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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерна інженерія”

спеціальності 123 “ Комп’ютерна інженерія ”

на тему: Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері

Виконала : студентка __4__ курсу, групи ІВ-93
(шифр групи)

_____Трибунська Кароліна Євгенівна_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____доцент, к.т.н. Марковський О. П._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) старший викладач Виноградов Ю. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __ декан ФПМ, д.т.н., професор Дичка І.А._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 р.

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Трибунської Кароліни Євгенівни

1. Тема проєкту *Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері*
керівник проєкту Марковський Олександр Петрович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2023 року №1139-с

2. Термін здачі студентом закінченого проєкту *10 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані*.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд існуючих методів і технологій виконання мультиплікативних операцій модулярної арифметики над числами великої розрядності.

Розділ 2. Розробка методу прискореного мультипроцесорного модулярного множення з використанням групової редукції Монтгомері.

Розділ 3. Розробка програмних засобів мультипроцесорного модулярного множення з використанням групової редукції Монтгомері.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, діаграма класів, алгоритм мультипроцесорного множення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Виноградов Ю. М.		

7. Дата видачі завдання «12» січня 2023 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.01.2023-12.01.2023*	
2.	*Вивчення та аналіз завдання*	*12.01.2023-15.03.2023*	
3.	*Розробка алгоритму*	*15.03.2021-25.03.2023*	
4.	*Програмна реалізація системи*	*5.04.2023-15.04.2023*	
5.	*Оформлення пояснювальної записки*	*15.04.2023-20.05.2023*	
6.	*Захист програмного продукту*	*25.04.2023*	
7.	*Передзахист*	*26.05.2023*	
9.	*Захист*	*22.06.2023*	

Студент-дипломник _____ Кароліна ТРИБУНСЬКА
(підпис)

Керівник проєкту _____ Олександр МАРКОВСЬКИЙ
(підпис)

АНОТАЦІЯ

Дипломний проект порушує проблему підвищення швидкодії широкого кола криптографічних алгоритмів захисту, для яких базовою є операція модулярного множення. З огляду на сучасний етап розвитку інформаційних технологій, стає актуальним питання прискорення операції модулярного множення .

У дипломному проекті пропонується новий підхід до здійснення операції модулярного множення, який побудований на технології Монтгомері і полягає у проведенні обробки декількох бітів проміжного результату одночасно Теоретично доведено та експериментально підтверджено, що розроблений метод дозволяє прискорити операцію модулярного множення в 8 разів.

Ключові слова: модулярне множення, технологія Монтгомері, модулярне експоненціювання, асиметричне шифрування.

ANNOTATION

The diploma work raises the problem of increasing the speed of a wide range of cryptographic security algorithms, for which the base operation is modular multiplication. Given the current stage of development of information technology, the issue of accelerating the operation of modular multiplication becomes relevant.

The diploma work proposes a new approach to the implementation of the modular multiplication operation, built on the Montgomery technology and consisting in processing several bits of the intermediate result simultaneously. It has been theoretically proven and experimentally confirmed that the developed method makes it possible to speed up the operation of modular multiplication by 8 times.

Key words: modular multiplication, Montgomery technology, modular exponentiation, asymmetric encryption.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.468243.002 ТЗ***	Метод мультипроцесорного	5		
			модулярного множення з			
			використанням групової			
			редукції Монтгомері			
			Технічне завдання			
	А4	***ІАЛЦ.468243.003 ПЗ***	Метод мультипроцесорного	66		
			модулярного множення з			
			використанням групової			
			редукції Монтгомері			
			Пояснювальна записка			
	А1	***ІАЛЦ.468243.004 Д1***	Метод прискореного	1		
			модулярного множення для			
			систем криптографічного			
			захисту даних			
			Блок схема алгоритму			
	А1	***ІАЛЦ.468243.005 Д2***	Метод мультипроцесорного	1		
			модулярного множення з			
			використанням групової			
			редукції Монтгомері			
			Діаграма класів			
	А1	***ІАЛЦ.468243.006 Е1***	Метод мультипроцесорного	1		
			модулярного множення з			
			використанням групової			
			редукції Монтгомері			
			Структурна схема			
	А1	***ІАЛЦ. 468243.007 Д3***	Метод мультипроцесорного	15		
			модулярного множення з			
			використанням групової			
			редукції Монтгомері			
			Текст програмного коду			

					ІАЛЦ.468243.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Трибунська			*Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері* Опис альбому
Перев		Марковський			

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, ІВ-93

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері*»

Київ – 2023

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив	Трибунська			**Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері** Технічне завдання	Літ.	Аркуш	Аркушів
Перевірив	Марковський					1	5
					КПІ ім. Ігоря Сікорського, ФІОТ, ІВ-93		
Н. Контр.	Виноградов						
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері.

Область застосування: технології захисту даних та програм в комп’ютерних системах та мережах.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп’ютерної інженерії», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Мета розробки полягає в прискоренні обчислень реалізації операції модулярного множення, як частини операції модулярного експоненціювання.

Призначення розробки полягає в забезпеченні швидкого та безпечного виконання операцій модулярного множення на мультипроцесорних системах. Використання групової редукції Монтгомері та паралельних обчислень на багатоядерних процесорах дозволяє значно прискорити ці операції.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки дипломного проекту є офіційна документація, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

4.1 Кабір Л.А. Метод прискорення модулярного множення за технологією Монтгомері / Л.А. Кабір, О.В. Русанова, І.О. Гуменюк //Альманах науки No 1(52).- 2022.- С.44-46.

4.2 Трибунська К.Є. Метод прискорення модулярного множення з використанням групової редукції /К.Є. Трибунська // Актуальні питання розвитку науки та освіти: матеріали М Міжнародної науково-практичної конференції м.Львів, 30-31 березня 2022 року.-Львів: Львівський науковий форум, 2022.- С.38- 46.

4.3 Osadchyy V. The Order of Edwards and Montgomery Curves «, / V.Osadchyy // WSEAS Transactions on Mathematics, - 2020.- Vol. 19.- No 25, - P. 253-264.

4.4 Haches G. Montgomery multiplication with no final subtraction./ G. Haches, J.J. Quisquater // Cryptographic Hardware and Embedded System-CHES’2000. LNCS-1965, Springer-Verlag. — 2000.- P. 293-301.

4.5 Марковський О.П. Метод прискорення експоненціювання з використанням передобчислень / О.П. Марковський, О.В. Русанова, А.А. Олієвський, В.М.Черевик // Телекомунікаційні та інформаційні технології. - 2018.- No 1(58). – С.31-39.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс системи.

- Розробити метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері.
- Забезпечити можливість ефективного множення чисел великого розміру за допомогою розподілених обчислень на багатьох процесорах.
- Забезпечити можливість інтеграції розробленого методу з існуючими системами та програмним забезпеченням.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- PyCharm 2020 IDE версії або вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- Оперативна пам’ять не менше 500 МБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.01.2023-15.01.2023
Вивчення та аналіз завдання	15.01.2023-15.03.2023
Розробка архітектури та загальної структури системи	15.03.2023-25.03.2023
Розробка структур окремих частин системи	25.03.2023-5.04.2023
Програмна реалізація системи	5.04.2023-15.04.2023
Виправлення помилок	15.04.2023-20.05.2023
Оформлення пояснювальної записки	26.04.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері*»

Київ – 2023

ЗМІСТ стр.

ПЕРЕЛІК СКОРОЧЕНЬ

DFT	(Discrete Fourier Transform) Дискретне перетворення Фур’є
DFT	(Fast Fourier Transform) Швидке перетворення Фур’є
SLD	(Small and Large Digit) Метод малої та великої кількості розрядів
BR	(Binary Reduction) Метод двійкової редукції
CM	(Carry-save Multiplication) Метод множення з переносом

ВСТУП

Поліпшення інформаційної інтеграції в сучасних умовах було б неможливим без використання і розвитку хмарних технологій. Через це, з'явилася необхідність постійно удосконалювати методи захисту даних.

Забезпечення високого рівня захисту інформації в криптографічних системах стає необхвдною умовою для того, щоб досягти ефективної інформаційної інтеграції.

Значним досягненням останнього століття у сфері комп'ютерних технологій є розробка технології хмарних обчислень. Ця технологія дає можливість отримати потужні ресурси пам'яті для виконання складнх обчислень. До цих ресурсів можна отримати дистанційний доступ, що зумовлює широке застосування цієї технологііі в багатьох сферах. Разом з можливостями, які надають хмарні технології, існують проблеми, які породило їх створення. Наприклад, надані ресурси можна використати з метою порушення криптографічних протоколів. Оскільки такі технології можуть використовуватися з метою зламу, необхідно підвищити рівень безпеки криптографічних алгоритмів захисту.

Збільшення пропускної здатності інформаційних каналів створює потребу у збільшенні ефективності заходів щодо забезбечення безпеки інформації в комп'ютерних мережах. Найбільш поширеними методами захисту даних, які пердаються за допомогою комп'ютерних мереж, є алгоритми і протоколи захисту інформації. Велика кількість криптографічних протоколів і методів мають за основу асиметричне шифрування.

Такі технології застосовують, як основу, важкі в обчислювальному плані задачі з теорії чисел.

Одна з основних обчислювальних операцій методів цього типу – модулярне множення чисел, кількість розрядів яких більша ніж розрядність

процесора. Час здійснення цих операцій залежить від розрядності чисел, які опрацьовують алгоритми. Зазвичай, використовують числа довжиною дві тисячі сорок вісім бітів.

Ця довжина дозволяє досягти балансу між швидкістю виконання операцій та рівнем захищеності даних. Важливими в криптографічних протоколах є операції модулярної арифметики. З ціллю забезпечення кращого захисту інформації довжина чисел, що застосовуються в модулярній арифметиці, є декілька тисяч і має можливість збільшуватися далі.Тому реалізація операцій модулярного експоненціювання і множення із застосуванням чисел такої довжини потребує значних ресурсів.

В основному в криптографічних протоколах з асиметричним шифруванням використовується операція модулярного експоненціювання, виконується обчислення $G^Z mod\ M$.

Надійність захисту даних в даному випадку повністю залежить від вибраної розрядності чисел. У сучасних системах криптографічного захисту часто використовують розрядність 2048 або 4096. Очевидно, щоб підвищити безпеку зберігання даних, необхідно збільшити розрядність використовуваних чисел до 8192.

Між тим, якщо збільшити кількість бітів з яких складаються числа, то збільшаться і обсяги обчислень, які виконуються при модулярному експоненціюванні. Особливо помітне погіршення часу виконання обчислень на телефонах або непотужних термінальних пристроях, які реалізують мережеві протоколи. Якщо виконувати операцію модулярного експоненціювання на таких пристроях із застосуванням чисел, довжина яхих більша за дві тисячі сорок вісім бітів, то час реалізації криптографічного протоколу буде критичним. Зважаючи на це, існує актуальна задача пришвидшення цієї базової операції на телефонах та непотужних термінальних пристроях, які реалізують криптографічні протоколи.

Дослідження і розробка нових ефективних алгоритмів пришвидшення операції модулярного експоненціювання стають ключою задачею. Її вирішення дозволить працювати в реальному часі.

Наукові дослідження, які направлені на акселерацію програмних або апаратних засобів реалізації операцій модулярної арифметики, сприяють покращенню ефективності та надійності криптографічних протоколів в мобільних телефонах та непотужних пристроях.

Забезпечення конфіденційності доступу до даних вимагає покращення існуючих та розробку нових методів. Розробка і реалізація нових методів прискорення операції модулярного експоненціювання зможуть забезпечити вищий рівень безпеки в хмарних середовищах.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ МЕТОДІВ І ТЕХНОЛОГІЙ ВИКОНАННЯ МУЛЬТИПЛІКАТИВНИХ ОПЕРАЦІЙ МОДУЛЯРНОЇ АРИФМЕТИКИ НАД ЧИСЛАМИ ВЕЛИКОЇ РОЗРЯДНОСТІ

Множення є однією з найважливіших операцій у криптографічних алгоритмах. Продуктивність та швидкодія множення мають велике значення для оптимізації обчислювальних процесів та забезпечення ефективного використання обчислювальних ресурсів.

У зв'язку з цим, виникла потреба у розробці та вдосконаленні методів прискореного множення, які дозволяють зменшити обчислювальну складність та забезпечити швидкодію операцій множення. Розробники та дослідники присвятили значний час і зусилля розробці різних алгоритмів та технологій, що прискорюють множення чисел.

1.1 Аналіз особливостей використання операцій модулярної арифметики в системах криптографічного захисту даних

Для того щоб виділити основні особливості застосування операцій модулярної арифметики в сучасних криптографічних системах та оцінити алгоритми для їх реалізації необхідно визначити критеріїї, за якими буде оцінюватися їх ефективність. Значущість деяких критеріїв, в умовах застосування методу в конкретному протоколі, залежить від певних умов його практичного використання. Крім того, для кожного алгоритма існують різні критірії щодо визначення його якості, що зумовлено його математичними особливостями. Наприклад, для криптографічних алгоритмів з асиметричним

шифруванням (шифрування з «відкритим ключем») головним критерієм є час обчислення закритого і відкритого ключів. А для систем із симетричним шифруванням цей критерій взагалі відсутній. Можна виділити декілька спільних критеріїв, які пред'являються до всіх алгоритмів, де використовуються операції модулярної арифметики:

- Ступінь надійності зберігання інформації. Цей критерій оцінюється в основному тим, скільки ресурсів необхідно витратити, щоб порушити алгоритм захисту даних.
- Час, що витрачається на реалізацію алгоритма. Алгоритм може бути реалізований з використанням програмних методів або апаратними засобами.
- Надійність зберігання даних на носіях або віддалених хмарних сховищах. Критерієм оцінюється, чи можуть бути відновлені дані, якщо втратиться їх частина.
- Стійкість алгоритму до помилок, які виникають при запису даних або при виконанні обчислень над ними.

Значущими параметрами для комерційної криптографії є:

1. Високий рівень захисту повідомлення під час застосування криптографічного алгоритму. Він має бути таким, щоб порушення криптографічних механізмів було фінансово невигідним. Інакше кажучи, прибуток від отриманої інформації шляхом взламу алгоритму має бути меншим ніж витрати, які використали на обчислення.
2. Вартість реалізації криптографічного алгоритму має бути вигідною.

Значно вплинула на криптографічні алгоритми поява асиметричного шифрування. Зокрема, першим алгоритмом став RSA, для шифрування в цьому алгоритмі використовуються закритий і відкритий ключі. На основі цієї технології було розроблено багато протоколів, алгоритмів цифрового підпису

та ідентифікації. Основна операція, що використовується в алгоритмі RSA і в подібних до нього алгоритмах – операція зведення в степінь.

Алгоритм RSA використовує за основу малу теорему Ферма, також застосовується узагальнення Ейлера цієї теореми. Мала теорема Ферма стверджує, якщо x менше за y та y є простим числом, виконується наступна умова:

$$x^{y-1}\ mod\ y = 1 \qquad (1.1).$$

Наводиться наступний приклад: якщо y = 13 і x = 7, тоді $7^{12}\ mod\ 13 = 1$. Тоді можна сказти, що за теоремою Ферма операція зведення до степеню за модулем y виконується в циклі і повторюється кожні y – 1 кроків. Якщо виконувати цей цикл довжиною y -1 з кроком довжиною E, тоді закритий ключ необхідно підібрати таким чином, щоб за цикл виконувалася деяка кількість кроків з довжиною E , така щоб була пройдена ціла кількість кіл і ще один крок.

Така умова при підбиранні закритого ключа необхідно для того, щоб при множенні двох ключів, утворювалася мультиплікативна інверсія за модулем y. Наприклад, якщо модуль дорівнює 13 y = 13, то значення відкритого ключа можна взяти сім E = 7.Тоді, щоб сформувати закритий ключ Q треба вирішити таке рівняння: 7*Q = 12*k + 1. Підібравши коефіцієнт k рівний чотирьом k = 4, можна обрахувати значення Q, яке дорівнює семи Q = 7. Необхідно зауважити, що значення k має бути цілим числом. Згідно з цим розв’язком, відкритий і закритий ключі асиметричного шифрування мають значення чотири і сім (k = 4, Q = 7).

Такий алгоритм асиметричного шифрування, заснований на малій теоремі Ферма, має значний недолік. Він полягає в тому, що людині, яка робить спробу порушити криптографічну безпеку даних, відоме значення модуля. Тоді задача зловмисника зводиться до закритого ключа, що є аналогічним завданням з тими, хто створюють пару закритого і відкритого ключів.

Зокрема, вразливістю алгоритму є те, що, знаючи значення модуля, стає відомим і довжина циклу y – 1 крок. Щоб зловмисник не мав змоги швидко виконати обчислення для віднаходження ключа, йому не має бути відома довжина циклу.

Таку проблему можна вирішити за допомогою узагальнення Ейлера. Це розширення малої теореми Ферма. Вважаючи, що v, w – прості числа і значення у менше ніж x, можна записати формулу узагальнення Ейлерав наступному вигляді:

$$x^{(v-1)(w-1)} mod\ v \cdot w = 1 \qquad (1.2).$$

В даному випадку, можна стверджувати, що довжина циклу не дорівнює довжині модуля:

$$L = (v-1) \cdot (w-1) \neq y = v \cdot w \qquad (1.3).$$

Через те, що зловмиснику відомий модуль y, завдання зводиться до того, щоб розкласти модуль на два множника, які є простими числами. Це, власне, задача факторизації.

Навіть при тому, що в алгоритмі симетричного шифрування буде застосована така сама розрядність модуля, алгоритм асиметричного шифрування RSA[1] виконується на 4 порядки повільніше, якщо застосовувати програмні засоби. При застосуванні апаратної реалізації швидкодія зменшується ще на 1-2 порядки.

Окрім асиметричного шифрування, операція модулярного експоненціювання широко використовується у системах криптографічно строгої ідентифікації та автентифікації віддалених абонентів. Ці системи, з математичної точки зору, базуються на незворотних перетвореннях теорії чисел. Особливо популярними стали схеми криптографічно строгої ідентифікації, які ґрунтуються на незворотних багатозначних перетвореннях класичної теорії чисел. Серед перших схем такого типу в історії була FFSIS (Feige Fiat Shamir Identification Scheme) [2]. Ця схема також базується на

розширенні малої теореми Ферма Ейлера і включає генерацію ключів, де абонент обирає два секретні прості числа p і q, а їх добуток формує модуль M=p*q. Абонент знає секретні значення обраних простих чисел p і q, що дає можливість йому генерувати правильні сеансові паролі за допомогою довжини циклу повторення кодів при модулярному експоненціюванні: (p-1)*(q-1).

Недолік цієї схеми полягає в тому, що обчислювальні процедури, передбачені нею, мають високу складність, що призводить до тривалого часу ідентифікації. Процес ідентифікації складається з багатьох циклів акредитації, кожен з яких включає обмін даними та операції модулярного множення довгих чисел з розрядністю n, яка значно перевищує розрядність процесора.

Інші відомі схеми криптографічно строгої ідентифікації, такі як схема Guillou-Quisquater [3] та Schnorr [4], мають високий рівень складності, оскільки вони також використовують операції модулярного експоненціювання в системі. Ці схеми передбачають менше число пересилок інформації для ідентифікації, але процес все ще включає обчислення A^E mod M. Усі компоненти цієї операції, включаючи показник E та модуль M, мають велику розрядність n (зазвичай n = 2048). Це призводить до значної обчислювальної складності ідентифікаційних операцій та витрат часу.

Для забезпечення автентичності і цілісності документів використовуються криптографічні механізми цифрового підпису DSS [5] та ГОСТ Р34.10-94. Ці механізми ґрунтуються на операціях модулярного експоненціювання, де основною є операція обчислення $G^Z \ mod \ M$.

Існують також інші відомі схеми криптографічно строгої ідентифікації, такі як схема Guillou-Quisquater[7] та схема Schnorr[8], які передбачають менше число пересилок ідентифікаційної інформації. Однак, ці схеми також використовують модулярне експоненціювання, а саме обчислення A^E mod M, де компоненти, такі як показник E та модуль M, мають велику розрядність (наприклад, n=2048). Це призводить до складних обчислювальних операцій ідентифікації та значних витрат часу.

Зокрема, в багатьох криптографічних системах використовується алгоритм цифрового підпису DSS (Digital Signature Standard)[9, 10], який базується на незворотних перетвореннях класичної теорії чисел з метою перевірки автентичності електронних документів.

При аналізі математичних принципів, що лежать в основі DSS, виявлено використання двох простих чисел p і q, а також числа g з певними властивостями. Процес ідентифікації включає в себе послідовні цикли акредитації, в яких здійснюються обміни даними та операції модулярного множення чисел, які мають більшу розрядність, ніж процесор.

Проведений аналіз виявив, що багато сучасних криптографічних механізмів, які використовуються для захисту інформації, ґрунтуються на модулярному множенні та модулярному експоненціюванні з числами значно більшої довжини, ніж розрядність процесора. Однак, найбільш серйозною проблемою ефективності цих механізмів є їх повільна комп'ютерна реалізація через високу обчислювальну складність зазначених мультиплікативних операцій модулярної арифметики. Отже, для поліпшення ефективності цих криптографічних механізмів необхідно шукати шляхи прискорення виконання мультиплікативних операцій.

1.2 Огляд методів прискореного множення

Аналіз показав, що більшість реальних алгоритмів криптографічного захисту, які ґрунтуються на незворотних задачах теорії чисел, використовують незворотні перетворення у формі модулярного експоненціювання. Це означає обчислення виразу $A^E \bmod m$, де A, E і m - цілі числа. Виконання цієї операції вимагає пройдення через n циклів, де n - розрядність числа m. Залежно від порядку, в якому аналізуються розряди експоненти E, можна виділити два основних алгоритми модулярного експоненціювання [11]:

- За першим способом виконується перебір розрядів, починаючи з молодшого біту.
- За другим варіантом перебір розрядів реалізується, починаючи зі старшого біту.

Значення модулярного множення викликає активні дослідження з прискорення ресурсоємких обчислень. Існують два основних класи методів для обчислення операції модулярного множення $X \cdot Y \ mod \ Z$. У першому класі, множення та редукція добутку виконуються послідовно, у другому класі - одночасно.

Методи, які виконують операції множення і редукції роздільно такі: метод Карацуби [12], метод множення Фюрера [13], метод множення Toom-Cook [14,15].

До першого класу відноситься класичний метод модулярного множення [16]. Класичний метод модулярного множення є основною операцією в криптографічних протоколах. Він використовується для отримання залишку від ділення добутку двох чисел на модуль.

Метод класичного модулярного множення[17] можна розділити на кілька етапів. Нехай G та H - множники, а M - модуль. Основна ідея полягає у послідовному обчисленні добутку $F = G \cdot H$, а потім виконанні редукції добутку F за модулем M.

1. Обчислення добутку: Спочатку ми обчислюємо добуток $F = G \cdot H$. Це може бути здійснено за допомогою звичайного множення чисел X та Y.
2. Редукція добутку: Після обчислення добутку F, потрібно виконати редукцію, щоб отримати залишок від ділення F на модуль M. Це здійснюється шляхом віднімання від F кратного модулю M, доки F не стане менше за M. Формула для редукції може мати вигляд:

$R = F - k \cdot M$,

де k - кратне модулю M, щоб забезпечити, що R буде меншим за M.

3. Остаточний результат: Після виконання редукції, R буде представляти залишок від ділення добутку $G \cdot H$ на модуль M. Це є остаточним результатом класичного модулярного множення.

Важливо зазначити, що класичний метод [17] модулярного множення може бути обчислювально витратним для великих чисел, особливо якщо розрядність чисел значно перевищує розрядність процесора. Тому виникає потреба в розробці ефективних алгоритмів модулярного множення, таких як метод Баррета або метод Монтгомері, які забезпечують оптимізацію та прискорення цього процесу.

Один з методів першого класу - метод Баррета [18], виконує прискорену модулярну редукцію $A \ mod \ Z$. Цей метод віднімає результат цілочисельного ділення добутку A на модуль Z від самого добутку. Проте, основним недоліком цього методу є складність операції ділення.

У зв'язку з цим, бажано замінити операцію ділення на операцію зсуву. Застосування методу Баррета у криптографічних системах з використанням числення з основою b = 2 дозволяє попередньо розрахувати складову $b2 \cdot k \cdot Z$ перед початком обчислень.

Однак, аналіз формули (2) показує, що час виконання редукції Баррета мінімум вдвоє перевищує час операції множення $2n$-розрядних чисел і може бути ефективним лише з використанням ефективних методів множення.

Найшвидший метод модулярного множення, який відноситься до другого класу, був запропонований Монтгомері [19, 20]. Доведено , що час виконання цього методу складає $T_{\text{М}} = 2 \cdot n \cdot t$, де t - час виконання однієї операції (додавання або зсуву). Алгоритм модулярного множення за методом Монтгомері включає наступну послідовність дій:

1. Встановлення значення результату та лічильника R = 0, i = 0. Множник Y представлений у двійковій системі числення, наприклад, $Y = 2^{n-1} \cdot y^{n-1} + 2^{n-2} \cdot y_{n-2} + \cdots + 2 \cdot y_1 + y_0$.

2. Якщо поточний розряд множника yi дорівнює одиниці, то до проміжного результату додається множене X.
3. Якщо найменший розряд проміжного результату дорівнює одиниці, то до нього додається модуль Z.
4. Проміжний результат зсувається на один біт вправо R >> 1.
5. Якщо значення лічильника i < n - 1, де n - розрядність множника, то інкрементується значення лічильника і переходиться до кроку 2.
6. Якщо результат перевищує модуль R > Z, то від проміжного результату віднімається модуль: R = R - Z.
7. Остаточний результат, отриманий за методом Монтгомері, отримується шляхом обчислення $R = R \cdot 2^n \ mod \ Z$.

Проведений аналіз існуючих методів показує, що вони не використовують повний потенціал для підвищення швидкодії операції модулярного множення.

ВИСНОВОК ДО РОЗДІЛУ 1

Результатом проведених досліджкень, які провелися в першому розділі диплому і які направлені на застосування операцій модулярної арифметики в сучасних методах криптографічних системах захичту даних, стали наступні висновки:

1. Проаналізувавши сучасний арсенал криптографічних методів захисту даних, можна сказати, що в них застосовуються операції модулярного множення й експоненціювання. Операції ці проводяться над числами великої розрядності, зокрема довжина модуля може становити в сучасних системах 2048 бітів. Через це програмна реалізація цих алгоритмів виконується повільно. Тому, швидкодію таких методів можна підвищити за рахунок скорочення часу обчислень операцій модулярного множення і експоненціювання.
2. Виконаний аналіз протоколів, які засновані на криптографічних асиметричних алгоритмах, зокрема такі алгоритми як El-Gamal, RSA, DSS (алгоритм, що використовується для генерування цифрового підпису). Цей аналіз демонструє, що ключі цих асиметричних алгоритмів змінюють дуже рідко через те, що відкритий ключ ідентифікує користувачів системи. Відповідно, модуль *M,* що використовується в цих криптографічних алгоритмах, можна вважати сталою величиною і завдяки цьому можна зменшити обчислювальну складність операцій модулярного експоненціювання й множення.
3. З попереднього аналізу вже існуючих алгоритмів випливає, що операція модулярного експоненціювання не може бути розпаралелена через те, що дії алгоритму виконуються строго послідовно. Тоді єдиним способом для прискорення цієї операції є прискорення частини цього алгоритму, зокрема операції модулярного множення.

4. Проаналізувавши метод секційного множення, можна дійти висновку, що при застосуванні декількох процесорів для кожної з частин, кількість операцій може бути скорочена.
5. З вище розглянутих методів, що направлені на ефективне виконання модулярного множення можна виділити метод Монтгомері. Основною перевагою цього методу є те, що редукція результату і множення робляться одночасно, що дозволяє досягти найбільшої швидкодії.

РОЗДІЛ 2
РОЗРОБКА МЕТОДУ ПРИСКОРЕНОГО МУЛЬТИПРОЦЕСОРНОГО МОДУЛЯРНОГО МНОЖЕННЯ З ВИКОРИСТАННЯМ ГРУПОВОЇ РЕДУКЦІЇ МОНТГОМЕРІ

Базовою операцією в багатьох криптограічних задачах (симетричне шифрування, технологія цифрового підпису та інші) є операції модулярної ариффметики. Ці операції виконуються над числами розрядністю 2048 або 4096. Використання такої розрядніості передбачено більшістю сучасних протоколів шифрування. Щоб зламати такі алгоритми треба використовувати технологію факторизації, тобто розкласти модуль на два простих числа. Ресуркоємкість даного завдання повністю залежить від кількості бітів в модулі. З цього випливає, що для того, щоб підвищити рівень надійності аглоритмів з відкритим ключем існує потреба збільшити довжину модуля. Зараз існують технології хмарних обчислень, які дозволяють спростити задачу факторизації, скоротити час на її вирішення. Час, що витрачається на розклад модуля на прості числа, все ще є тривалим. Однак з кожним роком обчислювальні ресурси збільшуються, що, очевидно, робить алгоритми з відкритим ключем більш вразливими до зламу тому, що час на їх взлам зменшується. Проблему можна вирішити збільшивши розрядність модуля, таким чином рівень захищеності підвищиться. Однак це призведе до нових труднощів, що пов’язані з критичним часом обчислення модулярних операцій. Наприклад, якщо довжина модуля збільшиться у два рази , то кількість тактів, що виконується при обчисленнях, збільшиться у 8 разів.

Модулярне експоненціювання має чітку послідовність операцій для отримання кінцевого результату, тому обчислення не можуть бути розділені на менші підзадачі, які виконується одночасно і незалежно одне від одного. Але можна прискорити операцію модулярного множення, що є частиною операції модулярного експоненціювання.

Обчислення модулярного множення зводиться до двох основних операцій: множення бітів множеного на множник та редукції отриманого результату. Існує багато методів модулярної редукції. Одним із найбільш швидкодіючих методів модулярної редукції є метод Монтгомері, де на відміну від інших методів операції множення і редукції виконуються одночасно.

2.1 Розробка методу мультипроцесорного модулярного множення

Можна виділити дві варіації алгоритму модулярного експоненціювання. Вони відрізняються різним порядком перебору бітів експоненти.

Якщо змінна X підноситься до степені експоненти E, тоді перша варіація може бути представлена наступними кроками:

1. Змінна R прирівнюється до одиниці, лічильник і набуває значення нуля.
2. За умови, що і-ий біт експоненти дорівнює одиниці, значення поточного результату множиться на X.
3. Змінна результату підноситься до другої степені.
4. При умові, що значення лічильнику і менше за довжину експоненти, до лічильника додається одиниця і виконується перехід до кроку два.

Проаналізувавши цей метод, можна зробити висновок, що модулярне експоненціювання представляє собою операції модулярного множення та зведення числа до другої степені. Виконання цих двох операцій триває майже однаковий час на комп’ютері, тому можна сказати, що більшість часу виділяється на модулярне множення. І якщо прискорити цю частину обчислень, то час модулярного експоненціювання суттєво зменшиться.

Модулярне множення можна здійснити різними підходами. Один із них передбачає отримання добутку двох чисел, а потім реалізацію операцію ділення за модулем. За іншим, операції множення і редукції виконуються разом на кожному такті алгоритму.

Алгоритм Монтгомері, який відповідає другому підходу і має більшу швидкодію в порвняні з іншими відомими алгоритмами, є основою запропонованого методу, тому доцільно описати послідовність дій, для його реалізації. Враховуючи, що G та H, відповідно, множене і множник розрядністю n, а M модуль, запишемо алгоритм:

1. Створюємо змінну R, якій присвоюємо значення 0. В цій змінній буде зберігатися результат модулярного множення. Лічильник і також приймає нульове значення.
2. Якщо біт множника H під номером і дорівнює одиниці, то до результату (до змінної R) додається значення множеного G.
3. Якщо молодший біт поточного результату (змінної R) дорівнює одиниці, то до результату додається ще значення модуля M.
4. На цьому кроці виконується зсув поточного результату R на один біт вправо.
5. Якщо значення лічильника і менше розрядності множника H, то значення лічильника збільшується на одиницю і виконується повернення на другий крок алгоритму.
6. Якщо значення поточного результату більше за модуль, здійснюємо операцію за модулем.
7. Обраховується справжній результат молулярного множення домноженням поточного результату на двійку в степені n та модулярним діленням на M.

Слід зазначити, що перебір бітів множника відбувається, починаючи з молодшого розряду.

На рисунку 2.1 зображена блок-схема алгоритму модулярного множення Монтгомері.

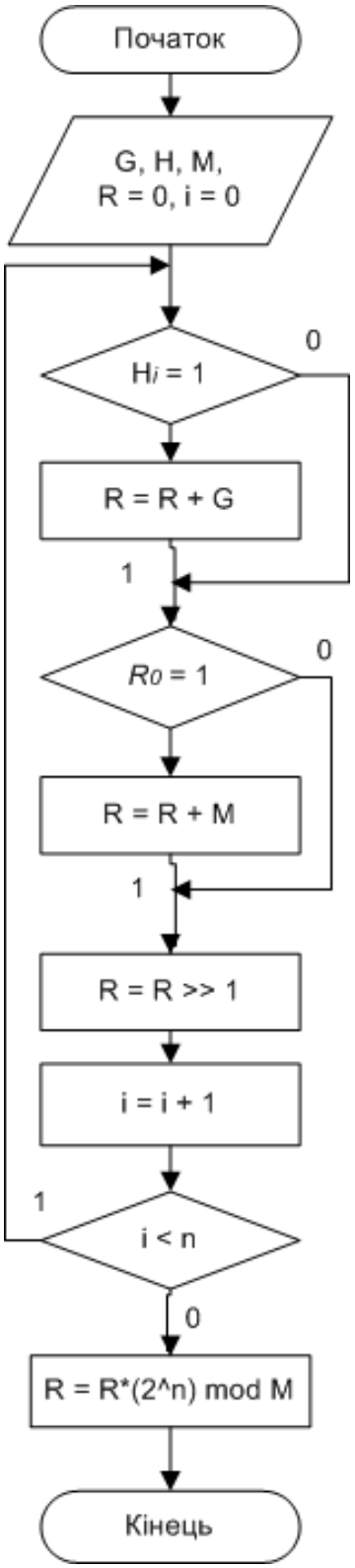


Рисунок.2.1 - Метод модулярного множення Монтгомері

Наводиться приклад, що ілюструє модулярне множення за методом Монтгомері для таких чисел G = 3036, H = 3291, M = 9030059.

3036*3291 mod 9030059 = 961417

Таблиця 2.1 - Приклад модулярного множення з методом Монтгомері при G=3036, H=3291, M=9030059

H_i	Додавання G	R_0	Додавання M	Зсув впараво
1	0 + 3036 = 3036	0	-	3036/2 = 1518
1	1518 + 3036 = 4554	0	-	4554/2 = 2277
0	-	1	2277+9030059 = 9032336	9032336/2=4516168
1	4516168+3036= 4519204	0	-	4519204/2= 2259602
1	2259602+3036= 2262638	0	-	2262638/2= 1131319
0	-	1	1131319+9030059=10161378	10161378/2= 5080689
1	5080689+3036= 5083725	1	5083725+9030059=14113784	14113784/2= 7056892
1	7056892+3036 = 7059928	0	-	7059928/2= 3529964
0	-	0	-	3529964/2= 1764982
0	-	0	-	1764982/2 = 882491
1	882491 + 3036 = 885527	1	885527 + 9030059 = 9915586	9915586/2= 4957793
1	4957793+3036 = 4960829	1	4960829+9030059=13990888	13990888/2= 6995444

Отриманий поточний результат перетворюємо на кінцевий: R=6995444*(2^12) mod 9030059 = 961417.

Ціллю вивчення є прискорення операції модулярного множенням за рахунок поділення множника на часткові множники і виконання над ними групової редукції Монтгомері.

Одним із дієвих способів організувати прискорення модулярного множення є розпаралелювання його обчислень [21].

Щоб реалізувату дану мету висувається пропозиція модифікувати метод Монтгомері таким чином, щоб покращити його швидкодію.

Модифікація полягає в наступному:

1. Побудувати таблицю передобчислень для здійснення групової редукції Монтгомері для заданого модуля і кількості бітів для одночасної обробки.
2. Розділити множник H на часткові множники.
3. Модулярний добуток множеного і кожного часткового множника виконується на окремому процесорі одночасно.
4. Групова редукція Монтгомері починає виконуватися на тій частині часткогвого множника, де всі біти від поточного до молодшого дорівнюють нулю.
5. Групова редукція виконується над групою чотирьох бітів за один раз.

Оскільки модуль в більшості криптографічних системах є постійною величиною доцільно завчасно побудувати таблицю передобчислень [22] один раз на початку програми.

Алгоритм запрпонованого методу зображений на малюнку 2.2.

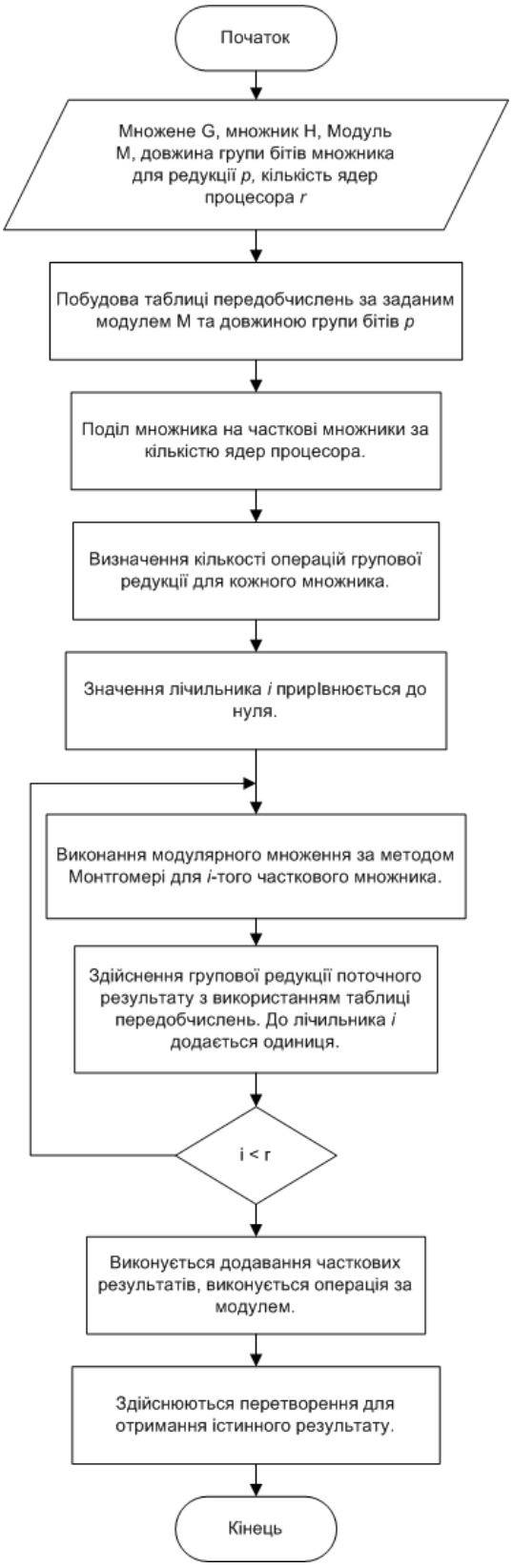


Рисунок 2.2 - Алгоритм прискореного модулярного множення

Якщо говорити про таблицю передобчислень, то її основними параметрами є молодші n розрядів поточного результату та значення добутку модуля помножене на деяку константу. Технологія побудови таблиці передобчислень така, що при додаванні добутку модуля та константи до поточного результату, останні n бітів поточного результату обнулюються, що дозволяє нам виконати зсув вправо на n розрядів. Такий підхід дозволить виконати один такт з операцією додавання і зсуву замість *n*.

Формування таблиці передобчислень відбувається наступним чином:

1. Визначається довжина *p* групи бітів над якими виконується модулярна редукція. Створюється словник, ключі якого – варіанти останніх *p* розрядів поточного результату, а значення – добуток модуля M на деяку константу. Створюється масив чисел array_d від нуля до 2^p і масив array_M добутків модуля та константи.
2. Значеннння лічильника *i* прирівнюється до нуля.
3. До масиву array_M додається добуток *i*M.*
4. До масиву array_d додається значення *i.*
5. До лічильника додається 1, якщо його значення менше ніж 2^d, здійснюється перехід до пункту 3.
6. Значеннння лічильників *i, j* прирівнюються до нуля.
7. Якщо останні *p* розрядів суми значень array_d[i] і array_M[j] мають нульове значення, то до таблиці передобчислень додається ключ array_d[i] і значення array_M[j].
8. Лічильник *j* інкрементується на одиницю, якщо його значення менше довжини масиву модулів, виконується повернення на крок сім.
9. Лічильник *i* інкрементується на одиницю, якщо його значення менше довжини масиву array_d, виконується повернення на крок сім.

Алгоритм поділу множника на часткові множники описується нижче:

1. Створюється пустий масив індексів розмірів часткових множників sizes і масив часткових множників parts.

2. Лічильник *i* прирівнюється нулю.
3. Значення лічильника і додається до масиву sizes.
4. До лічильника *i* додається значення довжини множника поділене на кількість ядер процесора *r*.
5. Якщо значення лічильника менше ніж довжина множника здійснюється перехід до кроку три.
6. Лічильник *i* прирівнюється до нуля.
7. Розряди множника H під номерами від sizes[*i*] до sizes[*i* + 1] утворюють частковий множник, який додається до масиву часткових множників parts.

Ураховуючи наявні ресурси пам'яті для зберігання таблиці переобчислень, кількість множень, які можна суміщувати в часі, може складати шістнадцять. За таких умов, число значущих розрядів множника *H*, що обробляються на кожному з *r* процесорних ядер, може бути визначене. Організація паралельної мультипроцесорної обробки з об'єднанням часткових результатів дозволяє реалізувати фазу залишкової редукції з групуванням, тобто шляхом комбінування декількох ітерацій модулярної редукції Монтгомері за одну ітерацію.

Організацію роботи запропонованого методу можна описати наступними пунктами:

1. Змінна R прирівнюється до нуля. В ній зберігається результат модулярного множення. Створюється масив часткових результатів part_results. Лічильник m дорівнює нулю. За допомогою алгоритму поділу множника на часткові множники отримується їх масив parts. Формується таблиця передобчислень згідно з вище наведеним алгоритмом.
2. Лічильник b прирівнюється до нуля.
3. Поточний результат res = 0.
4. Лічильник *i* приймає нульове значення.

5. Якщо біт часткового множника parts[b] під номером *i* дорівнює одиниці, то до змінної res додається значення множеного G.
6. Якщо молодший біт поточного результату res дорівнює одиниці, то до результату додається значення модуля M.
7. Виконується зсув поточного результату res на один біт вправо.
8. Якщо значення лічильника *i* менше розрядності часткового множника parts[b], то значення лічильника збільшується на одиницю і виконується повернення на п’ятий крок алгоритму.
9. Якщо значення поточного результату більше за модуль, здійснюємо операцію за модулем.
10. Значення лічильника *i* прирівнюється до нуля.
11. Якщо m = 0, до m і до b додається одиниця і здійснюється перехід до кроку три.
12. За останніми *p* розрядами поточного результату в таблиці передобчислень знаходиться добуток модуля і константи і додається до поточного результату.
13. Виконується зсув поточного результату на значення *p*.
14. Якщо значення поточного результату більше за модуль, здійснюємо операцію за модулем.
15. До *i* додається одиниця, якщо значення лічильника *i* менше за *m*, виконується перехід до кроку дванадцять.
16. Поточний результат додається до масиву part_results.
17. До лічильника b додається одиниця. Якщо значення b менше ніж довжина масиву часткових множників parts, здійснюється повернення до пункту три.
18. Обраховується сума часткових результатів з масиву part_results. Якщо значення поточного результату більше за модуль, здійснюємо операцію за модулем.

19. Обраховується справжній результат домноженням поточного результату на двійку в степені n та взяттям за модулем M.

Проілюструємо працездатність запропонованого методу наступним прикладом. Нехай множене G = 3036, множник H = 3291, модуль дорівнює 9030059 M = 9030059. Таким чином, розрядність модуля буде перевищувати довжину множника у два рази.

Першим кроком відбувається формування таблиці передобчислень. Нехай кількість розрядів над якими одночасно виконується групова редукція становить чотири. Будуємо таблицю таким способом, аби при додаванні двох компонет таблиці з однієї строки утворився би результат, останні чотири біти якого мають нульове значення. Після побудови таблиці можна використовувати її у фазі редукції. Таблиця передобчислень сформована і представлена таблицею 2.2.

Таблиця 2.2 - Приклад таблиці передоблислень для M = 90300590 і n = 4

$h_3h_2h_1h_0$	c*M
0000	0
0001	117390767
0010	90300590
0011	63210413
0100	36120236
0101	9030059
0110	126420826
0111	99330649
1000	72240472
1001	45150295
1010	18060118
1011	135450885
1100	108360708
1101	81270531
1110	54180354
1111	27090177

Другим кроком розділяється множник H, в двійковій системі представлений як $H = 110011011011_2$ на часткові множники. Поділ множника [23] відбувається так, щоб кожен частковий множник мав однакову довжину з початковим множником. Це необхідно, бо старші нульові розряди часткових множників також обробляються і використовуються для редукції часткових результатів. Припустимо, що кількість процесорів, які задіяні для паралельної обробки часткових множників наявно три, тоді створюються часткові множники h1, h2 та h3:

$h1 = 1100\ 0000\ 0000_2$

$h2 = 0000\ 1101\ 0000_2$

$h3 = 0000\ 0000\ 1011_2$

Третім кроком обрахуємо часткові результати R1, R2, R3 модулярних множень h1, h2 та h3 на множене G.

Послідовність обчислень часткового результату R1 = h1 * G показана в таблиці 2.3. R1 = h1 * G = 2277.

Таблиця 2.3 - Послідовність обчислень часткового результату R1 = h1 * G

$h1_i$	Додавання G	$R1_0$	Додавання c*M	Зсув вправо
0	-	0	-	-
0	-	0	-	-
1	0 + 3036 = 3036	0	-	3036 >> 1 = 1518
1	1518 + 3036 = 4554	0	-	4554 >> 1 = 2277

Послідовність обчислень часткового результату R2 = h2 * G показана в таблиці 2.4. R2 = h2 * G = 4374089.

Таблиця 2.4 - Послідовність обчислень часткового результату R2 = h2 * G

$h2_i$	Додавання G	$R2_0$	Додавання M	Зсув вправо
1	0 + 3036 = 3036	0	-	3036/2 = 1518
0	-	0	-	1518 >> 1 = 759
1	-	1	2277+9030059 = 9032336	9032336>>1= 4516168
1	4516168+3036= 4519204	1	4519963 + 9030059 = 13550022	13550022 >> 1 = 6775011
Виконання групової редукції				
$R2_{3210}$	Додавання c*M			
0011	6775011 + 63210413 = 69985424			69985424 >> 4 = 4374089

Послідовність обчислень часткового результату R3 = h3 * G показана в таблиці 2.5. R3 = h3 * G = 2619078.

Таблиця 2.5 Послідовність обчислень часткового результату R3 = h3 * G

$h3_i$	Додавання G	$R3_0$	Додавання M	Зсув вправо
1	0 + 3036 = 3036	0	-	3036/2 = 1518
1	1518 + 3036 = 4554	0	-	4554 >> 1 = 2277
0	-	1	2277+9030059 = 9032336	9032336>>1= 4516168
1	4516168+3036= 4519204	0	-	4519204 >> 1 = 2259602
Виконання групової редукції				
$R3_{3210}$	Додавання c*M			
0010	2259602 + 90300590 = 92560192			92560192 >> 4 = 5785012
0100	5785012 + 36120236 = 41905248			41905248 >> 4 = 2619078

Щоб отримати остаточний результат обчислень, необхідно додати три часткові результати. R = R1 + R2 + R3 = 2277 + 4374089 + 2619078 = 6995444. Отриманий результат не більший ніж модуль, тому віднімати від нього значення модуля не потрібно.

Щоб компенсувати всі корекції модулярного множення під час яких відбувався зсув результату вправо, і відповідно його зменшення, треба домножити отриманий результат на двійку в степені значення довжини основного множника.

Перетворимо результат на істинний. Для цього домножимо його на двійку в степені дванадцять (значення довжини множника) і візьмемо за модулем R=R*(2^12) mod M = 6995444*(2^12) mod 9030059 = 961417. Отриманий справжній результат модулярного множення R = 3036*3291 mod 9030059 = 961417.

2.2 Реалізація методу на кількох процесорах комп'ютера. Оцінка ефективності.

Щоб оцінити ефективність розробленого методу, необхідно обрахувати коефіцієнт прискорення α. Він розраховується, як відношення часу T_M здійснення модулярного множення з використанням методу Монтгомері до часу T виконання обрахунків із застосуванням розробленого методу:

$$\alpha = \frac{T}{T_M} \qquad (2.1).$$

Для того, щоб розрахувати час модулярного множення необхідно кількість виконаних операцій помножити на час виконання кожної з них. З алгоритмів виконання методу Монтгомері і запропонованого методу, видно, що в них виконуються дві основні операції – це додавання до поточного результату множеного чи модуля та зсув вправо. Позначимо час виконання операції додавання як t_d, а час зсуву як t_z. Виконання цих операцій для комп'ютера

рівнозначно за складністю, тому час обчислень цих операцій однаковий, тоді $t_d = t_z$.

Розглянемо швидкодію метода Монтгомері. Ймовірність того, що в одному такті виконується операція додавання множеного дорівнює 0.5, така ж ймовірність додавання модуля до поточного результату, так як в одному такті завжди виконується операція зсуву на один біт вправо, то ймовірність зсуву дорівнює одиниці. Підсумовуючи, в середньому за один такт виконується 2 операції. Враховуючи, що кількість тактів дорівнює кількості бітів множника H, то сумарна кількість операцій для обчислення модулярного множення за методом Монтгомері складає 2*n. Якщо цю кількість операцій помножити на час виконання операції додавання, то час модулярного множення за методом Монтгомері можна записати наступною формулою:

$$T_M = 2*n* t_d \qquad (2.2).$$

В розробленому методі модулярне множення кожного часткового множника на множене виконується паралельно на окремому процесорі.

Швидкодія запропонованого методу досягається за рахунок поділення множника на окремі частини, а їх обробка виконується паралельно на різних процесорах. Для розпаралельвання використовуються *r* ядер багатоядерного процесора комп'ютера.

В запропоновоному методі множник ділиться на частини з використанням секційного поділу. Більшість процесорів мають засоби для реалізації прискореного множення [24]. Частина розрядів, які послідовно йдуть одне за одним, зберігають свої значення, а інші біти множника обнулюються. Потім береться наступна група розрядів, а інші обнулюються. Так формується декілька часткових множників. Організувати поділ множника можна й іншим способом так, щоб часткові множники мали однакову кількість значущих бітів. Але такий поділ дає незначного прискорення.

Щоб досягти найбільшого прискорення модулярного множення пропонується об'єднати групову редукцію та паралельне виконання

модулярного множення на багатьох процесорах. Паралельне виконання можливе за рахунок програмних засобів вибраної мови програмування, що підтримує багатопоточне виконання, та апаратних засобів так, як майже всі сучасні комп'ютера мають декілька ядер для виконання задач.

Найбільша кількість операцій виконується над третім частковим множником. Враховуючи це, обчислювати час модулярного множення запропонованим методом обраховується за часом обробки третього часткового множника. В середньому розряди множника, які оброблюються за методом Монтгомері, потребують виконання двох операцій за один такт, кількість цих розрядів становить $\frac{n}{r}$. Інші розряди, кількість яких складає $n - \frac{n}{r}$ обробляються з використанням 0.5 операцій за такт.

Якщо скласти кількість операцій виконаних під час обробки розрядів множника за методом Монтгомері і з використанням групової редукції і помножимо на час виконання цих операцій t_d, тоді загальний час T виконнаня модулярного множення записується і наступному вигляді:

$$T = (2\frac{n}{r} + 0.5(n - \frac{n}{r}))*t_d \quad (2.3).$$

Оцінка швидкодії розробленого метода проілюстрована графіками на малюнку 2.1.

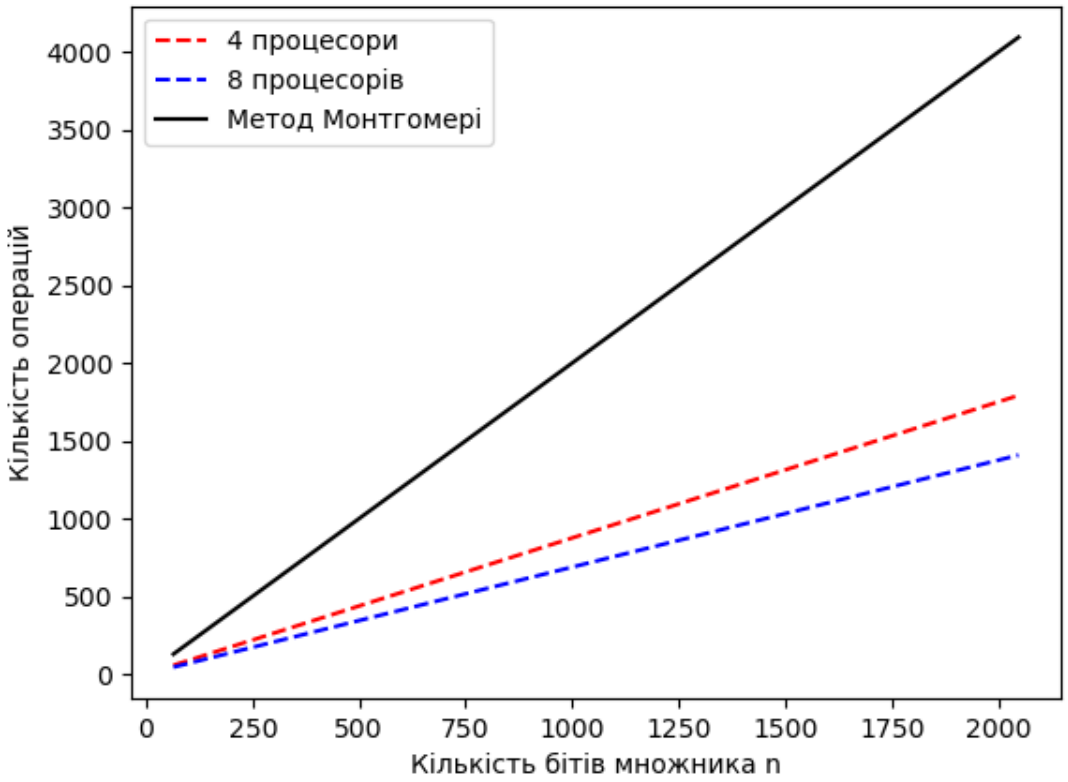


Рисунок 2.3 - Залежність кількості виконаних операцій множником під час модулярного множення від довжини множника

З таблиць обчислень часткових модулярних добутків видно, що найбільша кількість обчислень виконується над третім множником (таблиця 2.5) і становить дванадцять операцій, в той самий час, кількість операцій для модулярного множення за методом Монтгомері [25] з таблиці 2.1 складає двадцять п’ять операцій, що в два рази більше ніж в запропонованому методі. Враховуючи, що обчислення часткових результатів будуть обраховуватися паралельно на різних процесорах комп’ютера, тоді швидкодія у порівнянні з методом Монтгомері значно збільшиться.

Основний потенціал для збільшення швидкодії модулярного множення – виконання обчислень паралельно з використанням декількох потоків (ядер комп’ютера).

Для обчисленння модулярного добутку на k процесорах кількість бітів множника, які оброблюються на першому процесорі - h_1, на другому - h_2, на

останньому k-тому процесорі - h_k. При цьому, без втрати загалу, можна вважати, що молодші h_1 розряди множника оброблюються на першому процесорі, наступні h_2 оброблюються на другому процесорі, і так далі. Відповідно, старші h_k розряди множника обробляються на k-тому процесорі. Зрозуміло, що $h_1 + h_2 + \cdots + h_k = n$.

Максимальний ефект прискорення модулярного множення при його реалізації на k процесорах досягається при повній завантаженості кожного із процесорів, тобто за умови, що час виконання операцій на кожному із процесорів приблизно однаковий. Час виконання t_j операцій на j-тому процесорі, $j \in \{1,2,\dots,k\}$ складається з двох частин: h_j раз виконується цикл множення з редукцією та $(h_{j+1} + h_{j+2} + h_k)$ на j-ий частковий множник та редукції результату і визначається наступною формулою:

$$t_j = h_j \cdot t_{cm} + (n - \sum_{i=1}^{j} h_j) \cdot \frac{t_{cr}}{q}, \quad (2.4).$$

де t_{cm} – середній час виконання циклу множення і редукції, t_{cr} – середній час виконання циклу редукції, q – кількість розрядів, над якими одночасно виконується операція редукції.

Як зазначалося вище, за умови однакової завантаженості кожного із процесорів,час виконання операцій на кожному з них однаковий, тобто виконується наступна умова: $t_j = t_{j+1}$, яка може бути представлена наступним чином:

$$h_j \cdot t_{cm} + \left(n - \sum_{i=1}^{j} h_j\right) \cdot \frac{t_{cr}}{q} = h_{j+1} \cdot t_{cm} + \left(n - \sum_{i=1}^{j} h_j\right) \cdot \frac{t_{cr}}{q} - \frac{t_{cr}}{q} \cdot h_{j+1}, \quad (2.5).$$

Після скорочень однакових компонентів у лівій та правій частинах, умова (9) може бути представлена у більш компактному вигляді:

$$h_j \cdot t_{cm} = h_{j+1} \cdot t_{cm} - \frac{t_{cr}}{q} \cdot h_{j+1}, \quad (2.6).$$

З умови (2) може бути визначене співвідношення між кількістю розрядів в $(j+1)$-му та j-му часткових множниках:

$$\frac{h_{j+1}}{h_j} = \frac{t_{cm}}{t_{cm} - \frac{t_{cr}}{q}} = \gamma \quad (2.7).$$

Чисельне значення кількості h_1 двійкових розрядів у першому частковому множнику можна визначити із наступного рівняння:

$$h_1 + \beta \cdot h_1 + \beta^2 \cdot h_1 + \cdots + \beta^{k-1} \cdot h_1 = n, \quad (2.8).$$

В підсумку, значення h_1 – кількості бітів молодшого часткового множника визначається наступним чином:

$$h_1 = \frac{n}{\sum_{i=1}^{j} \beta^i} \quad (2.9).$$

Цілком очевидно, що сума у виразів (4) являє собою суму кінцевої геометричої прогресії зі значенням $\beta > 1$. Підставивши значення вказаної суми у вираз (6), можна отримати кінцеву формулу для визначення h_1 – кількості двійкових розрядів моодшого часткового множника:

$$h_1 = \frac{n \cdot (\beta - 1)}{\beta^k - 1}, \quad (2.10).$$

Для ефективного застосування запропонованого методу прискореного обчислення модулярного множення на декількох ядрах важливо визначити розрядність h_j кожного часткового множника.

В таблиці 2.6 наведені результати теоретичних обчислень h_1, h_2, h_3, h_4, обраховані за формулою (4) а також коефіцієнт прискорення β, розрахунки якого здійснюються за виразом (4). Обчислення здійснены для чотирьохядерного процесора (k = 4) і розрядності чисел n = 2048, що використовується в значній частині криптографічних систем захисту даних. Проведені обчислення виконані для різної кількості розрядів q, що одночасно обробляються процесором.

Таблиця 2.6 - Теоретичне визначення розрядності h_j для чотирьохядерного процесора

q	h_1	h_2	h_3	h_4	β
4	365,1	449,3	553,0	680,6	3,0
8	439,1	484,5	534,6	589,9	3,5
16	475,7	499,1	523,7	549,4	3,7
32	493,9	505,8	517,9	530,4	3,9

Теоретичні значення h_j мають десятковий формат, очевидно, що розрядність має бути цілим числом. Оскільки редукція h_j-тих розрядів виконується групами по q бітів доцільно привести h_j до значення, що кратне q. З урахуванням цього, округлення теоретичних результатів здійснюється наступною послідовністю дій: величини, починаючи з h_k і закінчуючи h_2 округлюються до найближчого цілого числа, що кратне q, а h_1 обраховується, як різниця розрядності процесора й суми всіх попередньо округлених h_j: $h_1 = n - \sum_{j=2}^{k} h_j$. Практичні результати обчислень наведені в таблиці 2.7.

Табл.2.7 - Практичне визначення розрядності h_j для чотирьохядерного процесора

q	h_1	h_2	h_3	h_4	β
4	368	448	552	680	3,0
8	432	488	536	592	3,5
16	480	496	528	544	3,8
32	480	512	512	544	3,8

Аналогічні результати обчислень для восьмиядерного процесора(k = 8) наведені в таблиці 2.8.

Табл.2.8 - Теоретичне визначення розрядності h_j для восьмиядерного процесора

q	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	β
4	110,8	136,4	167,9	206,6	254,3	312,9	385,2	474,0	4,3
8	176,9	195,2	215,3	237,6	262,2	289,3	319,2	352,3	5,8
16	215,1	225,7	236,8	248,4	260,6	273,5	286,9	301,0	6,8
32	235,3	240,9	246,7	252,6	258,7	264,9	271,2	277,7	7,4

Для восиядерного процесора довжини наведені в таблиці 2.9.

Табл.2.9 - Практичне визначення розрядності h_j для восьмиядерного процесора

q	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	β
4	108	136	168	208	256	312	384	476	4,3
8	176	192	216	240	264	288	320	352	5,8
16	208	224	240	256	256	272	288	304	6,7
32	224	256	256	256	256	256	256	288	7,1

Також наводяться значення для шістнадцяти процесорів в таблиці 2.10.

Табл.2.10 - Практичне визначення розрядності h_j для шістнадцятирозрядного процесора

	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	betta
q = 4	24	20	28	32	40	48	60	76	92	116	140	172	212	264	324	400	5,12
q = 8	64	64	64	72	80	88	96	112	120	136	152	160	184	200	216	240	8,5
q = 16	80	96	96	96	112	112	112	128	128	128	144	144	160	160	176	176	11,6
q = 32	96	96	96	128	128	128	128	128	128	128	128	128	128	160	160	160	12,8
q = 64	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	16
q = 128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	16

Аналіз наведених результатів показує, що зі збільшенням кількість розрядів q, над якими одночасно виконується операція редукції, до 32 бітів,

коефіцієнт прискорення β наближається до значення кількості ядер процесора, на яких виконуються розрахунки.

Отже, можна стверджувати, з урахуванням вище зроблених обрахунків коефіцієтів ефективності, що розроблений метод прискорення модулярного множення із застосуванням багаторозрядних чисел надає можливість пришвидшити обчислення цієї базової операції криптографічних алгоритмів.

Звачаючи на таблиці, де продемонстровано залежність довжини групи одночасно розроблюваних розрядів та кількость застосованих процесорів від коефіцієнту прискорення, можна помітити, що найбільша швидкодія отримується при однаковій навантаженості кожного ядра процесора комп'ютера. Однак, при цьому значне збільшення ядер процесору, що задіяні для реалізації запропонованого методу, не призводить до поліпшення часу здійснення обчислень. Іншими словами, існує границя значення кількості ядер, і якщо перевищити це значення, то за її межами коефіцієнт прискорення вже не буде збільшуватися.

З останньої раблиці передобчислень видно, що при збільшенні значення довжини групи одночасно оброблюваних розрядів, коефіцієнт ефективності залишається незмінним.

ВИСНОВОК ДО РОЗДІЛУ 2

В другому розділі бакалаврської роботи були проведені дослідження, направленні на пошук і розробку методу прискорення швидкодії операції модулярного множення, яка застосовується в криптографічних алгоритмах цифрового підпису, криптографічних системах з відкритим ключем, криптографічних протоколах. Результати цих досліджень можуть бути представлені наступними пунктами:

1. Основним потенціалом для прискорення операції модулярного множення є використання передобчислень в частині здійснення обчислень модулярної редукції. Перспективно. є можливість виконувати одночасно операцій модулярного множення і модулярної редукції з використанням обчислень одночасно. Це, в свою чергу, дає можливість скоротити час здійснення цих операцій і усунути роботу з багаторозрядними числами, що за розміром більше ніж довжина модуля.
2. Розроблено метод мультипроцесорного модулярного множення, теоретично описана послідовність дій для його реалізації, досліджена його швидкодія. Особливість методу полягає в розпаралелюванні обчисдень і використанні групової редукції часткових множників.
3. Запропонований метод відрізняється тим, що виконується поділ множника на часткові множники, кожний з яких обробляється на окремому ядрі комп’ютера, а над нульовим групами старших розрядів множника виконується групова редукція, що значно зменшує кількість виконаних операцій.
4. Виконано аналіз розробленого методу, згідно з яким визначено оптимальні розміри часткових множників при яких досягається найбільша швидкодія з суміщенням виконання групової редукції розрядів множника.

5. Запропонований метод може бути застосований в криптографічних системах з асиметричним шифруванням, для підвищення надійності зберігання даних.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ МУЛЬТИПРОЦЕСОРНОГО МОДУЛЯРНОГО МНОЖЕННЯ З ВИКОРИСТАННЯМ ГРУПОВОЇ РЕДУКЦІЇ МОНТГОМЕРІ

Згідно з методом, описаним в попередньому розділі, необхідно розробити програмні засоби для його реалізації. Для цього треба обрати відповідну для цієї задачі мову програмування, описати основні класи програми та розробити користувацький інтерфейс.

Програмний додаток надає можливість здійснювати методи модулярного множення Монтгомері та методу мультипроцесорного модулярного множення з використанням групової редукції Монтгомері. Також надаються функції, які аналізують швидкодію методів та виводять результати у вигляді графіків.

Програма використовує спеціальні бібліотеки для розроблення користувацького інтерфейсу та ввідображення графіків швидкодії.

3.1 Розробка класів програмного забезпечення

Нижче описані основні класи і модулі розробленої програми.

Клас Graphic

Клас Graphic представляє програму для генерації та візуалізації графіку часу виконання алгоритму модулярного множення.

У конструкторі класу Graphic ініціалізуються властивості, які зберігатимуть вихідні дані та результати виконання програми. M, A та B є списками, в яких зберігаються випадково згенеровані значення для параметрів модуля M та чисел A та B. time1 є списком, в якому буде зберігатись час виконання кожного обчислення.

Метод generate() використовується для генерації випадкових значень параметрів M, A та B для кожного розміру вхідних даних. Розміри вибираються зі списку [64, 128, 256, 512, 1024, 2048].

Метод calc() виконує обчислення для кожного набору параметрів M, A та B за допомогою класу Montgomery. Створюється список об'єктів Montgomery, налаштованих з відповідними параметрами, і для кожного об'єкта викликається метод calculate(), який виконує обчислення та зберігає час виконання властивості calculation_time. Час виконання для кожного набору параметрів зберігається у списку time1.

Метод vizualize() використовує бібліотеку matplotlib для візуалізації графіку часу виконання. Значення розмірів даних (64, 128, 256, 512, 1024, 2048) використовуються як осі x, а значення зі списку time1 використовуються як осі y. Графік відображається за допомогою plt.plot() та plt.show().

Ця програма дозволяє згенерувати випадкові дані для виконання алгоритму модулярного множення та візуалізувати графік часу. Графік дозволяє порівняти час виконання алгоритму для різних розмірів даних.

Застосування цієї програми може бути особливо корисним у дослідженнях з областей, де модулярне множення використовується, таких як криптографія. Вона дозволяє експериментувати з різними розмірами даних та оцінювати швидкість виконання алгоритму, що допомагає знайти оптимальні рішення для конкретних завдань.

Клас FastMontgomery

Клас MontgomeryFast представляє програму для виконання мультипроцесорного модулярного множення з використанням групової редукції Монтгомері. Конструктор класу приймає параметри A, B, M та number_of_cores, які визначають вхідні дані для операції модулярного множення, а також кількість ядер процесора, на яких буде виконуватись обчислення.

Клас має наступні властивості та методи:

1. make_precalculate_table(): Цей метод створює попередньо обчислену таблицю для оптимізації процесу множення. Він генерує значення таблиці на основі вхідного модуля M та кількості бітів d, а також зберігає створену таблицю у властивості precalculate_table.
2. make_parts(): Цей метод розбиває число B на частини з рівномірним розподілом кількості бітів між ядрами процесора. Результати розбиття зберігаються у властивості parts.
3. montgomery(): Цей метод виконує операцію модулярного множення на кожній частині числа B. Він використовує групову редукцію Монтгомері для прискорення обчислень та зберігає результати у властивості part_results.
4. use_table(): Цей метод використовує попередньо обчислену таблицю для подальшої редукції результатів обчислень. Він застосовує значення з таблиці до кожного результату властивості part_results та виконує додаткові операції редукції. Загальний результат зберігається у змінній res.

Після створення об'єкту класу MontgomeryFast та виклику всіх необхідних методів, результати обчислень можна отримати з властивості part_results або змінної res. Загальний час обчислень зберігається у властивості calculation_time.

Клас Montgomery

Клас Montgomery, виконує обчислення за методом Монтгомері для множення чисел A та B за модулем M. Метод Монтгомері є одним з методів прискореного множення, основаним на арифметиці в полі з великим модулем.

Опис методів класу Montgomery:

- Конструктор __init__ приймає параметри A, B і M та ініціалізує змінні об'єкта. Змінні A, B та M відповідають числам, які будуть множитись та модулю, відповідно. Змінна calculation_time ініціалізується нулем і

використовується для відстеження кількості операцій, виконаних під час обчислення.

- Метод calculate виконує обчислення за методом Монтгомері і виводить результати на екран. Спочатку числа A та B множаться звичайним способом (a * b). Далі число B перетворюється в бінарне представлення і проводяться додавання, зсуви та модульна арифметика для кожного біта. Кінцевий результат обчислюється за формулою (res * 2 ** len(b)) % m. Результати виводяться на екран, включаючи проміжні кроки обчислення.
- Метод show виводить кроки обчислення методом Монтгомері для чисел A та B за модулем M. Кожен крок виводиться на екран, включаючи числа та проміжні результати. Результати також зберігаються у вигляді рядка і повертаються для подальшого використання.

Використовуючи клас Montgomery, можна виконувати швидке множення чисел за модулем з використанням методу Монтгомері. Програма надає зручний спосіб виводу результатів та кроків обчислення для аналізу та наочності.

Модуль make_table.py

Код починається з імпорту бібліотеки openpyxl, яка дозволяє працювати з Excel-файлами. Далі створюється нова робоча книга (Workbook) та активний аркуш (Worksheet) в цій книзі.

Функція make_M приймає два аргументи - M (модуль) і d (кількість бітів для представлення чисел). Функція створює два списки array_M і array_d, які містять числа від 0 до 2^d - 1, збільшені на M і відповідно самі числа від 0 до 2^d - 1. Далі створюється порожній список precalculate_table, який буде містити передобчислені значення. Потім проходиться подвійний цикл, де для кожного елемента i з array_d та для кожного елемента j з array_M перевіряється умова, чи останні d бітів суми i та j рівні d нулів. Якщо умова виконується, то пара (бінарне представлення числа i, j/M, j) додається до списку precalculate_table та додається новий рядок у аркуш Excel з відповідними значеннями. На

завершення, значення precalculate_table виводиться на екран. Результат роботи функції відображений на малюнку 3.1.

rl...r0	D	
0000	0	0
0001	28399	110111011110000
0010	56798	1101110111100000
0011	20285	100111101000000
0100	48684	1011111000110000
0101	12171	10111110010000
0110	40570	1001111010000000
0111	4057	111111100000
1000	32456	111111011010000
1001	60855	1110110111000000
1010	24342	101111100100000
1011	52741	1100111000010000
1100	16228	11111101110000
1101	44627	1010111001100000
1110	8114	1111111000000
1111	36513	1000111010110000

Рисунок 3.1 - Результат роботи функції precalculate_table

Функція make_M1 має подібну структуру до функції make_M, але має деякі відмінності. В цій функції також створюються списки array_M і array_d, але тут умова перевіряється на останні d бітів суми i та j рівні d нулів або коли сума i та j дорівнює 0. Якщо умова виконується, то пара (бінарне представлення числа i, j, бінарне представлення суми i та j) додається до списку precalculate_table, а також значення j додається до списку table. Далі значення precalculate_table виводиться на екран.

На завершення, викликається функція make_M1 з вибраними аргументами значення table виводиться на екран. На кінці коду, робоча книга зберігається у файл "pre_calculate.xlsx".

Модуль make_table.py

Код починається з імпорту бібліотеки openpyxl, яка дозволяє працювати з Excel-файлами. Далі створюється нова робоча книга (Workbook) та активний аркуш (Worksheet) в цій книзі.

Функція create_table обчислює теоретичну довжину часткових множників і приймає три аргументи - k (кількість часткових множників), q (основа для обчислення часткових множників) та n (загальна довжина). Функція формує таблицю з величинами часткових множників та значенням betta. Спочатку створюється список h, до якого додається рядок зі значенням q. Далі обчислюється значення gamma за формулою 2/(2 - 1.5/q) і значення h1 за формулою n*(gamma - 1)/(gamma**k - 1). Далі, за допомогою циклу, обчислюються значення часткових множників і додаються до списку h.

На останньому кроці обчислюється значення betta (n/h[-1]) і додається до списку h. Значення h виводиться на екран та додається у відповідні рядки аркуша Excel, рисунок 3.2 демонструє сформовану програмою таблицю для чотирьох процесорів.

Theoretical					
	h1	h2	h3	h4	betta
q = 4	365,0674	449,3138	553,0015	680,6173	3,009033
q = 8	439,0513	484,4704	534,588	589,8902	3,471832
q = 16	475,7396	499,1367	523,6844	549,4393	3,727436
q = 32	493,9322	505,7866	517,9255	530,3557	3,861559

Рисунок 3.2 - Результат роботи функції create_table

Результат роботи функції для восьми процесорів зображений на рисунку 3.3.

	h1	h2	h3	h4	h5	h6	h7	h8	betta
q = 4	110,8079	136,379	167,851	206,5859	254,2595	312,9348	385,1505	474,0314	4,320389
q = 8	176,8553	195,1507	215,3387	237,6151	262,196	289,3197	319,2493	352,2751	5,813638
q = 16	215,0999	225,6785	236,7775	248,4223	260,6398	273,4581	286,9069	301,0171	6,803601
q = 32	235,2605	240,9068	246,6886	252,6091	258,6717	264,8798	271,2369	277,7466	7,373627

Рисунок 3.3 - Результат роботи функції create_table

Функція create_table2 обчислює практичну довжину часткових множників. Вона також має аргументи k, q і n. В цій функції також створюються список h та обчислюються значення gamma і h1. Потім за допомогою циклу зворотнього проходу обчислюються значення часткових множників і додаються до списку h. Після цього виконується додаткова обробка значень часткових множників. Для кожного значення h[i] порівнюється відхилення a (різниця між h[i] і найбільшим цілим числом, більшим або рівним h[i], що кратним q) і відхилення b (різниця між h[i] і найбільшим цілим числом, меншим або рівним h[i]). Якщо a менше за b, то значення h[i] замінюється на найбільше ціле число, більше або рівне h[i] і кратне q, і навпаки. Після цього обчислюється значення betta і додається до списку h. Значення h виводиться на екран та додається у відповідні рядки аркуша Excel.

Далі в коді визначаються заголовки для таблиці та викликаються функції create_table та create_table2 з різними значеннями аргументів для формування таблиць. Результати записуються у відповідні рядки аркуша Excel. На кінці робоча книга зберігається у файл "make_table.xlsx". Сформована функцією таблиця зображена на рисунку 3.4.

	h1	h2	h3	h4	betta
q = 4	368	448	552	680	3,011765
q = 8	432	488	536	592	3,459459
q = 16	480	496	528	544	3,764706
q = 32	480	512	512	544	3,764706

Рисунок 3.4 - Результат роботи функції create_table

	h1	h2	h3	h4	h5	h6	h7	h8	betta
q = 4	108	136	168	208	256	312	384	476	4,302521
q = 8	176	192	216	240	264	288	320	352	5,818182
q = 16	208	224	240	256	256	272	288	304	6,736842
q = 32	224	256	256	256	256	256	256	288	7,111111

Рисунок 3.5 - Результат роботи функції create_table

Також сформована таблиця для 16 процесорів (рисунок 3.6).

	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	betta
q = 4	24	20	28	32	40	48	60	76	92	116	140	172	212	264	324	400	5,12
q = 8	64	64	64	72	80	88	96	112	120	136	152	160	184	200	216	240	8,533333
q = 16	80	96	96	96	112	112	112	128	128	128	144	144	160	160	176	176	11,63636
q = 32	96	96	96	128	128	128	128	128	128	128	128	128	128	160	160	160	12,8
q = 64	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	16
q = 128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	16

Рисунок 3.6 - Результат роботи функції create_table

При застосуванні значень довжин, сформованих програмою, досягається найбільше прискорення.

3.2 Опис реалізації алгоритму

Виконання алгоритму розпочинається з етапу ініціалізації даних, вводу початкових даних масиву часткових множників *H*, множеного *G*, значення модуля *M*, визначення списку *R* часткових результатів обробки часткових множників. Ці дії виконуються в блоці 1 схеми алгоритму.

У блоці два алгоритмум також виконується ініціалізація, вводяться дані кількості *n* розрядів множника , кількість ядер *r*, на яких виконується паралельна обробка часткових множників з масиву *H*, довжина *p* групи бітів, над якими виконується групова редукція Монтгомері.

У третьому блоці алгоритму формується таблиця передобчислень згідно з параметрами модуля *M* та довжиною *p* групи бітів, над якими одночасно виконується редукція. Таблиця формується таким чином, щоб при додаванні першого елемента таблиці - добутка константи і модуля - до другого елемента - чотирьох бітів – *p* останніх бітів суми були нульовими.

У блоці чотири алгоритма формується два списки: *H, group*. Перший з них представляє собою список часткових множників множника *H*, а другий список - кількість операцій групової редукції, яка виконується над кожним частковим множником. На цьому етап ініціалізіціїї завершується.

Для того, щоб організувати цикли, ініціалізуються лічильники *i, j,* яким присвоюються нульові значенняв блоці 5.

В блоці шість виконується перевірка *j*-того розряду *i*-того часкового множника в списку *H.*

Якщо поточний біт дорівнює одиниці, то до *i*-того часткового результату додається значення множеного *G* (блок 7 алгоритму). В іншому випадку, частковий результат залишається незмінним і виконується перехід до блоку вісім.

В блоці вісім здійснюється перевірка молодшого розряду *i*-того часткового результату.

Якщо молодший розряд часткового результату дорівнює одиниці, то до нього додається значення модуля. Інакше здійснюється перехід до блоку десять алгоритму.

У десятому блоку виконується зсув *i*-того часткового результату на один біт вправо. Таким чином, його значення зменшиться вдвічі.

В одинадцятому блоці алгоритму виконується інкремент лічильника *j*.

Блок дванадцять виконує перевірку лічильника ,*j*, чи є він меншим за довжину *i*-того часткового множника. Якщо умова не виконується, алгоритм повертається до шостого блоку. У випадку, коли умова справджується виконується тринадцятий блок алгоритму.

Тринадцятий блок алгоритму виконує над *i*-тим частковим результатом операцію за модулем.

В чотирнадцятому блоці виконується ініціалізація лічильника *m,* йому присвоюється нульове значення.

В п’ятнадцятому блоці здійснюється додавання до *i*-того часткового результату елементу з таблиці передобчислень, ключ якого відповідає останнім *p* молодшим розрядам результату.

Інкремент лічильника *m* виконується в шістнадцятому блоці.

Робиться перевірка лічильника *m* (блок 17), якщо його значення менше значення *i*-того елемента списка *group*, то виконується перехід до кроку п’ятнадцять, інакше, здійснюється перехід до наступного блоку.

Перевіряється, чи перевищує значення *i*-того часткового результату значення модуля (блок 18). Якщо перевищує, то виконується операція за модулем (блок 19), інакше здійснюється перехід до блоку 20.

В двадцятому блоці лічильник *i* збільшується на одиницю.

Блок 21. Якщо значення лічильника *i* менше за довжину списку часткових множників, то алгоритм повертається на блок 6. Або ж в блоку 22 підсумовуються значення часткових результатів у змінну *result.*

Контролюється чи не перевищує результа значення модуля (блок 23). Якщо перевищує, то значення береться за модулем (блок 24).

Обраховується дійсний результат модулярного множення, домноженням результату на 2^n і береться за модулем.

На цьому алгоритм запропонованим методом завершується.

3.3 Мова програмування для розроблення програмного забезпечення

Для розроблення програмного забезбечення була обрана мова програмування Python, її стандартні функціїї та бібліоетки. Вибір на користь цієї мови програмування зроблений через широке застосування модулів Python для реалізації криптографічних і математичних задач. Відповідно, існує багато бібліотек для роботи з криптографічними функціями і документації до них, що спрощує програмну реалізацію запропонованого методу.

Мова програмування Python – це динамічно типізована інтерпретована мова програмування, що робить її придатною для вирішування великої кількості задач різних типів. Важливим фактором для даного завдання мови Python є гнучкість в доступі й управлінні пам'яттю.

3.4 Користувацький інтерфейс

Для взаємодії з користувачем, вводу даних і відображення результатів роботи програми був розроблений користовацький інтерфейс.

Головне вікно програми (рисунок 3.7) створене за допомогою команди root = Tk() містить три кнопки - montgomery_calculate_button, fast_montgomery_calculate_button, graphics_button - які створені за допомогою функції Button(). При натисканні кнопок викликаються відповідні функції montgomery_calculation(), fast_montgomery_calculation(), graphics(). Нижче описано дії, які виконуються при виклику цих функцій.

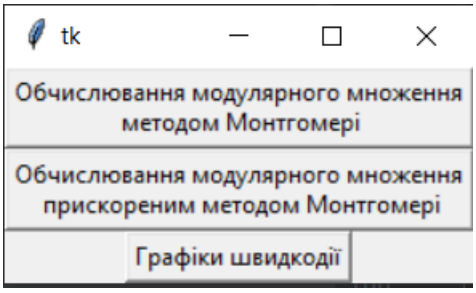


Рисунок 3.7 - Головне вікно програми

Функція montgomery_calculation() відповідає за створення нового вікна за допомогою функції Toplevel(). Вікно win містить набір елементів інтерфейсу таких як поля для введення множеного A, множника B та модуля M (label_A, label_B, label_M), що створені функцією Entry(), мітки A, B, M (entry_A, entry_B, entry_M) над кожним полем, утворені методом Label(), кнопка result_button і мітка label_result. За допомогою кнопки result_button з текстом "Обчислити результат" викликається функція button_calculate() визначена

всередині функції montgomery_calculation(). Вона отримує значення введених користувачем множників A, B і модуля M з полів, передає їх до об'єкту Montgomery, обчислює результат і встановлює текст результату в мітку label_result.

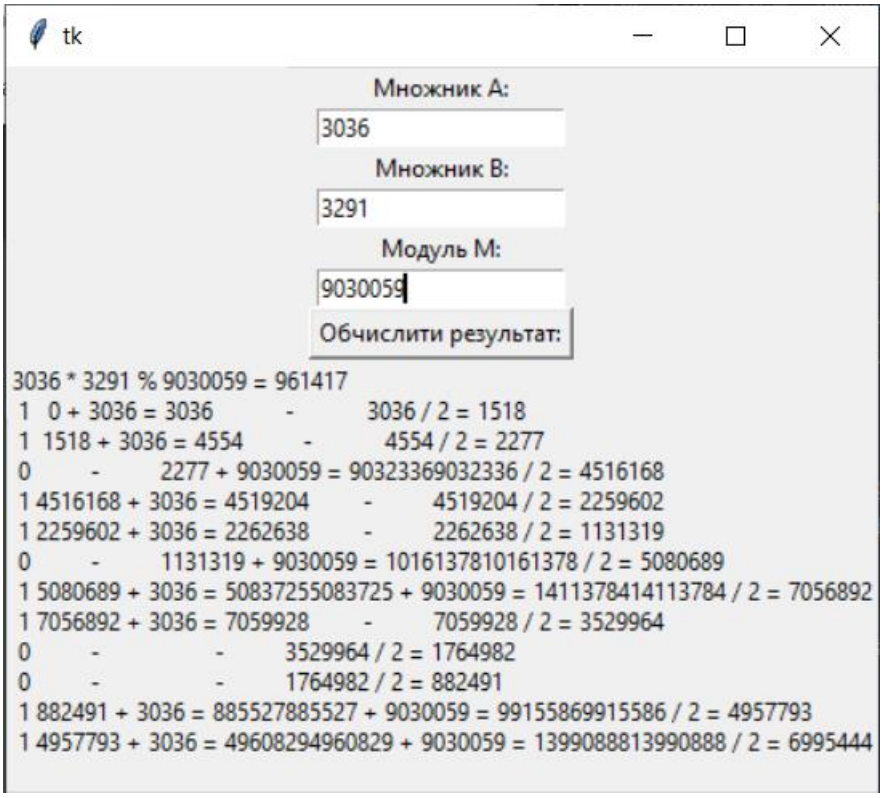


Рисунок 3.8 - Вікно обчислення модулярного множення методом Монтгомері

Функція fast_montgomery_calculation() створює нове вікно. Воно містить поля для введення множеного A, множника B та модуля M, мітки A, B, M над кожним полем, кнопку обчислення результату і мітку для відображення результату. За допомогою кнопки result_button викликається функція button_calculate(). Вона отримує значення введених користувачем множників A, B і модуля M з полів, передає їх до об'єкту FastMontgomery, обчислює результат прискореного модулярного множення і повертає текст результату.

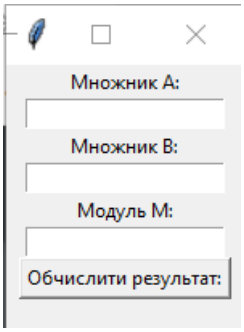


Рисунок 3.9 - Вікно обчислення модулярного множення запропонованим методом

Функція graphics() відповідає за створення графіків швидкодії. Ця функція створює об'єкт Graphic, який має методи для генерації даних, обчислення та візуалізації графіків.

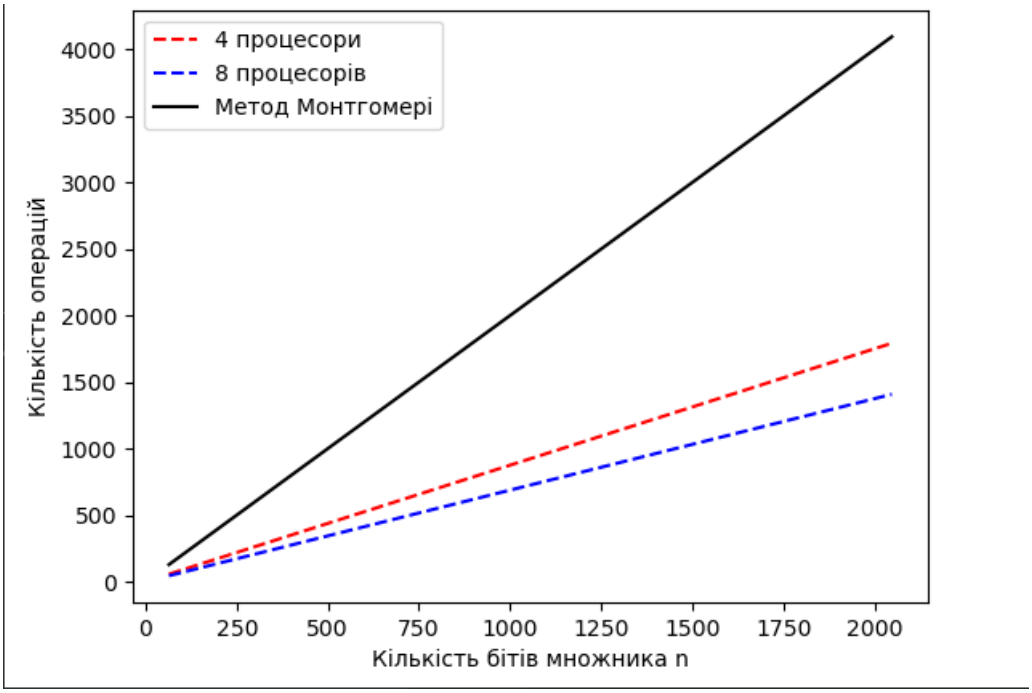


Рисунок 3.10 - Вікно відображення графіків

Метод generate() генерує дані для графіків, метод calc() обчислює значення для графіків, метод vizualize() відображає графіки. На графіку відображено два графіки: швидкодія методу Монтгомері та швидкодія

розробленого методу мультипроцесорного модулярного множення з використанням групової редукції.

3.5 Опис структур даних в програмі

Структури даних в програмах застосовуються для організації та зберігання даних з метою ефективного доступу до них, для виконання над ними операцій та для того, щоб оптимізувати виконання програми. Існує декілька типів структур даних, і вибір конкретної залежить від вимог і характеристик конкретної задачі.

Для зручності застосування цього типу даних, використовується ітератор, щоб мати доступ до будь-якого розряду числа.

Багато даних розробленої програми зберігаються в списках. Список представляє собою структуру даних, де зберігаються елементи одного або декількох типів даних і доступ до цих елементів можна отримати за допомогою індексу. Використання списків доцільне в даній програмі, через те, що необхідно зберігати часткові множники та часткові результати, здійснювати над ними операції додавання, зсуву та доступу до елементів. В даній програмі, зокрема, створені таки списки:

- parts – список, що містить часткові множники. Список є частиною класу MontgomeryFast. Спочатку список ініціалізується пустим, а його значення формуються протягом запуску і виконання функції make_parts().
- part_results – створений список ініціалізується пустим, значення до нього додаються по мірі виконання внутрішнього методу montgomery(), а потім над ними виконується редукції за допомогою методу use_table(), після чого над елементами списку виконується множення і отримуються істинні значення результатів. У підсумку значення елементів цього

списку сумується і корелюється, завдяки чому отримується остаточний результат модулярного множення.

- array_M – цей список містить в собі значення модуля *M* помножені на константи від нуля до двійки в степені значення *d* довжини групової редукції. Список є складовою класа MontgomeryFast і його елементи створюються у циклі при виклику функції make_precalculate_table.
- array_d – список включає значення чисел від нуля до двійки зведеної в степінь *d*. За допомогою функції make_precalculate_table елементи цього списку додаються до елементів списку array_M і пари елементів, останні d розрядів суми яких дорівнюють нулям, записуються до таблиці передобчислень precalculate_table.
- time1 – список міститься у класі Graphic. В ньому розміщеі елементи, значення яких дорівнюють часу виконання модулярного множення методом Монтгомері на множниками різної довжини, починаючи від 64 розрядів до 2048 бітів. Час отримується, як параметр об’єктів класу Montgomery атрибуту calculation_time.
- time2 – список є складовою класа Graphic. В ньому містяться елементи, значення яких дорівнюють часу виконання модулярного множення запропонованим методом над множниками різної довжини, починаючи від 64 розрядів до 2048 бітів. Час отримується, як параметр об’єктів класу Montgomery атрибуту calculation_time.
- list_of_objects – цей список створений для того, щоб зберігати об’єкти класу Montgomery з різними параметрами ініціалізаціїї цих об’єктів (змінюється довжина множника).
- x – список створений у методі vizualize() класу Graphic. У нього включені числа 64, 128, 256, 512, 1024, 2048. Цей список являє собою довжини множника і використовується, що визначити кількість

операцій, що виконуються під час модулярного множення на множниками таких довжин.

- y – це список цілочисельних значень кількості операцій, які потрібні для операції модулярного множення за методом Монтгомері. Елементи списку обраховуються після виклику функції внутрішнього методу calc() класу Graphic.
- y1 – це список цілих значень кількості операцій, які потрібні для виконання операції модулярного множення з використанням розробленого методу. Елементи списку обраховуються після виклику функції внутрішнього методу calc() класу Graphic.

У програмі також наявні змінні, які встановлюються на початку виконання програми, задаються користувачем у відповідних полях графічного інтерфейсу або обраховуються в ході виконанні операції модулярного множення. В розробленій програмі наявні такі змінні:

- A – змінна, яка містить значення множеного. Може вводитися коритувачем у відповідному вікні абож генерується внутрішнім методом generate() класу Graphic.
- B – змінна, яка містить значення множника. Може вводитися коритувачем у відповідному вікні абож генерується внутрішнім методом generate() класу Graphic.
- M – змінна, яка містить значення модуля. Може вводитися коритувачем у відповідному вікні абож генерується внутрішнім методом generate() класу Graphic.
- number_of_cores – ця змінна визначена на початку програми і складає значення кількості процесорів, на яких одночасно виконується операція обробки часткових множників.
- calculation_time – створена змінна кількість операцій, які витрачаються на обрахування модулярного множення, в подальшому значення цієї

змінної може використовуватися для побудови графіку швидкодії (залежність довжини ножника від кількості виконаних операцій).

- res – змінна, в якій зберігається поточний результат, він змінюється по мірі виконання програми, а при завершенні циклу значення цієї змінної додається до списку part_results.
- d – змінна, де зберігається значення кількості групи бітів, які оброблюються одночасно з використанням таблиці передобчислень, сформованою функцією make_table().

Деякі дані зберігаються у словниках. Слоники містять у собі пари ключ-значення і доступ до значень словника відбувається за ключем. Наприклад в розробленій програмі використовується наступний словник:

- precalculate_table – словник у якому зберігається таблиця передобчислень. Ключами виступають різні комбінації чотирьох бітів, а значеннями лінійні комбінації модуля та деякої константи.

У програмі наявні структури типу byte. Ця структура необхідна для того, щоб пребирати біти введеного користувачем множника. Оскільки множник вводиться користувачем в десятковій системі числення.

ВИСНОВОК ДО РОЗДІЛУ 3

Разом з теоретичним і практичним дослідженнями, зробленими у другому розділі, були створені програмні засоби для прискорення операції модулярного множення. Підсумовуючи, можна виділити наступні цілі досягнені в третьому розділі:

1. Методи, направлені на пришвидшення операції модулярного множення, які детально описані у другому розділі, реалізовані за допомогою програмних засобів і можуть бути застосовані при обробці багаторозрядних чисел.
2. Були розроблені основні класи для реалізації методу мультипроцесорного модулярного множення з використанням групової редукції Монтгомері. Класи виконують задачі обчислення результату модулярного множення двома методами: методом запропонованим в бакалаврському проекті.
3. Практичні досліди, які виконувалися із застосуванням розробленої програми, показують, що теоретично розраховане прискорення у другому розділі збігається з експериметальним значенням.
4. Практичні дослідження підтверджують, що існує обмеження на кількість часткових множників, на які можна поділити множник.
5. Експериметальними дослідженнями доведено, що довжина групи бітів має обмеження. Вибір великої кількості бітів для одночасної обробки не сприяє значному прискоренню модулярного множення.
6. Створені програмні засоби можна використовувати не тільки в експериментальних цілях, але і на практиці, в таких криптографічних протоколах як DSS, ISO-979.

ВИСНОВКИ

У ході опрацювання дипломної роботи направленої на прискорення операції модулярного множення, отримано наступні результати.

1. Проаналізувавши сучасний арсенал криптографічних методів захисту даних, можна сказати, що в них застосовуються операції модулярного множення й експоненціювання. Операції ці проводяться над числами великої розрядності, зокрема довжина модуля може становити в сучасних системах 2048 бітів. Через це програмна реалізація цих алгоритмів виконується повільно. Тому, швидкодію таких методів можна підвищити за рахунок скорочення часу обчислень операцій модулярного множення і експоненціювання.
2. Виконаний аналіз протоколів, які засновані на криптографічних асиметричних алгоритмах, зокрема такі алгоритми як El-Gamal, RSA, DSS (алгоритм, що використовується для генерування цифрового підпису). Цей аналіз демонструє, що ключі цих асиметричних алгоритмів змінюють дуже рідко через те, що відкритий ключ ідентифікує користувачів системи. Відповідно, модуль *M,* що використовується в цих криптографічних алгоритмах, можна вважати сталою величиною і завдяки цьому можна зменшити обчислювальну складність операцій модулярного експоненціювання й множення.
3. З попереднього аналізу вже існуючих алгоритмів випливає, що операція модулярного експоненціювання не може бути розпаралелена через те, що дії алгоритму виконуються строго послідовно. Тоді єдиним способом для прискорення цієї операції є прискорення частини цього алгоритму, зокрема операції модулярного множення.

4. Проаналізувавши метод секційного множення, можна дійти висновку, що при застосуванні декількох процесорів для кожногї з частин, кількість операцій може бути скорочена.
5. З вище розглянутих методів, що направлені на ефективне виконання модулярного множення можна виділити метод Монтгомері. Основною перевагою цього методу є те, що редукція результату і множення робляться одночасно, що дозволяє досягти найбільшої швидкодії.
6. Основним потенціалом для прискорення операції модулярного множення є використання передобчислень в частині здійснення обчислень модулярної редукції. Перспективно. є можливість виконувати одночасно операцій модулярного множення і модулярної редукції з використанням обчислень одночасно. Це, в свою чергу, дає можливість скоротити час здійснення цих операцій і усунути роботу з багаторозрядними числами, що за розміром більше ніж довжина модуля.
7. Розроблено метод мультипроцесорного модулярного множення, теоретично описана послідовність дій для його реалізації, досліджена його швидкодія. Особливість методу полягає в розпаралелюванні обчисдень і використанні групової редукції часткових множників.
8. Запропонований метод відрізняється тим, що виконується поділ множника на часткові множники, кожний з яких обробляється на окремому ядрі комп'ютера, а над нульовим групами старших розрядів множника виконується групова редукція, що значно зменшує кількість виконаних операцій.
9. Виконано аналіз розробленого методу, згідно з яким визначено оптимальні розміри часткових множників при яких досягається найбільша швидкодія з суміщенням виконання групової редукції розрядів множника.

10. Запропонований метод може бути застосований в криптографічних системах з асиметричним шифруванням, для підвищення надійності зберігання даних.
11. Методи, направлені на пришвидшення операції модулярного множення, які детально описані у другому розділі, реалізовані за допомогою програмних засобів і можуть бути застосовані при обробці багаторозрядних чисел.

    Були розроблені основні класи для реалізації методу мультипроцесорного модулярного множення з використанням групової редукції Монтгомері. Класи виконують задачі обчислення результату модулярного множення двома методами: методом запропонованим в бакалаврському проекті.
12. Практичні досліди, які виконувалися із застосуванням розробленої програми, показують, що теоретично розраховане прискорення у другому розділі збігається з експериметальним значенням.
13. Практичні дослідження підтверджують, що існує обмеження на кількість часткових множників, на які можна поділити множник.
14. Експериметальними дослідженнями доведено, що довжина групи бітів має обмеження. Вибір великої кількості бітів для одночасної обробки не сприяє значному прискоренню модулярного множення.
15. Створені програмні засоби можна використовувати не тільки в експериментальних цілях, але і на практиці, в таких криптографічних протоколах як DSS, ISO-979.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кабір Л.А. Метод прискорення модулярного множення за технологією Монтгомері / Л.А. Кабір, О.В. Русанова, І.О. Гуменюк //Альманах науки № 1(52).- 2022.- С.44-46.

2. Трибунська К.Є. Метод прискорення модулярного множення з використанням групової редукції /К.Є. Трибунська // Актуальні питання розвитку науки та освіти: матеріали М Міжнародної науково-практичної конференції м.Львів, 30-31 березня 2022 року.-Львів: Львівський науковий форум, 2022.- С.38- 46.

3. Кот О.С. Організація прискореного експоненціювання на полях Галуа з використанням редукції Монтгомері / О.С.Кот, О.П. Марковський // Альманах науки.- 2020.- № 3 (36).- С.34-37.

4. Thangaval M. Improved secure rsacryptosystem (ISRSAC) for data confidentiality in cloud / M. Thangaval, P.Varalakshmi // International Journal of Information Systems and Change Managerment,- 2017.- Vol.9,- No.4, - P.46-53.

5. Калмиков І.А. Розробка методу нелінійного шифрування інформації з використанням операції піднесення до степеня для кінцевого поля Галуа / І.А. Калмиков, Е.С. Степанова, К.Т. Тинчеров// Сучасні наукомісткі технології. - 2019.- № 9. - С.84—89.

6. Osadchyy V. The Order of Edwards and Montgomery Curves «, / V.Osadchyy // WSEAS Transactions on Mathematics, - 2020.- Vol. 19.- № 25, - P. 253-264.

7. Haches G. Montgomery multiplication with no final subtraction./ G. Haches, J.J. Quisquater // Cryptographic Hardware and Embedded System- CHES’2000. LNCS-1965, Springer-Verlag. — 2000.- P. 293-301.

8. Марковський О.П. Метод прискорення експоненціювання з використанням передобчислень / О.П. Марковський, О.В. Русанова, А.А.

Олієвський, В.М.Черевик // Телекомунікаційні та інформаційні технології. - 2018.- № Зм. Арк. № докум. Підпис Дата Арк. 68 ІАЛЦ.468243.003 ПЗ 1(58). – С.31-39.

9. Elfard S. Justification of Montgomery Modular Reductions / S. Elfard //Advanced Computing.- 2012. - № 11. – P.41-45.

10. Анисимов А.В. Быстрое прямое вычисление модулярной редукции // Кибернетика и системный анализ.-1999.-№ 4.-С.3-12.

11. Che Wun Chion. Parallel modular multiplication with table look-up. / Che Wun Chion, Ted C.Yang // International Journal of Computer Mathematics.- 1998.- Vol.69.-Issue 1-2.- P.22-23.

12. Zuras D., More on squaring and multiplying larges integers, IEEE Transactions on Computers,-1994.- Vol. 43, - № 8,- P. 899–908.

13. Березовский А.И. О тестировании быстродействия алгоритмов и программ вычисления основных операций ассиметричной криптографии/ Березовский А.И., Задирака В.К., Шевчук Л.Б. //Кибернетика и системный анализ №5, 1999. – С. 59-66.

14. Giorgi P. Parallel modular multiplication on multi-core processors. / Giorgi P., Imbert L., Izard T. // IEEE Symposium on Computer Arithmetic, Apr 2013, Austin, TX, United States. - P.135-142.

15. Buhrow B. Parallel modular multiplication using 512-bit advanced vector instructions: RSA fault-injection countermeasure via interleaved parallel multiplication / Buhrow B., Gilbert B., Haider C. // Journal of Cryptographic Engineering.-2021.- № 2.- P.46-53.

16. Karatsuba, A.Multiplication of many-digital numbers by automatic computers. Proc. USSR Acad. Sci. =1962.- Vol.145,- P. 293–294.

17. Анісімов А.В. Алгоритмічна теорія великих чисел / А.В. Анісімов // К.: Академперіодика. —2001. — С.153.

18. Blum T., Paar C. Montgomary Modular Exponentiontiation on Reconfigurable Hardware.//Proc.14-th IEEE Symp.on Comput. Arithmetic, Adelaide,14-16 April 1999,-IEEE Press,-1999- P.70-77.

19. Hong S-M. New Modular Multiplication algorithms for fast modular Зм. Арк. № докум. Підпис Дата Арк. 69 ІАЛЦ.468243.003 ПЗ exponeniation / Hong S-M., Oh S-Y., Yoon H // Advances in CryptologyProceedings of Eurocrypt '96. –Springer-Verlag. -1996. – P.166-177.

20. Laszlo Hars. Long Modular multiplication for Cryptographic Applications.//Cryptographic Hardware and Embedded System- CHES'2004. LNCS-3156, Springer-Verlag, - 2004. - P.45-61.

21. Walter C.D. Faster Modular Multiplication by Operand Scaling / C.D. Walter // Proceeding of Advances in Cryptology-CRYPTO-91, LNCS-576, SpringerVerlag. – 1992. – P. 313-323.

22. Kawamura S. A fast modular exponentiation algorithm / S. Kawamura, K. Takabayashi, A. Shimbo // IEEE Transaction on Information Theory. – Vol. 94. – № 6. – 2015. – P.2136-2142.

23. Самофалов К.Г. Эффективная реализация мультипликативных операций модулярной арифметики в системах защиты информации/ К.Г. Самофалов, Г.М.Луцкий, А.П. Марковский // Proceeding of International scientific conference UNITECH-09. Gabrovo 20-21 November 2009.- Technical University of Gabrovo. - 2009.— V.1.— P.435-437.

24. Харин Ю.С. Математические основы криптологии./ Ю.С.Харин, В.И. Берник, Г.В. Матвеев//Минск.: Изд-во БГУ —1999. — 319 с.

25. Montgomery P.L. Modular multiplication without trial division / P. L. Montgomery // Mathematics of Computation, - 1985.- Vol.44. - № 4.- P.519-523.

ДОДАТОК 1

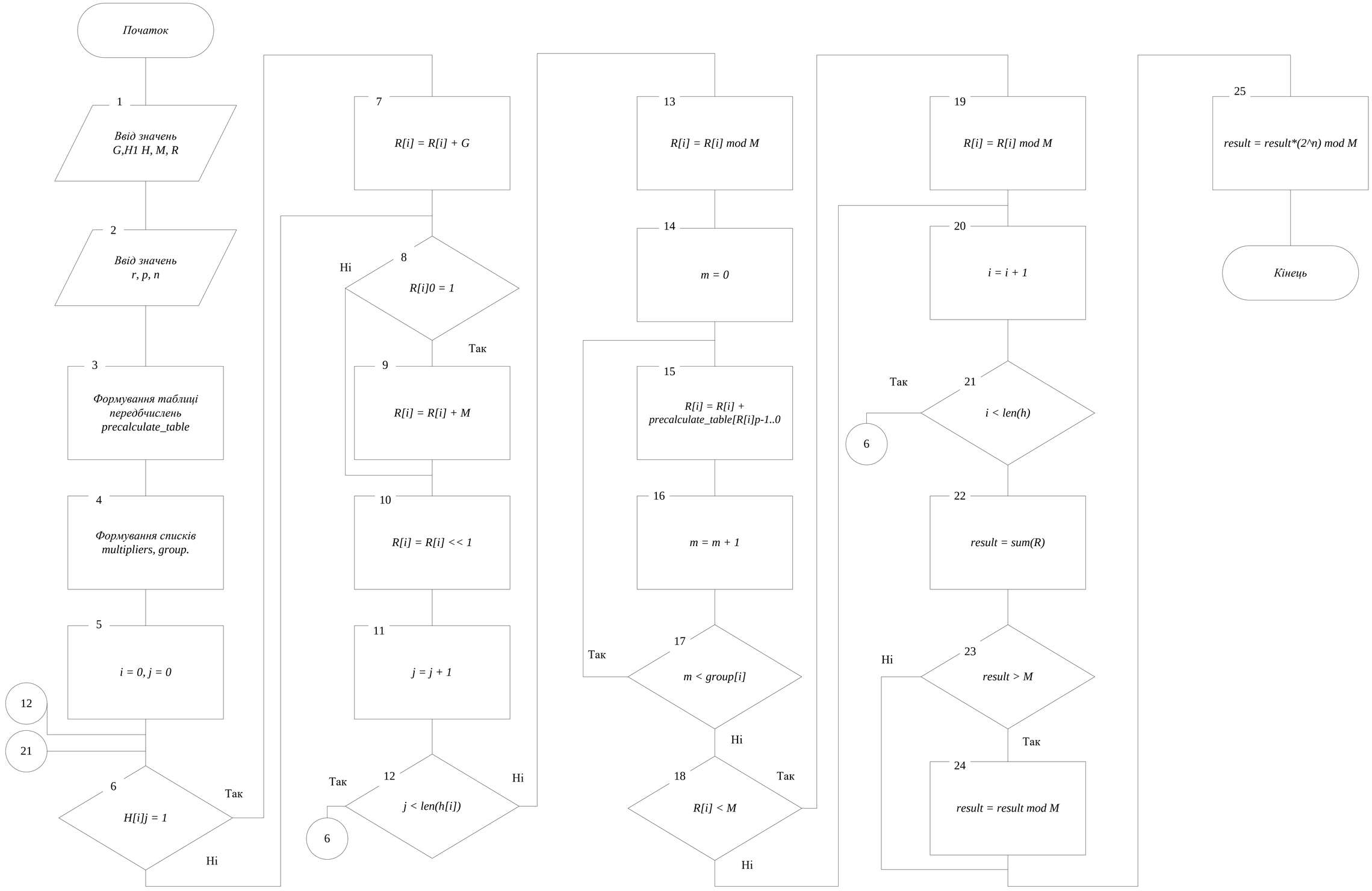
Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері

Блок схема алгоритму

ІАЛЦ.468243.004 Д1

Аркушів 1

Київ 2023 р



						ІАЛЦ.468243.004.Д1	
Зм	Лист	№докум.	Підп.	Дата			
Розроб.		Трибунська			Метод мультипроцесорного множення Схема алгоритму	Лит.	Маса / Масштаб
Перев.		Марковський				Д	
Т.контр.						Лист 1	Листів1
Н.контр.		Виноградов				КПІ ім. Ігоря Сікорського ФІОТ, ІВ-93	
Затв.							

ДОДАТОК 2

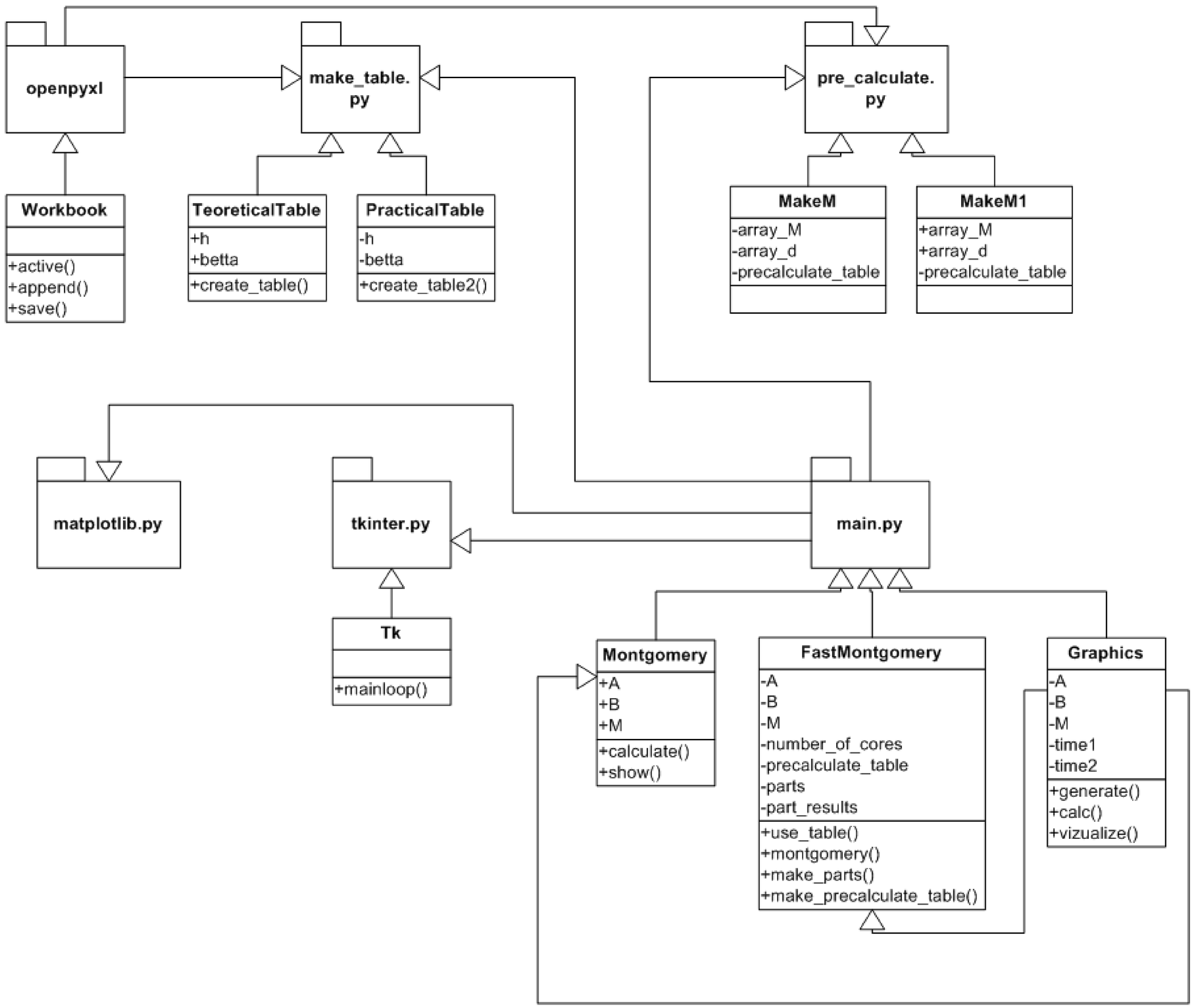
Метод мультипроцесорного модулярного множення з використанням групової редукції Монтгомері

Діаграма класів

ІАЛЦ.468243.005 Д2

Аркушів 1

Київ 2023 р



					ІАЛЦ.468243.005.Д2	
Зм	Лист	№докум.	Підп.	Дата	Метод мультипроцесорного множення Діаграма класів	Лит. Д; Маса; Масштаб
Розроб.		Трибунська				Лист 1; Листів1
Перев.		Марковський				
Т.контр.						КПІ ім. Ігоря Сікорського ФІОТ, ІВ-93
Н.контр.		Виноградов				
Затв.						

ДОДАТОК 3

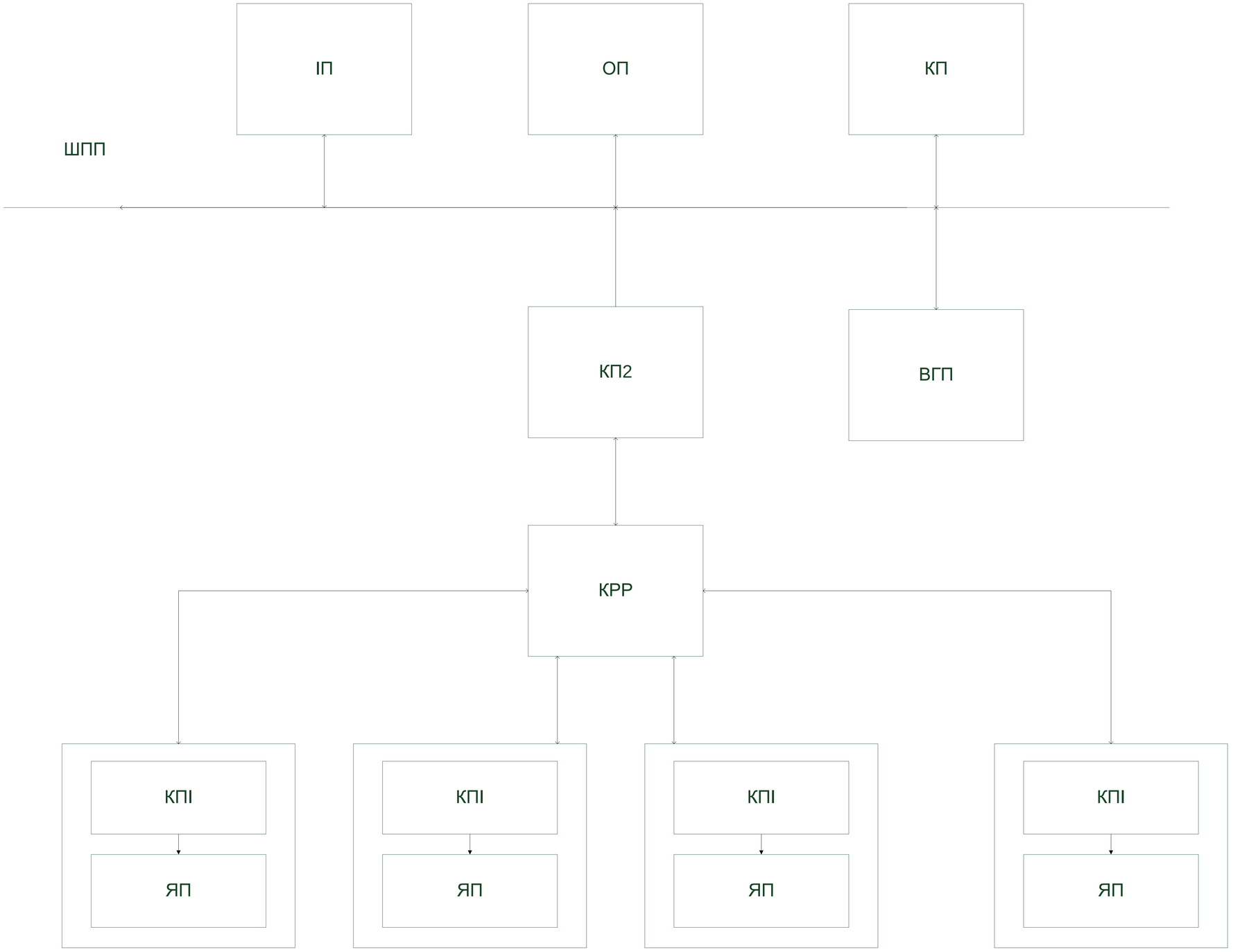
Система мультипроцесорного множення

Схема електрична структурна

ІАЛЦ.468243.006 Е1

Аркушів 1

Київ 2023 р



ВГП - векторний графічний процесор
ІП - інтегрований процесор
КОП – конструктор основної пам'яті
КП1 – кеш-пам'ять першого рівня
КП2- кеш-пам'ять другого рівня
КРР – контролер розподілу ресурсів
ОП – основна пам'ять
ЯП – процесорне ядро
ШПП – швидкісна процесорна шина

					ІАЛЦ.468243.006.Е1		
Зм	Лист	№докум.	Підп.	Дата	Метод мультипроцесорного множення Схема електрична структурна	Лит.	Маса / Масштаб
Розроб.		Трибунська				Д	
Перев.		Марковський				Лист 1	Листів1
Т.контр.						КПІ ім. Ігоря Сікорського ФІОТ, ІВ-93	
Н.контр.		Виноградов					
Затв.							

ДОДАТОК 4

Метод мультипроцесорного модулярного множення з використанням групової редукції

Текст програмного коду

ІАЛЦ.468243.007 Д3

Аркушів 15

Київ 2023 р

```
# import libraries
import matplotlib.pyplot as plt
from tkinter import *
import time
import random


class Montgomery:
  def __init__(self, A, B, M):
      self.A = A
      self.B = B
      self.M = M
      self.calculation_time = 0

  def calculate(self):
      a = self.A
      b = self.B
      m = self.M
      print(a * b % m)
      b = str(bin(b))[2:][::-1]
      res = 0

      for i in b:
          if i == "1":
              res += a
              self.calculation_time += 1
          if res % 2 == 1:
              res += m
              self.calculation_time += 1
          res = res >> 1
          self.calculation_time += 1
      res %= m
      res = res * 2 ** len(b) % m
      print(res)
      print(bin(m))
```

```
    print(len(bin(m))-2)

def show(self):
    center_num = 20
    string1 = ""
    print(f"A = {self.A}, B = {self.B}, M = {self.M}")
    a = self.A
    b = self.B
    m = self.M
    print(f"{a} * {b} % {m} = {a * b % m}")
    string1 += f"{a} * {b} % {m} = {a * b % m}" + "\n"
    b = str(bin(b))[2:][::-1]
    res = 0
    for i in b:
        print(i, end=" ")
        string1 += f"{i}".center(3)
        if i == "1":
            print(f"{res} + {a} = {res + a}".center(center_num), end=" ")
            string1 += f"{res} + {a} = {res + a}".center(center_num)
            res += a
        else:
            string1 += "-".center(center_num)
            print("-".center(center_num), end=" ")
        if res % 2 == 1:
            print(f"{res} + {m} = {res + m}".center(center_num), end=" ")
            string1 += f"{res} + {m} = {res + m}".center(center_num)
            res += m
        else:
            string1 += "-".center(center_num)
            print("-".center(center_num), end=" ")
        print(f"{res} / {2} = {res // 2}".center(center_num))
        string1 += f"{res} / {2} = {res // 2}".center(center_num) + "\n"
        res = res >> 1
    res %= m
    res = res * 2 ** len(b) % m
```

```
    print(res)
    return string1


class MontgomeryFast:
  def __init__(self, A, B, M, number_of_cores):
    self.A = A
    self.B = B
    self.M = M
    self.number_of_cores = number_of_cores
    self.precalculate_table = {}
    self.parts = []
    self.part_results = []
    self.calculation_time = 0
    self.make_precalculate_table()
    self.make_parts()
    self.montgomery()
    self.use_table()

  def make_precalculate_table(self):
    table = [0, 0]
    d = 4
    M = self.M
    array_M = []
    array_d = []
    for i in range(2 ** d):
      array_M.append(i * M)
      array_d.append(i)
    for i in array_d:
      for j in array_M:
        if bin(i + j)[-d::] == d * "0" or (i + j) == 0:
          self.precalculate_table[bin(i)[2::].zfill(d)] = j
          table.append(j)

    print(self.precalculate_table)
```

```
    return self.precalculate_table

def make_parts(self):
    sizes = []
    print(f"A = {self.A}, B = {self.B}, M = {self.M}")
    print(f"{self.A} * {self.B} % {self.M} = {self.A * self.B % self.M}")
    print(bin(self.A), bin(self.B), bin(self.M), sep="\n")

    for i in range(0, len(str(bin(self.B))) - 1, (len(str(bin(self.B))) - 2)//self.number_of_cores):
        sizes.append(i)
    print(sizes)
    B = str(bin(self.B))[2:]
    for i in range(len(sizes)-1):
        self.parts.append(B[sizes[i]:sizes[i + 1]][::-1])
    print(self.parts)

def montgomery(self):
    for b in self.parts:
        self.calculation_time = 0
        res = 0
        for i in b:
            if i == "1":
                res += self.A
                self.calculation_time += 1
            if res % 2 == 1:
                res += self.M
                self.calculation_time += 1
            res = res >> 1
            self.calculation_time += 1
        res %= self.M
        print(res)
        self.part_results.append(res)
    print(self.part_results)
    # print(self.calculation_time)
```

```
  def use_table(self):
      time_reduction = 0
      max_reduction_iterations = (((len(str(bin(self.B))) - 2) - (len(str(bin(self.B))) - 2)//self.number_of_cores))//4
      print(max_reduction_iterations)
      for i in range(len(self.part_results)):
          time_reduction = 0
          for j in range(i):
              self.part_results[i] += self.precalculate_table[str(bin(self.part_results[i]))[-4:]]
              time_reduction += 1
              self.part_results[i] = self.part_results[i] >> 4
              time_reduction += 1
          self.part_results[i] %= self.M
      self.calculation_time += time_reduction
      print(self.part_results)
      res = sum(self.part_results) * 2 ** (len(str(bin(self.B))) - 2) % self.M
      print(res)
      print(self.calculation_time)



class Graphic:
  def __init__(self):
      self.M = []
      self.A = []
      self.B = []

      self.time1 = []
      self.time2 = []

  def generate(self):
      for i in [64, 128, 256, 512, 1024, 2048]:
          self.A.append(random.randint(2**(i-1), 2**i))
          self.B.append(random.randint(2**(i-1), 2**i))
          self.M.append(random.randint(2**(2*i-1), 2**(2*i)))
```

```
  def calc(self):
      list_of_objects = []
      for i in range(len(self.M)):
          list_of_objects.append(Montgomery(self.A[i], self.B[i], self.M[i]))
          list_of_objects[i].calculate()
          self.time1.append(list_of_objects[i].calculation_time)

  def vizualize(self):
      x = [64, 128, 256, 512, 1024, 2048]
      y = self.time1
      print(y)

      x1 = [1, 2, 3, 4, 5]
      y1 = [2, 4, 6, 8, 10]
      plt.plot(x, y)
      # plt.plot(x1, y1)
      plt.show()

  def compare(self, set1, set2):
      # the function vizualizes the calculational speed of two methods
      pass


class MontgomeryTable:
  def __init__(self, A, B, M):
      self.A = A
      self.B = B
      self.M = M


def montgomery_calculation():
  def button_calculate():
      result = Montgomery(int(entry_A.get()), int(entry_B.get()), int(entry_M.get()))
      label_result["text"] = result.show()
  win = Toplevel(root)
```

```
  label_A = Label(win, text="Множник A:")
  entry_A = Entry(win)
  label_B = Label(win, text="Множник B:")
  entry_B = Entry(win)
  label_M = Label(win, text="Модуль M:")
  entry_M = Entry(win)
  result_button = Button(win, text="Обчислити результат:", command=button_calculate)
  label_result = Label(win, text="", justify=LEFT)

  label_A.pack()
  entry_A.pack()
  label_B.pack()
  entry_B.pack()
  label_M.pack()
  entry_M.pack()
  result_button.pack()
  label_result.pack()

def fast_montgomery_calculation():
  win = Toplevel(root)
  label_A = Label(win, text="Множник A:")
  entry_A = Entry(win)
  label_B = Label(win, text="Множник B:")
  entry_B = Entry(win)
  label_M = Label(win, text="Модуль M:")
  entry_M = Entry(win)
  result_button = Button(win, text="Обчислити результат:")
  label_result = Label(win, text="")

  label_A.pack()
  entry_A.pack()
  label_B.pack()
  entry_B.pack()
  label_M.pack()
  entry_M.pack()
```

```
  result_button.pack()
  label_result.pack()


def graphics():
  data = Graphic()
  data.generate()
  data.calc()
  data.vizualize()


root = Tk()


montgomery_calculate_button = Button(root, text="Обчислювання модулярного множення\n методом Монтгомері", command=montgomery_calculation)
fast_montgomery_calculate_button = Button(root, text="Обчислювання модулярного множення\n прискореним методом Монтгомері", command=fast_montgomery_calculation)
graphics_button = Button(root, text="Графіки швидкодії", command=graphics)


montgomery_calculate_button.pack()
fast_montgomery_calculate_button.pack()
graphics_button.pack()
root.mainloop()
```

graf.py

```
import matplotlib.pyplot as plt
from pylab import *

x = [64, 128, 256, 512, 1024, 2048]
y = [2*n for n in x]
y4 = []
y8 = []
```

```
r4 = 4
r8 = 8


def graf1(n, r):
    y = []
    for n in x:
        y.append(2*(n/r) + 0.5*(n - n/r))
    print(y)
    return y


y4 = graf1(x, r4)
y8 = graf1(x, r8)

plt.title("Оцінка швидкодії")

plt.plot(x, y4, "r--", label="4 процесори")
plt.plot(x, y8, "b--", label="8 процесорів")
plt.plot(x, y, "k-", label="Метод Монтгомері")

plt.legend()

plt.xlabel("Кількість бітів множника n")
plt.ylabel("Кількість операцій")
plt.show()
```

precalculate.py

```
from openpyxl import *
wb = Workbook()
ws = wb.active


def make_M(M, d):
```

```
    def sort_col(i):
        return i[1]

    array_M = []
    array_d = []
    precalculate_table = []
    ws.append(["d", "c", "c*M"])
    for i in range(2**d):
        array_M.append(i*M)
        array_d.append(i)
    for i in array_d:
        for j in array_M:
            if bin(i + j)[-d::] == d*"0":
                precalculate_table.append([bin(i)[2::].zfill(d), j//M, j])
                ws.append([bin(i)[2::].zfill(d), str(j//M), j])
    ws.append(["", ""])
    precalculate_table.sort(key=sort_col)
    ws.append(["d", "c", "c*M"])
    for i in range(len(precalculate_table)):
        ws.append(precalculate_table[i])

    print(precalculate_table)


def make_M1(M, d):
    def sort_col(i):
        return i[1]

    array_M = []
    array_d = []
    precalculate_table = []
    ws.append(["rl...r0", "D"])
    for i in range(2**d):
        array_M.append(i*M)
        array_d.append(i)
```

```
    for i in array_d:
        for j in array_M:
            if bin(i + j)[-d::] == d*"0" or (i + j) == 0:
                precalculate_table.append([bin(i)[2::].zfill(d), j, bin(j + i)[2::]])
                table.append(j)
                ws.append([bin(i)[2::].zfill(d), j, bin(j + i)[2::]])
    ws.append(["", ""])
    # precalculate_table.sort(key=sort_col)
    # ws.append(["d", "c", "c*M"])
    # for i in range(len(precalculate_table)):
    #     ws.append(precalculate_table[i])

    print(precalculate_table)



table = [0, 0]
make_M1(4057, 4)
print(table)

wb.save("pre_calculate.xlsx")
```

make_table.py

```
from openpyxl import *
wb = Workbook()
ws = wb.active



def create_table(k, q, n):
    h = []
    h.append(f"q = {q}")
    gamma = 2/(2 - 1.5/q)
    h1 = n*(gamma - 1)/(gamma**k - 1)
    h.append(h1)
    for i in range(k-1):
```

```
        h1 *= gamma
        h.append(h1)
    betta = n/h[-1]
    h.append(betta)
    print(h)
    ws.append(h)


def create_table2(k, q, n):
    h = []
    h.append(f"q = {q}")
    gamma = 2/(2 - 1.5/q)
    h1 = n*(gamma - 1)/(gamma**k - 1)
    h.append(h1)
    for i in range(k-1, 0, -1):
        h1 *= gamma
        h.append(h1)
    for i in range(1, len(h)):
        a = abs(h[i] - (int(h[i]) + q - int(h[i]) % q))
        b = abs(h[i] - (int(h[i]) - int(h[i]) % q))
        if a < b:
            h[i] = int(h[i]) + q - int(h[i]) % q
        else:
            h[i] = int(h[i]) - int(h[i]) % q

    h[1] = 2048 - sum(h[2::])
    betta = n/h[-1]
    h.append(betta)
    print(h)
    ws.append(h)


title1 = []
title1.append(" ")
for i in range(1, 5):
```

```
    title1.append("h" + str(i))
    if i == 4:
        title1.append("betta")

ws.append(["Theoretical"])
ws.append([" ", " "])
ws.append(title1)
print(title1)

create_table(4, 4, 2048)
create_table(4, 8, 2048)
create_table(4, 16, 2048)
create_table(4, 32, 2048)

ws.append([" ", " "])

title2 = []
title2.append(" ")
for i in range(1, 9):
    title2.append("h" + str(i))
    if i == 8:
        title2.append("betta")

ws.append(title2)
print(title2)

create_table(8, 4, 2048)
create_table(8, 8, 2048)
create_table(8, 16, 2048)
create_table(8, 32, 2048)

#-------------------------------------------------

title1 = []
title1.append(" ")
```

```
for i in range(1, 5):
    title1.append("h" + str(i))
    if i == 4:
        title1.append("betta")

ws.append(["Practical"])
ws.append([" ", " "])
ws.append(title1)
print(title1)

create_table2(4, 4, 2048)
create_table2(4, 8, 2048)
create_table2(4, 16, 2048)
create_table2(4, 32, 2048)

ws.append([" ", " "])

title2 = []
title2.append(" ")
for i in range(1, 9):
    title2.append("h" + str(i))
    if i == 8:
        title2.append("betta")

ws.append(title2)
print(title2)

create_table2(8, 4, 2048)
create_table2(8, 8, 2048)
create_table2(8, 16, 2048)
create_table2(8, 32, 2048)

ws.append([" ", " "])

title3 = []
```

```
title3.append(" ")
for i in range(1, 17):
    title3.append("h" + str(i))
    if i == 16:
        title3.append("betta")

ws.append(title3)
print(title3)

create_table2(16, 4, 2048)
create_table2(16, 8, 2048)
create_table2(16, 16, 2048)
create_table2(16, 32, 2048)
create_table2(16, 64, 2048)
create_table2(16, 128, 2048)

wb.save("make_table.xlsx")
```