FPGA debugging using the MSO-19

Link Instruments, Inc.
17A Daniel Rd East, Fairfield, NJ 07004 973-808-8990 fax: 306-3000
Internet: Sales@LinkInstruments.com http://www.LinkInstruments.com

The MSO-19 is a very powerful design tool. It is a combination of an Oscilloscope, Logic Analyzer and Pattern Generator with a simple Windows software interface. The goal of this exercise is to demonstrate how to streamline the FPGA design process by using the MSO-19.

FPGA based evaluation boards are invaluable tools in the initial stages of product development. Ideas can be quickly tried out before committing to a PCB layout. Software simulation can assist in the design process, but if the design involves other ICs that are connected the FPGA the task becomes more complicated. One of best method to assist in design is to use an I/O analyzer. What is an I/O analyzer? It is basically a combination of a Logic Analyzer and a Pattern Generator. A typical FPGA design contains the Input ports, Output ports, Clocks, IP cores and glue logic to tie them together. When a design fails to work, the ability to see what is going on inside the FPGA becomes critical. Three of the methods to debug the design are:

1) FPGA manufactures have IP core based logic analyzers that can be compiled into the design for debugging purposes. These soft Logic Analyzers are usually costly and consume valuable resources in a smaller FPGA.
2) By bonding out all of the signals of interest to external I/O pins an external Logic Analyzer can be used to monitor the signals. This method can be a problem if the design does not have enough spare I/O pins.
3) Sometimes JTAG ports can be used to monitor internal register states. Unfortunately user interface software is generally lacking and data update rates are limited.
4) Another method is to write simple debugging code to compact data into serial data streams. External stimulus can also be applied via this method. This method combines the I/O savings of the JTAG method and the simplicity of an external Logic Analyzer.

We will be demonstrating method 4 using the MSO-19 Logic Analyzer and Pattern Generator.

For our exercises, we will be using the Lattice MachXO Starter Evaluation board. Our goal is to create a software loadable timer via a SPI port before we've selected a MCU (microcontroller) for our project. The MachXO starter board contains 9 LEDs, a bank of DIP switches, a 33.33Mhz oscillator and a slew of bond out pads to test our design. Only 4 pins on the MSO-19 will be use simulate a simple SPI I/O port.

1. First order of business for a hardware designer is turn on the LED, it's our way of saying "Hello World". The LEDs showed a pattern of 010101010.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Lights is

port(
  LED: out std_logic_vector ( 8 downto 0 );
  DPSw: in std_logic_vector ( 7 downto 0 );
  SW2: in std_logic ;
  SW3: in std_logic ;
  XCLK: in std_logic ;
  SCE: in std_logic ;
  SCK: in std_logic ;
  SDI: in std_logic ;
  SDO: in std_logic  );

end;

architecture a of Lights is

signal TLED: std_logic_vector(8 downto 0);

begin

TLED <= "010101010";

LED <= TLED;

end a;
```

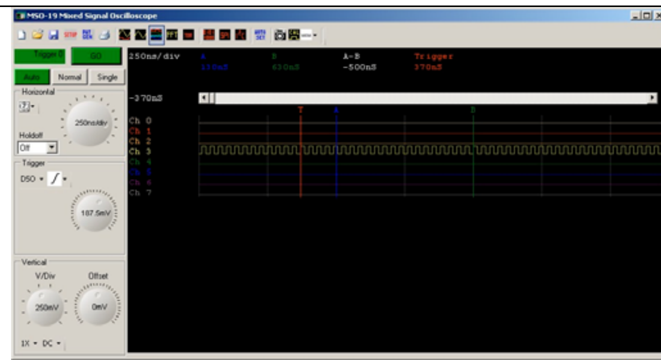
2. Once we have access to output pins, it's time to work on the input pins. Let's wire up the board so if that the SW2 is pressed the value of DIP switches are displayed on the LEDs otherwise the LED will display the pattern of 010101010.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;


entity Lights is

port(
  LED: out std_logic_vector ( 8 downto 0 );
  DPSw: in std_logic_vector ( 7 downto 0 );
  SW2: in std_logic ;
  SW3: in std_logic ;
  XCLK: in std_logic ;
  SCE: in std_logic ;
  SCK: in std_logic ;
  SDI: in std_logic ;
  SDO: in std_logic  );

end;

architecture A of Lights is

signal TLED: std_logic_vector(8 downto 0);

begin

  LedSel: process(SW2) begin
    if(SW2 = '0') then
      TLED <= DPSw & '0';
    else
      TLED <= "010101010";
    end if;
  end process;

  LED <= TLED;

end A;
```

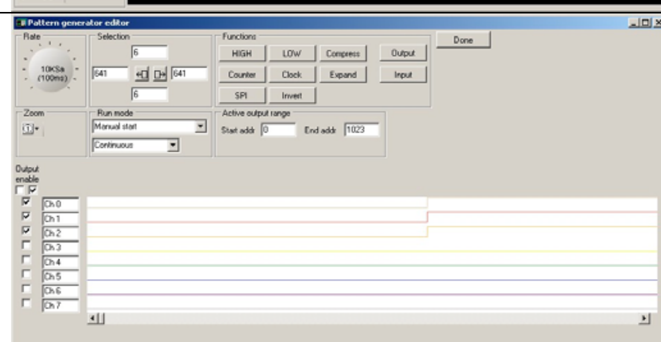
3. Next we'll connect 4 digital I/O pins from MSO-19 to the starter board. Create a simple bypass circuit to send the outputs on the MSO-19 pins to the LEDs. We'll also feed the oscillator clock back to pin 4 of the I/O pins.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;


entity Lights is

port(
   LED: out std_logic_vector ( 8 downto 0 );
   DPSw: in std_logic_vector ( 7 downto 0 );
   SW2: in std_logic ;
   SW3: in std_logic ;
   XCLK: in std_logic ;
   SCE: in std_logic ;
   SCK: in std_logic ;
   SDI: in std_logic ;
   SDO: out std_logic  );

end;

architecture A of Lights is

signal TLED: std_logic_vector(8 downto 0);
signal TSW: std_logic;

begin


TSW <= DPsw(0) and DPsw(1) and DPsw(2) and DPsw(3)
    and DPsw(4) and DPsw(5) and DPsw(6) and DPsw(7)
    and SW2 and SW3;
-- dummy statament

LED(8) <= not TSW;

LED(0) <= SCE;
LED(1) <= SCK;
LED(2) <= SDI;
SDO <= XCLK;
LED(7 downto 3) <= "11111";

end A;
```

Logic Analyzer channel 3 shows the 33 MHz clock.



The MSO-19's pattern generator clock is independent from the logic analyzer clock. We can run the LA section at 200MSa/s to see the 33Mhz oscillator while running the PG at 10KSa/S, slow enough to see the LEDs blink.



4. It’s time to create an internal shift register so that we can control the LEDs from the MSO-19. The easiest format is to follow the SPI protocol. We’ll create an internal version of the SCE, SCK and SDI.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity Lights is
port(
  LED: out std_logic_vector ( 8 downto 0 );
  DPSw: in std_logic_vector ( 7 downto 0 );
  SW2: in std_logic ;
  SW3: in std_logic ;
  XCLK: in std_logic ;
  SCE: in std_logic ;
  SCK: in std_logic ;
  SDI: in std_logic ;
  SDO: out std_logic  );
end;

architecture A of Lights is
signal TLED: std_logic_vector(7 downto 0);
signal TSW: std_logic;
signal LEDBuf:  std_logic_vector(7 downto 0);
signal ISCE: std_logic;
signal ISCK: std_logic;
signal ISDI:  std_logic;
```

All inbound signal edges will be synchronized to the 33Mhz oscillator.

```
RegSCE:process(XCLK) begin
  if(rising_edge(XCLK)) then
    ISCE <= SCE;
  end if;
end process;

RegSCK:process(XCLK) begin
  if(rising_edge(XCLK)) then
    ISCK <= SCK;
  end if;
end process;

RegSDI:process(XCLK) begin
  if(rising_edge(XCLK)) then
    ISDI <= SDI;
  end if;
end process;
-- synchronize the inbound signals
```

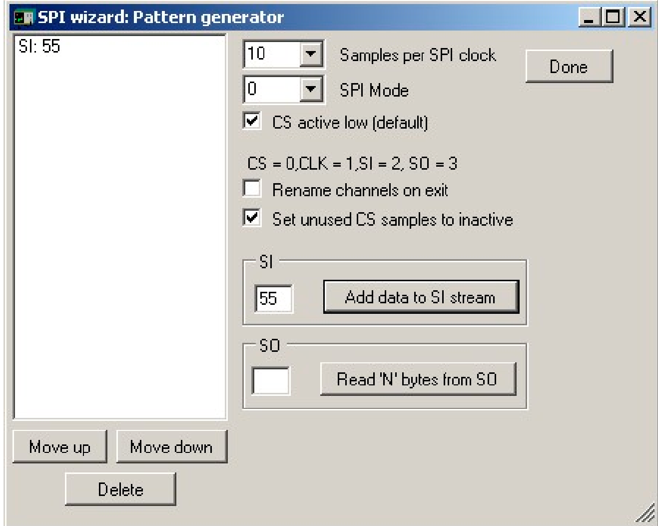
All the unused signals are AND together into a dummy statement.

```
TSW <= DPsw(0) and DPsw(1) and DPsw(2) and DPsw(3)
   and DPsw(4) and DPsw(5) and DPsw(6) and DPsw(7)
   and SW2;

LED(8) <= not TSW;
-- dummy statament
```

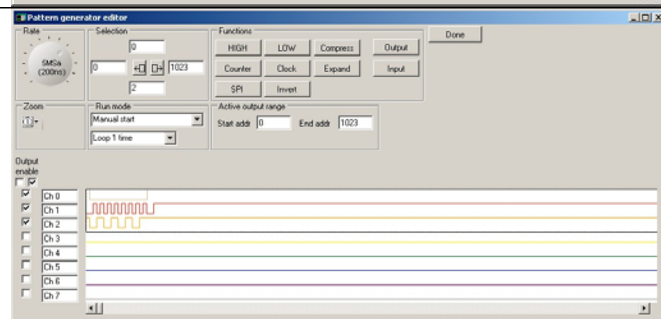
ISCE is the enabler signal for the shift register, which clocks in the ISDI data using the rising edge of ISCK. At the end of the ISCE cycle, we use the rising edge of the ISCE to transfer the 8 bit data in the shift register to the LED register.

```
RegLedBuf:process(ISCE,ISCK) begin
  if(rising_edge(ISCK) and (ISCE = '0')) then
    LEDBuf(7 downto 0) <=  LEDBuf(6 downto 0) & ISDI;
  end if;
end process;
-- shifts SDI into the register

RegBufTxf:process(ISCE) begin
  if(rising_edge(ISCE)) then
    TLED <= LEDBuf;
  end if;
end process;
-- copies the SPI shift register data to the LED buffer

SDO <= XCLK;
LED(7 downto 0) <= not(TLED);
end A;
```

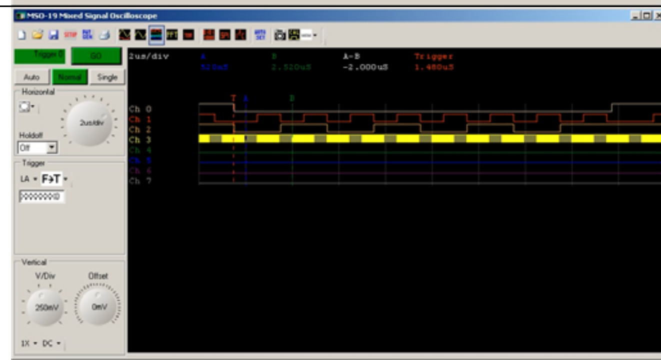
We can generate the necessary SPI control signals using the SPI generator feature of the MSO-19's pattern generator



The PG buffer now contains the data we want to send to the FPGA



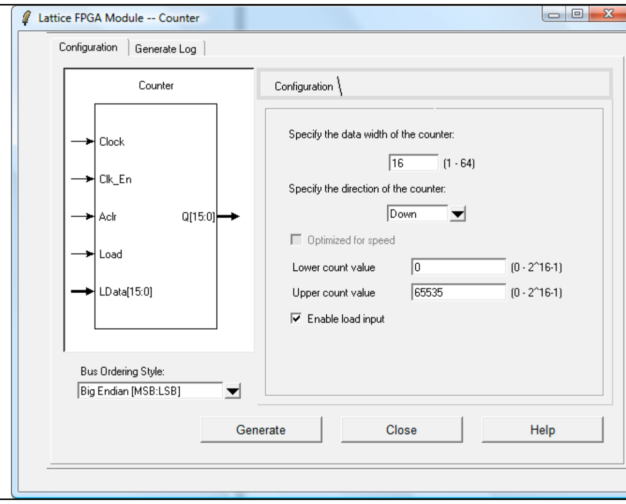
Running the PG and LA allows us to see result on the LEDs and the LA screen.



5. Next we'll expand the SPI buffer to 16 bits. We'll also wire in SW3 as a selector so we can examine the content of the upper and lower byte of the 16 bit register on the LED. Just like the above example, create two byte data using the SPI generator in the PG setup screen. By lengthening the SPI/IN data buffer we can increase the number of FPGA registers that we want to control.

```
signal TLED: std_logic_vector(7 downto 0);
signal TSW: std_logic;
signal SPIBuf:  std_logic_vector(15 downto 0);
signal ISCE: std_logic;
signal ISCK: std_logic;
signal ISDI:  std_logic;
signal Tcnt:  std_logic_vector(15 downto 0);
signal Fcnt:  std_logic_vector(15 downto 0);

begin
RegLedBuf:process(ISCE,ISCK) begin
  if(rising_edge(ISCK) and (ISCE = '0')) then
    SPIBuf(15 downto 0) <=  SPIBuf(14 downto 0) & ISDI;
  end if;
end process;
-- shifts SDI into the register

RegBufTxf:process(ISCE) begin
  if(rising_edge(ISCE)) then
    Fcnt <= SPIBuf;
  end if;
end process;
-- copies the SPI shift register data to the LED buffer

LedSel:process(SW3) begin
  if(SW3 = '0') then
    TLED <= Fcnt(7 downto 0);
  else
    TLED <= Fcnt(15 downto 8);
  end if;
end process;

LED(7 downto 0) <= not(TLED(7 downto 0));
```

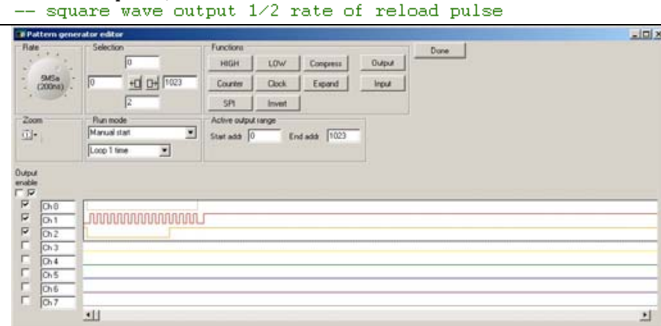
6. On to the main event, we are going to create a reloadable 16 bit timer that we can control via the MSO-19. First create a 16bit loadable down counter via IPexpress.



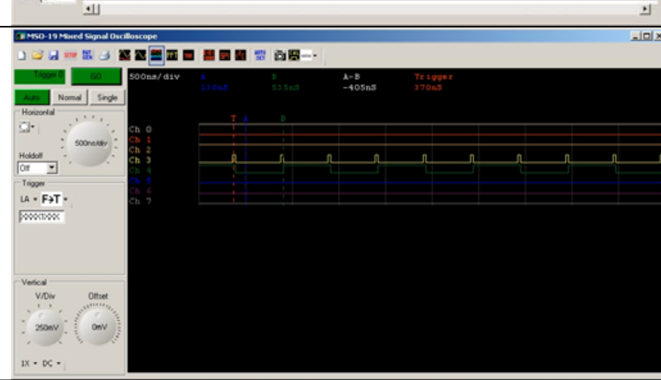
Next create a 16 input AND gate to trap the 0x0000 condition. We'll also create a toggle div/2 register that operates off the zero detect. The zero detect signal serves two purposes, the load signal for the down counter and the toggle enable for the toggle output.

```
ZeroP <= not(Tcnt(15)) and not(Tcnt(14)) and not(Tcnt(13))
   and not(Tcnt(12)) and not(Tcnt(11)) and not(Tcnt(10))
   and not(Tcnt(9)) and not(Tcnt(8)) and not(Tcnt(7))
   and not(Tcnt(6)) and not(Tcnt(5)) and not(Tcnt(4))
   and not(Tcnt(3)) and not(Tcnt(2)) and not(Tcnt(1))
   and not(Tcnt(0));


RegZero:process(XCLK) begin
  if(rising_edge(XCLK)) then
    ZeroQ <= ZeroP;
  end if;
end process;

SDO <= ZeroQ;
-- end of count, reload pulse

RegSqw:process(XCLK,ZeroQ) begin
  if(rising_edge(XCLK) and ZeroQ = '1') then
    if(Sqwave = '0') then
      Sqwave <= '1';
    else
      Sqwave <= '0';
    end if;
  end if;
end process;

Tout <= Sqwave;
-- square wave output 1/2 rate of reload pulse
```

Create the count of interest using the SPI generator.



We can see the SDO pin pulsing as the 16 bit counter resets and reload the SPI count. The Tout pin also shows a square wave that toggles at ½ the rate of SDO.



But how do you read the registers back from the FPGA? We can use the same SCE signal to latch the internal register to a SPI out data buffer. And clock the data out on SCK via the SDO pin while we transfer data into the SDI pin. We will use the falling edge of the SCE signal. Reading from a SPI port is a little trickier to do than writing into the SPI port. We need to detect the falling edge of SCE. This is accomplished by creating a 2 stage pipeline, compare the current and previous state of SCE, once the comparator detects a H > L transition, it will generate a pulse of one oscillator clock wide. This signal CEQQ will be the signal that will use to transfer the data from internal register to the SDO buffer.

```
RegSCE:process(XCLK) begin
  if(rising_edge(XCLK)) then
    CEP <= ISCE;
  end if;
end process;

CEQ <= CEP and not(ISCE);

RegSCE:process(XCLK) begin
  if(rising_edge(XCLK)) then
    CEQQ <= CEQ;
  end if;
end process;
---- CE low edge detector
```

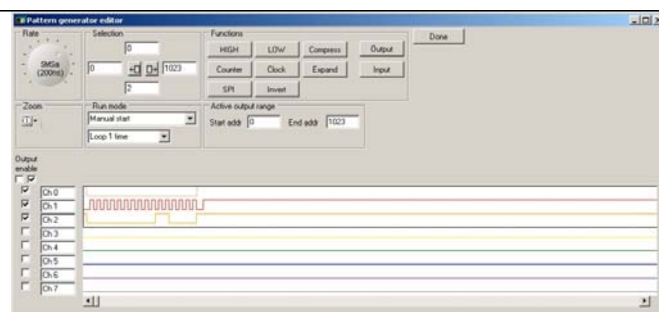
SCK signal will also need an edge detector. CKQQ is our pulsed clock.

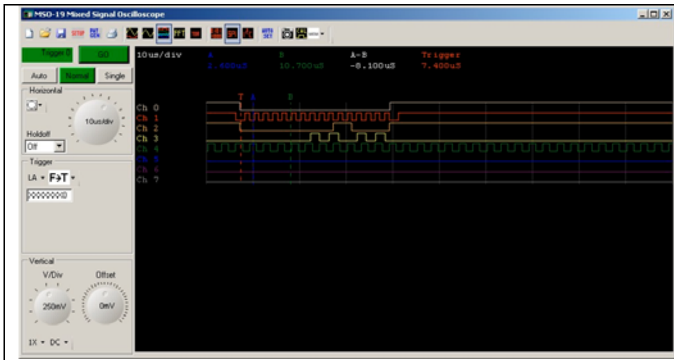
```
RegSCK:process(XCLK) begin
  if(rising_edge(XCLK)) then
    CKP <= ISCK;
  end if;
end process;

CKQ <= not(CKP) and ISCK;

RegSCK:process(XCLK) begin
  if(rising_edge(XCLK)) then
    CKQQ <= CKQ;
end if;
end process;

---- CK high edge detector
```

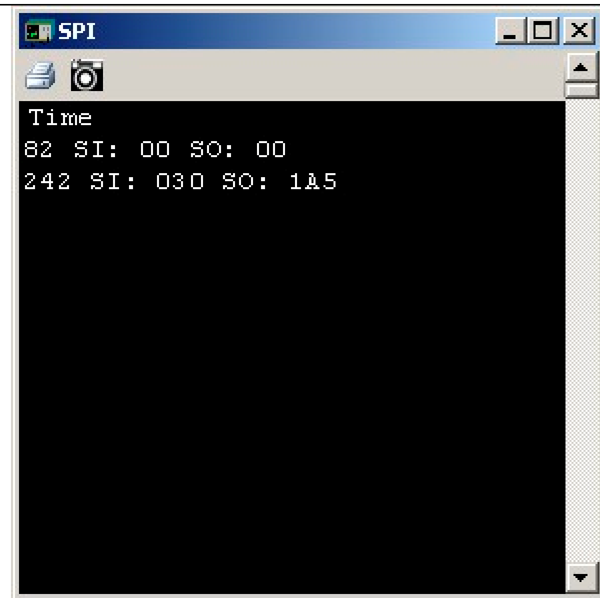
The CKQQ signal combined with the ISCE signal becomes the enabler signal as the shift register clocks data out via SDO.

```
RegOutBufTxf:process(ISCE) begin

  if(rising_edge(XCLK)) then
    if (CEQQ = '1') then
      SPIOutBuf <= "10100101" & DPsw;
    elsif (CKQQ = '1' and ISCE = '0') then
      SPIOutBuf(15 downto 0) <= '0' & SPIOutBuf(15 downto 1);
    end if;
  end if;
end process;

SDO <= SPIOutBuf(0);
-- copy register to SPIOutBuf on falling edge of
-- SCE and clock out on rising edge of SCK
```

In this example we've shown that we can transfer the Dip Switch settings to the SDO buffer and clock it out while we clock in the new timer value. One can confirm this in the SPI display box.







As one can see from the above examples, the I/O analyzer function on the MSO-19 can be a very power tool in debugging complex FPGA designs. This technique can be expanded to further control and debug internal FSM and external circuitry attached to the FPGA. With the MSO-19, one can create a very affordable comprehensive digital lab to explore the flexibility of FPGA designs.