CS200 PROJECT

Project: Verilog, ciphers and real-life fault attack (implementation)

By: - Brij Bhushan (1301016)

Computer Science And Engineering Department

Indian Institute of Information Technology

Guwahati , Assam.

Under :- Dr. Sandeep Karmakar

Contents:

Introduction

What is HDL?

HDL is an abbreviation of **Hardware Description Language.** Any digital system can be represented in **a REGISTER TRANSFER LEVEL (RTL)** and HDLs are used to describe this RTL**.**

Verilog:

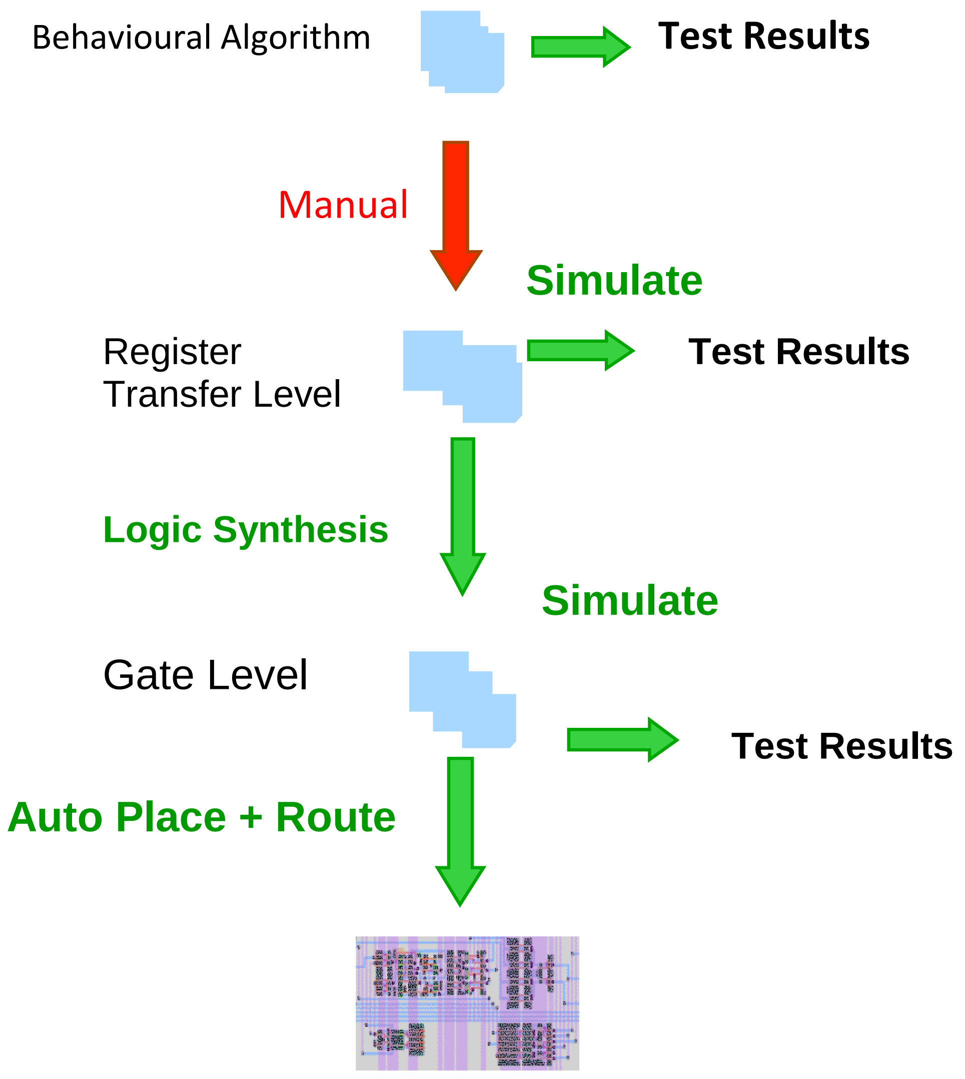
- Verilog is one of the two major Hardware Description Languages (HDL) used by hardware designers in industry and academia.
- VHDL is another one.
- Verilog is easier to learn and use than VHDL.
- Verilog is C-like. VHDL is very Aad -like.
- Verilog HDL allows a hardware designer to describer designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i.e., gate and switch levels).

Why use Verilog HDL ?

- Digital system are highly complex.
- Verilog language provides the digital designer a software platform.
- Verilog allow user to express their design with BEHAVIORAL CONSTRUCTS.

- A program tool can convert the verilog program to a description that was used to make exactly chip, like VLSI.

Verilog in the Design Process:



Verilog HDL: Syntax and Semantics:

Lexical Convention :

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language.

Verilog HDL is a case-sensitive language. All keywords are in lowercase.

- Numbers are specified in the traditional form or below .

  <size><base format><number>

- Size: contains *decimal* digitals that specify the size of the constant in the number of bits.
- Base format: is the single character ‘ followed by one of the following characters *b(binary),d(decimal),o(octal),h(hex).*
- Number: legal digital.
- Example :

```
347   // decimal number
 4’b101  //  4- bit binary number 0101
  2’o12  // 2-bit octal number
5’h87f7   // 5-bit hex number h87f7
 2’d83  // 2-bit decimal number
```

- String in double quote ---

  “ this is a introduction”

- Operator are one, two, or three characters and are use in the expressions.

  just like C++.

- Identifier: specified by a letter or underscore followed by more letter or digits, or signs.

  identifier can up to 1024 characters

Data Types:

There are 2 groups of data types in *Verilog*, namely **physical** and **abstract**.

a) **Physical data type** :

- Net (`wire`, `wand`, `wor`, `tri`, `triand`, `trior`). Default value is `z`. Used mainly in structural modeling.
- Register (`reg`). Default value is `x`. Used in dataflow/RTL and behavioral modelings.
- Charge storage node (`trireg`). Default value is `x`. Used in gate-level and switch level modelings.

b) **Abstract data type** : used only in behavioral modeling and test fixture.

- Integer (`integer`) stores 32-bit signed quantity.
- Time (`time`) stores 64-bit unsigned quantity from system task `$time`.
- Real (`real`) stores floating-point quantity.
- Parameter (`parameter`) substitutes constant.
- Event (`event`) is only name reference — does not hold value.

Unfortunately, the current standard of *Verilog* does not support user-defined types, unlike *VHDL*.

Every signal has a data type associated with it. Data types are:

- ***Explicitly declared*** with a declaration in the Verilog code.
- ***Implicitly declared*** with no declaration but used to connect structural building blocks in the code. Implicit declarations are always net type "wire" and only one bit wide.

Verilog Statements: Verilog has two basic types of statements.

1. Concurrent statements (combinational)
(things are happening concurrently, ordering does not matter)

- ♉ Gate instantiations
  and (z, x, y), **or** (c, a, b), **xor** (S, x, y), etc.
- ♉ Continuous assignments
  assign Z = x & y; c = a | b; S = x ^ y

2. Procedural statements (sequential)
(executed in the order written in the code)

- ♉ **always @** - executed continuously when the event is active
  always @ (posedge clock)
- ♉ **initial** - executed only once (used in simulation)
- ♉ **if then else** statements

Modules:

- Module are the building blocks of Verilog designs .
- You create design hierarchy by instantiating modules in other modules.
- An instance of a module can be called in another, higher-level module.

Syntax:

module module_name (port_list);
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
... *statements* ...

endmodule

Ports :

• Ports allow communication between a module and its environment.
• All but the top-level modules in a hierarchy have ports.
• Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is :
input [range_val:range_var] list_of_identifiers;
output [range_val:range_var] list_of_identifiers;
inout [range_val:range_var] list_of_identifiers;

Program structure:

module *<module name> (< port list>);*

< declares>

<module items>

Endmodule

Module name

an identifier that uniquely names the module.

Port list

a list of input, inout and output ports which are used to other modules.

Declare: *section specifies data objects as registers, memories and wires as well as procedural constructs such as functions and tasks.*

Module items

initial constructs

always constructs

assignment

Test Module structure:

```
module <test module name> ;
// Data type declaration
// Instantiate module ( call the module that is going to be tested)
// Apply the stimulus
// Display results
endmodule
```

IMPLEMENTATION:

1- NAND GATE:

```
  //Behavioral model of a NAND gate
    // program for NAND gate
      module NAND(in1, in2, out);
              input in1,in2;  output out;
         assign out=~(in1&in2);
             endmodule
```

Test module test_nand for the NAND

```
  module test_nand;
            reg a,b;   wire  out1;
       NAND test_nand (a,b,out1); // call the module NAND.
 initial begin     // apply the stimulus, test data
            a=0; b=0; // initial value
             #1 a=1; // delay one simulation cycle,then change a=1.
             #1 b=1;
        end
   endmodule
```

2- HALF ADDER:-

```
  //Behavioral model of a NAND gate
 // program for NAND gate
module halfadder(
                    input a, b,
                          output s,c   );
                                         assign s = (a^b);
                                            assign c = a&b;
endmodule
```

Test module ha_tb for the halfadder :

```
module ha_tb;
          reg a, reg b; // inputs
          wire s, wire c; // Outputs
          // Instantiate the Unit Under Test (UUT)
          halfadder uut (
                    .a(a),
                    .b(b),
                    .s(s),
                    .c(c)    );
          initial begin
                        a = 0;b = 0;
                       #10 a = 0;b = 1;    // Initialize Inputs
                        #10 a = 1;b = 0;
                           #10 a = 1;b = 1;

          end
      endmodule
```

3- FULL ADDER:-

```
  //Behavioral model of a NAND gate
  // program for NAND gate
module fulladder(
   input a,b,c,
   output s,cout
   );
assign s = a^b^c;
assign cout = (a&b)|(b&c)|(c&a);
endmodule
```

Test module fulladder_tb for the fulladder :

```
module fulladder_tb;

                reg a, reg b, reg c;  // inputs

                wire s , wire cout; // Outputs

                // Instantiate the Unit Under Test (UUT)
                fulladder uut (
                        .a(a),
                        .b(b),
                        .c(c),
                        .s(s),
                        .cout(cout)   );
                initial begin
                        // Initialize Inputs
                           a = 0;b = 0;c = 0;
                        #010;a = 0;b = 1;c = 0;
                        #010;a = 0;b = 1;c = 1;
                        #010;a = 1;b = 1;c = 1;

                        end
     endmodule
```

4- EDGE TRIGGERED J K FLIP FLOP:

Assign characteristic function to Q on rising clock edge (Q_+ = JQ' + K'Q)

Code:

```
        module JKflipflop(J, K, Clock, Q);
                input J,K, Clock;
                 output Q;
                  reg Q;
                 always @(posedge Clock)
                   Q = J && ~Q || ~K && Q; // Q+ = JQ' + K'Q
endmodule
```

5- 8-bit up Counter :

```
module up_down_counter    (
out      ,            // Output of the counter
up_down  ,      // up_down control for counter
clk      ,            // clock input
reset                // reset input
);
output [7:0] out; //Output Ports

input [7:0] data;                         //Input Ports
input up_down, clk, reset;
reg [7:0] out;

always @(posedge clk)
if (reset) begin                         // active high reset
  out <= 8'b0 ;
    end else if (up_down) begin
        out <= out + 1;
       end else begin
         out <= out - 1;
end
```

D FLIP FLOP:

```
`timescale 1ns / 1ps
module dff( input D, clk , rst,
            output reg Q,
        output reg Qn );
  always @ ( posedge clk or negedge rst)
         begin
             if(rst == 1'b0)
                 begin
                    Q <= 0;
                          Qn <= 1;
                       end
              else
                 begin
                          Q <= D;
                          Q <= !D;
                        end
        end
                 endmodule
```

Test module for the D flip flop :

```
`timescale 1ns / 1ps
module dff_tb;
  localparam T = 20;
        // Inputs
        reg D_t;
        reg clk_t;
        reg rst_t;
        // Outputs
        wire Q_t;
```

```
	wire Qn_t;
	// Instantiate the Unit Under Test (UUT)
	dff uut (
		.D(D_t),
		.clk(clk_t),
		.rst(rst_),
		.Q(Q_t),
		.Qn(Qn_t)
	);
	always
	begin
       clk_t= 1'b1;
		    #(T/2);
       clk_t = 1'b0;
		   #(T/2);
		  end

			initial begin
		   rst_t=1'b1;
						#(T/2);
						clk_t= 1'b0;

		end
	initial begin
		D_t=0;
		#T;
		D_t=1;
		#T;
		D_t=0;
		#T;

	end

endmodule
```

GRAIN CIPHER IMPLEMENTATION

Grain Cipher:

```
module CLOCK (

            output reg clock);

            initial

            #5 clock =1;

            always

            #50 clock = ~ clock;

endmodule

module Grain_128 ( wire Clock, wire Z);

               CLOCK Clock_gen (Clock);

              Grain GRAIN_128 (Z,Clock);

endmodule

module Grain( output reg Z, input clock );

 reg [127:0] LFSR =
128'b0000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;

 reg [127:0] NFSR =
128'b0000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;

 reg [127:0] IV =
96'b101010101111010010100101010100010101010101010101010010101000011110;

 reg [127:0] Key =
128'b1010101110001010001010000010111110101010101100010010101000001010101010
10111111010010101010010101;

 reg [127:0] NFSR_copy;

 reg [127:0] LFSR_copy;
 reg [9:0] i;
 reg H,z;
 initial begin
 for(i=0 ; i < 128; i=i+1)
 NFSR[i] = Key[i];
 for(i=0 ; i < 96; i=i+1)
 LFSR[i] = IV[i];
 for(i=96 ; i < 128; i=i+1)
 LFSR[i] = 1;
 for(i=0 ; i < 256; i=i+1) begin
 H = (NFSR[12] & LFSR[8]) ^ (NFSR[95] & LFSR[42]) ^ (LFSR[13] & LFSR[20]) ^ (LFSR[60] &
LFSR[79]) ^ (NFSR[12] & NFSR[95] & LFSR[95]) ;
 z = NFSR[2] ^ NFSR[15] ^ NFSR[36] ^ NFSR[45] ^ NFSR[64] ^ NFSR[73] ^ NFSR[89] ^ LFSR[93] ^H;
 copy (NFSR_copy, NFSR);
 copy (LFSR_copy, LFSR);
 NFSR_Update (NFSR, NFSR_copy, LFSR[0]);
 LFSR_Update (LFSR, LFSR_copy);
 NFSR[127] = NFSR[127] ^ z;
 LFSR[127] = LFSR[127] ^ z;
  end
 end
 always@(posedge clock) begin
 H = (NFSR[12] & LFSR[8]) ^ (NFSR[95] & LFSR[42]) ^ (LFSR[13] & LFSR[20]) ^ (LFSR[60] &
LFSR[79]) ^ (NFSR[12] & NFSR[95] & LFSR[95]) ;
 Z = NFSR[2] ^ NFSR[15] ^ NFSR[36] ^ NFSR[45] ^ NFSR[64] ^ NFSR[73] ^ NFSR[89] ^ LFSR[93] ^
H;
 copy (NFSR_copy, NFSR);
 copy (LFSR_copy, LFSR);
 NFSR_Update (NFSR, NFSR_copy, LFSR[0]);
 LFSR_Update (LFSR, LFSR_copy);
```

```
end
task copy (
 output reg [127:0] r1,
 input [127:0] r2);
 begin: copy
 reg [8:0] i;
 for(i=0;i<128;i=i+1)
 r1[i]=r2[i];
 end
 endtask
task shiftRegister(
           inout [127:0] R1 );
           begin:shift
           reg [8:0] i;
           for(i=0;i< 127; i=i+1)
           R1[i]=R1[i+1];
           end
           endtask

task LFSR_Update (
                       output reg [127:0] L_,
                       input [127:0] L);
begin: Update
reg temp ;
temp = L[0] ^ L[7] ^ L[38] ^ L[70] ^ L[86] ^ L[96];
copy (L_,L);
shiftRegister (L_);
L_[127] = temp;
```

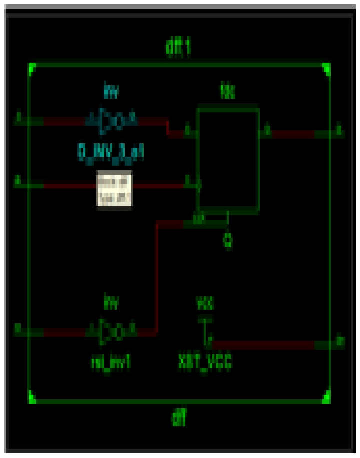
```
 end
 endtask

 task NFSR_Update (output reg [127:0] L_,input [127:0] L, S);
 begin: Update
 reg temp;
 temp  = S ^ L[0] ^ L[26] ^ L[56] ^ L[91] ^ (L[3] & L[67] )^ (L[13] & L[11] ) ^ (L[17] & L[18] ) ^
(L[27] & L[59] ) ^ (L[40] & L[48] ) ^ (L[61] & L[65] ) ^ (L[84] & L[68] );
 copy (L_,L);
 shiftRegister (L_);
 L_[127] = temp;
  end
 endtask
  endmodule
```

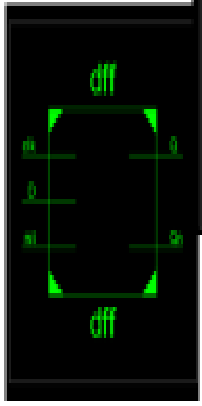
Test Module for Grain Cipher:

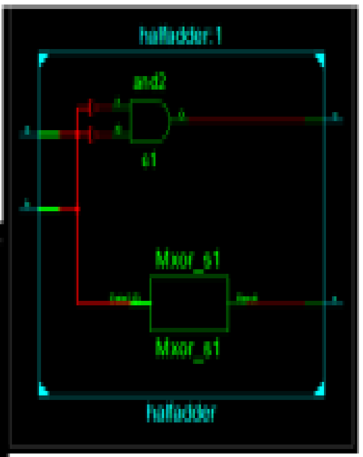
```
module Grain_tb;
	// Instantiate the Unit Under Test (UUT)
	Grain_128 uut (
	.Clock(Clock),
	.Z(Z)
	);

	initial begin
		// Initialize Inputs
		//Clock =0;
		//Z=0;
		// Wait 100 ns for global reset to finish
		#100;
```

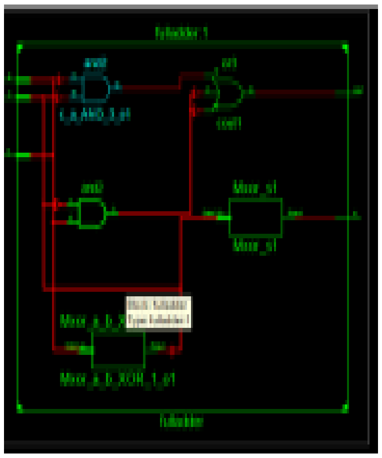
```
                    // Add stimulus here
          end
     endmodule
```

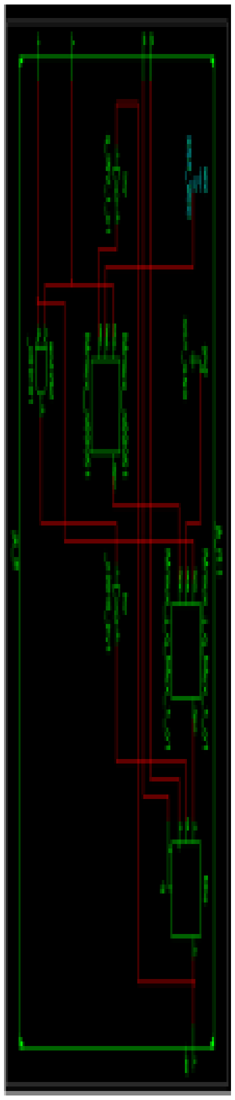
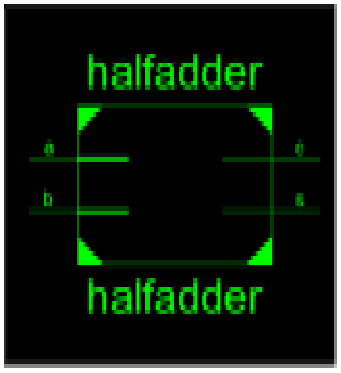
Output of the Implementations:

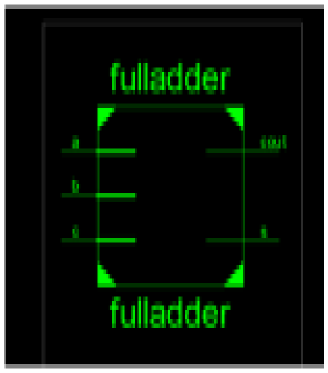


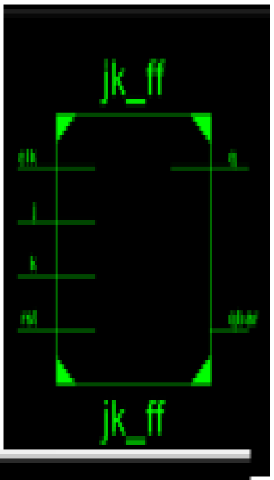


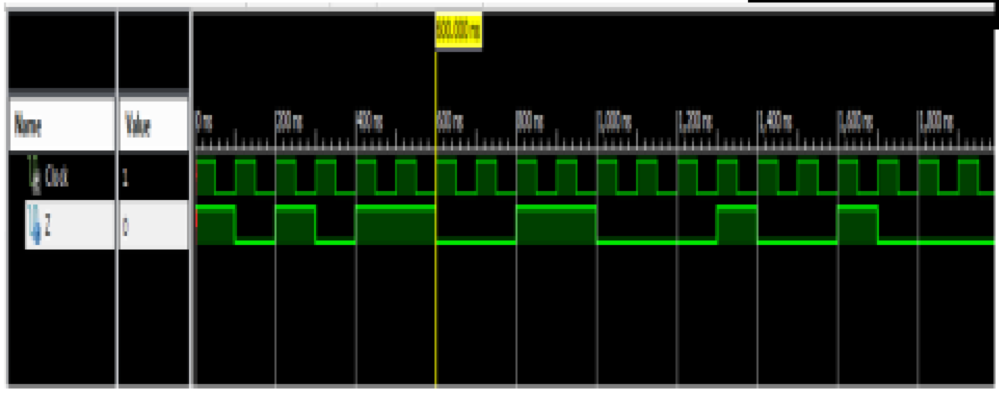




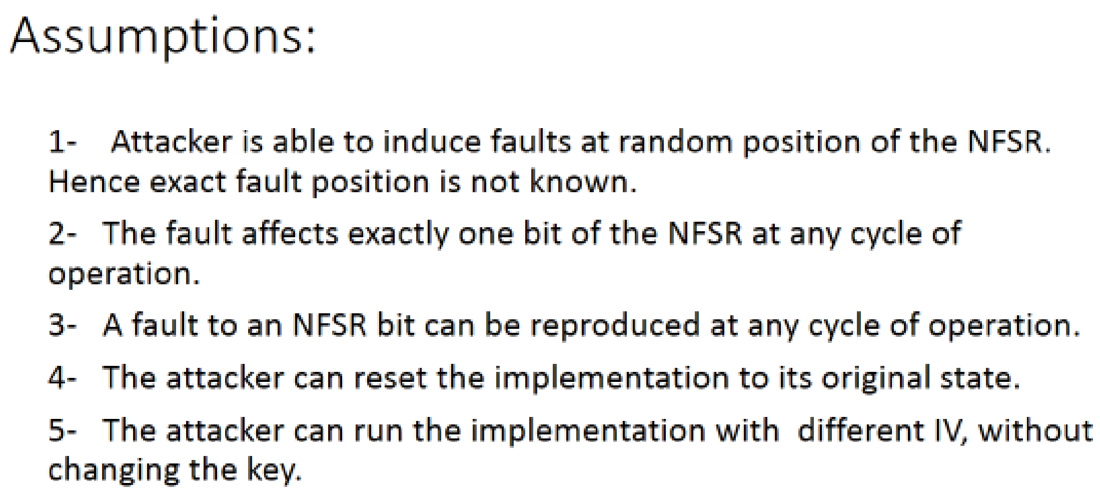









Study of fault analysis of Grain-128 Cipher:


Assumptions:

1- Attacker is able to induce faults at random position of the NFSR. Hence exact fault position is not known.

2- The fault affects exactly one bit of the NFSR at any cycle of operation.

3- A fault to an NFSR bit can be reproduced at any cycle of operation.

4- The attacker can reset the implementation to its original state.

5- The attacker can run the implementation with different IV, without changing the key.

Steps in faults analysis:

1-Inducing fault in NFSR

2-Determining fault location

3-Determining NFSR bits

4-Determining LFSR bits

- Determining fault location

$Z_i = b_{i+2} + b_{i+15} + b_{i+36} + b_{i+45} + b_{i+64} + b_{i+73} + b_{i+89} + h + s_{i+93}$

- Determining NFSR bits

$Z_i = S_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}$

- Determining NFSR bits

$Z_i = b_{i+2} + b_{i+15} + b_{i+36} + b_{i+45} + b_{i+64} + b_{i+73} + b_{i+89} + b_{i+12} S_{i+8} + S_{i+13} S_{i+20} + b_{i+95} S_{i+42} + S_{i+60} S_{i+79} + b_{i+12}b_{i+95} S_{95} + s_{i+93}$

XILINX TOOL:

The Xilinx –ISE Project Navigator provides basic plateform for the design development of the '.v' file designed by the user.

Once the design files have been created , designer can synthesize his design using Xilinx Synthesis Technology (XST).

The synthesis process will check code syntax, and analyze the hierarchy of your design.

These Processes will ensure your design is optimized for design architecture you have selected.

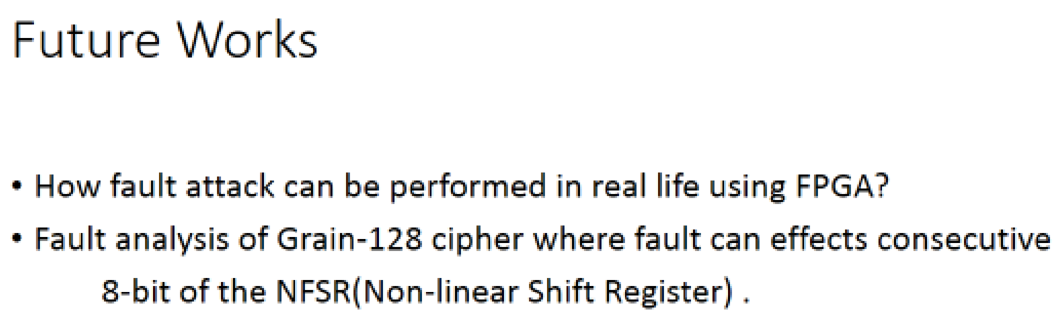
XST is executed from the main window of Xilinx –ISE project Navigator.

When the HDL code is completed with Verilog there are several steps that must be performed in order to synthesize the HDL design.

The “check_syntax “ command verifies the design and identifies the errors to be corrected.

Using XST can be a very time consuming task and for the particular design , it took approximately one or two minutes to create a netlist file.

The RTL schematic option will display the graphical structure of Verilog code.



Future Works

- How fault attack can be performed in real life using FPGA?
- Fault analysis of Grain-128 cipher where fault can effects consecutive 8-bit of the NFSR(Non-linear Shift Register) .

<u>THANK YOU…</u>