Overview of SNOW 3G [4]

We will get a brief overview of SNOW 3G - A New Stream Cipher that forms the heart of the 3GPP confidentiality algorithm UEA2 and the 3GPP integrity algorithm UIA2.

Introduction –

The two basic issues in mobile communications security are Data Confidentiality and Data Integrity. The term of Data Confidentiality is referred to keeping information secret from all but those who are authorized to see it while the term of Data Integrity is referred to ensuring information has not been altered by unauthorized or unknown means.

The current radio interface protection algorithms for Universal Mobile Telecommunication System (UMTS), UEA1 for confidentiality, and UIA1 for integrity of signaling messages - were designed by SAGE/ETSI Security Algorithms Group of Experts. No weakness has been discovered in these algorithms, and there is no indication that a weakness is likely to be found. However, if one ever were found, it would be much better to have a replacement. So the 3rd Generation Partnership Project (3GPP), together with the GSM Association, called SAGE wishes to specify a second set of algorithms, UEA2 (confidentiality algorithm) and UIA2 (integrity algorithm).Each of these algorithms is based on the SNOW 3G algorithm.

SNOW 3G is a word-oriented stream cipher that generates a sequence of 32-bit words under the control of a 128-bit Key and a 128-bit Initialisation Variable (IV).

Design –

SNOW 3G is a word-oriented stream cipher that generates a sequence of 32-bit words under the control of a 128-bit key and a 128-bit initialization variable. First a key initialization is performed and the cipher is clocked without producing output. Then the cipher operates in key-generation mode and it produces a 32-bit ciphertext / plaintext word output in every clock cycle.

Linear Feedback Shift Register (LFSR) –

The Linear Feedback Shift Register (LFSR) consists of sixteen registers s0, s1, s2, ..., s15 each holding 32 bits.

Finite State Machine (FSM) –

The Finite State Machine (FSM) has three 32-bit registers R1, R2 and R3. It also contains the 32x32-bit S-boxes S1 and S2 which are used to update the registers R2 and R3. And R1 is updated using XOR and Modulo Addition.

The 32x32-bit S-Box –

The S-Box S1 maps a 32-bit input to a 32-bit output. Let $w = w_0||w_1||w_2||w_3$ the 32-bit input with w_0 the most and w_3 the least significant byte. Let $S1(w) = r_0||r_1||r_2||r_3$ with r_0 the most and r_3 the least significant byte. The r_0, r_1, r_2, r_3 are defined using the function MULx and input words w_0, w_1, w_2, w_3.

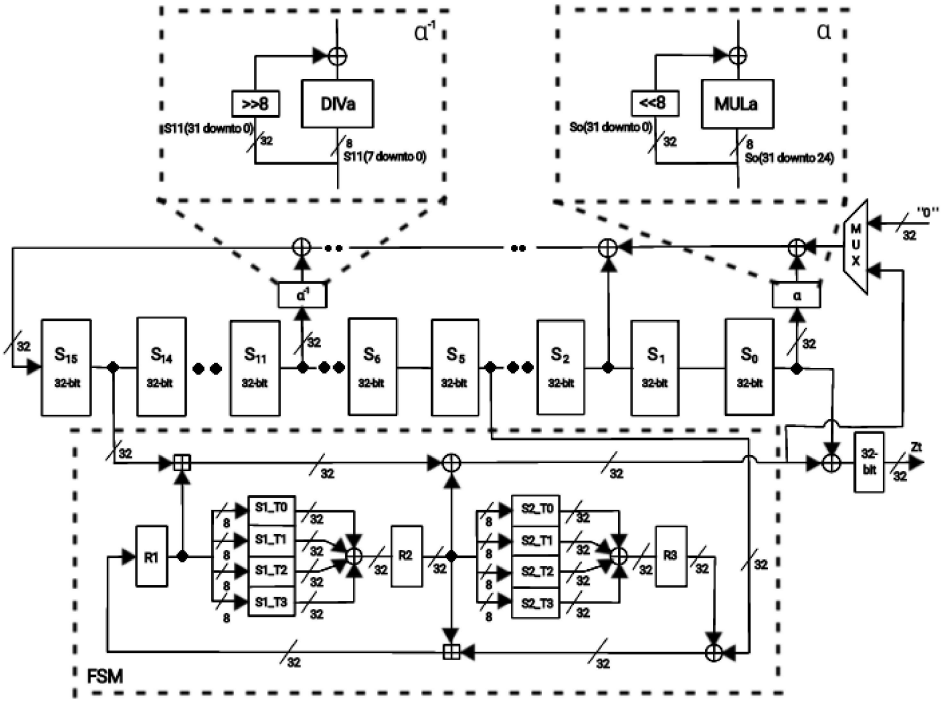


Figure 6: Hardware Architecture of SNOW 3G

Functions Used –

- **MULx :-** MULx maps 16 bits to 8 bits. Let V and c be 8-bit input values. Then MULx is defined :-

```
if(V & 8'h80)
        MULx = (V « 1) ĉ;
 else
        MULx = V « 1;
```

- **MULxPOW :-** MULxPOW maps 16 bits and an positive integer i to 8 bit. Let V and c be 8-bit input values, then MULxPOW(V, i, c) is recursively defined :-

```
if(i == 8'd0)
     MULxPOW = V;
 else
     MULxPOW = MULx(MULxPOW(V,c,(i-1)),c);
```

- MUL_α : – This function maps 8 bits to 32 bits. Let c be the 8-bit input, then

$$MUL_\alpha(c) = (MULxPOW(c, 23, 0xA9)||MULxPOW(c, 245, 0xA9)|| MULxPOW(c, 48, 0xA9)||MULxPOW(c, 239, 0xA9)).$$

- DIV_α : – This function maps 8 bits to 32 bits. Let c be the 8-bit input, then

$$DIV_\alpha(c) = (MULxPOW(c, 16, 0xA9)||MULxPOW(c, 39, 0xA9)|| MULxPOW(c, 6, 0xA9)||MULxPOW(c, 64, 0xA9)).$$

Operation –

- **Key Initialization :-** During key initialization process the LFSR and the internal FSM registers fetch their initial values. It is initialized with a 128-bit key consisting of four 32-bit words k_0, k_1, k_2, k_3 and an 128-bit initialisation variable consisting of four 32-bit words IV_0, IV_1, IV_2, IV_3 as follows. Let 1 be the all-ones word (0xffffffff). Additionally, the R1, R2

$s_{15} = \mathbf{k_3} \oplus \mathbf{IV_0}$	$s_{14} = \mathbf{k_2}$	$s_{13} = \mathbf{k_1}$	$s_{12} = \mathbf{k_0} \oplus \mathbf{IV_1}$
$s_{11} = \mathbf{k_3} \oplus \mathbf{1}$	$s_{10} = \mathbf{k_2} \oplus \mathbf{1} \oplus \mathbf{IV_2}$	$s_9 = \mathbf{k_1} \oplus \mathbf{1} \oplus \mathbf{IV_3}$	$s_8 = \mathbf{k_0} \oplus \mathbf{1}$
$s_7 = \mathbf{k_3}$	$s_6 = \mathbf{k_2}$	$s_5 = \mathbf{k_1}$	$s_4 = \mathbf{k_0}$
$s_3 = \mathbf{k_3} \oplus \mathbf{1}$	$s_2 = \mathbf{k_2} \oplus \mathbf{1}$	$s_1 = \mathbf{k_1} \oplus \mathbf{1}$	$s_0 = \mathbf{k_0} \oplus \mathbf{1}$

  and R3 FSM registers are set to zero. The clocking process takes 32 clock cycles in order the initialization process to be completed.

- **Keystream Generation :-** After the key-generation process the system is up to process the data in order to encrypt or decrypt. First, the cipher is clocked once and the output is discarded. Finally, the produced output sequence, called running key, is added bitwise to the plaintext sequence. The result is the ciphertext sequence. In decryption, the same operation is done. In every clock cycle the 32-bit ciphertext word $z_t = F \oplus s_0$ produced.

Conclusion –

The author proposed a new stream cipher called SNOW 3G. It forms the heart of the 3GPP confidentiality algorithm UEA2 and the 3GPP integrity algorithm UIA2, offering reliable security services in Universal Mobile Telecommunication System (UMTS).It is found to be much faster & secure than SNOW 2.0. It is also proven that the SNOW 3G is a very good solution not only for 3G mobile devices however also for applications with high speed demands. The hardware implementation of the SNOW 3G is also attached from next page.

Hardware Implementation of SNOW 3G

SNOW-3G (Top Module)

```
`timescale 1ns / 1ps

module SNOW_3G(
    input [127:0] s_key,
    input [127:0] IV,
    input clk,
    output [32:0] keystream
    );
    LFSR l1(.s_key(s_key), .IV(IV), .clk(clk), .keystream(keystream));
endmodule
```

Linear Feedback Shift Register (LFSR)

```
`timescale 1ns / 1ps

module LFSR(
    input [127:0] s_key,
    input [127:0] IV,
    input clk,
    output wire [31:0] keystream
    );

    reg [31:0] s0;
    reg [31:0] s1;
    reg [31:0] s2;
    reg [31:0] s3;
    reg [31:0] s4;
    reg [31:0] s5;
    reg [31:0] s6;
    reg [31:0] s7;
    reg [31:0] s8;
    reg [31:0] s9;
    reg [31:0] s10;
    reg [31:0] s11;
    reg [31:0] s12;
    reg [31:0] s13;
    reg [31:0] s14;
    reg [31:0] s15;

    wire [31:0] FSM_out;
    wire [31:0] alpha_out;
    wire [31:0] alpha_inv_out;

    always @ (posedge clk)
    begin
        s0 = (s_key[127:96] ^ IV[31:0]);
        s1 = s_key[95:64];
        s2 = s_key[63:32];
        s3 = (s_key[31:0] ^ IV[63:32]);
        s4 = (s_key[127:96] ^ 32'h11111111);
        s5 = (s_key[95:64] ^ 32'h11111111 ^ IV[95:64]);
        s6 = (s_key[63:32] ^ 32'h11111111 ^ IV[127:96]);
        s7 = (s_key[31:0] ^ 32'h11111111);
        s8 = s_key[127:96];
        s9 = s_key[95:64];
        s10 = s_key[63:32];
        s11 = s_key[31:0];
        s12 = (s_key[127:96] ^ 32'h11111111);
        s13 = (s_key[95:64] ^ 32'h11111111);
        s14 = (s_key[63:32] ^ 32'h11111111);
        s15 = (s_key[31:0] ^ 32'h11111111);
        #640;
        s0 <= (alpha_inv_out ^ (s13 ^ (alpha_out ^ FSM_out)));
```

```
        s1 <= s0;
        s2 <= s1;
        s3 <= s2;
        s4 <= s3;
        s5 <= s4;
        s6 <= s5;
        s7 <= s6;
        s8 <= s7;
        s9 <= s8;
        s10 <= s9;
        s11 <= s10;
        s12 <= s11;
        s13 <= s12;
        s14 <= s13;
        s15 <= s14;
    end

    assign keystream = (s15 ^ FSM_out);

    FSM f1(.in1(s0), .in2(s10), .clk(clk), .FSM_out(FSM_out));
    alpha_inv a1(.alpha_inv_in(s4), .alpha_inv_out(alpha_inv_out));
    alpha a2(.alpha_in(s15), .alpha_out(alpha_out));

endmodule
```

Alpha

```
`timescale 1ns / 1ps

module alpha(
    input [31:0] alpha_in,
    output [31:0] alpha_out
    );

    function [7 : 0] MULx(input [7 : 0] V, input [7 : 0] c);
    begin
        if(V & 8'h80)
            MULx = (V << 1) ^ c;
        else
            MULx = V << 1;
    end
    endfunction

    function [7 : 0] MULxPOW(input [7 : 0] V, input [7 : 0] c, input [7:0] i);
    begin
        if(i == 8'd0)
            MULxPOW = V;
        else
            MULxPOW = MULx(MULxPOW(V,c,(i-1)),c);
    end
    endfunction

    function [31 : 0] DIValpha(input [7 : 0] c);
    begin
        DIValpha[31:24] = (MULxPOW(8'd16,c,8'ha9) << 24);
        DIValpha[23:16] = (MULxPOW(8'd39,c,8'ha9) << 16);
        DIValpha[15:8] = (MULxPOW(8'd6,c,8'ha9) << 8);
        DIValpha[7:0] = MULxPOW(8'd64,c,8'ha9);
    end
    endfunction

    assign alpha_out = ((alpha_in >> 8) & 32'h00ffffff) ^ (DIValpha(alpha_in) & 8'hff);

endmodule
```

Alpha Inverse

```
`timescale 1ns / 1ps

module alpha_inv(
    input [31:0] alpha_inv_in,
    output [31:0] alpha_inv_out
    );

    function [7 : 0] MULx(input [7 : 0] V, input [7 : 0] c);
    begin
        if(V & 8'h80)
            MULx = (V << 1) ^ c;
        else
            MULx = V << 1;
    end
    endfunction

    function [7 : 0] MULxPOW(input [7 : 0] V, input [7 : 0] c, input [7:0] i);
    begin
        if(i == 8'd0)
            MULxPOW = V;
        else
            MULxPOW = MULx(MULxPOW(V,c,(i-1)),c);
    end
    endfunction

    function [31 : 0] MULalpha(input [7 : 0] c);
    begin
        MULalpha[31:24] = (MULxPOW(8'd23,c,8'ha9) << 24);
        MULalpha[23:16] = (MULxPOW(8'd245,c,8'ha9) << 16);
        MULalpha[15:8] = (MULxPOW(8'd48,c,8'ha9) << 8);
        MULalpha[7:0] = MULxPOW(8'd239,c,8'ha9);
    end
    endfunction

    assign alpha_inv_out = ((alpha_inv_in << 8) & 32'hffffff00) ^ (MULalpha(alpha_inv_in) & 8'hff);

endmodule
```

Finite-State Machine (FSM)

```
`timescale 1ns / 1ps

module FSM (
  input [31:0] in1,
  input [31:0] in2,
  input clk,
  output [31:0] FSM_out
);
  reg [31:0] R1 = 32'd0;
  reg [31:0] R2 = 32'd0;
  reg [31:0] R3 = 32'd0;
  wire [31:0] x0;
  wire [31:0] x1;
  wire [31:0] x2;
  wire [31:0] x3;

  always @ (posedge clk)
  begin
    R1 <= x1;
    R2 <= x2;
    R3 <= x3;
  end

  assign FSM_out = (x0 ^ R2);

  Modulo_Addition m1(.a(in1), .b(R1), .sum(x0));
```

```
  Modulo_Addition m2(.a(R3^in2), .b(R2), .sum(x1));
  S_Box s1(.in(R1), .out(x2));
  S_Box s2(.in(R2), .out(x3));
endmodule
```

S-Box

```
`timescale 1ns / 1ps

module S_Box(
    input [31:0] in,
    output [31:0] out
);
  reg [8:0] t0=8'd0;
  reg [8:0] t1=8'd0;
  reg [8:0] t2=8'd0;
  reg [8:0] t3=8'd0;

  function [7 : 0] sbox(input [7 : 0] z);
    begin
      case (z)
        8'h00: sbox = 8'h63;
        8'h01: sbox = 8'h7c;
        8'h02: sbox = 8'h77;
        8'h03: sbox = 8'h7b;
        8'h04: sbox = 8'hf2;
        8'h05: sbox = 8'h6b;
        8'h06: sbox = 8'h6f;
        8'h07: sbox = 8'hc5;
        8'h08: sbox = 8'h30;
        8'h09: sbox = 8'h01;
        8'h0a: sbox = 8'h67;
        8'h0b: sbox = 8'h2b;
        8'h0c: sbox = 8'hfe;
        8'h0d: sbox = 8'hd7;
        8'h0e: sbox = 8'hab;
        8'h0f: sbox = 8'h76;
        8'h10: sbox = 8'hca;
        8'h11: sbox = 8'h82;
        8'h12: sbox = 8'hc9;
        8'h13: sbox = 8'h7d;
        8'h14: sbox = 8'hfa;
        8'h15: sbox = 8'h59;
        8'h16: sbox = 8'h47;
        8'h17: sbox = 8'hf0;
        8'h18: sbox = 8'had;
        8'h19: sbox = 8'hd4;
        8'h1a: sbox = 8'ha2;
        8'h1b: sbox = 8'haf;
        8'h1c: sbox = 8'h9c;
        8'h1d: sbox = 8'ha4;
        8'h1e: sbox = 8'h72;
        8'h1f: sbox = 8'hc0;
        8'h20: sbox = 8'hb7;
        8'h21: sbox = 8'hfd;
        8'h22: sbox = 8'h93;
        8'h23: sbox = 8'h26;
        8'h24: sbox = 8'h36;
        8'h25: sbox = 8'h3f;
        8'h26: sbox = 8'hf7;
        8'h27: sbox = 8'hcc;
        8'h28: sbox = 8'h34;
        8'h29: sbox = 8'ha5;
        8'h2a: sbox = 8'he5;
        8'h2b: sbox = 8'hf1;
        8'h2c: sbox = 8'h71;
        8'h2d: sbox = 8'hd8;
        8'h2e: sbox = 8'h31;
```

```
62          8'h2f: sbox = 8'h15;
63          8'h30: sbox = 8'h04;
64          8'h31: sbox = 8'hc7;
65          8'h32: sbox = 8'h23;
66          8'h33: sbox = 8'hc3;
67          8'h34: sbox = 8'h18;
68          8'h35: sbox = 8'h96;
69          8'h36: sbox = 8'h05;
70          8'h37: sbox = 8'h9a;
71          8'h38: sbox = 8'h07;
72          8'h39: sbox = 8'h12;
73          8'h3a: sbox = 8'h80;
74          8'h3b: sbox = 8'he2;
75          8'h3c: sbox = 8'heb;
76          8'h3d: sbox = 8'h27;
77          8'h3e: sbox = 8'hb2;
78          8'h3f: sbox = 8'h75;
79          8'h40: sbox = 8'h09;
80          8'h41: sbox = 8'h83;
81          8'h42: sbox = 8'h2c;
82          8'h43: sbox = 8'h1a;
83          8'h44: sbox = 8'h1b;
84          8'h45: sbox = 8'h6e;
85          8'h46: sbox = 8'h5a;
86          8'h47: sbox = 8'ha0;
87          8'h48: sbox = 8'h52;
88          8'h49: sbox = 8'h3b;
89          8'h4a: sbox = 8'hd6;
90          8'h4b: sbox = 8'hb3;
91          8'h4c: sbox = 8'h29;
92          8'h4d: sbox = 8'he3;
93          8'h4e: sbox = 8'h2f;
94          8'h4f: sbox = 8'h84;
95          8'h50: sbox = 8'h53;
96          8'h51: sbox = 8'hd1;
97          8'h52: sbox = 8'h00;
98          8'h53: sbox = 8'hed;
99          8'h54: sbox = 8'h20;
100         8'h55: sbox = 8'hfc;
101         8'h56: sbox = 8'hb1;
102         8'h57: sbox = 8'h5b;
103         8'h58: sbox = 8'h6a;
104         8'h59: sbox = 8'hcb;
105         8'h5a: sbox = 8'hbe;
106         8'h5b: sbox = 8'h39;
107         8'h5c: sbox = 8'h4a;
108         8'h5d: sbox = 8'h4c;
109         8'h5e: sbox = 8'h58;
110         8'h5f: sbox = 8'hcf;
111         8'h60: sbox = 8'hd0;
112         8'h61: sbox = 8'hef;
113         8'h62: sbox = 8'haa;
114         8'h63: sbox = 8'hfb;
115         8'h64: sbox = 8'h43;
116         8'h65: sbox = 8'h4d;
117         8'h66: sbox = 8'h33;
118         8'h67: sbox = 8'h85;
119         8'h68: sbox = 8'h45;
120         8'h69: sbox = 8'hf9;
121         8'h6a: sbox = 8'h02;
122         8'h6b: sbox = 8'h7f;
123         8'h6c: sbox = 8'h50;
124         8'h6d: sbox = 8'h3c;
125         8'h6e: sbox = 8'h9f;
126         8'h6f: sbox = 8'ha8;
127         8'h70: sbox = 8'h51;
128         8'h71: sbox = 8'ha3;
129         8'h72: sbox = 8'h40;
130         8'h73: sbox = 8'h8f;
```

```
131         8'h74: sbox = 8'h92;
132         8'h75: sbox = 8'h9d;
133         8'h76: sbox = 8'h38;
134         8'h77: sbox = 8'hf5;
135         8'h78: sbox = 8'hbc;
136         8'h79: sbox = 8'hb6;
137         8'h7a: sbox = 8'hda;
138         8'h7b: sbox = 8'h21;
139         8'h7c: sbox = 8'h10;
140         8'h7d: sbox = 8'hff;
141         8'h7e: sbox = 8'hf3;
142         8'h7f: sbox = 8'hd2;
143         8'h80: sbox = 8'hcd;
144         8'h81: sbox = 8'h0c;
145         8'h82: sbox = 8'h13;
146         8'h83: sbox = 8'hec;
147         8'h84: sbox = 8'h5f;
148         8'h85: sbox = 8'h97;
149         8'h86: sbox = 8'h44;
150         8'h87: sbox = 8'h17;
151         8'h88: sbox = 8'hc4;
152         8'h89: sbox = 8'ha7;
153         8'h8a: sbox = 8'h7e;
154         8'h8b: sbox = 8'h3d;
155         8'h8c: sbox = 8'h64;
156         8'h8d: sbox = 8'h5d;
157         8'h8e: sbox = 8'h19;
158         8'h8f: sbox = 8'h73;
159         8'h90: sbox = 8'h60;
160         8'h91: sbox = 8'h81;
161         8'h92: sbox = 8'h4f;
162         8'h93: sbox = 8'hdc;
163         8'h94: sbox = 8'h22;
164         8'h95: sbox = 8'h2a;
165         8'h96: sbox = 8'h90;
166         8'h97: sbox = 8'h88;
167         8'h98: sbox = 8'h46;
168         8'h99: sbox = 8'hee;
169         8'h9a: sbox = 8'hb8;
170         8'h9b: sbox = 8'h14;
171         8'h9c: sbox = 8'hde;
172         8'h9d: sbox = 8'h5e;
173         8'h9e: sbox = 8'h0b;
174         8'h9f: sbox = 8'hdb;
175         8'ha0: sbox = 8'he0;
176         8'ha1: sbox = 8'h32;
177         8'ha2: sbox = 8'h3a;
178         8'ha3: sbox = 8'h0a;
179         8'ha4: sbox = 8'h49;
180         8'ha5: sbox = 8'h06;
181         8'ha6: sbox = 8'h24;
182         8'ha7: sbox = 8'h5c;
183         8'ha8: sbox = 8'hc2;
184         8'ha9: sbox = 8'hd3;
185         8'haa: sbox = 8'hac;
186         8'hab: sbox = 8'h62;
187         8'hac: sbox = 8'h91;
188         8'had: sbox = 8'h95;
189         8'hae: sbox = 8'he4;
190         8'haf: sbox = 8'h79;
191         8'hb0: sbox = 8'he7;
192         8'hb1: sbox = 8'hc8;
193         8'hb2: sbox = 8'h37;
194         8'hb3: sbox = 8'h6d;
195         8'hb4: sbox = 8'h8d;
196         8'hb5: sbox = 8'hd5;
197         8'hb6: sbox = 8'h4e;
198         8'hb7: sbox = 8'ha9;
199         8'hb8: sbox = 8'h6c;
```

```
200         8'hb9: sbox = 8'h56;
201         8'hba: sbox = 8'hf4;
202         8'hbb: sbox = 8'hea;
203         8'hbc: sbox = 8'h65;
204         8'hbd: sbox = 8'h7a;
205         8'hbe: sbox = 8'hae;
206         8'hbf: sbox = 8'h08;
207         8'hc0: sbox = 8'hba;
208         8'hc1: sbox = 8'h78;
209         8'hc2: sbox = 8'h25;
210         8'hc3: sbox = 8'h2e;
211         8'hc4: sbox = 8'h1c;
212         8'hc5: sbox = 8'ha6;
213         8'hc6: sbox = 8'hb4;
214         8'hc7: sbox = 8'hc6;
215         8'hc8: sbox = 8'he8;
216         8'hc9: sbox = 8'hdd;
217         8'hca: sbox = 8'h74;
218         8'hcb: sbox = 8'h1f;
219         8'hcc: sbox = 8'h4b;
220         8'hcd: sbox = 8'hbd;
221         8'hce: sbox = 8'h8b;
222         8'hcf: sbox = 8'h8a;
223         8'hd0: sbox = 8'h70;
224         8'hd1: sbox = 8'h3e;
225         8'hd2: sbox = 8'hb5;
226         8'hd3: sbox = 8'h66;
227         8'hd4: sbox = 8'h48;
228         8'hd5: sbox = 8'h03;
229         8'hd6: sbox = 8'hf6;
230         8'hd7: sbox = 8'h0e;
231         8'hd8: sbox = 8'h61;
232         8'hd9: sbox = 8'h35;
233         8'hda: sbox = 8'h57;
234         8'hdb: sbox = 8'hb9;
235         8'hdc: sbox = 8'h86;
236         8'hdd: sbox = 8'hc1;
237         8'hde: sbox = 8'h1d;
238         8'hdf: sbox = 8'h9e;
239         8'he0: sbox = 8'he1;
240         8'he1: sbox = 8'hf8;
241         8'he2: sbox = 8'h98;
242         8'he3: sbox = 8'h11;
243         8'he4: sbox = 8'h69;
244         8'he5: sbox = 8'hd9;
245         8'he6: sbox = 8'h8e;
246         8'he7: sbox = 8'h94;
247         8'he8: sbox = 8'h9b;
248         8'he9: sbox = 8'h1e;
249         8'hea: sbox = 8'h87;
250         8'heb: sbox = 8'he9;
251         8'hec: sbox = 8'hce;
252         8'hed: sbox = 8'h55;
253         8'hee: sbox = 8'h28;
254         8'hef: sbox = 8'hdf;
255         8'hf0: sbox = 8'h8c;
256         8'hf1: sbox = 8'ha1;
257         8'hf2: sbox = 8'h89;
258         8'hf3: sbox = 8'h0d;
259         8'hf4: sbox = 8'hbf;
260         8'hf5: sbox = 8'he6;
261         8'hf6: sbox = 8'h42;
262         8'hf7: sbox = 8'h68;
263         8'hf8: sbox = 8'h41;
264         8'hf9: sbox = 8'h99;
265         8'hfa: sbox = 8'h2d;
266         8'hfb: sbox = 8'h0f;
267         8'hfc: sbox = 8'hb0;
268         8'hfd: sbox = 8'h54;
```

```
        8'hfe: sbox = 8'hbb;
        8'hff: sbox = 8'h16;
      endcase
    end
  endfunction

  always @ (*)
  begin
    t0 <= in[7:0];
    t1 <= in[15:8];
    t2 <= in[23:16];
    t3 <= in[31:24];
  end

  assign out[7:0] = sbox(t0);
  assign out[15:8] = sbox(t1);
  assign out[23:16] = sbox(t2);
  assign out[31:24] = sbox(t3);
endmodule
```

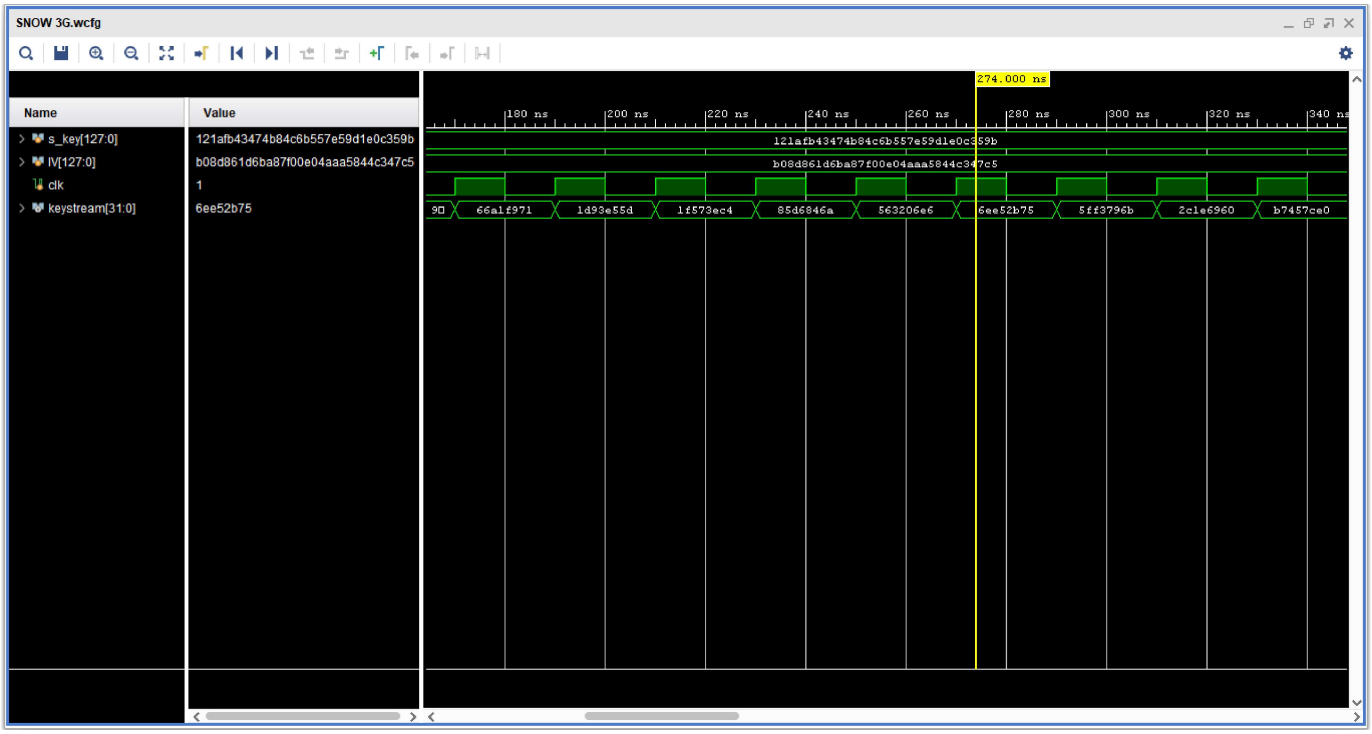
Modulo Addition

```
`timescale 1ns / 1ps

module Modulo_Addition #(parameter two_32=40'd2147483648)(
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] sum
);
    integer a1,b1,sum1;

    always @ (*)
    begin
        a1 <= a;
        b1 <= b;
        sum1 <= a1 + b1;
        if (sum1 < two_32)
            sum <= sum1;
        else if (sum1 == two_32)
            sum <= 32'd0;
        else
            sum <= sum1 % two_32;
    end
endmodule
```

Testbench

```
`timescale 1ns / 1ps

module testbench;
    reg [127:0] s_key;
    reg [127:0] IV;
    reg clk = 0;
    wire [31:0] keystream;

    SNOW_3G tb(.s_key(s_key), .IV(IV), .clk(clk), .keystream(keystream));

    initial
    begin
        s_key = 128'h121afb43474b84c6b557e59d1e0c359b;
        IV = 128'hb08d861d6ba87f00e04aaa5844c347c5;
        clk=0;
        forever #10 clk = ~clk;
    end

endmodule
```

Simulation of SNOW 3G Algorithm



"SNOW 3G" – Test values Result of Simulation

Test values of s_key IV taken in testbench –

- S_key = 128'h121afb43474b84c6b557e59d1e0c359b
- IV = 128'hb08d861d6ba87f00e04aaa5844c347c5
- Clock = 50 MHz (20 ns time period)

Result –

- From 0 ns to 30 ns, there is no output in on the keystream for security and initialization.
- From 30 ns, keystream starts giving output as : ..,7c0fe10e,..,157c9d2a,. . . , 14654a8c,. . . , 96e6bbd3,. . . , e17cce94,...and so on.