Secure Sequence Recognizer

Author:Iacopo Pacini

Index

1 Introduction

A Secure Sequence Recognizer is a Mealy sequential state machine(In a Mealy machine, output depends on the present state and the external input).

It can detect the beginning of a packet of asynchronous data, like that coming in over wireless or a serial port. Can be used in a remote control application too, such as for a TV or garage door opener.

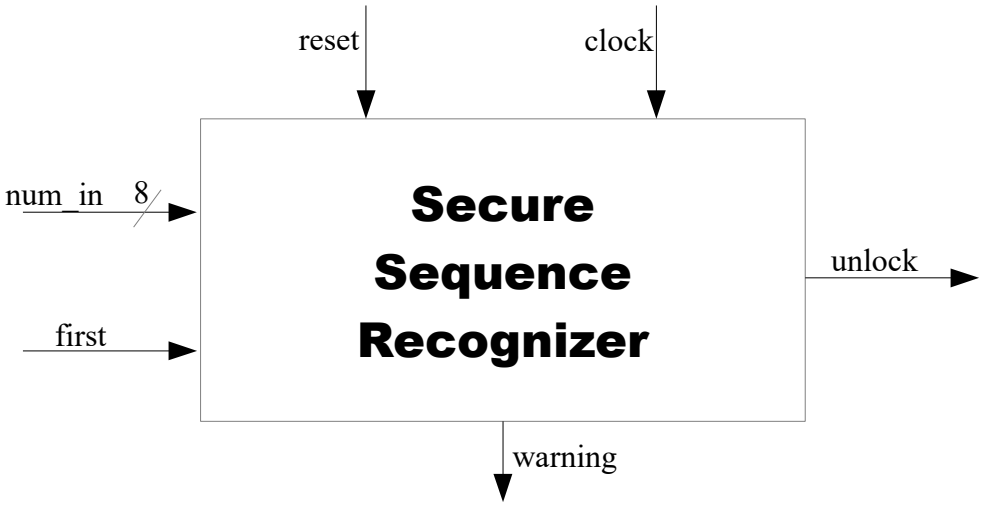
The device implemented is a synchronous sequence recognizer, the sequence numbers are five 8-bit integers. At the beginning of the process the signal must be set to '1' for one clock cycle and the following must be send to the SSR at the rate of 1 clock cycle.

The “first” pin must be kept high for no more than 1 clock cycle otherwise the SSR stops working until a future reset.

The sampling duration is always five clock cycles long and at the end the pin “unlock” is kept high for one clock cycle if the sequence has been inserted correctly.

Instead,if there is at least a wrong number in the sequence the pin “warning” goes to '1' for a clock cycle.

If it is being inserted a wrong sequence more than three times consecutively the SSR stops working and “warning” goes to '1' permanently or at least until the input pin “reset” goes to '0'



Picture 1: An I\O view of the device

2 Architecture

The SSR is a finite state machine whose state is described in the STAR register,the latter is a 3-bit register and the possible state are:

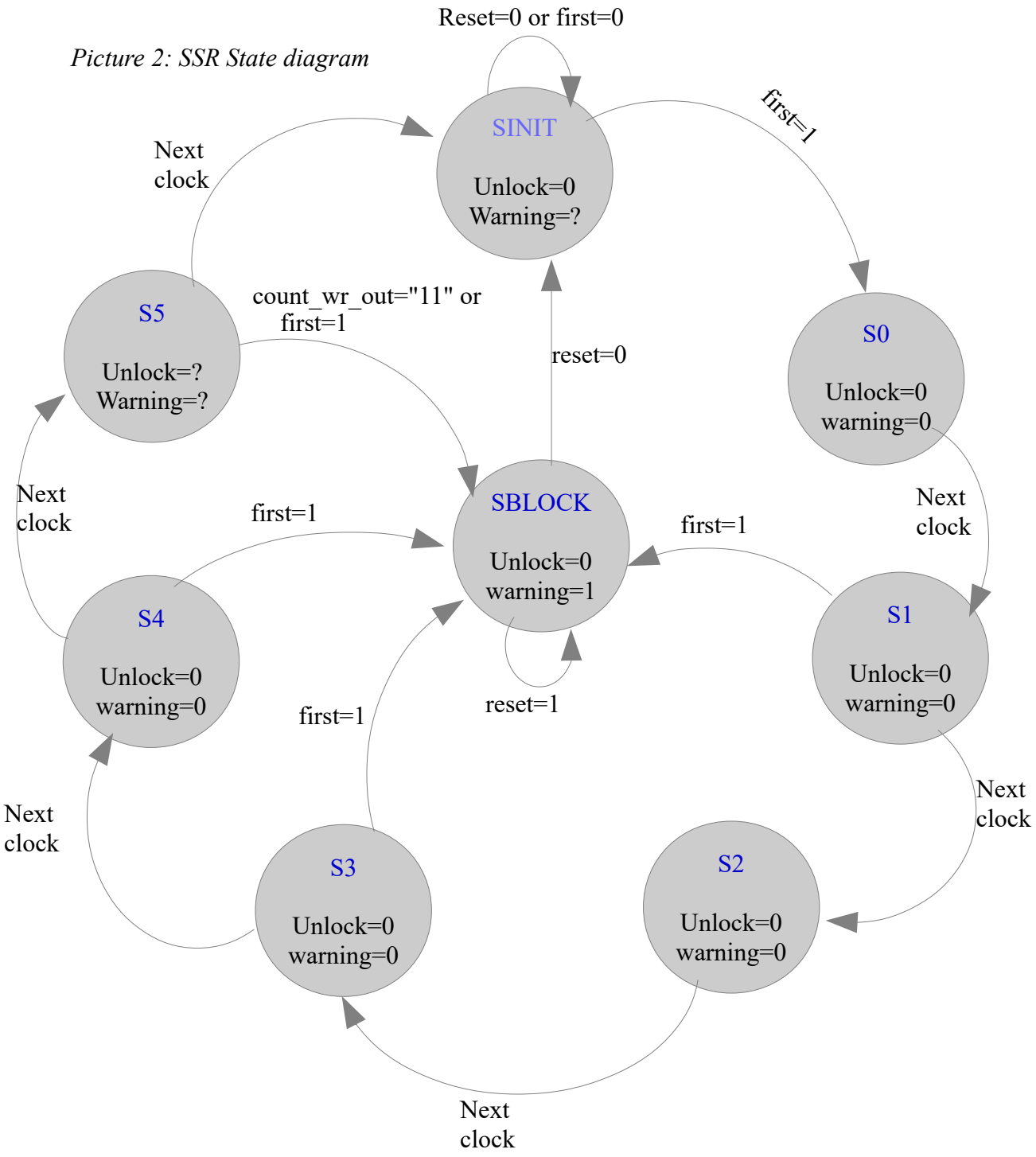
1) **SINIT:**the SSR in this state waits for the first sequence number and enter this state whenever the pin “reset” goes to '1'
2) **S0,S1,S2,S3,S4:**In each of those states the SSR samples the input sequences and checks for their correctness
3) **S5:**In this state the “unlock” pin is set to '1' if there were no input errors otherwise the 'warning' pin goes to '1' for a clock cycle.
4) **SBLOCK**: The SSR enters this state whenever the 'first' pin is kept high too long or in the wrong moment and if the sequence is inserted incorrectly for at least three consecutive times. The device leaves this state only if the 'reset' becomes high.

Some other tools are needed in order to make a correct sampling:

- OK.: 1-bit register that contains 1 if until now the sequences are correct, otherwise 0.
- lut_seq: Lookup table that contains the correct sequence.
- SEQNUMBER: 3-bit counter that drives the input of lut_seq.
- COUNT_WRONG:2-bit counter that counts the number of consecutive wrong inputs.

State Diagram

Picture 2: SSR State diagram



3 Code

Secure Sequence Recognizer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity SecureSequenceRec is
port (
        clock:  in std_logic; --- clock signal
        reset:   in std_logic; -- reset signal active low
        first:    in std_logic; -- start sampling signal
        num_in:         in std_logic_vector(7 downto 0); --input sequence number

        unlock:          out std_logic; -- signal of success
        warning: out std_logic  -- error signal

);
end SecureSequenceRec;

architecture SSR_beh of SecureSequenceRec is
        SUBTYPE STATE_TYPE_COUNT is STD_LOGIC_VECTOR (1 DOWNTO 0) ;
        CONSTANT SMEM: STATE_TYPE_COUNT  :="00"; -- when in this state the counter just
keeps the exit constant
        CONSTANT SINC: STATE_TYPE_COUNT  :="01"; -- when in this state the counter
increments the exit
        CONSTANT SRES: STATE_TYPE_COUNT  :="11"; -- when in this state the counter
keeps the exit equals to zero


        SUBTYPE STATE_TYPE is STD_LOGIC_VECTOR (2 DOWNTO 0) ;
        -- constants to drive the state
        CONSTANT SINIT: STATE_TYPE:="000";
        CONSTANT S0: STATE_TYPE:="001";
        CONSTANT S1: STATE_TYPE:="010";
        CONSTANT S2: STATE_TYPE:="011";
        CONSTANT S3: STATE_TYPE:="100";
        CONSTANT S4: STATE_TYPE:="101";
        CONSTANT S5: STATE_TYPE:="110";
        CONSTANT SBLOCK: STATE_TYPE:="111";


        signal signal_star_in  : STATE_TYPE;
        signal signal_star_out : STATE_TYPE;

        signal lut_in  : std_logic_vector(2 downto 0);
        signal lut_out : std_logic_vector(7 downto 0);

        signal signal_ok_in  : std_logic_vector(0 downto 0);
        signal signal_ok_out : std_logic_vector(0 downto 0);-- 0 when a wrong number is inserted
```

```
signal count_wr_in  : STATE_TYPE_COUNT; -- count the wrong sequences
signal count_wr_out : std_logic_vector(1 downto 0);

signal counter_state_in : STATE_TYPE_COUNT;

component DFF is
        generic(N_bit : integer);
        port (
                clk: in std_logic;
                reset: in std_logic;
                D: in std_logic_vector(N_bit-1 downto 0);
                Q: out std_logic_vector(N_bit-1 downto 0)
        );
end component;

component LUT is

        port(
                address : in STD_LOGIC_VECTOR(2 downto 0);
                data : out STD_LOGIC_VECTOR(7 downto 0)
        );
end component;

component Counter_mod is
generic(N_bitc : integer);
 port(
         count_clk      : in     STD_LOGIC;
         count_reset    : in     STD_LOGIC;
         enabler_in     : in     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         D_out                   : out    STD_LOGIC_VECTOR(N_bitc-1 downto 0)
    );

end component;

begin

OK: DFF --1 if the seq number is correct,0 otherwise
generic map(N_bit => 1)
port map(clock,reset,signal_ok_in,signal_ok_out);

STAR: DFF --status register
generic map(N_bit => 3)
port map(clock,reset,signal_star_in,signal_star_out);

SEQNUMBER: counter_mod --Drives the lut
generic map(N_bitc => 3)
port map(clock,reset,counter_state_in,lut_in);

COUNT_WRONG: counter_mod --Keeps track of the wrong sequences
generic map(N_bitc =>2)
port map(clock,reset,count_wr_in,count_wr_out);
```

```
        lut_seq : LUT --contains the sequence numbers
        port map(lut_in,lut_out);


        LOGIC:process(signal_star_out,reset,first)
        begin
                if(reset='0') then
                        signal_star_in<=SINIT;
                        unlock<='0';
                        warning<='0';
                        signal_ok_in<="0";
                        counter_state_in<=SRES;
                        count_wr_in<=SRES;
                else
                        case(signal_star_out) is
                                when SINIT=> -- the SSR  in this state waits for the first sequence
number
                                        unlock<='0';

                                        if(first='1' ) then
                                                signal_star_in<=S0;
                                                counter_state_in<=SINC;      --start counting

                                            else
                                                signal_star_in<=SINIT;
                                                counter_state_in<=SMEM;
                                        end if;
                                        warning<='0';
                                        count_wr_in<=SMEM;
                                        signal_ok_in<="0";
                                when S0 =>
                                        count_wr_in<=SMEM;
                                        unlock<='0';
                                        warning<='0';
                                        signal_star_in<=S1;

                                        if(num_in = lut_out ) then --Checking Sequence number 0
                                                signal_ok_in<="1";
                                        else
                                                signal_ok_in<=signal_ok_out;
                                        end if;

                                        counter_state_in<= SINC;
                                when S1 =>
                                        unlock<='0';
                                        warning<='0';
                                        count_wr_in<=SMEM;
                                        if(first='1')then --From now on it is an error if first goes to '1'
and the device is being blocked
                                                signal_ok_in<="0";
                                                signal_star_in<=SBLOCK;
                                        else
```

```
                        if(num_in = lut_out)then --Checking Sequence number
1
                                signal_ok_in<=signal_ok_out;
                        else
                                signal_ok_in<="0";
                        end if;
                        signal_star_in<=S2;
                end if;
                counter_state_in<= SINC;
        when S2=>
                unlock<='0';
                warning<='0';
                count_wr_in<=SMEM;
                if(first='1')then
                        signal_ok_in<="0";
                        signal_star_in<=SBLOCK;
                else
                        if(num_in = lut_out)then --Checking Sequence number
2
                                signal_ok_in<=signal_ok_out;
                        else
                                signal_ok_in<="0";
                        end if;
                        signal_star_in<=S3;
                end if;
                counter_state_in<= SINC;
        when S3=>
                unlock<='0';
                warning<='0';
                count_wr_in<=SMEM;
                if(first='1')then
                        signal_ok_in<="0";
                        signal_star_in<=SBLOCK;
                else
                        if(num_in = lut_out)then --Checking Sequence number
3
                                signal_ok_in<=signal_ok_out;
                        else
                                signal_ok_in<="0";
                        end if;

                        signal_star_in<=S4;
                end if;
                counter_state_in<= SMEM;
        when S4=>
                unlock<='0';
                warning<='0';
                count_wr_in<=SMEM;
                if(first='1')then
                        signal_ok_in<="0";
                        signal_star_in<=SBLOCK;
                else
```

```
                                    if(num_in = lut_out)then --Checking Sequence number
4
                                        signal_ok_in<=signal_ok_out;
                                    else
                                        signal_ok_in<="0";
                                    end if;
                                    signal_star_in<=S5;
                                end if;
                                counter_state_in<= SRES;
                            when S5 =>
                                unlock<= signal_ok_out(0);
                                warning <= not signal_ok_out(0);
                                if(signal_ok_out(0)='1') then
                                    count_wr_in<=SRES;
                                else
                                    count_wr_in<=SINC;
                                end if;
                                counter_state_in<= SRES;
                                if(first='1' or (signal_ok_out(0)='0' and count_wr_out="10" ))
then
                                    signal_star_in<=SBLOCK;
                                else
                                    signal_star_in<=SINIT;
                                end if;
                                signal_ok_in<="0";
                            when SBLOCK=>
                                unlock<='0';
                                warning<='1';
                                signal_star_in<=SBLOCK; --Blocking the SSR until a reset
event happens
                                counter_state_in<= SRES;
                                count_wr_in<=SRES;
                                signal_ok_in<="0";

                            when others =>  -- avoiding unexpected behaviour due to bad driving
of the inner state
                               unlock<='0';
                                warning<='1';
                                signal_star_in<=SBLOCK; --Blocking the SSR until a reset
event happens
                                counter_state_in<= SRES;
                                count_wr_in<=SRES;
                                signal_ok_in<="0";

                        end case;
                end if;
        end process;

end SSR_beh;
```

Counter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Counter_mod is
        generic(N_bitc : integer);
         port(
                count_clk      : in     STD_LOGIC;
                count_reset    : in     STD_LOGIC;
                enabler_in     : in     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                D_out                   : out    STD_LOGIC_VECTOR(N_bitc-1 downto 0)
            );
end Counter_mod;

architecture Counter_beh of Counter_mod is

        SUBTYPE STATE_TYPE is STD_LOGIC_VECTOR (1 DOWNTO 0) ;
        CONSTANT SMEM: STATE_TYPE  :="00"; -- when in this state the counter just keep the
exit constant
        CONSTANT SINC: STATE_TYPE  :="01"; -- when in this state the counter increment the
exit
        CONSTANT SRES: STATE_TYPE  :="11"; -- when in this state the counter reset the exit

        signal s_RPCA          :std_logic_vector(N_bitc-1 downto 0);
        signal retro_add :std_logic_vector(N_bitc-1 downto 0);
        signal dff_in    :std_logic_vector(N_bitc-1 downto 0);
        signal add_in    :std_logic_vector(N_bitc-1 downto 0);

        signal cout_RPCA :std_logic;
        component RPCA is
                generic(Nbit : integer);
                port(
                        a       : in  std_logic_vector(Nbit-1 downto 0);
                        b       : in  std_logic_vector(Nbit-1 downto 0);
                        cin     : in  std_logic;
                        s       : out std_logic_vector(Nbit-1 downto 0);
                        cout    : out std_logic
                );
        end component;

        component DFF is
                generic(N_bit : integer);
                port (
                        clk: in std_logic;
                        reset: in std_logic;
                        D: in std_logic_vector(N_bit-1 downto 0);
                        Q: out std_logic_vector(N_bit-1 downto 0)
                );
        end component;

begin
```

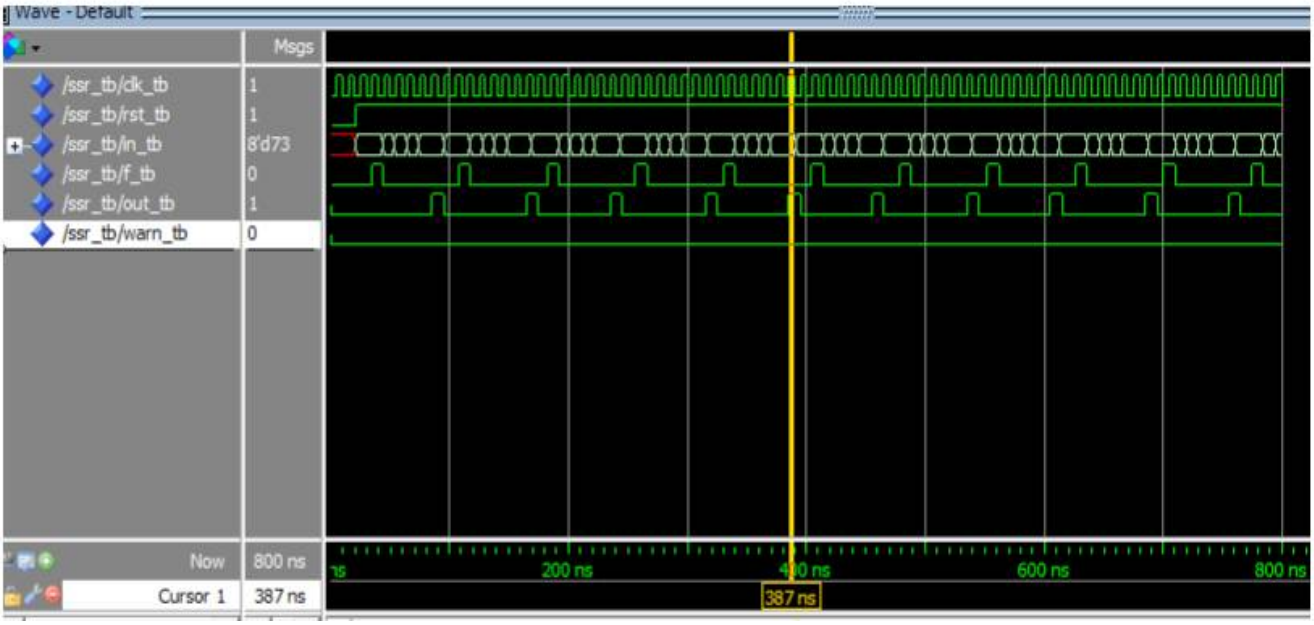
```
        rpca1   : RPCA
        generic map(Nbit => N_bitc)
        port map(add_in,retro_add,'0',s_RPCA,cout_RPCA);

        dff1    : DFF
        generic map(N_bit => N_bitc)
        port map(count_clk,count_reset,dff_in,retro_add);
        counting_proc: process(enabler_in,s_RPCA,count_reset,retro_add)
        begin
                add_in<= (N_bitc-1 downto 1 => '0',others =>'1');
                if(count_reset='0') then
                        dff_in<= (others => '0');
                else
                        case(enabler_in) is
                        when SMEM=>
                                dff_in<=retro_add;
                        when SINC=>
                                dff_in<=s_RPCA;
                        when SRES=>
                                dff_in <= (others => '0');
                        when others =>
                           dff_in <= (others => '0');
                        end case;
                end if;
                D_out<=retro_add;
        end process;
end Counter_beh;
```

4 Test_benches

Correct sequence Testbench

```
library IEEE;
use IEEE.std_logic_1164.all;

entity SSR_tb is
end SSR_tb;


architecture SSR_tb_beh of SSR_tb is

        constant T_CLK   : time := 10 ns; -- Clock period
        constant T_RESET : time := 21 ns; -- Period before the reset deassertion
        constant T_FIRST : time := 13 ns; -- Period before the reset deassertion

        signal clk_tb  : std_logic := '0'; -- clock signal, intialized to '0'
        signal rst_tb  : std_logic := '0'; -- reset signal
        signal in_tb   : std_logic_vector(7 downto 0) ;
        signal f_tb    : std_logic :='0';
        signal out_tb  : std_logic;
        signal warn_tb         : std_logic;

        component SecureSequenceRec is
                port (
                        clock:  in std_logic; --- clock signal
                        reset:   in std_logic; -- reset signal active low
                        first:    in std_logic; -- start sampling signal
                        num_in:         in std_logic_vector(7 downto 0); --input sequence number

                        unlock:          out std_logic; -- signal of success
                        warning: out std_logic -- error signal

                );
        end component;
        begin

        clk_tb <= not(clk_tb) after T_CLK / 2;
        rst_tb <= '1' after T_RESET; -- Deasserting the reset after T_RESET nanosecods.

        SSR_test: SecureSequenceRec
        port map(clk_tb,rst_tb,f_tb,in_tb,out_tb,warn_tb);

        r_process:process
        begin
                wait for T_RESET;
                in_tb<="00100100";
                wait for T_FIRST;
                f_tb<='1';
                wait for T_CLK;
                f_tb<='0';
```

```
        in_tb<="00010011";
        wait for T_CLK;
        in_tb<="00111000";
        wait for T_CLK;
        in_tb<="01100101";
        wait for T_CLK;
        in_tb<="01001001";

    end process;

end SSR_tb_beh;
```



Picture 3: Simulation of a correct Sequence

Wrong sequence Testbench

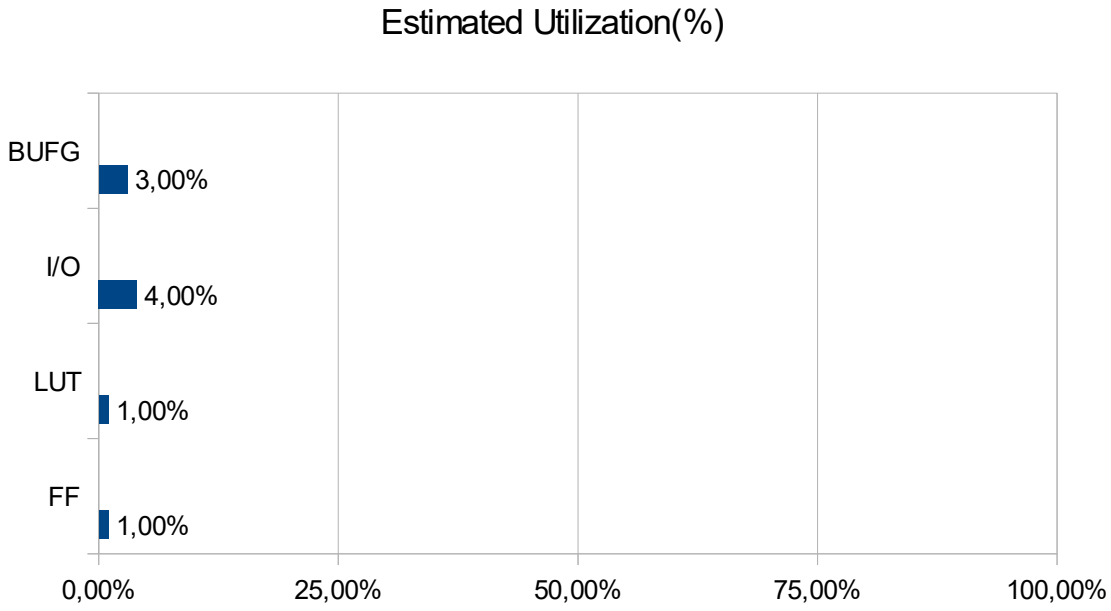
```
library IEEE;
use IEEE.std_logic_1164.all;

entity SSR_tb is
end SSR_tb;


architecture SSR_tb_beh of SSR_tb is

        constant T_CLK   : time := 10 ns; -- Clock period
        constant T_RESET : time := 21 ns; -- Period before the reset deassertion
        constant T_FIRST : time := 13 ns; -- Period before the reset deassertion

        signal clk_tb  : std_logic := '0'; -- clock signal, intialized to '0'
        signal rst_tb  : std_logic := '0'; -- reset signal
        signal in_tb   : std_logic_vector(7 downto 0) ;
        signal f_tb    : std_logic :='0';
        signal out_tb  : std_logic;
        signal warn_tb         : std_logic;

        component SecureSequenceRec is
                port (
                        clock:  in std_logic; --- clock signal
                        reset:   in std_logic; -- reset signal active low
                        first:   in std_logic; -- start sampling signal
                        num_in:         in std_logic_vector(7 downto 0); --input sequence number

                        unlock:          out std_logic; -- signal of success
                        warning: out std_logic -- error signal

                );
        end component;
        begin

        clk_tb <= not(clk_tb) after T_CLK / 2;
        rst_tb <= '1' after T_RESET; -- Deasserting the reset after T_RESET nanosecods.

        SSR_test: SecureSequenceRec
        port map(clk_tb,rst_tb,f_tb,in_tb,out_tb,warn_tb);

        r_process:process
        begin
                wait for T_RESET;
                in_tb<="00100100";
                wait for T_FIRST;
                f_tb<='1';
                wait for T_CLK;
                f_tb<='0';
                in_tb<="10010011";
                wait for T_CLK;
                in_tb<="00111000";
```

```
            wait for T_CLK;
            in_tb<="01100101";
            wait for T_CLK;
            in_tb<="01001001";

        end process;

end SSR_tb_beh;
```

Picture 4: Simulation of a wrong Sequence

5 Synthesis on Vivado

Resource utilization



Critical path

Design timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack	8,333 ns	Worst Hold Slack	0,208 ns	Worst Pulse Width	4,650 ns
Total Negative Slack	0,000 ns	Total Hold Slack	0,000 ns	Total PWS	0,000 ns
Clock Frequency	100 MHz				

All user specified timing constraints are met

Warnings

[Synth 8-614] signal 'num_in' is read in the process but is not in the sensitivity list
The sampling must be synchronous with the clock,so just once every clock cycle. Putting the signal in the sensitivity list would have meant corrupting the integrity of the sampling.

[Synth 8-614] signal 'signal_ok_out' is read in the process but is not in the sensitivity list
[Synth 8-614] signal 'count_wr_out' is read in the process but is not in the sensitivity list
[Synth 8-614] signal 'lut_out' is read in the process but is not in the sensitivity list

Those are inner signal used by the SSR ,it would be uselless insert them in the sensitivity list