

A Whitepaper on SRAM FPGA security

Written by Ray Schouten
x1.2

1.0 Introduction

SRAM-based FPGAs are non-volatile devices. Upon powerup, They are required to be programmed from an external source. This procedure allows anyone to easily monitor the bit-stream, and clone the device. The problem then becomes how can you effectively protect your intellectual property from others in an architecture where the part is externally programmed?

Implementing a microprocessor to configure the device does not address this security issue. The microprocessor must still write the configuration data externally. The configuration data is of finite length and can therefore be captured and used to configure another FPGA.

Most FPGA vendors do not publish the definition of the bit-stream. It is therefore very difficult to reverse engineer a design from a configuration bit-stream.

A FPGA bit-stream cannot be encrypted externally, because there is no way to decrypt the data before it programs the SRAM elements. However, you can program the FPGA and require an external device to write a random number that will enable the operation of the FPGA. Without the proper access code, the FPGA is programmed, but disabled and non-functional.

2.0 References

1. B. Schneier, "Applied Cryptography 2nd edition", Wiley ISBN 0-471-11709-9

3.0 Scope

The method discussed in this document exploits the fact that it is difficult to reverse engineer a design from the configuration bit-stream. This method also requires a simple secure programmable device such as an E²PROM PLD or embedded microcontroller with security bits.

This method uses a pseudo-random sequence generator in both the secure device and the FPGA. Upon powerup, the FPGA gets programmed externally. After the FPGA is programmed, it is disabled and non-functional. The secure device detects that the FPGA has been programmed successfully and begins to communicate with the FPGA. Using pseudo-random-sequence generators, the logic passes data to the FPGA. The outputs of both sequence generators are compared in the FPGA, and the FPGA is enabled if the sequences compare.

It is important that the pseudo-random sequence satisfy the following two requirements:

1. It must be of sufficient length to make capturing the entire sequence impractical.

2. It must be very difficult to determine the seeds (or keys) to the pseudo-random sequence generator, even if the architecture and configuration of the sequence generator is known.

While it is never possible to guarantee security, the method described in this document should make the SRAM FPGA design as secure (or very close to as secure) from duplication as it would be if implemented in a more secure technology such as Antifuse FPGA, E²PROM PLD or custom ASIC.

4.0 Requirements

The pseudo-random sequence generator used in this design must have the characteristics outlined in 3.0 above, and must be easy and compact to implement in hardware or software. In this application, the algorithm does not need to be fast.

There is some logic that must be implemented in a secure device. The logic has several parameters that can increase the complexity of the security algorithm, which will also increase the amount of logic required. The level of complexity we have chosen is such that the logic will fit into a 32 macrocell device, Altera's EPM 7032. Again, if you increase the complexity, the secure device logic will increase.

5.0 Background

This design uses Linear Feedback Shift Registers (LFSRs) to generate the pseudo-random data stream. A feedback shift register is made up of two parts as shown in Figure 1, a shift register and a feedback function.

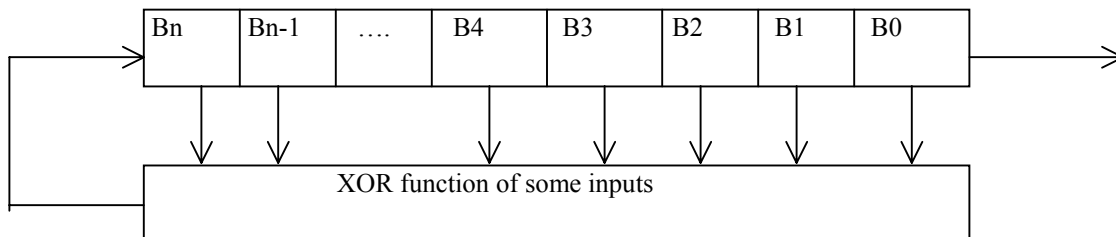


Fig. 1 – A Linear Feedback Shift Register

The sequence begins with a seed (or key) which is present (or loaded into) the shift register at the start of operation.

If it is configured as a maximal length LFSR, the sequence generated will be of length $2^n - 1$ bits, where n is the number of bits in the LFSR. The -1 factor is because a seed of 0 will cause an infinite sequence of 0s to be generated. It is therefore important not to use a seed of 0.

Not all LFSRs will be maximal length LFSRs. The following table provides register lengths and corresponding taps to generate maximal length LFSRs. This table does not necessarily cover every possible maximal length configuration.

Table 1 – Some Primitive Polynomials Mod 2

LFSR Length	Tap 1	Tap 2	Tap 3	Tap 4	Tap 5
2	1	0			
3	2	0			
4	3	0			
5	4	1			
6	5	0			
7	6	0			
7	6	2			
8	7	3	2	1	
9	8	3			
10	9	2			
11	10	1			
12	11	5	3	0	
13	12	3	2	0	

Because of the relatively short sequence length from individual LFSRs, LFSRs are combined in different ways to generate longer, more random and more secure sequences. A long sequence is not necessarily a secure sequence. The security of several algorithms has been evaluated by military applications. How secure an algorithm is depends on how easy it is to crack. A factor known as 'linear complexity' is one method of evaluating different algorithms. The linear complexity of an algorithm is defined as the minimum length LFSR that can mimick the output of the algorithm. An algorithm with a high linear complexity is not necessarily secure.

Reference 1 describes several algorithms and discusses their relative merits. The one chosen for this application is the Gollman Cascade.

The Gollman Cascade consists of k LFSRs of any length. The inverse of the output of the first LFSR enables the clock to the second LFSR. The exclusive-OR of this enable and the output of the 2nd generator enables the 3rd and so on.

If all the LFSRs in the Gollman cascade were of equal length n , the linear complexity of the Gollman cascade would be $n(2^n - 1)^{k-1}$.

The Gollman cascade was chosen because it is relatively secure as well as being easy to implement.

Figure 2 shows the interconnection between the non-secure FPGA and the secure EPLD. Data2, secure, and cmp_ena are test points that are not necessarily required in the final design.

Figure 3 shows the actual logic that is implemented in the FPGA. It includes the attached VHDL files.

Figure 4 shows the logic in the secure EPLD. It includes one of the attached VHDL file.

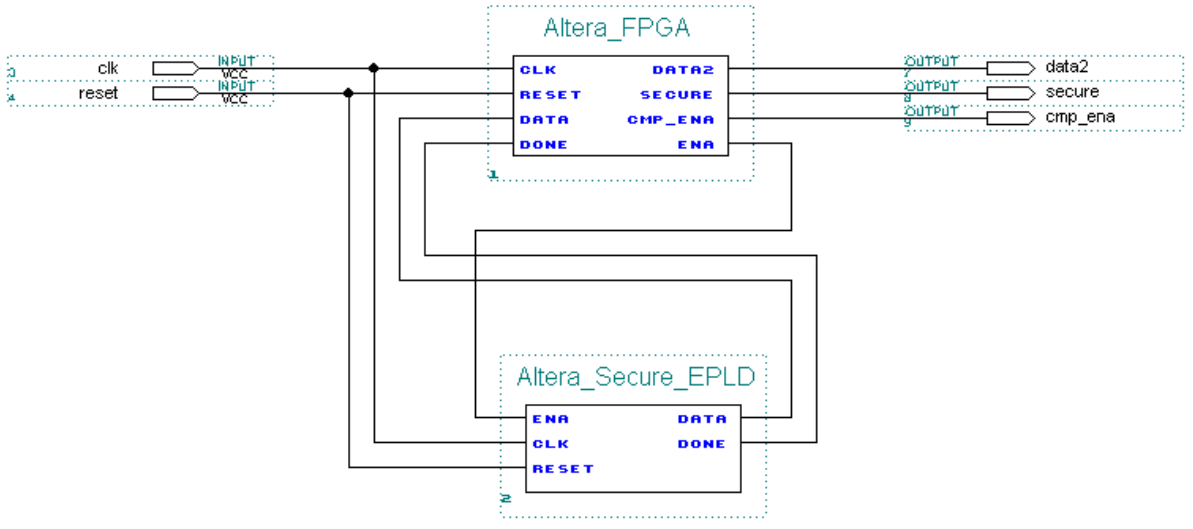


Figure 2. FPGA and EPLD interconnection

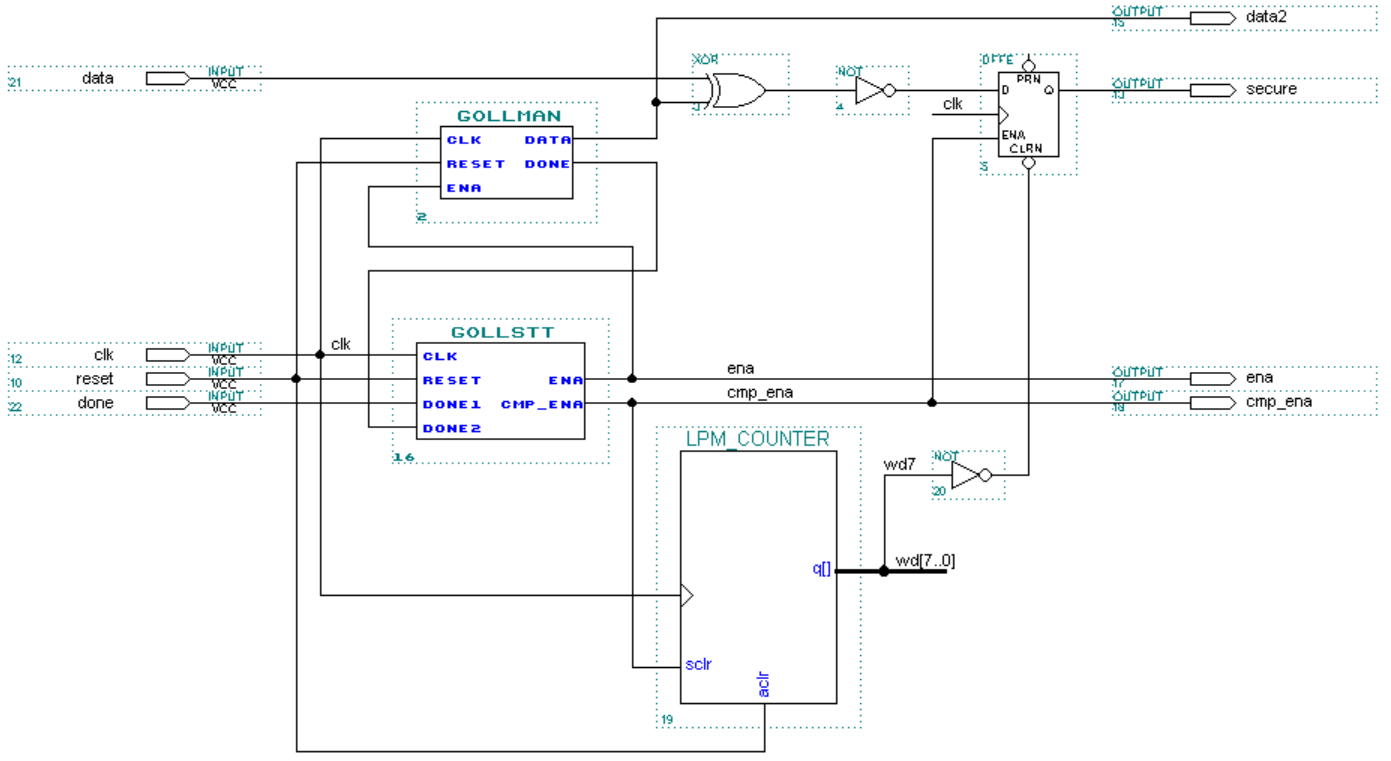
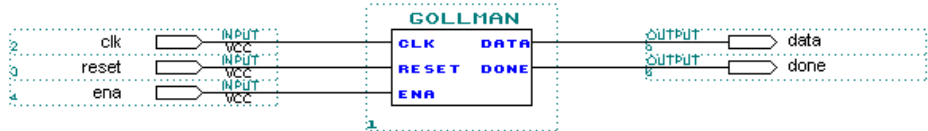


Figure 3. FPGA schematic



Gollman.vhd

```
-- Synchronous VHDL implementation of Gollman Cascade pseudo-random sequence generator
-- See Applied Cryptology 2nd edition by Bruce Schneier - ISBN 0-471-11709-9 p.387
-- Uses ena/done signals as handshake, making it compatible with multi-clocked
-- Implementations (like microprocessor based)
-- This implementation will generate a non-repeating sequence of > 2^1500282 bits
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY gollman IS
    PORT (
        clk          : IN std_logic;
        reset        : IN std_logic;
        ena          : IN std_logic;
        data         : OUT std_logic;
        done         : OUT std_logic);
END gollman;

ARCHITECTURE synth OF gollman IS

    CONSTANT keya      : std_logic_vector(6 downto 0) := "1011011";      -- 1st LFR key
    CONSTANT keyb      : std_logic_vector(3 downto 0) := "1001";        -- 2nd LFR key
    CONSTANT keyc      : std_logic_vector(2 downto 0) := "011";        -- 3rd LFR key
    CONSTANT keyd      : std_logic_vector(11 downto 0) := "011010111010"; -- 4th LFR key

    SIGNAL sra          : std_logic_vector(6 downto 0);      -- 1st LFR instantiation
    SIGNAL srb          : std_logic_vector(3 downto 0);      -- 2nd
    SIGNAL src          : std_logic_vector(2 downto 0);      -- 3rd
    SIGNAL srd          : std_logic_vector(11 downto 0);      -- 4th

    SIGNAL enab         : std_logic;      -- Clock enables for cascade LFRs
    SIGNAL enac         : std_logic;
    SIGNAL enad         : std_logic;

    SIGNAL last_ena     : std_logic;      -- Registered 'ena
    SIGNAL done1        : std_logic;      -- internal 'done' node

    SIGNAL iena         : std_logic;      -- internal 'single clock' ena
    SIGNAL oena         : std_logic;      -- registered version of ena - for edge detect

BEGIN

    -- Process to edge detect enable signal to generate single clock duration enable
    enable: PROCESS(clk,reset)
    BEGIN
        IF reset = '1' THEN
            iena <= '0';
            oena <= '0';
        ELSIF clk'event AND clk = '1' THEN
            oena <= ena;
            IF ena = '1' AND oena = '0' THEN
                iena <= '1';
            ELSE
                iena <= '0';
            END IF;
        END IF;
    END PROCESS enable;

    -- Implementation of 1st LFR in this cascade
    lfr1: PROCESS(clk,reset)
    BEGIN
        IF reset = '1' THEN
            sra <= keya;
        ELSIF clk'event AND clk = '1' THEN
            IF iena = '1' THEN
                sra(5 downto 0) <= sra(6 downto 1); -- Perform Shift
                sra(6) <= sra(6) XOR sra(2); -- XOR bits 6,2 for maximal
            END IF;
        END IF;
    END PROCESS lfr1;
END ARCHITECTURE synth;
```

```

ELSE
    sra <= sra;
END IF;
enab <= NOT sra(1);
-- length
-- Generate enable for next
-- LFR in cascade

ELSE
    sra <= sra;
    enab <= enab;
END IF;
END PROCESS lfr1;

-- Implementation of 2nd LFR in this cascade
lfr2: PROCESS(clk,reset)
BEGIN
    IF reset = '1' THEN
        srb <= keyb;
    ELSIF clk'event AND clk = '1' THEN
        IF iena = '1' AND enab = '1' THEN
            srb(2 downto 0) <= srb(3 downto 1); -- Perform Shift
            srb(3) <= srb(3) XOR srb(0); -- XOR bits 3,0 for
            -- maximal length
        ELSE
            srb <= srb;
        END IF;
        enac <= (NOT sra(1)) XOR srb(1); -- Generate enable for
        -- next LFR in cascade
    ELSE
        srb <= srb;
        enac <= enac;
    END IF;
END PROCESS lfr2;

-- Implementation of 3rd LFR in cascade
lfr3: PROCESS(clk,reset)
BEGIN
    IF reset = '1' THEN
        src <= keyc;
    ELSIF clk'event AND clk = '1' THEN
        IF iena = '1' AND enac = '1' THEN
            src(1 downto 0) <= src(2 downto 1); -- Perform Shift
            src(2) <= src(2) XOR src(0); -- XOR bits 2,0 for
            -- maximal length
        ELSE
            src <= src;
        END IF;
        enad <= (NOT sra(1)) XOR srb(1) XOR src(1); -- Generate enable for
        -- next LFR in cascade
    ELSE
        src <= src;
        enad <= enad;
    END IF;
END PROCESS lfr3;

-- Implementation of 4th LFR in cascade
lfr4: PROCESS(clk,reset)
BEGIN
    IF reset = '1' THEN
        srd <= keyd;
    ELSIF clk'event AND clk = '1' THEN
        IF iena = '1' AND enad = '1' THEN
            srd(10 downto 0) <= srd(11 downto 1); -- Perform Shift
            srd(11) <= srd(11) XOR srd(5) XOR srd(3) XOR srd(0);
            -- XOR bits 11,5,3,0 for maximal length
        ELSE
            srd <= srd;
        END IF;
    ELSE
        srd <= srd;
    END IF;
    data <= srd(0);
    -- Assign output
END PROCESS lfr4;

```

```
-- Simple process to handle DONE signal in this h/w implementation
-- In order to be compatible with a microprocessor-type implementation, this process
-- implements a handshake. Done goes Low after ENA goes high, and then High after
-- ENA goes low. As this is hardware, the result is available immediately, but in a
-- micro, done would go high once the next result is available, ie. many clocks after ENA
-- goes low.
-- The top-level schematic handles the other side of the handshake.
```

```
hshake: PROCESS(clk,reset)
BEGIN
    IF reset = '1' THEN
        donei <= '1';
    ELSIF clk'event AND clk = '1' THEN
        last_ena <= ena;
        IF ena = '1' AND last_ena = '0' THEN
            donei <= '0';
        ELSIF iena = '0' AND last_ena = '1' THEN
            donei <= '1';
        ELSE
            donei <= donei;
        END IF;
    END IF;
    done <= donei;
END PROCESS hshake;

END synth;
```

Gollstt.vhd

```
-- State machine to implement handshake between two unsynchronized pseudo-random
-- sequence generators. Using this handshake technique, one of the sequence generators
-- can be implemented in h/w and one in s/w.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY gollstt IS
    PORT (
        clk          : IN std_logic;
        reset        : IN std_logic;
        done1        : IN std_logic;           -- Generator 1 done signal
        done2        : IN std_logic;           -- Generator 2 done signal
        ena          : OUT std_logic;          -- Generator Enable
        cmp_ena      : OUT std_logic);        -- Comparator output register

enable
END gollstt;

ARCHITECTURE synth OF gollstt IS

TYPE state_values IS (st0 , st1 , st2 , st3);
SIGNAL pres_s, next_s : state_values;           -- State Vector

BEGIN

-- This state machine's outputs depend only on state variable, regardless of
-- implementation, therefore this is a Moore State Machine

-- This process simply resets the state m/c and
state_reg: PROCESS(clk,reset)
BEGIN
    IF reset = '1' THEN
        pres_s <= st0;
    ELSIF clk'event AND clk = '1' THEN
        pres_s <= next_s;
    END IF;
END PROCESS state_reg;

-- This is the combinatorial process
state_mc: PROCESS(done1,done2,pres_s)
BEGIN
    CASE pres_s IS
        WHEN st0 =>
            next_s <= st1;
        WHEN st1 =>
            IF done1 = '1' OR done2 = '1' THEN
                next_s <= st1;
            ELSE
                next_s <= st2;
            END IF;
        WHEN st2 =>
            IF done1 = '0' OR done2 = '0' THEN
                next_s <= st2;
            ELSE
                next_s <= st3;
            END IF;
        WHEN st3 =>
            next_s <= st1;
        WHEN others =>
            next_s <= st0;
    END CASE;
END PROCESS state_mc;

-- This process assigns the outputs
outputs: PROCESS(pres_s)
BEGIN
    IF pres_s = st1 THEN
        ena <= '1';           -- Enable during state 1
    ELSE

```

```
        ena <= '0';
    END IF;
    IF pres_s = st3 THEN
        cmp_ena <= '1';
    ELSE
        cmp_ena <= '0';
    END IF;
END PROCESS outputs;

END synth;
```

-- Comparator enable during state 3