DCD Mini Project

SECURE HASHING ALGORITHM (SHA-256) USING VERILOG HDL

Submitted for mini project of DCD laboratory as partial fulfillment of Btech course.

By Batch II

21ECB0F03 Aaesha Akram

21ECB0F04 Levin Joji Mathews

21ECB0F05 Tanay Bhale

Under the guidance of

Dr. Prithvi

Pothupogu

Dr. Vadthiya

Narendar

Department of Electronics and Communication Engineering,
November 2022

Index

1)-Abstract

a)-Motivation:

Data insecurity has been a prevalent issue in modern times where every important data is sent and received through various devices. Encryption of this data becomes very essential so it can be sent without the worry of unwanted users being able to read the data and provide a layer of security in many password driven technologies and services. So we aim to understand the Secure Encryption of data.

b)-Problem statement

Simulating a digital circuit, specially designed for Secure Hashing Algorithm in Xilinx Vivado suite.

c)-Methodology:

The project will be implemented using Verilog as the programming language. The main process of secure hashing is first by converting the data to binary and padding it to a bit value that's a multiple of 512, Secondly it initializes the hash values and round constants. It then chunks the loop and creates a message schedule for running the compression loop to modify the final values. Finally it concatenates the values together. The flow diagram at the end will show the process.

d)-Outcome:

We will see the output waveform of the designed SHA-256 hash algorithm generated through the software.

2)-Theory

a)-Introduction to Secure Hashing Algorithm:

- **What is SHA?**

SHA stands for secure hashing algorithm. SHA is a modified version of MD5 (The MD5 message-digest algorithm is a cryptographically broken but still widely used hash function producing a 128-bit hash value) and used for hashing data and certificates. A hashing algorithm shortens the input data into a smaller form that cannot be understood by using bitwise operations, modular additions, and compression functions.

- **Can SHA's be decrypted similar to encryptions?**

Hashing is similar to encryption, the only difference between hashing and encryption is that hashing is one-way, meaning once the data is hashed, the resulting hash digest cannot be cracked, unless a brute force attack is used.

- **How secure is SHA?**

SHA works in such a way even if a single character of the message changed, then it will generate a different hash. For example, hashing of two similar, but different messages i.e., Heaven and heaven is different. However, there is only a difference between a capital letter and a small letter. This is referred to as the avalanche effect. This effect is important in cryptography, as it means even the slightest change in the input message completely changes the output.

b) Introduction to Verilog HDL:

- **What is Verilog HDL**

Verilog is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits. In 2009, the Verilog standard (IEEE 1364-2005) was merged into the System Verilog standard, creating IEEE Standard 1800-2009. Since then, Verilog is officially part of the System Verilog language. The current version is IEEE standard 1800-2017.

- **How does Verilog help us implement secure hashing algorithms?**

The approach is to communicate ASCII characters to the FPGA using UART serial communication. The board will then encrypt the characters using the SHA-256 encryption algorithm. The hash output value will then be displayed on a computer display via VGA communication.

3) Code

```
module sha2_round #(
    parameter WORDSIZE=0
) (
    input [WORDSIZE-1:0] Kj, Wj,
    input [WORDSIZE-1:0] a_in, b_in, c_in, d_in, e_in, f_in, g_in, h_in,
    input [WORDSIZE-1:0] Ch_e_f_g, Maj_a_b_c, S0_a, S1_e,
    output [WORDSIZE-1:0] a_out, b_out, c_out, d_out, e_out, f_out, g_out, h_out
    );
wire [WORDSIZE-1:0] T1 = h_in + S1_e + Ch_e_f_g + Kj + Wj;
wire [WORDSIZE-1:0] T2 = S0_a + Maj_a_b_c;
assign a_out = T1 + T2;
assign b_out = a_in;
assign c_out = b_in;
assign d_out = c_in;
assign e_out = d_in + T1;
assign f_out = e_in;
assign g_out = f_in;
assign h_out = g_in;
endmodule
// Ch(x,y,z)
```

```
module Ch #(parameter WORDSIZE=0) (
    input wire [WORDSIZE-1:0] x, y, z,
    output wire [WORDSIZE-1:0] Ch
    );
assign Ch = ((x & y) ^ (~x & z));
endmodule
// Maj(x,y,z)
module Maj #(parameter WORDSIZE=0) (
    input wire [WORDSIZE-1:0] x, y, z,
    output wire [WORDSIZE-1:0] Maj
    );
assign Maj = (x & y) ^ (x & z) ^ (y & z);
endmodule
// the message schedule: a machine that generates Wt values
module W_machine #(parameter WORDSIZE=1) (
    input clk,
    input [WORDSIZE*16-1:0] M,
    input M_valid,

    output [WORDSIZE-1:0] W_tm2, W_tm15,
    input [WORDSIZE-1:0] s1_Wtm2, s0_Wtm15,
```

```
    output [WORDSIZE-1:0] W
    );

// W(t-n) values, from the perspective of Wt_next
assign W_tm2 = W_stack_q[WORDSIZE*2-1:WORDSIZE*1];
assign W_tm15 = W_stack_q[WORDSIZE*15-1:WORDSIZE*14];
wire [WORDSIZE-1:0] W_tm7 = W_stack_q[WORDSIZE*7-1:WORDSIZE*6];
wire [WORDSIZE-1:0] W_tm16 = W_stack_q[WORDSIZE*16-1:WORDSIZE*15];
// Wt_next is the next Wt to be pushed to the queue, will be consumed in 16 rounds
wire [WORDSIZE-1:0] Wt_next = s1_Wtm2 + W_tm7 + s0_Wtm15 + W_tm16;
reg [WORDSIZE*16-1:0] W_stack_q;
wire [WORDSIZE*16-1:0] W_stack_d = {W_stack_q[WORDSIZE*15-1:0], Wt_next};
assign W = W_stack_q[WORDSIZE*16-1:WORDSIZE*15];

always @(posedge clk)
begin
    if (M_valid) begin
        W_stack_q <= M;
    end else begin
        W_stack_q <= W_stack_d;
    end
```

```
end
endmodule


// block processor
// NB: master must continue to assert H_in until we have signaled output_valid
module sha256_block (
    input clk, rst,
    input [255:0] H_in,
    input [511:0] M_in,
    input input_valid,
    output [255:0] H_out,
    output output_valid
    );


reg [6:0] round;
wire [31:0] a_in = H_in[255:224], b_in = H_in[223:192], c_in = H_in[191:160], d_in =
H_in[159:128];
wire [31:0] e_in = H_in[127:96], f_in = H_in[95:64], g_in = H_in[63:32], h_in =
H_in[31:0];
reg [31:0] a_q, b_q, c_q, d_q, e_q, f_q, g_q, h_q;
wire [31:0] a_d, b_d, c_d, d_d, e_d, f_d, g_d, h_d;
```

```
wire [31:0] W_tm2, W_tm15, s1_Wtm2, s0_Wtm15, Wj, Kj;

assign H_out = {

   a_in + a_q, b_in + b_q, c_in + c_q, d_in + d_q, e_in + e_q, f_in + f_q, g_in + g_q, h_in

+ h_q

};

assign output_valid = round == 64;

always @(posedge clk)

begin

   if (input_valid) begin

      a_q <= a_in; b_q <= b_in; c_q <= c_in; d_q <= d_in;

      e_q <= e_in; f_q <= f_in; g_q <= g_in; h_q <= h_in;

      round <= 0;



end else begin

      a_q <= a_d; b_q <= b_d; c_q <= c_d; d_q <= d_d;

      e_q <= e_d; f_q <= f_d; g_q <= g_d; h_q <= h_d;

      round <= round + 1;

   end

end
```

```
sha256_round sha256_round (
    .Kj(Kj), .Wj(Wj),
    .a_in(a_q), .b_in(b_q), .c_in(c_q), .d_in(d_q),
    .e_in(e_q), .f_in(f_q), .g_in(g_q), .h_in(h_q),
    .a_out(a_d), .b_out(b_d), .c_out(c_d), .d_out(d_d),
    .e_out(e_d), .f_out(f_d), .g_out(g_d), .h_out(h_d)
);
sha256_s0 sha256_s0 (.x(W_tm15), .s0(s0_Wtm15));
sha256_s1 sha256_s1 (.x(W_tm2), .s1(s1_Wtm2));
W_machine #(.WORDSIZE(32)) W_machine (
    .clk(clk),
    .M(M_in), .M_valid(input_valid),
    .W_tm2(W_tm2), .W_tm15(W_tm15),

  .s1_Wtm2(s1_Wtm2), .s0_Wtm15(s0_Wtm15),
    .W(Wj)
);
sha256_K_machine sha256_K_machine (
    .clk(clk), .rst(input_valid), .K(Kj)
);
endmodule
```

```
// round compression function
module sha256_round (
    input [31:0] Kj, Wj,
    input [31:0] a_in, b_in, c_in, d_in, e_in, f_in, g_in, h_in,
    output [31:0] a_out, b_out, c_out, d_out, e_out, f_out, g_out, h_out
    );
wire [31:0] Ch_e_f_g, Maj_a_b_c, S0_a, S1_e;
Ch #(.WORDSIZE(32)) Ch (
    .x(e_in), .y(f_in), .z(g_in), .Ch(Ch_e_f_g)
);
Maj #(.WORDSIZE(32)) Maj (
    .x(a_in), .y(b_in), .z(c_in), .Maj(Maj_a_b_c)
);

sha256_S0 S0 (
    .x(a_in), .S0(S0_a)
);
sha256_S1 S1 (
    .x(e_in), .S1(S1_e)
);
sha2_round #(.WORDSIZE(32)) sha256_round_inner (
```

```
    .Kj(Kj), .Wj(Wj),
    .a_in(a_in), .b_in(b_in), .c_in(c_in), .d_in(d_in),
    .e_in(e_in), .f_in(f_in), .g_in(g_in), .h_in(h_in),
    .Ch_e_f_g(Ch_e_f_g), .Maj_a_b_c(Maj_a_b_c), .S0_a(S0_a), .S1_e(S1_e),
    .a_out(a_out), .b_out(b_out), .c_out(c_out), .d_out(d_out),
    .e_out(e_out), .f_out(f_out), .g_out(g_out), .h_out(h_out)
);
endmodule
// Σ0(x)
module sha256_S0 (
    input wire [31:0] x,
    output wire [31:0] S0
    );
assign S0 = ({x[1:0], x[31:2]} ^ {x[12:0], x[31:13]} ^ {x[21:0], x[31:22]});
endmodule


// Σ1(x)
module sha256_S1 (
    input wire [31:0] x,
    output wire [31:0] S1
    );
```

```
assign S1 = ({x[5:0], x[31:6]} ^ {x[10:0], x[31:11]} ^ {x[24:0], x[31:25]});
endmodule


// σ0(x)
module sha256_s0 (
    input wire [31:0] x,
    output wire [31:0] s0
    );
assign s0 = ({x[6:0], x[31:7]} ^ {x[17:0], x[31:18]} ^ (x >> 3));
endmodule
// σ1(x)
module sha256_s1 (
    input wire [31:0] x,
    output wire [31:0] s1
    );
assign s1 = ({x[16:0], x[31:17]} ^ {x[18:0], x[31:19]} ^ (x >> 10));
endmodule
// a machine that delivers round constants
module sha256_K_machine (
    input clk,
    input rst,
```

```
    output [31:0] K
    );
reg [2047:0] rom_q;
wire [2047:0] rom_d = { rom_q[2015:0], rom_q[2047:2016] };
assign K = rom_q[2047:2016];


always @(posedge clk)
begin



if (rst) begin
        rom_q <= {
            32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5,
            32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
            32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3,
            32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
            32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc,
            32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
            32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7,
            32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
            32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13,
```

```
        32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
        32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3,
        32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
        32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5,
        32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
        32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208,
        32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2
    };



end else begin
    rom_q <= rom_d;
  end
end
endmodule


// initial hash values
module sha256_H_0(
  output [255:0] H_0
  );
assign H_0 = {
```

```
    32'h6A09E667, 32'hBB67AE85, 32'h3C6EF372, 32'hA54FF53A,
    32'h510E527F, 32'h9B05688C, 32'h1F83D9AB, 32'h5BE0CD19
};
endmodule

// block processor
// NB: master must continue to assert H_in until we have signaled output_valid
module sha512_block (
    input clk, rst,

    input [511:0] H_in,
    input [1023:0] M_in,
    input input_valid,
    output [511:0] H_out,
    output output_valid
    );

reg [6:0] round;
wire [63:0] a_in = H_in[511:448], b_in = H_in[447:384], c_in = H_in[383:320], d_in =
H_in[319:256];
wire [63:0] e_in = H_in[255:192], f_in = H_in[191:128], g_in = H_in[127:64], h_in =
```

```
H_in[63:0];
reg [63:0] a_q, b_q, c_q, d_q, e_q, f_q, g_q, h_q;
wire [63:0] a_d, b_d, c_d, d_d, e_d, f_d, g_d, h_d;
wire [63:0] W_tm2, W_tm15, s1_Wtm2, s0_Wtm15, Wj, Kj;
assign H_out = {
   a_in + a_q, b_in + b_q, c_in + c_q, d_in + d_q, e_in + e_q, f_in + f_q, g_in + g_q, h_in
+ h_q
};
assign output_valid = round == 80;


always @(posedge clk)
begin
   if (input_valid) begin
      a_q <= a_in; b_q <= b_in; c_q <= c_in; d_q <= d_in;
      e_q <= e_in; f_q <= f_in; g_q <= g_in; h_q <= h_in;
      round <= 0;
   end else begin
      a_q <= a_d; b_q <= b_d; c_q <= c_d; d_q <= d_d;
      e_q <= e_d; f_q <= f_d; g_q <= g_d; h_q <= h_d;
      round <= round + 1;
   end
```

```
end


sha512_round sha512_round (
    .Kj(Kj), .Wj(Wj),
    .a_in(a_q), .b_in(b_q), .c_in(c_q), .d_in(d_q),
    .e_in(e_q), .f_in(f_q), .g_in(g_q), .h_in(h_q),
    .a_out(a_d), .b_out(b_d), .c_out(c_d), .d_out(d_d),
    .e_out(e_d), .f_out(f_d), .g_out(g_d), .h_out(h_d)
);


sha512_s0 sha512_s0 (.x(W_tm15), .s0(s0_Wtm15));
sha512_s1 sha512_s1 (.x(W_tm2), .s1(s1_Wtm2));
W_machine #(.WORDSIZE(64)) W_machine (
    .clk(clk),
    .M(M_in), .M_valid(input_valid),
    .W_tm2(W_tm2), .W_tm15(W_tm15),
    .s1_Wtm2(s1_Wtm2), .s0_Wtm15(s0_Wtm15),
    .W(Wj)
);
sha512_K_machine sha512_K_machine (
    .clk(clk), .rst(input_valid), .K(Kj)
```

```
);
endmodule


// round compression function
module sha512_round (
    input [63:0] Kj, Wj,
    input [63:0] a_in, b_in, c_in, d_in, e_in, f_in, g_in, h_in,
    output [63:0] a_out, b_out, c_out, d_out, e_out, f_out, g_out, h_out
    );


wire [63:0] Ch_e_f_g, Maj_a_b_c, S0_a, S1_e;
Ch #(.WORDSIZE(64)) Ch (
    .x(e_in), .y(f_in), .z(g_in), .Ch(Ch_e_f_g)
);
Maj #(.WORDSIZE(64)) Maj (
    .x(a_in), .y(b_in), .z(c_in), .Maj(Maj_a_b_c)
);
sha512_S0 S0 (
    .x(a_in), .S0(S0_a)
);
sha512_S1 S1 (
```

```
    .x(e_in), .S1(S1_e)
);
sha2_round #(.WORDSIZE(64)) sha256_round_inner (
    .Kj(Kj), .Wj(Wj),
    .a_in(a_in), .b_in(b_in), .c_in(c_in), .d_in(d_in),
    .e_in(e_in), .f_in(f_in), .g_in(g_in), .h_in(h_in),
    .Ch_e_f_g(Ch_e_f_g), .Maj_a_b_c(Maj_a_b_c), .S0_a(S0_a), .S1_e(S1_e),
    .a_out(a_out), .b_out(b_out), .c_out(c_out), .d_out(d_out),
    .e_out(e_out), .f_out(f_out), .g_out(g_out), .h_out(h_out)

);
endmodule

// Σ0(x)
module sha512_S0 (
    input wire [63:0] x,
    output wire [63:0] S0
    );
assign S0 = ({x[27:0], x[63:28]} ^ {x[33:0], x[63:34]} ^ {x[38:0], x[63:39]});
endmodule
```

```
// Σ1(x)
module sha512_S1 (
    input wire [63:0] x,
    output wire [63:0] S1
    );
assign S1 = ({x[13:0], x[63:14]} ^ {x[17:0], x[63:18]} ^ {x[40:0], x[63:41]});
endmodule
// σ0(x)
module sha512_s0 (

    input wire [63:0] x,
    output wire [63:0] s0
    );
assign s0 = ({x[0:0], x[63:1]} ^ {x[7:0], x[63:8]} ^ (x >> 7));
endmodule

// σ1(x)
module sha512_s1 (
    input wire [63:0] x,
    output wire [63:0] s1
    );
```

```
assign s1 = ({x[18:0], x[63:19]} ^ {x[60:0], x[63:61]} ^ (x >> 6));
endmodule


// a machine that delivers round constants
module sha512_K_machine (
    input clk,
    input rst,
    output [63:0] K
    );


reg [5119:0] rom_q;
wire [5119:0] rom_d = { rom_q[5055:0], rom_q[5119:5056] };
assign K = rom_q[5119:5056];
always @(posedge clk)
begin
    if (rst) begin
        rom_q <= {
            64'h428a2f98d728ae22,       64'h7137449123ef65cd,       64'hb5c0fbcfec4d3b2f,
64'he9b5dba58189dbbc,
            64'h3956c25bf348b538,       64'h59f111f1b605d019,       64'h923f82a4af194f9b,
64'hab1c5ed5da6d8118,
```

64'hd807aa98a3030242, 64'h12835b0145706fbe, 64'h243185be4ee4b28c, 64'h550c7dc3d5ffb4e2,

64'h72be5d74f27b896f, 64'h80deb1fe3b1696b1, 64'h9bdc06a725c71235, 64'hc19bf174cf692694,

64'he49b69c19ef14ad2, 64'hefbe4786384f25e3, 64'h0fc19dc68b8cd5b5, 64'h240ca1cc77ac9c65,

64'h2de92c6f592b0275, 64'h4a7484aa6ea6e483, 64'h5cb0a9dcbd41fbd4, 64'h76f988da831153b5,

64'h983e5152ee66dfab, 64'ha831c66d2db43210, 64'hb00327c898fb213f, 64'hbf597fc7beef0ee4,

64'hc6e00bf33da88fc2, 64'hd5a79147930aa725, 64'h06ca6351e003826f, 64'h142929670a0e6e70,

64'h27b70a8546d22ffc, 64'h2e1b21385c26c926, 64'h4d2c6dfc5ac42aed, 64'h53380d139d95b3df,

64'h650a73548baf63de, 64'h766a0abb3c77b2a8, 64'h81c2c92e47edaee6, 64'h92722c851482353b,

64'ha2bfe8a14cf10364, 64'ha81a664bbc423001, 64'hc24b8b70d0f89791, 64'hc76c51a30654be30,

64'hd192e819d6ef5218, 64'hd69906245565a910, 64'hf40e35855771202a, 64'h106aa07032bbd1b8,

64'h19a4c116b8d2d0c8, 64'h1e376c085141ab53, 64'h2748774cdf8eeb99,

```
64'h34b0bcb5e19b48a8,
        64'h391c0cb3c5c95a63,    64'h4ed8aa4ae3418acb,    64'h5b9cca4f7763e373,
64'h682e6ff3d6b2b8a3,
        64'h748f82ee5defb2fc,    64'h78a5636f43172f60,    64'h84c87814a1f0ab72,
64'h8cc702081a6439ec,
        64'h90befffa23631e28,    64'ha4506cebde82bde9,    64'hbef9a3f7b2c67915,
64'hc67178f2e372532b,
        64'hca273eceea26619c,    64'hd186b8c721c0c207,    64'heada7dd6cde0eb1e,
64'hf57d4f7fee6ed178,
        64'h06f067aa72176fba,    64'h0a637dc5a2c898a6,    64'h113f9804bef90dae,
64'h1b710b35131c471b,
        64'h28db77f523047d84,    64'h32caab7b40c72493,    64'h3c9ebe0a15c9bebc,
64'h431d67c49c100d4c,
        64'h4cc5d4becb3e42b6,    64'h597f299cfc657e2a,    64'h5fcb6fab3ad6faec,
64'h6c44198c4a475817
    };
  end else begin
    rom_q <= rom_d;
  end
end
```

```
endmodule


// initial hash values
module sha512_H_0(
    output [511:0] H_0
    );
assign H_0 = {
    64'h6A09E667F3BCC908,   64'hBB67AE8584CAA73B,   64'h3C6EF372FE94F82B,
64'hA54FF53A5F1D36F1,
    64'h510E527FADE682D1,   64'h9B05688C2B3E6C1F,   64'h1F83D9ABFB41BD6B,
64'h5BE0CD19137E2179
};
endmodule
```

- **TestBench**

```
module testbed;
reg input_valid = 0;
wire output_valid_256;
wire [255:0] H_0_256, H_out_256;
sha256_H_0 sha256_H_0 (.H_0(H_0_256));
sha256_block sha256_block (
    .clk(clk), .rst(rst),
```

```
    .H_in(H_0_256), .M_in(M_sha256_abc),
    .input_valid(input_valid),
    .H_out(H_out_256),
    .output_valid(output_valid_256)
);

wire output_valid_512;
wire [511:0] H_0_512, H_out_512;
sha512_H_0 sha512_H_0 (.H_0(H_0_512));
sha512_block sha512_block (
    .clk(clk), .rst(rst),
    .H_in(H_0_512), .M_in(M_sha512_abc),
    .input_valid(input_valid),
    .H_out(H_out_512),
    .output_valid(output_valid_512)
);

// the "abc" test vector, padded
wire [511:0] M_sha256_abc = {
256'h6162638000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000018
```

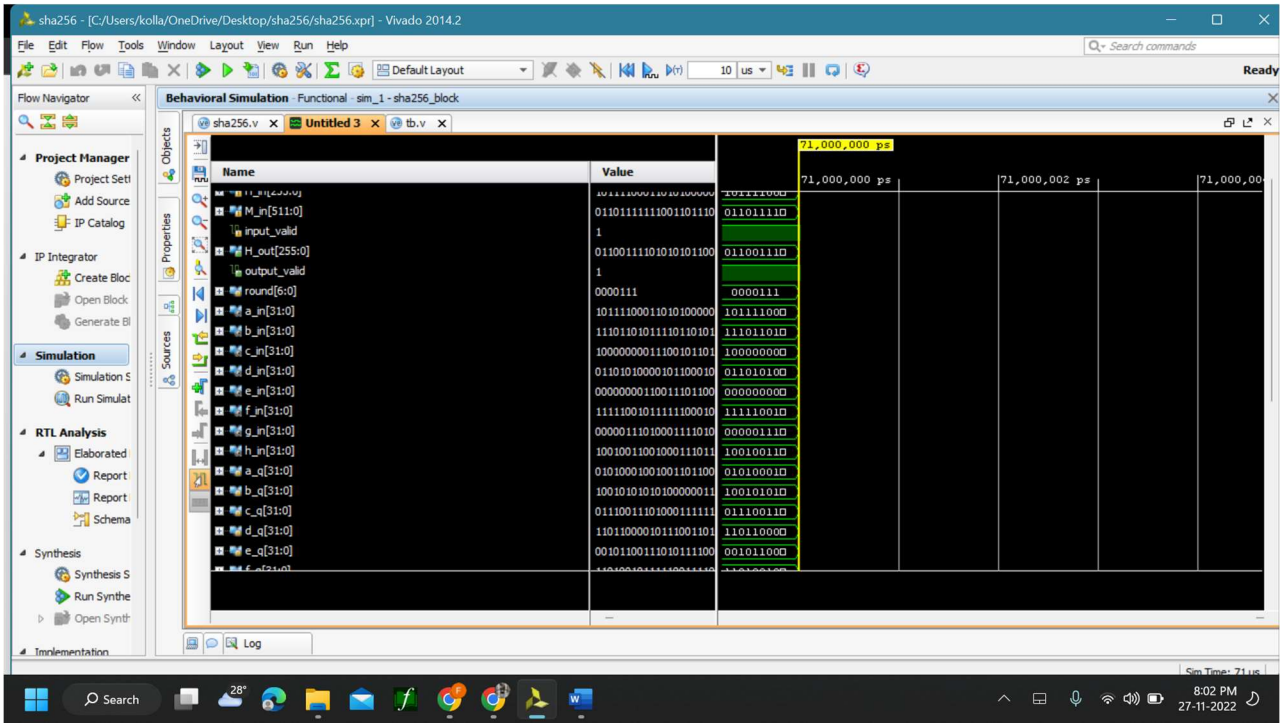
```
};
wire [1023:0] M_sha512_abc = {
256'h6162638000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000018
};
// the "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq" test vector, padded
wire [1023:0] M_sha256_2block = {
256'h6162636462636465636465666465666765666768666768696768696A68696A6B,
256'h696A6B6C6A6B6C6D6B6C6D6E6C6D6E6F6D6E6F706E6F7071800000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000,
256'h00000000000000000000000000000000000000000000000000000000000001C0
};
wire [511:0] M_sha256_2block_a = M_sha256_2block[1023:512];
wire [511:0] M_sha256_2block_b = M_sha256_2block[511:0];
// a null message
wire [511:0] M_sha256_null = {
256'h8000000000000000000000000000000000000000000000000000000000000000,
```

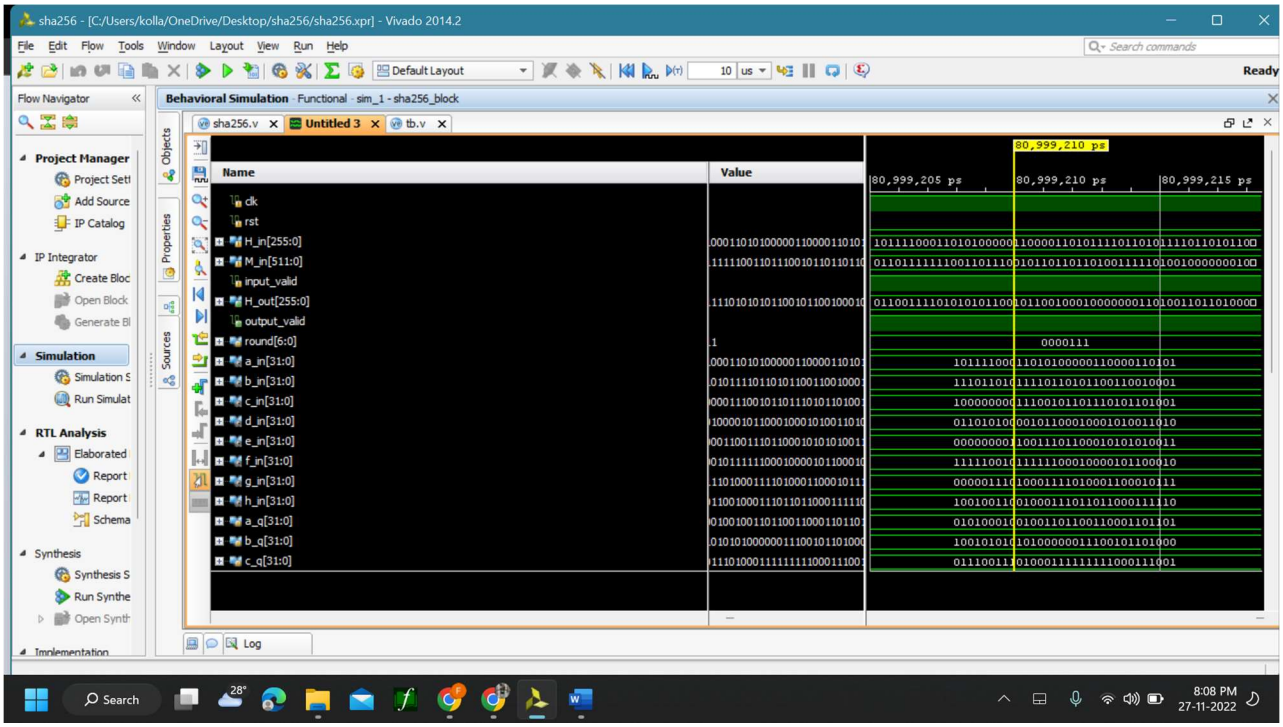
```
256'h0000000000000000000000000000000000000000000000000000000000000000
};
wire [1023:0] M_sha512_null = {
256'h8000000000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000,
256'h0000000000000000000000000000000000000000000000000000000000000000
};
// driver
reg [31:0] ticks = 0;
reg clk = 1'b0;
reg rst = 1'b0;
initial begin
  $display("starting");
  tick;
  input_valid = 1'b1;
  tick;
  input_valid = 1'b0;
  repeat (90) begin
    tick;
  end
```

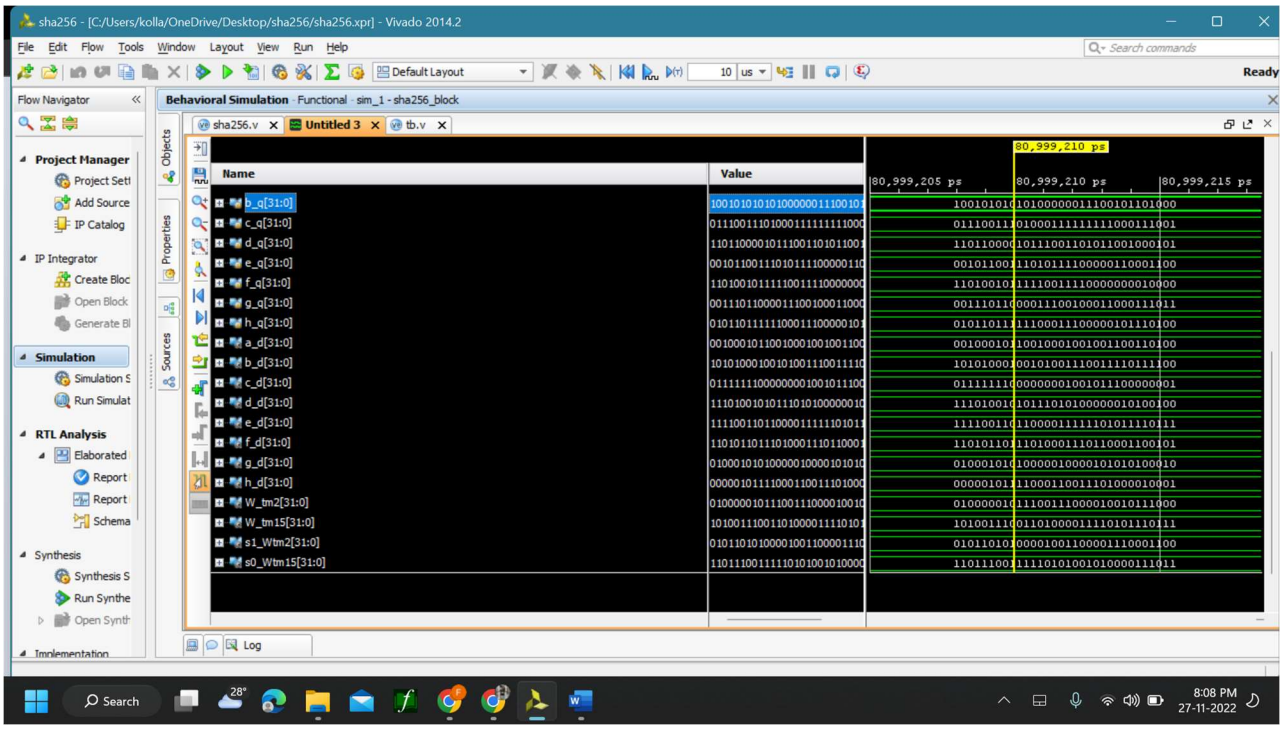
```
  $display("done");
  $finish;
end
task tick;
begin
  #1;
  ticks = ticks + 1;
  clk = 1;
  #1;
  clk = 0;
  dumpstate;
end
endtask
task dumpstate;
begin
  $display("%b %d %h", input_valid, ticks, ticks);
  $display("%b %h", output_valid_256, H_out_256);
  $display("%b %h", output_valid_512, H_out_512);
end
endtask
endmodule
```

4) Results

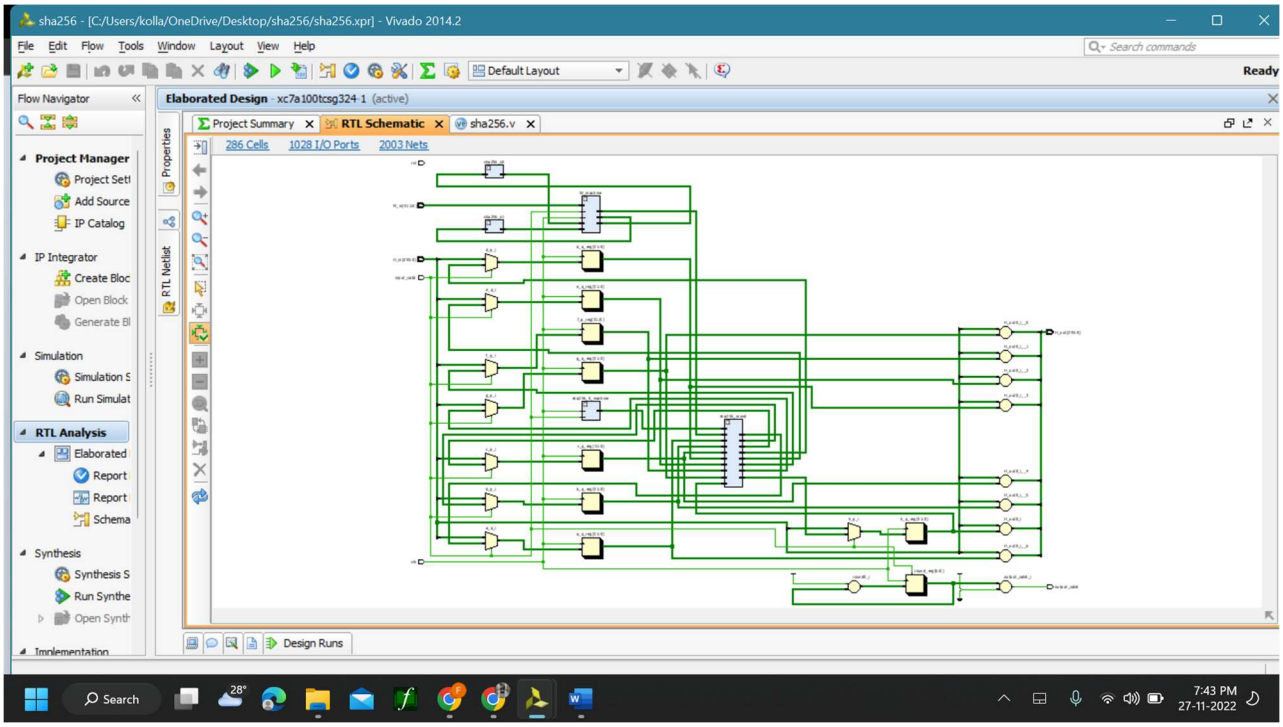
Simulations

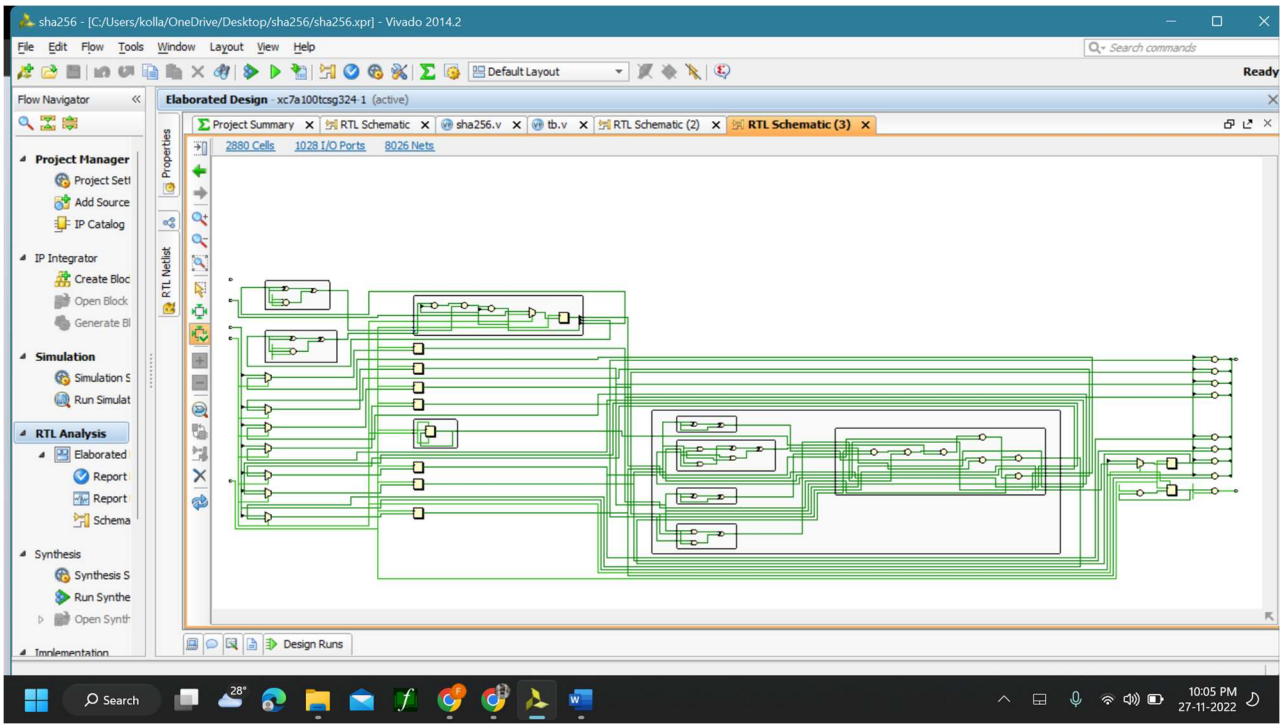






Block Diagram





5)- Conclusion

Our goal of design is to compress a long message to become a short and safe message abstract with an acceptable throughput. SHA is a famous message compress standard used in computer cryptography. Its improved version SHA-256 algorithm has been analyzed in this work, and implied by Verilog HDL (hardware description language).

6) Future Scope and applications:

In the age of hacking and cyber-crime, encryption is no longer a safe solution for massive web service providers who take sensitive information from users to perform tasks such as online payments, messaging, etc. Encryptions are no longer optimal as they have a decryption protocol that can be cracked. However, the Secure Hashing Algorithm eliminates that risk as it is one-way, meaning once the data is hashed, the resulting hash digest cannot be cracked, unless a brute force attack is used.

7) References

- https://en.wikipedia.org/wiki/Secure_Hash_Algorithms
- https://ijesc.org/upload/cad83e47fb4fc727a0147010762d6c2e.Implementation%20of%20Secure%20Hash%20Algorithm-256.pdf
- https://scholarworks.calstate.edu/concern/theses/dz010w13x?locale=en