Практичне завдання:

**“Реалізація алгоритму гешування”**

**Мета лабораторної роботи:** вивчити принципи роботи алгоритмів гешування та реалізувати один з них

# Теоретичні відомості

Розвиток криптографії почався ще з давніх-давен, а її історія налічує понад 4 тисячі років. Спочатку криптографія вирішувала лише одне завдання – забезпечення конфіденційності під час передачі та зберігання даних. На даний момент криптографія також використовується для забезпечення цілісності та автентичності даних.

На сьогоднішній день криптографія ґрунтується на математичних операціях, визначених у таких розділах математики, як теорія чисел, теорія груп, кілець, полів тощо.

Сучасна криптографія складається із схем симетричного та асиметричного шифрування, схем хешування даних, схем цифрового підпису, методів управління ключами, схем доказів з нульовим розголошенням, методів криптоаналізу та постквантової криптографії тощо.

Основним завданням інформаційної безпеки є забезпечення:

* конфіденційність даних,
* цілісність даних,
* доступність даних,
* автентичність даних.

*Конфіденційність* передбачає, що неавторизовані особи не можуть отримати доступ до даних, що зберігаються або передаються. Ця послуга безпеки може забезпечуватись за допомогою алгоритмів шифрування.

*Цілісність* передбачає, що дані не було змінено під час передачі чи виконання операції над ними. Для перевірки цілісності даних використовують хеш-функції.

*Доступність* передбачає, що суб'єкти, які мають право доступу до інформації, гарантовано отримають доступ до неї. Ця послуга безпеки не може забезпечуватися під час використання криптографічних методів.

*Автентичність* передбачає можливість доказу, що дані справді були отримані від конкретного автора. Перевірка автентичності даних виконується за допомогою механізму цифрового підпису (або MAC кодів).

## 1.1 Геш функції

Геш-функція перетворює дані довільної довжини рядок фіксованої довжини. Завдяки властивостям геш-функцій неможливо отримати вихідні дані, маючи лише їхнє хеш-значення. При внесенні навіть незначних змін у вихідному значенні, результуюче значення значно відрізнятиметься.

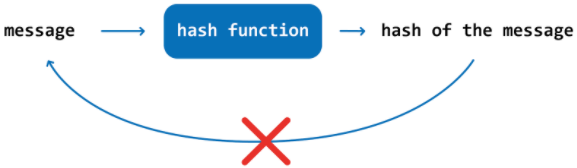


Рисунок 1.1 – Принцип роботи геш-функції

# Хід роботи

Для виконання лабораторної роботи ви можете використовувати будь-яку зручну для вас мову програмування. Завдання полягає у реалізації одного з запропонованих алгоритмів (можна обрати для реалізації такий алгоритм, який не описаний у даному методичному посібнику).

Опції виконання лабораторної роботи:

| Варіант | Алгоритм для реалізації | Basic level | Advanced level |
| --- | --- | --- | --- |
| 2 | Геш функція | SHA-1 | Keccak або будь-який інший алгоритм на власний розсуд. |

Зазначимо, що вибір рівня виконання завдання не передбачає необхідність реалізації всіх зазначених алгоритмів. Реалізація алгоритмів у кількості більше за 1 заохочується, але не є необхідною.

## 

## 2.1 SHA-1

Фактично, особливості функціонування SHA-1 полягає в циклічному перемішуванні та використанні основних бітових операцій над вхідними даними. Схема роботи одного циклу алгоритму SHA-1 виглядає так:

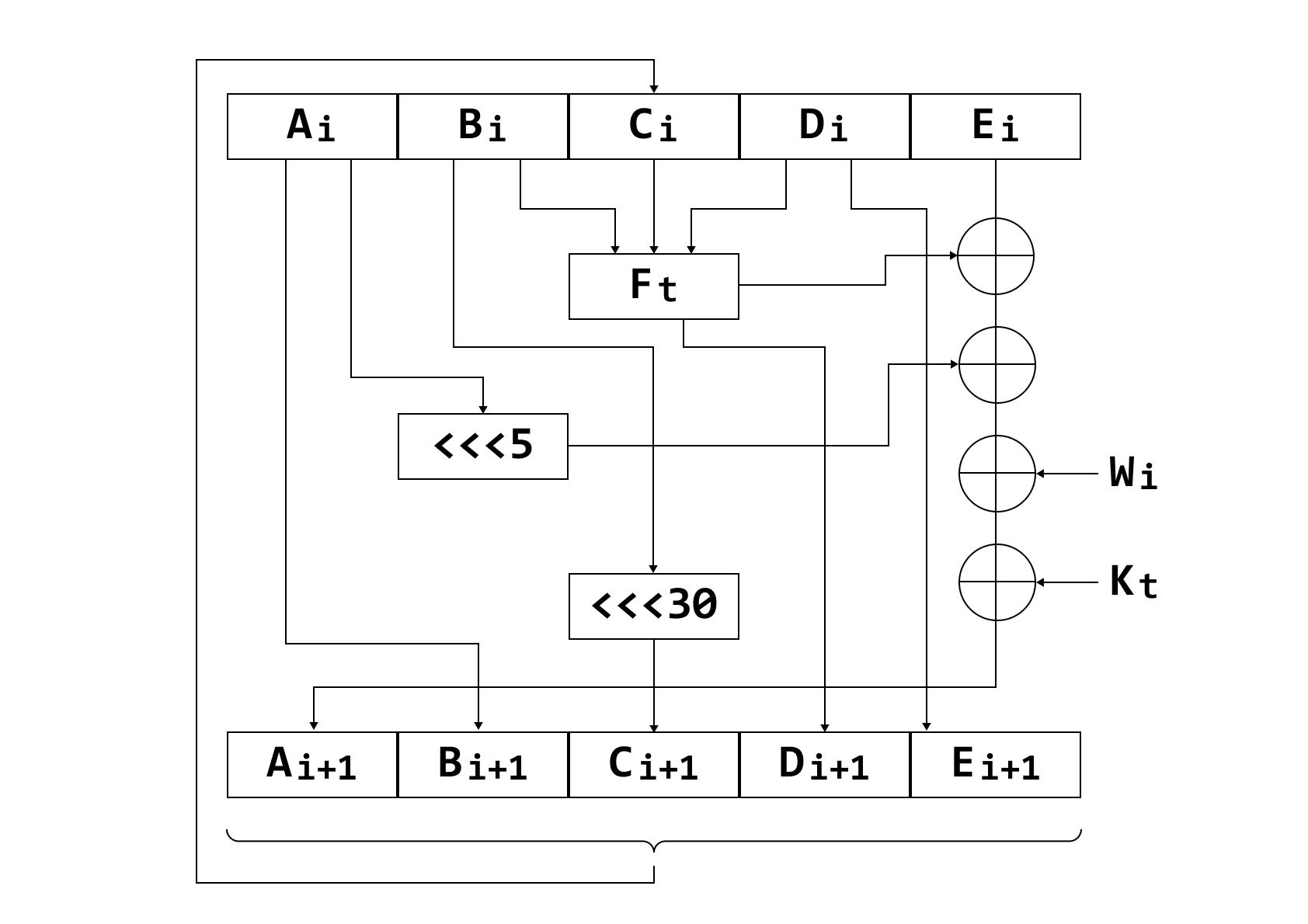


Рисунок 2.7 – Схема виконання одного циклу алгоритму SHA-1

де – константи;

– змінна функція (змінюється кожні 20 циклів);

– модифікований елемент вхідного повідомлення (4 байт);

<<< *x* – циклічний зсув вліво.

Сам текст стандарту та тестові вектори ви можете знайти за наступним посиланням:

<https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf>

**Pseudo-code:**

function GetSHA1Hash (message)

begin

h0 = 0x67452301

h1 = 0xEFCDAB89

h2 = 0x98BADCFE

h3 = 0x10325476

h4 = 0xC3D2E1F0 //variables initialization

ml = message length in bits

break message into 512-bit chunks (after message processing)

for each chunk

break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

for i from 16 to 79

w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrot 5

a = h0

b = h1

c = h2

d = h3

e = h4

for i from 0 to 79 {

if 0 ≤ i ≤ 19 then

f = (b and c) or ((not b) and d)

k = 0x5A827999

else if 20 ≤ i ≤ 39

f = b xor c xor d

k = 0x6ED9EBA1

else if 40 ≤ i ≤ 59

f = (b and c) or (b and d) or (c and d)

k = 0x8F1BBCDC

else if 60 ≤ i ≤ 79

f = b xor c xor d

k = 0xCA62C1D6

temp = (a leftrot 5) + f + e + k + w[i]

e = d

d = c

c = b leftrotate 30

b = a

a = temp

}

h0 = h0 + a

h1 = h1 + b

h2 = h2 + c

h3 = h3 + d

h4 = h4 + e

hash = (h0 lshift 128) or (h1 lshift 96) or (h2 lshift 64) or (h3 lshift 32) or h4 // concat

## 2.2 Keccak

Keccak (SHA-3) – алгоритм хешування змінної розрядності. Геш-функція побудована на основі конструкції криптографічної губки, в якій дані спочатку вбираються в губку, при цьому вихідне повідомлення *M* зазнає багатораундових перестановок *f*, а потім результат *Z* «віджимається» з губки. На етапі «вбирання» блоки повідомлення сумуються за модулем 2 з підмножиною стану, після чого весь стан перетворюється за допомогою функції перестановки *f*. На етапі «віджимання» вихідні блоки зчитуються з одного і того ж підмножини стану, зміненого функцією перестановок *f*.

Основою функції стиснення алгоритму є функція *f*, вщо виконує перемішування внутрішнього стану алгоритму. Стан A представляється у вигляді масиву 5×5, елементами якого є 64-бітові слова, ініціалізовані нульовими бітами (тобто розмір стану становить 1600 бітів). Функція f виконує 24 раунди, у кожному з яких виробляються покрокові перетворення θ, ρ, π, χ, ι.

На початку перетворення виконується padding (заповнення/додавання). Padding необхідний для приведення вихідного повідомлення до рядка фіксованої довжини. Потім виконується функція "Absorb". Ця функція призначена для розбиття вхідного (що пройшла через padding) рядки на складові, які будуть послідовно використовуватися для наступних перетворень:

* 1. θ перетворення;
  2. ρ і π перетворення;
  3. χ перетворення;
  4. ι перетворення.

Після цього виконується процедура Squeezing (стиснення/видавлювання). Ця процедура необхідна для обчислення кінцевого хеш-значення.

**Pseudo-code:**

Keccak[r,c](Mbytes || Mbits) {

# Padding

d = 2^|Mbits| + sum for i=0..|Mbits|-1 of 2^i\*Mbits[i]

P = Mbytes || d || 0x00 || … || 0x00

P = P xor (0x00 || … || 0x00 || 0x80)

# Initialization

S[x,y] = 0, for (x,y) in (0…4,0…4)

# Absorbing phase

for each block Pi in P

S[x,y] = S[x,y] xor Pi[x+5\*y], for (x,y) such that x+5\*y < r/w

S = **Keccak-f[r+c](S)**

# Squeezing phase

Z = empty string

while output is requested

Z = Z || S[x,y], for (x,y) such that x+5\*y < r/w

S = **Keccak-f[r+c](S)**

return Z

}

Постійні значення для RC[i]:

RC[0] 0x0000000000000001 RC[12] 0x000000008000808B

RC[1] 0x0000000000008082 RC[13] 0x800000000000008B

RC[2] 0x800000000000808A RC[14] 0x8000000000008089

RC[3] 0x8000000080008000 RC[15] 0x8000000000008003

RC[4] 0x000000000000808B RC[16] 0x8000000000008002

RC[5] 0x0000000080000001 RC[17] 0x8000000000000080

RC[6] 0x8000000080008081 RC[18] 0x000000000000800A

RC[7] 0x8000000000008009 RC[19] 0x800000008000000A

RC[8] 0x000000000000008A RC[20] 0x8000000080008081

RC[9] 0x0000000000000088 RC[21] 0x8000000000008080

RC[10] 0x0000000080008009 RC[22] 0x0000000080000001

RC[11] 0x000000008000000A RC[23] 0x8000000080008008

**Pseudo-code of Keccak-f:**

Keccak-f[b](A) {

for i in 0…n-1 // n=24

A = Round[b](A, RC[i])

return A

}

Round[b](A,RC) {

# θ step

C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4], for x in 0…4

D[x] = C[x-1] xor rot(C[x+1],1), for x in 0…4

A[x,y] = A[x,y] xor D[x], for (x,y) in (0…4,0…4)

# ρ and π steps

B[y,2\*x+3\*y] = rot(A[x,y], r[x,y]), for (x,y) in (0…4,0…4)

# χ step

A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]), for (x,y) in (0…4,0…4)

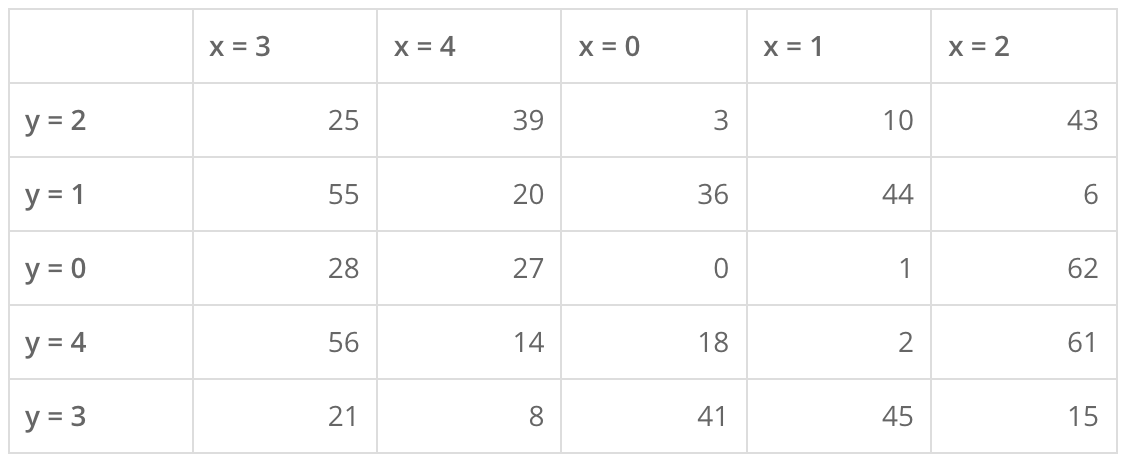
# ι step

A[0,0] = A[0,0] xor RC

return A

}

Постійні значення *r*[x,y]:



Сам текст стандарту та тестові вектори ви можете знайти за наступним посиланням:

<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>

## 