Circuit for SHA-1

by Damian Cisneros

CPE 405

The SHA cryptographic hash functions have been used for Internet security for over 20 years. It generates a message or key based on the message it is given. That key is then used to secure passwords without saving the password itself. Another application of it is to use it to authenticate PDF copies by comparing the key attached to the PDF to the key the publisher generates with their message. It can be considered a one-way encryption that cannot be decoded. The SHA-1 published in 1995 is still used today even though it has been cracked by Google. Google has been pushing websites to use the newer SHA-2 and SHA-3 which are much more secure. I decided to do my project on the SHA-1 since I wanted to understand more about Internet security.

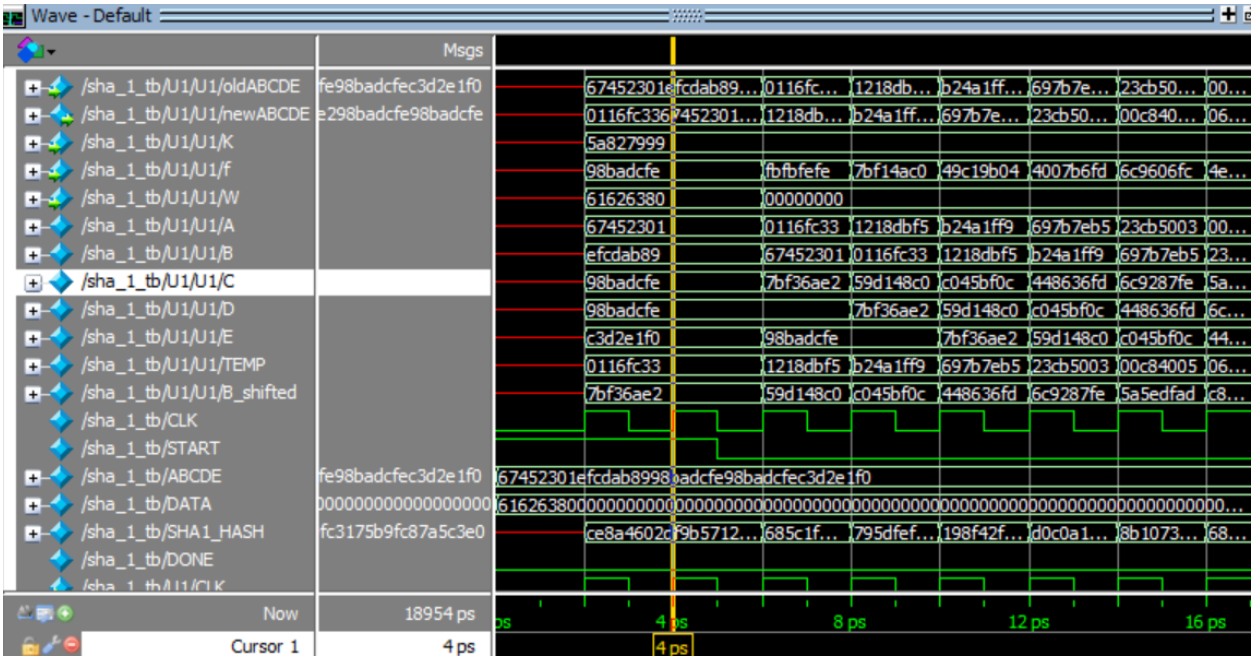
Before going into the algorithm, I will explain a bit of how the SHA-1 function works based on the paper published by the NSA on the SHA-1. The function takes in a message in string from 0 length to 2^64 bit length. It splits that message into 512-bit blocks, this process is called message padding. In the algorithm there is an initial state or 5-word buffer named H0, H1, H2, H3, and H4 each being 32-bit. This state is initialized based on constants they defined. There is also another 5-word buffer which is used inside the compression function, A, B, C, D, and E. It then runs its compression function, which repeats 80 times. Inside the function, it is taking in parts of the message and doing a series of logical operations such as AND, XOR, and ORs to randomize the message. At the end of each iteration, it is adding the result to the state buffer of H0-H4. When it is done with iterations, the result in the state buffer H0-H4 is the 160-bit unique message or also known as key that was generated. If at this point there is another block of data, it will repeat the same steps except this time it will use the state buffer that the previous block generated as the initial state rather than the constants they defined.

A much simpler way to explain this process is to visualize it as a washing machine. The hash function would be the inside of the machine and your message would be clothing. If you were to add clothing one at a time in a certain order, then pausing and adding more clothing as the wash continues, at the end you will have mixed order of clothing.

The approach I wanted to take was what goes on inside the machine where the message is mixed and for one 512-bit block. The message padding is what I did by hand and the data I put in my testbench to send to the machine or compression function. I had two modules for my SHA-1 algorithm. One(sha_1.v)was for where my initializing values, constants, and where I did the logic operations. The other module(sha1_logic) for generating the new A-E values after each iteration with the data received which had its logic for that iteration already done.

As for my results which I tested with the string “abc,” I would generate an incorrect hash of b2b5619fc6086elcafbce4a767e18f47ef590c43 when a9993e364706816aba3e25717850c26c9cd0d89d was expected. After simulation, I noticed that my C and D values were incorrect because they were the same(shown in the waveform below)

which is not supposed to be the case based on the algorithm. My error I believe is in grabbing the wrong bits in the buffer for C and D. I checked my code to make sure that I did not grab the same exact bits but did not make that mistake. It would take a longer look at the simulation results to find where exactly the issue happened. After this project, I believe I have learned a bit more of Internet security and what goes on behind the scenes. I would like to continue studying Internet security and taking courses for it.



My result:

```
# Hash is: b2b5619fc6086e1cafbce4a767e18f47ef590c43

VSIM 78>
```

Expected result:

sha1: a9993e364706816aba3e25717850c26c9cd0d89d

My code:

```
//Method 1 SHA-1

//block = 512-bit string
//only works with 512-bit

module sha_1(
   input CLK,
//  input ENABLE;
   input START,
   input wire [511:0]DATA,//data given to convert to SHA-1
   output wire[159:0]SHA1_HASH, //finished generation of SHA-1
   output wire DONE,
```

```
   input wire [159:0]ABCDE
);

//copy of DATA
reg [511:0]data_block;

reg [159:0]updatedABCDE;
wire [159:0]_afterSHA1logic; //ABCDE after SHA-1 logic on them


//initialize H values
wire [31:0]H0 = ABCDE[159:128]; //H0 = A
wire [31:0]H1 = ABCDE[127:96]; //H1 = B
wire [31:0]H2 = ABCDE[95:64]; //H2 = C
wire [31:0]H3 = ABCDE[63:32]; //H3 = D
wire [31:0]H4 = ABCDE[31:0]; //H4 = E

//set A-E values
wire [31:0]A = updatedABCDE[159:128];
wire [31:0]B = updatedABCDE[127:96];
wire [31:0]C = updatedABCDE[95:64];
wire [31:0]D = updatedABCDE[63:32];
wire [31:0]E = updatedABCDE[31:0];

//single word wire
reg [31:0]K;
wire [31:0]TEMP;
wire [31:0]f;

//wires for W (as shown in SHA-1 algorithm)
wire [31:0]Wt_3;
wire [31:0]Wt_8;
wire [31:0]Wt_14;
wire [31:0]Wt_16;
wire [31:0]Wt_before_shift;
wire [31:0]Wt;
wire [31:0]W;

//f constants (from SHA-1 algorithm)
wire [31:0]f_0;
wire [31:0]f_20;
wire [31:0]f_40;
wire [31:0]f_60;

//variables
reg [6:0]i; //need 80 so using 7bits; 2^6=64|(2^7=128)
reg finish;

initial begin
   finish = 0;
end

//get values for Wt -- W(t) = S^1(W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)).
assign Wt_3 = data_block[95:64]; //from (16-3=13*32=>512-416=96)
assign Wt_8 = data_block[255:224]; //from (16-8=8*32=>512-256=256)
assign Wt_14 = data_block[447:416]; //from (16-14=2*32=>512-64=448)
```

```
assign Wt_16 = data_block[511:480]; //from (16-16=0*32=>512-0=512)
assign Wt_before_shift = Wt_3 ^ Wt_8 ^ Wt_14 ^ Wt_16;
assign Wt = {Wt_before_shift[30:0], Wt_before_shift[31]}; //circular left shift 1
assign W = data_block[511:480];

//SHA-1 Logic Operations for f
assign f_0 = (B & C) | (~B & D);
assign f_20 = B ^ C ^ D;
assign f_40 = (B & C) | (B & D) | (C & D);
assign f_60 = B ^ C ^ D;

//Changing f depending on iteration
assign f = (i < 20) ? f_0 : (i < 40) ? f_20 : (i < 60) ? f_40 : f_60;

assign SHA1_HASH = {H0+A,H1+B,H2+C,H3+D,H4+E}; //update state based on changing A-E

assign DONE = finish;

//instantiate and get new A-E values
sha1_logic U1(.oldABCDE(updatedABCDE), .newABCDE(_afterSHA1logic), .K(K), .f(f), .W(W));

always @(posedge CLK) begin
   if (START) begin
      i <= 0; //start counting
      updatedABCDE <= ABCDE; //initialize with original A-E
      data_block <= DATA; //copy data
   end

   else begin
      data_block <= {data_block[479:0], Wt};
      updatedABCDE <= _afterSHA1logic; //update new A-E values
      i <= i + 1;

   end

end

always @(posedge CLK) begin
   if(i == 80) begin
      finish = 1;
   end
end

//K(t) sequence given by SHA-1
always @(*) begin
   if(i <= 19) begin
      K <= 32'h5A827999;
      //f <= (B & C) | (~B & D);
   end
   else if(i <= 39) begin
      K <= 32'h6ED9EBA1;
      //f <= B ^ C ^ D;
   end
   else if(i <= 59) begin
      K <= 32'h8F1BBCDC;
      //f <= (B & C) | (B & D) | (C & D);
```

```
    end
    else begin //(60 <= t <= 79)
        K <= 32'hCA62C1D6;
        //f <= B ^ C ^ D;
    end
end

endmodule


module sha1_logic(
    input wire [159:0] oldABCDE,
    output wire [159:0] newABCDE,
    input  [31:0]K,
    input wire[31:0]f,
    input wire[31:0]W

);
//grab A-E values passed in
wire [31:0]A = oldABCDE[159:128];
wire [31:0]B = oldABCDE[127:96];
wire [31:0]C = oldABCDE[95:64];
wire [31:0]D = oldABCDE[63:32];
wire [31:0]E = oldABCDE[31:0];

wire [31:0]TEMP = {A[26:0], A[31:27]} + f + E + W + K; //
wire [31:0]B_shifted = {B[1:0], B[31:2]}; //circular left shift B

//circular left shift A by 5 bits + f(t;B,C,D) + E + W[s] + K(t)
assign newABCDE = {TEMP,A,B_shifted,C,D};

endmodule


module sha_1_tb;

reg CLK;
reg START;

//initial values given by algorithm
wire [159:0]ABCDE = {32'h67452301,32'hEFCDAB89,32'h98BADCFE,32'h98BADCFE,32'hC3D2E1F0};

wire [511:0]DATA;
wire [159:0]SHA1_HASH;
wire DONE;

//giving string with padding to sha_1.v
assign DATA = {"abc", 8'h80, 416'd0, 64'd24};

//send data and receive hash
sha_1 U1(.CLK(CLK), .START(START), .DATA(DATA), .SHA1_HASH(SHA1_HASH), .DONE(DONE),
.ABCDE(ABCDE));

always begin
    #1
    CLK <= 0;
```

```
    #1
    CLK <= 1;
end
initial begin
    //$display("Hash starting for: %h", DATA);
    CLK = 0;
    START = 1;
    #5 START = 0;

    wait(DONE == 1); //wait until 80 iterations to print result

    $display ("Hash is: %h", SHA1_HASH);
end

endmodule
```