CASE STUDY

**SHA-256**

**Submitted By**

Varun Kumar LS

INTRODUCTION SHA-256

* A cryptographic hash function takes an arbitrary block of data and returns a fixed-size bit string.
* The data to be encoded is often called the "message," and the hash value is sometimes called the message digest or simply digest.
* Used in digital signatures and message authentication codes.

**Function of SHA-256**

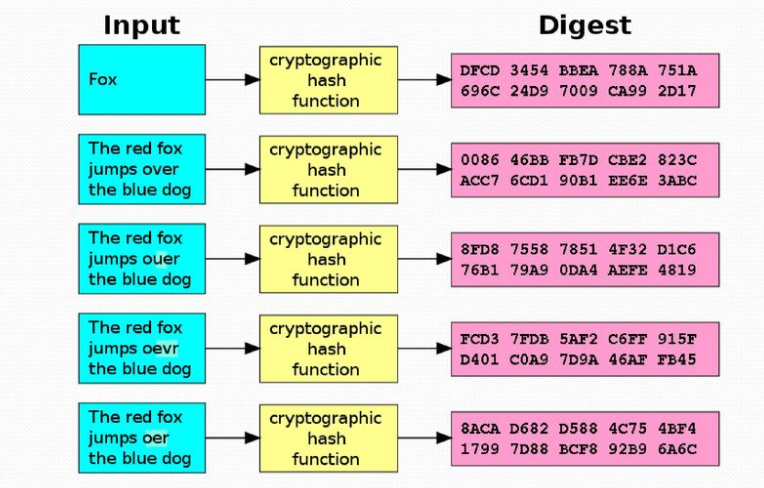


Fig1: Cryptographic hash function

**Properties of SHA:**

* It is easy to compute the hash value for any given message
* It is infeasible to generate a message that has a given hash
* It is infeasible to modify a message without changing the hash
* It is infeasible to find two different messages with the same hash

**Where is SHA-256**

* Cryptographic Applications : Used in password hashing
* Blockchain & Cryptocurrencies: Bitcoin mining uses SHA-256 in Proof-of-Work (PoW).

Ensures transaction integrity and block validation.

* Digital Signatures & Authentication
* Secure Communications (TLS, SSL):

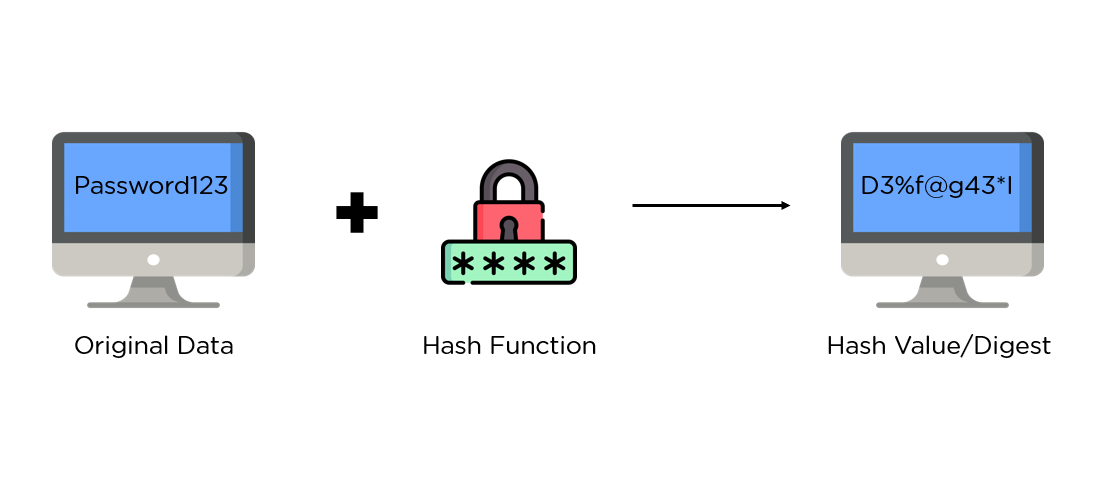


Fig 2:  hash value/digest

**SHA-256 Constants**

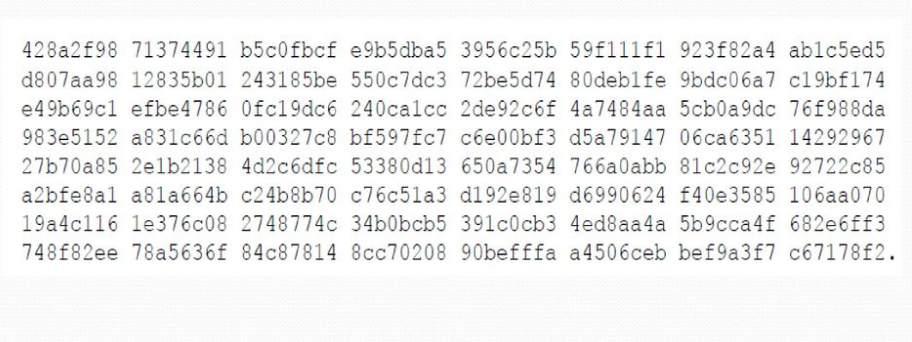


Fig 3 : Digest Constants

The image contains SHA-256 constants, likely the 64 round constants (K0–K63) used in the compression function. These constants are derived from the fractional parts of the cube roots of the first 64 prime numbers and help in ensuring diffusion and security in the hashing process.

**Comparison of SHA Family**

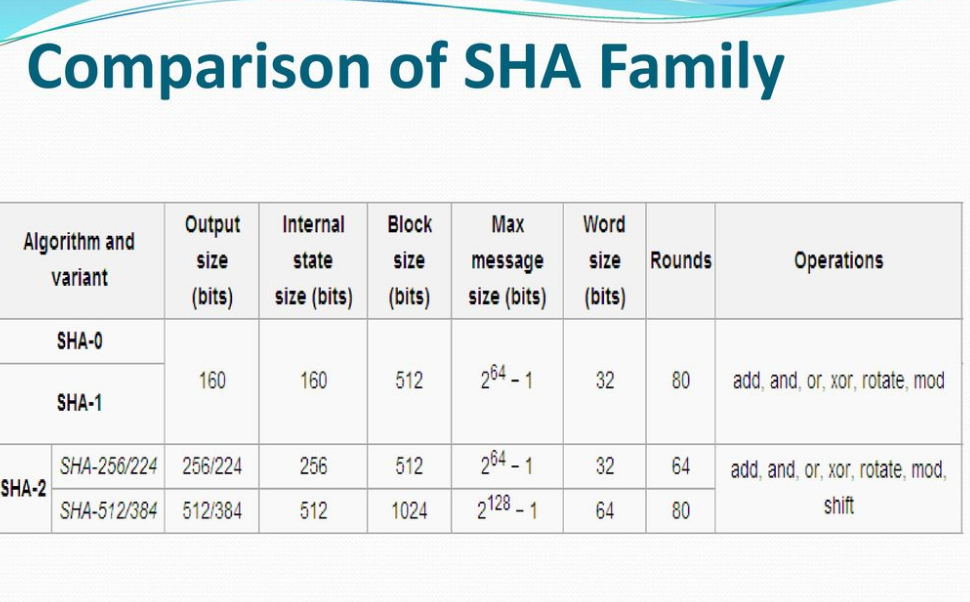


Fig 4: SHA Family Table

* SHA-256 and SHA-512 offer stronger security than SHA-1.
* Higher bit sizes (SHA-512) provide better resistance to attacks.
* More rounds improve security but increase computation time.

**Steps involved in SHA (Secure Hash Algorithm):**

1. **Preprocessing**
2. **Hashing**

**Preprocessing**

It consists of three steps:

* Padding
* Parsing
* Setting initial hash value

**Padding**

* The message, M, shall be padded before hash computation begins.
* The purpose of padding is to ensure that the padded message is a multiple of 512 or 1024 bits, depending on the algorithm.
* SHA-1 and SHA-256: 1+1+k=448 mod  5121 + 1 + k = 448 \mod 5121+1+k=448mod512
* SHA-384 and SHA-512: 1+1+k=896 mod  10241 + 1 + k = 896 \mod 10241+1+k=896mod1024

**Parsing**

* After a message has been padded, it must be parsed into Nm-bit blocks before the hash computation can begin.
* SHA-1 and SHA-256
* SHA-384 and SHA-512

**Appendix**

**Complete Verilog code:**

* **Verilog code for SHA-256:**

module sha256(input\_data, input\_valid, input\_ready, last\_word, clk, rst, output\_valid, hash\_data);

// Registers and control signals for SHA-256 processing

reg [4:0] length\_counter;

reg [63:0] size;

reg [1:0] special;

reg last\_word\_delayed;

reg [1:0] last\_next;

// Reset logic - Initializes control registers

always @(negedge rst)

begin

input\_ready <= 1'b1;

length\_counter <= 5'd31;

special <= 2'd0;

size <= 64'hffffffffffffffe0;

last\_word\_delayed <= 1'b0;

end

// Instantiation of SHA-256 processing engine

sha\_engine Eng(output\_data, clk, block\_counter, rst, output\_valid, hash\_data, last\_word, last\_next);

// Input handling - Stops accepting input at 14th word

always @(posedge clk)

begin

if(input\_valid && input\_ready)

begin

if(length\_counter == 5'd14)

input\_ready <= 1'b0;

length\_counter = length\_counter + 1;

end

end

// Delay last\_word signal for padding logic

always @(posedge clk)

begin

if(last\_word)

last\_word\_delayed <= 1'b1;

end

// Determines padding strategy based on message length

always @(posedge last\_word\_delayed)

begin

if(length\_counter <= 13)

special <= 2'b01;

else

special <= 2'b11;

end

// Data processing and padding logic

always @(posedge clk)

begin

case(special)

2'b00 : output\_data <= input\_data; // Normal data processing

2'b01 : last\_next <= 2'b01; // First padding block

2'b11 : last\_next <= 2'b10; // Final padding block

endcase

end

// Reset counters after block processing

always @(posedge clk)

begin

if(block\_counter == 64)

begin

length\_counter <= 5'd31;

input\_ready <= 1'b1;

end

end

// Update total message size

always @(posedge clk)

begin

if(input\_ready && special == 0)

size <= size + 32;

end

endmodule

* **Verilog code for SHA\_engine 256:**

module sha\_engine(word, clk, index, rst, output\_valid, hash\_data, last\_word, last\_next);

// Internal registers for SHA-256 computation

reg [31:0] W[63:0]; // Message schedule array

reg [31:0] H[7:0]; // Hash value storage

reg [31:0] K[63:0]; // SHA-256 constants

reg [31:0] a, b, c, d, e, f, g, h; // Working variables

reg [31:0] sum0, sum1, Ch, Maj, T1, T2; // Intermediate values

reg D; // Final block flag

output reg output\_valid;

output [255:0] hash\_data;

output reg [6:0] index;

// Reset logic - Initializes hash values and constants

always @(negedge rst)

begin

index = 7'd126;

output\_valid = 1'b0;

H[0] = 32'h6a09e667;

H[1] = 32'hbb67ae85;

H[2] = 32'h3c6ef372;

H[3] = 32'ha54ff53a;

H[4] = 32'h510e527f;

H[5] = 32'h9b05688c;

H[6] = 32'h1f83d9ab;

H[7] = 32'h5be0cd19;

// Initialize SHA-256 constants

K[0] = 32'h428a2f98;

K[1] = 32'h71374491;

K[2] = 32'hb5c0fbcf;

K[3] = 32'he9b5dba5;

K[4] = 32'h3956c25b;

K[5] = 32'h59f111f1;

K[6] = 32'h923f82a4;

K[7] = 32'hab1c5ed5;

K[8] = 32'hd807aa98;

K[9] = 32'h12835b01;

K[10] = 32'h243185be;

K[11] = 32'h550c7dc3;

K[12] = 32'h72be5d74;

K[13] = 32'h80deb1fe;

K[14] = 32'h9bdc06a7;

K[15] = 32'hc19bf174;

K[16] = 32'he49b69c1;

K[17] = 32'hefbe4786;

K[18] = 32'h0fc19dc6;

K[19] = 32'h240ca1cc;

K[20] = 32'h2de92c6f;

K[21] = 32'h4a7484aa;

K[22] = 32'h5cb0a9dc;

K[23] = 32'h76f988da;

K[24] = 32'h983e5152;

K[25] = 32'ha831c66d;

K[26] = 32'hb00327c8;

K[27] = 32'hbf597fc7;

K[28] = 32'hc6e00bf3;

K[29] = 32'hd5a79147;

K[30] = 32'h06ca6351;

K[31] = 32'h14292967;

K[32] = 32'h27b70a85;

K[33] = 32'h2e1b2138;

K[34] = 32'h4d2c6dfc;

K[35] = 32'h53380d13;

K[36] = 32'h650a7354;

K[37] = 32'h766a0abb;

K[38] = 32'h81c2c92e;

K[39] = 32'h92722c85;

K[40] = 32'ha2bfe8a1;

K[41] = 32'ha81a664b;

K[42] = 32'hc24b8b70;

K[43] = 32'hc76c51a3;

K[44] = 32'hd192e819;

K[45] = 32'hd6990624;

K[46] = 32'hf40e3585;

K[47] = 32'h106aa070;

K[48] = 32'h19a4c116;

K[49] = 32'h1e376c08;

K[50] = 32'h2748774c;

K[51] = 32'h34b0bcb5;

K[52] = 32'h391c0cb3;

K[53] = 32'h4ed8aa4a;

K[54] = 32'h5b9cca4f;

K[55] = 32'h682e6ff3;

K[56] = 32'h748f82ee;

K[57] = 32'h78a5636f;

K[58] = 32'h84c87814;

K[59] = 32'h8cc70208;

K[60] = 32'h90befffa;

K[61] = 32'ha4506ceb;

K[62] = 32'hbef9a3f7;

K[63] = 32'hc67178f2;

end

// Initialize working variables at the start of processing

always @(posedge clk)

begin

if(index == 127)

begin

a <= H[0];

b <= H[1];

c <= H[2];

d <= H[3];

e <= H[4];

f <= H[5];

g <= H[6];

h <= H[7];

end

// Expand message schedule W[0] to W[63]

if(index < 16)

W[index] <= word;

else if(index < 64)

begin

W[index] <= ({W[index-2][16:0],W[index-2][31:17]} ^

{W[index-2][18:0],W[index-2][31:19]} ^

(W[index-2] >> 10)) +

W[index-7] +

({W[index-15][6:0],W[index-15][31:7]} ^

{W[index-15][17:0],W[index-15][31:18]} ^

(W[index-15] >> 3)) +

W[index-16];

end

index <= index + 1;

end

// Main SHA-256 compression function

always @(posedge clk)

begin

if(index > 0 && index < 65)

begin

sum1 = {e[5:0],e[31:6]}^{e[10:0],e[31:11]}^{e[24:0],e[31:25]};

sum0 = {a[1:0],a[31:2]}^{a[12:0],a[31:13]}^{a[21:0],a[31:22]};

Ch = (e & f) ^ ((~e) & g);

Maj = (a & b) ^ (b & c) ^ (c & a);

T1 = h + sum1 + Ch + K[index-1] + W[index-1];

T2 = sum0 + Maj;

h <= g;

g <= f;

f <= e;

e <= d + h + sum1 + Ch + K[index-1] + W[index-1];

d <= c;

c <= b;

b <= a;

a <= T1 + T2;

end

// Final hash value update after all rounds

if(index == 65)

begin

H[0] <= a + H[0];

H[1] <= b + H[1];

H[2] <= c + H[2];

H[3] <= d + H[3];

H[4] <= e + H[4];

H[5] <= f + H[5];

H[6] <= g + H[6];

H[7] <= h + H[7];

if(last\_word && (last\_next[0] || D))

output\_valid = 1'b1;

if(last\_next[1])

D <= 1'b1;

end

end

assign hash\_data = {H[0],H[1],H[2],H[3],H[4],H[5],H[6],H[7]};

endmodule

**Test Bench:**

module shatb();

reg [31:0]input\_data; // 32-bit input data for SHA-256

reg input\_valid; // Flag indicating valid input data

reg last\_word; // Flag indicating the last word in the input

reg [6:0]blocks; // Counter for the number of blocks processed

reg clk; // Clock signal

reg rst; // Reset signal

wire [255:0]hash\_data; // 256-bit hash output from SHA-256

wire output\_valid; // Flag indicating hash computation is complete

integer fd,scan\_file; // File descriptor and variable for reading input

// Initial block: Sets up initial conditions

initial begin

clk = 1'b0; // Initialize clock to 0

last\_word = 1'b0; // Set last\_word flag to 0 initially

input\_valid = 1'b1; // Mark input as valid

blocks = 0; // Initialize block counter

fd = $fopen("C:/Users/HP/Desktop/my\_file.txt","r"); // Open input file for reading

rst = 1'b1; // Activate reset

#2; // Wait for 2 time units

rst = 1'b0; // Release reset

end

// Instantiate the SHA-256 module

sha256 uut(

input\_data, // Input data to the SHA-256 module

input\_valid, // Input valid signal

input\_ready, // Input ready signal from SHA-256 module

last\_word, // Last word flag

clk, // Clock signal

rst, // Reset signal

output\_valid, // Output valid signal

hash\_data // 256-bit hash output

);

// Generate clock: Toggles based on output\_valid status

always #10 clk = (~output\_valid) ^ clk;

// Read input data from file when SHA-256 is ready for new data

always @(posedge clk) begin

if(input\_ready == 1'b1 && last\_word == 1'b0) begin

scan\_file = $fscanf(fd, "%x\n", input\_data); // Read next 32-bit word from file

blocks <= blocks + 1; // Increment block count

end

end

// Detect last word condition

always @(blocks) begin

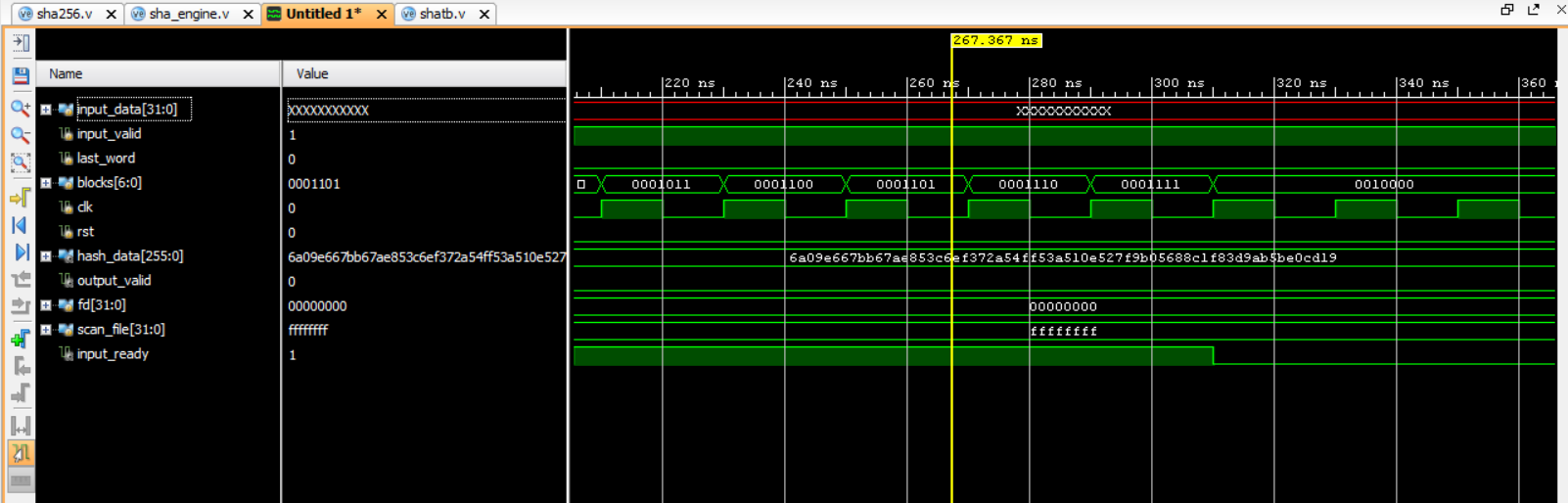
if(blocks == 24) // If 24 blocks have been read, set last\_word flag

last\_word = 1'b1;

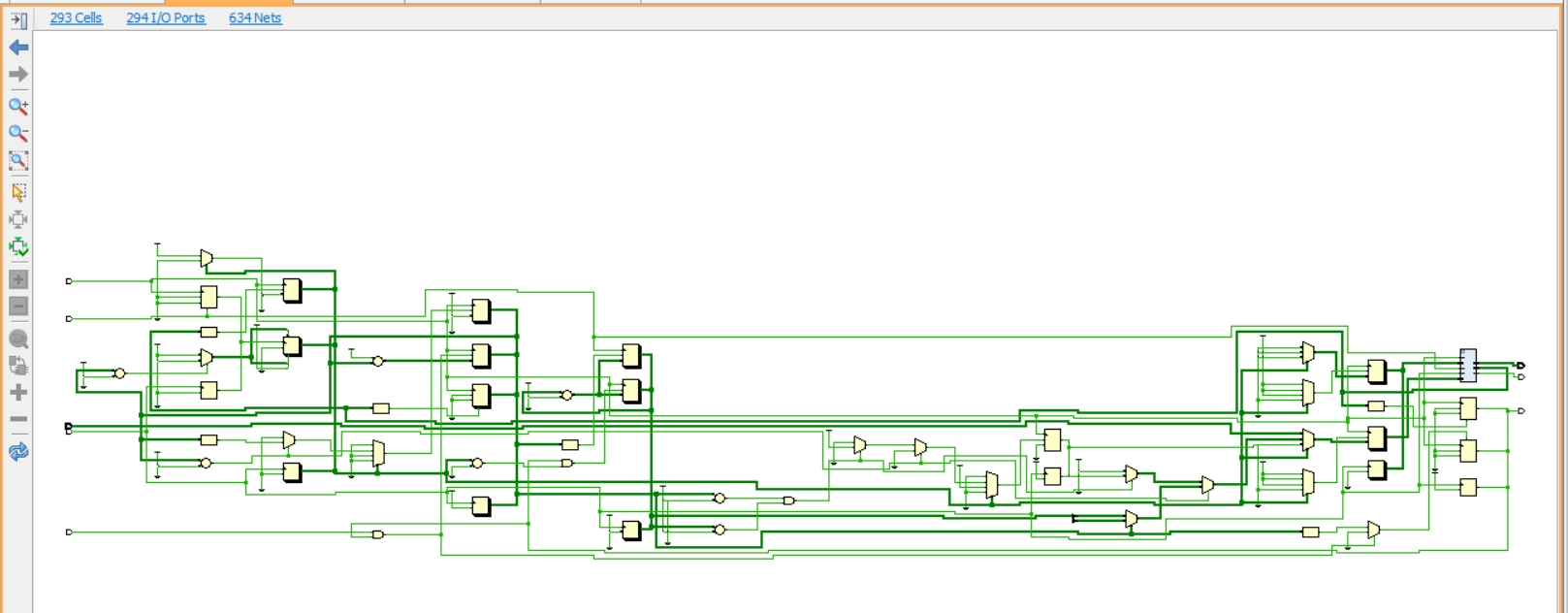
end

endmodule

**Simulation Output:**



* **Elaborated design:**



**Drive Link of simulation :**

https://drive.google.com/file/d/1k7T0JsQUvPz8c14puMLb1FKkEufNwhoa/view?usp=sharing

**Conclusion :**

This report presents a structured analysis and implementation of the SHA-256 hashing algorithm using Verilog. The SHA-256 algorithm provides high-security standards for cryptographic applications, ensuring data integrity, authenticity, and confidentiality. The Verilog implementation successfully simulates the hashing process using FPGA hardware in Xilinx Vivado, demonstrating its feasibility in real-time applications such as blockchain, secure communication, and digital signatures. Further optimization can be explored to enhance performance and resource utilization in FPGA implementations.

