Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Специальность: 1-40 81 02 «Технологии виртуализации и облачных вычислений»

ОТЧЕТ

ЛАБОРАТОРНАЯ РАБОТА

по курсу «Прикладная криптография»

Выполнил: А.И. Андрадэ

магистрант группы 758441

Проверил: А.В. Станкевич

доцент кафедры ЭВС, к.т.н.

МИНСК 2018

СОДЕРЖАНИЕ

[**1 sha-256** 3](#_Toc531303285)

[**2 Псевдокод** 4](#_Toc531303286)

[**3 Реализация VHDL** 7](#_Toc531303287)

[**4 Тест** 10](#_Toc531303288)

# **1 sha-256**

Аббревиатура SHA 256 расшифровывается как Secure Hashing Algorithm – востребованный механизм хэширования, созданный специалистами из NSA. Ключевая задача алгоритма – преобразование случайной информации в значения с фиксированной длиной, в дальнейшем оно будет использоваться с целью идентификации данных сведений.

* 1. Технические параметры

Протокол предназначен для данных, которые разделены на части, объем каждой – 64 байта. Алгоритм обеспечивает консолидацию, вследствие которой и появляется 256-битный код. Основывается технология шифрования на относительно несложном раунде, цикличность которого – 64 раза.

* 64-байтный размер блока;
* Максимальная длина зашифровываемого кода – 33 byte;
* Параметры дайджеста сообщения – 32 byte;
* Размер слова по умолчанию – 4 byte;
* Количество повторений в рамках одного цикла – 64;

Как уже было сказано ранее, протокол SHA-256 основывается на концепции Меркла-Дамгарда, а значит, сначала проводится разделение на блоки, а уже потом на отдельные слова.

Набор информации проходит сквозь диапазон повторений – 64 или 80. Каждый цикл сопровождается преобразованием блок слов. Итоговый хэш-код генерируется посредством суммирования первоначальных значений.

**2 Псевдокод**

*Пояснения:*

*message* — *исходное двоичное сообщение*

*m* — *преобразованное сообщение*

*Инициализация переменных*

(первые 32 бита дробных частей *квадратных корней первых восьми простых* *чисел [от 2 до 19]):*

h0 := 0x6A09E667

h1 := 0xBB67AE85

h2 := 0x3C6EF372

h3 := 0xA54FF53A

h4 := 0x510E527F

h5 := 0x9B05688C

h6 := 0x1F83D9AB

h7 := 0x5BE0CD19

*Таблица констант*

*(первые 32 бита* дробных частей *кубических корней первых 64 простых чисел* *[от 2 до 311]):*

k[0..63] :=

0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5, 0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,

0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,

0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC, 0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,

0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7, 0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,

0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13, 0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,

0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3, 0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,

0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5, 0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,

0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208, 0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2

*Предварительная обработка:*

m := message ǁ [*единичный бит*]

m := m ǁ [k *нулевых бит*], *где* ***k****— наименьшее неотрицательное число, такое что (L + 1 + K) mod 512 = 448, где L — число бит в сообщении (сравнима по модулю 512 c 448)*

m := m ǁ Длина(message) — *длина исходного сообщения в битах в виде 64-битного числа с порядком байтов от старшего к младшему*

*Далее сообщение обрабатывается последовательными порциями по 512 бит:*

разбить сообщение на куски по 512 бит

**для** каждого куска

разбить кусок на 16 слов длиной 32 бита (с порядком байтов от старшего к младшему внутри слова): w[0..15]

*Сгенерировать дополнительные 48 слов:*

**для** i **от** 16 **до** 63

s0 := (w[i-15] **rotr** 7) **xor** (w[i-15] **rotr** 18) **xor** (w[i-15] **shr** 3)

s1 := (w[i-2] **rotr** 17) **xor** (w[i-2] **rotr** 19) **xor** (w[i-2] **shr** 10)

w[i] := w[i-16] **+** s0 **+** w[i-7] **+** s1

*Инициализация вспомогательных переменных:*

a := h0

b := h1

c := h2

d := h3

e := h4

f := h5

g := h6

h := h7

*Основной цикл:*

**для** i **от** 0 **до** 63

Σ0 := (a **rotr** 2) **xor** (a **rotr** 13) **xor** (a **rotr** 22)

Ma := (a **and** b) **xor** (a **and** c) **xor** (b **and** c)

t2 := Σ0 + Ma

Σ1 := (e **rotr** 6) **xor** (e **rotr** 11) **xor** (e **rotr** 25)

Ch := (e **and** f) **xor** ((**not** e) **and** g)

t1 := h + Σ1 + Ch + k[i] + w[i]

h := g

g := f

f := e

e := d + t1

d := c

c := b

b := a

a := t1 + t2

*Добавить полученные значения к ранее вычисленному результату:*

h0 := h0 + a

h1 := h1 + b

h2 := h2 + c

h3 := h3 + d

h4 := h4 + e

h5 := h5 + f

h6 := h6 + g

h7 := h7 + h

*Получить итоговое значение хеша:*

digest = hash = h0 ǁ h1 ǁ h2 ǁ h3 ǁ h4 ǁ h5 ǁ h6 ǁ h7

1. **Реализация VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity sha\_256 is

Port ( message : in STD\_LOGIC\_VECTOR (255 downto 0);

digest : out STD\_LOGIC\_VECTOR (255 downto 0));

end sha\_256;

architecture Behavioral of sha\_256 is

*--Initialize hash values: The hash values are initialized to the first*

*--32 bits of the fractional parts of the square roots of the first 8*

*--prime numbers (2..19).*

type vector\_array\_8by32 is array (0 to 7) of STD\_LOGIC\_VECTOR(31 downto 0);

constant h\_original : vector\_array\_8by32 :=

(x"6a09e667", x"bb67ae85", x"3c6ef372", x"a54ff53a", x"510e527f",

x"9b05688c", x"1f83d9ab", x"5be0cd19");

*--Initialize array of round constants: Use first 32 bits of fractional parts*

*--of the cubed roots of the first 64 primes (2..311).*

type vector\_array\_64by32 is array (0 to 63) of STD\_LOGIC\_VECTOR(31 downto 0);

constant k : vector\_array\_64by32 :=

(x"428a2f98", x"71374491", x"b5c0fbcf", x"e9b5dba5", x"3956c25b", x"59f111f1",

x"923f82a4", x"ab1c5ed5", x"d807aa98", x"12835b01", x"243185be", x"550c7dc3",

x"72be5d74", x"80deb1fe", x"9bdc06a7", x"c19bf174", x"e49b69c1", x"efbe4786",

x"0fc19dc6", x"240ca1cc", x"2de92c6f", x"4a7484aa", x"5cb0a9dc", x"76f988da",

x"983e5152", x"a831c66d", x"b00327c8", x"bf597fc7", x"c6e00bf3", x"d5a79147",

x"06ca6351", x"14292967", x"27b70a85", x"2e1b2138", x"4d2c6dfc", x"53380d13",

x"650a7354", x"766a0abb", x"81c2c92e", x"92722c85", x"a2bfe8a1", x"a81a664b",

x"c24b8b70", x"c76c51a3", x"d192e819", x"d6990624", x"f40e3585", x"106aa070",

x"19a4c116", x"1e376c08", x"2748774c", x"34b0bcb5", x"391c0cb3", x"4ed8aa4a",

x"5b9cca4f", x"682e6ff3", x"748f82ee", x"78a5636f", x"84c87814", x"8cc70208",

x"90befffa", x"a4506ceb", x"bef9a3f7", x"c67178f2");

begin

*--This process statement contains the workings of the hash algorithm. The inputs include*

*--the original message array and the arays of constants defined above. The output is*

*--the final processed hash value.*

algorithm: process(message)

*--General Variable Declarations (See working variable declaration below)*

variable w : vector\_array\_64by32;

variable temp\_message : STD\_LOGIC\_VECTOR(511 downto 0) := (OTHERS => '0');

variable s0: STD\_LOGIC\_VECTOR(31 downto 0);

variable s1: STD\_LOGIC\_VECTOR(31 downto 0);

variable ch: STD\_LOGIC\_VECTOR(31 downto 0);

variable temp1: STD\_LOGIC\_VECTOR(31 downto 0);

variable temp2: STD\_LOGIC\_VECTOR(31 downto 0);

variable maj: STD\_LOGIC\_VECTOR(31 downto 0);

variable h\_new : vector\_array\_8by32;

--Declare Working Variables.

variable a : STD\_LOGIC\_VECTOR(31 downto 0);

variable b : STD\_LOGIC\_VECTOR(31 downto 0);

variable c : STD\_LOGIC\_VECTOR(31 downto 0);

variable d : STD\_LOGIC\_VECTOR(31 downto 0);

variable e : STD\_LOGIC\_VECTOR(31 downto 0);

variable f : STD\_LOGIC\_VECTOR(31 downto 0);

variable g : STD\_LOGIC\_VECTOR(31 downto 0);

variable h : STD\_LOGIC\_VECTOR(31 downto 0);

begin

*--Initialize to first values of hash constants*.

initialize\_h: for i in 0 to 7 loop

h\_new(i) := h\_original(i);

end loop initialize\_h;

*--Preprocessing: Append the bit '1' to message, followed by zeros and the length of the message as a 64-bit big-endian integer*

*--Initialize message array and pad with zeros.*

temp\_message(511 downto 256) := message;

temp\_message(255) := '1';

temp\_message(63 downto 0) := STD\_LOGIC\_VECTOR(to\_unsigned(message'length, 64)); *-- length of message as a 64-bit big-endian integer*

*--Process Message in Successive 512-bit Chuncks*

*--Initialize message array.*

init\_w: for i in 0 to 63 loop

w(i) := (OTHERS => '0');

end loop init\_w;

*--Copy chunk into first 16 words of message schedule array.*

copy\_first16: for i in 0 to 15 loop

w(i) := temp\_message((32\*(16-i))-1 downto (32\*(15-i)));

*--0 -> 511 downto 480*

*--1 -> 479 downto 448*

*--...*

*--15 -> 31 downto 0*

end loop copy\_first16;

*--Extend first 16 words into remaining 48 words of message schedule array.*

extend\_next48: for i in 16 to 63 loop

s0 := STD\_LOGIC\_VECTOR(rotate\_right(unsigned(w(i-15)), 7))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(w(i-15)), 18))

XOR STD\_LOGIC\_VECTOR(shift\_right(unsigned(w(i-15)), 3));

s1 := STD\_LOGIC\_VECTOR(rotate\_right(unsigned(w(i-2)), 17))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(w(i-2)), 19))

XOR STD\_LOGIC\_VECTOR(shift\_right(unsigned(w(i-2)), 10));

w(i) := STD\_LOGIC\_VECTOR(unsigned(w(i-16)) + unsigned(s0) + unsigned(w(i-7)) + unsigned(s1));

end loop extend\_next48;

*--Initialize Working Variables to Current Hash Value*

a := h\_original(0);

b := h\_original(1);

c := h\_original(2);

d := h\_original(3);

e := h\_original(4);

f := h\_original(5);

g := h\_original(6);

h := h\_original(7);

*--Compression Function Main Loop*

--Only the first iteration of the loop is implemented here because the NEXYS 2 board used in this project

*--could not supply adequate resources for running 64 iterations simultaneously. To generate a complete*

*--hash, simply change the second index from 0 to 63.*

compression\_adjustments: for index1 in 0 to 63 loop

s1 := STD\_LOGIC\_VECTOR(rotate\_right(unsigned(e), 6))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(e), 11))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(e), 25));

ch := (e AND f) XOR ((NOT e) AND g);

temp1 := STD\_LOGIC\_VECTOR(unsigned(h) + unsigned(s1) + unsigned(ch) + unsigned(k(index1)) + unsigned(w(index1)));

s0 := STD\_LOGIC\_VECTOR(rotate\_right(unsigned(a), 2))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(a), 13))

XOR STD\_LOGIC\_VECTOR(rotate\_right(unsigned(a), 22));

maj := (a AND b) XOR (a AND c) XOR (b AND c);

temp2 := STD\_LOGIC\_VECTOR(unsigned(s0) + unsigned(maj));

h := g;

g := f;

f := e;

e := STD\_LOGIC\_VECTOR(unsigned(d) + unsigned(temp1));

d := c;

c := b;

b := a;

a := STD\_LOGIC\_VECTOR(unsigned(temp1) + unsigned(temp2));

end loop compression\_adjustments;

*--Add Compressed Chunk to Current Hash Value*

h\_new(0) := STD\_LOGIC\_VECTOR(unsigned(h\_new(0)) + unsigned(a));

h\_new(1) := STD\_LOGIC\_VECTOR(unsigned(h\_new(1)) + unsigned(b));

h\_new(2) := STD\_LOGIC\_VECTOR(unsigned(h\_new(2)) + unsigned(c));

h\_new(3) := STD\_LOGIC\_VECTOR(unsigned(h\_new(3)) + unsigned(d));

h\_new(4) := STD\_LOGIC\_VECTOR(unsigned(h\_new(4)) + unsigned(e));

h\_new(5) := STD\_LOGIC\_VECTOR(unsigned(h\_new(5)) + unsigned(f));

h\_new(6) := STD\_LOGIC\_VECTOR(unsigned(h\_new(6)) + unsigned(g));

h\_new(7) := STD\_LOGIC\_VECTOR(unsigned(h\_new(7)) + unsigned(h));

*--Produce Final Hash Value (Big-Endian)*

digest <= h\_new(0) & h\_new(1) & h\_new(2) & h\_new(3) & h\_new(4) &

h\_new(5) & h\_new(6) & h\_new(7);

end process algorithm;

end Behavioral;

1. **Тест**

В качестве теста захешируем строку 0x0000000000000000000000000000000000000000000000000000000000000003

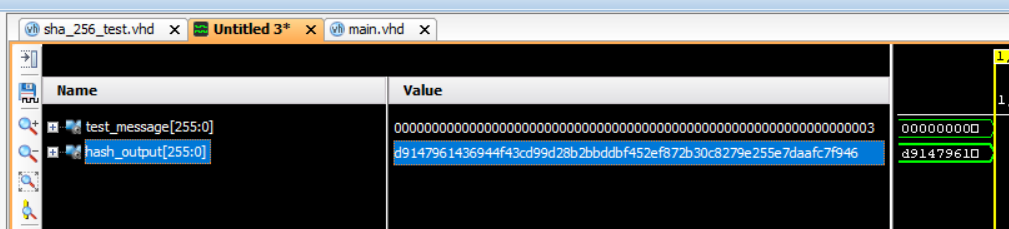


Рис 1 – Результат симуляции в Vivado

Хеширование с hashlib на Python

import hashlib

from bitstring import Bits

test\_input = Bits(bin=("0"\*254)+"11")

print(hashlib.sha256(test\_input.bytes).hexdigest())

Результат

d9147961436944f43cd99d28b2bbddbf452ef872b30c8279e255e7daafc7f946