1,845	5,474	18,474	63,961	250,611
Розширений алгоритм Нортона	0,197	0,502	1,390	4,294	14,767	52,601	205,945
Модифікований розширений алгоритм Нортона	0,144	0,349	0,913	2,689	8,867	30,419	117,740
RS-бінарний алгоритм без множення	0,149	0,379	1,079	3,459	12,134	43,709	172,617
Модифікація Ласло Харса *RS*-бінарного алгоритму без множення	0,143	0,368	1,055	3,418	11,957	43,124	170,380
Подвійний *RS*-бінарний плюс-мінус алгоритм без множення	0,129	0,321	0,899	2,839	9,814	34,983	137,693
RS-бінарний алгоритм Ласло Харса із затримкою ділення на 2 без множення	0,182	0,471	1,372	4,451	15,702	56,787	224,871
RS-бінарний плюс-мінус алгоритм із затримкою ділення на 2 без множення	0,160	0,393	1,080	3,330	11,289	39,721	155,134
LS-бінарний алгоритм без множення	3,210	15,089	69,890	329,606	1666,392	8846,481	57129,350
Модифікація Ласло Харса *LS*-бінарного алгоритму без множення	3,847	15,073	69,719	329,945	1671,481	8809,356	56892,530
SE-алгоритм без множення	0,121	0,314	0,892	2,812	11,549	40,128	149,262
Модифікація Ласло Харса *SE*-алгоритму без множення	0,284	0,675	1,703	4,738	16,849	54,899	198,133

Таблиця 5.8

Порівняння часу роботи методів, що ґрунтуються на пошуку НСД для непарного модуля, мс

Назва методу	Довжина модуля, біт						
	128	256	512	1024	2048	4096	8192
Модифікація Майкла Пенка розширеного *RS*-бінарного алгоритму Евкліда для непарних модулів	0,128	0,306	0,877	2,382	7,836	27,418	104,372
Модифікація розширеного *RS*-бінарного алгоритму Евкліда для непарних модулів	0,120	0,297	0,839	2,370	7,928	28,038	107,309
Розширений алгоритм Евкліда	0,046	0,108	0,287	0,607	1,631	4,672	15,784
Модифікація Бредлі розширеного алгоритму Евкліда	0,035	0,081	0,221	0,475	1,294	3,805	12,967
Удосконалена модифікація Бредлі розширеного алгоритму Евкліда	0,034	0,079	0,217	0,468	1,254	3,631	12,379
Розширений *RS*-бінарний алгоритм Евкліда	0,172	0,421	1,274	3,451	11,536	40,879	157,181
Розширений плюс-мінус алгоритм	0,292	0,715	1,906	5,477	18,142	63,426	242,564
Модифікований розширений плюс-мінус алгоритм	0,269	0,670	1,811	5,475	18,495	65,998	253,397
Розширений алгоритм Нортона	0,197	0,500	1,382	4,309	14,803	54,135	207,078
Модифікований розширений алгоритм Нортона	0,143	0,347	0,910	2,693	8,910	31,352	118,719
RS-бінарний алгоритм без множення	0,147	0,377	1,077	3,466	12,158	45,016	173,501
Модифікація Ласло Харса *RS*-бінарного алгоритму без множення	0,143	0,366	1,054	3,398	12,003	44,392	171,092
Подвійний *RS*-бінарний плюс-мінус алгоритм без множення	0,129	0,321	0,898	2,836	9,846	36,050	138,552
RS-бінарний алгоритм Ласло Харса із затримкою ділення на 2 без множення	0,183	0,471	1,370	4,453	15,752	58,508	226,148
RS-бінарний плюс-мінус алгоритм із затримкою ділення на 2 без множення	0,159	0,392	1,074	3,323	11,307	40,925	155,989
LS-бінарний алгоритм без множення	3,211	15,078	69,826	329,677	1675,078	9165,202	58580,950
Модифікація Ласло Харса *LS*-бінарного алгоритму без множення	3,850	15,061	69,617	328,455	1668,000	9125,329	58321,130
SE-алгоритм без множення	0,120	0,314	0,890	2,814	11,567	41,358	149,878
Модифікація Ласло Харса *SE*-алгоритму без множення	0,283	0,674	1,700	4,743	16,896	56,692	199,093

Таблиця 5.9

Порівняння часу роботи алгоритмів, що ґрунтуються на на пошуку НСД для простого модуля, мс

Назва методу	Довжина модуля, біт						
	128	256	512	1024	2048	4096	8192
Модифікація Майкла Пенка розширеного *RS*-бінарного алгоритму Евкліда для непарних модулів	0,128	0,306	0,801	2,383	7,929	28,275	101,272
Модифікація розширеного *RS*-бінарного алгоритму Евкліда для непарних модулів	0,119	0,297	0,785	2,380	18,159	28,940	104,416
Розширений алгоритм Евкліда	0,046	0,108	0,248	0,607	1,637	4,978	15,175
Модифікація Бредлі розширеного алгоритму Евкліда	0,035	0,082	0,191	0,478	1,304	4,015	12,500
Удосконалена модифікація Бредлі розширеного алгоритму Евкліда	0,034	0,080	0,185	0,471	1,253	3,830	11,898
Розширений *RS*-бінарний алгоритм Евкліда	0,172	0,424	1,132	3,440	11,558	42,362	153,220
Розширений плюс-мінус алгоритм	0,291	0,715	1,851	5,469	18,159	65,609	236,071
Модифікований розширений плюс-мінус алгоритм	0,269	0,669	1,793	5,446	18,578	67,945	247,231
Розширений алгоритм Нортона	0,197	0,504	1,380	4,299	14,802	55,153	203,360
Модифікований розширений алгоритм Нортона	0,143	0,349	0,907	2,688	8,911	32,152	115,970
RS-бінарний алгоритм без множення	0,147	0,378	1,080	3,461	12,170	45,889	170,350
Модифікація Ласло Харса *RS*-бінарного алгоритму без множення	0,143	0,367	1,052	3,394	11,975	45,183	168,123
Подвійний *RS*-бінарний плюс-мінус алгоритм без множення	0,127	0,321	0,898	2,837	9,843	36,772	135,807
RS-бінарний алгоритм Ласло Харса із затримкою ділення на 2 без множення	0,181	0,471	1,367	4,449	15,733	59,562	221,983
RS-бінарний плюс-мінус алгоритм із затримкою ділення на 2 без множення	0,158	0,393	1,077	3,321	11,326	41,740	152,986
LS-бінарний алгоритм без множення	3,205	15,100	70,187	329,7268	1675,817	9603,931	56143,180
Модифікація Ласло Харса *LS*-бінарного алгоритму без множення	3,841	15,082	70,067	328,734	1668,572	9555,875	55887,460
SE-алгоритм без множення	0,120	0,314	0,891	2,810	11,574	42,074	147,402
Модифікація Ласло Харса *SE*-алгоритму без множення	0,283	0,675	1,708	4,736	16,887	58,072	195,201

Запропоновано три модифікації алгоритмів обчислення мультиплікативно оберненого елемента: модифікований розширений *RS*-бінарний алгоритм Евкліда для непарних модулів, удосконалена модифікація Бредлі розширеного алгоритму Евкліда, модифікований розширений плюс-мінус алгоритм, модифікований розширений алгоритм Нортона. Кожна із запропонованих модифікацій дає приріст швидкодії порівняно зі своїм базовим алгоритмом.

Серед існуючих алгоритмів обчислення мультиплікативно оберненого елемента найкращі часові показники має модифікація Бредлі розширеного алгоритму Евкліда. Порівнюючи запропоновані модифікації існуючих алгоритмів бачимо, що найкращі часові показники дає удосконалена модифікація Бредлі розширеного алгоритму Евкліда та, залежно від довжини операндів, забезпечує приріст швидкості виконання обчислень на 3-5% порівняно з існуючими методами.

5.3. Дослідження ефективності запропонованого методу виконання операцій над елементами поля $GF(2^m)$

Оцінимо кількість ітерацій при перетворенні елементів поля зі степеневого подання у числове і навпаки.

Перетворення зі степеневого подання у числове виконується щонайбільше за $\left\lfloor \frac{k}{2} \right\rfloor$ ітерацій (зсув на один біт та додавання незвідного многочлена).

При перетворенні з числового подання у степеневе найбільша кількість ітерацій (зсув на один біт та додавання незвідного многочлена) буде дорівнювати максимальному значенню попарних сусідніх різниць показників степеня.

Наприклад, якщо $k = 3$, то залишаться такі показники степеня в розрідженій таблиці (рис. 3.9): 4; 5; 14; 6; 12. Впорядкуємо за неспаданням попередньо додавши нуль: 0; 4; 5; 6; 12; 14. Обчислимо попарні різниці: 4; 1; 1; 6; 2. Найбільше значення з попарних різниць дорівнює 6. Отже, в найгіршому випадку для виконання перетворення з числового у степеневе подання знадобиться 6 ітерацій.

У розглянутих прикладах використовувалось значення k, що дорівнює 3, але для апаратної реалізації необхідно обирати значення ступеня розрідженості, що є степенем 2. При такому підході кожне наступне значення ступеня розрідженості дозволяє скоротити розрядність адреси на один розряд, а також значно спростити операцію цілочисельного ділення на k в алгоритмах перетворення з числового у степеневе подання і навпаки, оскільки в такому випадку операція цілочисельного ділення вироджується в операцію побітового зсуву.

Для оцінювання ефективності використання розрідженої таблиці, порівняємо необхідну кількість елементарних операцій, що необхідна для виконання операцій над елементами поля $GF(2^m)$, при використанні розрідженої таблиці та поліноміального базису.

Операції обчислення мультиплікативно оберненого елемента та піднесення до степеня вимагають один операнд (унарні операції), що є елементом поля $GF(2^m)$, а операції множення та ділення – два (бінарні операції). Тому при використанні розрідженої таблиці для унарних операцій необхідно виконувати перетворення з многочленного у степеневе подання лише одного операнда, а для бінарних – двох. Після виконання кожної з перелічених операцій, отриманий результат необхідно перетворити зі степеневого подання у многочленне.

Над степеневим поданням виконання операції обчислення мультиплікативно оберненого елемента вимагає дві m-бітові операції (інвертування та редукція за модулем $2^m - 1$), множення теж дві m-бітові операції (додавання та редукція за модулем $2^m - 1$), ділення – три m-

бітові операції (інвертування, додавання та редукція за модулем $2^m - 1$), піднесення до степеня – $2.5 \cdot t$ операцій над m-бітовим операндом, де t – розрядність показника степеня.

Для визначення середньої кількості операцій додавань/зсувів при використанні розрідженої таблиці розроблено програмне забезпечення. Експериментальні дослідження проводились на випадкових наборах вхідних даних, що містять 1500 операндів та є елементами заданого поля (табл. 5.10 та 5.11).

Таблиця 5.10

Експериментально отримана середня оцінка кількості операцій додавань/зсувів при використанні таблиці зі ступенем розрідження 8

Операції над елементами скінченного поля	m									
	4	5	10	13	15	16	17	18	19	20
Обчислення мультиплікативно оберненого елемента	15.00	14.88	20.55	21.80	23.12	22.00	20.12	20.79	20.66	20.77
Піднесення до степеня	13.00 + 2.5*t	12.88 + 2.5*t	18.55 + 2.5*t	19.80 + 2.5*t	21.12 + 2.5*t	20.00 + 2.5*t	18.12 + 2.5*t	18.79 + 2.5*t	18.66 + 2.5*t	18.77 + 2.5*t
Множення	27.95	27.91	39.00	41.62	44.24	42.02	38.14	39.64	40.95	39.42
Ділення	28.95	28.91	40.00	42.62	45.24	43.02	39.14	40.64	41.95	40.42

Знайдемо середню кількість операцій при застосуванні виключно многочленного подання для виконання операцій над елементами поля $GF(2^m)$. Операція множення елементів поля (множення многочленів та приведення за модулем незвідного) вимагає в середньому m зсувів та m додавань над m-бітовими операндами; операція обчислення мультиплікативно оберненого елемента (з використанням розширеного алгоритму Евкліда) – $2m$ зсувів та $2m$ додавань; ділення (пошук мультиплікативно оберненого елемента до другого операнда та домноження отриманого результату на перший операнд) – $3m$ зсувів та $3m$ додавань;

піднесення до степеня (бінарний алгоритм множення) – $1.5 \cdot m \cdot t$ зсувів та $1.5 \cdot m \cdot t$ додавань, де t – розрядність показника степеня. Середня оцінка кількості бітових операцій при застосуванні многочленного подання для виконання операцій над елементами поля $GF(2^m)$ наведена у табл. 5.12.

Таблиця 5.11

Експериментально отримана середня оцінка кількості операцій додавань/зсувів при використанні таблиці зі ступенем розрідження 16

Операції над елементами скінченного поля	m								
	5	10	13	15	16	17	18	19	20
Обчислення мультиплікативно оберненого елемента	29.48	39.54	43.02	56.67	45.90	41.30	42.15	46.25	42.81
Піднесення до степеня	27.48 + 2.5*t	37.54 + 2.5*t	41.02 + 2.5*t	54.67 + 2.5*t	43.90 + 2.5*t	39.30 + 2.5*t	40.15 + 2.5*t	44.25 + 2.5*t	40.81 + 2.5*t
Множення	57.32	77.19	84.06	111.12	87.76	80.60	82.58	90.51	83.54
Ділення	58.32	78.19	85.06	112.12	90.76	81.60	83.58	91.51	84.54

Таблиця 5.12

Середня оцінка кількості операцій додавань/зсувів при застосуванні многочленного подання

Операції над елементами скінченного поля	m									
	4	5	10	13	15	16	17	18	19	20
Обчислення мультиплікативно оберненого елемента	16	20	40	52	60	64	68	72	76	80
Піднесення до степеня	12*t	15*t	30*t	39*t	45*t	48*t	51*t	54*t	57*t	60*t
Множення	8	10	20	26	30	32	34	36	38	40
Ділення	24	30	60	78	90	96	102	108	114	120

На рис. 5.1 – 5.3 наведено діаграми, що дають можливість порівняти обчислювальну складність методів виконання операцій над елементами поля $GF(2^m)$ при апаратній реалізації за кількістю елементарних операцій Q (операцій зсуву та порозрядного додавання). Для запропонованого методу обрано ступінь розрідження таблиці, який дорівнює 8.

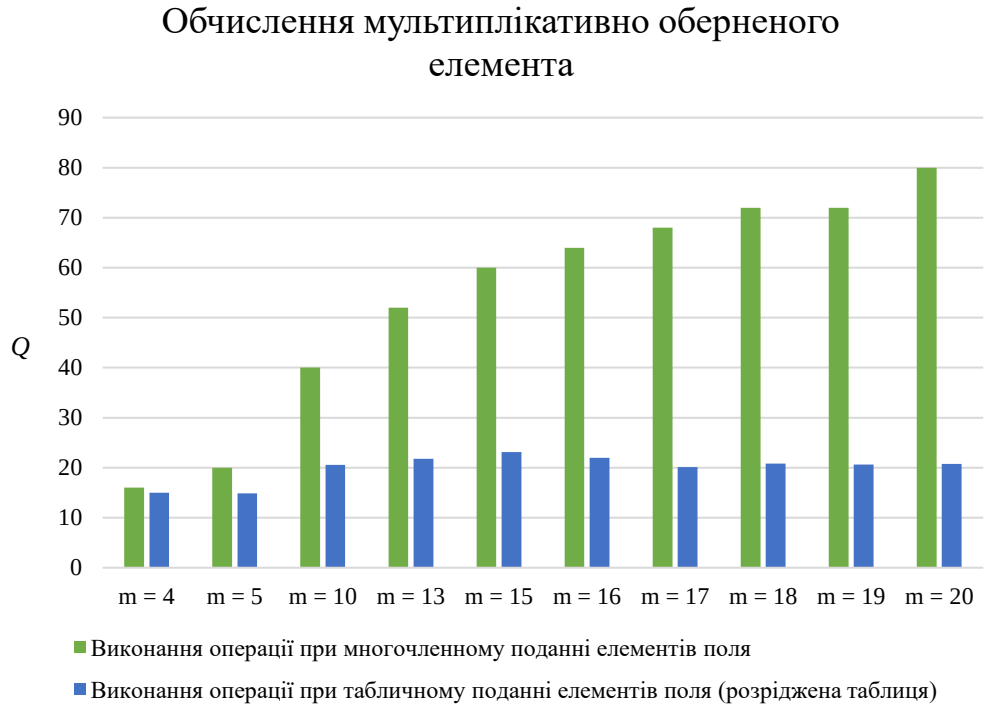


Рис. 5.1. Кількість елементарних операцій при обчисленні мультиплікативно оберненого елемента поля $GF(2^m)$

З наведених діаграм можна зробити висновок, що запропонований метод характеризується слабкою залежністю кількості операцій зсуву і порозрядного додавання для виконання операції обчислення мультиплікативно оберненого елемента та ділення від параметра m поля. Для операції множення запропонований метод дає зменшення кількості операцій лише починаючи зі значення $m = 20$. Окрім цього, звертає на себе увагу той факт, що зі збільшенням значенням m при фіксованому ступені розрідження перевага запропонованого методу стає більш значною.

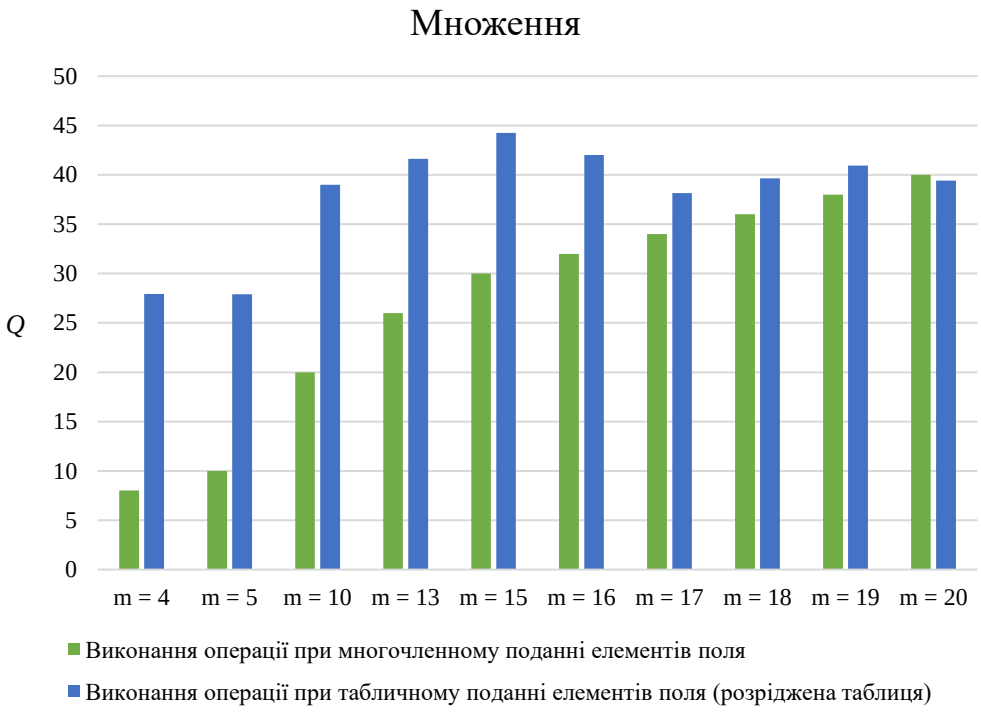


Рис. 5.2. Кількість елементарних операцій
при множенні елементів поля $GF(2^m)$

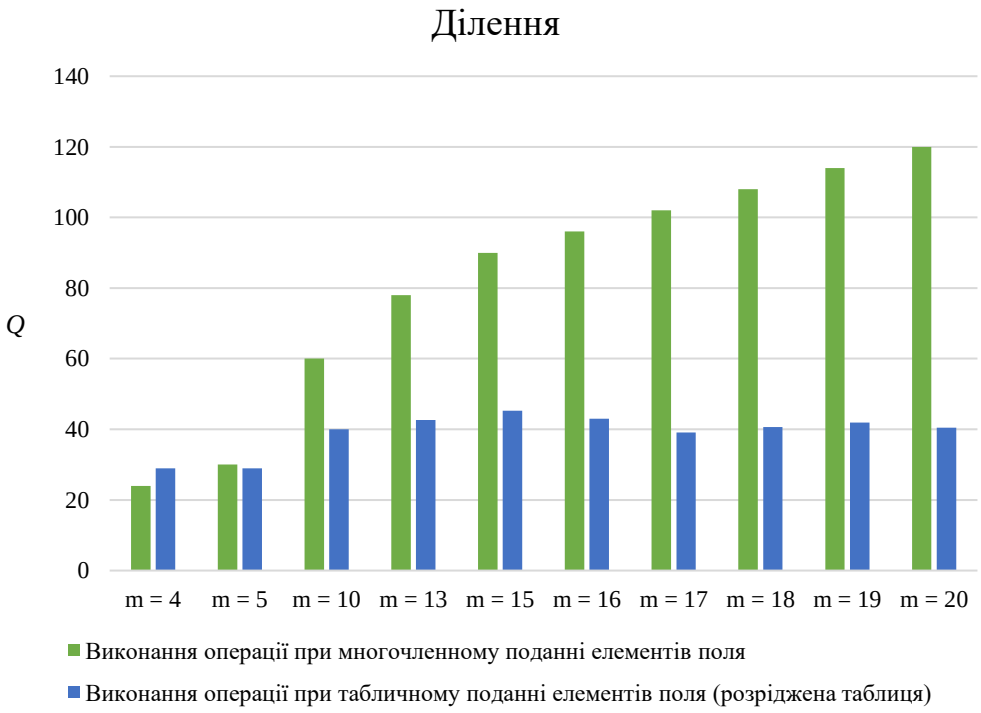


Рис. 5.3. Кількість елементарних операцій
при діленні елементів поля $GF(2^m)$

На рис. 5.4 наведено графік приросту швидкодії (відношення кількості бітових операцій) для операції піднесення до степеня залежно від обраного поля (для розробленого методу обрано ступінь розрідженості 16).

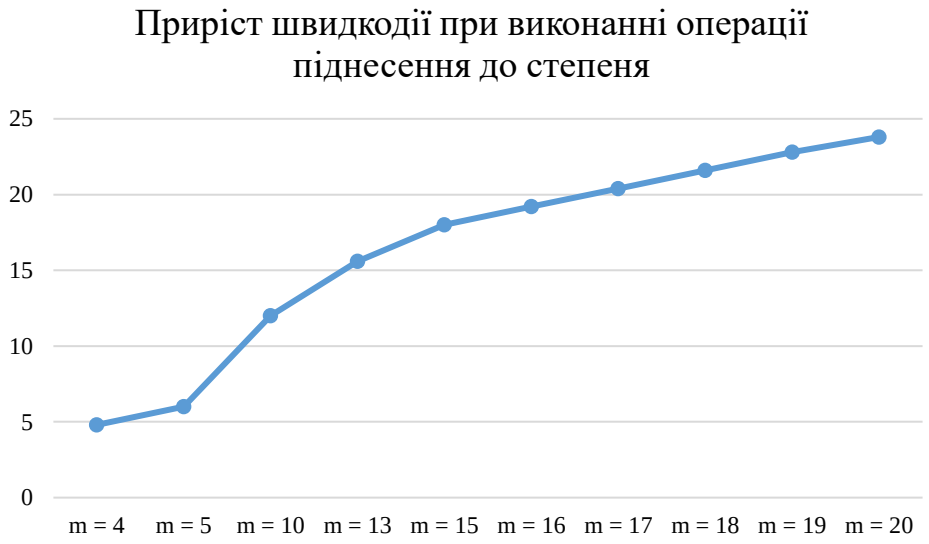


Рис. 5.4. Коефіцієнт приросту швидкодії виконання операції піднесення до степеня елементів поля $GF(2^m)$

Таким чином, оптимальним ступенем розрідження таблиці є значення 8, що забезпечує, в середньому приріст швидкодії на 15% порівняно із використанням поліноміального базису.

5.4. Аналіз апаратних ресурсів, необхідних для реалізації запропонованих методів на ПЛІС

ПЛІС мають обмежені ресурси, кожна інтегральна схема має певну кількість портів для вводу-виводу, тригерів, CLB, таблиць істинності (LUT). Необхідно проектувати такі апаратні схеми, які можна буде розмістити на цільовій ПЛІС, такі схеми мають використовувати не більше ресурсів ніж може надати цільова ПЛІС.

Ця проблема є особливо актуальною коли цільовою є ПЛІС малого, або середнього розміру і/або щільності, саме такою є Spartan-3AN, на яку орієнтована розробка апаратних схем у цій роботі.

Операція обчислення мультиплікативно оберненого елемента у скінченному полі є актуальною як для чисел малої та середньої розрядності, наприклад, при кодуванні стійкими до помилок кодами Ріда-Соломона, так і для чисел великої розрядності, наприклад, при застосуванні криптографічних алгоритмів. Виходячи з цього, було вирішено провести дослідження 16-, 32- та 64-розрядних версій апаратних реалізацій алгоритмів обчислення мультиплікативно-оберненого елемента.

Схеми, що реалізують розширений *RS*-бінарний алгоритм Евкліда, його модифікацію, та розширений плюс-мінус алгоритм Евкліда, є параметризованими, і можуть бути синтезовані для роботи з операндами будь-якої розрядності. Схема, що реалізує алгоритм Бредлі, через використання елемента для виконання ділення, фіксована для роботи з операндами довжини 16 розрядів, і не може бути масштабована.

На рис. 5.5 зображено діаграми мінімальної кількості кожного типу ресурсів ПЛІС, необхідної для успішного розміщення апаратної схеми. Різними кольорами позначено версії схем для операндів різної розрядності. Групи стовпців, позначені цифрами 1, 2, 3 і 4, відповідають алгоритму Бредлі, розширеному *RS*-бінарному алгоритму Евкліда, удосконаленому алгоритму Бредлі та модифікованому плюс-мінус алгоритму відповідно. Горизонтальна лінія позначає наявну кількість ресурсу кожного типу на цільовій ПЛІС.

Як видно з наведених діаграм, 16-розрядні версії усіх схем можна розмістити на Spartan-3AN. З усіх спроектованих апаратних схем лише апаратна реалізація запропонованої модифікації розширеного *RS*-бінарного алгоритму Евкліда зможе розміститися на цільовій ПЛІС у версії для операндів довжиною 32 розряди. Жодна схема у 64-розрядній версії не може бути розміщена на Spartan-3AN.

Слід зауважити що ПЛІС Virtex-7 має достатньо ресурсів усіх типів для розміщення 64-розрядних версій усіх спроектованих схем.

Важливим для дослідження апаратних реалізацій алгоритмів обчислення мультиплікативно-оберненого елемента у скінченному полі є вимірювання їх швидкодії.

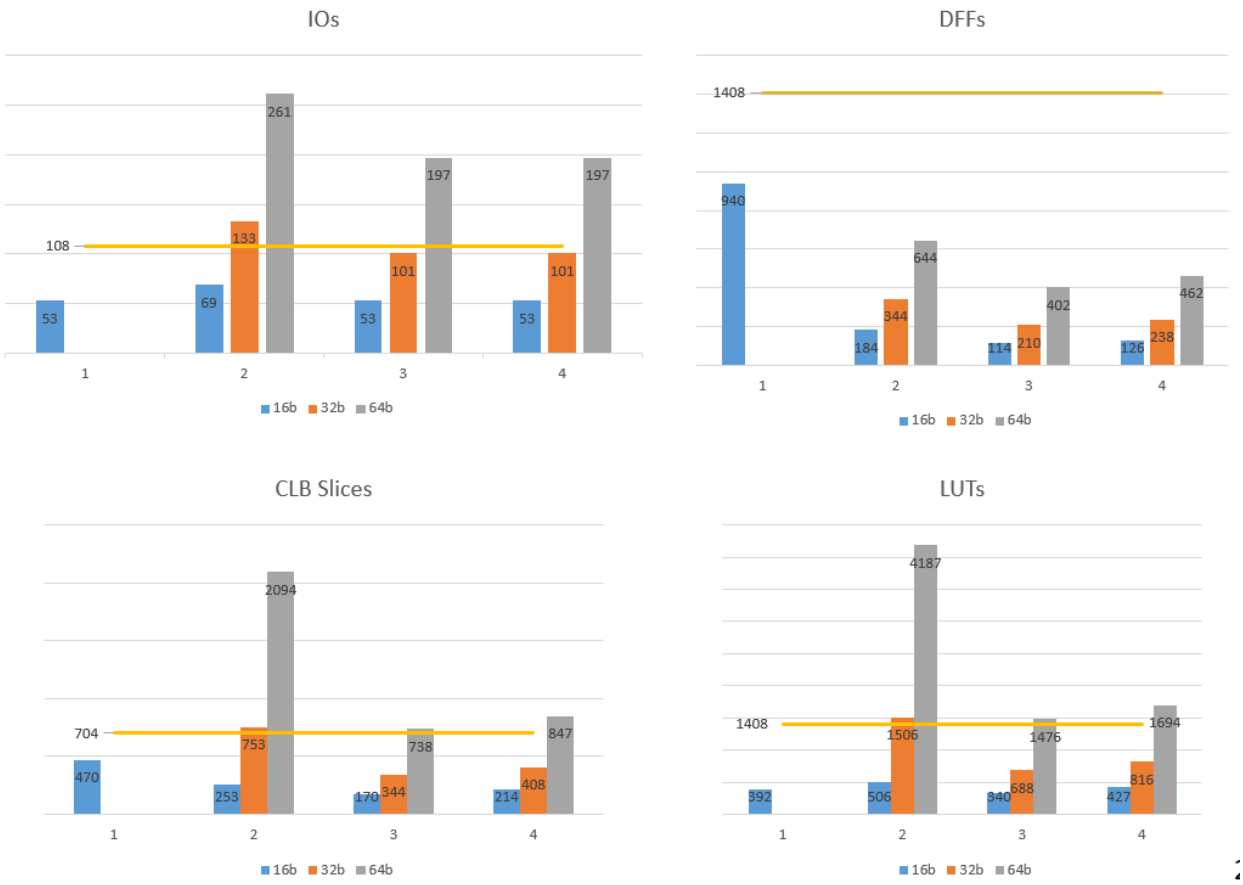


Рис. 5.5. Використання схемами ресурсів ПЛІС
залежно від розрядності операндів

Використовуючи симулятор функціонування апаратних схем, такий як ModelSim, можна знайти кількість тактів, необхідних при апаратній реалізації алгоритму для виконання операції обчислення мультиплікативно-оберненого елемента у скінченному полі.

Використовуючи програмний комплекс для тестування алгоритмів, було згенеровано два набори випадкових модулів і елементів поля:

перший – з операндами довжиною 16 двійкових розрядів, другий – 32. На кожній схемі у версії для 16- і 32-розрядних операндів було виконано пошук мультиплікативно оберненого елемента за модулем для кожної пари елемент-модуль зі згенерованого набору операндів відповідної довжини. На рис. 5.6 представлено діаграму середнього часу роботи схем у тактах. Різними кольорами позначено версії схем для операндів різної розрядності. Групи стовпців, позначені цифрами 1, 2, 3 і 4, відповідають алгоритму Бредлі, розширеному *RS*-бінарному алгоритму Евкліда, удосконаленому алгоритму Бредлі та модифікованому плюс-мінус алгоритму відповідно.

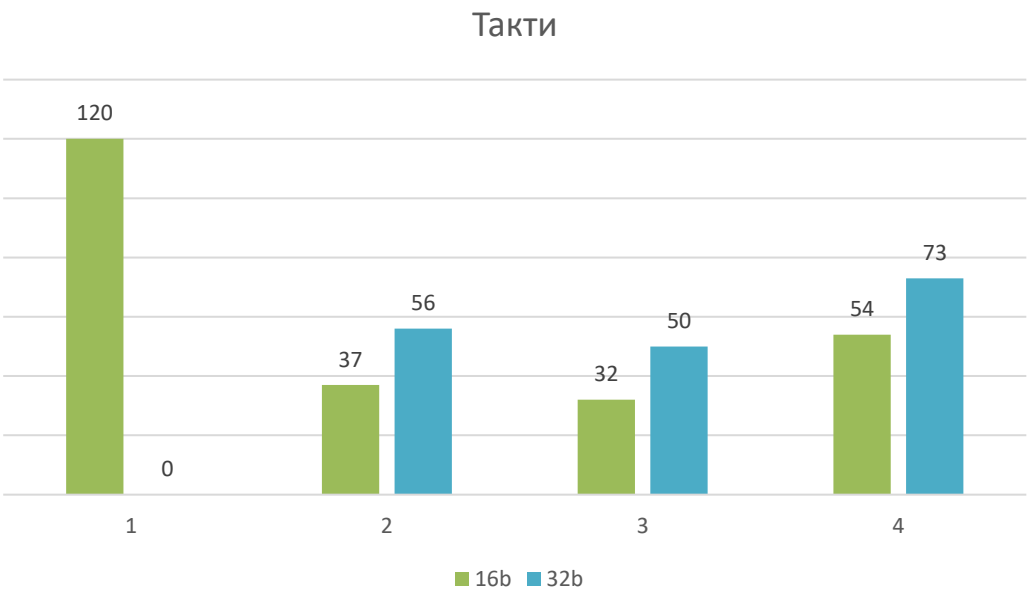


Рис. 5.6. Середній час роботи схем у тактах

З метою коректного порівняння швидкодії програмних і апаратних реалізацій алгоритмів, необхідно виконати перехід від вимірювання середнього часу виконання схемами операції обчислення мультиплікативно оберненого елемента у тактах, до вимірювання у фізичних одиницях часу.

Такий перехід дозволяють зробити використані інструменти синтезу апаратних схем, а саме *Precision RTL* від *Mentor Graphics*. Цей програмний пакет виконує статичний аналіз спроектованої апаратної схеми, і беручи до

уваги її складність, розмір і критичний шлях, визначає максимальну тактову частоту та мінімальну довжину такту у наносекундах, з якою дана схема може коректно працювати на конкретній цільовій ПЛІС.

Інформація, надана інструментами синтезу про мінімальну довжину такту у нс, для розроблених апаратних схем представлена у табл. 5.13.

Таблиця 5.13

Мінімальна довжина такту апаратних схем у нс

№	Алгоритм	Довжина такту, нс (16 біт)	Довжина такту, нс (32 біт)
1	Алгоритм Бредлі	16	—
2	Розширений *RS*-бінарний алгоритм Евкліда	10	14
3	Удосконалений алгоритм Бредлі	8	10
4	Розширений плюс-мінус алгоритм (наша модифікація)	10	9

Маючи дані про середню кількість тактів, необхідну схемам на виконання операції обчислення мультиплікативно-оберненого елемента, та дані про мінімальну довжину такту кожної схеми у наносекундах, можна зробити висновки про середній час виконання операції обчислення мультиплікативно-оберненого елемента кожною з розроблених апаратних схем у 16- та 32-розрядних варіантах. Ці дані у вигляді діаграм зображено на рис. 5.7. Різними кольорами позначено версії схем для операндів різної розрядності. Групи стовбців, позначені цифрами 1, 2, 3 і 4, відповідають алгоритму Бредлі, розширеному *RS*-бінарному алгоритму Евкліда, його модифікації та модифікованому плюс-мінус алгоритму відповідно.

Спостерігається прискорення апаратної реалізації запропонованої модифікації розширеного *RS*-бінарного алгоритму Евкліда в середньому

на 20 – 30% порівняно з апаратною реалізацією базового *RS*-бінарного алгоритму Евкліда.

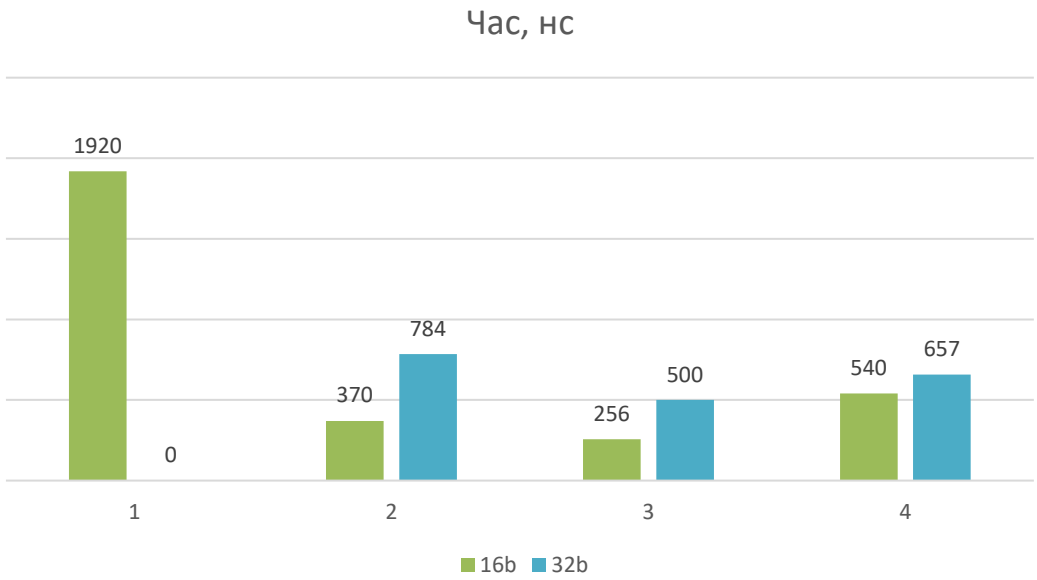


Рис. 5.7. Середній час роботи схем у нс

Оцінимо кількість апаратних ресурсів, необхідних для реалізації на ПЛІС розробленого процесора Галуа та складових частин даного процесора, в залежності від значення *m* (табл. 5.14-5.16).

Таблиця 5.14

Використання ресурсів ПЛІС для значення $m = 4$

	Процесор Галуа	Блок виконання операцій над елементами поля $GF(2^m)$	*ROM*	Блок виконання операцій за модулем $2^m - 1$
IOs	45	33	12	43
Global Buffers	1	3	0	1
LUTs	5468	359	8	101
CLBs	3317	180	4	51
Dffs	6634	156	0	27

Таблиця 5.15

Використання ресурсів ПЛІС для значення $m = 8$

	Процесор Галуа	Блок виконання операцій над елементами поля $GF(2^m)$	*ROM*	Блок виконання операцій за модулем $2^m - 1$
IOs	53	45	24	56
Global Buffers	1	3	0	1
LUTs	7258	851	336	178
CLBs	4378	426	168	89
Dffs	8755	197	0	52

Таблиця 5.16

Використання ресурсів ПЛІС для значення $m = 13$

	Процесор Галуа	Блок виконання операцій над елементами поля $GF(2^m)$	*ROM*	Блок виконання операцій за модулем $2^m - 1$
IOs	63	60	39	72
Global Buffers	1	3	0	1
LUTs	27551	19761	17718	284
CLBs	13776	9881	8859	142
Dffs	11406	248	0	83

5.5. Висновки до розділу 5

1. Досліджено запропоновану модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном за двома

показниками: швидкістю роботи та необхідною кількістю операцій множення. Зменшення кількості операцій множення дозволяє забезпечити приріст швидкості на 7-9%. Зі збільшенням розрядності операндів збільшується рекомендована кількість елементів таблиці передобчислень, а саме: для довжини показника степеня до 128 біт оптимальною кількістю елементів таблиці передобчислень запропонованого методу є 7, для довжини показника степеня від 256 біт до 1024 біт оптимальним значенням є 31 елемент таблиці передобчислень, для довжини показника степеня від 2048 біт до 8192 біт оптимальним значенням є 63 елементи таблиці передобчислень.

2. Проведено дослідження двох груп методів обчислення мультиплікативно оберненого елемента поля $GF(p)$: методів, що засновані на модулярному піднесенні до степеня, та методів, що ґрунтуються на алгоритмі Евкліда знаходження НСД. Результати дослідження показують, що менш обчислювально витратними є методи, що ґрунтуються на алгоритмі Евкліда знаходження НСД. Запропонована удосконалена модифікація алгоритму Бредлі показує приріст швидкодії 3-5% порівняно з існуючими методами обчислення мультиплікативно оберненого елемента.
3. Запропоновано спосіб отримання верхньої оцінки кількості елементарних однотактних операцій для методів обчислення мультиплікативно оберненого елемента та операції піднесення до степеня в полі $GF(2^m)$. Отримано оптимальні значення ступеня розрідженості таблиці елементів поля $GF(2^m)$. Для операції обчислення мультиплікативно оберненого елемента для значення $m = 20$ запропонований метод дає приріст швидкодії приблизно в 4 рази, а для операції піднесення до степеня – майже у 25 разів.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальну науково-прикладну задачу – підвищення продуктивності систем цифрової обробки даних, забезпечення завадостійкості зберігання і передачі даних та криптографічних перетворень за рахунок створення ефективних технічних засобів для виконання обчислень у скінченних полях шляхом структурно-логічної оптимізації архітектур апаратних засобів, що реалізують процеси виконання операцій у полях Галуа. При цьому отримано такі теоретичні й практичні результати.

1. На основі проведеного аналізу, з урахуванням виділених пріоритетних ознак, виконано класифікацію методів виконання найбільш обчислювально витратних операцій у скінченних полях, а саме: обчислення мультиплікативно оберненого елемента та піднесення до степеня, що дало можливість провести ґрунтовне дослідження та сформувати напрямки розвитку зазначених методів.
2. Запропоновано метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та відповідні структури апаратних засобів для його реалізації, що характеризуються універсальністю. Дослідження показали, що за рахунок табличного зберігання елементів поля у многочленному та степеневому їх поданні забезпечується максимальна швидкодія та універсальність арифметико-логічного пристрою. При використанні операндів великої розрядності запропоновано розріджене формування таблиці елементів поля, що дозволяє у кілька разів скоротити витрати пам’яті для її зберігання. Розроблений метод забезпечує зростання швидкодії в середньому на 15% порівняно з існуючим методом.
3. Розроблено модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном. Відмінність від існуючих методів

полягає в тому, що при формуванні таблиці передобчислень використовуються показники степеня, що є простими числами. Для досягнення високої швидкодії при побудові таблиці передобчислень рекомендовано використовувати заздалегідь обчислені адитивні ланцюжки з метою мінімізації кількості операцій множення, що дозволяє отримувати кожен наступний елемент таблиці за одну-дві операції модулярного множення. За допомогою розробленої моделі обчислювального процесу встановлено, що запропонована модифікація методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном забезпечує приріст швидкодії на 7-9%. Модифікацію методу піднесення до степеня елементів поля $GF(p)$ з ковзним вікном можна застосовувати в еліптичній криптографії для поліпшення часових характеристик операції скалярного множення точки еліптичної кривої на число.

4. Розроблено модель обчислювального процесу при виконанні операцій у скінченних полях, яка дозволяє на основі заданих наборів вхідних даних виконувати порівняння методів та здійснювати оптимальний вибір параметрів і форм подання операндів, що забезпечує зростання швидкодії при реалізації обчислювальних операцій на ПЛІС. На основі запропонованої моделі розроблено методики дослідження нових способів апаратної реалізації обчислень у полях Галуа. Моделювання у середовищі розробки *Xilinx ISE* та за допомогою програми *Mentor Graphics Precision* показало, що розроблені структури апаратних засобів характеризуються мінімальною апаратною складністю та високою швидкодією.
5. Дістала подальший розвиток теорія обчислень у скінченних полях, яка характеризується спрямованістю на апаратну реалізацію операцій, високою функціональністю архітектурних рішень на основі ПЛІС та дозволяє сформувати інструментально забезпечене

обчислювальне середовище, пристосоване до організації високоефективних обчислень у скінченних полях, зокрема з використанням ПЛІС фірми Xilinx.

6. Розроблено архітектуру та систему команд спеціалізованого процесора Галуа, орієнтованого на виконання операцій у скінченних полях, який забезпечує зростання продуктивності обчислень на 27% порівняно з універсальними обчислювальними засобами. Процесор Галуа можна використовувати в універсальній обчислювальній системі як співпроцесор, доповнюючи систему команд центрального процесора, або як спецобчислювач на основі ПЛІС, який порівняно з універсальними обчислювальними засобами підвищує продуктивність обробки інформації в реальному часі. Особливістю архітектури розробленого процесора є те, що не змінюючи інтерфейсу процесора, шляхом зміни арифметико-логічного пристрою можна здійснити перехід до поля Галуа $GF(p)$ або до $GF(2^m)$.
7. Отримані в роботі результати дозволяють на практиці підвищити продуктивність систем захисту інформації, систем цифрової обробки сигналів та завадостійкого кодування даних.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Айерленд, К. Классическое введение в современную теорию чисел [Текст] / К. Айерленд, М. Роузен // Пер. с англ. – М. : Мир, 1987. – 416 с.
2. Алферов, А. П. Основы криптографии [Текст] / А. П. Алферов, А. Ю. Зубов, А. С. Кузьмин, А. В. Черемушкин. – М. : Гелиос АРВ. – 2002. – 480 с.
3. Амосов, В.В. Схемотехника и средства проектирования цифровых устройств [Текст] / В.В. Амосов. – СПб. : БХБ-Петербург, 2007. – 560 с. : ил.
4. Андерсон, Дж. Дискретная математика и комбинаторика [Текст] / Дж. Андерсон ; пер. с англ. – М. : Издательский дом “Вильямс”, 2004. – 960 с. : ил. – Парал. тит. англ.
5. Ахо, А.В. Структуры данных и алгоритмы [Текст] / А.В. Ахо, Дж. Хопкрофт, Дж.Д. Ульман // пер. с англ. уч. пос. – М. : Издательский дом “Вильямс”, 2007. – 400 с. : ил. – Парал. тит. англ.
6. Баричев, С.Г. Основы современной криптографии [Текст] / С.Г. Баричев, В.В. Гончаров, Р.Е. Серов. – М. : Горячая линия-Телеком, 2001. – 120 с. : ил.
7. Берлекэмп, Э. Алгебраическая теория кодирования [Текст] / Э. Берлекэмп. – М. : Мир, 1971. – 479 с.
8. Бернет, С. Криптография. Официальное руководство RSA Security [Текст] / С. Бернет. – Изд. 2-е, стереотипное. – М. : ООО “Бином-Пресс”, 2007. – 384 с. : ил.
9. Болотов, А.А. Алгоритмические основы эллиптической криптографии [Текст] / А.А. Болотов. – М. : Изд-во, 2004. – 499 с.
10. Болотов, А.А. Элементарное введение в эллиптическую криптографию: Алгебраические и алгоритмические основы [Текст] /

А.А. Болотов, С.Б. Гашков, А.Б. Фролов, А.А. Часовских. – М. : КомКнига, 2006. – 328 с.

11. Боюн, В.П. Динамическая теория информации. Основы и приложения [Текст] / В.П. Боюн. – К. : Ин-т кибернетики им. В. М. Глушкова НАН Украины, 2001. – 326 с.
12. Брассар, Ж. Современная криптология [Текст] / Ж. Брассар; пер. с англ. – М. , Издательско-полиграфическая фирма ПОЛИМЕД, 1999. – 176 с., илл.
13. Василенко, О.Н. Теоретико-числовые алгоритмы в криптографии [Текст] / О.Н. Василенко. – 2-е изд., доп. – М. : МЦНМО, 2006. – 336 с.
14. Вейль, Г. Алгебраическая теория чисел [Текст] / Г. Вейль; пер. с англ. – Изд. 5-е. – М. : Едиториал УРСС, 2011. – 224 с.
15. Виноградов, И.М. Основы теории чисел [Текст] / И. М. Виноградов. – Издание шестое, исправленное – Москва-Ленинград : Государственное издательство технико-теоретической литературы, 1952. – 181 с.
16. Грэхем, Р. Конкретная математика. Основание информатики [Текст] / Р. Грэхем, Д. Кнут, О. Паташник; пер. с англ. – М. : Мир; БИНОМ. Лаборатория знаний, 2009. – 703 с. : ил.
17. Державний патент України №111351, МПК G06F 7/00, G06F 7/50. Схема для пошуку мультиплікативно оберненого елемента за довільним модулем [Текст] / Дичка І.А., Онай М.В., Приходько Е.В. ; заявник та патентовласник Дичка І.А., Онай М.В., Приходько Е.В. – № u201604179; дата подання заявки 15.04.2016; дата публ. 10.11.2016, бюл. №21, 2016 р. – 7 с.
18. Державний патент України №40145, МПК G06F 7/00. Пристрій для ділення елементів скінченних полів $GF(2^n)$ [Текст] / Жуков І.А., Кубицкий В.І., Синельніков О.О. ; заявник та патентовласник Національний авіаційний університет. – № u200812736; дата подання заявки 30.10.2008; дата публ. 25.03.2009, бюл. №6, 2009 р. – 8 с.

19. Державний патент України №54637, МПК G06F 7/50. Суматор за модулем простого числа [Текст] / Дичка І.А., Онай М.В. ; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201001074; дата подання заявки 02.02.2010; дата публ. 25.11.2010, бюл. №22, 2010 р. – 14 с.
20. Державний патент України №57281, МПК G06F 7/48. Суматор елементів поля $GF(p^m)$ [Текст] / Дичка І.А., Онай М.В. ; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201004903; дата подання заявки 23.04.2010; дата публ. 25.02.2011, бюл. №4, 2011 р. – 17 с.
21. Державний патент України №73309, МПК G06F 7/50. Пристрій для виконання обчислень в полі $GF(2^n)$ [Текст] / Дичка І.А., Онай М.В. ; заявник та патентовласник Національний технічний університет України “Київський політехнічний інститут”. – № u201115679 ; дата подання заявки 30.12.2011; дата публ. 25.09.2012, бюл. №18, 2012 р. – 10 с.
22. Дичка, І.А. Апаратна реалізація обчислень у скінченних полях характеристики два / І.А. Дичка, М.В. Онай, Ю.В. Бухтіяров // Наукові вісті НТУУ “КПІ”. – 2013. – №6. – С. 20-27.
23. Дичка, І.А. Апаратна реалізація операторів та функцій в полях Галуа [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Хмельницького національного університету. – 2012. – Вип. 5. – С. 234-240.
24. Дичка, І.А. Апаратна реалізація процедур множення і ділення многочленів у скінченних полях [Текст] / І.А. Дичка, В.І. Голуб, М. В. Онай // Наукові вісті НТУУ “КПІ”. – 2012. – №5. – С. 61-66.
25. Дичка, І.А. Архітектура проблемно-орієнтованого процесора для реалізації арифметики скінченних полів [Текст] / І.А. Дичка, М.В. Онай, О.В. Ващілін // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №12 (183), ч.2. – С. 99-106.

26. Дичка, І.А. Застосування *k*-арного методу Евкліда для пошуку мультплікативно оберненого елемента у кільці лишків за модулем *m* [Текст] / І.А. Дичка, М.В. Онай, А.Ю. Бартков'як // Матеріали статей П'ятої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Івано-Франківськ : п. Голіней О.М., 2015. – С. 151-153.
27. Дичка, І.А. Модифікований віконний метод однократного множення точки еліптичної кривої на скаляр у полі $GF(p)$ / І.А. Дичка, М.В. Онай, Т.П. Дрозда // Науковий журнал "Радіоелектроніка, інформатика, управління". – Запоріжжя. – 2016. – №2. – С. 95-102.
28. Дичка, І.А. Організація проблемно-орієнтованого процесора для реалізації операцій в полях Галуа виду $GF(2^m)$ / І.А. Дичка, М.В. Онай // Тези доповідей Четвертої Міжнародної науково-практичної конференції "Методи та засоби кодування, захисту й ущільнення інформації" м. Вінниця, 23-25 квітня 2013 року. – Вінниця : ПП "ТД "Едельвейс і К", 2013. – С. 270-273
29. Дичка, І.А. Організація системи команд співпроцесора Галуа / І.А. Дичка, М.В. Онай // Міжнародна науково-технічна конференція "Радіотехнічні поля, сигнали, апарати та системи". Київ, 11-15 березня 2013 р. : матеріали конференції. – Київ: 2013. – С. 212-213.
30. Дичка, І.А. Організація спеціалізованих комп'ютерних систем для реалізації обчислень у скінченних полях / І.А. Дичка, В.І. Голуб, М.В. Онай // Вісник Східноукраїнського національного університету ім. В. Даля. – 2012. – №6 (177). – С. 268-278.
31. Дичка, І.А. Особливості апаратної реалізації операції віднімання в кільці лишків за модулем $2^m - 1$ / І.А. Дичка, М.В. Онай // Всеукраїнська науково-практична конференція "Проблеми інформатики та комп'ютерної техніки". Тези доповідей. – Чернівці : Видавничий дім "Родовід", 2013. – С. 111-113.

32. Дичка, І.А. Особливості апаратної реалізації операції додавання в кільці лишків за модулем $2^m - 1$ / І.А. Дичка, М.В. Онай // Інформатика та системні науки (ІСН-2013): матеріали IV Всеукр. наук.-практ. конф., (м. Полтава, 21-23 берез. 2013 р.) / за ред. Ємця О.О. – Полтава : ПУЕТ, 2013. – С. 78-81
33. Дичка, І.А. Подання інформації у графічно-кодованому вигляді: технологія кодування та декодування / І.А. Дичка, М.В. Онай, М. В. Новосад // Реєстрація, зберігання і обробка даних (Data Recording, Storage & Processing). – 2010. – Т. 12, №2. – С. 69-80.
34. Дичка, І.А. Спосіб зберігання в пам'яті ЕОМ різних форм подання елементів скінченного поля характеристики 2 / І.А. Дичка, М.В. Онай // Комп'ютерні інтелектуальні системи та мережі. Матеріали VI Всеукраїнської WEB-конференції аспірантів, студентів та молодих вчених (19-21 березня 2013 р.). – Кривий Ріг : Криворізький національний університет, 2013. – С. 56-59.
35. Дичка, І.А. Способи знаходження мультиплікативного оберненого елемента в скінченних полях [Текст] / І.А. Дичка, М.В. Онай // Друга наукова конференція магістрантів та аспірантів присвячена 20-річчю факультету прикладної математики «Прикладна математика та комп'ютинг ПМК-2010» : Київ, 14-16 квіт. 2010 р. : зб. тез доп. / ред кол. : Дичка І.А. [та ін.] – К. : Просвіта, 2010. – С. 313-317.
36. ДСТУ 4145-2002. Державний стандарт України. Інформаційні технології. Криптографічний захист інформації. Цифровий підпис, що ґрунтується на еліптичних кривих. Формування та перевірка [Текст]. – Введ. 28–02–2002. – Київ : Держстандарт України, 2003. – 44 с.
37. Зензин, О.С. Стандарт криптографической защиты – AES. Конечные поля [Текст] / О.С. Зензин, М.А. Иванов; под ред. М.А. Иванова – М. : КУДИЦ-ОБРАЗ, 2002. – 176 с.
38. Коблиц, Н. Курс теории чисел и криптографии [Текст] / Н. Коблиц. – М. : Научное изд-во ТВП, 2001. – 254 с.

39. Кормен, Т.Х. Алгоритмы: построение и анализ [Текст] / Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К. Штайн. – 2-е издание, пер. с англ. – М. : Издательский дом “Вильямс”, 2009. – 1296 с. : ил. – Парал. тит. англ.
40. Корнійчук, В.І. Основи комп’ютерної арифметики [Текст] / В.І. Корнійчук, В.П. Тарасенко, О.В. Тарасенко-Клятченко. – К. : “Корнійчук”, 2006. – 164 с.
41. Коутинхо, С. Введение в теорию чисел. Алгоритм RSA [Текст] / С. Коутинхо. – Москва : Постмаркет, 2001. – 328 с.
42. Ленков, С.В. Методы и средства защиты информации [Текст]. В 2-х томах. Том I. Несанкционированное получение информации / С.В. Ленков, Д.А. Перегудов, В.А. Хорошко; под ред. В.А. Хорошко. – К. : Арий, 2008. – 464 с. : ил.
43. Ленков, С.В. Методы и средства защиты информации [Текст]. В 2-х томах. Том II. Информационная безопасность / С.В. Ленков, Д.А. Перегудов, В.А. Хорошко; под ред. В.А. Хорошко. – К. : Арий, 2008. – 344 с. : ил.
44. Мак-Вильямс, Ф.Дж. Теория кодов, исправляющих ошибки [Текст] / Мак-Вильямс Ф.Дж., Слоэн Н. Дж. А.; пер. с англ. – М. : Связь, 1979. – 744 с., ил.
45. Мао, В. Современная криптография: теория и практика [Текст] / В. Мао; пер. с англ. – М. : Издательский дом «Вильямс», 2005. – 768 с. : ил. – Парал. тит. англ.
46. Масленников, М.Е. Практическая криптография [Текст] / М.Е. Масленников/ – СПб.: БХВ-Петербург, 2003. – 464 с. : ил.
47. Молдовян, А.А. Криптография [Текст] / А.А. Молдовян, Н.А. Молдовян, Б.Я. Советов. – СПб. : Издательство «Лань», 2001. – 224 с. : ил.

48. Молдовян, Н.А. Введение в криптосистемы с открытым ключом [Текст] / Н.А. Молдовян, А.А. Молдовян. – СПб. : БХВ-Петербург, 2005. – 288 с. : ил.
49. Николайчук Я.М. Коди поля Галуа: теорія та застосування : монографія [Текст] / Я.М. Николайчук. – Тернопіль : Вид-во ТНЕУ. – 2012. – 576 с.
50. Онай, М.В. Алгоритми виконання низькорівневих операцій на еліптичній кривій [Текст] / М.В. Онай, О.С. Князькіна // Міжнародна науково-практична конференція «Проблеми інформатики та комп'ютерної техніки». Праці конференції. – Чернівці : Видавничий дім "Родовід", 2014. – С. 87-89.
51. Онай, М.В. Віконні методи множення точки еліптичної кривої на число [Текст] / М.В. Онай, А.В. Соколовська // Міжнародна науково-практична конференція «Проблеми інформатики та комп'ютерної техніки». Праці конференції. – Чернівці : Видавничий дім "Родовід", 2015. – С. 93-95.
52. Онай, М.В. Ефективний алгоритм множення точки еліптичної кривої над основним полем Галуа з використанням системи числення з подвійною основою [Текст] / М.В. Онай, А.В. Соколовська // Інтелектуальні технології в системному програмуванні (ІТСП-2015). IV Всеукраїнська науково-практична конференція молодих учених та студентів. Збірник наукових праць. Хмельницький, 22-24 квітня 2015 року. – Хмельницький : ПП Гонта А. С., 2015. – С. 52.
53. Онай, М.В. Знаходження мультиплікативно оберненого елемента у кільці лишків за довільним модулем методом Джої-Пейє [Текст] / М.В. Онай, А.Ю. Бартков'як // Міжнародна науково-практична конференція «Проблеми інформатики та комп'ютерної техніки». Праці конференції. – Чернівці : Видавничий дім "Родовід", 2015. – С. 91-93.

54. Онай, М.В. Ієрархічна модель виконання операцій еліптичної криптографії над полем $GF(p)$ [Текст] / М.В. Онай, Т.П. Дрозда // Тези доповідей Четвертої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Вінниця : ВНТУ, 2014. – С. 176-179.
55. Онай, М.В. Модифікований метод однократного множення точки еліптичної кривої на ціле число [Текст] / М.В. Онай, О.С. Князькіна // Тези доповідей Четвертої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Вінниця : ВНТУ, 2014. – С. 172-175.
56. Онай, М.В. Оцінка стійкості криптографічних протоколів, заснованих на складності задачі дискретного логарифмування у скінченному полі $GF(p^m)$ [Текст] / М.В. Онай, А.І. Дичка // Тези доповідей П'ятої Міжнародної науково-практичної конференції «Методи та засоби кодування, захисту й ущільнення інформації». м. Вінниця, 19-21 квітня 2016 року. – Вінниця : ТОВ "Нілан-ЛТД", 2016. – С. 94-97.
57. Онай, М.В. Пошук мультиплікативно оберненого елементу у кільці лишків за довільним модулем методами, що грунтуються на модулярному піднесенні до степеня [Текст] / М.В. Онай, А.Ю. Бартков'як // Шістнадцята міжнародна наукова конференція імені академіка Михайла Кравчука, 14-15 травня, 2015 р., Київ : Матеріали конф. Т. 2. Алгебра. Геометрія. Математичний аналіз. – К. : НТУУ "КПІ", 2015. – С. 139-141.
58. Онай, М.В. Скалярне множення точки еліптичної кривої у полі $GF(p)$ з поданням множника у системі числення з подвійною основою [Текст] / М.В. Онай, А.В. Соколовська // Матеріали статей П'ятої Міжнародної науково-практичної конференції «Інформаційні технології та комп'ютерна інженерія». м. Івано-Франківськ : п. Голіней О.М., 2015. – С. 178-180.

59. Онай, М.В. Спосіб апаратної реалізації операцій над елементами поля $GF(2^m)$ з використанням логарифма Зеча [Текст] / М.В. Онай, А.І. Дичка // Інтелектуальні технології в системному програмуванні (ІТСП-2015). IV Всеукраїнська науково-практична конференція молодих учених та студентів. Збірник наукових праць. Хмельницький, 22-24 квітня 2015 року. – Хмельницький : ПП Гонта А. С., 2015. – С. 51.
60. Онай, М.В. Спосіб генерування випадкового простого числа заданої довжини / М.В. Онай, О.С. Князькіна // П’ятнадцята всеукраїнська (десята міжнародна) студентська наукова конференція з прикладної математики та інформатики СНКПМІ-2012 : Тези доповідей. – Львів : ЛНУ 2012. – С. 52-54.
61. Онай, М.В. Спосіб перетворення многочленного подання елементів поля $GF(p^m)$ у степеневе / М.В. Онай, Ю.В. Вальчук // Шістнадцята всеукраїнська (одинадцята міжнародна) студентська наукова конференція з прикладної математики та інформатики СНКПМІ-2013 : Тези доповідей, 11-12 квітня, 2013 р. – Львів : ЛНУ 2013. – С. 46-47.
62. Онай, М.В. Спосіб прискорення алгоритмів множення точки еліптичної кривої на число в полі $GF(p)$ [Текст] / М.В. Онай, О.С. Князькіна // Прикладна математика та комп’ютинг. ПМК, 2014 : шоста наук. конф. магістрантів та аспірантів, Київ, 16-18 квітня 2014 р. : зб. тез доп. / [ред кол.: Дичка І.А. та ін.]. – К. : Просвіта, 2014. – С. 148-154.
63. Онай, М.В. Узагальнений метод скалярного множення точки еліптичної кривої над полем $GF(p)$ у системі числення з мультиосновою [Текст] / М.В. Онай, Т.П. Дрозда // Прикладна математика та комп’ютинг – ПМК-2016 : восьма наук. конф. магістрантів та аспірантів, Київ, 20-22 квітня 2016 р. : зб. тез доп. / [ред кол. : Дичка І.А. та ін.]. – К. : Просвіта, 2016. – С. 218-225.

64. Опанасенко, В.Н. Высокопроизводительные реконфигурируемые компьютеры на базе FPGA [Текст] / В.Н. Опанасенко // Проблеми інформатизації та управління, Том 3, Випуск 27. – 2012. – С. 114-118.
65. Оре, О. Приглашение в теорию чисел [Текст] / О. Оре; пер. с англ. – Изд. 2-е, стереотипное. – М. : Едиториал УРСС, 2003. – 128 с.
66. Ростовцев, А.Г. Теоретическая криптография [Текст] / А.Г. Ростовцев, Е.Б. Маховенко. – НПО “Профессионал”, Санкт-Петербург, 2004. – 480 с.
67. Столлингс, В. Криптография и защита сетей: принципы и практика [Текст] / В. Столлингс. – 2-е издание, пер. с англ. – М. : Издательский дом “Вильямс”, 2001. – 672 с.
68. Сушкевич, А.К. Теория чисел. Элементарный курс [Текст] / А.К. Сушкевич. – М. : Вузовская книга, 2007. – 240 с.
69. Тарасенко, В.П. Штрихові коди та їх застосування [Текст] / В.П. Тарасенко, І.А. Дичка, В.І. Голуб. – К. : “Корнійчук”, 2000. – 175 с., іл.
70. Толов А.В. Авторское свидетельство СССР №1635193, кл. G06F 15/31. Вычислительное устройство в поле Галуа $GF(2^n)$ [Текст] / А.В. Толов, Б.А. Савельев, Н.Б. Залялов, С.Н. Комраков, Н.И. Басманов. – №4689561/24 ; заявл. 11.05.1989 ; опубл. 07.08.1992, Бюл. №29.
71. Торстейнсон, П. Криптография и безопасность в технологи .NET [Текст] / П. Торстейнсон, Г. А. Ганеш; пер. с англ. – М. : БИНОМ. Лаборатория знаний, 2007. – 479 с. : ил.
72. Уилкинсон, Б. С. Основы проектирования цифровых схем [Текст] / Б.С. Уилкинсон. – Пер. с англ. – М. : Издательский дом “Вильямс”, 2004. – 320 с.
73. Уоррен, Г.С. Алгоритмические трюки для программистов [Текст] / Г.С. Уоррен. – Испр. изд. ; пер. с англ. – М. : Издательский дом “Вильямс”, 2004. – 288 с.

74. Alyabieva, V.G. Applications of the Finite Fields [Text] / V.G. Alyabieva // Ярославский педагогический вестник. Том III (Естественные науки). – 2012 – №3.
75. Amanor, D.N. Efficient Hardware Architectures for Modular Multiplication on FPGAs [Text] / D.N. Amanor, C. Paar, J. Pelzl, V. Bunimov, M. Schimmler // In: 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, IEEE Circuits and Systems Society, Piscataway, New Jersey, August 2005. – pp. 539-542.
76. Anderson, R.J. Security Engineering: A Guide to Building Dependable Distributed Systems [Text] / R.J. Anderson. – 2nd Edition. – Published April 1st 2008 by John Wiley & Sons. – 1040 p.
77. Bardis, N.G. Fast implementation zero knowledge identification schemes using the Galois Fields arithmetic [Text] / N.G. Bardis, O.P. Markovskyi, N. Doukas, F. Drigas // Proceeding of IX International Symposium IEEE on Telecommunications – BIHTEL-2012, October 25-27, 2012, Sarajevo, Bosnia and Herzegovina.
78. Batina, L. Hardware architectures for public key cryptography [Text] / L. Batina, S.B. Ors, B. Preneel, J. Vandewalle // Elsevier, INTEGRATION, the VLSI journal 34 (2003). – P. 1-64.
79. Benvenuto, C.J. Galois Field in Cryptography [Electronic resource] / Christoforus Juan Benvenuto. – 2012. – Mode of access : https://sites.math.washington.edu/~morrow/336_12/papers/juan.pdf – Last access : April 2017.
80. Bishop, M. Introduction to Computer Security [Text] / M. Bishop. – Addison-Wesley, 2004.
81. Blahut, R.E. Theory and Practice of Error Control Codes [Text] / R.E. Blahut. – Addison-Wesley. – 1983. – 500 p.

82. Bojanczyk, A.W. A systolic algorithm for extended GCD [Text] / A.W. Bojanczyk, R.P. Brent // Comput. Math. Applic. – Vol. 14, No. 4. – 1987. – P. 233-238.
83. Bradley G.H. Algorithm and bound for the greatest common divisor of n integers / G.H. Bradley // Communications of the ACM. – Volume 13 Issue 7. – 1970. – P. 433-436.
84. Bunimov, V. Area and time efficient modular multiplication of large integers [Text] / V. Bunimov, M. Schimmler // IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands, June 2003.
85. Cancellieri, G. Polynomial Theory of Error Correcting Codes [Text] / G. Cancellieri. – Springer, 2015. – 751 p.
86. Christophe, D. Progress in Cryptology-Indocrypt [Text] / Christophe Doche, Laurent Imbert. – Berlin Heidelberg : Springer, 2006. – 348 p.
87. Cocke, J. High Speed Arithmetic in a Parallel Device [Text] / J. Cocke, D.W. Sweeney // Technicak Report IBM, Feb. 1957.
88. Cohen, H. A Course in Computational Algebraic Number Theory [Text] / C. Henri. – Springer Science & Business Media. – 1993. – 534 p.
89. Cohen, H. Handbook of Elliptic and Hyperelliptic Curve Cryptography [Text] / Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, Frederik Vercauteren. – Chapman and Hall/CRC. – 2006. – 842 p.
90. Crandall, R. Prime Numbers: A Computational Perspective [Text] / Richard Crandall, Carl Pomerance. – Springer-Verlag, New York, USA. – 2005. – 597 p.
91. Deschamps, J.-P. Hardware Implementation of Finite-Field Arithmetic [Text] / J.-P. Deschamps, J. L. Imana, G. D. Sutter. – McGraw-Hill, 2009. – 347 p.
92. Deschamps, J.-P. Synthesis of Arithmetic Circuits / J.-P. Deschamps, G. Bioul, G. Sutter. – Wiley, Hoboken, New Jersey, 2006. – 556 p.

93. Dickson, L.E. Of Finite Algebras [Text] / L.E. Dickson // Nachrichten von der Königlichen Gesellschaft der Wissenschaften und der Georg August Universität zu Göttingen. – 1905. – P. 358-393.
94. Diffie, W. New directions in cryptography [Text] / W. Diffie, M. E. Hellman // IEEE Transactions on Information Theory, IT-22, 1976. – P. 644-654.
95. Dimitrov, V. Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains [Text] / Vassil Dimitrov, Laurent Imbert, Pradeep Kumar Mishra. – Heidelberg : Springer, 2005. – 20 p.
96. Doche, C. Extended Double-Base Number System with Applications to Elliptic Curve Cryptography [Text] / Doche C., Imbert L. – Heidelberg : Springer, 2006. – P. 335-348.
97. Ercegovac, M.D. Digital Arithmetic / M.D. Ercegovac, T. Lang, M. Kaufmann. – San Francisco, 2004. – 710 p.
98. Euclid's Elements : all thirteen books complete in one volume [Text] / Thomas L. Heath translation, Dana Densmore editor // Santa Fe, N.M. : Green Lion Press, 2002. – 499 p.
99. Ferguson, N. Cryptography Engineering: Design Principles and Practical Applications [Text] / N. Ferguson, B. Schneier, T. Kohno; 1st Edition, Wiley Pub., Incorporated, 2010. – 353 p.
100. Fischer, W. Note on fast computation of secret RSA exponents [Text] / Fischer W., Seifert J.-P. // Information Security and Privacy (ACISP 2002), vol. 2384 of Lecture Notes in Computer Science, Springer-Verlag. – 2002. – P. 136–143.
101. Galbraith, S.D. Mathematics of Public Key Cryptography [Text] / Steven D. Galbraith. – Cambridge University, 2012. – 615 p.
102. Guerric Meurice de Dormale Efficient Modular Division Implementation ECC over *GF*(*p*) Affine Coordinates Application [Text] / Meurice de Dormale Guerric, Bulens Philippe, Quisquater Jean-Jacques // The 14th International Conference on Field Programmable Logic and Applications,

FPL 2004, Volume 3203 of Lecture Notes in Computer Science. – 2004. – P. 231-240.

103. Hankerson, D. Guide to Elliptic Curve Cryptography [Text] / D. Hankerson, A. Menezes, S. Vanstone // Springer-Verlag New York, USA. – 2004. – 311 p.
104. Henk C. A. van Tilborg Encyclopedia of Cryptography and Security [Text] / Henk C. A. van Tilborg. – Springer, August 10, 2005. – 684 p.
105. Hennesy, J.L. Computer Architecture: A Quantitative Approach [Text] / J.L. Hennesy, D.A. Patterson. – 5th Edition. – Imprint: Morgan Kaufmann, 2011. – 856 p.
106. Houghton, A. The Engineer's Error Coding Handbook [Text] / A. Houghton. – Springer, 1997. – 272 p.
107. Joye, M. GCD-Free Algorithms for Computing Modular Inverses [Text] / Joye Marc, Paillier Pascal // CHES 2003: Cologne, Germany. – 2003. – P. 243-253.
108. Katz, J. Introduction to Modern Cryptography: Principles and Protocols [Text] / J. Katz, Y. Lindell. Second Edition – Chapman & Hall / CRC Cryptography and Network Security Series. – 2014. – 582 p.
109. Kisos, P. An Efficient Reconfigurable Multiplier Architecture for Galois Field $GF(2^m)$ [Text] / P. Kisos, G. Thedoridis, O. Koufopavlou // Elsevier, Microelectronics Journal 34 (2003), P. 975-980.
110. Knuth D.E. Art of Computer Programming. Volume 2: Seminumerical Algorithms / D.E. Knuth. – 3rd Edition by Addison-Wesley Professional, Canada. – 1997. – 784 p.
111. Koç, Ç.K. High Speed RSA Implementation (Tech. Rep.) [Text] / Çetin Kaya Koç. – RSA Laboratories, Tech. Rep. Version 2.0, 1994.
112. Kumar, L. Implementation of Galois Field Arithmetic Unit on *FPGA* [Text] / L. Kumar, K. Sudha // International Journal of Innovative Research in Computer and Communication Engineering. Vol. 2, Issue 6, June 2014, P. 4709-4715.

113. Lee, C.Y. Efficient Bit-Parallel Multipliers over Finite Fields $GF(2^m)$ [Text] / Chiou-Yng Lee, Pramod Kumar Meher // Computers and Electrical Engineering. – Volume 36, Issue 5, September 2010, P. 955-968.
114. Lidl, R. Finite Fields (Encyclopedia of Mathematics and its Applications Volume 20) [Text] / Rudolf Lidl, Harald Niederreiter. – 2nd Edition. – Cambridge University Press, 1997. – 755 p.
115. Lorencz, R. New Algorithm for Classical Modular Inverse [Text] / R. Lorencz // Cryptographic Hardware and Embedded Systems. International Workshop. – 2002. – P. 57-70.
116. Menezes, A. Handbook of Applied Cryptography [Text] / A. Menezes, P. van Oorschot, S. Vanstone. – Published October 16th 1996 by CRC Press, 810 p.
117. Menezes, A.J. Application of Finite Fields [Text] / A.J. Menezes, I.F. Blake, S. Gao, R.C. Mullin, S.A. Vanstone, T. Yactobian // N. Y., Knuwer Academic Published. – 1993. – 387 p.
118. Miller, V.S. Use of Elliptic Curves in Cryptography [Text] / V.S. Miller // Advances in Cryptology – CRYPTO’85 Proceedings. CRYPTO 1985. Lecture Notes in Computer Science, vol. 218. – Springer, Berlin, Heidelberg. – P. 417–426.
119. Milne, J.S. Fields and Galois Theory / J.S. Milne // Version 4.52, March 17, 2017 [Electronic resource]. – Mode of access : http://www.jmilne.org/math/CourseNotes/FT.pdf – Last access : 07.05.2017.
120. Montgomery, P.L. Modular Multiplication Without Trial Division [Text] // Mathematics of Computation, Vol. 44, №. 170. – Apr., 1985. – P. 519-521.
121. Morales-Sandoval, M. An area/performance trade-off analysis of a $GF(2^m)$ multiplier architecture for elliptic curve cryptography [Text] / Miguel Morales-Sandoval, Claudia Feregrino-Uribe, René Cumplido, Ignacio Algredo-Badillo // Computers and Electrical Engineering. – Volume 35, Issue 1, 2009. – P. 54-58.

122. Moustafa, A.A. Fast Exponentation in Galois Fields GF(2^m) Using Precomputations [Text] // Contemporary Engineering Sciences. – Vol. 7, №. 4, 2014. – P. 193-206.
123. Nedjah, N. Efficient Pre-Processing for Large Window-Based Modular Exponentiation Using Ant Colony [Text] / Nadia Nedjah, Luiza de Macedo Mourelle // Knowledge-Based Intelligent Information and Engineering Systems. KES 2005. Lecture Notes in Computer Science, vol 3684. Springer, Berlin, Heidelberg. – P. 640-646.
124. Onai, M.V. Modification of Norton's extended algorithm for search of multiplicative inverse element modulo m [Text] / M.V. Onai, T.P. Drozda // П'ятнадцята міжнародна наукова конференція імені академіка М. Кравчука, 15-17 травня, 2014 р., Київ : Матеріали конф. Т. 2. Алгебра. Геометрія. Математичний аналіз. – К. : НТУУ "КПІ", 2014. – С. 25-26.
125. Parhami, B. Computer Arithmetic: Algorithms and Hardware Designs [Text] / B. Parhami. – Oxford University Press, New York, 2000. – 490 p.
126. Parthasarathy, S. Multiplicative inverse in mod (*m*) / S. Parthasarathy // Algologic Technical Report #1/2012. – P. 1-3.
127. Patel, V. Arithmetic operations in Multi-Valued logic [Text] / V. Patel, K.S. Gurumurthy // International Journal of VLSI design & Communication Systems (VLSICS). – Vol. 1. – №. 1, March 2010. – P. 21-32.
128. Popovici, E. Algorithm and architecture for a multiplicative Galois field processor [Text] / E. Popovici, P. Fitzpatrick // IEEE Transactions on Information Theory, Vol. 49, Issue 12. – 2003. – P. 3303-3307.
129. Rajesh, K.N. A reversible visual cryptography technique for color images using Galois field arithmetic [Text] / K.N. Rajesh, G. Manikandan, K.R. Bala, N.R. Raajan, N. Sairam // Biomedical Research, Vol. 28, Issue 5. – 2017. – P. 2036-2039.

130. Rivain, M. Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves [Text] / Matthieu Rivain. – IACR Cryptology ePrint Archive, 2011. – 338 p.
131. Rivest, R. L. A Method for obtaining digital signatures and public-key cryptosystems [Text] / R. L. Rivest, A. Shamir, L. Adleman // Comm. ACM, Vol. 21, Issue 2. – 1978. – P. 120-126.
132. Robertson, J.E. A New Class of Digital Division Methods [Text] / J.E. Robertson // IEEE Transactions on Computers, Electronic Computers, EC-7, Sep. 1958, P. 218-222.
133. Saeed, M. Methods of finding multiplicative inverses in GF(2^8) [Text] / M. Saeed, M.S. Mian // Computer Communications, Vol. 31, Issue 17. – 2008. P. 4117-4123.
134. Savas E. Finite field arithmetic for cryptography [Text] / Erkay Savaş, Çetin Kaya Koç // IEEE Circuits and Systems Magazine. – Vol. 10, № 2. – 2010. – P. 40-56.
135. Schneier, B. Applied Cryptography: Protocols, Algorithms, and Source Code in C [Text] / B. Schneier; Second Edition. – Published November 2nd 1995 by Wiley, 784 p.
136. Sorenson J. Two fast GCD algorithms [Text] / J. Sorenson // Journal of Algorithms, Vol. 16, Issue 1. – 1994. – P. 110-144.
137. Stinson D.R. Discrete Mathematics and its applications: Cryptography Theory and Practice [Text] / D.R. Stinson. – Third Edition. – Chapman & Hall / CRC, Taylor & Francis Group, Boca Raton, London, New York. – 2006. – 593 p.
138. Tocher, K.D. Techniques of Multiplication and Division for Automatic Binary Computers [Text] / K.D. Tocher //Quart. J. Mech. Appl. Math., Vol. 11, Issue 3. – 1958. – P. 364-384.
139. Verbik, K.V. Factorization algorithms for cryptographic analysis of asymmetric crypto systems [Text] / K.V. Verbik, Y.M. Nikolaichuk, S.V. Ivasiev, L.M. Timoshenko // Informatics and mathematical methods in modelling.– Vol. 4, № 4. – 2014. – P. 342-348.

140. WolframMathWorld [Electronic resource]. – Mode of access : http://mathworld.wolfram.com/CarmichaelFunction.html – Last access : 03.08.2015.
141. Xilinx Inc. Website & Data Sheets [Electronic resource]. – http://www.xilinx.com – Last access : 07.05.2016.
142. Yatskiv V. Two-Dimensional Corrective Codes Based on Modular Arithmetic / Vasyl Yatskiv, Taras Tsavolyk // The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM), 13th International Conference, 2015. – P. 291-294.
143. Zhengbing, Hu The Analysis and Investigation of Multiplicative Inverse Searching Methods in the Ring of Integers Modulo M / Hu Zhengbing, I.A. Dychka, Onai Mykola, Bartkoviak Andrii // International Journal of Intelligent Systems and Applications (IJISA), 2016. – Vol. 8, No. 11. – P. 9-18.

ДОДАТКИ

ДОДАТОК А

Алгоритми реалізації методів виконання операцій у скінченних полях

Алгоритм А.1. Узагальнений алгоритм цілочисельного ділення

Вхід : $x \in \mathbb{N}, m \in \mathbb{N}$

Вихід : $r = x \pmod m$

1. $w \leftarrow x$
2. $y \leftarrow m \cdot 2^{\tilde{n}-\tilde{k}}$
3. *for* i *from* 0 *to* $\tilde{n}-\tilde{k}$ *do*
   3.1. *if* $quotient(w; y) = 1$ *then* $r \leftarrow w - y$
   3.2. *elseif* $quotient(w; y) = 0$ *then* $r \leftarrow w$
   3.3. *else* $r \leftarrow w + y$
   3.4. $w \leftarrow 2r$
3. *end for*
4. $r \leftarrow \left\lfloor \frac{r}{2^{\tilde{n}-\tilde{k}}} \right\rfloor$
5. *if* $r < 0$ *then* $r \leftarrow r + m$
6. *return* r

Алгоритм А.2. Узагальнений алгоритм цілочисельного ділення

Вхід: $x \in \mathbb{N}, m \in \mathbb{N}$

Вихід: $r = x \pmod m$

1. $w \leftarrow x$
2. $y \leftarrow m \cdot 2^{n-2}$
3. *for* i *from* 0 *to* $n-2$ *do*
   3.1. *if* $w \geq 0$ *then* $r \leftarrow w - y$
   3.2. *else* $r \leftarrow w + y$
   3.3. $w \leftarrow 2r$
3. *end for*
4. $r \leftarrow \left\lfloor \frac{r}{2^{n-2}} \right\rfloor$
5. *if* $r < 0$ *then* $r \leftarrow r + m$
6. *return* r

Алгоритм А.3. Алгоритм перетворення числа з двійкової системи числення зі знаком у звичайну двійкову систему числення

Вхід: $q' = \left(q'_{n-2}, ..., q'_1, q'_0\right)_{2\ with\ sign}$

Вихід: $q = \left(q_{n-1}, ..., q_1, q_0\right)_2$

1. $q_0 \leftarrow 1$
2. *for* i *from* 1 *to* $n-2$ *do*
   2.1. *if* $q'_{i-1} = -1$ *then* $q_i \leftarrow 0$
   2.2. *else* $q_i \leftarrow 1$
2. *end for*
3. *if* $q'_{n-2} = -1$ *then* $q_{n-1} \leftarrow 1$
4. *else* $q_{n-1} \leftarrow 0$
5. *return* q

Алгоритм А.4. Узагальнений алгоритм цілочисельного ділення з отримання частки

Вхід: $x \in \mathbb{N}, m \in \mathbb{N}$

Вихід: $r = x \pmod m, q = \left\lfloor \frac{x}{m} \right\rfloor$

1. $w \leftarrow x$
2. $y \leftarrow m \cdot 2^{n-2}$
3. *for* i *from* $n-1$ *downto* 1 *do*
   3.1. *if* $w \geq 0$ *then* $r \leftarrow w - y, q_i \leftarrow 1$
   3.2. *else* $r \leftarrow w + y, q_i \leftarrow 0$
   3.3. $w \leftarrow 2r$
3. *end for*
4. $r \leftarrow \left\lfloor \frac{r}{2^{n-2}} \right\rfloor, q_0 \leftarrow 1, q_{n-1} \leftarrow 1 - q_{n-1}$
5. *if* $r < 0$ *then* $r \leftarrow r + m, q \leftarrow q - 1$
6. *return* r, q

Алгоритм А.5. Звичайний бінарний *RL*-алгоритм піднесення до степеня

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1}, ..., k_1, k_0)_2 \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow 1$
2. *for* i *from* 0 *to* $l-1$ *do*
   2.1. *if* $k_i = 1$ *then* $b \leftarrow b \cdot a \pmod m$
   2.2. $a \leftarrow a^2 \pmod m$
3. *end for*
4. *return* b

Алгоритм А.6. Звичайний бінарний *LR*-алгоритм піднесення до степеня

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_2 \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow 1$
2. *for* i *from* $l-1$ *downto* 0 *do*
   2.1. $b \leftarrow b^2 \pmod m$
   2.2. *if* $k_i = 1$ *then* $b \leftarrow b \cdot a \pmod m$
3. *end for*
4. *return* b

Алгоритм А.7. *RL*-алгоритм Джої

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_2 \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow 1, R_1 \leftarrow a, R_0 \leftarrow b$
2. *for* $i = 0$ *to* $l-1$ *do*
   2.1. $R_{1-k_i} \leftarrow R_{1-k_i}^2 \cdot R_{k_i}$
3. *end for*
4. $b \leftarrow R_0$
5. *return* b

Алгоритм А.8. *LR*-алгоритм Монтогомері

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_2$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow 1, R_1 \leftarrow a, R_0 \leftarrow b$
2. *for* $i = l-1$ *downto* 0 *do*
   2.1. $R_{1-k_i} \leftarrow R_{1-k_i} \cdot R_{k_i}$
   2.2. $R_{k_i} \leftarrow R_{k_i}^2$
3. *end for*
4. $b \leftarrow R_0$
5. *return* b

Алгоритм А.9. Бінарний віконний *RL*-алгоритм

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_{2^w}$

Вихід: $b = a^k \pmod m$

1. *for* $i = 1$ *to* $2^w - 1$ *do*
   1.1. $Table[i] \leftarrow a^i \pmod m$
2. *end for*
3. $b \leftarrow 1$
4. *for* $i = 0$ *to* $l - 1$ *do*
   4.1. *if* $k_i > 0$ *then* $b \leftarrow b \cdot Table[k_i] \pmod m$
   4.2. $Table \leftarrow Table^{2^w} \pmod m$
5. *end for*
6. *return* b

Алгоритм А.10. Бінарний віконний *LR*-алгоритм

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_{2^w}$

Вихід: $b = a^k \pmod m$

1. *for* $i = 1$ *to* $2^w - 1$ *do*
   1.1. $Table[i] \leftarrow a^i \pmod m$
2. *end for*
3. $b \leftarrow 1$
4. *for* $i = l - 1$ *downto* 0 *do*
   4.1. $b \leftarrow b^{2^w} \pmod m$
   4.2. *if* $k_i > 0$ *then* $b \leftarrow b \cdot Table[k_i] \pmod m$
5. *end for*
6. *return* b

Алгоритм А.11. Бінарний *LR*-алгоритм з ковзним вікном

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = (k_{l-1},...,k_1,k_0)_2$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow 1,\ i \leftarrow \lceil \log_2 k \rceil$
2. *for* $i = 0$ *to* $2^{w-1} - 1$
   2.1. $Table[i] \leftarrow a^{2^{w-1}+i} \pmod m$
3. *end for*
4. *while* $i > 0$ *do*
   4.1. *if* $(k_i = 0)$ *then* $b = b^2 \pmod m,\ i = i - 1$
   4.2. *else*
      4.2.1. *if* $i \geq w - 1$ *then*
         a) $t \leftarrow (k_i,...,k_{i-w+1})_2$
         b) $b \leftarrow b^{2^w} \pmod m$
         c) $b \leftarrow b \cdot Table[t - 2^{w-1}] \pmod m$
      4.2.2. *else*
         a) Викликати бінарний *LR*-алгоритм
      4.2.3 $i \leftarrow i - w$
5. *end while*
6. *return Q*

Алгоритм А.12. Перетворення додатного цілого числа *k* у *NAF*-подання

Вхід: $k = (k_{l-1},...,k_1,k_0)_2 \in \mathbb{N}$

Вихід: $NAF(k)$

1. $i \leftarrow 0$
2. while $k \geq 1$ do
   2.1. *if* k є непарним *then*
      2.1.2. $k_i \leftarrow 2 - (k \bmod 4)$
      2.1.3. $k \leftarrow k - k_i$
   2.2. else $k_i \leftarrow 0$
   2.3. $k \leftarrow \frac{k}{2}$
   2.4. $i \leftarrow i + 1$
3. *end while*
4. *return* $\{k_{l-1}, k_{l-2},..., k_1, k_0\}$.

Алгоритм А.13. *RL*-алгоритм з поданням показника степеня у вигляд *NAF*

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. Використати алгоритм Д.12 для обчислення $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$
2. $b \leftarrow 1$
3. *for* i *from* 0 *to* $l-1$ *do*
   3.1. *if* $k_i = 1$ *then* $b \leftarrow b \cdot a \pmod m$
   3.2. *if* $k_i = -1$ *then* $b \leftarrow \frac{b}{a} \pmod m$
   3.3. $a \leftarrow a^2$
4. *return b.*

Алгоритм А.14. *LR*-алгоритм з поданням показника степеня у вигляді *NAF*

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. Використати алгоритм А.12 для обчислення $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$

2. $b \leftarrow 1$

3. *for* i *from* $l-1$ *downto* 0 *do*

   3.1. $b \leftarrow b^2$

   3.2. *if* $k_i = 1$ *then* $b \leftarrow b \cdot a$

   3.3. *if* $k_i = -1$ *then* $b \leftarrow \frac{b}{a}$

4. *return b.*

Алгоритм А.15. Перетворення додатного цілого числа *k* у *wNAF* подання

Вхід: додатне ціле *k*, ширина вікна *w*
Вихід: *wNAF(k)*
1. $i \leftarrow 0$
2. *while* $k \geq 1$ *do*
   2.1. *if* k є непарним *then*
      2.1.1. $t_i \leftarrow k \bmod 2^w$
      2.1.2. *if* $t_i \geq 2^w$ *then* $t_i \leftarrow t_i - 2^w$
      2.1.3. $k \leftarrow k - t_i$
   2.2. *else*
      2.2.1. $t_i \leftarrow 0$
   2.3. $k \leftarrow \frac{k}{2}, i \leftarrow i+1$
3. *end while*
4. *return* $\{t_{n-1}, t_{n-2}, ..., t_1, t_0\}$

Алгоритм А.16. *LR*-алгоритм з поданням показника степеня у вигляд *wNAF*

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}$, ширина вікна w, $k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. Використати алгоритм А.15 для обчислення $wNAF(k) = \sum_{i=0}^{n-1} t_i 2^i$
2. *for* $i = 1$ *to* $2^{w-1} - 1$ з кроком 2 *do*
   2.1. $Table\left[\frac{i+1}{2}\right] = a^i \pmod m$
3. *end for*
4. $b \leftarrow 1$
5. *for* $i = n-1$ *downto* 0 *do*
   5.1. $b \leftarrow b^2 \pmod m$
   5.2. *if* $t_i \neq 0$ *then*
      5.2.1. *if* $t_i > 0$ *then* $b \leftarrow b \cdot Table\left[\frac{t_i+1}{2}\right] \pmod m$
      5.2.2. *else* $b \leftarrow \frac{b}{Table\left[\frac{t_i+1}{2}\right]} \pmod m$
6. *end for*
7. *return* b

Алгоритм А.17. *LR*-алгоритм з вікном змінної довжини та поданням показника степеня у вигляді *NAF*

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}$, ширина вікна w, $k \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. Використати алгоритм А.12 для обчислення $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$
2. *for* $i = 1$ *to* $\frac{2 \cdot (2^w - (-1)^w)}{3} - 1$ з кроком 2 *do*
   2.1. $Table\left[\frac{i+1}{2}\right] = a^i \pmod m$
3. *end for*
4. $b \leftarrow 1,\ i \leftarrow l - 1$
5. *while* $i \geq 0$ *do*
   5.1. *if* $k_i = 0$ *then* $t \leftarrow 1,\ u \leftarrow 0$
   5.2. *else* знайти $\max(t) \leq w$ таке,
   що $u \leftarrow (k_i, ..., k_{i-t+1})$ є непарним
   5.3. $b \leftarrow b^{2^t}$
   5.4. *if* $u > 0$ *then* $b \leftarrow b \cdot Table[t] \pmod m$
   5.5. *else if* $u < 0$ *then* $b \leftarrow \frac{b}{Table[t]} \pmod m$
   5.6. $i \leftarrow i - t$
6. *end while*
7. *return* Q

Алгоритм А.18. Жадібний алгоритм перетворення в *DBNS*

Вхід: $k, x_{\max}, y_{\max} > 0.$

Вихід: Послідовність (s_i, x_i, y_i) така, що $k = \sum_{i=1}^{l} s_i 2^{x_i} 3^{y_i}$, де

$x_1 \geq ... \geq x_l \geq 0$ і $y_1 \geq ... \geq y_l \geq 0.$

1. $s_i \leftarrow 1, i = 1$
2. *while* $k > 0$ *do*
   2.1. знайти значення x_i та y_i, такі що дають найкраще наближення до числа k виразом $2^{x_i} 3^{y_i}$
   2.2. $z = 2^{x_i} 3^{y_i}$
   2.3. $x_{\max} \leftarrow x_i, y_{\max} \leftarrow y_i$
   2.4. *if* $k < z$ *then* $s_{i+1} \leftarrow -s_i$
   2.5. $k \leftarrow |k - z|$
   2.6. $i \leftarrow i + 1;$
3. *end while*
4. *return* (s_i, x_i, y_i)

Алгоритм А.19. *LR*-алгоритм піднесення до степеня з поданням показника степеня у вигляді *DBNS*

Вхід: $k = \sum_{i=1}^{l} s_i 2^{x_i} 3^{y_i} \in \mathbb{N}$, де $s_i \in S$, і таке, що

$x_1 \geq ... \geq x_l \geq 0$ і $y_1 \geq ... \geq y_l \geq 0$, $a \in \mathbb{N}$, $m \in \mathbb{N}$

Вихід: $b = a^k \pmod m$

1. $b \leftarrow a^{s_1} \pmod m$
2. *for* $i = 1$ *to* $l - 1$ *do*
   - 2.1. $u \leftarrow x_i - x_{i+1}$
   - 2.2. $v \leftarrow y_i - y_{i+1}$
   - 2.3. $b \leftarrow b^{3^v} \pmod m$
   - 2.4. $b \leftarrow b^{2^u} \pmod m$
   - 2.5. $b \leftarrow b \cdot a^{s_{i+1}} \pmod m$
3. *end for*
4. *return b.*

Алгоритм А.20. Модифікований алгоритм цілочисельного ділення

Вхід : $x \in \mathbb{N}, m \in \mathbb{N}$

Вихід : $r = x \pmod m$

1. *if* $x < m$ *then* $r \leftarrow x$
2. *elseif* $x = m$ *then* $r \leftarrow 0$
3. *else*
   - 3.1. $w \leftarrow x;\ y \leftarrow m;\ t \leftarrow 0$
   - 3.2. *while* $\text{not}\left(\left(\left(x \text{ xor } y\right) > x\right) \text{and } y > x\right)$ *do*
     - 3.2.1. $y \leftarrow 2y;\ t \leftarrow t + 1$
   - 3.2. *end while*
   - 3.3. $y \leftarrow \left\lfloor \frac{y}{2} \right\rfloor$
   - 3.4. *for* i *from* 1 *to* t *do*
     - 3.4.1. *if* $w \geq 0$ *then* $r \leftarrow w - y$
     - 3.4.2. *else* $r \leftarrow w + y$
     - 3.4.3. $w \leftarrow 2r$
   - 3.4. *end for*
   - 3.5. $r \leftarrow \left\lfloor \frac{r}{2^{t-1}} \right\rfloor$
   - 3.6. *if* $r < 0$ *then* $r \leftarrow r + m$
4. *return r*

Алгоритм А.21. Модифікований алгоритм цілочисельного ділення з отриманням частки

Вхід: $x \in \mathbb{N}, m \in \mathbb{N}$

Вихід: $r = x \pmod m, q = \left\lfloor \frac{x}{m} \right\rfloor$

1. *if* $x < m$ *then* $r \leftarrow x$
2. *elseif* $x = m$ *then* $r \leftarrow 0$
3. *else*
   - 3.1. $w \leftarrow x; y \leftarrow m; t \leftarrow 0$
   - 3.2. *while* $\text{not}\left(\left(\left(x \text{ xor } y\right) > x\right) \text{and } y > x\right)$ *do*
     - 3.2.1. $y \leftarrow 2y; t \leftarrow t + 1$
   - 3.2. *end while*
   - 3.3. $y \leftarrow \left\lfloor \frac{y}{2} \right\rfloor$
   - 3.4. *for* i *from* t *downto* 1 *do*
     - 3.4.1. *if* $w \geq 0$ *then* $r \leftarrow w - y, q_i \leftarrow 1$
     - 3.4.2. *else* $r \leftarrow w + y, q_i \leftarrow 0$
     - 3.4.3. $w \leftarrow 2r$
   - 3.4. *end for*
   - 3.5. $r \leftarrow \left\lfloor \frac{r}{2^{t-1}} \right\rfloor, q_0 \leftarrow 1, q_t \leftarrow 1 - q_t$
   - 3.6. *if* $r < 0$ *then* $r \leftarrow r + m, q \leftarrow q - 1$
4. *return* r, q

Алгоритм А.22. *LR*-алгоритм реалізації узагальненого методу піднесення до степеня

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = \sum_{i=1}^{l} s_i \prod_{j=1}^{n} d_j^{c_{ij}}$,

де $s_i \in S, d_j \in D, c_{ij} \in C, S, D, C \in \mathbb{Z}$

Вихід: $b = a^k \pmod m$

1. *foreach i in S*

1.1. $Table[i] \leftarrow a^i \pmod m$

2. *end foreach*

3. $b = a$

4. *foreach i in decomposition*

4.1. *for* $j = 1$ *to D.Length do*

4.1.1. *for* $l = 1$ *to* $i.pows_j$ *do*

a) $b \leftarrow Pow(b, D[j]) \pmod m$

4.1.2. *end for*

4.2. *end for*

4.3. *if* $i.s != 0$ *then*

4.3.1. $b \leftarrow Multiplication(b, Table[i.s]) \pmod m$

4.4. *end if*

5. *end foreach*

6. *return b*

Алгоритм А.23. *RL*-алгоритм реалізації узагальненого методу піднесення до степеня

Вхід: $a \in \mathbb{N}, m \in \mathbb{N}, k = \sum_{i=1}^{l} s_i \prod_{j=1}^{n} d_j^{c_{ij}}$,

де $s_i \in S, d_j \in D, c_{ij} \in C, S, D, C \in \mathbb{Z}$

Вихід: $b = a^k \pmod m$

1. *foreach i in S*

1.1. $Table[i] \leftarrow a^i \pmod m$

2. *end foreach*

3. $b = 1$

4. *foreach i in decomposition*

4.1. *for* $j = 1$ *to D.Length do*

4.1.1. *for* $l = 1$ *to* $i.pows_j$ *do*

a) *foreach t in S*

a.a) $Table[t] \leftarrow Pow(Table[t], D[j]) \pmod m$

b) *end foreach*

4.1.2. *end for*

4.2. *end for*

4.3. *if* $i.s != 0$ *then*

4.3.1. $b \leftarrow Multiplication(b, Table[i.s])$

4.4. *end if*

5. *end foreach*

6. *return b*

ДОДАТОК Б

Приклади роботи алгоритмів реалізації методів виконання операцій у скінченних полях

Розглянемо приклади роботи алгоритмів. Тут і надалі, для наочності подання прикладу, не будемо виконувати зведення за модулем (це не впливає на основну ідею алгоритму).

Приклад роботи алгоритма А.5 для $a=5$ та $k=13_{10}=1101_2$.

1. Ініціалізація: $b=1$
2. $k_0=1$: $b=b\cdot a \Rightarrow b=1\cdot 5=5$
   $a=a^2 \Rightarrow a=5^2$
3. $k_1=0$: $a=a^2 \Rightarrow a=\left(5^2\right)^2=5^4$
4. $k_2=1$: $b=b\cdot a \Rightarrow b=5\cdot 5^4=5^5$
   $a=a^2 \Rightarrow a=\left(5^4\right)^2=5^8$
5. $k_3=1$: $b=b\cdot a \Rightarrow b=5^5\cdot 5^8=5^{13}$
   $a=a^2 \Rightarrow a=\left(5^8\right)^2=5^{16}$
6. Результат виконання операції піднесення до степеня знаходиться у змінній $b=5^{13}$.

RL-розкладання для $k=13_{10}=1101_2$:

$$a^{13}=a^{1\cdot 2^0+0\cdot 2^1+1\cdot 2^2+1\cdot 2^3}=a^1\cdot a^0\cdot a^4\cdot a^8$$

Приклад роботи алгоритма А.6 для $a = 5$ та $k = 13_{10} = 1101_2$.

1. Ініціалізація: $b = 1$
2. $k_0 = 1$: $b = b^2 \Rightarrow b = 1^2 = 1$
   $b = b \cdot a \Rightarrow b = 1 \cdot 5 = 5$
3. $k_1 = 1$: $b = b^2 \Rightarrow b = 5^2$
   $b = b \cdot a \Rightarrow b = 5^2 \cdot 5 = 5^3$
4. $k_2 = 0$: $b = b^2 \Rightarrow b = \left(5^3\right)^2 = 5^6$
5. $k_3 = 1$: $b = b^2 \Rightarrow b = \left(5^6\right)^2 = 5^{12}$
   $b = b \cdot a \Rightarrow b = 5^{12} \cdot 5 = 5^{13}$
6. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{13}$.

LR-розкладання для $k = 13_{10} = 1101_2$:

$$a^{13} = a^{2\cdot(2\cdot(2\cdot1+1))+1} = \left(\left(\left(1 \cdot a\right)^2 \cdot a\right)^2\right)^2 \cdot a$$

Приклад роботи *RL*-алгоритму Джої (алгоритм А.7) для $a = 5$ та $k = 13_{10} = 1101_2$.

1. Ініціалізація: $R_0 = 1, R_1 = 5$
2. $k_i = 1$: $R_0 = R_0^2 \cdot R_1 \Rightarrow R_0 = 1^2 \cdot 5 = 5$
3. $k_i = 0$: $R_1 = R_1^2 \cdot R_0 \Rightarrow R_1 = 5^2 \cdot 5 = 5^3$
4. $k_i = 1$: $R_0 = R_0^2 \cdot R_1 \Rightarrow R_0 = 5^2 \cdot 5^3 = 5^5$
5. $k_i = 1$: $R_0 = R_0^2 \cdot R_1 \Rightarrow R_0 = \left(5^5\right)^2 \cdot 5^3 = 5^{13}$
6. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{13}$.

Приклад роботи *LR*-алгоритму Монтогомері (алгоритм А.8) для $a = 5$ та $k = 13_{10} = 1101_2$.

1. Ініціалізація: $R_0 = 1, R_1 = 5$.
2. $k_i = 1$: $R_0 = R_0 \cdot R_1 \Rightarrow R_0 = 1 \cdot 5 = 5$
   $R_1 = R_1^2 \Rightarrow R_1 = 5^2$
3. $k_i = 1$: $R_0 = R_0 \cdot R_1 \Rightarrow R_0 = 5 \cdot 5^2 = 5^3$
   $R_1 = R_1^2 \Rightarrow R_1 = \left(5^2\right)^2 = 5^4$
4. $k_i = 0$: $R_1 = R_1 \cdot R_0 \Rightarrow R_1 = 5^4 \cdot 5^3 = 5^7$
   $R_0 = R_0^2 \Rightarrow R_0 = \left(5^3\right)^2 = 5^6$
5. $k_i = 1$: $R_0 = R_0 \cdot R_1 \Rightarrow R_0 = 5^6 \cdot 5^7 = 5^{13}$
   $R_1 = R_1^2 \Rightarrow R_1 = \left(5^7\right)^2 = 5^{14}$
6. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{13}$.

Приклад роботи алгоритма А.9 для $a=5$, $k=441_{10}$ та $w=3$.

1. На вхід подається скаляр k у системі числення 2^w: $k=441_{10}=110111001_2=671_{2^3}$.
2. Будується таблиця передобчислень для $w=3$:

№ елемента	Значення елемента
1	5^1
2	5^2
3	5^3
4	5^4
5	5^5
6	5^6
7	5^7

3. Ініціалізація: $b=1$.
4. Розглядаючи розряди скаляра $k=671_{2^3}$ справа наліво отримуємо:

$$b=b\cdot Table[1]=1\cdot 5^1=5$$
$$Table[1]=\left(Table[1]\right)^{2^w}=5^{2^3}=5^8$$
$$k_0=1 \quad\Rightarrow\quad Table[2]=\left(Table[2]\right)^{2^w}=\left(5^2\right)^{2^3}=5^{16}$$
$$\dots$$
$$Table[7]=\left(Table[7]\right)^{2^w}=\left(5^7\right)^{2^3}=5^{56}$$

$$b=b\cdot Table[7]=5\cdot 5^{56}=5^{57}$$
$$Table[1]=\left(Table[1]\right)^{2^w}=\left(5^8\right)^{2^3}=5^{64}$$
$$k_1=7 \quad\Rightarrow\quad Table[2]=\left(Table[2]\right)^{2^w}=\left(5^{16}\right)^{2^3}=5^{128}$$
$$\dots$$
$$Table[7]=\left(Table[7]\right)^{2^w}=\left(5^{56}\right)^{2^3}=5^{448}$$

$$b = b \cdot Table[6] = 5^{57} \cdot 5^{384} = 5^{441}$$

$$Table[1] = (Table[1])^{2^w} = \left(5^{64}\right)^{2^3} = 5^{512}$$

$$k_2 = 6 \quad \Rightarrow \quad Table[2] = (Table[2])^{2^w} = \left(5^{128}\right)^{2^3} = 5^{1024}$$

$$\dots$$

$$Table[7] = (Table[7])^{2^w} = \left(5^{448}\right)^{2^3} = 5^{3584}$$

5. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{441}$.

Приклад роботи алгоритма А.10 для $a = 5$, $k = 441_{10}$ та $w = 3$.

1. На вхід подається скаляр k у системі числення 2^w: $k = 441_{10} = 110111001_2 = 671_{2^3}$.
2. Таблиця передобчислень для $w = 3$ буде мати вигляд аналогічний до таблиці в *RL*-алгоритмі.
3. Ініціалізація: $b = 1$.
4. Розглядаючи розряди скаляра $k = 671_{2^3}$ зліва направо отримуємо:

$$k_2 = 6 \quad \Rightarrow \quad \begin{aligned} b &= b^{2^3} = 1^{2^3} = 1 \\ b &= b \cdot Table[6] = 1 \cdot 5^6 = 5^6 \end{aligned}$$

$$k_1 = 7 \quad \Rightarrow \quad \begin{aligned} b &= b^{2^w} = \left(5^6\right)^{2^3} = 5^{48} \\ b &= b \cdot Table[7] = 5^{48} \cdot 5^7 = 5^{55} \end{aligned}$$

$$k_0 = 1 \quad \Rightarrow \quad \begin{aligned} b &= b^{2^w} = \left(5^{55}\right)^{2^3} = 5^{440} \\ b &= b \cdot Table[1] = 5^{440} \cdot 5 = 5^{441} \end{aligned}$$

5. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{441}$.

RL- та *LR*- розкладання для $k = 441_{10}$ та $w = 3$ виконується наступним чином:

1. Перетворимо скаляр k в двійковий вигляд: $k = 441_{10} = 110111001_2$.
2. Розділяємо розряди скаляра на групи по три біти: $\underbrace{110}_{6}\underbrace{111}_{7}\underbrace{001}_{1}$.
3. *RL*-розкладання буде мати вигляд:

$$a^{441} = a^{1\cdot\left(2^3\right)^0+7\cdot\left(2^3\right)^1+6\cdot\left(2^3\right)^2} = a\cdot\left(a^7\right)^{2^3}\cdot\left(a^6\right)^{\left(2^3\right)^2}.$$

4. *LR*-розкладання – $a^{441} = a^{2^3\cdot(2^3\cdot(6+0)+7)+1} = \left(\left(1\cdot a^6\right)^{2^3}\cdot a^7\right)^{2^3}\cdot a$.

Приклад роботи алгоритма А.11 для $a = 5$, $k = 1577_{10}$ та $w = 3$.

1. Показник степеня *k* у двійковому вигляді: $k = 11000101001_{(2)}$.
2. Ініціалізація: $b = 1,\ i = 11_{10}$.
3. Побудова таблиці передобчислень:

№ елемента	Значення елемента
0	5^4
1	5^5
2	5^6
3	5^7

4. Поки $i > 0$ виконуються наступні кроки алгоритму:

$$4.1.\ k_{11} = 1 \Rightarrow \begin{cases} t = 110_2 = 6_{10} \\ b = b^{2^3} = 1^{2^3} = 1 \\ b = b\cdot Table[t - 2^{w-1}] = b\cdot Table[6 - 2^2] = 1\cdot 5^6 = 5^6 \\ i = 11 - 3 = 8 \end{cases}$$

4.2. $k_8 = 0 \Rightarrow \begin{cases} b = b^2 = \left(5^6\right)^2 = 5^{12} \\ i = 8 - 1 = 7 \end{cases}$

4.3. $k_7 = 0 \Rightarrow \begin{cases} b = b^2 = \left(5^{12}\right)^2 = 5^{24} \\ i = 7 - 1 = 6 \end{cases}$

4.4. $k_6 = 1 \Rightarrow \begin{cases} t = 101_2 = 5_{10} \\ b = b^{2^3} = \left(5^{24}\right)^{2^3} = 5^{192} \\ b = b \cdot Table\,[t - 2^{w-1}] = b \cdot Table\,[5 - 2^2] = 5^{192} \cdot 5^5 = 5^{197} \\ i = 6 - 3 = 3 \end{cases}$

4.5. $k_3 = 0 \Rightarrow \begin{cases} b = b^2 = \left(5^{197}\right)^2 = 5^{394} \\ i = 3 - 1 = 2 \end{cases}$

4.6. $k_2 = 0 \Rightarrow \begin{cases} b = b^2 = \left(5^{394}\right)^2 = 5^{788} \\ i = 2 - 1 = 1 \end{cases}$

4.7. $k_1 = 1 \Rightarrow \begin{cases} \text{Викликати бінарний } LR\text{-алгоритм} \\ b = b^2 = \left(5^{788}\right)^2 = 5^{1576} \\ b = 5^{1576} \cdot 5 = 5^{1577} \end{cases}$

5. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{1577}$.

Приклад роботи алгоритм А.12 перетворення цілого *k* у *NAF* подання для $k = 150$.

1. Ініціалізація : $i = 0$
2. Поки $k \geq 1$ виконуємо наступні дії:

2.1. *False*
2.2. $k_0 = 0$
2.3. $k = \frac{k}{2} = \frac{150}{2} = 75$
2.4. $i = i + 1 = 1$

$\Rightarrow$

2.1. *True*
2.1.1. $k_1 = 2 - (75 \bmod 4) = -1$
2.1.2. $k = 75 - (-1) = 76$
2.2. *False*
2.3. $k = \frac{k}{2} = \frac{76}{2} = 38$
2.4. $i = i + 1 = 2$

$\Rightarrow$

2.1. *False*
2.2. $k_2 = 0$
2.3. $k = \frac{k}{2} = \frac{38}{2} = 19$
2.4. $i = i + 1 = 3$

$\Rightarrow$

2.1. *True*
2.1.1. $k_3 = 2 - (19 \bmod 4) = -1$
2.1.2. $k = 19 - (-1) = 20$
2.2. *False*
2.3. $k = \frac{k}{2} = \frac{20}{2} = 10$
2.4. $i = i + 1 = 4$

$\Rightarrow$

2.1. *False*
2.2. $k_4 = 0$
2.3. $k = \frac{k}{2} = \frac{10}{2} = 5$
2.4. $i = i + 1 = 5$

$\Rightarrow$

2.1. *True*
2.1.1. $k_5 = 2 - (5 \bmod 4) = 1$
2.1.2. $k = 5 - 1 = 4$
2.2. *False*
2.3. $k = \frac{k}{2} = \frac{4}{2} = 2$
2.4. $i = i + 1 = 6$

$\Rightarrow$

2.1. *False*
2.2. $k_6 = 0$
2.3. $k = \frac{k}{2} = \frac{2}{2} = 1$
2.4. $i = i + 1 = 7$

$\Rightarrow$

2.1. *True*
2.1.1. $k_7 = 2 - (1 \bmod 4) = 1$
2.1.2. $k = 1 - 1 = 0$
2.2. *False*
2.3. $k = \frac{k}{2} = \frac{1}{2}$
2.4. $i = i + 1 = 8.$

3. Отже, отримане ціле k у формі *NAF* подається таким чином: $k = 150_{10} = 1010\bar{1}0\bar{1}0_{NAF}$

Приклад роботи алгоритма А.13 для $a = 5$ та $k = 150$.

1. За алгоритмом А.12 отримали *NAF* подання для k: $k = 150_{10} = 1010\bar{1}0\bar{1}0_{NAF}$
2. Ініціалізація: $b = 1$
3. Розглянемо розряди *NAF* подання цілого k справа наліво та отримуємо:

$$k_0 = 0 \Rightarrow a = a^2 = 5^2$$

$$k_1 = -1 \Rightarrow \begin{cases} b = \frac{b}{a} = \frac{1}{5^2} = 5^{-2} \\ a = a^2 = \left(5^2\right)^2 = 5^4 \end{cases}$$

$$k_2 = 0 \Rightarrow a = a^2 = \left(5^4\right)^2 = 5^8$$

$$k_3 = -1 \Rightarrow \begin{cases} b = \frac{b}{a} = \frac{5^{-2}}{5^8} = 5^{-10} \\ a = a^2 = \left(5^8\right)^2 = 5^{16} \end{cases}$$

$$k_4 = 0 \Rightarrow a = a^2 = \left(5^{16}\right)^2 = 5^{32}$$

$$k_5 = 1 \Rightarrow \begin{cases} b = b \cdot a = 5^{-10} \cdot 5^{32} = 5^{22} \\ a = a^2 = \left(5^{32}\right)^2 = 5^{64} \end{cases}$$

$$k_6 = 0 \quad \Rightarrow \quad a = a^2 = \left(5^{64}\right)^2 = 5^{128}$$

$$k_7 = 1 \quad \Rightarrow \begin{bmatrix} b = b \cdot a = 5^{22} \cdot 5^{128} = 5^{150} \\ a = a^2 = \left(5^{128}\right)^2 = 5^{256} \end{bmatrix}$$

4. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{150}$.

Приклад роботи алгоритма А.14 для $a = 5$ та $k = 150$.

1. За алгоритмом А.12 отримали *NAF* подання для *k*: $k = 150_{10} = 10101\bar{0}1\bar{0}_{NAF}$
2. Ініціалізація: $b = 0$
3. Розглянувши розряди *NAF* подання цілого *k* зліва направо та отримуємо:

$$k_7 = 1 \quad \Rightarrow \begin{bmatrix} b = b^2 = 1^2 = 1 \\ b = b \cdot a = 1 \cdot 5 = 5 \end{bmatrix}$$

$$k_6 = 0 \quad \Rightarrow \; b = b^2 = 5^2$$

$$k_5 = 1 \quad \Rightarrow \begin{bmatrix} b = b^2 = \left(5^2\right)^2 = 5^4 \\ b = b \cdot a = 5^4 \cdot 5 = 5^5 \end{bmatrix}$$

$$k_4 = 0 \quad \Rightarrow \; b = b^2 = \left(5^5\right)^2 = 5^{10}$$

$$k_3 = -1 \Rightarrow \begin{bmatrix} b = b^2 = \left(5^{10}\right)^2 = 5^{20} \\ b = \dfrac{b}{a} = \dfrac{5^{20}}{5} = 5^{19} \end{bmatrix}$$

$$k_2 = 0 \;\Rightarrow \; b = b^2 = \left(5^{19}\right)^2 = 5^{38}$$

$$k_1 = -1 \Rightarrow \begin{bmatrix} b = b^2 = \left(5^{38}\right)^2 = 5^{76} \\ b = \dfrac{b}{a} = \dfrac{5^{76}}{5} = 5^{75} \end{bmatrix}$$

$$k_0 = 0 \quad \Rightarrow b = b^2 = \left(5^{75}\right)^2 = 5^{150}$$

4. Результат виконання операції піднесення до степеня знаходиться у змінній $b = 5^{150}$.

Приклад роботи алгоритму перетворення додатного цілого числа k у *wNAF*-подання для $k = 150_{10}$ та $w = 3$.

1. Ініціалізація: $i = 0$
2. Поки $k \geq 1$ виконати наступні кроки:

$$\begin{array}{lcll} t_0 = 0 & & t_1 = 75 \bmod 8 = 3 & t_2 = 0 \\ k = 75 & \Rightarrow & k = 75 - 3 = 72 & \Rightarrow \quad k = \dfrac{36}{2} = 18 \\ i = 1 & & k = \dfrac{72}{2} = 36 & i = 3 \\ & & i = 2 & \end{array}$$

$$\begin{array}{lcll} t_3 = 0 & & t_4 = 9 \bmod 8 = 1 & t_5 = 0 \\ k = \dfrac{18}{2} = 9 & \Rightarrow & k = 9 - 1 = 8 & \Rightarrow \quad k = \dfrac{4}{2} = 2 \\ i = 4 & & k = \dfrac{8}{2} = 4 & i = 6 \\ & & i = 5 & \end{array}$$

$$\begin{array}{lcl} t_6 = 0 & & t_7 = 1 \bmod 8 = 1 \\ k = 1 & \Rightarrow & k = 1 - 1 = 0 \\ i = 7 & & i = 8 \end{array}$$

3. Отримано *wNAF*-подання для числа $k = 10010030_{wNAF}$.

Приклад роботи алгоритма А.16 з поданням показника степеня у вигляді *wNAF* для $a=5$, $k=150_{10}$ та $w=3$.

1. Використали алгоритм А.15 для перетворення показника степеня у вигляд *wNAF*: $k=10010030_{wNAF}$
2. Побудова таблиці передобчислень:

№ елементу	Значення елемента
1	5^1
2	5^3

3. Ініціалізація: $b=1$
4. Проходячи скаляр *k* зліва направо виконуються наступні кроки:

$t_7=1$	$\Rightarrow$	$b=b^2=1^2=1$ $b=b\cdot a=1\cdot 5=5$
$t_6=0$	$\Rightarrow$	$b=b^2=5^2$
$t_5=0$	$\Rightarrow$	$b=b^2=\left(5^2\right)^2=5^4$
$t_4=1$	$\Rightarrow$	$b=b^2=\left(5^4\right)^2=5^8$ $b=b\cdot Table[1]=5^8\cdot 5=5^9$
$t_3=0$	$\Rightarrow$	$b=b^2=\left(5^9\right)^2=5^{18}$
$t_2=0$	$\Rightarrow$	$b=b^2=\left(5^{18}\right)^2=5^{36}$
$t_1=3$	$\Rightarrow$	$b=b^2=\left(5^{36}\right)^2=5^{72}$ $b=b\cdot Table[2]=5^{72}\cdot 5^3=5^{75}$
$t_0=0$	$\Rightarrow$	$b=b^2=\left(5^{75}\right)^2=5^{150}$

5. Результат виконання операції піднесення до степеня знаходиться у змінній $b=5^{150}$.

Приклад роботи алгоритма А.17 з вікном змінної довжини та поданням показника степеня у вигляді *NAF* для $a=5$, $k=150$ та $w=3$.

1. За алгоритмом А.12 отримали *NAF*-подання для *k*: $k=150_{10}=10101\bar{0}1\bar{0}0_{NAF}$.
2. Побудова таблиці передобчислень для $w=3$:

№ елемента	Значення елемента
1	5^1
2	5^3
3	5^5

3. Ініціалізація: $b=1,\ i=7$.
4. Розглядаємо розряди *NAF*-подання цілого *k* зліва направо та отримуємо:

$$k_7=1 \quad \Rightarrow \quad \begin{aligned} &t=3, u=101_{NAF}=5_{10}\\ &b=b^{2^t}=b^{2^3}=1^8=1\\ &b=b\cdot Table[t]=1\cdot 5^5=5^5\\ &i=i-t=7-3=4 \end{aligned}$$

$$k_4=0 \quad \Rightarrow \quad \begin{aligned} &t=1, u=0\\ &b=b^{2^t}=b^{2^1}=\left(5^5\right)^2=5^{10}\\ &i=4-1=3 \end{aligned}$$

$$k_3=-1 \quad \Rightarrow \quad \begin{aligned} &t=3, u=\bar{1}0\bar{1}_{NAF}=-5_{10}\\ &b=b^{2^t}=b^{2^3}=\left(5^{10}\right)^8=5^{80}\\ &b=\frac{b}{Table[t]}=\frac{5^{80}}{5^5}=5^{75}\\ &i=3-3=0 \end{aligned}$$

$$k_0=0 \quad \Rightarrow \quad \begin{aligned} &t=1, u=0\\ &b=b^{2^t}=b^{2^1}=\left(5^{75}\right)^2=5^{150} \end{aligned}$$

5. Результат виконання операції піднесення до степеня знаходиться у змінній $b=5^{150}$.

Приклад роботи алгоритму А.19 для $a=5$ та $k=841232$.

1. Показник степеня *k* представлений у вигляді:

$$k=841232=2^7 3^8+2^1 3^6-2^0 3^3-2^0 3^2+2^0 3^1-2^0 3^0$$

2. Ініціалізація: $b=a^{s_1}=5^1=5$

3. Поки $i\le 5$ виконуються наступні кроки:

$$\begin{aligned}
& & u &= x_1-x_2=7-1=6\\
& & v &= y_1-y_2=8-6=2\\
i=1 & \Rightarrow & b &= b^{3^v}=b^{3^2}=5^9\\
& & b &= b^{2^u}=b^{2^6}=\left(5^9\right)^{2^6}=\left(5^9\right)^{64}=5^{576}\\
& & b &= b\cdot a^{s_2}=5^{576}\cdot 5=5^{577}
\end{aligned}$$

$$\begin{aligned}
& & u &= x_2-x_3=1-0=1\\
& & v &= y_2-y_3=6-3=3\\
i=2 & \Rightarrow & b &= b^{3^v}=b^{3^3}=\left(5^{577}\right)^{3^3}=\left(5^{577}\right)^{27}=5^{15579}\\
& & b &= b^{2^u}=b^{2^1}=\left(5^{15579}\right)^2=5^{31158}\\
& & b &= b\cdot a^{s_3}=5^{31158}\cdot 5^{-1}=5^{31157}
\end{aligned}$$

$$\begin{aligned}
& & u &= x_3-x_4=0\\
& & v &= y_3-y_4=3-2=1\\
i=3 & \Rightarrow & b &= b^{3^v}=b^{3^1}=\left(5^{31157}\right)^3=5^{93471}\\
& & b &= b^{2^u}=b^{2^0}=\left(5^{93471}\right)^1=5^{93471}\\
& & b &= b\cdot a^{s_4}=5^{93471}\cdot 5^{-1}=5^{93470}
\end{aligned}$$

$$\begin{aligned}
& & u &= x_4-x_5=0\\
& & v &= y_4-y_5=2-1=1\\
i=4 & \Rightarrow & b &= b^{3^v}=b^{3^1}=\left(5^{93470}\right)^3=5^{280410}\\
& & b &= b^{2^u}=b^{2^0}=\left(5^{280410}\right)^1=5^{280410}\\
& & b &= b\cdot a^{s_5}=5^{280410}\cdot 5^1=5^{280411}
\end{aligned}$$

$$u = x_5 - x_6 = 0$$

$$v = y_5 - y_6 = 1 - 0 = 1$$

$$i = 5 \quad \Rightarrow \quad b = b^{3^v} = b^{3^1} = \left(5^{280411}\right)^3 = 5^{841233}$$

$$b = b^{2^u} = b^{2^0} = \left(5^{841233}\right)^1 = 5^{841233}$$

$$b = b \cdot a^{s_6} = 5^{841233} \cdot 5^{-1} = 5^{841232}$$

4. Результат піднесення до степеня знаходиться у змінній $b = 5^{841232}$.

ДОДАТОК В

Схеми алгоритмів роботи керуючих автоматів складових частин процесора Галуа

Схеми алгоритмів роботи керуючого автомата блока виконання операцій за модулем $2^m - 1$

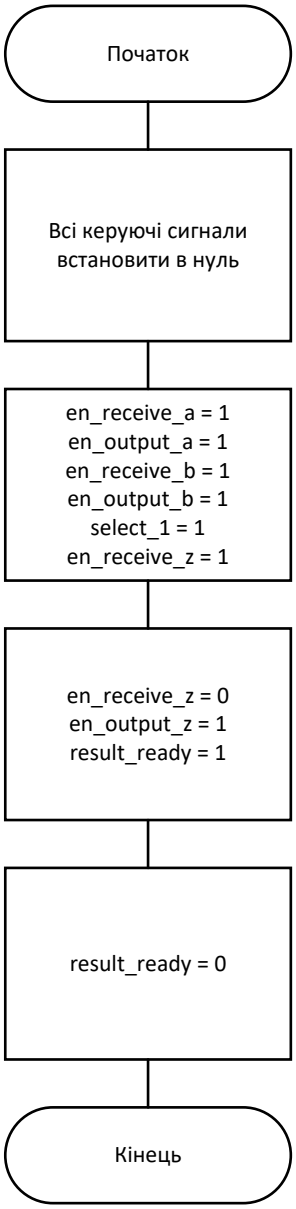


Рис. В.1. Алгоритм виконання мікрооперації додавання за модулем $2^m - 1$

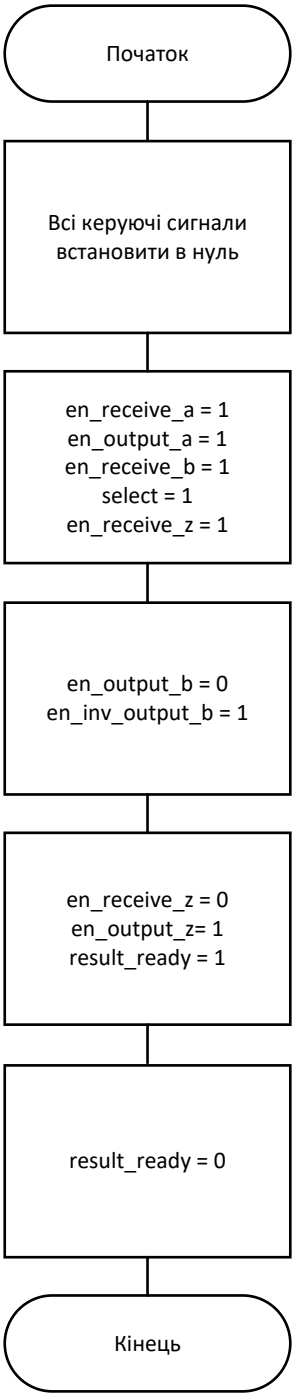


Рис. В.2. Алгоритм виконання мікрооперації віднімання за модулем $2^m - 1$

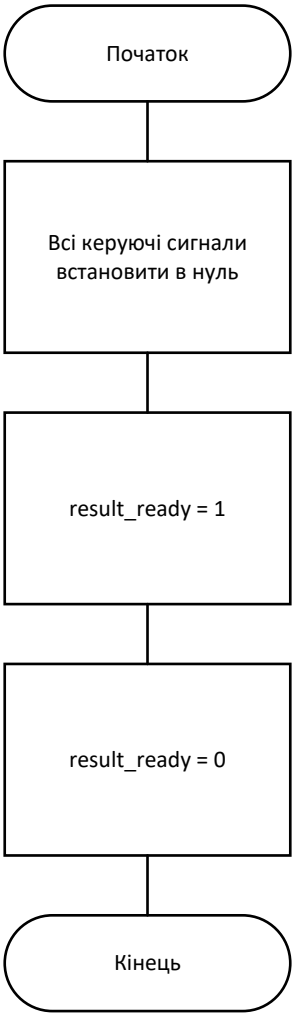


Рис. В.3. Алгоритм виконання мікрооперації інвертування

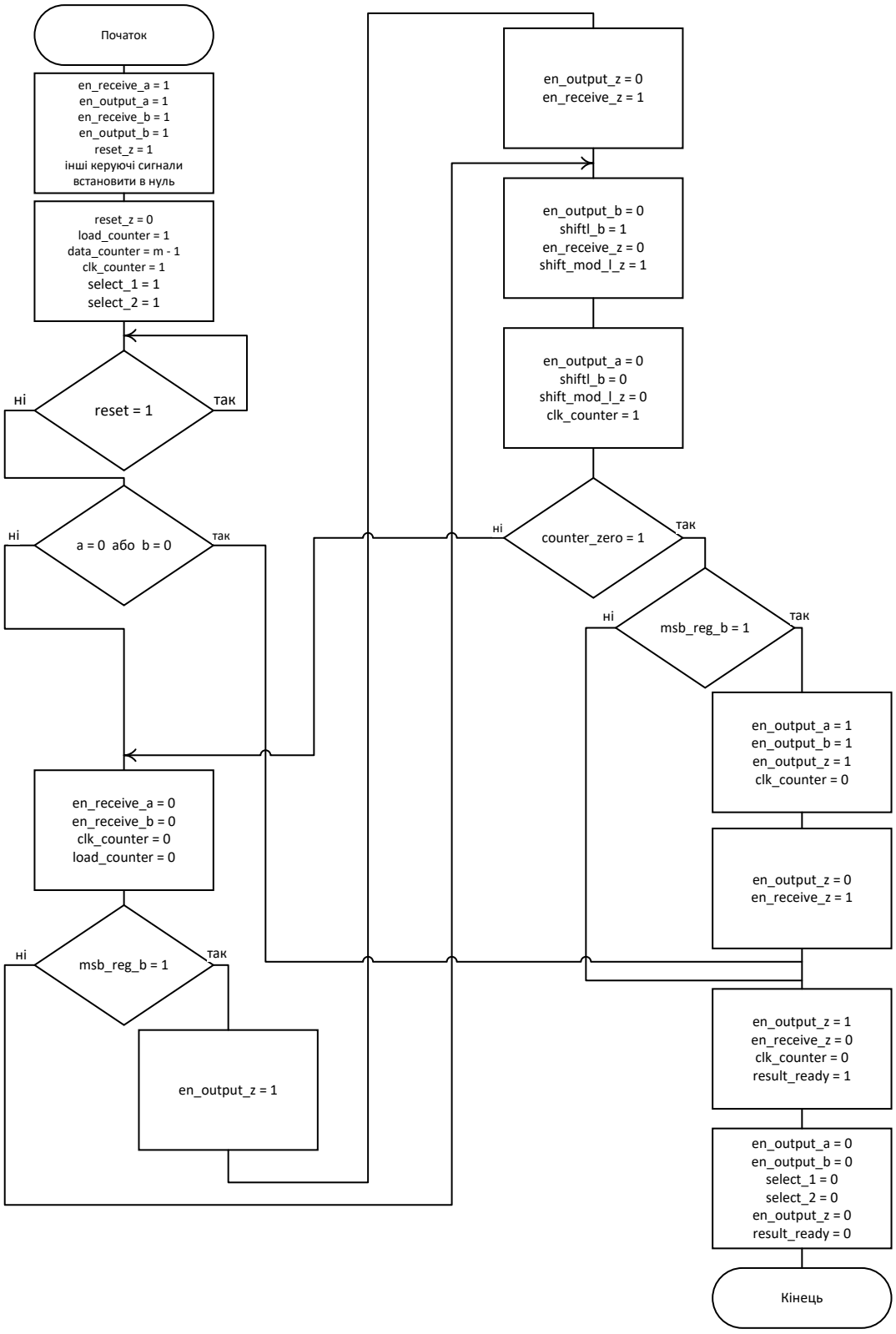


Рис. В.4. Алгоритм виконання мікрооперації множення за модулем $2^m - 1$

Схеми алгоритмів роботи керуючого автомата блока виконання операцій над елементами поля $GF(2^m)$

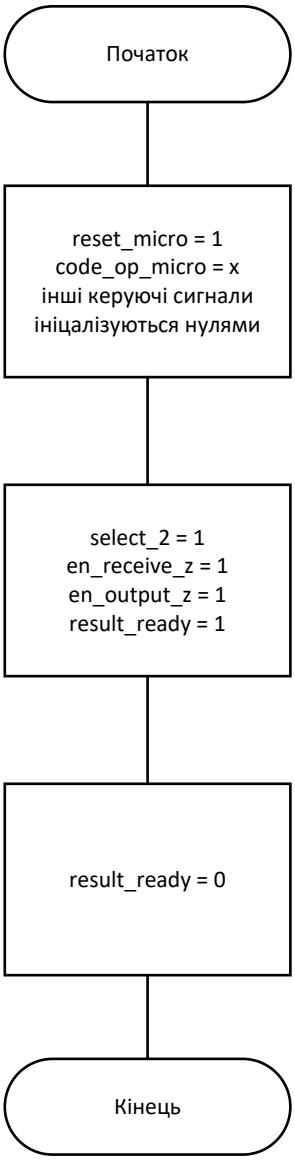


Рис. В.5. Алгоритм виконання операції додавання та віднімання у полі $GF(2^m)$

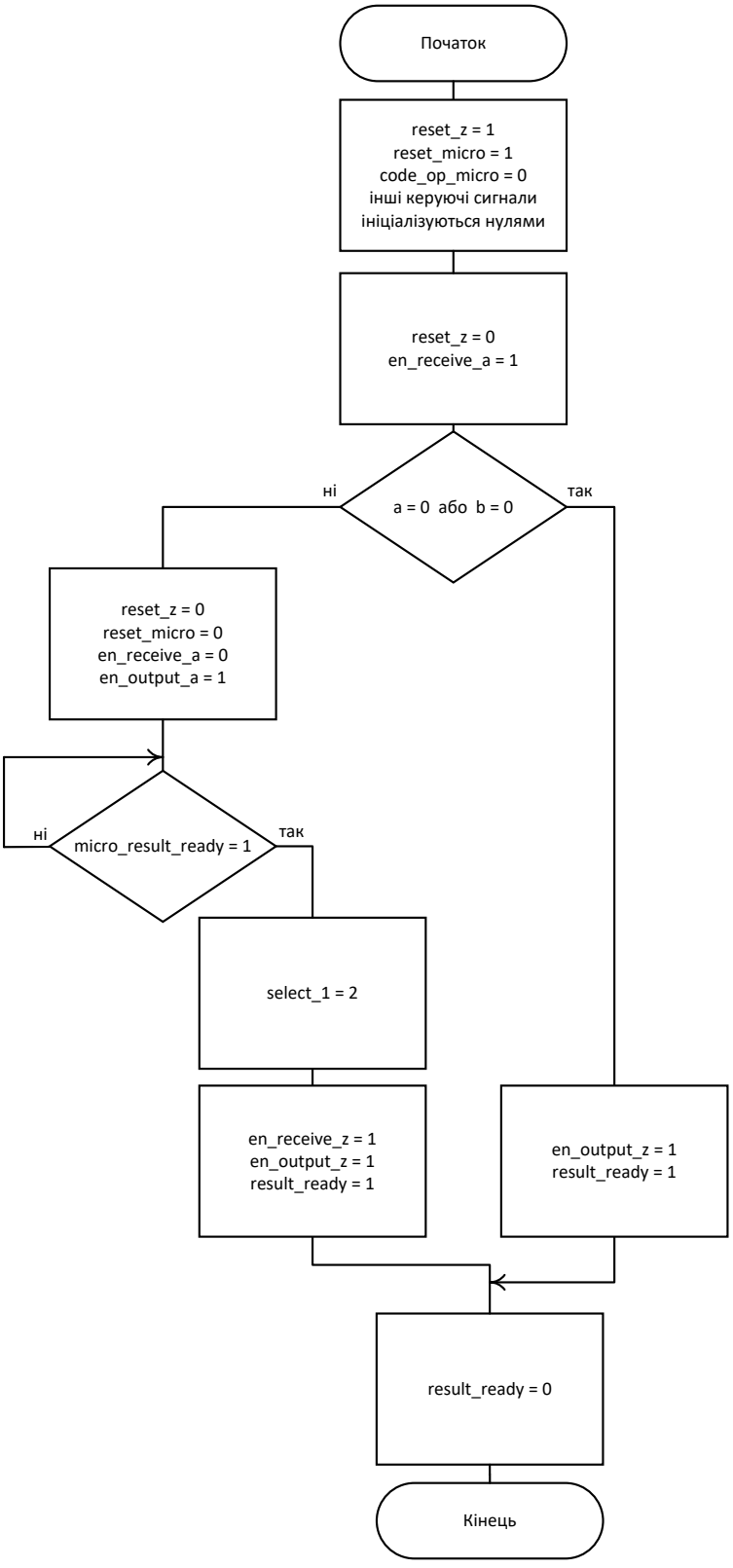


Рис. В.6. Алгоритм виконання операції множення у полі $GF(2^m)$

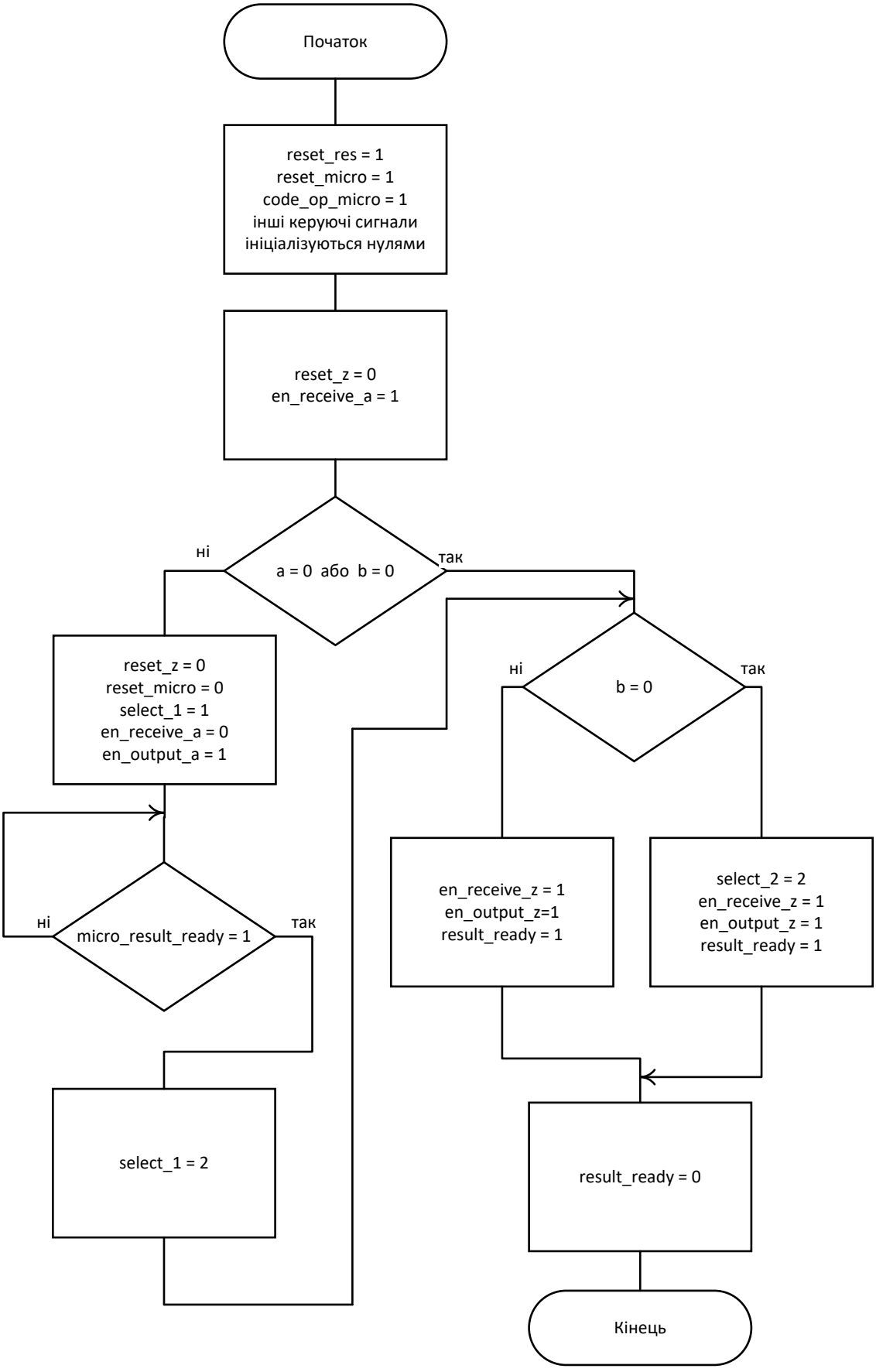


Рис. В.7. Алгоритм виконання операції ділення у полі $GF(2^m)$

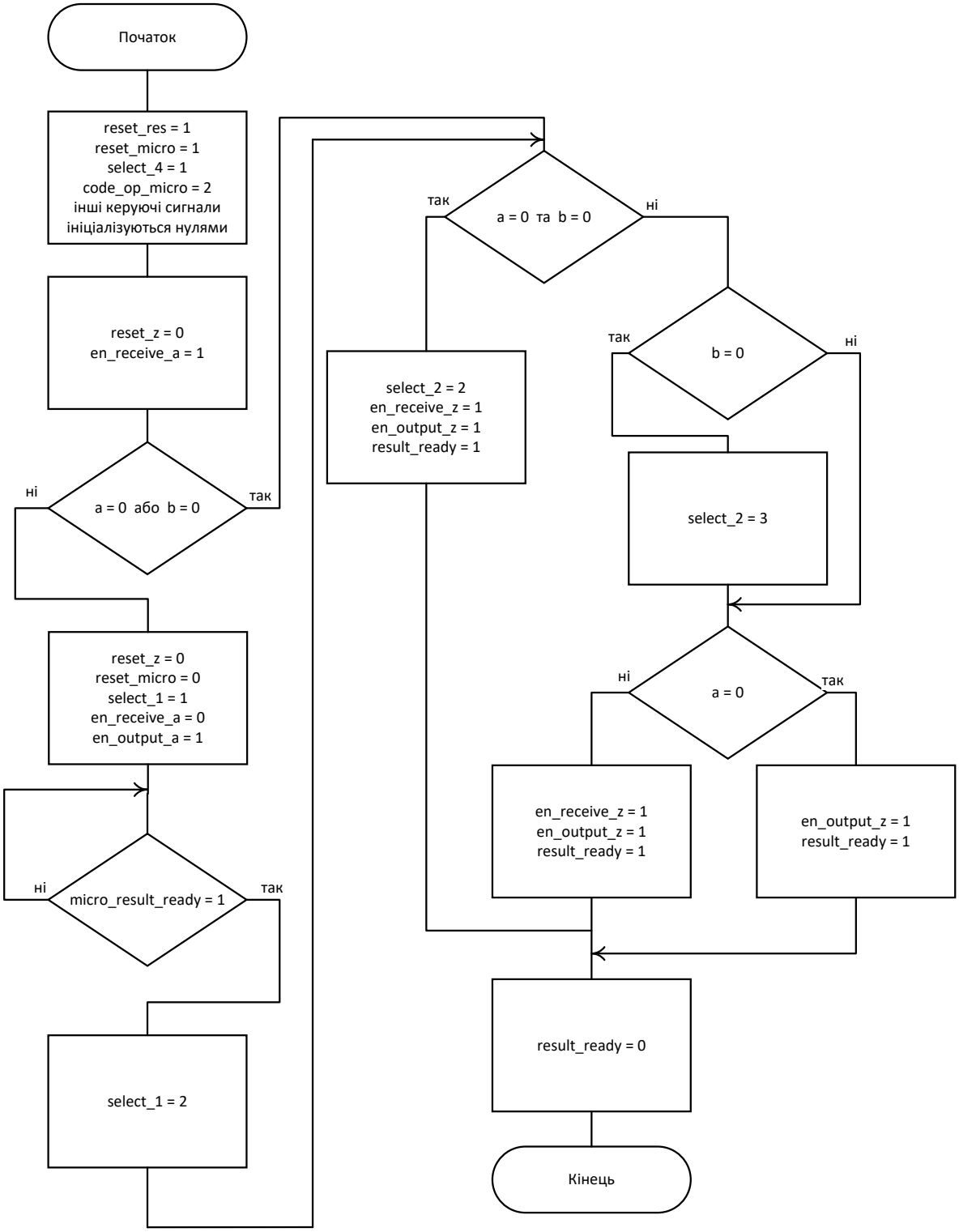


Рис. В.8. Алгоритм виконання операції піднесення до степеня у полі $GF(2^m)$

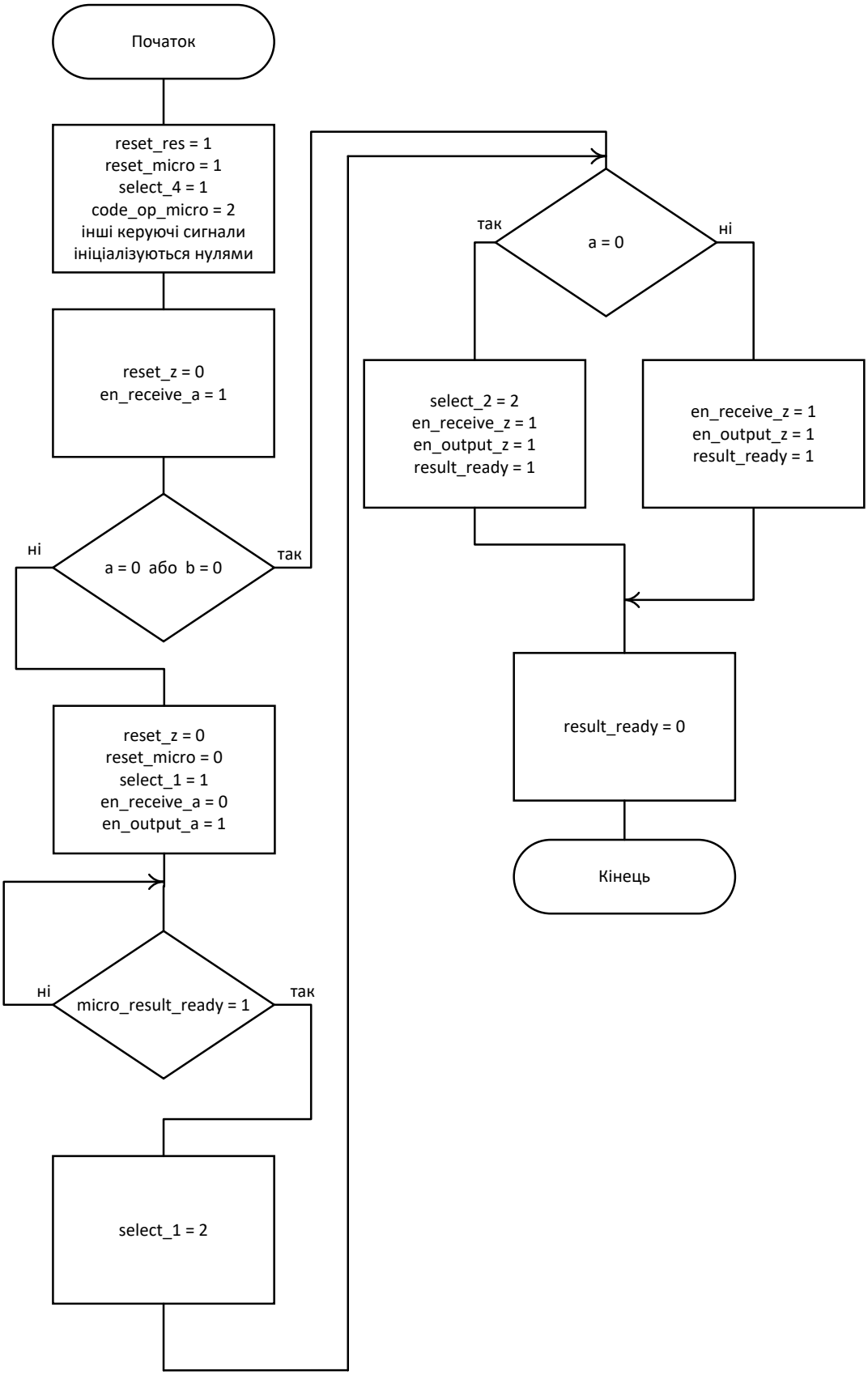


Рис. В.9. Алгоритм виконання операції знаходження мультиплікативно оберненого елемента у полі $GF(2^m)$

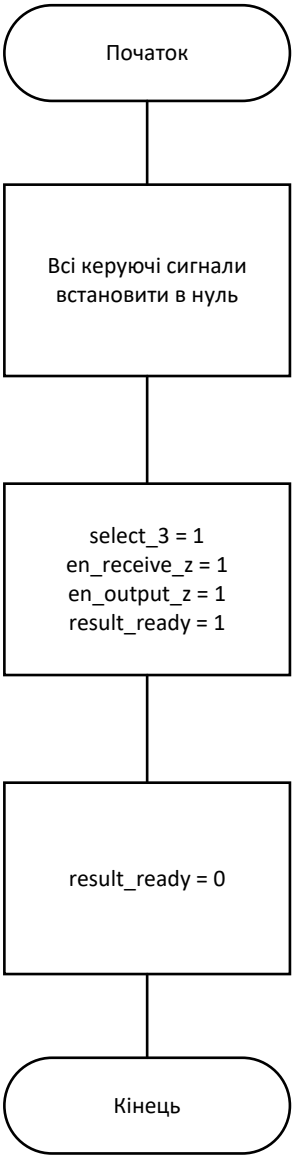


Рис. В.10. Алгоритм виконання операції переведення елементу поля $GF(2^m)$ з числового подання у степеневе

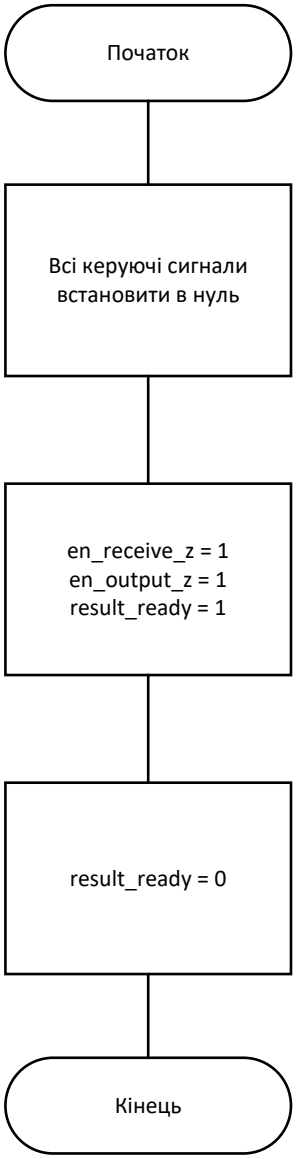


Рис. В.11. Алгоритм виконання операції переведення елементу поля $GF(2^m)$ зі степеневого подання у числове

Схеми алгоритмів роботи керуючого автомата процесора Галуа

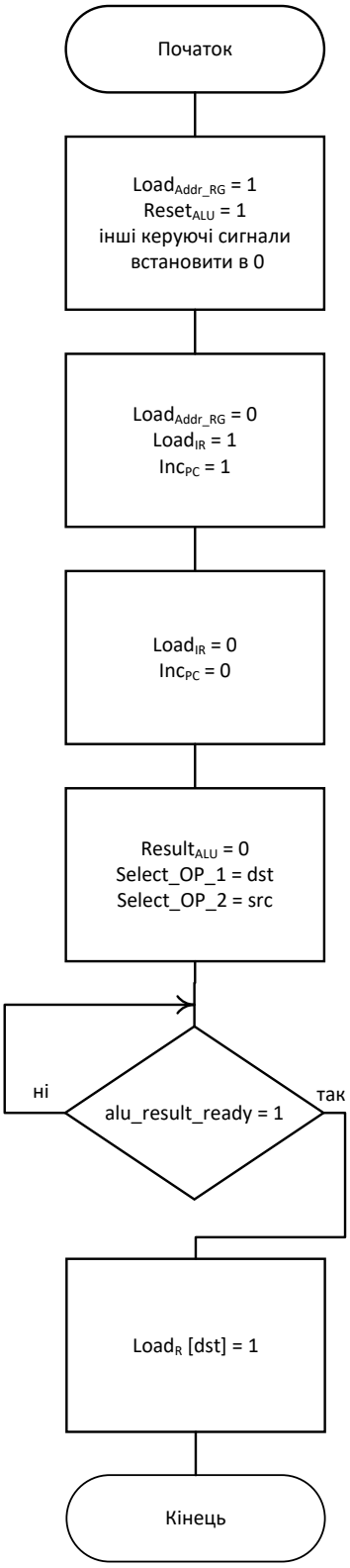


Рис. В.12. Алгоритм виконання команди *ADD*, *SUB*, *MUL*, *DIV*, *POW*, *INVM*, *CDP*, *INVA*, *CPD*, *SUB*

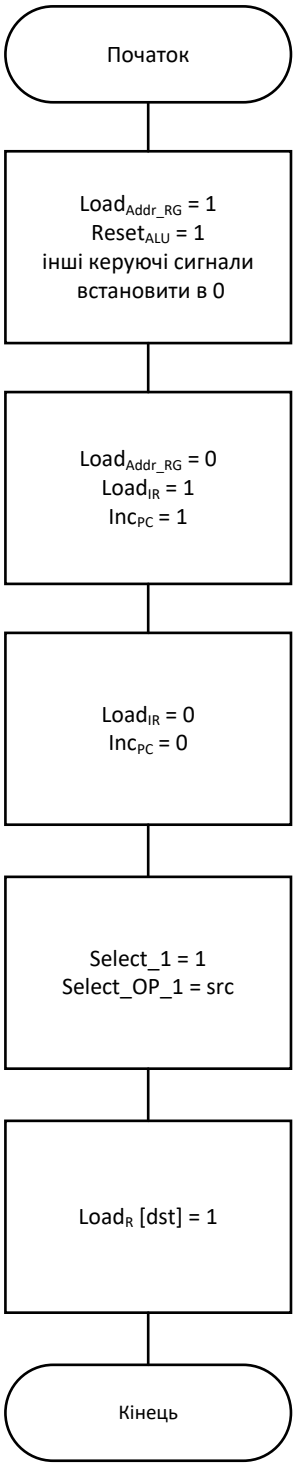


Рис. В.13. Алгоритм виконання однобайтної команди *MOV*

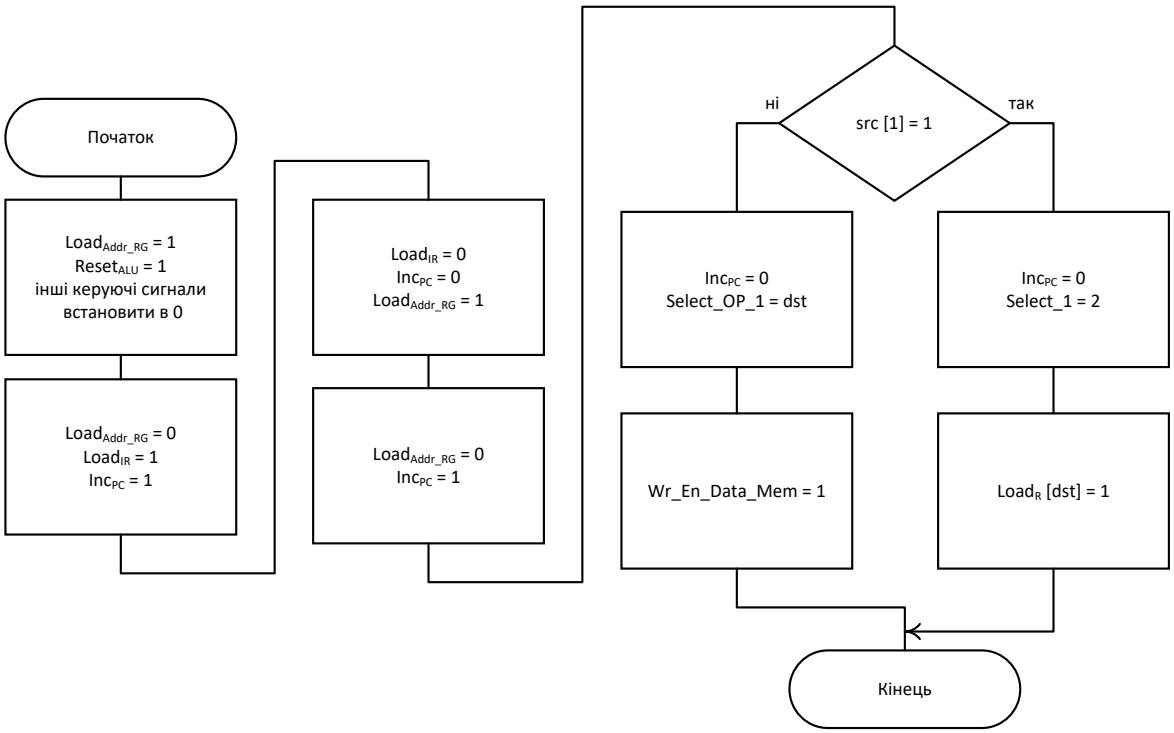


Рис. В.14. Алгоритм виконання двобайтної команди *MOV*

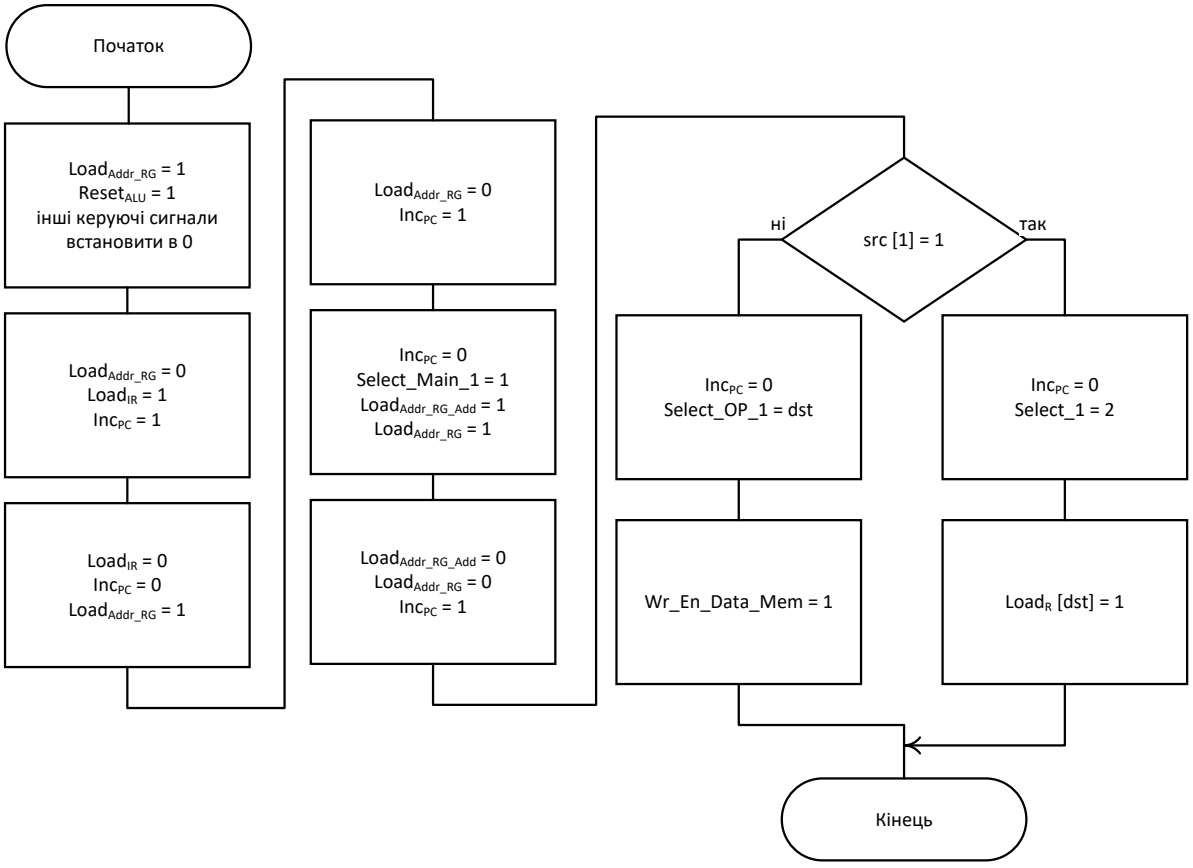


Рис. В.15. Алгоритм виконання трибайтної команди *MOV*

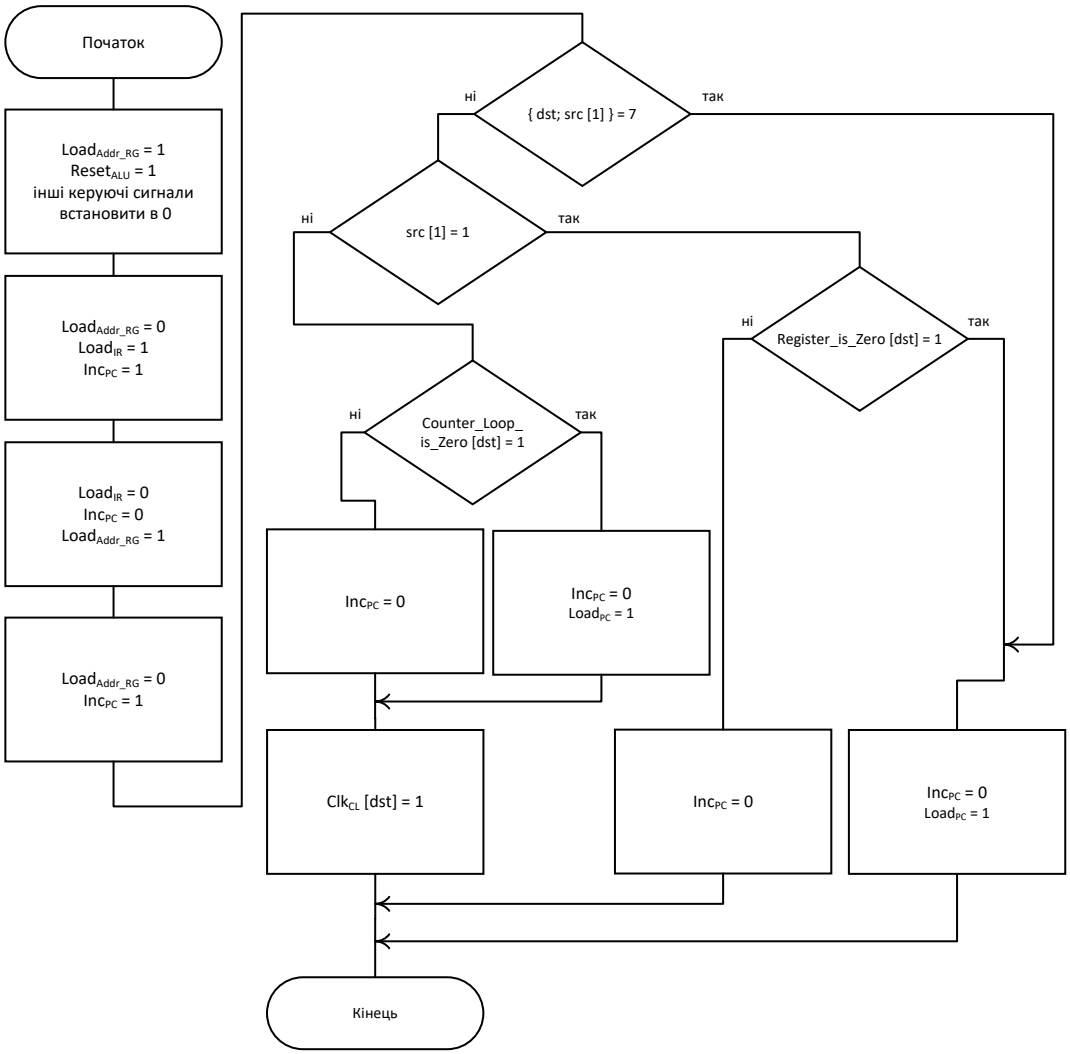


Рис. В.16. Алгоритм виконання команди *JMP*

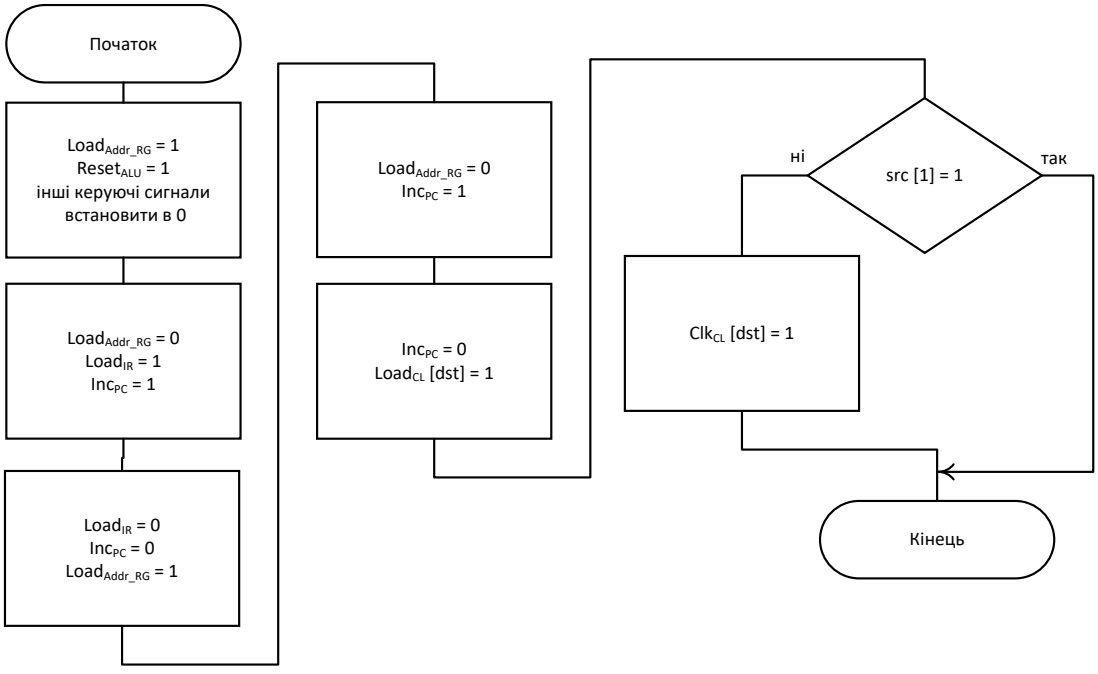


Рис. В.17. Алгоритм виконання команди *LOOP*

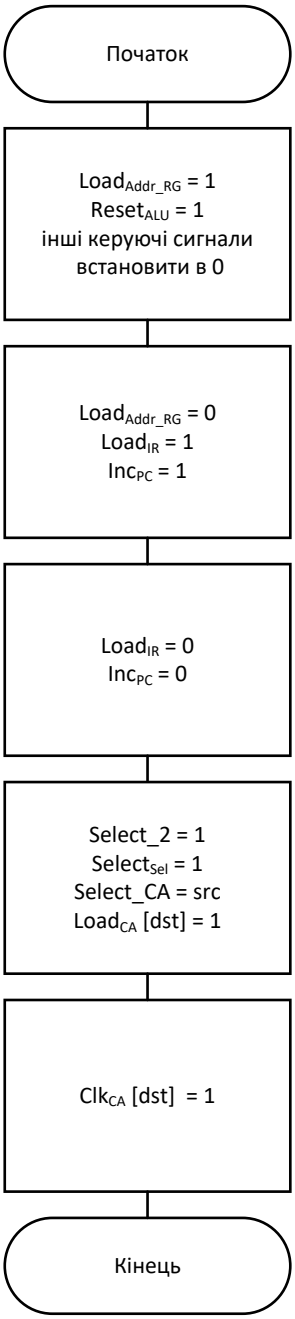


Рис. В.18. Алгоритм виконання однобайтної команди *LOAD*

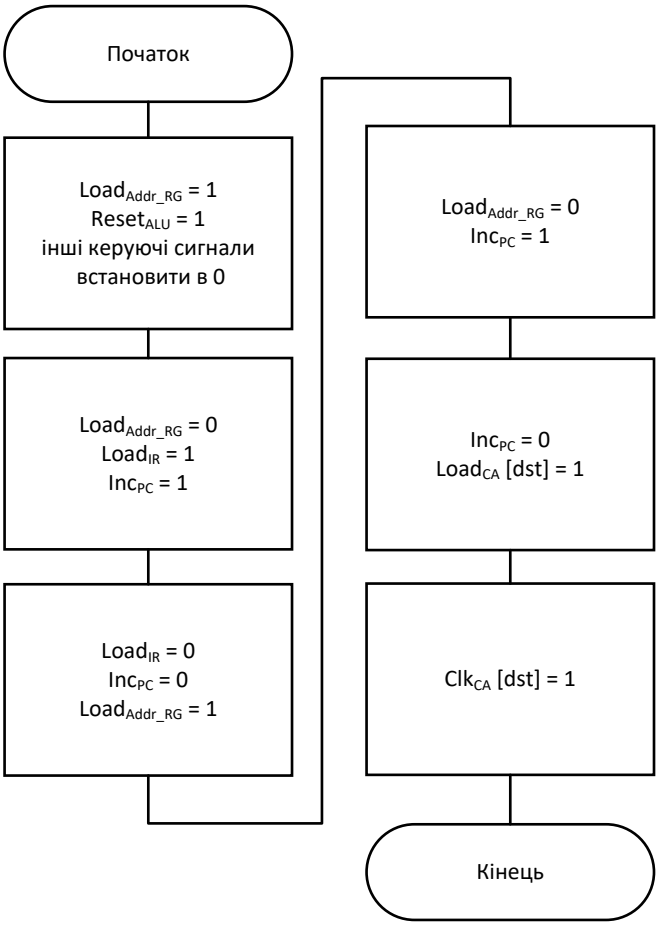


Рис. В.19. Алгоритм виконання двобайтної команди *LOAD*

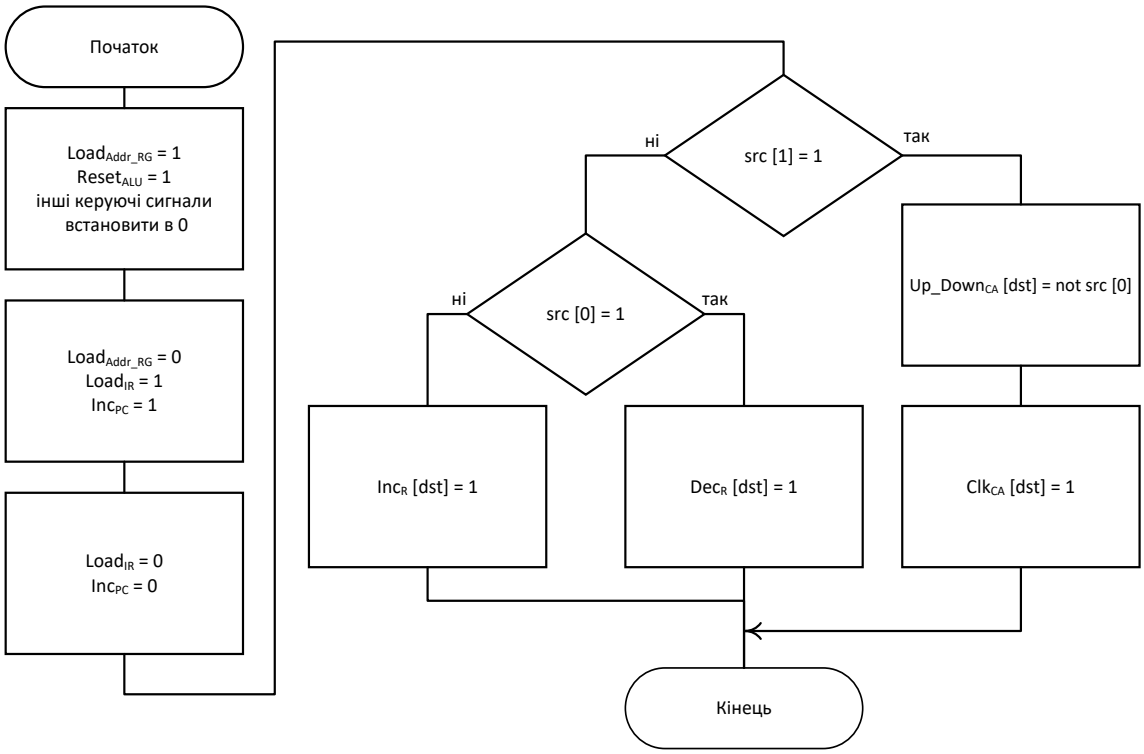


Рис. В.20. Алгоритм виконання команд *INC* та *DEC*

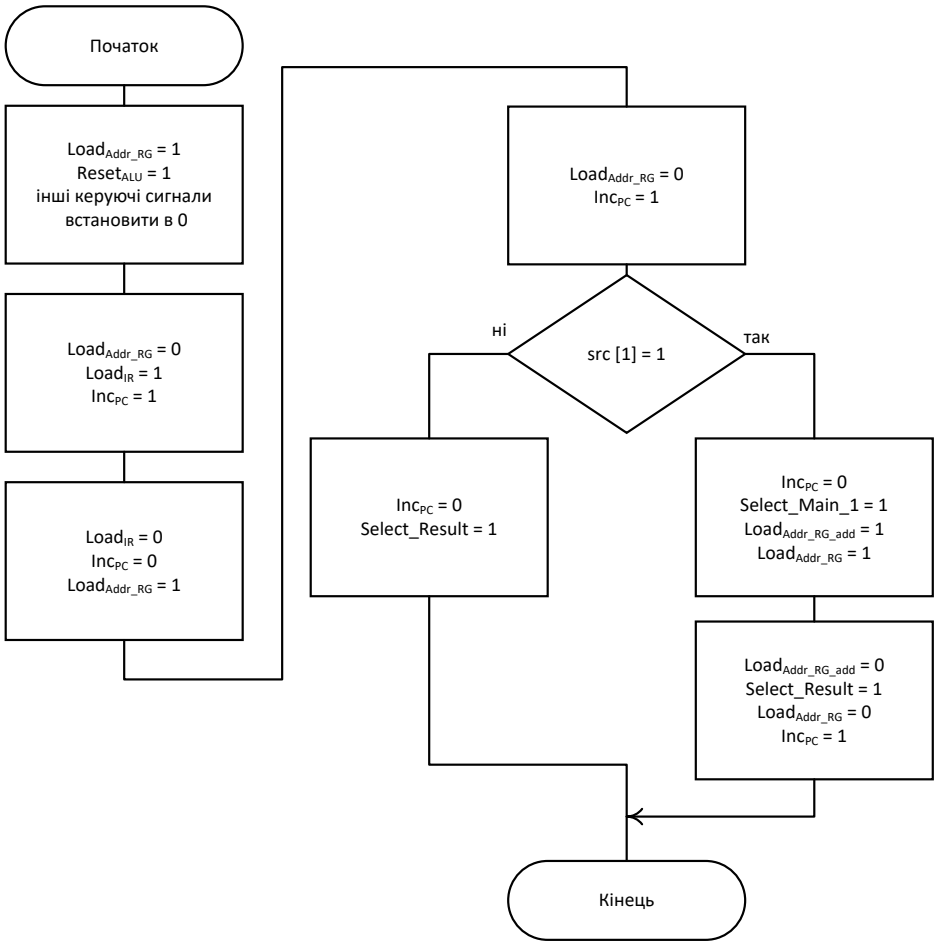


Рис. В.21. Алгоритм виконання команди *OUT*

ДОДАТОК Г

Формати команд процесора Галуа

Команда ADD R1, R2 (однобайтна)

Для поля $GF(p)$ виконується додавання значень (для поля $GF(2^m)$ операція додавання є тотожною операції віднімання), що знаходяться у двох регістрах, які є операндами для даної команди, результат записується у регістр що вказаний першим.

```
ADD R0, R1 // додавання до значення в регістрі R0 значення з регістру R1 та запис результату в регістр R0
```

Біти	Опис
7 – 4	Код операції (0000)
3 – 2	Номер першого регістра (регістра який є першим операндом та в який буде записано результат виконання операції)
1 – 0	Номер другого регістра (регістра який є другим операндом)

Команда MUL R1, R2 (однобайтна)

Виконується множення значень у двох регістрах, результат записується у регістр, що вказаний першим.

```
MUL R0, R1 // множення значення в регістрі R0 на значення з регістру R1 та запис результату в регістр R0
```

Біти	Опис
7 – 4	Код операції (0001)
3 – 2	Номер першого регістра (регістра який є першим операндом та в який буде записано результат виконання операції)
1 – 0	Номер другого регістра (регістра який є другим операндом)

Команда DIV R1, R2 (однобайтна)

Виконується ділення значень, результат записується у регістр, що вказаний першим.

```
DIV R0, R1 // ділення значення в регістрі R0 на значення з регістру R1 та запис результату в регістр R0
```

Біти	Опис
7 – 4	Код операції (0010)
3 – 2	Номер першого регістра (регістра який є першим операндом та в який буде записано результат виконання операції)
1 – 0	Номер другого регістра (регістра який є другим операндом)

Команда POW R1, R2 (однобайтна)

Виконується піднесення до степеня (показник степеня знаходиться у другому операнді) значення першого операнду, результат записується у регістр який вказаний першим.

```
POW R0, R1 // піднесення значення в регістрі R0 до степеня, що дорівнює значенню з регістру R1 та запис результату в регістр R0
```

Біти	Опис
7 – 4	Код операції (0011)
3 – 2	Номер першого регістра (регістра який є першим операндом та в який буде записано результат виконання операції)
1 – 0	Номер другого регістра (регістра який є другим операндом)

Команда INVM R (однобайтна)

Виконується пошук мультиплікативно оберненого елемента до значення, що знаходиться у регістрі, результат записуються у цей самий регістр.

```
INVM R0 // пошук мультиплікативно оберненого елемента до значення з регістру R0
```

Біти	Опис
7 – 4	Код операції (0100)
3 – 2	Номер регістра (регістр який є операндом та в який буде записано результат виконання операції)
1 – 0	Не використовуються

Команда INVA R (однобайтна)

Дана команда використовується тільки для поля $GF(p)$. Виконується пошук адитивно оберненого елемента до значення, що знаходиться у регістрі, результат записується у цей самий регістр.

```
INVA R0 // пошук адитивно оберненого елемента до значення з регістру R0
```

Біти	Опис
7 – 4	Код операції (0101)
3 – 2	Номер регістра (регістр який є операндом та в який буде записано результат виконання операції)
1 – 0	Не використовуються

Команда CDP R (однобайтна)

Дана команда використовується тільки для поля $GF\left(2^m\right)$. Виконується перетворення значення, що знаходиться у регістрі з многочленного подання у степеневе, результат записується у той самий регістр.

```
CDP R0 // приклад використання команди CDP
```

Біти	Опис
7 – 4	Код операції (0101)
3 – 2	Номер регістра (регістр який є операндом та в який буде записано результат виконання операції)
1 – 0	Не використовуються

Команда SUB R1, R2 (однобайтна)

Дана команда використовується тільки для поля $GF\left(p\right)$. Виконується віднімання значень, результат записується у регістр що вказаний першим.

```
SUB R0, R1 // віднімання від значення в регістрі R0 значення з регістру R1 та запис результату в регістр R0
```

Біти	Опис
7 – 4	Код операції (0110)
3 – 2	Номер першого регістра (регістра який є першим операндом та в який буде записано результат виконання операції)
1 – 0	Номер другого регістра (регістра який є другим операндом)

Команда CPD R (однобайтна)

Дана команда використовується тільки для поля $GF\left(2^m\right)$. Виконується перетворення значення, що знаходиться у регістрі зі степеневого подання у многочленне, результат записується у той самий регістр.

```
CPD R0 // приклад використання команди CPD
```

Біти	Опис
7 – 4	Код операції (0110)
3 – 2	Номер регістра (регістр який є операндом та в який буде записано результат виконання операції)
1 – 0	Не використовуються

Команда MOV R1, R2 (однобайтна)

Виконується запис значення, що знаходиться у другому регістрі у перший регістр.

```
MOV R0, R1 // запис значення з регістру R1 в R0
```

Біти	Опис
7 – 4	Код операції (0111)
3 – 2	Номер першого регістра
1 – 0	Номер другого регістра

Команда MOV R, M (двобайтна)

Виконується запис значення з пам’яті даних до регістру.

```
DATA
const num = 27
variable_1 = b’101
CODE
MOV R1, num // в регістр R1 завантажити значення константи num
MOV R0, variable_1 // в регістр R0 завантажити значення змінної variable_1
```

Байт	Біти	Опис
1	7 – 4	Код операції (1000)
	3 – 2	Номер регістра
	1	Вказує на те з пам’яті даних у регістр пишемо чи з регістру у пам’ять даних (якщо 1, то виконуємо запис з пам'яті в регістр; якщо 0, то виконуємо запис з регістру в пам'ять)
	0	Старший біт адреси у пам’яті даних
2	7 – 0	Молодші біти адреси у пам’яті даних

Команда MOV M, R (двобайтна)

Виконується запис значення з регістру до пам’яті даних.

```
DATA
variable_1 = b’101
variable_2 = 0
CODE
MOV variable_2, R0 // у змінну variable_2 завантажити значення з регістру R0
MOV variable_1, R3 // у змінну variable_1 завантажити значення з регістру R3
```

Команда MOV R, A [AC ± disp] (трибайтна)

Використовується для роботи з масивами. Існують дві мнемоніки для такого типу команди. При звертанні до елементу масиву використовується базова індексна адресація.

Виконується запис значення з пам’яті даних (елемент масиву) в регістр.

```
MOV R1, A [AC_1 + 3] // приклад використання команди MOV
```

де *A* – ім’я масиву.

Адреса у пам’яті даних з якої потрібно скопіювати дані у відповідний регістр формується як початкова адреса масиву плюс значення у лічильнику адрес (*AC*0, *AC*1, *AC*2, *AC*3) плюс/мінус зміщення, яке задається явно.

Байт	Біти	Опис
1	7 – 4	Код операції (1001)
	3 – 2	Номер регістра
	1	Вказує на те з пам’яті даних у регістр пишемо чи з регістру у пам’ять даних (якщо 1, то виконуємо запис з пам'яті в регістр; якщо 0, то виконуємо запис з регістру в пам'ять)
	0	Старший біт адреси у пам’яті даних (початкова адреса масиву)
2	7 – 0	Молодші біти адреси у пам’яті даних (початкова адреса масиву)
3	7 – 6	Номер лічильника адрес
	5	Не використовується
	4	Знак зміщення (якщо 0, то плюс; якщо 1, то мінус)
	3 – 0	Значення зміщення

Команда MOV A [AC ± disp], R (трибайтна)

Виконується запис значення з регістра в пам’ять даних.

```
MOV A [AC1 + 5], R2 // приклад використання команди MOV
```

де *A* – ім’я масиву.

```
MOV A [AC0 - 12], R2 // приклад використання команди MOV
```

де *A* – ім’я масиву.

Адреса у пам’яті даних до якої потрібно скопіювати дані з відповідного регістру формується як початкова адреса масиву + значення у лічильнику адрес (*AC*0, *AC*1, *AC*2, *AC*3) плюс/мінус зміщення, яке задається явно.

Команда JMP Label (двобайтна)

Використовується для безумовного переходу за адресою.

```
JMP Label // безумовний перехід до мітки Label
```

де *Label* – мітка в коді програми, при компіляції, компілятор замість мітки у машинний код команди підставляє адресу пам’яті команди до якої потрібно перейти.

Байт	Біти	Опис
1	7 – 4	Код операції (1010)
	3 – 1	Значення 111 в цих бітах вказує на те що це безумовний перехід
	0	Старший біт адреси у пам’яті команд
2	7 – 0	Молодші біти адреси у пам’яті команд

Команда JMP R, Label (двобайтна)

Використовується для безумовного переходу за адресою.

```
JMP R1, Label // умовний перехід, якщо значення у регістрі R1 дорівнює нулю, то виконуємо перехід до мітки Label
```

де *Label* – мітка в коді програми, при компіляції, компілятор замість мітки у машинний код команди підставляє адресу пам’яті команди до якої потрібно перейти.

Байт	Біти	Опис
1	7 – 4	Код операції (1010)
	3 – 2	Номер регістру (можна використовувати тільки 00, 01 та 10, тобто регістр *R*3 не можна використовувати, оскільки комбінація 11 зарезервована для безумовного переходу)
	1	На те що це умовний перехід за нульовим значенням регістру вказує значення 1 у цьому біті
	0	Старший біт адреси у пам’яті команд
2	7 – 0	Молодші біти адреси у пам’яті команд

Окрім цього команда *JMP* неявним чином використовується при роботі з циклами (при формуванні машинного коду компілятором додається ця команда до машинного коду автоматично)

Байт	Біти	Опис
1	7 – 4	Код операції (1010)
	3 – 2	Номер лічильника циклу
	1	На те що це умовний перехід за нульовим значенням лічильника циклу вказує значення 0 у цьому біті
	0	Старший біт адреси у пам’яті команд
2	7 – 0	Молодші біти адреси у пам’яті команд

Команда LOOP N, val (двобайтна)

Повторити цикл за значенням лічильника

```
LOOP 1, val // задаємо номер циклу (номер лічильника, який буде використовуватись) та кількість повторень циклу (значення, яке буде записано у лічильник циклу)
```

де *val* –будь-який арифметичний вираз.

Байт	Біти	Опис
1	7 – 4	Код операції (1011)
	3 – 2	Номер лічильника циклу (перший операнд у команді)
	1	Не використовується
	0	Старший біт адреси у пам'яті даних, де знаходиться кількість повторень циклу
2	7 – 0	Молодші біти адреси у пам'яті даних, де знаходиться кількість повторень циклу

Команда *LOOP* є командою початку циклу. Для позначення кінця циклу використовується службове мнемонічне позначення *END_LOOP*.

Під час компіляції програми замість *END_LOOP N*, де $N \in \{0; 1; 2; 3\}$ компілятор вставляє команду безумовного переходу JMP <адреса переходу>.

Наприклад

```
LOOP 1, val
    // тіло циклу
END_LOOP 1 // кінець циклу (обов'язково необхідно вказати номер циклу), ця команда при перетворенні у машинний код замінюється на команду JMP безумовного переходу за адресою наступної команди машинного коду після команди LOOP.
```

Декремент лічильника циклу виконується неявно кожного разу коли доходимо до команди *END_LOOP*. Оскільки під номер лічильника циклу виділено лише 2 розряди, то можна використовувати не більше 4 лічильників циклу, що обумовлює обмеження на кількість вкладених

циклів, не більше чотирьох у програмі. Звертає на себе увагу той факт, що в цій команді є вільний біт і наче можна було б використати його для збільшення кількості лічильників циклу, але у команді умовного переходу немає такої можливості.

Характерною особливістю для формату команди роботи з циклом є те, що такі дані як кількість повторень циклу не записується безпосередньо у код команди, а записується в пам’ять даних. В такому випадку у код команди записується адреса де ці дані знаходяться у пам’яті даних, оскільки розрядність даних може перевищувати значення розрядності адрес даних ($addr_2$ = 9 біт), яке відводиться під поле адреси в коді команди.

Команда LOAD AC1, AC2 (однобайтна)

Виконується запис значення з другого лічильника адрес у перший (лічильник адрес використовується в якості індексу при роботі з масивом).

```
LOAD AC3, AC2 // завантажити значення адреси з другого лічильника адрес у третій
```

Біти	Опис
7 – 4	Код операції (1100)
3 – 2	Номер лічильника адрес, що є першим операндом (лічильник-отримувач)
1 – 0	Номер лічильника адрес, що є другим операндом (лічильник-джерело)

Команда LOAD AC, val (двобайтна)

Виконується запис значення другого операнда у лічильник адрес (лічильник адрес використовується в якості індексу при роботі з масивом).

```
LOAD AC1, val // завантажити значення val у перший лічильник адрес
```

де *val* –будь-який арифметичний вираз.

Байт	Біти	Опис
1	7 – 4	Код операції (1101)
	3 – 2	Номер лічильника адрес
	1	Не використовується
	0	Старший біт адреси, що записується у лічильник адрес
2	7 – 0	Молодші біти адреси, що записується у лічильник адрес

Команди INC (однобайтна)

Виконується інкремент значення у відповідному регістрі або лічильнику адрес (лічильник адрес використовується в якості індекса при роботі з масивом)

```
INC R1 \\ виконуємо інкремент значення в регістрі R1
INC AC2 \\ виконуємо інкремент значення в другому лічильнику адрес
```

Біти	Опис
7 – 4	Код операції (1110)
3 – 2	Номер лічильника адрес або регістра
1	Вказує з чим працюємо: з регістром чи лічильником адрес (0 – регістр, 1 – лічильник адрес)
0	Вказує інкремент чи декремент маємо виконувати (0 – інкремент, 1 – декремент)

Команди DEC (однобайтна)

Виконується декремент значення у відповідному регістрі або лічильнику адрес (лічильник адрес використовується в якості індекса при роботі з масивом)

```
DEC R3 \\ виконуємо декремент значення в регістрі R0
DEC AC0 \\ виконуємо декремент значення в третьому лічильнику адрес
```

Команда OUT M (двобайтна)

Виконується видача на вихід схеми значення, що знаходиться за вказаною адресою у пам’яті даних

```
OUT val \\ приклад двобайтної команди
```

де *val* – константа або змінна описана в секції DATA.

Байт	Біти	Опис
1	7 – 4	Код операції (1111)
	3 – 2	Не використовуються
	1	Вказує двобайтна чи трибайтна команда (якщо 1, то трибайтна; якщо 0, то двобайтна)
	0	Старший біт адреси (для трибайтної команди початкова адреса масиву)
2	7 – 0	Молодші біти адреси (для трибайтної команди початкова адреса масиву)
3	7 – 6	Номер лічильника адрес
	5	Не використовується
	4	Знак зміщення (якщо 0, то плюс; якщо 1, то мінус)
	3 – 0	Значення зміщення

Команда OUT A [AC ± disp] (трибайтна)

Виконується видача на вихід схеми значення, що знаходиться за вказаною адресою у пам’яті даних

```
OUT A [AC_1 + 3] \\ приклад трибайтної команди
OUT A [AC_2 - 10] \\ приклад трибайтної команди
```

Байт 01001111 не використовується жодною з команд, він є зарезервованим в якості командного *STOP*-слова. Таке слово має міститись в кінці кожної програми. Під час виконання програми, коли процесор отримує *STOP*-слово це є сигналом, що програма виконана повністю і на виході процесору формується активний сигнал *ready*.

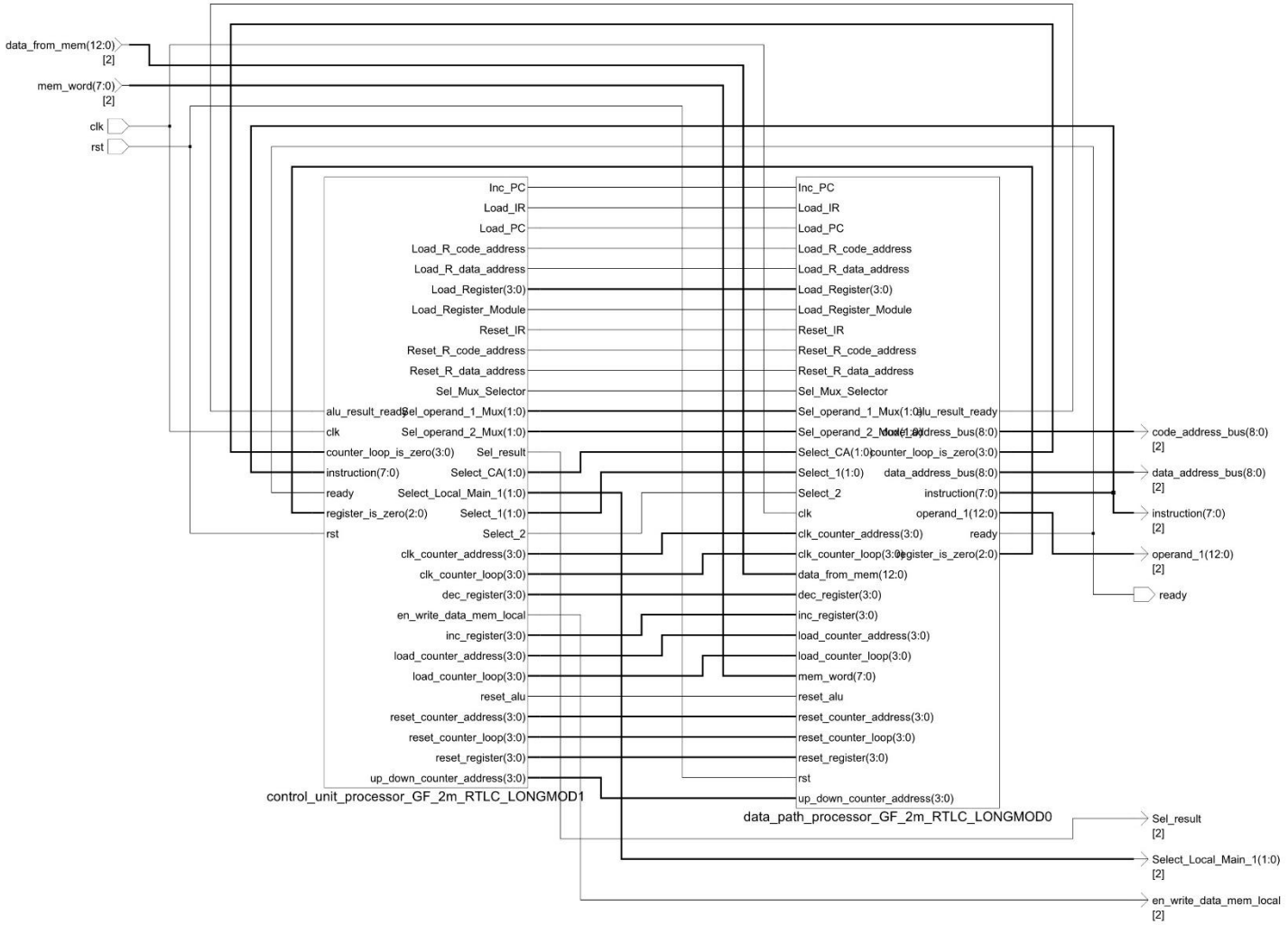
ДОДАТОК Д

Реалізація процесора Галуа на ПЛІС Xilinx

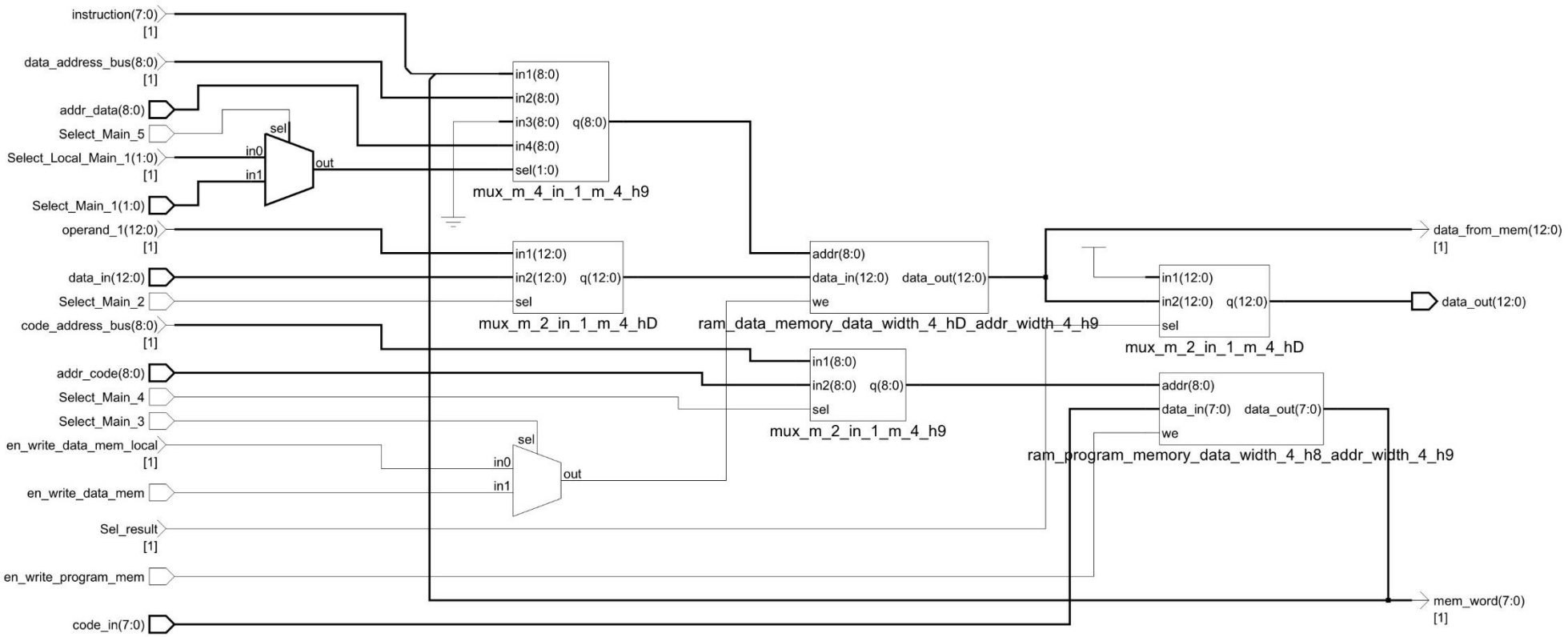
Додаток Д.1

Функціональні схеми процесора Галуа

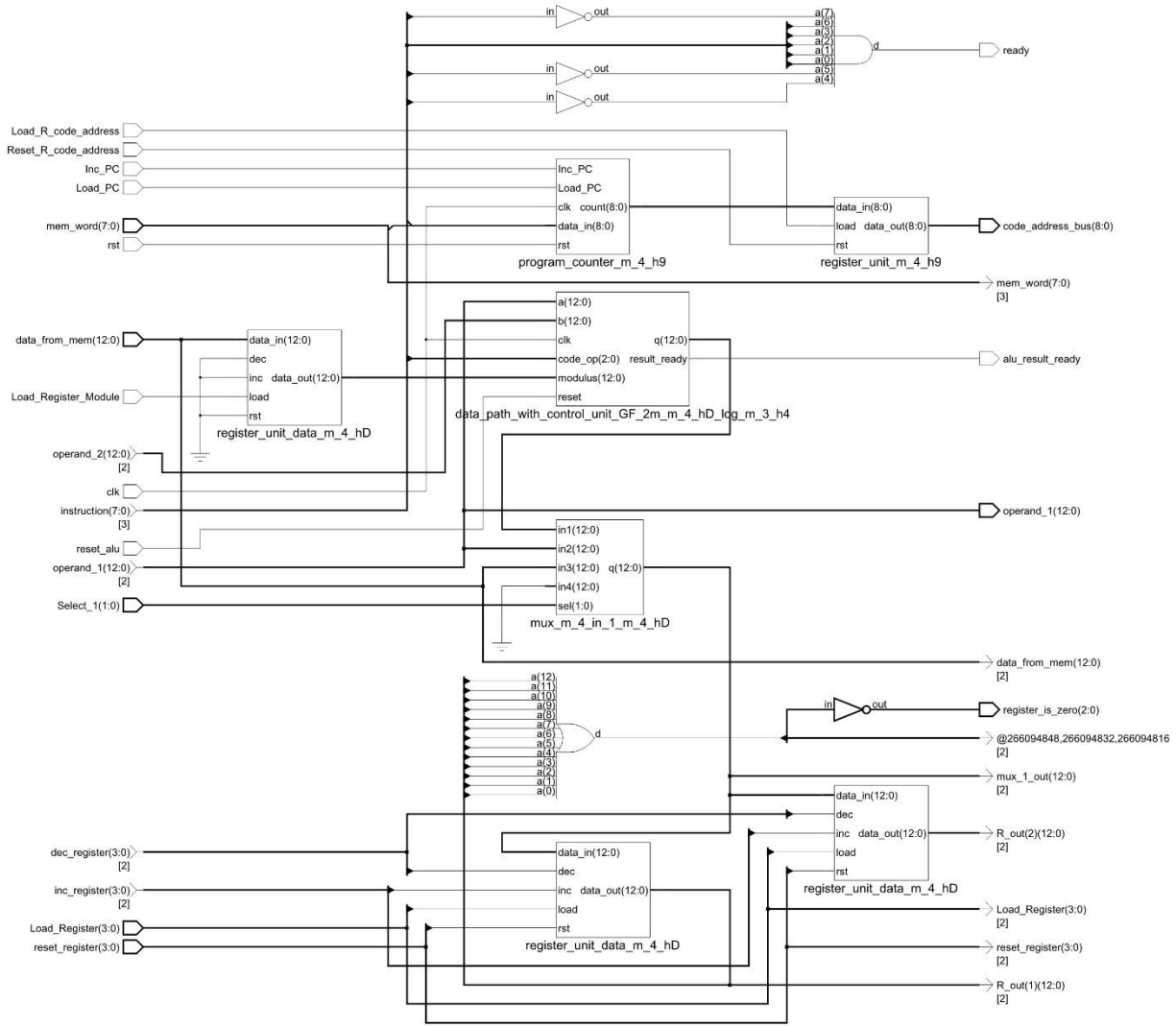
Функціональна схема процесора Галуа (частина 1)



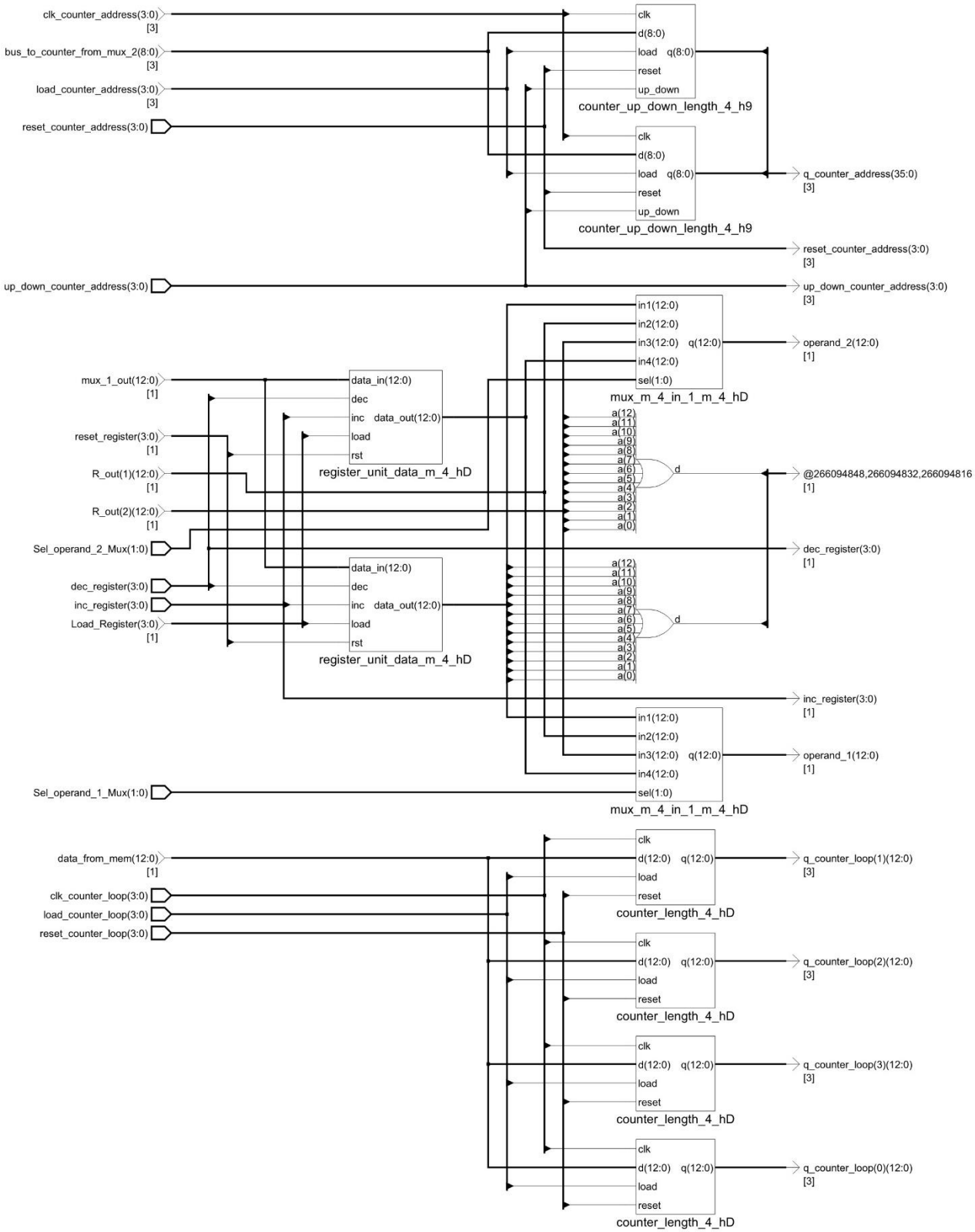
Функціональна схема процесора Галуа (частина 2)



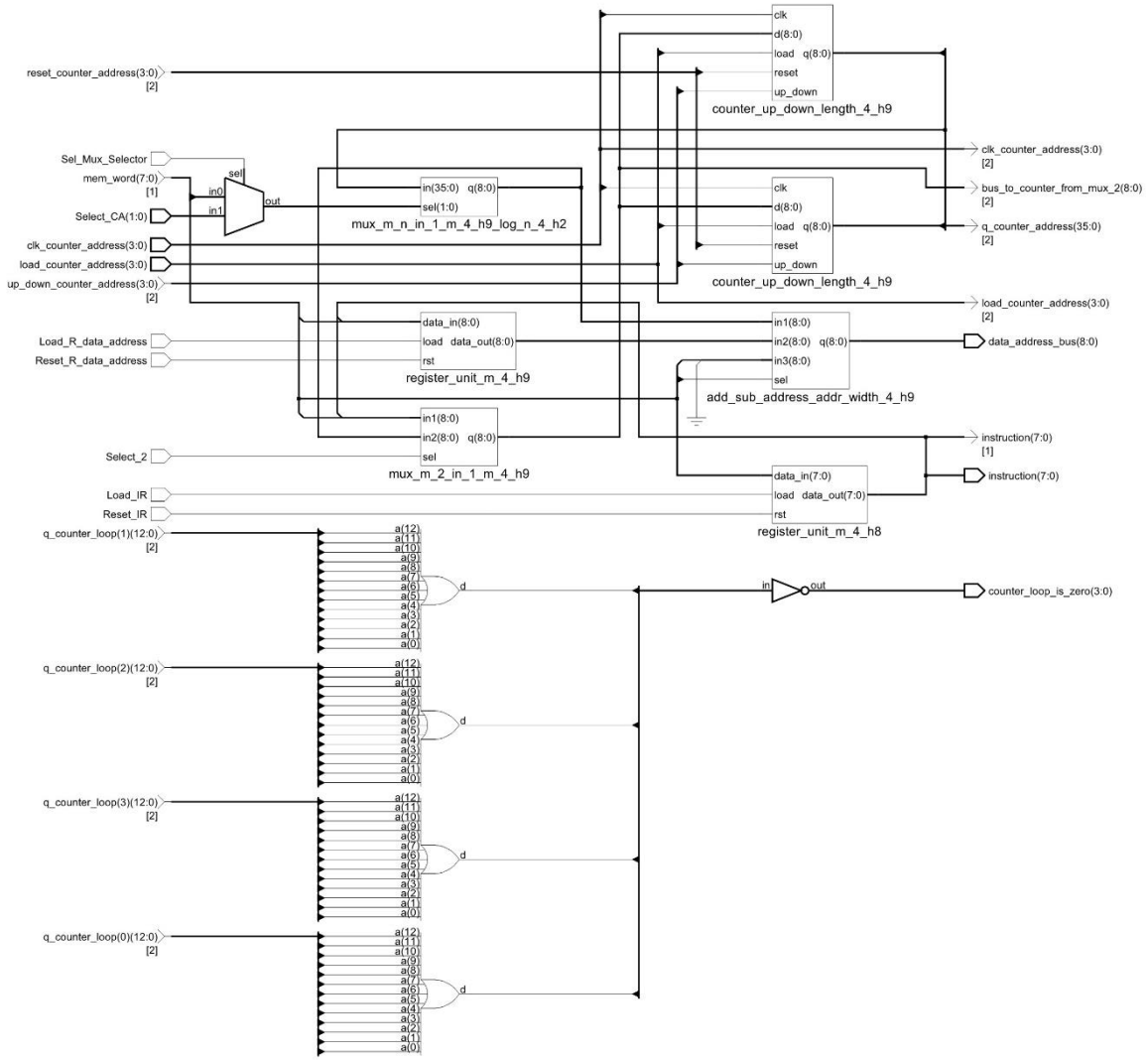
Функціональна схема операційного автомату процесора Галуа (частина 1)



Функціональна схема операційного автомату процесора Галуа (частина 2)



Функціональна схема операційного автомату процесора Галуа (частина 3)



Додаток Д.2

Програмний код процесора Галуа мовою Verilog

module test_processor_GF_2m

```
`timescale 1ps / 1ps

module test_processor_GF_2m;
	parameter m = 4'b1101;
	parameter log_m = 3'b100;
   parameter instruction_width = 4'b1000;
	parameter Sel_size = 2'b10;
	parameter addr_program_width = instruction_width + 1'b1;
	parameter addr_data_width = instruction_width + 1'b1;

	// Inputs
	reg clk, rst;
	reg en_write_data_mem, en_write_program_mem;
	reg [m-1: 0] data_in;
	reg [instruction_width-1: 0] code_in;
	reg [addr_program_width-1:0] addr_code;
	reg [addr_data_width-1:0] addr_data;
	reg [Sel_size-1:0] Select_Main_1;
	reg Select_Main_2, Select_Main_3, Select_Main_4, Select_Main_5;

	// Outputs
	wire [m-1:0] data_out;
	wire ready;

	// Instantiate the Unit Under Test (UUT)
	processor_GF_2m #(m, log_m) uut (
		.clk(clk),
		.rst(rst),
		.en_write_data_mem(en_write_data_mem),
		.en_write_program_mem(en_write_program_mem),
		.data_in(data_in),
		.code_in(code_in),
		.addr_code(addr_code),
		.addr_data(addr_data),
		.Select_Main_1(Select_Main_1),
		.Select_Main_2(Select_Main_2),
		.Select_Main_3(Select_Main_3),
		.Select_Main_4(Select_Main_4),
		.Select_Main_5(Select_Main_5),
		.data_out(data_out),
		.ready(ready)
	);
```

```
	initial begin

		clk = 0;
		rst = 1;
		en_write_data_mem = 0;
		en_write_program_mem = 0;
		data_in = 0;
		code_in = 0;
		addr_code = 0;
		addr_data = 0;
		Select_Main_1 = 0;
		Select_Main_2 = 0;
		Select_Main_3 = 0;
		Select_Main_4 = 0;
		Select_Main_5 = 0;
		#50;
		rst = 0;
		#100;


	end

	always #10 clk = ~clk;

endmodule
```

module processor_GF_2m

```
`timescale 1ns / 1ps

module processor_GF_2m(
            clk, rst,
            en_write_data_mem,
            en_write_program_mem,
            Select_Main_1, Select_Main_2, Select_Main_3, Select_Main_4, Select_Main_5,
            addr_code, addr_data,
            data_in,
            code_in,
            data_out,
            ready
    );

      parameter m = 4'b1101, log_m = 3'b100;
      parameter instruction_width = 4'b1000;
      parameter addr_program_width = instruction_width + 1'b1;
      parameter addr_data_width = instruction_width + 1'b1;
      parameter log_count_register = 4'b0010;
      parameter count_counter_loop = 4'b0100;
      parameter log_count_counter_address = 4'b0010;
      parameter Sel_size = 2;

      localparam count_counter_address = 1 << log_count_counter_address;
      localparam count_register = 1 << log_count_register;

      input clk, rst, en_write_data_mem, en_write_program_mem;
      input [m-1:0] data_in;
      input [instruction_width-1:0] code_in;
      input [addr_program_width-1:0] addr_code;
      input [addr_data_width-1:0] addr_data;
      input [Sel_size-1:0] Select_Main_1;
      input Select_Main_2, Select_Main_3, Select_Main_4, Select_Main_5;
      output [m-1:0] data_out;
      output ready;

      wire [m-1:0] data_from_mem;
      wire [instruction_width-1:0] mem_word;
      wire reset_alu;
      wire [count_register-1:0] Load_Register, inc_register, dec_register, reset_register;
      wire [Sel_size-1:0] Sel_operand_1_Mux, Sel_operand_2_Mux, Select_1;
      wire Select_2;
```

```
	wire [log_count_counter_address-1:0] Select_CA;
	wire Load_PC, Inc_PC, Load_IR, Reset_IR, Load_R_code_address, Reset_R_code_address, Load_R_data_address, Reset_R_data_address,
                                                                                                                    Load_Register_Module;
	wire [count_counter_loop-1:0] clk_counter_loop, reset_counter_loop, load_counter_loop;
	wire [count_counter_address-1:0] clk_counter_address, reset_counter_address, load_counter_address, up_down_counter_address;

	wire [instruction_width-1: 0] instruction;
	wire [m-1:0] operand_1;
	wire [addr_program_width-1:0] code_address_bus;
	wire [addr_data_width-1:0] data_address_bus;
	wire [count_counter_loop-1:0] counter_loop_is_zero;
	wire [count_register-2:0] register_is_zero;
	wire alu_result_ready;

	wire en_write_data_mem, Sel_Mux_Selector, Sel_result;
	wire [addr_data_width-1:0] address_bus_to_data_memory;
	wire [m-1:0] data_bus_to_data_memory;
	wire [addr_program_width-1:0] address_bus_to_program_memory;
	wire [Sel_size-1:0] bus_Select_Main_1, Select_Local_Main_1;

	wire bus_en_write_data_mem, ready;




	data_path_processor_GF_2m #(m, log_m, addr_program_width, addr_data_width, instruction_width, log_count_register,
count_counter_loop, log_count_counter_address) M0_Processor (

		// input for data_path
		.clk(clk),
		.rst(rst),

		.data_from_mem(data_from_mem),
		.mem_word(mem_word),

		.reset_alu(reset_alu),

		.Load_Register(Load_Register),
		.inc_register(inc_register),
		.dec_register(dec_register),
		.reset_register(reset_register),
```

```
        .Sel_operand_1_Mux(Sel_operand_1_Mux),
        .Sel_operand_2_Mux(Sel_operand_2_Mux),
        .Select_1(Select_1),
        .Select_2(Select_2),
        .Sel_Mux_Selector(Sel_Mux_Selector),
        .Select_CA(Select_CA),

        .Load_PC(Load_PC),
        .Inc_PC(Inc_PC),
        .Load_IR(Load_IR),
        .Reset_IR(Reset_IR),
        .Load_R_code_address(Load_R_code_address),
        .Reset_R_code_address(Reset_R_code_address),
        .Load_R_data_address(Load_R_data_address),
        .Reset_R_data_address(Reset_R_data_address),
        .Load_Register_Module(Load_Register_Module),

        .clk_counter_loop(clk_counter_loop),
        .reset_counter_loop(reset_counter_loop),
        .load_counter_loop(load_counter_loop),

        .clk_counter_address(clk_counter_address),
        .reset_counter_address(reset_counter_address),
        .load_counter_address(load_counter_address),
        .up_down_counter_address(up_down_counter_address),

         // output for data_path
        .instruction(instruction),
        .operand_1(operand_1),
        .code_address_bus(code_address_bus),
        .data_address_bus(data_address_bus),
        .counter_loop_is_zero(counter_loop_is_zero),
        .register_is_zero(register_is_zero),
        .alu_result_ready(alu_result_ready),
        .ready(ready)
);
```

```
control_unit_processor_GF_2m #(m, log_m, addr_program_width, instruction_width, log_count_register, count_counter_loop,
                                                                                    log_count_counter_address) M1_Controller (

        // input for control_unit
       .clk(clk),
       .rst(rst),
       .instruction(instruction),
       .counter_loop_is_zero(counter_loop_is_zero),
       .register_is_zero(register_is_zero),
       .alu_result_ready(alu_result_ready),
       .ready(ready),


       // output for control_unit
       .reset_alu(reset_alu),

       .Load_Register(Load_Register),
       .inc_register(inc_register),
       .dec_register(dec_register),
       .reset_register(reset_register),

       .Sel_operand_1_Mux(Sel_operand_1_Mux),
       .Sel_operand_2_Mux(Sel_operand_2_Mux),
       .Select_1(Select_1),
       .Select_2(Select_2),
       .Sel_Mux_Selector(Sel_Mux_Selector),
       .Select_CA(Select_CA),

       .Load_PC(Load_PC),
       .Inc_PC(Inc_PC),
       .Load_IR(Load_IR),
       .Reset_IR(Reset_IR),
       .Load_R_code_address(Load_R_code_address),
       .Reset_R_code_address(Reset_R_code_address),
       .Load_R_data_address(Load_R_data_address),
       .Reset_R_data_address(Reset_R_data_address),
       .Load_Register_Module(Load_Register_Module),

       .clk_counter_loop(clk_counter_loop),
       .reset_counter_loop(reset_counter_loop),
       .load_counter_loop(load_counter_loop),
```

```
        .clk_counter_address(clk_counter_address),
        .reset_counter_address(reset_counter_address),
        .load_counter_address(load_counter_address),
        .up_down_counter_address(up_down_counter_address),

        .Select_Local_Main_1(Select_Local_Main_1),
        .Sel_result(Sel_result),
        .en_write_data_mem_local(en_write_data_mem_local)
    );



    ram_program_memory #(instruction_width, addr_program_width) M2_SRAM (
        .data_out(mem_word),
        .data_in(code_in),
        .addr(address_bus_to_program_memory),
        .we(en_write_program_mem)
    );

    ram_data_memory #(m, addr_data_width) M3_SRAM (
        .data_out(data_from_mem),
        .data_in(data_bus_to_data_memory),
        .addr(address_bus_to_data_memory),
        .we(bus_en_write_data_mem)
    );


    mux_m_4_in_1 #(addr_data_width) Main_Mux_1 (.q(address_bus_to_data_memory), .in1({ instruction[0], mem_word }),
                                .in2(data_address_bus), .in3({addr_data_width{1'b0}}), .in4(addr_data), .sel(bus_Select_Main_1));
    mux_m_2_in_1 #(m) Main_Mux_2 (.q(data_bus_to_data_memory), .in1(operand_1), .in2(data_in), .sel(Select_Main_2));
    mux_m_2_in_1 #(1) Main_Mux_3 (.q(bus_en_write_data_mem), .in1(en_write_data_mem_local), .in2(en_write_data_mem), .sel(Select_Main_3));
    mux_m_2_in_1 #(addr_program_width) Main_Mux_4 (.q(address_bus_to_program_memory), .in1(code_address_bus), .in2(addr_code), .sel(Select_Main_4));
    mux_m_2_in_1 #(2) Main_Mux_5 (.q(bus_Select_Main_1), .in1(Select_Local_Main_1), .in2(Select_Main_1), .sel(Select_Main_5));


    mux_m_2_in_1 #(m) Mux_Result (.q(data_out), .in1({m{1'bz}}), .in2(data_from_mem), .sel(Sel_result));

endmodule
```

module data_path_processor_GF_2m

```
`timescale 1ns / 1ps

module data_path_processor_GF_2m #(parameter m = 3'b100, log_m = 2'b10, addr_program_width = 3'b100, addr_data_width = 3'b100,
         instruction_width = 4'b1000, log_count_register = 4'b0010, count_counter_loop = 4'b0010, log_count_counter_address = 4'b0001)
       (
              clk, rst,
              data_from_mem,
              mem_word,
              reset_alu,
              Load_Register, inc_register, dec_register, reset_register,
              Load_PC, Inc_PC, Load_IR, Reset_IR, Load_R_data_address, Reset_R_data_address, Load_R_code_address, Reset_R_code_address,
                                                                                                                    Load_Register_Module,
              Sel_operand_1_Mux, Sel_operand_2_Mux, Select_1,
              Select_2, Sel_Mux_Selector,
              Select_CA,
              clk_counter_loop, reset_counter_loop, load_counter_loop,
              clk_counter_address, reset_counter_address, load_counter_address, up_down_counter_address,

              instruction,
              operand_1,
              code_address_bus,
              data_address_bus,
              counter_loop_is_zero,
              register_is_zero,
              alu_result_ready,
              ready
    );


       localparam op_size = 4;
       localparam Sel_size = 2;
       localparam length = 1'b1;
       localparam count_counter_address = 1 << log_count_counter_address;
       localparam count_register = 1 << log_count_register;

       input clk, rst;
       input [m-1:0] data_from_mem;
       input [instruction_width-1:0] mem_word;
       input reset_alu;
       input [count_register-1:0] Load_Register, inc_register, dec_register, reset_register;
```

```
input Load_PC, Inc_PC, Load_IR, Reset_IR, Load_R_data_address, Reset_R_data_address, Load_R_code_address, Reset_R_code_address,
                                                                                                                    Load_Register_Module;
input [Sel_size-1: 0] Sel_operand_1_Mux, Sel_operand_2_Mux, Select_1;
input Select_2, Sel_Mux_Selector;
input [log_count_counter_address-1:0] Select_CA;
input [count_counter_loop-1:0] clk_counter_loop, reset_counter_loop, load_counter_loop;
input [count_counter_address-1:0] clk_counter_address, reset_counter_address, load_counter_address, up_down_counter_address;

output [instruction_width-1:0] instruction;
output [m-1:0] operand_1;
output [addr_program_width-1:0] code_address_bus;
output [addr_data_width-1:0] data_address_bus;
output [count_counter_loop-1:0] counter_loop_is_zero;
output [count_register-2:0] register_is_zero;
output alu_result_ready, ready;

wire [addr_program_width*count_counter_address-1:0] q_counter_address;
wire [addr_program_width-1:0] local_code_address_bus;
wire [addr_data_width-1:0] bus_to_counter_from_mux_2, q_counter_address_bus, data_from_r_data_address;
wire [m-1:0] q_counter_loop [count_counter_loop-1:0];
wire [m-1:0] R_out [count_register-1:0];
wire [m-1:0] operand_2, mux_1_out, module_to_alu, alu_out;
wire [log_count_counter_address-1:0] bus_from_mux_selector;
wire [op_size-1:0] opcode = instruction [instruction_width-1: instruction_width-op_size];


genvar i;
generate
      for (i = 0; i < count_register; i = i + 1)
      begin
            register_unit_data #(m) Reg (.data_out(R_out[i]), .data_in(mux_1_out), .load(Load_Register[i]),
                                                      .inc(inc_register[i]), .dec(dec_register[i]), .rst(reset_register[i]));

            if (i < count_register - 1)
                  assign register_is_zero[i] = ~( | R_out[i] );
      end
endgenerate

register_unit_data #(m) Reg_Module (.data_out(module_to_alu), .data_in(data_from_mem), .load(Load_Register_Module), .inc(1'b0),
                                                                                                    .dec(1'b0), .rst(1'b0));
program_counter #(addr_program_width) PC (.count(local_code_address_bus), .data_in({instruction[length-1:0], mem_word}),
                                                                  .Load_PC(Load_PC), .Inc_PC(Inc_PC), .clk(clk), .rst(rst));
register_unit #(addr_program_width) R_code_address (.data_out(code_address_bus), .data_in(local_code_address_bus),
                                                                    .load(Load_R_code_address), .rst(Reset_R_code_address));
```

```
register_unit #(instruction_width) IR (.data_out(instruction), .data_in(mem_word), .load(Load_IR), .rst(Reset_IR));
assign ready = instruction == 'b01001111;

mux_m_4_in_1 #(m) Mux_OP_1 (.q(operand_1), .in1(R_out[0]), .in2(R_out[1]), .in3(R_out[2]), .in4(R_out[3]), .sel(Sel_operand_1_Mux));
mux_m_4_in_1 #(m) Mux_OP_2 (.q(operand_2), .in1(R_out[0]), .in2(R_out[1]), .in3(R_out[2]), .in4(R_out[3]), .sel(Sel_operand_2_Mux));
mux_m_4_in_1 #(m) Mux_1 (.q(mux_1_out), .in1(alu_out), .in2(operand_1), .in3(data_from_mem), .in4({m{1'b0}}), .sel(Select_1));

generate
	for (i = 0; i < count_counter_loop; i = i + 1)
	begin
		counter #(m) counter_loop (
				.clk(clk_counter_loop [i]),
				.reset(reset_counter_loop [i]),
				.load(load_counter_loop [i]),
				.d(data_from_mem),
				.q(q_counter_loop [i])
			);
		assign counter_loop_is_zero [i] = ~(|q_counter_loop [i]);
	end
endgenerate


mux_m_2_in_1 #(addr_data_width) Mux_2 (.q(bus_to_counter_from_mux_2), .in1({ instruction[length-1:0], mem_word }),
                                                                       .in2(q_counter_address_bus), .sel(Select_2));


generate
	for (i = 0; i < count_counter_address; i = i + 1)
		counter_up_down #(addr_data_width) counter_address (
				.clk(clk_counter_address [i]),
				.reset(reset_counter_address [i]),
				.load(load_counter_address [i]),
				.up_down(up_down_counter_address [i]),
				.d(bus_to_counter_from_mux_2),
				.q(q_counter_address [(i+1)*addr_program_width-1:i*addr_program_width])
			);
endgenerate

mux_m_2_in_1 #(log_count_counter_address) Mux_Selector (.q(bus_from_mux_selector), .in1(mem_word[7:8-log_count_counter_address]),
                                                                                   .in2(Select_CA), .sel(Sel_Mux_Selector));
```

```
    mux_m_n_in_1 #(addr_data_width, log_count_counter_address) mux_counter_address (
           .sel(bus_from_mux_selector),
           .in(q_counter_address),
           .q(q_counter_address_bus)
           );



    register_unit #(addr_data_width) R_data_address (.data_out(data_from_r_data_address), .data_in({instruction[0], mem_word}),
                                                                                    .load(Load_R_data_address), .rst(Reset_R_data_address));

    add_sub_address #(addr_data_width) add_sub_addr
    (
           .sel(mem_word[4]),
           .in1(q_counter_address_bus),
           .in2(data_from_r_data_address),
           .in3({5'b00000, mem_word[3:0]}),
           .q(data_address_bus)
);


    data_path_with_control_unit_GF_2m #(m, log_m) ALU (
           .clk(clk),
           .reset(reset_alu),
           .code_op(opcode[op_size-2:0]),
           .a(operand_1),
           .b(operand_2),
           .modulus(module_to_alu),
           .result_ready(alu_result_ready),
           .q(alu_out)
    );


endmodule
```

module control_unit_processor_GF_2m

```
`timescale 1ns / 1ps

module control_unit_processor_GF_2m(
            clk, rst,
            instruction,
            counter_loop_is_zero,
            register_is_zero,
            alu_result_ready,
            ready,

            reset_alu,
            Load_Register, inc_register, dec_register, reset_register,
            Sel_operand_1_Mux, Sel_operand_2_Mux, Select_1,
            Select_2, Sel_Mux_Selector,
            Select_CA,
            Load_PC, Inc_PC, Load_IR, Reset_IR, Load_R_code_address, Reset_R_code_address, Load_R_data_address, Reset_R_data_address,
                                                                                                                  Load_Register_Module,
            clk_counter_loop, reset_counter_loop, load_counter_loop,
            clk_counter_address, reset_counter_address, load_counter_address, up_down_counter_address,
            Select_Local_Main_1,
            Sel_result, en_write_data_mem_local
      );


      parameter m = 8;
      parameter log_m = 3;
      parameter addr_program_width = 4'b1000;
      parameter instruction_width = 4'b1000;
      parameter log_count_register = 4'b0010;
      parameter count_counter_loop = 4'b0010;
      parameter log_count_counter_address = 4'b0010;
      parameter op_size = 4, src_size = 2, dst_size = 2;

      localparam count_counter_address = 1 << log_count_counter_address;
      localparam count_register = 1 << log_count_register;
      localparam state_size = 5, Sel_size = 2;
      localparam R0 = 0, R1 = 1, R2 = 2, R3 = 3;
```

```
input clk, rst;
input [instruction_width-1:0] instruction;
input [count_counter_loop-1:0] counter_loop_is_zero;
input [count_register-2:0] register_is_zero;
input alu_result_ready;
input ready;

output reg reset_alu;
output reg [count_register-1:0] Load_Register, inc_register, dec_register, reset_register;
output reg [Sel_size-1:0] Sel_operand_1_Mux, Sel_operand_2_Mux, Select_1;
output reg Select_2, Sel_Mux_Selector;
output reg [log_count_counter_address-1:0] Select_CA;
output reg Load_PC, Inc_PC, Load_IR, Reset_IR, Load_R_code_address, Reset_R_code_address, Load_R_data_address,
                                                                                Reset_R_data_address, Load_Register_Module;
output reg [count_counter_loop-1:0] clk_counter_loop, reset_counter_loop, load_counter_loop;
output reg [count_counter_address-1:0] clk_counter_address, reset_counter_address, load_counter_address,
                                                                                                    up_down_counter_address;
output reg [Sel_size-1:0] Select_Local_Main_1;
output reg Sel_result, en_write_data_mem_local;



wire [op_size-1: 0] code_op = instruction [instruction_width-1: instruction_width - op_size];
wire [src_size-1: 0] dst = instruction [dst_size + src_size - 1: src_size];
wire [dst_size-1: 0] src = instruction [src_size - 1:0];

reg [state_size-1: 0] state, state_next;


localparam SELECT_MODULE_1 = 5'b10000;
localparam SELECT_MODULE_2 = 5'b10001;
localparam INIT =   5'b10010;
localparam CALC_0 = 5'b10011;
localparam CALC_1 = 5'b10100;
localparam CALC_2 = 5'b10101;
localparam CALC_3 = 5'b10110;
localparam CALC_4 = 5'b10111;
localparam CALC_5 = 5'b11000;
localparam CALC_6 = 5'b11001;
localparam CALC_7 = 5'b11010;
```

```
	localparam ADD_SUB = 4'b0000;
	localparam MULT = 4'b0001;
	localparam DIV = 4'b0010;
	localparam POW = 4'b0011;
	localparam INVM = 4'b0100;
	localparam CDP_INVA = 4'b0101;
	localparam CPD_SUB = 4'b0110;
	localparam MOV = 4'b0111;
	localparam MOV_A = 4'b1000;
	localparam MOV_ARRAY = 4'b1001;
	localparam JMP = 4'b1010;
	localparam LOOP = 4'b1011;
	localparam LOAD_CA = 4'b1100;
	localparam LOAD_CA_A = 4'b1101;
	localparam INC_DEC = 4'b1110;
	localparam OUT = 4'b1111;


integer i;

always @( posedge clk  or negedge clk )
	begin

		case ( state )

			SELECT_MODULE_1 :
				begin
					Reset_IR <= 1'b1;
					Select_Local_Main_1 <= 2'b10; // беремо адресу пам'яті даних рівну 0
					en_write_data_mem_local <= 1'b0;
					Load_Register_Module <= 1'b0;
				end

			SELECT_MODULE_2 :
				begin
					Reset_IR <= 1'b0;
					Load_Register_Module <= 1'b1;
				end

			INIT :
				begin
					Inc_PC <= 1'b0;
					Reset_R_code_address <= 1'b0;
					Load_R_code_address <= 1'b1;	// завантаження регістру адрес значенням з лічильника адрес
```

```
                Load_Register_Module <= 1'b0;
                Load_PC <= 1'b0;
                Load_IR <= 1'b0;

                for (i = 0; i < count_register; i = i + 1)
                begin
                        Load_Register[i] <= 1'b0;
                        inc_register[i] <= 1'b0;
                        dec_register[i] <= 1'b0;
                        reset_register[i] <= 1'b0;
                end

                for (i = 0; i < count_counter_loop; i = i + 1)
                begin
                        clk_counter_loop [i] <= 1'b0;
                        load_counter_loop [i] <= 1'b0;
                        reset_counter_loop [i] <= 1'b0;
                end

                for (i = 0; i < count_counter_address; i = i + 1)
                begin
                        clk_counter_address [i] <= 1'b0;
                        load_counter_address [i] <= 1'b0;
                        up_down_counter_address [i] <= 1'b0;
                        reset_counter_address [i] <= 1'b0;
                end

                Select_Local_Main_1 <= 2'b00;  // беремо адресу пам'яті даних з пам'яті команд
                Sel_result <= 1'b0;            // видаємо на вихід третій стан
                Load_R_data_address <= 1'b0;
                Reset_R_data_address <= 1'b0;

                Sel_Mux_Selector <= 1'b0;
                Select_CA <= 2'b00;
                en_write_data_mem_local <= 1'b0;
                reset_alu <= 1'b1;
        end

CALC_0 :
        begin
                Load_R_code_address <= 1'b0;
                Load_IR <= 1'b1;              // завантаження регістру команд даними з виходу пам'яті команд
                Inc_PC <= 1'b1;
        end
```

```
CALC_1 :
            begin
                  Inc_PC <= 1'b0;
                  Load_IR <= 1'b0;
                  Load_R_code_address <= 1'b1;  // завантаження регістру адрес даними з лічильника адрес
            end

CALC_2 :
            begin
                  Load_R_code_address <= 1'b0;
                  Inc_PC <= 1'b1;
            end

CALC_3 :
            begin
                   Inc_PC <= 1'b0;
                case ( code_op )

                        MOV_ARRAY, OUT :
                                    // то виводиться елемент масиву і це трибайтна команда
                                    if ( (code_op == MOV_ARRAY) || src[1] )
                                          begin
                                                // беремо адресу пам'яті даних з суматора
                                                Select_Local_Main_1 <= 2'b01;
                                                // завантажуємо значення нульового біта першого байту та
                                                другого байту (початкова адреса масиву) у регістр аддрес
                                                даних перед зчитуванням третього байту
                                                Load_R_data_address <= 1'b1;
                                                // завантаження регістру адрес команд значенням з
                                                лічильника адрес команд
                                                Load_R_code_address <= 1'b1;
                                          end
                                    else
                                          // видаємо на вихід значення з пам'яті даних
                                          Sel_result <= 1'b1;
```

```
    INC_DEC :
                    if (src[1])
                       up_down_counter_address [dst] <= ~src[0];
                    else
                              if (src[0])
                                     dec_register[dst] <= 1'b1;
                              else
                                     inc_register[dst] <= 1'b1;


    LOAD_CA :
                  begin
                              Select_2 <= 1'b1;
                              Sel_Mux_Selector <= 1'b1;
                              Select_CA <= src;
                              load_counter_address [dst] <= 1'b1;
                  end

    LOAD_CA_A :
                  begin
                              Select_2 <= 1'b0;
                              load_counter_address [dst] <= 1'b1;
                  end

   MOV :
                  begin
                        Select_1 <= 2'b01;
                        Sel_operand_1_Mux <= src;
                  end

   MOV_A :
                  if (src[1]) // якщо 1, то виконуємо запис з пам'яті в регістр
                              // видача на шину Bus_1 значення числа з пам'яті даних
                              Select_1 <= 2'b10;
                  else         // якщо 0, то виконуємо запис з регістру в пам'ять
                              // видача на шину operand_1 значення числа з регістру, що має
                              номер dst
                              Sel_operand_1_Mux <= dst;

JMP :
     // безумовний перехід, якщо {dst,src[1]} == 3'b111
     if ({dst,src[1]}==3'b111 || (src[1] && register_is_zero[dst]) || (~src[1] && counter_loop_is_zero[dst]))
                 Load_PC <= 1'b1;    // завантаження лічильника адрес даними команди JMP
```

```
                        LOOP :
                                    load_counter_loop [dst] <= 1'b1;    // завантаження лічильника циклу даними

                        ADD_SUB, MULT, DIV, POW, INVM, CDP_INVA, CPD_SUB :
                                    begin
                                                reset_alu <= 1'b0;
                                                // видача на перший вхід ALU значення з регістру, що має номер dst
                                                Sel_operand_1_Mux <= dst;
                                                // видача на шину другого операнду коду з регістру, що має
                                                номер src (другий операнд для ALU)
                                                Sel_operand_2_Mux <= src;
                                                // видача на шину Bus_1 результату з ALU
                                                Select_1 <= 1'b0;
                                    end

                    endcase
            end

CALC_4 :
            begin
                case ( code_op )

                        MOV_ARRAY, OUT :
                                    begin
                                                Load_R_data_address <= 1'b0;
                                                Load_R_code_address <= 1'b0;
                                                Inc_PC <= 1'b1;
                                                // видаємо на вихід значення з пам'яті даних
                                                if ( code_op == OUT )
                                                      Sel_result <= 1'b1;
                                    end

                        LOAD_CA :
                                    clk_counter_address [dst] <= 1'b1;

                        INC_DEC, LOAD_CA_A :
                                    clk_counter_address [dst] <= 1'b1;

                        MOV_A :
                                    if (src[1]) // якщо 1, то виконуємо запис з пам'яті в регістр
                                          Load_Register[dst] <= 1'b1;
                                    else        // якщо 0, то виконуємо запис з регістру в пам'ять
                                          en_write_data_mem_local <= 1'b1;
```

```
                                        ADD_SUB, MULT, DIV, POW, INVM, CDP_INVA, CPD_SUB, MOV :
                                                        Load_Register[dst] <= 1'b1;

                                        LOOP, JMP :
                                                        if (~src[1]) // JMP саме для LOOP
                                                               clk_counter_loop [dst] <= 1'b1;
                                         endcase
                                end

                CALC_5 :
                                begin
                                       Inc_PC <= 1'b0;

                                       if (src[1]) // якщо 1, то виконуємо запис з пам'яті в регістр
                                              Select_1 <= 2'b10;    // видача на шину Bus_1 значення числа з пам'яті даних
                                       else        // якщо 0, то виконуємо запис з регістру в пам'ять
                                              Sel_operand_1_Mux <= dst;    // видача на шину operand_1 значення числа з регістру, що має номер dst
                                end

                CALC_6 :
                                if (src[1]) // якщо 1, то виконуємо запис з пам'яті в регістр
                                       Load_Register[dst] <= 1'b1;
                                else     // якщо 0, то виконуємо запис з регістру в пам'ять
                                       en_write_data_mem_local <= 1'b1;

        endcase
    end



always @( posedge clk or negedge clk )
    begin

        if(rst == 1'b1)
            if (state == SELECT_MODULE_1 || state == SELECT_MODULE_2)
                state <= SELECT_MODULE_2;
            else
                state <= SELECT_MODULE_1;
        else
            if (~ready)
                state <= state_next;

        case ( state )
```

```
SELECT_MODULE_1 :
            state_next = SELECT_MODULE_2;

SELECT_MODULE_2 :
            state_next = INIT;

INIT :
            state_next = CALC_0;

CALC_0 :
            state_next = CALC_1;

CALC_1 :
            if ( code_op == MOV_A || code_op == JMP || code_op == LOOP || code_op == LOAD_CA_A || code_op == MOV_ARRAY
                                                                                                    || code_op == OUT )
                  state_next = CALC_2;
            else
                  state_next = CALC_3;

CALC_2 :
            state_next = CALC_3;

CALC_3 :
                  case ( code_op )

                      LOOP, MOV, LOAD_CA, LOAD_CA_A, MOV_ARRAY :
                                    state_next = CALC_4;

                      INC_DEC :
                                    if (src[1])
                                          state_next = CALC_4;
                                    else
                                          state_next = INIT;

                      JMP :
                                    if (~src[1]) // JMP same для LOOP
                                          state_next = CALC_4;
                                    else
                                          state_next = INIT;

                      MOV_A :
                                    state_next = CALC_4;
```

```
                                    OUT :
                                                if ( src[1] ) // то виводиться елемент масиву і це трибайтна команда
                                                    state_next = CALC_4;
                                                else         // виводиться звичайна змінна і це двобайтна команда
                                                    state_next = INIT;

                                    ADD_SUB, MULT, DIV, POW, INVM, CDP_INVA, CPD_SUB, LOOP :
                                                if ( alu_result_ready )
                                                   state_next = CALC_4;

                                endcase

                CALC_4 :
                                case ( code_op )

                      ADD_SUB, MULT, DIV, POW, INVM, CDP_INVA, CPD_SUB, LOOP, LOOP, JMP, MOV, MOV_A, LOAD_CA, LOAD_CA_A, INC_DEC :
                                                state_next = INIT;

                                    MOV_ARRAY :
                                                state_next = CALC_5;

                                    OUT :
                                                state_next = INIT;

                                endcase

                CALC_5 :
                            state_next = CALC_6;

                CALC_6 :
                            state_next = INIT;

        endcase

    end


endmodule
```

module ram_program_memory

```
`timescale 1ns / 1ps

module ram_program_memory ( we, addr, data_in, data_out );

	parameter data_width = 8;
	parameter addr_width = 8;

	input we;
	input [addr_width - 1:0] addr;
	input [data_width - 1:0] data_in;
	output[data_width - 1:0] data_out;

	reg [addr_width - 1:0] addri;
	reg [data_width - 1:0] mem [(1'b1<<addr_width)-1:0];

	initial
		$readmemb("ram_init_program.txt", mem);

	always @( addr, we )
	begin
		if ( we )
			mem[addr] = data_in;
		addri = addr;
	end

	assign data_out = mem[addri];

endmodule
```

module ram_data_memory

```
`timescale 1ns / 1ps

module ram_data_memory ( we, addr, data_in, data_out );

	parameter data_width = 5;
	parameter addr_width = 8;

	input we;
	input [addr_width - 1:0] addr;
	input [data_width - 1:0] data_in;
	output[data_width - 1:0] data_out;

	reg [addr_width - 1:0] addri;
	reg [data_width - 1:0] mem [(1'b1<<addr_width)-1:0];

	initial
		$readmemb("ram_init_data.txt", mem);

	always @( addr, we )
	begin
		if ( we )
			mem[addr] = data_in;
		addri = addr;
	end

	assign data_out = mem[addri];

endmodule
```

module mux_m_2_in_1

```
`timescale 1ns / 1ps

module mux_m_2_in_1 (sel, in1, in2, q);

	parameter m = 10;

	input [m-1:0] in1, in2;
	output reg [m-1:0] q;
	input sel;

	always @(in1 or in2 or sel)
	case (sel)
		1'b0 : q <= in1;
		1'b1 : q <= in2;
	endcase

endmodule
```

module mux_m_4_in_1

```
`timescale 1ns / 1ps

module mux_m_4_in_1(sel, in1, in2, in3, in4, q);

	parameter m = 10;

	input [m-1:0] in1, in2, in3, in4;
	output reg [m-1:0] q;
	input [1:0] sel;

	always @(in1 or in2 or in3 or in4 or sel)
	case (sel)
		2'b00 : q <= in1;
		2'b01 : q <= in2;
		2'b10 : q <= in3;
		2'b11 : q <= in4;
	endcase

endmodule
```

module mux_m_n_in_1

```
`timescale 1ns / 1ps

module mux_m_n_in_1 (sel, in, q);

	parameter m = 10;
	parameter log_n = 10;
	localparam n = 1 << log_n;

	input [m*n-1:0] in;
	input [log_n-1:0] sel;
	output [m-1:0] q;

	genvar i,k;
	generate
		for (k = 0; k < m; k = k + 1)
		begin
			wire [n-1:0] tmp;
			for (i = 0; i < n; i = i + 1)
				assign tmp [i] = in[k+i*m];
			assign q[k] = tmp[sel];
		end
	endgenerate

endmodule
```

module register_unit_data

```
`timescale 1ns / 1ps

module register_unit_data ( data_out, data_in, load, inc, dec, /*clk,*/ rst );

	parameter m = 8;

	input [m-1:0] data_in;
	input load, inc, dec, rst;
	output reg [m-1:0] data_out;

	always @ (posedge dec or posedge inc  or posedge load or posedge rst)
		if (rst)
			data_out <= 0;
		else
			if (load)
				data_out <= data_in;
			else
				if (inc)
					data_out <= data_out + 1;
				else
					if (dec)
						data_out <= data_out - 1;

endmodule
```

module register_unit

```
`timescale 1ns / 1ps

module register_unit ( data_out, data_in, load, rst );

       parameter m = 8;

       input [m-1:0] data_in;
       input load,  rst;
       output reg [m-1:0] data_out;

       always @ ( posedge load or posedge rst )
              if (rst)
                     data_out <= 0;
              else
                     if (load)
                            data_out <= data_in;

endmodule
```

module program_counter

```
`timescale 1ns / 1ps

module program_counter ( count, data_in, Load_PC, Inc_PC, clk, rst );

       parameter m = 8;

       input [m-1:0] data_in;
       input Load_PC, Inc_PC, clk, rst;
       output reg [m-1: 0] count;

       always @ ( posedge clk or posedge rst )
       if (rst)
              count <= 0;
       else
              if (Load_PC)
                     count <= data_in;
       else
              if (Inc_PC)
                     count <= count +1;

endmodule
```

module counter

```
`timescale 1ns / 1ps

module counter#(parameter length = 4) ( clk, reset, load, d, q);


input   [length-1:0] d;
input   clk;
input   reset;
input   load;
output  [length-1:0] q;
reg     [length-1:0] cnt;


assign q = cnt;

always @ (posedge clk)
begin
    if (reset)
        cnt = 0;
    else if (load)
        cnt = d;
    else
        cnt = cnt - 1;
end

endmodule
```

module counter_up_down

```
`timescale 1ns / 1ps

module counter_up_down#(parameter length = 4) ( clk, reset, load, up_down, d, q);

input   [length-1:0] d;
input   clk, reset, load, up_down;
output  [length-1:0] q;
reg     [length-1:0] cnt;


assign q = cnt;

always @ (posedge clk)
begin
    if (reset)
        cnt = 0;
    else if (load)
        cnt = d;
    else
                    if (up_down)
                           cnt = cnt + 1;
                    else
                           cnt = cnt - 1;
end

endmodule
```

module add_sub_address

```
`timescale 1ns / 1ps

module add_sub_address(
              sel, in1, in2, in3, q
    );

       parameter addr_width = 3'b100;

       input sel;
       input [addr_width-1: 0] in1, in2, in3;
       output [addr_width-1: 0] q;

       reg [addr_width-1: 0] state;

       always @(in1 or in2 or in3 or sel)
              if (~sel)
                     state = in1 + in2 + in3;
              else
                     state = in1 + in2 - in3;

       assign q = state;

endmodule
```

ДОДАТОК Є. Інструкція з експлуатації інтегрованого середовища розробки “Асемблер Галуа”

Вимоги до комп’ютера та операційної системи

Необхідною умовою працездатності програмного модуля є використання персонального комп’ютера, що має процесор з тактовою частотою 1ГГц або більш продуктивний (рекомендовано 2ГГц), 1 Гб ОЗП для 32-х розрядної та 2 Гб ОЗП для 64-х розрядної системи (рекомендовано 4 Гb ОЗП) – для роботи з Windows 7, 8, 8.1 та 10. Окрім ОС на ПК має бути встановлена платформа .NET Framework не нижче версії 4.6. Інші додаткові програмні засоби, крім вищевказаних, на комп'ютері встановлювати не потрібно.

Керівництво з експлуатації

Програма “Асемблер Галуа IDE” забезпечує:

- Можливість написання програми мовою Асемблера Галуа в текстовому редакторі даного середовища розробки.
- Автоматичне підсвічування ключових слів тексту програми.
- Автоматичне вирівнювання тексту програми за допомогою табуляції.
- Виконання лексичного та синтаксичного аналізу тексту програми, з розпізнаванням типових лексичних та синтаксичних помилок.
- Динамічну компіляцію тексту програми.
- Перетворення тексту програми мовою Асемблера Галуа у машинний код.

Розглянемо докладніше функціональні можливості інтегрованого середовища розробки “Асемблер Галуа” (рис. Є.1).

Рис. Є.1. Вікно “Асемблер Галуа IDE”

Основне вікно програми складається з трьох областей: область пам’яті даних, область коду асемблера (текст програми мовою Асемблера Галуа) та область пам’яті команд.

Область коду асемблера призначена для створення тексту програми мовою Асемблера процесора Галуа (рис. Є.2). В області пам’ять даних виводиться машинний код пам’яті даних (вміст пам’яті даних), а в області пам’яті команд машинний код програми (вміст пам’яті команд). Рядки пам’яті команд та пам’яті даних автоматично коментуються компілятором, з метою наочності, тобто щоб користувачу було легко розуміти який саме рядок машинного коду відповідає певній частині тексту програми. Окрім цього автоматично виконується нумерація рядків машинного коду пам’яті команд та даних, з метою надати можливість програмісту легко бачити за якою адресою записані ті чи інші команди в пам’яті команд та дані у пам’яті даних.

Розглянемо докладніше меню інтегрованого середовища розробки “Асемблер Галуа”.

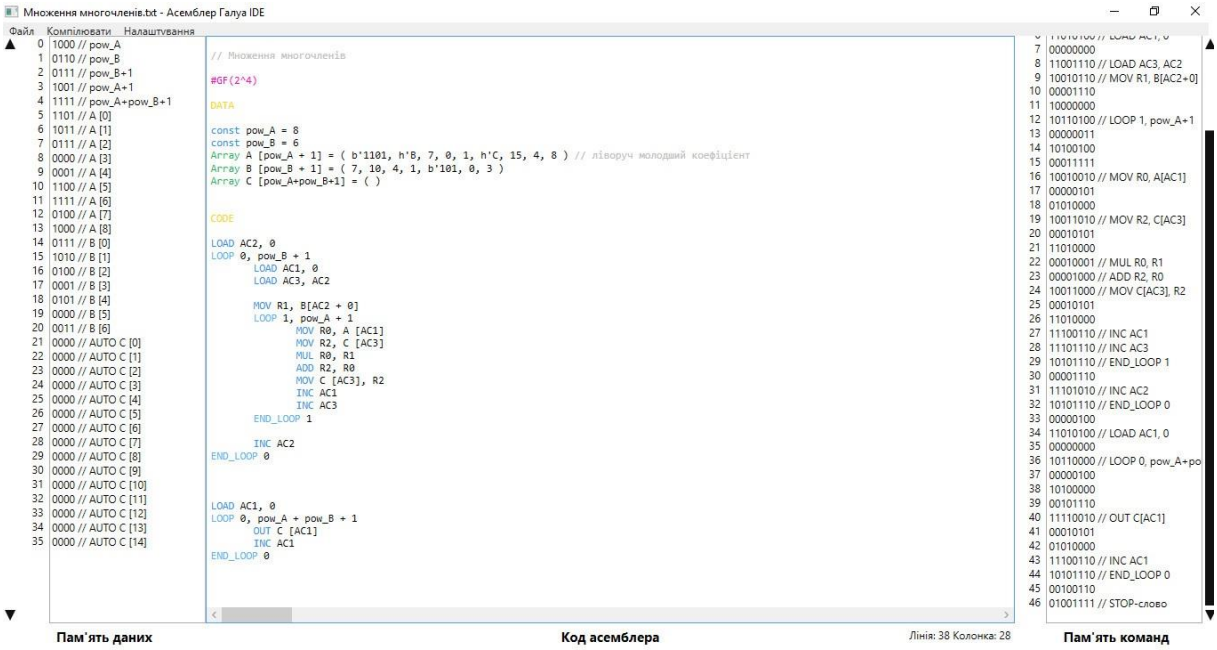


Рис. Є.2. Програма множення многочленів

Меню Файл (рис. Є.3) містить в якості підменю основні команди для роботи з текстовим редактором такі як “Створити новий файл”, “Відкрити файл”, “Зберегти”, “Зберегти як”, “Закрити файл” та “Вийти з програми”. Більшість з перелічених команд можуть виконуватись по натисканні відповідної “гарячої клавіші”.

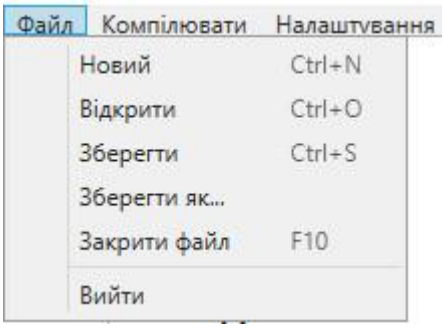


Рис. Є.3. Меню Файл

Меню Компілювати не містить пунктів підменю та працює як кнопка за натисканням на яку виконується лексичний та синтаксичний аналіз програми мовою Асемблера Галуа. У випадку наявності помилок з’являється відповідне інформаційне повідомлення. Якщо компіляція пройшла успішно, то машинний код записується у текстові файли (файл

пам’яті даних та файл пам’яті команд). Замість натискання на кнопку Компілювати можливо виконати ті самі дії шляхом натискання “гарячої клавіші” F5.

Меню Налаштування (рис. Є.4) містить в якості підменю елементи графічного інтерфейсу, що дозволяють користувачу керувати параметрами роботи програми “Асемблер Галуа IDE”. Кожен з цих параметрів має два стани, тому для керування ними було використано елементи графічного інтерфейсу “check box”.

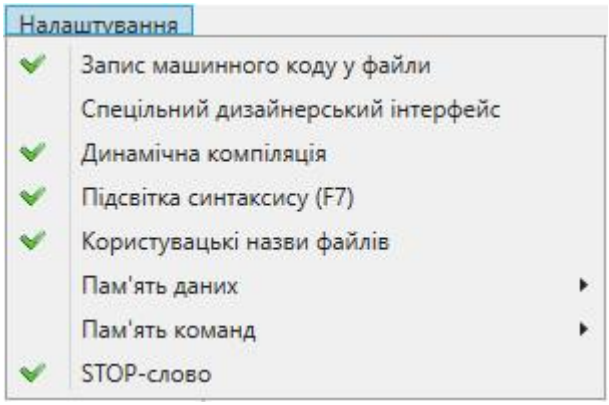


Рис. Є.4. Меню Налаштування

Параметр “Запис машинного коду у файли” вказує на те чи буде виконуватись запис машинного коду у файли після компіляції.

Параметр “Спеціальний дизайнерський інтерфейс” дозволяє перейти до інтерфейсу програми “Асемблер Галуа IDE” з темним фоном текстового редактора (рис. Є.5).

Параметр “Динамічна компіляція” при включеному режимі забезпечує виконання компіляція під час редагування програмного коду та відображає результат компіляції в області пам’яті команд та пам’яті даних вікна програми. Важливою особливість режиму динамічної компіляції є те, що вона виконується тільки коли написано коректний код. Якщо наявні певні помилки в коді, то динамічна компіляція не виконується, а інформацію про наявні помилки можна отримати при виконанні компіляції у ручному режимі (пункт меню Компілювати або клавіша F5).

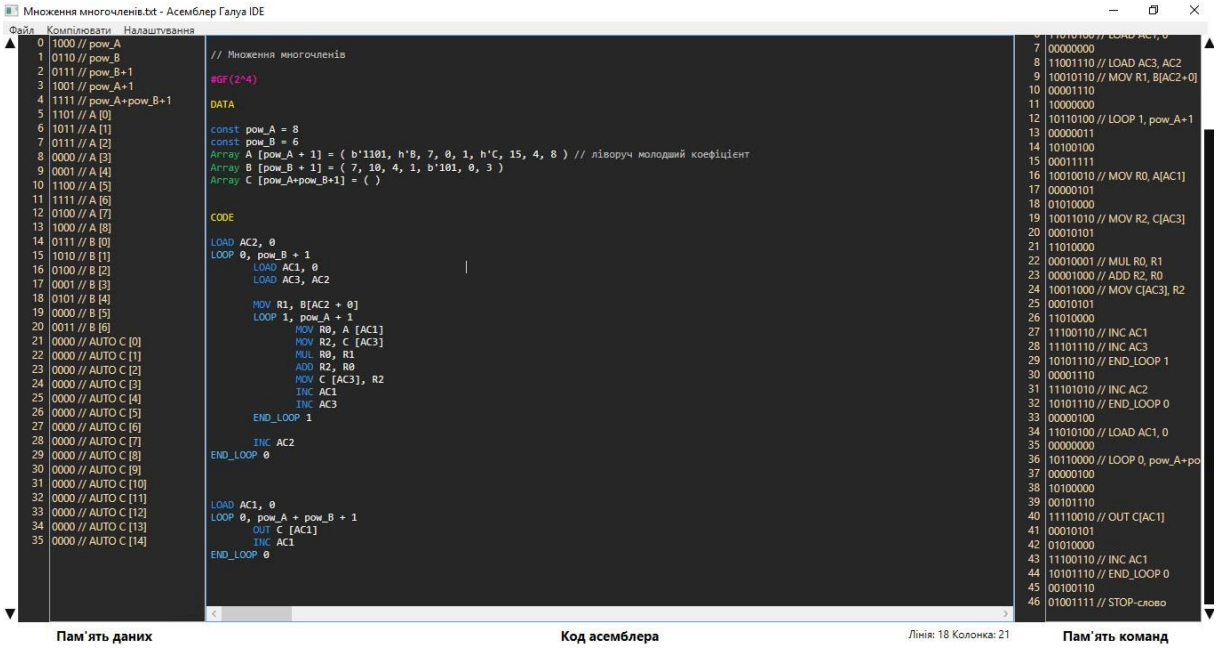


Рис. Є.5. Спеціальний дизайнерський інтерфейс

Параметр “Підсвітка синтаксису” дозволяє включити або виключити підсвітку програмного коду. Також є можливість виконувати підсвітку в ручному режимі за натисканням “гарячої клавіші” F7.

Параметр “Користувацькі назви файлів” відповідає за можливість вводити назву файлів для запису машинного коду пам’яті даних та пам’яті програм (рис. Є.6). Якщо цей параметр знаходиться у виключеному стані, то назва файлу пам’яті команд та даних автоматично формується як назва з файлу з текстом програми плюс суфікс “_code” та “_data” відповідно.

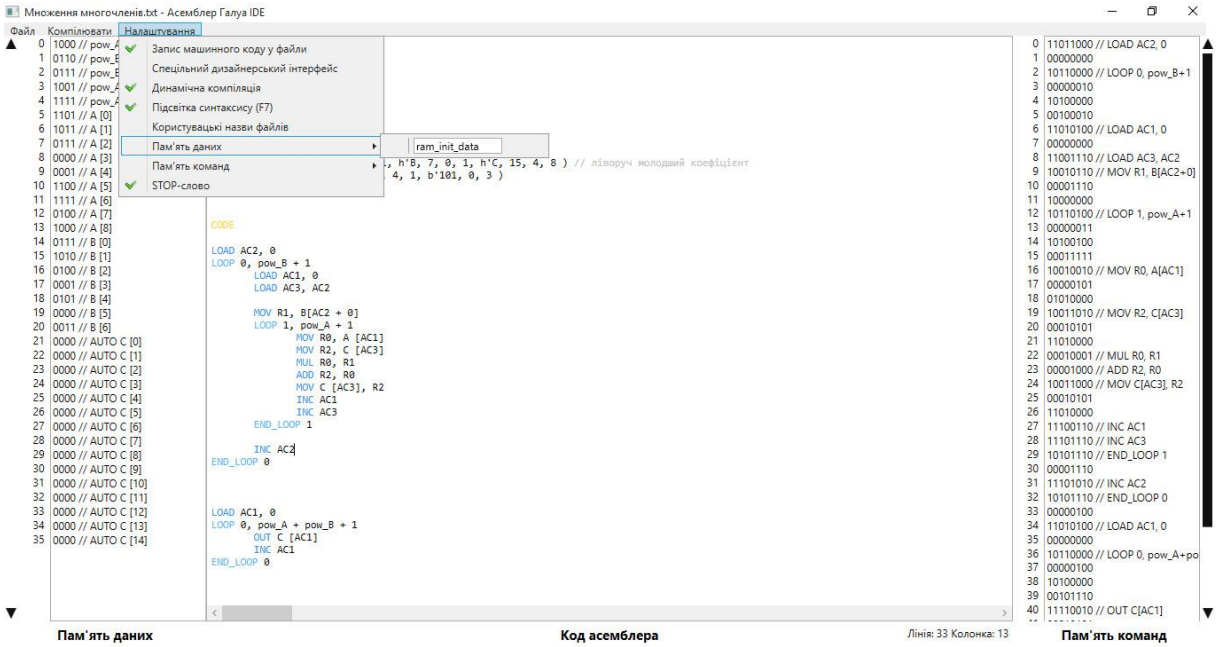


Рис. Є.6. Користувацькі назви файлів пам’яті команд та даних

Параметр *STOP*-слово відповідає за додавання в кінець машинного коду, що зберігається в пам’яті команд, службового слова, яке інтерпретується процесором як сигнал про кінець машинного коду програми.

ДОДАТОК Ж. Список повідомлень компілятора при трансляції тексту програми у машинний код

Відсутність директиви *GF*

При відсутності обов’язкової директиви *GF* у тексті програми, що вказує вид поля, компілятором формується відповідне інформаційне повідомлення (рис. Ж.1).

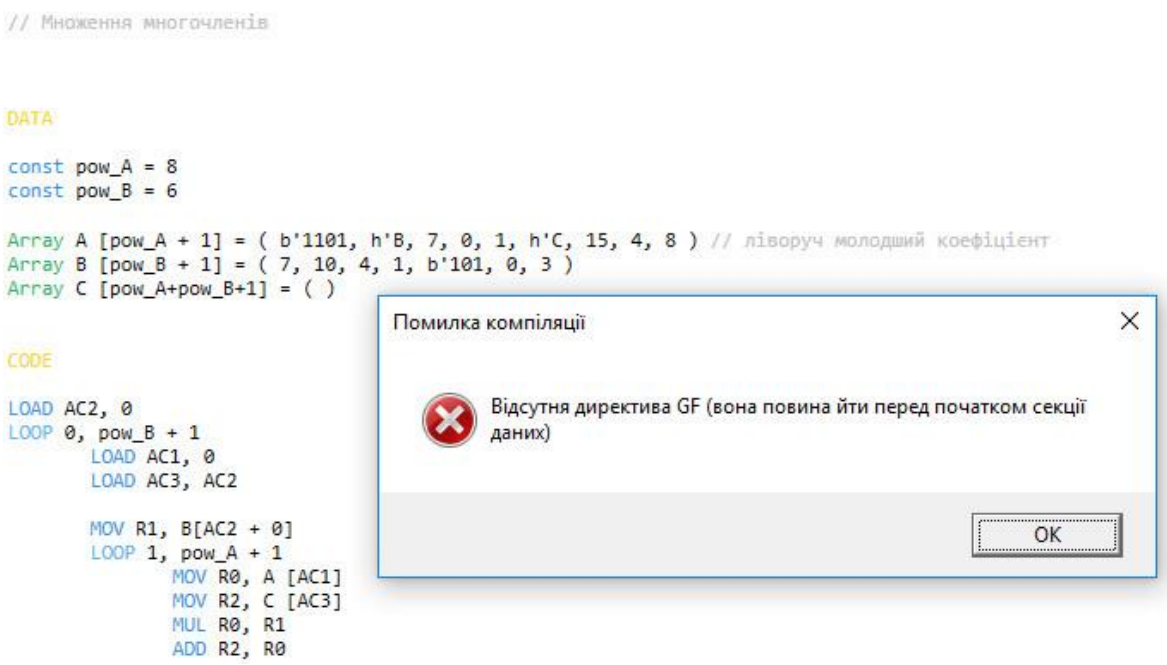


Рис. Ж.1. Відсутність директиви *GF*

Некоректне задання поля

Якщо вказана директива *#GF*(*p*), то розрядність слова пам'яті даних обчислюється як логарифм двійковий від *p* з округленням до найближчого більшого цілого числа. Число *p* задається за допомогою десяткової

константи. Наприклад #*GF*(13). Компілятором виконується перевірка чи є простим задане число *p*.

На рис. Ж.2 наведено фрагмент тексту програми, який передбачає роботу над елементами поля *GF*(*p*). Це задано першою директивою у програмі. У даній програмі програмістом було задане значення *p* = 25, яке не є простим числом, про що і сигналізує відповідне інформаційне повідомлення про помилку від компілятора.

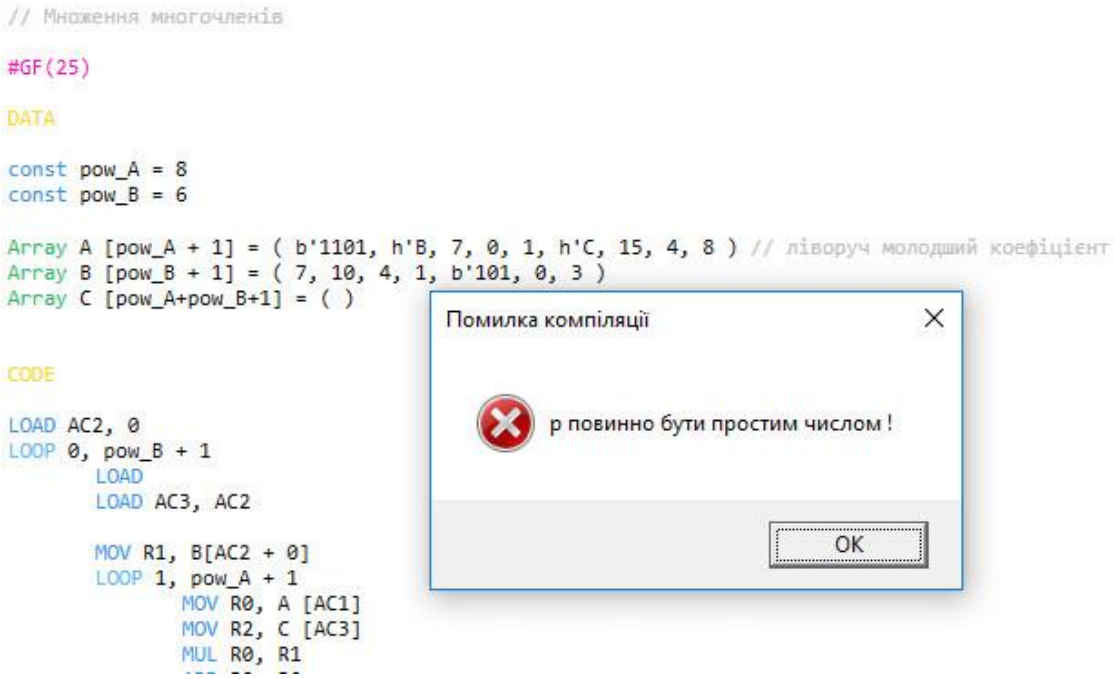


Рис. Ж.2. Некоректне задання поля

Відсутність секцій *DATA* та/або *CODE*

При відсутності обов’язкових секцій *DATA* та/або *CODE* у тексті програми, компілятором формується відповідне інформаційне повідомлення (рис. Ж.3).

```
// Множення многочленів

#GF(2^4)


const pow_A = 8
const pow_B = 6

Array A [pow_A + 1] = ( b'1101, h'B, 7, 0, 1, h'C, 15, 4, 8 ) // ліворуч молодший коефіцієнт
Array B [pow_B + 1] = ( 7, 10, 4, 1, b'101, 0, 3 )
Array C [pow_A+pow_B+1] = ( )


CODE

LOAD AC2, 0
LOOP 0, pow_B + 1
       LOAD AC1, 0
       LOAD AC3, AC2

       MOV R1, B[AC2 + 0]
       LOOP 1, pow_A + 1
              MOV R0, A [AC1]
              MOV R2, C [AC3]
```

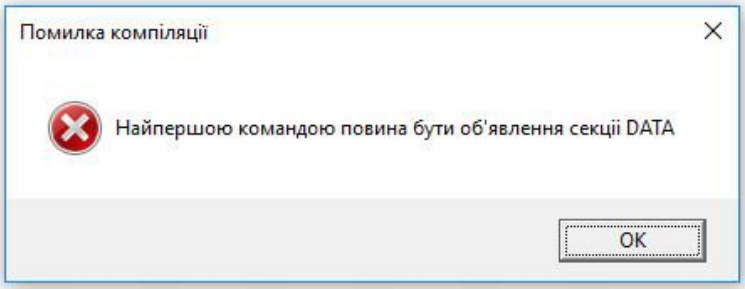


Рис. Ж.3. Відсутність секції *DATA*

Переповнення розрядної сітки

Компілятором контролюється переповнення розрядної сітки коли значення операндів перевищує величину $p-1$ для поля $GF(p)$ та величину $2^m - 1$ для поля $GF(2^m)$.

Наприклад, задане поле $GF(2^{13})$. Тоді значення даних мають належати діапазону $\left[0; 2^{13}-1=8191\right]$, а в тексті програми (рис. Ж.4) виконується присвоєння константі r числа 25000, що не входить до допустимого діапазону.

```
// Кодування даних

#GF(2^13)


DATA
const r = 25000
const pow_A = b'10011
//const pow_A = 19
const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = h'4
const null = 0

count_T = 2
```

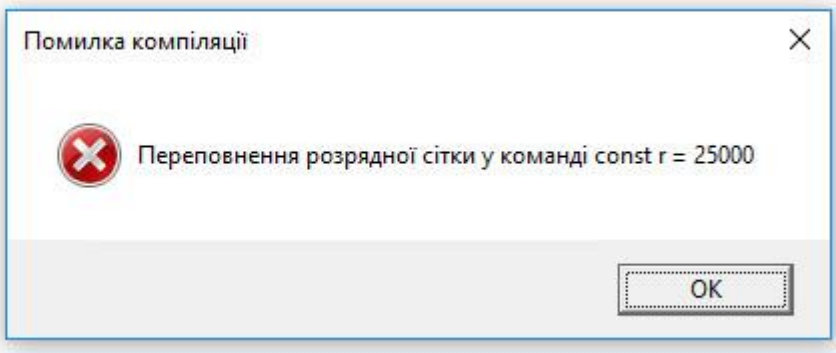


Рис. Ж.4. Переповнення розрядної сітки

Переповнення масиву

На початку секції *DATA* описана константа *pow_B*, якій присвоєне значення 6, далі описується масив *B*, з кількістю елементів *pow_B*, тобто масив, що має 6 елементів, але при ініціалізації задається 7 елементів, що викликає відповідну помилку (рис. Ж.5).

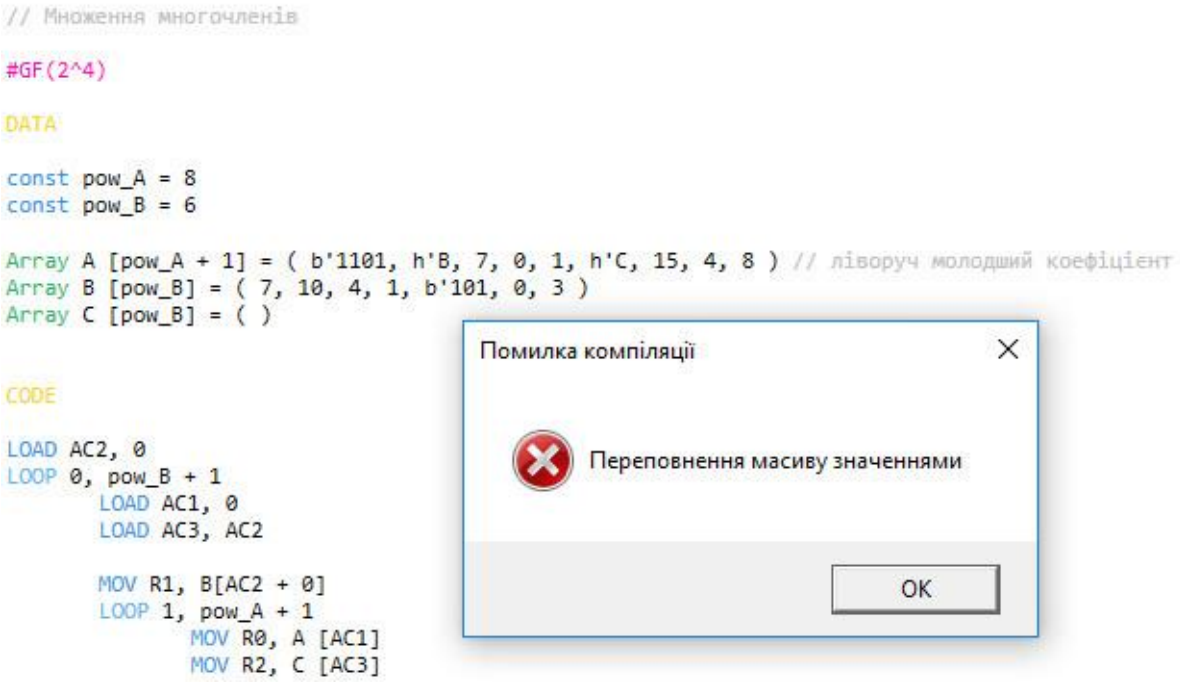


Рис. Ж.5. Переповнення масиву

Дублювання імен констант та змінних

У другому рядку тексту програми об’являється та ініціалізується масив *T*, далі в п’ятому рядку тексту програми знову виконується об’явленням масиву, що має ім’я *T* (рис. Ж.6). Така перевірка виконується також для констант та скалярних змінних.

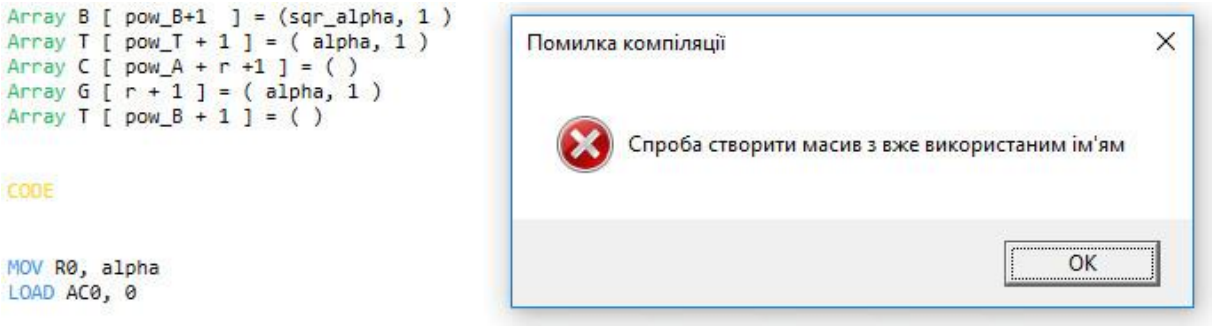


Рис. Ж.6. Дублювання імені масиву

Некоректний синтаксис команди

Цей тип помилки виникає коли команда з операндами, яка задана в тексті програми, не відповідає потрібному шаблону.

Розглянемо кілька прикладів.

Наприклад, в якості операнду “джерела даних” не може виступати арифметичний вираз, в якому аргументами є константи та змінні. Це контролюється компілятором, і на рис. Ж.7 наведено приклад, де в команді *MOV* використовується в якості джерела даних добуток константи *null* на змінну *count_T*.

На рис. Ж.8 наведено приклад, де в команді *LOAD* в якості першого операнду береться неіснуючий лічильник циклу *AC*4 (в системі дозволеними є лічильники циклу *AC*0, *AC*1, *AC*2 та *AC*3).

Також цей тип помилки буде спрацьовувати, якщо за допомогою команди *MOV* пробувати записати дані в константу. Зрозуміло, що така операція є недозволеною. Окрім наведених прикладів, в усіх інших випадках, коли команда з операндами не відповідає заданому шаблону, компілятором буде видане таке інформаційне повідомлення.

```
// Кодування даних

#GF(2^13)


DATA
const r = 25
const pow_A = b'10011
//const pow_A = 19
const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = h'4
const null = 0

count_T = 2

//Array A [ pow_A ] = (  5, h'F, b'1, 0, b'111, h'3 )
Array A [ pow_A + 1 ] = (  5, 27, 1013, 13, 71, 89, 1001, 579, 789, 234, 567, 5, 7, 2, 0, 0, 0, 1, 5, 1017 )
//Array A [ pow_A + 1 ] = ( 5, 15, 1, 0, 7, 3 )
Array B [ pow_B+1  ] = (sqr_alpha, 1 )
Array C [ pow_A + r +1 ] = ( )
Array G [ r + 1 ] = ( alpha, 1 )
Array T [ pow_T + 1 ] = ( alpha, 1 )


CODE


MOV R0, alpha
LOAD AC0, 0

// генерування твірного многочлена
LOOP 0, r - 1
       MOV R2, null*count_T
       LOAD AC1, 0
```

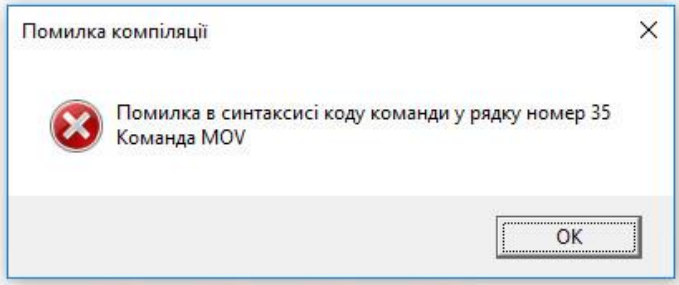


Рис. Ж.7. Некоректний синтаксис команди *MOV*

```
// Кодування даних

#GF(2^13)


DATA
const r = 25
const pow_A = b'10011
//const pow_A = 19
const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = h'4
const null = 0

count_T = 2

//Array A [ pow_A ] = (  5, h'F, b'1, 0, b'111, h'3 )
Array A [ pow_A + 1 ] = (  5, 27, 1013,                                                    , 1017 )
//Array A [ pow_A + 1 ] = ( 5, 15, 1, 0
Array B [ pow_B+1  ] = (sqr_alpha, 1 )
Array C [ pow_A + r +1 ] = ( )
Array G [ r + 1 ] = ( alpha, 1 )
Array T [ pow_T + 1 ] = ( alpha, 1 )


CODE


MOV R0, alpha
LOAD AC0, 0

// генерування твірного многочлена
LOOP 0, r - 1
       MOV R2, null
       LOAD AC4, 0
```

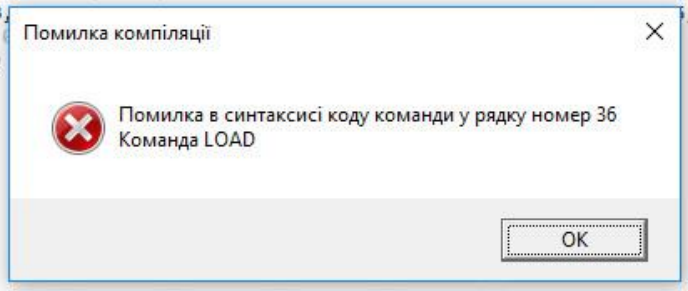


Рис. Ж.8. Некоректний синтаксис команди *LOAD*

Недопустима команда для заданого поля

На рис. Ж.9 наведено приклад використання команди *CPD* у тексті програми для поля *GF*(*p*). Оскільки ця команда, як і команда *CDP*, є командою для роботи з елементами поля *GF*(2^m), компілятор видає відповідне інформаційне повідомлення про помилку у тексті програми.

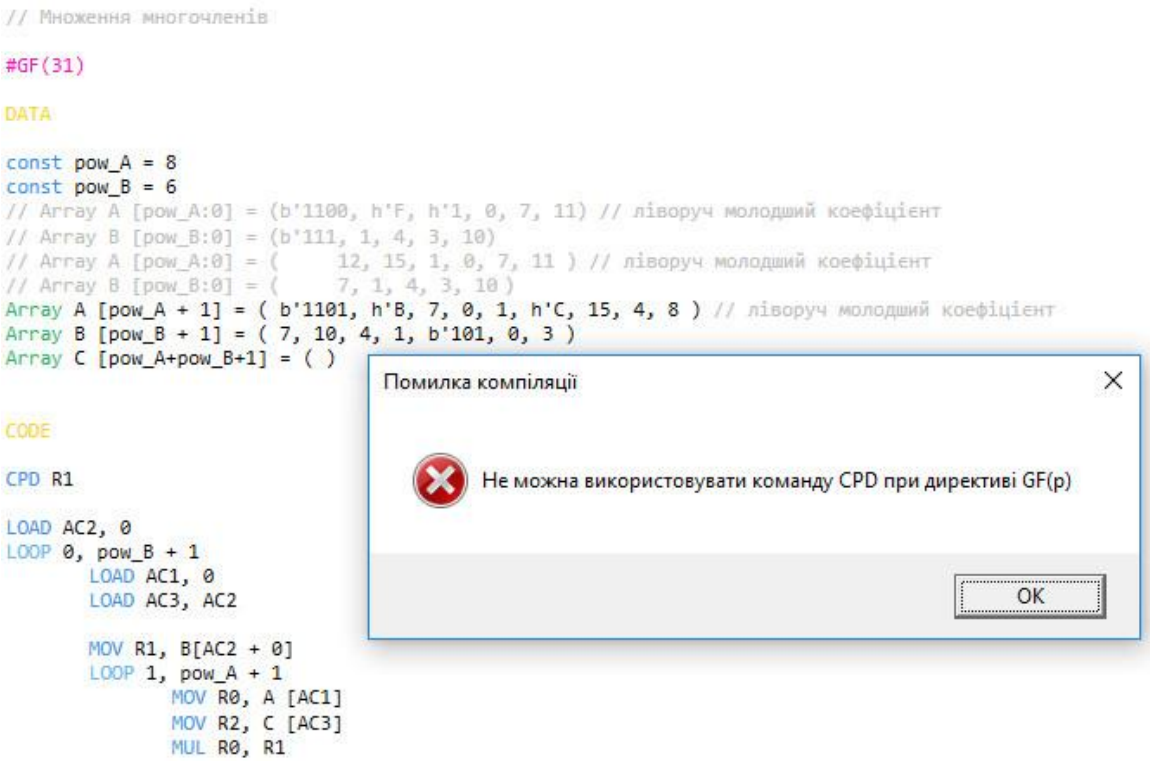


Рис. Ж.9. Недопустима команда для заданого поля *GF*(*p*)

На рис. Ж.10 наведено приклад використання команди *SUB* у тексті програми для поля *GF*(2^m). Оскільки ця команда, як і команда *INVA*, є командою для роботи з елементами поля *GF*(*p*), компілятор видає відповідне інформаційне повідомлення про помилку в тексті програми.

```
// Множення многочленів

#GF(2^4)

DATA

const pow_A = 8
const pow_B = 6

Array A [pow_A + 1] = ( b'1101, h'B, 7, 0, 1, h'C, 15, 4, 8 ) // ліворуч молодший коефіцієнт
Array B [pow_B + 1] = ( 7, 10, 4, 1, b'101, 0, 3 )
Array C [pow_A+pow_B+1] = ( )


CODE

SUB R1, R2


LOAD AC2, 0
LOOP 0, pow_B + 1
        LOAD AC1, 0
        LOAD AC3, AC2

        MOV R1, B[AC2 + 0]
```

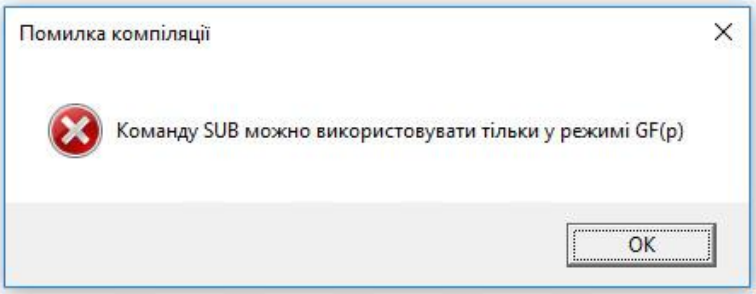


Рис. Ж.10. Недопустима команда для заданого поля $GF(2^m)$

Звертання до неіснуючої константи або змінної

У розділі *DATA* наявна константа *null*, а першою командою в циклі йде команда запису даних з контанти/змінної *null*_1 у регістр *R*2. Оскільки такий ідентифікатор не об’явлено в секції *DATA*, то компілятор формує відповідну помилку (рис. Ж.11).

```
// Кодування даних

#GF(2^13)


DATA
const r = 25
const pow_A = b'10011
const alpha = 2
const sqr_alpha = h'4
const null = 0
count_T = 2
Array A [ pow_A + 1 ] = (  5, 27, 1013, 13, 71, 89, 1001, 579, 789, 234, 567, 5, 7, 2, 0, 0, 0, 1, 5, 1017 )

CODE

MOV R0, alpha
LOAD AC0, 0

// генерування твірного многочлена
LOOP 0, r - 1
        MOV R2, null_1
        LOAD AC1, 0

        // переписуємо елементи маси
```

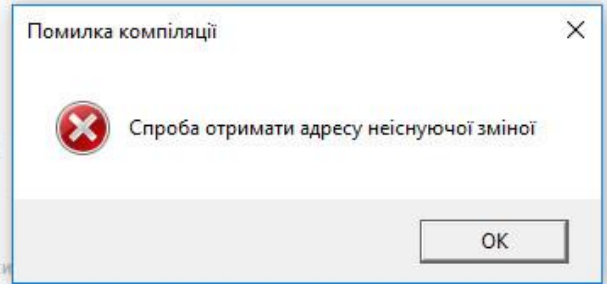


Рис. Ж.11. Звертання до необ’явленої змінної

Незакритий цикл

На рис. Ж.12 наведено фрагмент тексту програми, який містить початок циклу з ідентифікатором 0, але цей цикл не закривається.

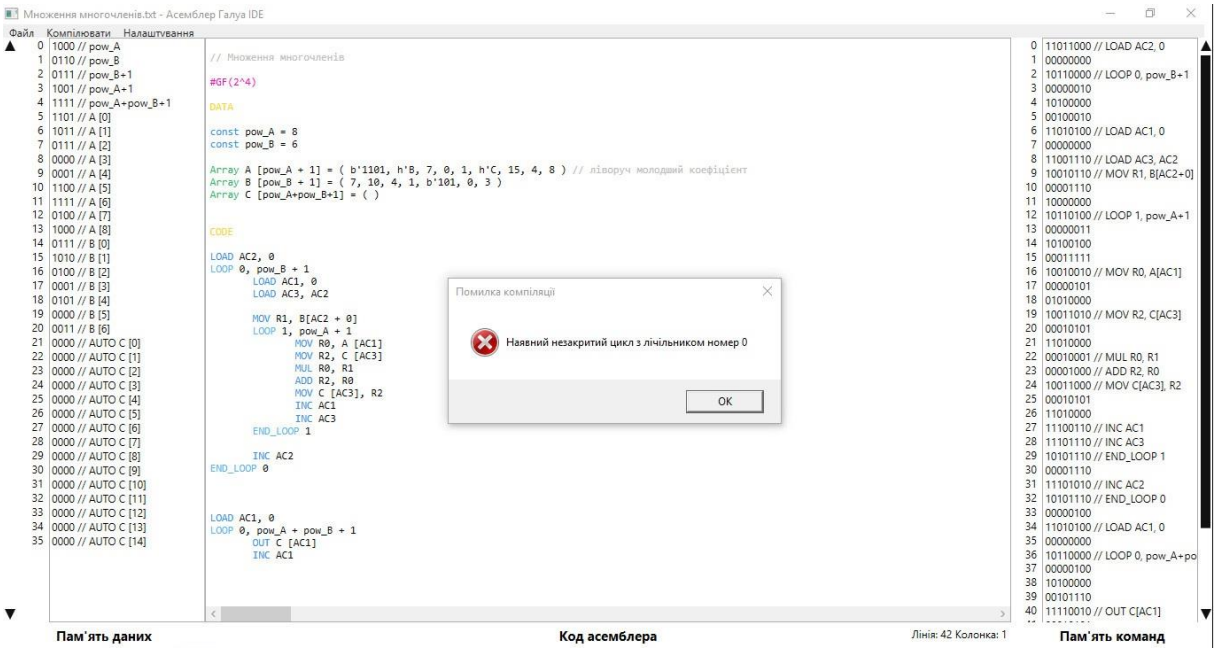


Рис. Ж.12. Незакритий цикл

Якщо було не закрито цикл з певним ідентифікатором і далі в тексті програми починається цикл з таким самим ідентифікатором, то це свідчить про помилку, яка наведена на рис. Ж.13.

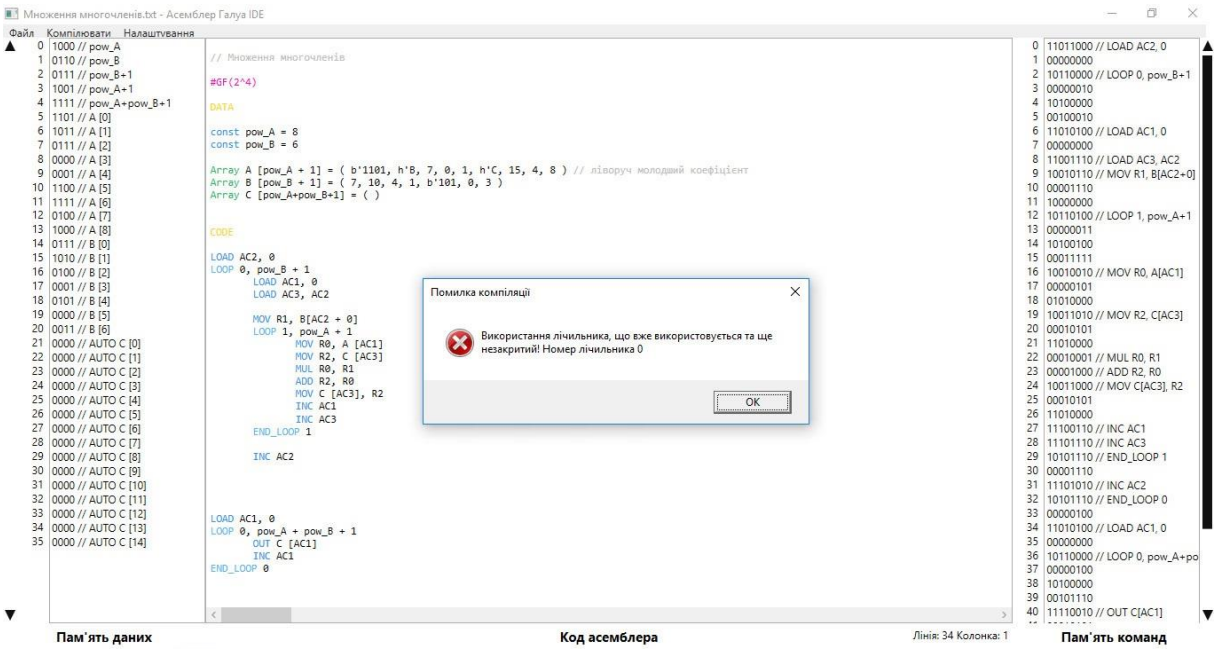


Рис. Ж.13. Незакритий цикл та повторне використання

Невідповідність ідентифікаторів початку і кінця циклу

На рис. Ж.14 наведено приклад ситуації, коли компілятор перевіряє відповідність початку та кінця циклу. Для даного прикладу компілятором було знайдено команду кінця циклу, але не знайдено відповідну команду початку циклу про що й поінформовано програміста.

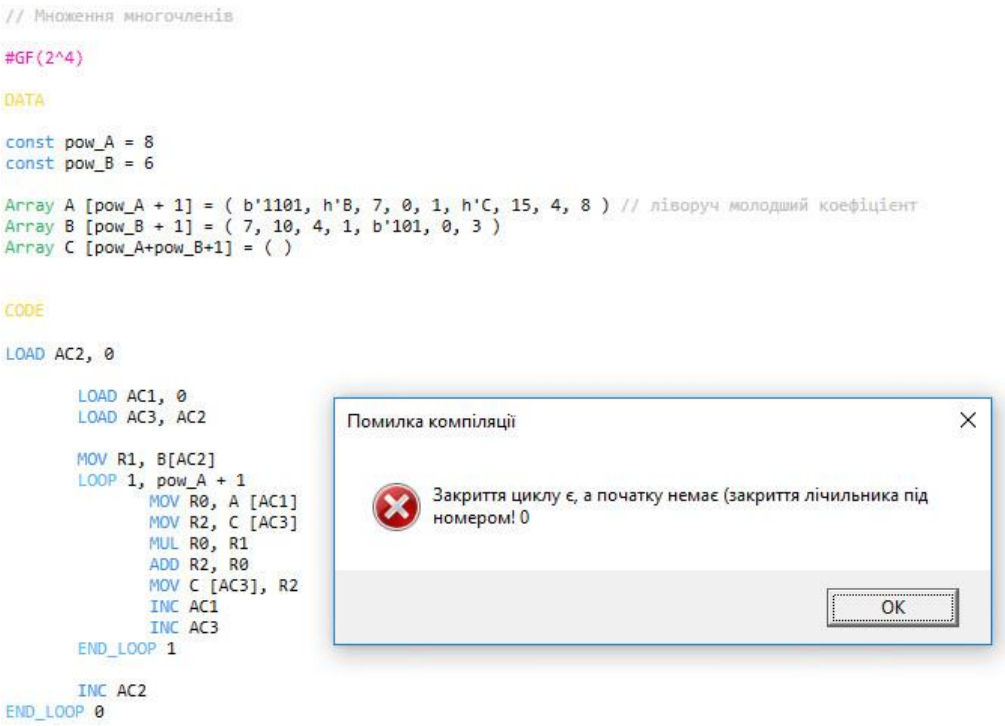


Рис. Ж.14. Не задано початок циклу

Окрім цього, компілятором перевіряється рівність ідентифікаторів початку та кінця циклу, наприклад, на рис. Ж.15 наведено текст програми, де початок циклу має ідентифікатор 0, а кінець циклу 1. Компілятором було успішно розпізнано такий тип помилки та поінформовано про це програміста.

```
// Множення многочленів

#GF(2^4)

DATA

const pow_A = 8
const pow_B = 6

Array A [pow_A + 1] = ( b'1101, h'B, 7, 0, 1, h'C, 15, 4, 8 ) // ліворуч молодший коефіцієнт
Array B [pow_B + 1] = ( 7, 10, 4, 1, b'101, 0, 3 )
Array C [pow_A+pow_B+1] = ( )


CODE

LOAD AC2, 0
LOOP 0, pow_B + 1
        LOAD AC1, 0
        LOAD AC3, AC2

        MOV R1, B[AC2 + 0]
        LOOP 1, pow_A + 1
                MOV R0, A [AC1]
                MOV R2, C [AC3]
                MUL R0, R1
                ADD R2, R0
                MOV C [AC3], R2
                INC AC1
                INC AC3
        END_LOOP 1

        INC AC2
END LOOP 1
```

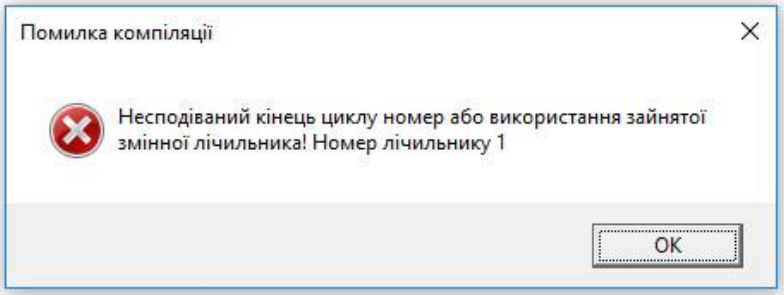


Рис. Ж.15. Не відповідність ідентифікаторів початку і кінця циклу

Недопустиме значення зміщення

У тексті програми на рис. Ж.16 при роботі з масивом використовується значення зміщення 25, хоча допустимі значення зміщення належать проміжку $[-15;15]$, про що компілятор інформує програміста.

```
// Множення многочленів

#GF(2^4)

DATA

const pow_A = 8
const pow_B = 6

Array A [pow_A + 1] = ( b'1101, h'B, 7, 0, 1, h'C, 15, 4, 8 ) // ліворуч молодший коефіцієнт
Array B [pow_B + 1] = ( 7, 10, 4, 1, b'101, 0, 3 )
Array C [pow_A+pow_B+1] = ( )

CODE

LOAD AC2, 0
LOOP 0, pow_B + 1
       LOAD AC1, 0
       LOAD AC3, AC2

       MOV R1, B[AC2 + 25]
       LOOP 1, pow_A + 1
              MOV R0, A [AC1]
              MOV R2, C [AC3]
              MUL R0, R1
              ADD R2, R0
              MOV C [AC3], R2
              INC AC1
              INC AC3
       END_LOOP 1

       INC AC2
END_LOOP 0
```

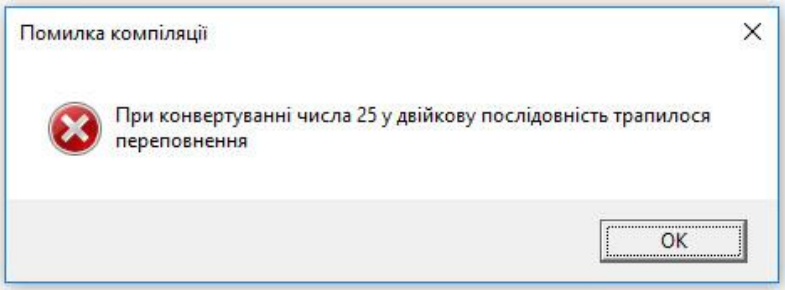


Рис. Ж.16. Недопустиме значення зміщення

Дублювання імені мітки

Всі мітки у тексті програми повинні мати унікальне ім’я, тому компілятором контролюється наявність в програмному коді міток з однаковими іменами. На рис. Ж.17 наведено фрагмент тексту програми, де наявні дві мітки з однаковим іменем *Label*_1.

```
CODE

Label_1 :
MOV R0, alpha

Label_1:
LOAD AC0, 0
```

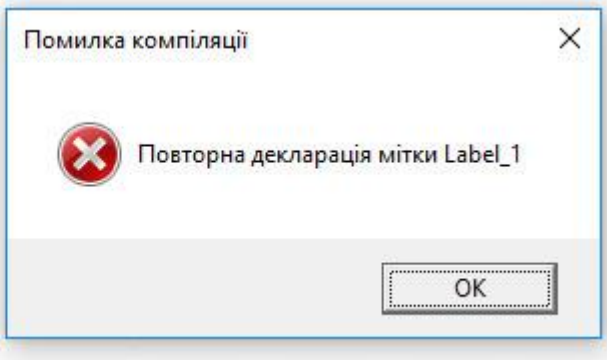


Рис. Ж.17. Мітки з однаковими іменами

Звертання до неіснуючої мітки

На рис. Ж.18 наведено фрагмент тексті програми, де в команді *JMP* помилково вказано перехід за міткою *Label_* замість *Label*_1. Компілятор в цьому випадку видає відповідне інформаційне повідомлення про те, що такої мітки не існує.

```
Array B [ pow_B+1  ] = (sqr_alpha, 1 )
Array C [ pow_A + r +1 ] = ( )
Array G [ r + 1 ] = ( alpha, 1 )
Array T [ pow_T + 1 ] = ( alpha, 1 )


CODE

Label_1 :
MOV R0, alpha
LOAD AC0, 0

JMP Label_
```

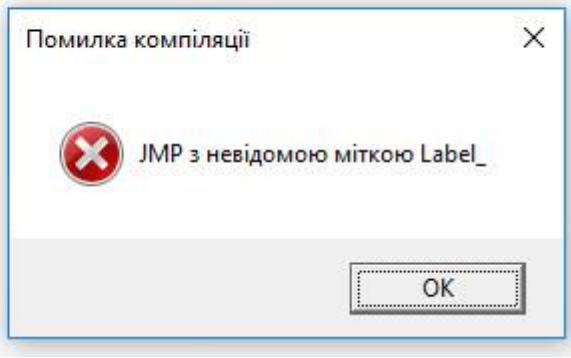


Рис. Ж.18. Неіснуюча мітка

Незакритий багаторядковий коментар

Така помилка з’являється, коли у тексті програми присутній незакритий багаторядковий коментар, а саме – присутня комбінація символів “/*” та немає комбінації символів “*/”, що їй відповідає (рис. Ж.19).

```
// Кодування даних
#GF(2^13)

DATA
const r = 25
const pow_A = b'10011

const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = h'4
const null = 0
/*
count_T = 2
```

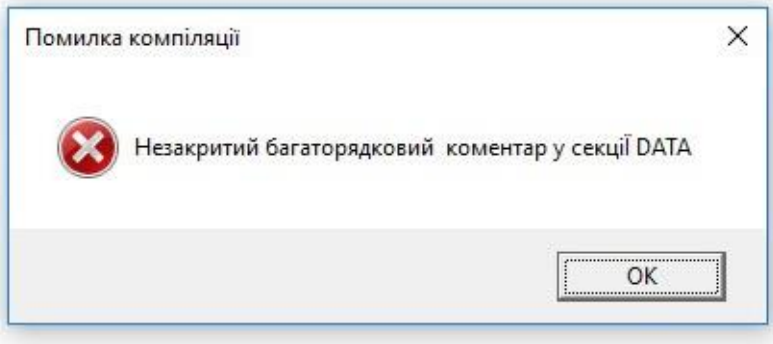


Рис. Ж.19. Незакритий багаторядковий коментар

ДОДАТОК К. Приклади програм мовою Асемблера процесора Галуа

1. Текст програми для множення многочленів

```
#GF(2^4)

DATA

const pow_A = 8
const pow_B = 6
Array A [pow_A + 1] = ( b'1101, h'B, 7, 0, 1, h'C, 15, 4, 8 ) // ліворуч молодший коефіцієнт
Array B [pow_B + 1] = ( 7, 10, 4, 1, b'101, 0, 3 )
Array C [pow_A+pow_B+1] = ( )

CODE

LOAD AC2, 0
LOOP 0, pow_B + 1
       LOAD AC1, 0

       LOAD AC3, AC2

       MOV R1, B[AC2 + 0]
       LOOP 1, pow_A + 1
              MOV R0, A [AC1]
              MOV R2, C [AC3]
              MUL R0, R1
              ADD R2, R0
              MOV C [AC3], R2
              INC AC1
              INC AC3
       END_LOOP 1

       INC AC2
END_LOOP 0



LOAD AC1, 0
LOOP 0, pow_A + pow_B + 1
       OUT C [AC1]
       INC AC1
END_LOOP 0
```

Пам’ять даних

```
 0  1000 // pow_A
 1  0110 // pow_B
 2  0111 // pow_B+1
 3  1001 // pow_A+1
 4  1111 // pow_A+pow_B+1
 5  1101 // A [0]
 6  1011 // A [1]
 7  0111 // A [2]
 8  0000 // A [3]
 9  0001 // A [4]
10  1100 // A [5]
11  1111 // A [6]
12  0100 // A [7]
13  1000 // A [8]
14  0111 // B [0]
15  1010 // B [1]
16  0100 // B [2]
17  0001 // B [3]
18  0101 // B [4]
19  0000 // B [5]
20  0011 // B [6]
21  0000 // AUTO C [0]
22  0000 // AUTO C [1]
23  0000 // AUTO C [2]
24  0000 // AUTO C [3]
25  0000 // AUTO C [4]
26  0000 // AUTO C [5]
27  0000 // AUTO C [6]
28  0000 // AUTO C [7]
29  0000 // AUTO C [8]
30  0000 // AUTO C [9]
31  0000 // AUTO C [10]
32  0000 // AUTO C [11]
33  0000 // AUTO C [12]
34  0000 // AUTO C [13]
35  0000 // AUTO C [14]
```

Пам’ять команд

```
 0  11011000 // LOAD AC2, 0
 1  00000000
 2  10110000 // LOOP 0, pow_B+1
 3  00000010
 4  10100000
 5  00100010
 6  11010100 // LOAD AC1, 0
 7  00000000
 8  11001110 // LOAD AC3, AC2
 9  10010110 // MOV R1, B[AC2+0]
10  00001110
11  10000000
12  10110100 // LOOP 1, pow_A+1
13  00000011
14  10100100
15  00011111
16  10010010 // MOV R0, A[AC1]
17  00000101
18  01010000
19  10011010 // MOV R2, C[AC3]
20  00010101
21  11010000
22  00010001 // MUL R0, R1
23  00001000 // ADD R2, R0
24  10011000 // MOV C[AC3], R2
25  00010101
26  11010000
27  11100110 // INC AC1
28  11101110 // INC AC3
29  10101110 // END_LOOP 1
30  00001110
31  11101010 // INC AC2
32  10101110 // END_LOOP 0
33  00000100
34  11010100 // LOAD AC1, 0
35  00000000
36  10110000 // LOOP 0, pow_A+pow_B+1
37  00000100
38  10100000
39  00101110
40  11110010 // OUT C[AC1]
41  00010101
42  01010000
43  11100110 // INC AC1
44  10101110 // END_LOOP 0
45  00100110
46  01001111 // STOP-слово
```

2. Текст програми для генерування твірного многочлена коду Ріда-Соломона

```
#GF(2^10)

DATA
const r = 14
const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = 4
const null = 0
count_T = 2
Array G [r + 1] = ( alpha, 1 )
Array T [pow_T + 1] = ( alpha, 1 )
Array B [pow_B + 1] = ( sqr_alpha, 1 )

CODE

MOV R0, alpha
LOAD AC0, 0
LOOP 0, r-1
       MOV R2, null

       LOAD AC1, 0
       LOOP 1, count_T
              MOV R1, G[AC1]
              MOV T [AC1], R1
              MOV G [AC1], R2
              INC AC1
       END_LOOP 1

       LOAD AC2, 0
       LOOP 1, pow_B + 1
              LOAD AC1, 0
              LOAD AC3, AC2
              MOV R2, B[AC2]

              LOOP 2, count_T
                     MOV R1, T[AC1]
                     MOV R3, G[AC3]
                     MUL R1, R2
                     ADD R3, R1
                     MOV G[AC3], R3
                     INC AC1
                     INC AC3
              END_LOOP 2

              INC AC2
       END_LOOP 1

       MOV R1, B[AC0]
       MUL R1, R0

       MOV B[AC0], R1
       MOV R1, count_T

       INC R1
       MOV count_T, R1
END_LOOP 0

LOAD AC1, 0
LOOP 0, r+1
       OUT G [AC1]
       INC AC1
END_LOOP 0
```

Пам’ять даних

```
 0  0000001110 // r
 1  0000000001 // pow_B
 2  0000001110 // pow_T
 3  0000000010 // alpha
 4  0000000100 // sqr_alpha
 5  0000000000 // null
 6  0000001101 // r-1
 7  0000000010 // pow_B+1
 8  0000001111 // r+1
 9  0000000010 // count_T
10  0000000010 // G [0]
11  0000000001 // G [1]
12  0000000000 // AUTO G [2]
13  0000000000 // AUTO G [3]
14  0000000000 // AUTO G [4]
15  0000000000 // AUTO G [5]
16  0000000000 // AUTO G [6]
17  0000000000 // AUTO G [7]
18  0000000000 // AUTO G [8]
19  0000000000 // AUTO G [9]
20  0000000000 // AUTO G [10]
21  0000000000 // AUTO G [11]
22  0000000000 // AUTO G [12]
23  0000000000 // AUTO G [13]
24  0000000000 // AUTO G [14]
25  0000000010 // T [0]
26  0000000001 // T [1]
27  0000000000 // AUTO T [2]
28  0000000000 // AUTO T [3]
29  0000000000 // AUTO T [4]
30  0000000000 // AUTO T [5]
31  0000000000 // AUTO T [6]
32  0000000000 // AUTO T [7]
33  0000000000 // AUTO T [8]
34  0000000000 // AUTO T [9]
35  0000000000 // AUTO T [10]
36  0000000000 // AUTO T [11]
37  0000000000 // AUTO T [12]
38  0000000000 // AUTO T [13]
39  0000000000 // AUTO T [14]
40  0000000100 // B [0]
41  0000000001 // B [1]
```

Пам’ять команд

```
 0  10000010 // MOV R0, alpha
 1  00000011
 2  11010000 // LOAD AC0, 0
 3  00000000
 4  10110000 // LOOP 0, r-1
 5  00000110
 6  10100000
 7  01001100
 8  10001010 // MOV R2, null
 9  00000101
10  11010100 // LOAD AC1, 0
11  00000000
12  10110100 // LOOP 1, count_T
13  00001001
14  10100100
15  00011100
16  10010110 // MOV R1, G[AC1]
17  00001010
18  01010000
19  10010100 // MOV T[AC1], R1
20  00011001
21  01010000
22  10011000 // MOV G[AC1], R2
23  00001010
24  01010000
25  11100110 // INC AC1
26  10101110 // END_LOOP 1
27  00001110
28  11011000 // LOAD AC2, 0
29  00000000
30  10110100 // LOOP 1, pow_B+1
31  00000111
32  10100100
33  00111110
34  11010100 // LOAD AC1, 0
35  00000000
36  11001110 // LOAD AC3, AC2
37  10011010 // MOV R2, B[AC2]
38  00101000
39  10010000
40  10111000 // LOOP 2, count_T
41  00001001
42  10101000
43  00111011
44  10010110 // MOV R1, T[AC1]
45  00011001
46  01010000
47  10011110 // MOV R3, G[AC3]
48  00001010
49  11010000
50  00010110 // MUL R1, R2
51  00001101 // ADD R3, R1
52  10011100 // MOV G[AC3], R3
53  00001010
54  11010000
55  11100110 // INC AC1
56  11101110 // INC AC3
57  10101110 // END_LOOP 2
58  00101010
59  11101010 // INC AC2
60  10101110 // END_LOOP 1
61  00100000
62  10010110 // MOV R1, B[AC0]
63  00101000
64  00010000
65  00010100 // MUL R1, R0
66  10010100 // MOV B[AC0], R1
67  00101000
68  00010000
69  10000110 // MOV R1, count_T
70  00001001
71  11100100 // INC R1
72  10000100 // MOV count_T, R1
73  00001001
74  10101110 // END_LOOP 0
75  00000110
76  11010100 // LOAD AC1, 0
77  00000000
78  10110000 // LOOP 0, r+1
79  00001000
80  10100000
81  01011000
82  11110010 // OUT G[AC1]
83  00001010
84  01010000
85  11100110 // INC AC1
86  10101110 // END_LOOP 0
87  01010000
88  01001111 // STOP-слово
```

3. Текст програми для кодування даних кодом Ріда-Соломона

```
#GF(2^13)

DATA
const r = 25
const pow_A = b'10011
const pow_B = 1
const pow_T = r
const alpha = 2
const sqr_alpha = h'4
const null = 0

count_T = 2

Array A [ pow_A + 1 ] = (  5, 27, 1013, 13, 71, 89, 1001, 579, 789, 234, 567, 5, 7, 2, 0,
0, 0, 1, 5, 1017 )
Array B [ pow_B+1  ] = (sqr_alpha, 1 )
Array C [ pow_A + r +1 ] = ( )
Array G [ r + 1 ] = ( alpha, 1 )
Array T [ pow_T + 1 ] = ( alpha, 1 )


CODE

MOV R0, alpha
LOAD AC0, 0

// генерування твірного многочлена
LOOP 0, r - 1
       MOV R2, null
       LOAD AC1, 0

       // переписуємо елементи масиву G у масив T, а масив G заповнюємо нулями
       LOOP 1, count_T
              MOV R1, G[AC1]
              MOV T [AC1], R1
              MOV G [AC1], R2
              INC AC1
       END_LOOP 1

       // множимо поточний твірний многочлен на чергову дужку (x - alpha^i)
       LOAD AC2, 0
       LOOP 1, pow_B + 1
              LOAD AC1, 0
              LOAD AC3, AC2
              MOV R2, B[AC2]

              LOOP 2, count_T
                     MOV R1, T[AC1]
                     MOV R3, G[AC3]
                     MUL R1, R2
                     ADD R3, R1
                     MOV G[AC3], R3
                     INC AC1
                     INC AC3
              END_LOOP 2

              INC AC2
       END_LOOP 1
```

```
        // множимо старший коефіцієнт масиву B на alpha, тим самим збільшуючи показник
степеня alpha на 1
        MOV R1, B[AC0]
        MUL R1, R0
        MOV B[AC0], R1

        // збільшуємо на 1 кількість задіяних елементів масиву T
        MOV R1, count_T
        INC R1
        MOV count_T, R1
END_LOOP 0

// множення твірного многочлена G на многочлен C, який необхідно закодувати
LOAD AC2, h'0
LOOP 0, pow_A+1
        LOAD AC1, b'0
        LOAD AC3, AC2
        MOV R1, A[AC2 ]

        LOOP 1, count_T
                MOV R0, G[AC1 + 0]
                MOV R2, C[AC3]
                MUL R0, R1
                ADD R2, R0
                MOV C[AC3],R2
                INC AC1
                INC AC3
        END_LOOP 1

        INC AC2
END_LOOP 0

LOAD AC0, 0
LOOP 0, pow_A + r + 1
        OUT C[AC0]
        INC AC0
END_LOOP 0
```

Пам’ять даних

```
 0  0000000011001 // r
 1  0000000010011 // pow_A
 2  0000000000001 // pow_B
 3  0000000011001 // pow_T
 4  0000000000010 // alpha
 5  0000000000100 // sqr_alpha
 6  0000000000000 // null
 7  0000000011000 // r-1
 8  0000000000010 // pow_B+1
 9  0000000010100 // pow_A+1
10  0000000101101 // pow_A+r+1
11  0000000000010 // count_T
12  0000000000101 // A [0]
13  0000000011011 // A [1]
14  0001111110101 // A [2]
15  0000000001101 // A [3]
16  0000001000111 // A [4]
17  0000001011001 // A [5]
18  0001111101001 // A [6]
19  0001001000011 // A [7]
20  0001100010101 // A [8]
21  0000011101010 // A [9]
22  0001000110111 // A [10]
23  0000000000101 // A [11]
24  0000000000111 // A [12]
25  0000000000010 // A [13]
26  0000000000000 // A [14]
27  0000000000000 // A [15]
28  0000000000000 // A [16]
29  0000000000001 // A [17]
30  0000000000101 // A [18]
31  0001111111001 // A [19]
32  0000000000100 // B [0]
33  0000000000001 // B [1]
34  0000000000000 // AUTO C [0]
35  0000000000000 // AUTO C [1]
36  0000000000000 // AUTO C [2]
37  0000000000000 // AUTO C [3]
38  0000000000000 // AUTO C [4]
39  0000000000000 // AUTO C [5]
40  0000000000000 // AUTO C [6]
41  0000000000000 // AUTO C [7]
42  0000000000000 // AUTO C [8]
43  0000000000000 // AUTO C [9]
44  0000000000000 // AUTO C [10]
45  0000000000000 // AUTO C [11]
46  0000000000000 // AUTO C [12]
47  0000000000000 // AUTO C [13]
48  0000000000000 // AUTO C [14]
49  0000000000000 // AUTO C [15]
50  0000000000000 // AUTO C [16]
51  0000000000000 // AUTO C [17]
52  0000000000000 // AUTO C [18]
53  0000000000000 // AUTO C [19]
54  0000000000000 // AUTO C [20]
55  0000000000000 // AUTO C [21]
56  0000000000000 // AUTO C [22]
57  0000000000000 // AUTO C [23]
58  0000000000000 // AUTO C [24]
59  0000000000000 // AUTO C [25]
60  0000000000000 // AUTO C [26]
61  0000000000000 // AUTO C [27]
62  0000000000000 // AUTO C [28]
63  0000000000000 // AUTO C [29]
64  0000000000000 // AUTO C [30]
65  0000000000000 // AUTO C [31]
66  0000000000000 // AUTO C [32]
67  0000000000000 // AUTO C [33]
68  0000000000000 // AUTO C [34]
69  0000000000000 // AUTO C [35]
70  0000000000000 // AUTO C [36]
71  0000000000000 // AUTO C [37]
72  0000000000000 // AUTO C [38]
73  0000000000000 // AUTO C [39]
74  0000000000000 // AUTO C [40]
75  0000000000000 // AUTO C [41]
76  0000000000000 // AUTO C [42]
77  0000000000000 // AUTO C [43]
78  0000000000000 // AUTO C [44]
79  0000000000010 // G [0]
80  0000000000001 // G [1]
81  0000000000000 // AUTO G [2]
82  0000000000000 // AUTO G [3]
83  0000000000000 // AUTO G [4]
84  0000000000000 // AUTO G [5]
85  0000000000000 // AUTO G [6]
86  0000000000000 // AUTO G [7]
87  0000000000000 // AUTO G [8]
88  0000000000000 // AUTO G [9]
89  0000000000000 // AUTO G [10]
90  0000000000000 // AUTO G [11]
91  0000000000000 // AUTO G [12]
92  0000000000000 // AUTO G [13]
93  0000000000000 // AUTO G [14]
94  0000000000000 // AUTO G [15]
95  0000000000000 // AUTO G [16]
96  0000000000000 // AUTO G [17]
97  0000000000000 // AUTO G [18]
98  0000000000000 // AUTO G [19]
99  0000000000000 // AUTO G [20]
100 0000000000000 // AUTO G [21]
101 0000000000000 // AUTO G [22]
102 0000000000000 // AUTO G [23]
103 0000000000000 // AUTO G [24]
104 0000000000000 // AUTO G [25]
105 0000000000010 // T [0]
106 0000000000001 // T [1]
107 0000000000000 // AUTO T [2]
108 0000000000000 // AUTO T [3]
109 0000000000000 // AUTO T [4]
110 0000000000000 // AUTO T [5]
111 0000000000000 // AUTO T [6]
112 0000000000000 // AUTO T [7]
113 0000000000000 // AUTO T [8]
114 0000000000000 // AUTO T [9]
115 0000000000000 // AUTO T [10]
116 0000000000000 // AUTO T [11]
117 0000000000000 // AUTO T [12]
118 0000000000000 // AUTO T [13]
119 0000000000000 // AUTO T [14]
120 0000000000000 // AUTO T [15]
121 0000000000000 // AUTO T [16]
122 0000000000000 // AUTO T [17]
123 0000000000000 // AUTO T [18]
124 0000000000000 // AUTO T [19]
125 0000000000000 // AUTO T [20]
126 0000000000000 // AUTO T [21]
127 0000000000000 // AUTO T [22]
128 0000000000000 // AUTO T [23]
129 0000000000000 // AUTO T [24]
130 0000000000000 // AUTO T [25]
```

Пам’ять команд

```
 0  10000010 // MOV R0, alpha
 1  00000100
 2  11010000 // LOAD AC0, 0
 3  00000000
 4  10110000 // LOOP 0, r-1
 5  00000111
 6  10100000
 7  01001100
 8  10001010 // MOV R2, null
 9  00000110
10  11010100 // LOAD AC1, 0
11  00000000
12  10110100 // LOOP 1, count_T
13  00001011
14  10100100
15  00011100
16  10010110 // MOV R1, G[AC1]
17  01001111
18  01010000
19  10010100 // MOV T[AC1], R1
20  01101001
21  01010000
22  10011000 // MOV G[AC1], R2
23  01001111
24  01010000
25  11100110 // INC AC1
26  10101110 // END_LOOP 1
27  00001110
28  11011000 // LOAD AC2, 0
29  00000000
30  10110100 // LOOP 1, pow_B+1
31  00001000
32  10100100
33  00111110
34  11010100 // LOAD AC1, 0
35  00000000
36  11001110 // LOAD AC3, AC2
37  10011010 // MOV R2, B[AC2]
38  00100000
39  10010000
40  10111000 // LOOP 2, count_T
41  00001011
42  10101000
43  00111011
44  10010110 // MOV R1, T[AC1]
45  01101001
46  01010000
47  10011110 // MOV R3, G[AC3]
48  01001111
49  11010000
50  00010110 // MUL R1, R2
51  00001101 // ADD R3, R1
52  10011100 // MOV G[AC3], R3
53  01001111
54  11010000
55  11100110 // INC AC1
56  11101110 // INC AC3
57  10101110 // END_LOOP 2
58  00101010
59  11101010 // INC AC2
60  10101110 // END_LOOP 1
61  00100000
62  10010110 // MOV R1, B[AC0]
63  00100000
64  00010000
65  00010100 // MUL R1, R0
66  10010100 // MOV B[AC0], R1
67  00100000
68  00010000
69  10000110 // MOV R1, count_T
70  00001011
71  11100100 // INC R1
72  10000100 // MOV count_T, R1
73  00001011
74  10101110 // END_LOOP 0
75  00000110
76  11011000 // LOAD AC2, h'0
77  00000000
78  10110000 // LOOP 0, pow_A+1
79  00001001
80  10100000
81  01101110
82  11010100 // LOAD AC1, b'0
83  00000000
84  11001110 // LOAD AC3, AC2
85  10010110 // MOV R1, A[AC2]
86  00001100
87  10010000
88  10110100 // LOOP 1, count_T
89  00001011
90  10100100
91  01101011
92  10010010 // MOV R0, G[AC1+0]
93  01001111
94  01000000
95  10011010 // MOV R2, C[AC3]
96  00100010
97  11010000
98  00010001 // MUL R0, R1
99  00001000 // ADD R2, R0
100 10011000 // MOV C[AC3], R2
101 00100010
102 11010000
103 11100110 // INC AC1
104 11101110 // INC AC3
105 10101110 // END_LOOP 1
106 01011010
107 11101010 // INC AC2
108 10101110 // END_LOOP 0
109 01010000
110 11010000 // LOAD AC0, 0
111 00000000
112 10110000 // LOOP 0, pow_A+r+1
113 00001010
114 10100000
115 01111010
116 11110010 // OUT C[AC0]
117 00100010
118 00010000
119 11100010 // INC AC0
120 10101110 // END_LOOP 0
121 01110010
122 01001111 // STOP-слово
```

ДОДАТОК Л

Тестування процесора Галуа

Тестування процесора Галуа виконується за допомогою наступного модуля

```
module test_processor_GF_2m;

	parameter m = 3'b100;
	parameter log_m = 2'b10;
	parameter instruction_width = 4'b1000;
	parameter Sel_size = 2'b10;
	parameter addr_program_width = instruction_width + 1'b1;
	parameter addr_data_width = instruction_width + 1'b1;

	// Inputs
	reg clk, rst;
	reg en_write_data_mem, en_write_program_mem;
	reg [m-1: 0] data_in;
	reg [instruction_width-1: 0] code_in;
	reg [addr_program_width-1:0] addr_code;
	reg [addr_data_width-1:0] addr_data;
	reg [Sel_size-1:0] Select_Main_1;
	reg Select_Main_2, Select_Main_3, Select_Main_4, Select_Main_5;

	// Outputs
	wire [m-1:0] data_out;
	wire ready;

	// Instantiate the Unit Under Test (UUT)
	processor_GF_2m #(m, log_m) uut (
		.clk(clk),
		.rst(rst),
		.en_write_data_mem(en_write_data_mem),
		.en_write_program_mem(en_write_program_mem),
		.data_in(data_in),
		.code_in(code_in),
		.addr_code(addr_code),
		.addr_data(addr_data),
		.Select_Main_1(Select_Main_1),
		.Select_Main_2(Select_Main_2),
		.Select_Main_3(Select_Main_3),
		.Select_Main_4(Select_Main_4),
		.Select_Main_5(Select_Main_5),
		.data_out(data_out),
		.ready(ready)
	);

	initial begin

		clk = 0;
		rst = 1;
		en_write_data_mem = 0;
		en_write_program_mem = 0;
		data_in = 0;
		code_in = 0;
		addr_code = 0;
		addr_data = 0;
		Select_Main_1 = 0;
		Select_Main_2 = 0;
		Select_Main_3 = 0;
		Select_Main_4 = 0;
		Select_Main_5 = 0;
		#50;
		rst = 0;
		#100;


	end

	always #10 clk = ~clk;

endmodule
```

В процесі тестування пам’ять команд та пам’ять даних процесора Галуа ініціалізується за допомогою текстових файлів, що містять машинний код програми та вхідні дані відповідно.

Модуль тестування за допомогою сигналу *rst* встановлює в початковий стан процесор Галуа та з фіксованою частотою змінює значення тактового сигналу *clk*. При зміні тактового сигналу відбувається зчитування машинного коду програми з пам’яті команд та її виконання.

Обов’язковими параметрами модуля тестування, які треба налаштувати перед початком тестування, є значення m та $\log_2 m$, що залежать від обраного поля Галуа.

Для тестування було обрано програми мовою Асемблера Галуа, що наведені у Додатку К.

Перша програма виконує множення многочленів (рис. Л.1-Л.6), коефіцієнтами яких є елементи поля Галуа $GF(2^4)$ з незвідним многочленом $x^4 + x + 1$:

$$P_8(y) = 13y^8 + 11y^7 + 7y^6 + y^4 + 12y^3 + 15y^2 + 4y + 8;$$

$$P_6(y) = 7y^6 + 10y^5 + 4y^4 + y^3 + 5y^2 + 3;$$

$$P_8(y) \cdot P_6(y) = 11y^{14} + 12y^{13} + 12y^{12} + 8y^{11} + 7y^{10} + 10y^9 + 10y^8 + \\ + 9y^7 + 2y^6 + 14y^5 + 15y^4 + 4y^3 + 5y^2 + 15y + 5.$$

Друга програма будує твірний многочлен коду Ріда-Соломона (рис. Л.7-Л.10) для кількості контрольних розрядів $r = 7$ (з метою збільшення наочності, значення r було зменшено порівняно з текстом програми, що наведено у Додатку К) та поля Галуа $GF(2^{10})$ з незвідним многочленом $x^{10} + x^3 + 1$:

$$g(y) = \prod_{i=1}^{7}\left(y - 2^i\right) = \\ = y^7 + 254y^6 + 1012y^5 + 607y^4 + 433y^3 + 428y^2 + 777y + 400.$$

Третя програма виконує кодування даних кодом Ріда-Соломона з кількістю контрольних розрядів $r = 5$ (з метою збільшення наочності,

значення r було зменшено порівняно з текстом програми, що наведено у Додатку К) над полем Галуа $GF(2^{13})$ з незвідним многочленом $x^{13}+x^4+x^3+x+1$. Вхідні дані для кодування: 5, 15, 1, 0, 7, 3 (з метою збільшення наочності, обсяг вхідних даних було зменшено порівняно з текстом програми, що наведено у Додатку К).

Кодування кодом Ріда-Соломона передбачає виконання таких кроків.

1. Представлення вхідних даних многочленом, коефіцієнти якого є елементами обраного поля Галуа:

$$a(y)=5y^5+15y^4+y^3+7y+3.$$

2. Побудова твірного многочлена для заданої кількості контрольних розрядів:

$$g(y)=\prod_{i=1}^{5}\left(y-2^i\right)=$$
$$=y^5+62y^4+792y^3+6336y^2+7213y+108.$$

3. Множення многочлена $a(y)$, що визначає вхідні дані, на твірний многочлен $g(y)$:

$$a(y)g(y)=3y^{10}+69y^9+1426y^8+18y^7+3627y^6+$$
$$+5654y^5+2186y^4+8057y^3+3677y^2+3792y+476.$$

Результат виконання третьої програми, процесором Галуа, наведено на рис. Л.11-Л.16.

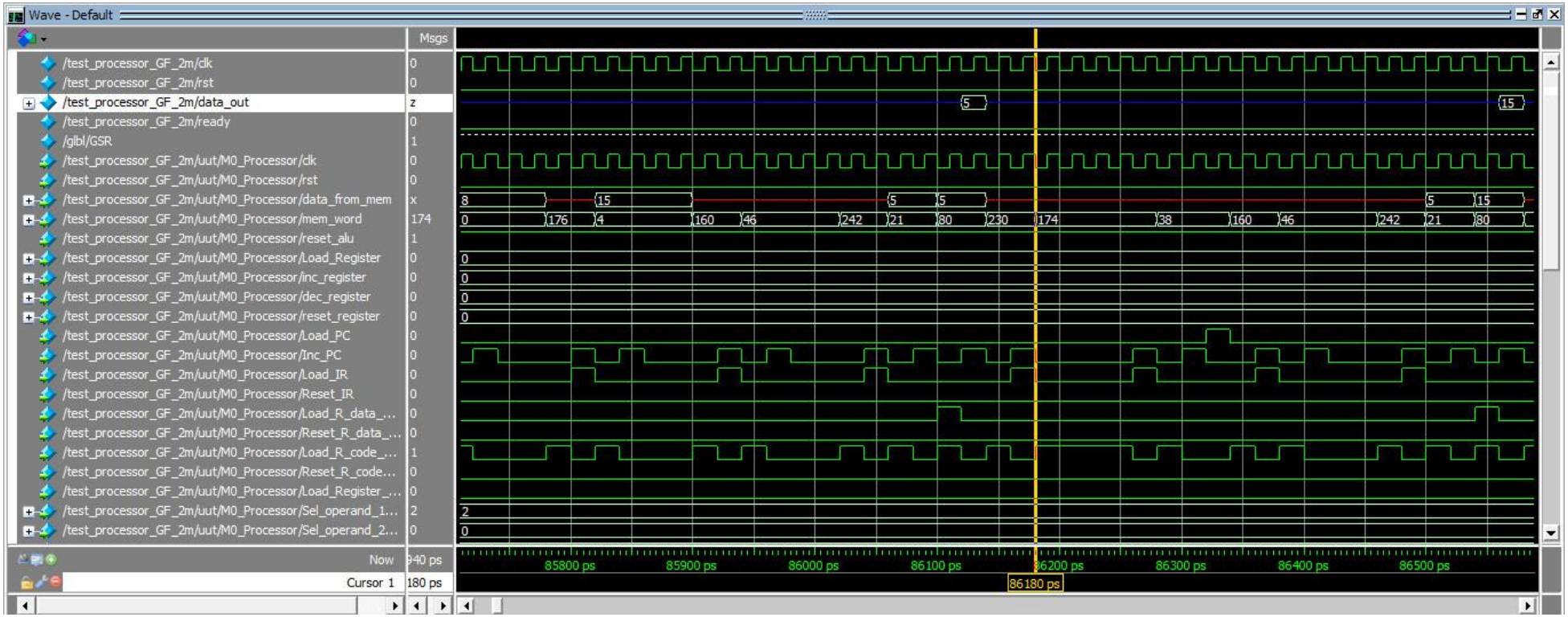


Рис. Л.1. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^0 та y^1)

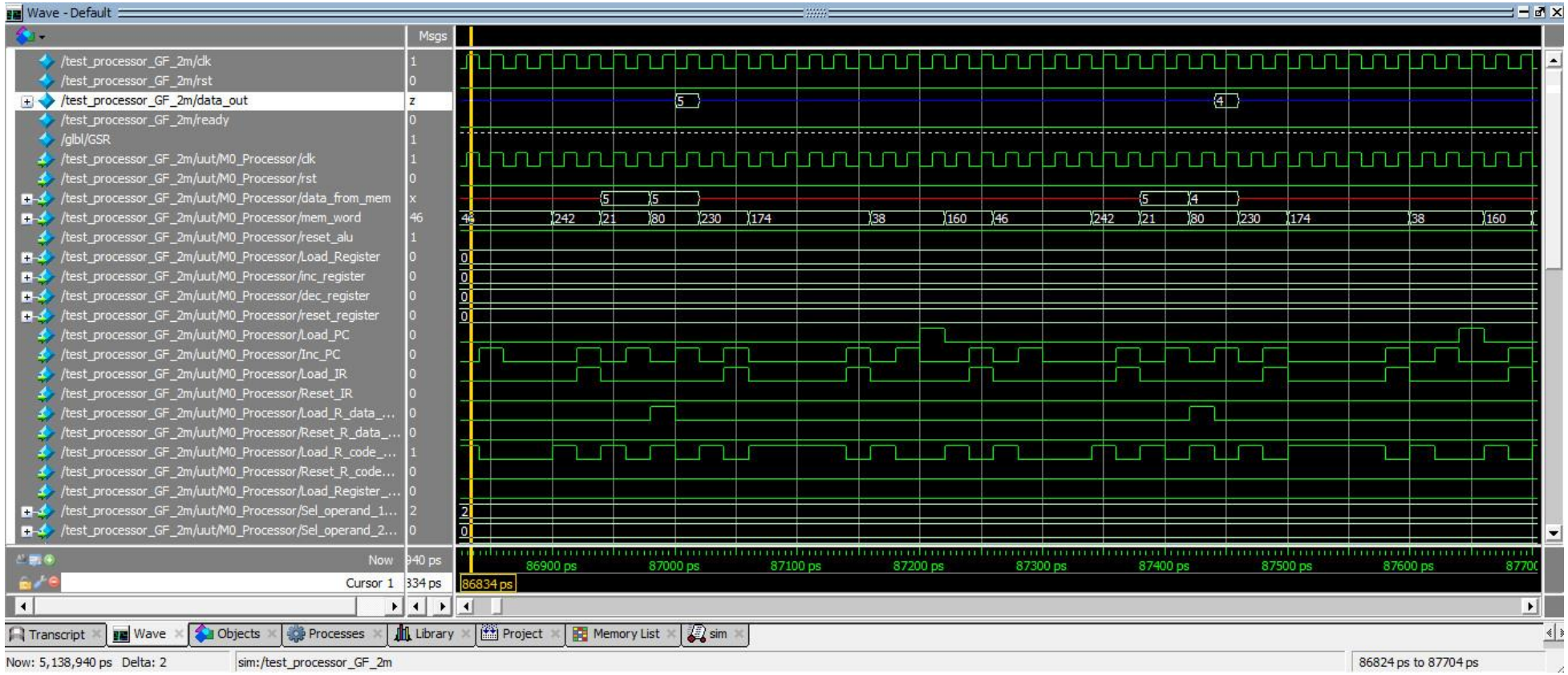


Рис. Л.2. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^2 та y^3)

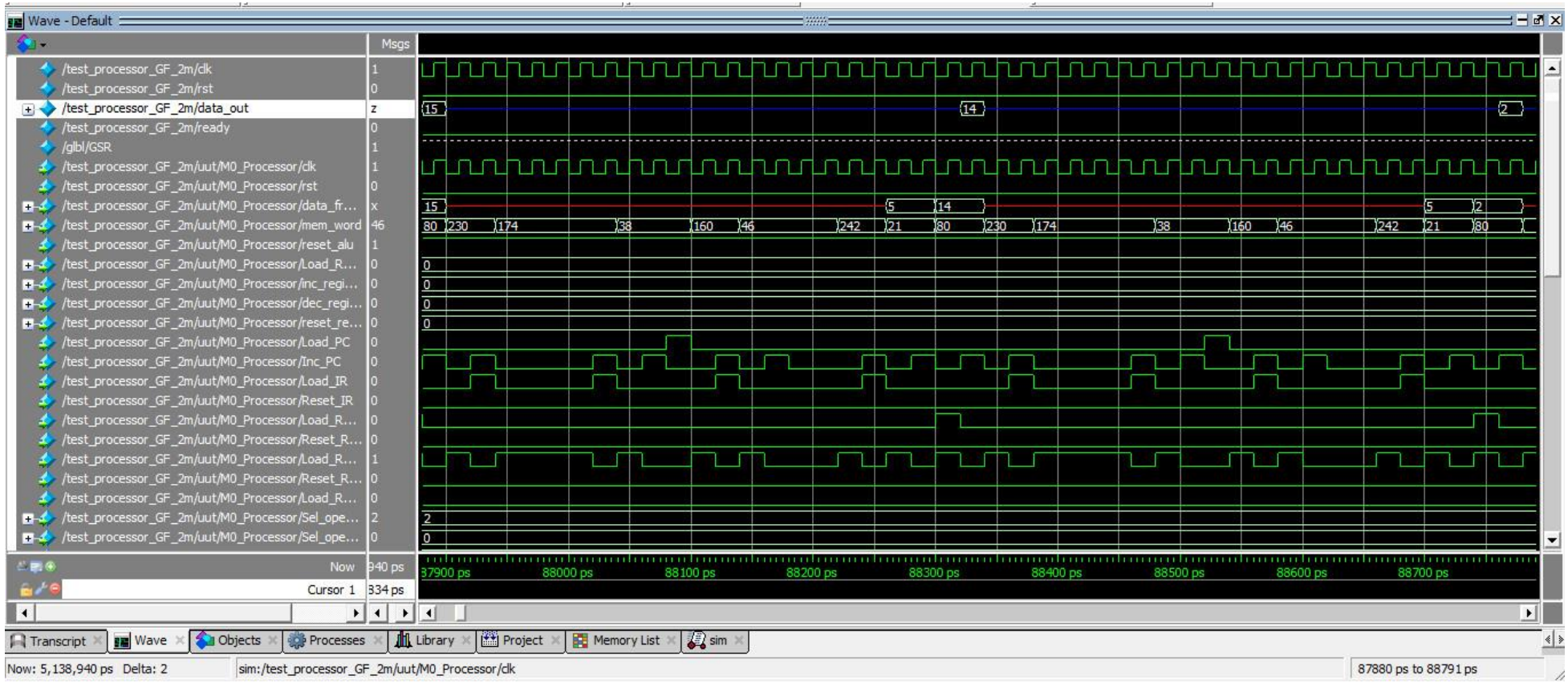


Рис. Л.3. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^4, y^5 та y^6)

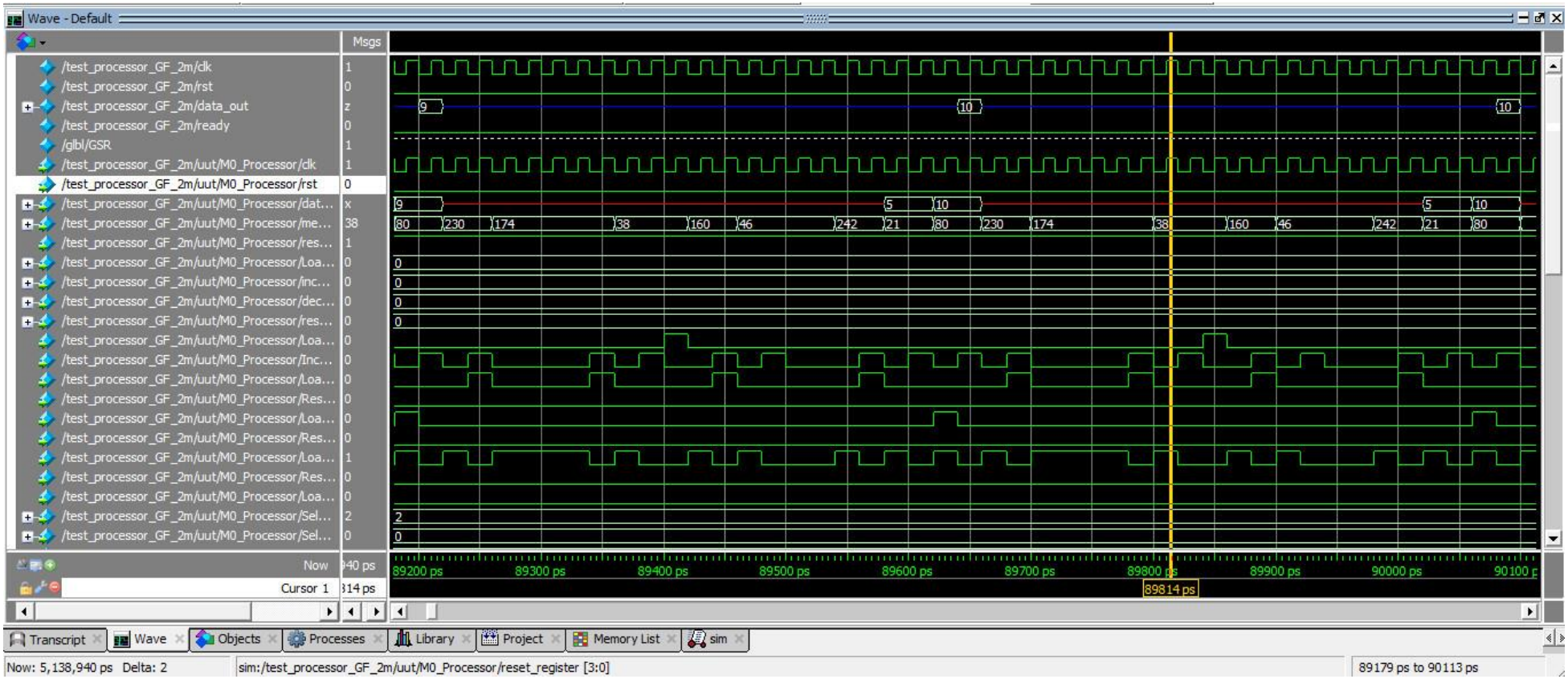


Рис. Л.4. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^7, y^8 та y^9)

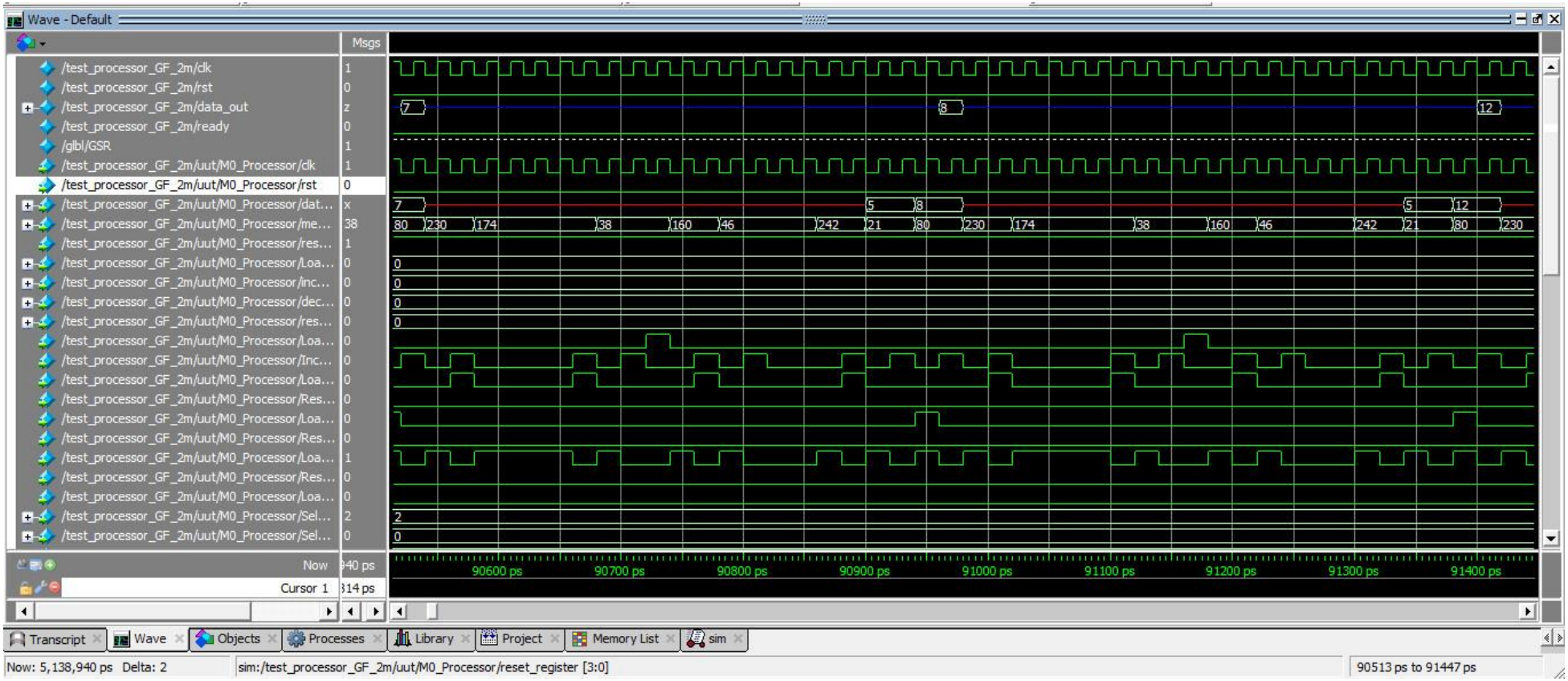


Рис. Л.5. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^8, y^9 та y^{10})

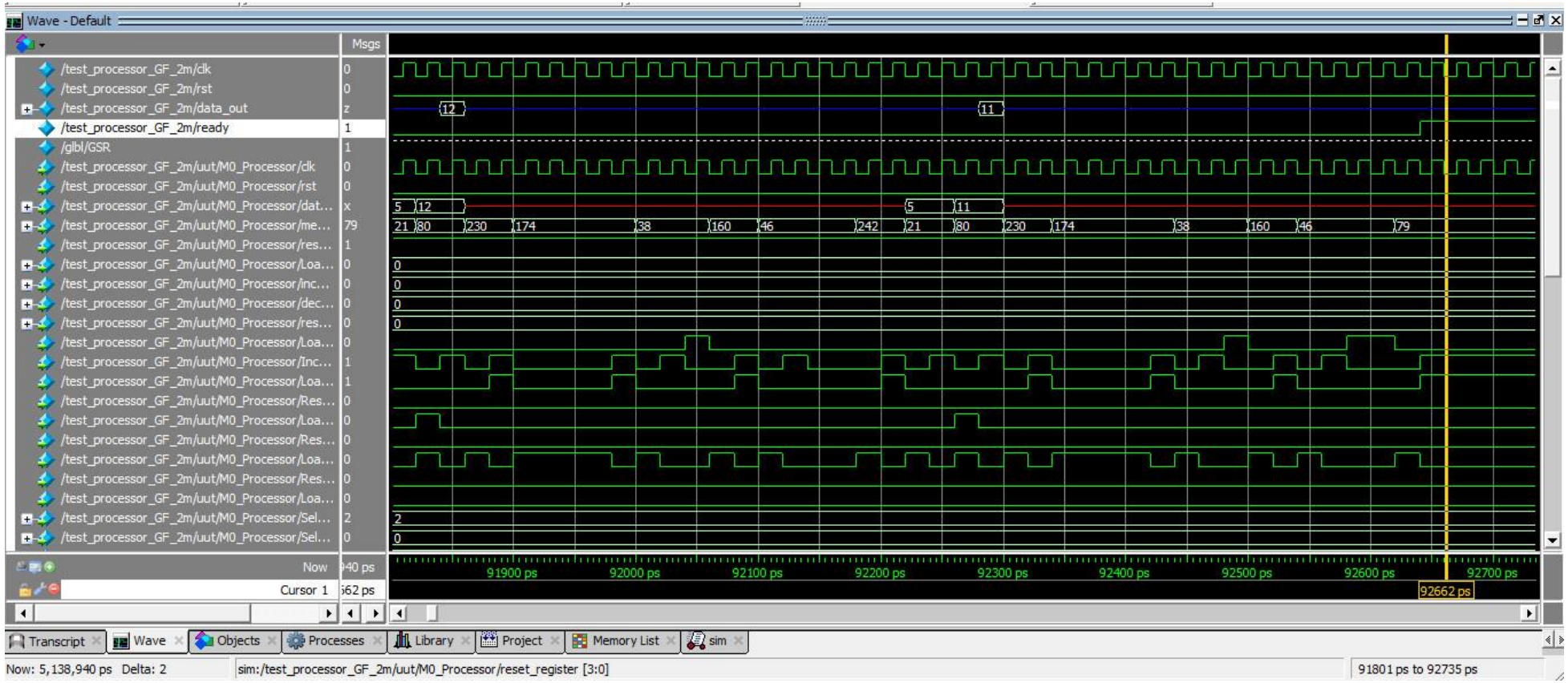


Рис. Л.6. Отримання результату множення многочленів на виході процесора Галуа (коефіцієнти многочлена при y^{11}, y^{12} та y^{13})

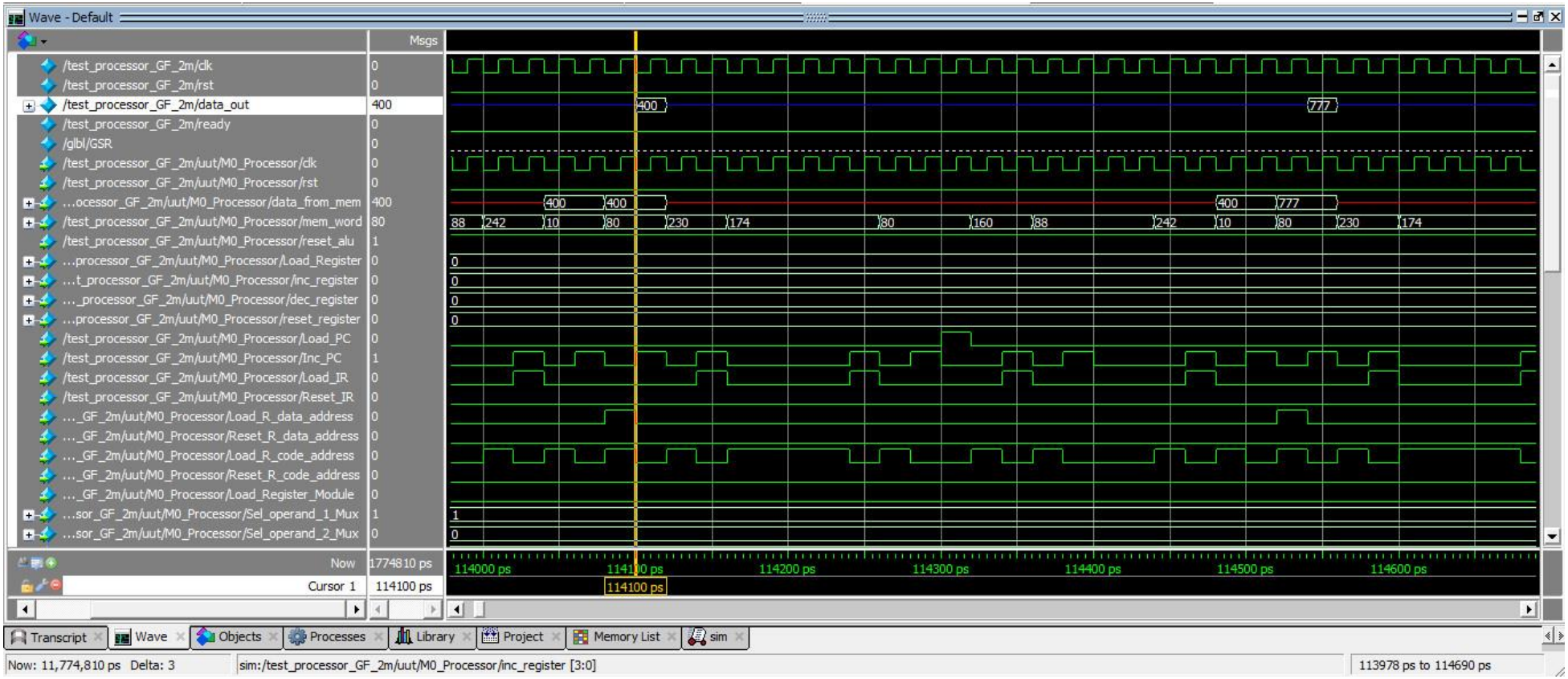


Рис. Л.7. Отримання твірного многочлена на виході процесора Галуа (коефіцієнти при y^0 та y^1)

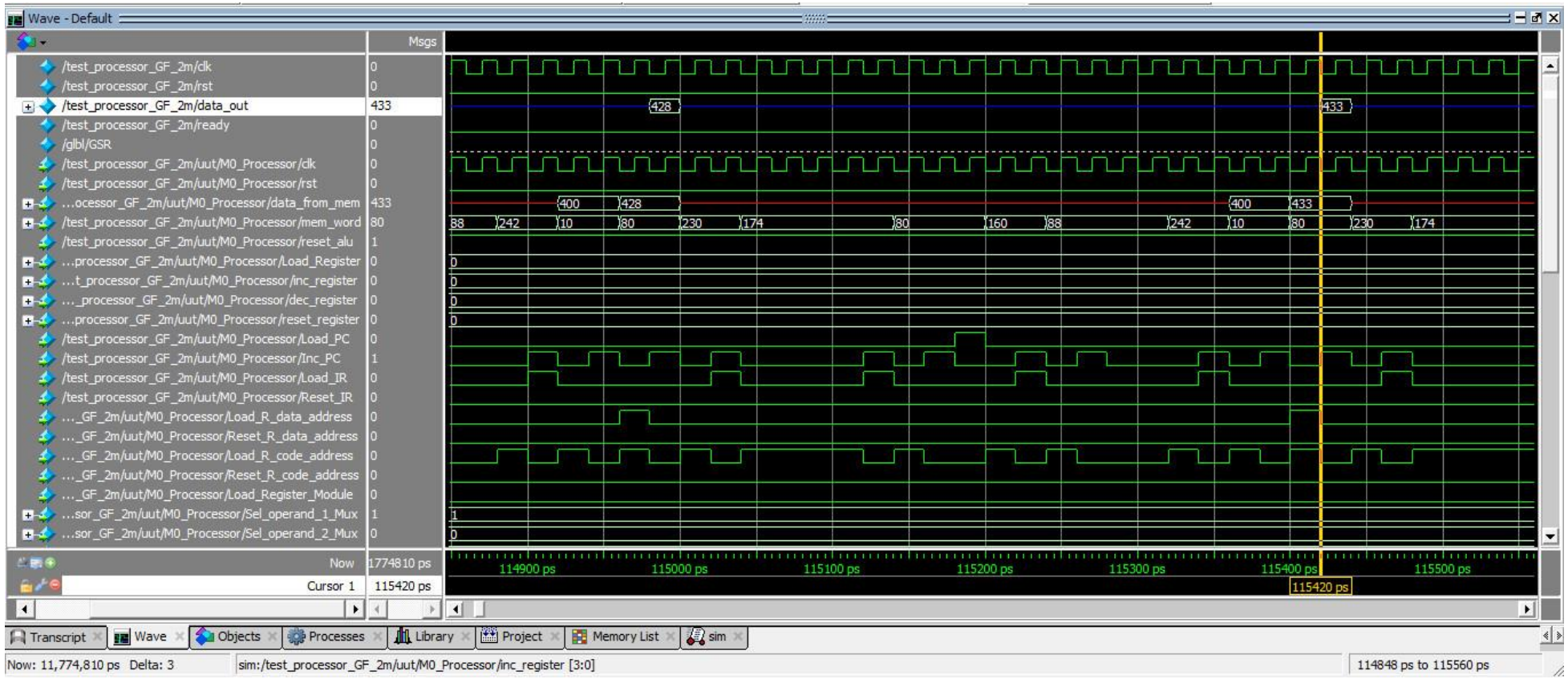


Рис. Л.8. Отримання твірного многочлена на виході процесора Галуа (коефіцієнти при y^2 та y^3)

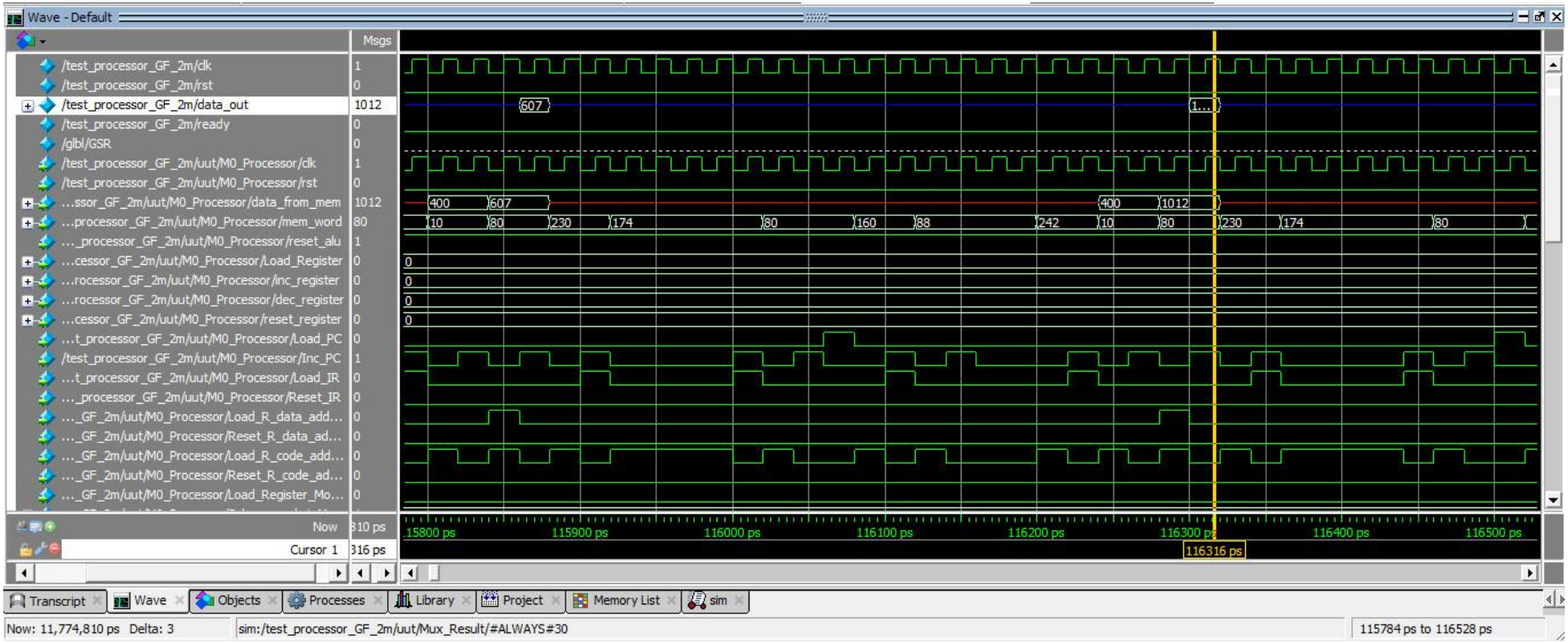


Рис. Л.9. Отримання твірного многочлена на виході процесора Галуа (коефіцієнти при y^4 та y^5)

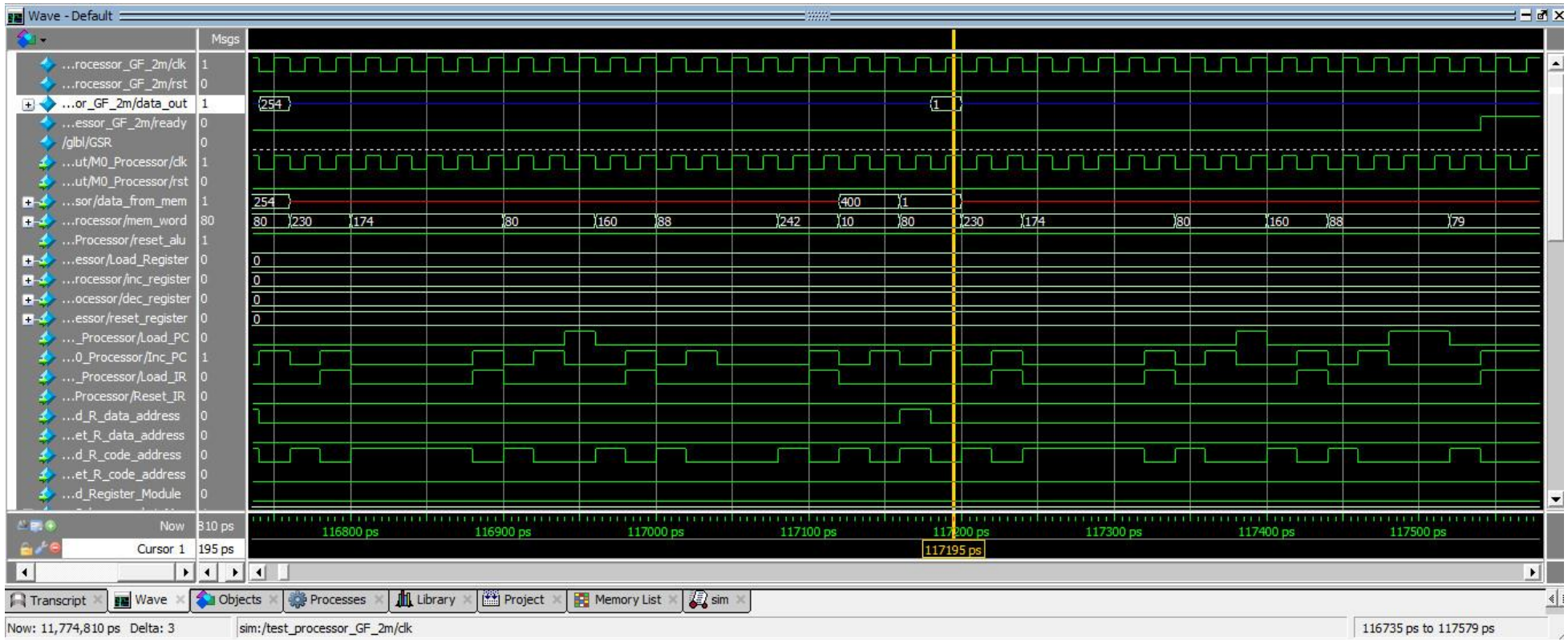


Рис. Л.10. Отримання твірного многочлена на виході процесора Галуа (коефіцієнти при y^6 та y^7)

Рис. Л.11. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнти многочлена при y^0 та y^1)

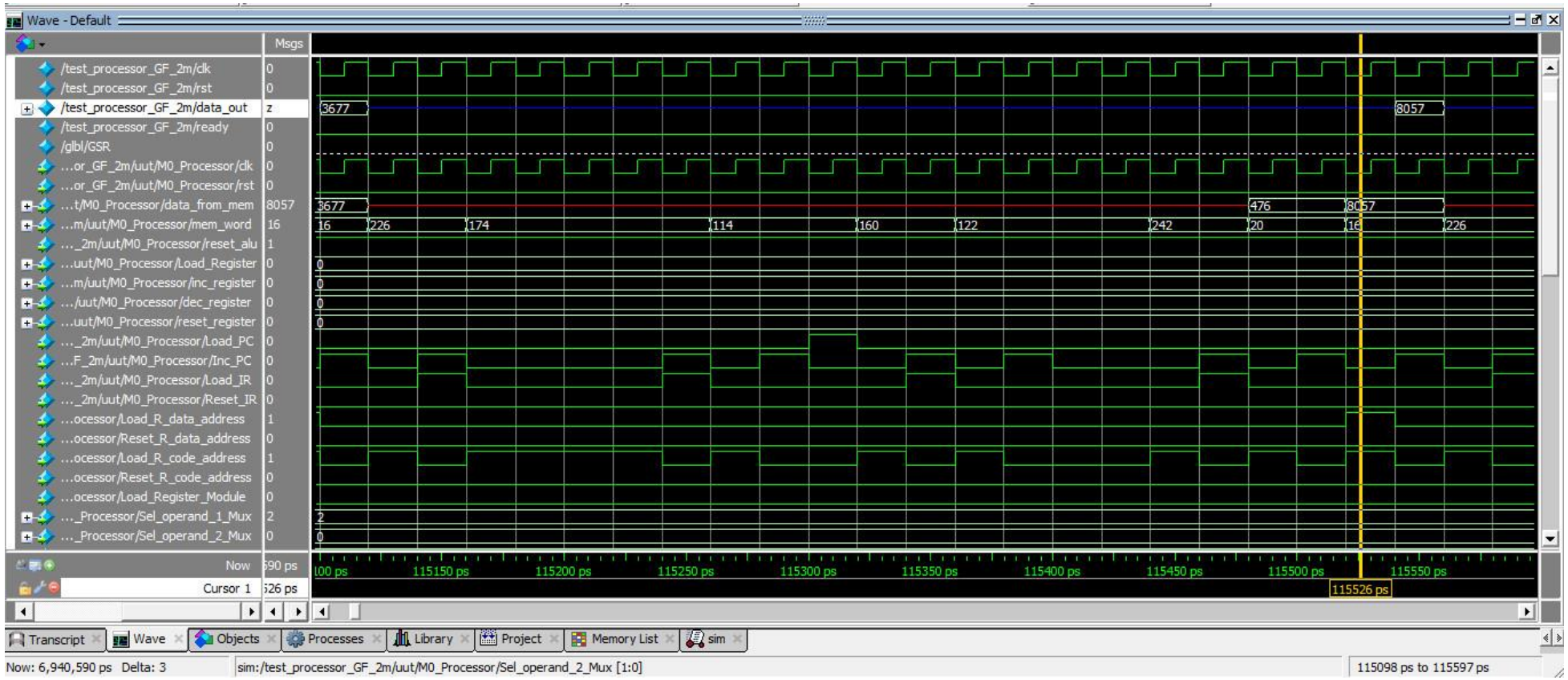


Рис. Л.12. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнти многочлена при y^2 та y^3)

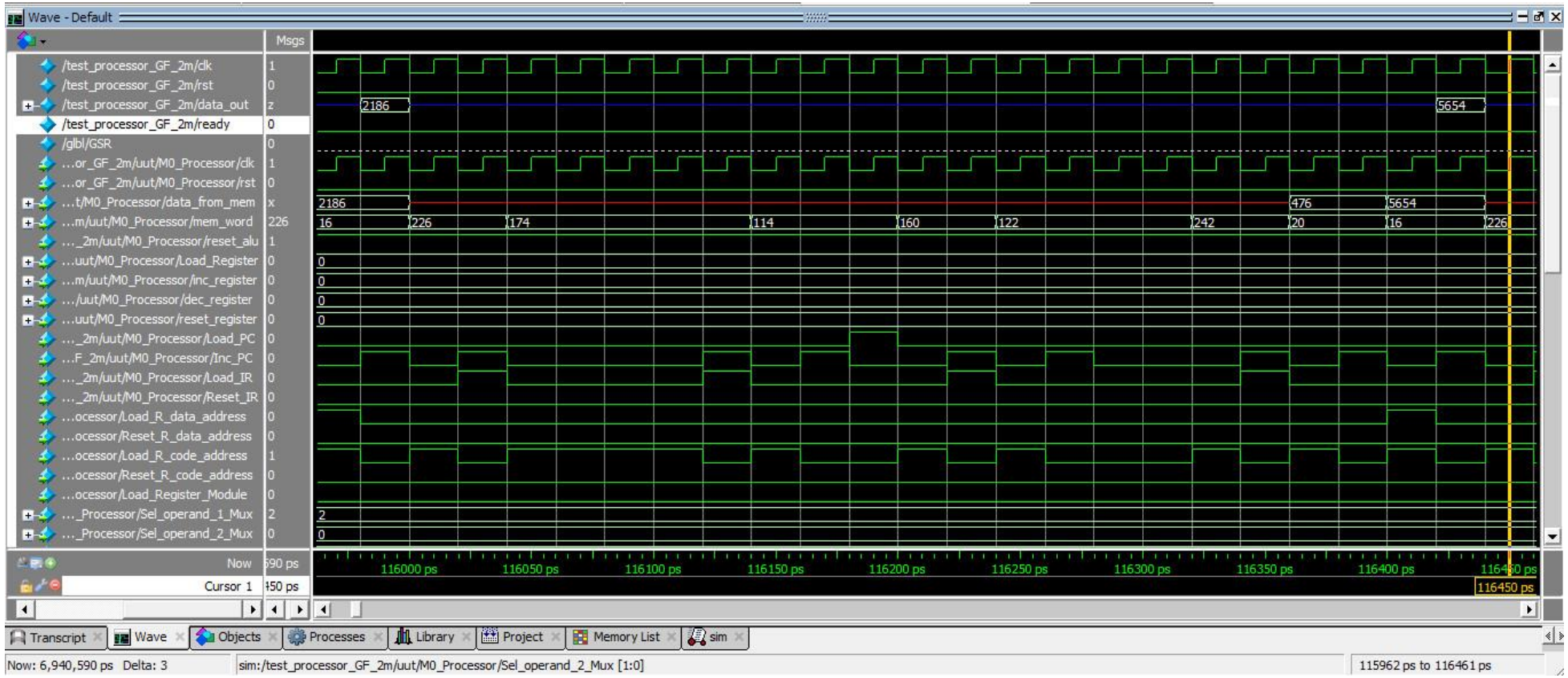


Рис. Л.13. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнти многочлена при y^4 та y^5)

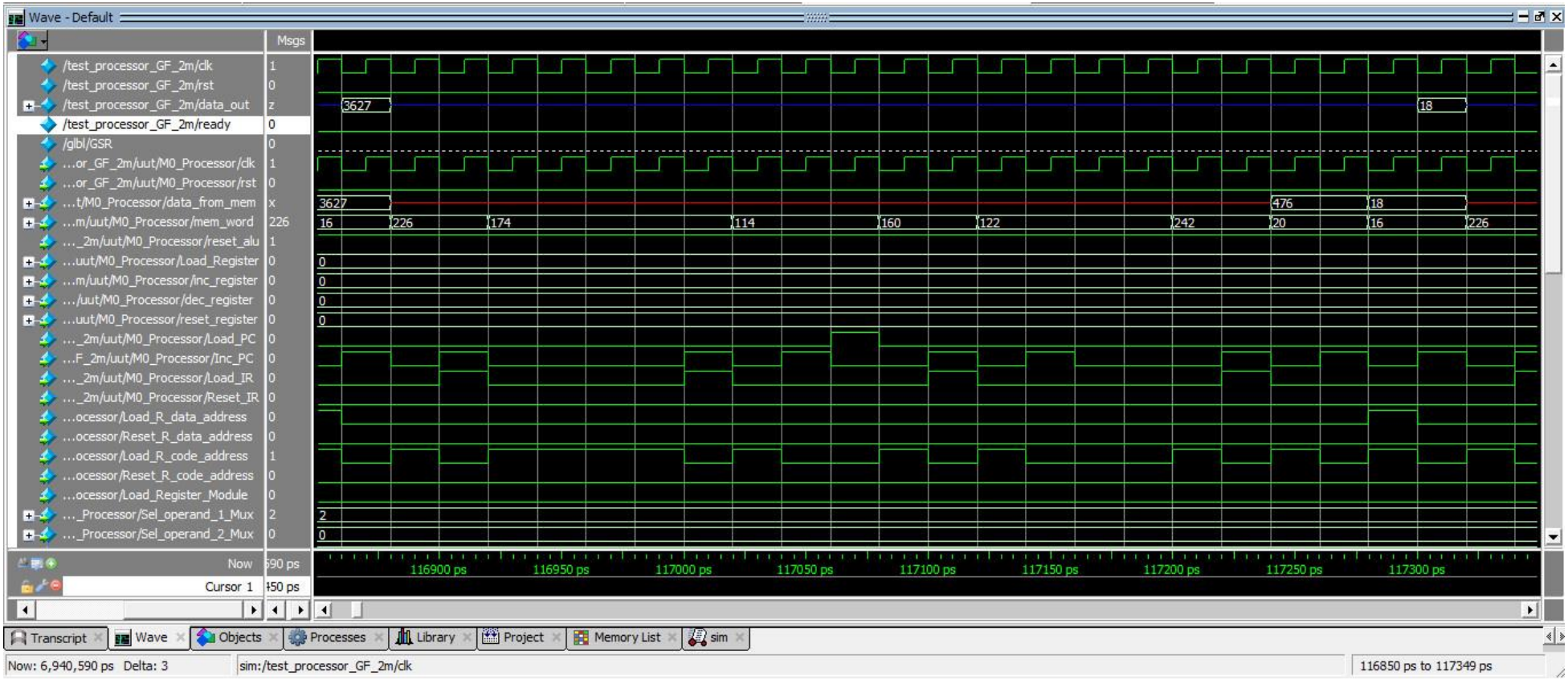


Рис. Л.14. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнти многочлена при y^6 та y^7)

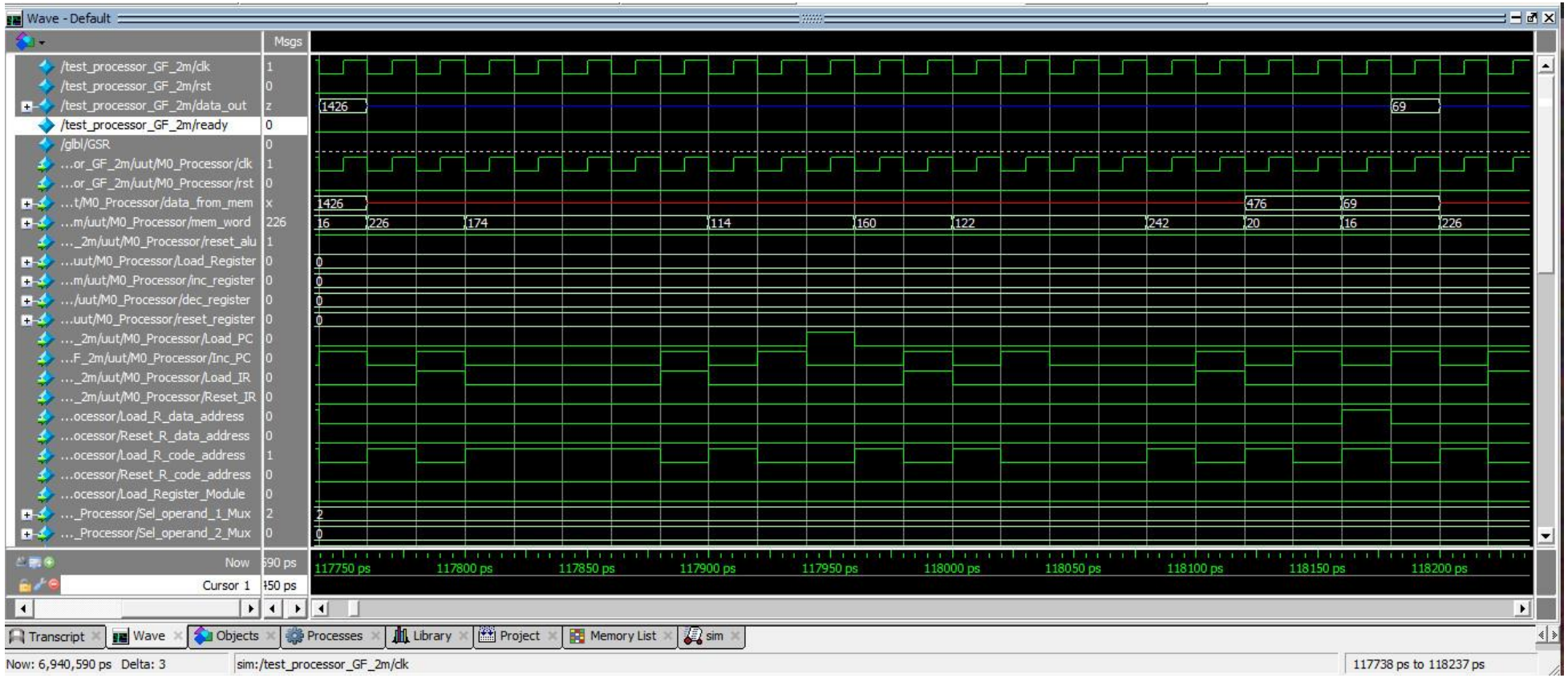


Рис. Л.15. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнти многочлена при y^8 та y^9)

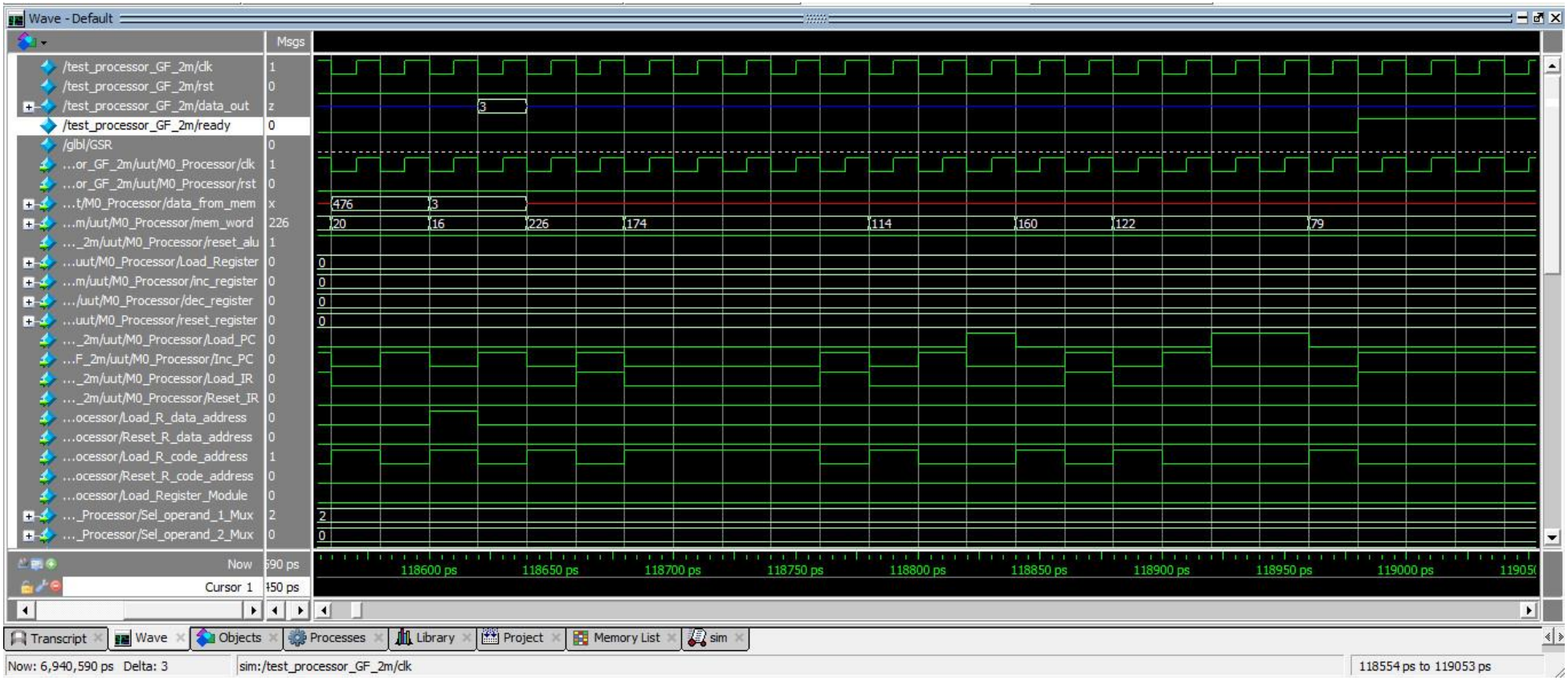


Рис. Л.16. Отримання даних, закодованих кодом Ріда-Соломона, на виході процесора Галуа (коефіцієнт многочлена при y^{10})

ДОДАТОК М

Акти про впровадження

ЗАТВЕРДЖУЮ
Перший проректор
Національного технічного університету України “Київський політехнічний інститут імені Ігоря Сікорського”

_____ Ю.І. Якименко

“ 30 ” травня 2017 р.

АКТ

про впровадження результатів наукових досліджень
Оная Миколи Володимировича,
виконаних у рамках дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях”, у навчальний процес Національного технічного університету України “Київський політехнічний інститут імені Ігоря Сікорського”
від 25 травня 2017 року

Комісія у складі: заступника декана факультету прикладної математики з навчально-методичної роботи к.т.н., доцента Сулеми Є.С. та в.о. завідувача кафедри програмного забезпечення комп’ютерних систем к.т.н., доцента Заболотньої Т.М. – склала цей акт про те, що результати дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях”, отримані особисто здобувачем Онаєм М.В., впроваджені в навчальний процес кафедри програмного забезпечення комп’ютерних систем факультету прикладної математики у 2017/2018 навчальному році.

Наукові результати, що впроваджуються	Форма впровадження	Ефект від впровадження
1. Архітектура та система команд спеціалізованого процесора Галуа, орієнтованого на виконання операцій у скінченних полях	Дисципліна “Архітектура комп’ютера”. Тема 3. Спеціалізовані процесори. Лекція 5. Криптографічні процесори. Лабораторна робота. Програмування процесора, орієнтованого на виконання операцій у скінченних полях.	Зростання продуктивності обчислень порівняно з універсальними обчислювальними засобами.

Наукові результати, що впроваджуються	Форма впровадження	Ефект від впровадження
2. Метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та відповідні структури апаратних засобів для його реалізації.	Дисципліна “Цифрова обробка сигналів і зображень”. Тема 5. Цифрові фільтри. Лекція 12. Фільтри у полях Галуа. Лабораторна робота. Дослідження методів виконання операцій у полях Галуа виду $GF(2^m)$.	Зростання швидкодії в середньому на 15% порівняно з існуючим методом.
3. Модифікований метод піднесення до степеня елементів поля $GF(p)$ з ковзним вікном	Дисципліна “Теорія інформації та кодування”. Тема 1. Завадостійке кодування. Лекція 3. Код Ріда-Соломона. Лабораторна робота. Кодування та декодування даних кодом Ріда-Соломона з використанням спеціалізованого процесора Галуа.	Зростання швидкодії в середньому на 7-9% порівняно з існуючим методом.

Заступник декана факультету прикладної математики з навчально-методичної роботи — Є.С. Сулема

В.о. завідувача кафедри програмного забезпечення комп’ютерних систем — Т.М. Заболотня

ЗАТВЕРДЖУЮ
Проректор з наукової роботи
Національного технічного
університету України “Київський
політехнічний інститут імені
Ігоря Сікорського”

_____ М.Ю. Ільченко

“__” _____ 2017 р.

АКТ

про впровадження результатів наукових досліджень
Оная Миколи Володимировича, виконаних у рамках дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях” у Національному технічному університеті України “Київський політехнічний інститут імені Ігоря Сікорського”
від 25 травня 2017 року

Комісія у складі: заступника декана факультету прикладної математики з навчально-методичної роботи к.т.н., доцента Сулеми Є.С. та в.о. завідувача кафедри програмного забезпечення комп’ютерних систем к.т.н., доцента Заболотньої Т.М. – склала цей акт про те, що результати дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях”, отримані особисто здобувачем Онаєм М.В., впроваджені у:

- НДР № 2854-п “Розроблення та дослідження високоефективних архітектур спеціалізованих комп’ютерних систем для реалізації обчислень у скінченних полях” (державний реєстраційний №0115U000319) за період 2015-2016 рр.
- НДР № 2525-п “Методи та засоби інформаційного забезпечення систем автоматизованого імпорту об’єктів на основі графічного кодування даних” (державний реєстраційний №0112U003175) за період 2012-2013 рр.

Наукові результати, що впроваджуються	Форма впровадження	Ефект від впровадження
1. Архітектура та система команд спеціалізованого процесора Галуа, орієнтованого на виконання операцій у скінченних полях	Модель процесора Галуа мовою *Verilog* у середовищі розробки *Xilinx*.	Зростання продуктивності обчислень порівняно з універсальними обчислювальними засобами у комп'ютерній системі.
2. Метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ та відповідні структури апаратних засобів для його реалізації.	Модель мовою *Verilog*. Текст програми мовою *C#*.	Зростання швидкодії в середньому на 15% порівняно з існуючим методом у комп'ютерній системі.

Заступник декана факультету прикладної математики з навчально-методичної роботи Є.С. Сулема

В.о. завідувача кафедри програмного забезпечення комп'ютерних систем Т.М. Заболотня

vit
зарождая технологии

ЗАТВЕРДЖУЮ
Директор
ТОВ “Відео Інтернет Технології”

_____ Ю.В. Бухтіяров

“28” вересня 2017 р.

АКТ

про впровадження результатів наукових досліджень
Оная Миколи Володимировича,
виконаних у рамках дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях”

Комісія у складі: голови – директора Бухтіярова Ю.В., технічного директора Сокура В.В, керівника відділу продажу Бондаря С.А розглянула результати впровадження дисертаційної роботи на здобуття наукового ступеня кандидата технічних наук на тему “Методи та засоби підвищення ефективності реалізації обчислювальних операцій у скінченних полях”, отримані особисто здобувачем Онаєм М.В., у систему обробки та аналізу зображень для розпізнавання реєстраційних номерів транспортних засобів, та встановила, що розроблена архітектура спеціалізованого процесора і метод високошвидкісного виконання адитивних та мультиплікативних операцій над елементами поля $GF(2^m)$ забезпечує приріст швидкодії на 7-10% порівняно з існуючою роботою системи обробки та аналізу зображень для розпізнавання реєстраційних номерів транспортних засобів.

Внаслідок поліпшення якості послуг, що надаються компанією, річний приріст прибутку, який очікується, від впровадження результатів дисертаційної роботи у систему обробки та аналізу зображень для розпізнавання реєстраційних номерів транспортних засобів компанії ТОВ “Відео Інтернет Технології”, за розрахунками, складає близько 80 тис. грн.

Україна * Товариство з обмеженою відповідальністю * м. Київ
“Відео Інтернет Технології”
Ідентифікаційний код 33497874

Голова комісії Ю.В. Бухтіяров

Члени комісії:

Технічний директор В.В. Сокур

Керівник відділу продажу С.А. Бондар

ООО «Видео Интернет Технологии»
03150, Украина, г. Киев, ул. Предславинская, 34-Б
тел/факс: +380 44 585 48 42
www.vitcompany.com