Digital Engineering
Project Task 2

Y3890959
Y3878784

28th February 2023

Task 2: Clock domain crossing using dual-clock FIFOs

1 - Description of FSMs

SOURCE_CTRL FSM

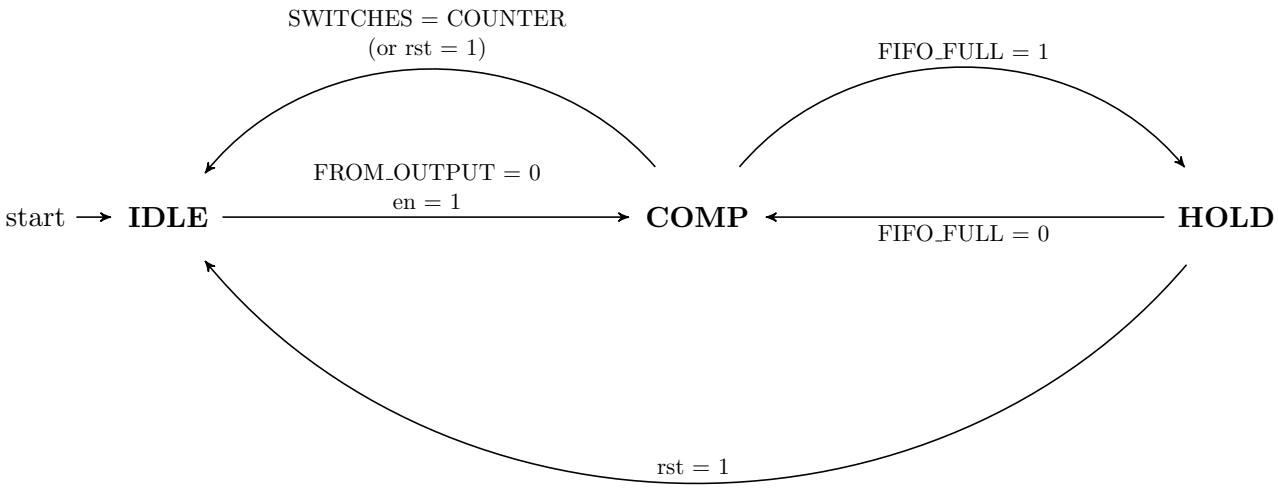


Figure 1: SOURCE_CTRL FSM state graph

The FSM starts at IDLE state, where all internal signals will reset and the LEDs are off. When the 'en' pushbutton is pressed and the output logic FSM has completed (FROM_SOURCE goes low) the state goes to COMP where the outputs are computed. As soon as 'en' is toggled, when the FSM is still in IDLE, the values of the SWITCHES is stored in a register 'LIMT_REG'. In the COMP state, the 'LIMT_CNT' is enabled and counts, 'EN_SOURCE' also goes high so the outputs are computed, and the 'FIFO_WR_EN' goes high so the values are stored. When the FIFO becomes full, the SOURCE, counter, and FIFO_WR_EN are disabled, and the FSM goes to HOLD state, where it stays until FIFO_FULL becomes low again, then it goes back to COMP. The counter will increment in the COMP state for every output computed and stored, once the counter value reaches the value it signifies that the total number of outputs as set by the user via the SWITCHES has been computed, in this case the FSM returns to the IDLE state. To make sure that the circuit doesn't break when the user changes the SWITCHES mid-operation, the values of the SWITCHES is stored in 'LIMT_REG', and the logic will compare this.

STATE	***EN_SOURCE***	***LIMT_CNT_EN***	***LIMT_CNT_RST***	***FIFO_WR_EN***
IDLE	1 when en = 1 and FROM_OUTPUT = 0	0	1	0
	0 otherwise			
COMP	1 when FIFO_FULL = 0	1 when FIFO_FULL = 0	0	1 when FIFO_FULL = 0
	0 otherwise	0 otherwise		0 otherwise
HOLD	0	0	0	0

Table 1: SOURCE_CTRL FSM table of outputs

OUTPUT_CTRL FSM

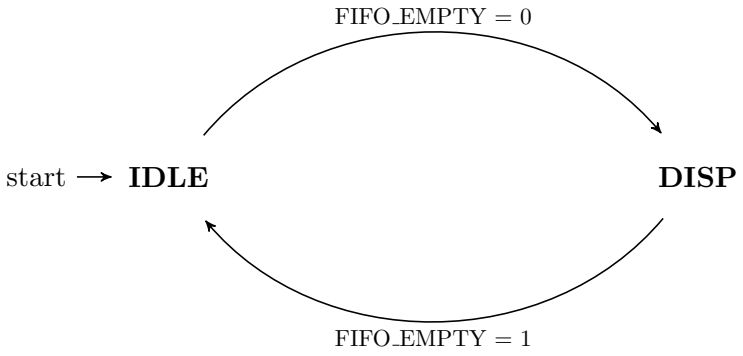


Figure 2: OUTPUT_CTRL FSM state graph

This FSM is quite simple and efficient with only two states, it completely ignores any user inputs via switches and pushbuttons and only relies on whether the FIFO is empty or not. In the IDLE state, the delay counter is disabled and reset and a 0 is output to 'TO_SOURCE', this is to tell the SOURCE logic that the output logic is inactive. Once 'FIFO_EMPTY' goes low, the FSM goes to the DISP state. In the DISP state, the 'DISP_CNT' counter is enabled (this is to add a delay where the LED outputs are held). As the FIFO is set to the First-Word-Fall-Through mode, the 'FIFO_WR_EN' needs to be high to pop the value at the end of the delay, therefore that signal goes high as soon as the delay counter reaches the last value. The delay counter will roll-over automatically. The LEDs will output the 'DATA_FROM_FIFO' vector as long as the FIFO is in DISP state.

STATE	***DISP_CNT_EN***	***DISP_CNT_RST***	***TO_SOURCE***	***FIFO_RD_EN***	***LEDs***
IDLE	0	1	0	0	00
DISP	1	0	1	1 when DISP_CNT_OUT = disp_delay-1 0 otherwise	DATA_FROM_FIFO

Table 2: OUTPUT_CTRL FSM table of outputs

2 - VHDL Entity Code

OUTPUT_CTRL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.all;


entity SOURCE_CTRL is
    -- Max value that the board switches can represent stored as a
    -- Generic.
    Generic (switch_limit : natural := 256);
    Port ( CLK : in  STD_LOGIC;
           USER_PB : in  STD_LOGIC_VECTOR (4 downto 0);
           SWITCHES : in  STD_LOGIC_VECTOR (7 downto 0);
           FIFO_FULL : in  STD_LOGIC;
           FIFO_WR_EN : out  STD_LOGIC;
           TO_OUTPUT : out  STD_LOGIC;
           FROM_OUTPUT : in  STD_LOGIC;
           RST_SOURCE : out  STD_LOGIC;
           EN_SOURCE : out  STD_LOGIC
    );
end SOURCE_CTRL;

architecture Behavioral of SOURCE_CTRL is

signal EN, RST : STD_LOGIC;
type state_type is (IDLE, COMP, HOLD);  -- The FSM states
signal state, next_state: state_type;   -- The states as signals

-- Internal signal for the board switch limit counter.
--  This counter limits the maximum number of outputs based on
--  the switches on the board. There are 8 switches on the board,
--  so a maximum of 256 values can be represented.
signal LIMT_CNT_OUT : UNSIGNED (log2(switch_limit)-1 downto 0);
signal LIMT_CNT_EN, LIMT_CNT_RST : STD_LOGIC;
-- To ignore any changes in the SWITCHES we need to store the value
-- of switches in an internal signal when the enablle button is
-- pressed during the IDLE state.
signal LIMT_REG_EN : STD_LOGIC;
signal LIMT_REG_RST : STD_LOGIC;
signal LIMT_REG_OUT : STD_LOGIC_VECTOR (7 downto 0);

begin

EN <= USER_PB(3);
RST <= USER_PB(1);
RST_SOURCE <= RST;

-- Port map for the value limit counter
LIMT_CNT : entity work.parameterizable_counter
GENERIC MAP (LIMIT => switch_limit)
PORT MAP(
        clk => clk,
        rst => LIMT_CNT_RST,
        enable => LIMT_CNT_EN,
        count_out => LIMT_CNT_OUT
);

-- A register to store the value of SWITCHES when enabled.
```

```
switch_register : process (clk) is
begin
    if rising_edge(clk) then
        if LIMT_REG_RST = '1' then
            LIMT_REG_OUT <= (others => '0');
        elsif LIMT_REG_EN = '1' then
            LIMT_REG_OUT <= SWITCHES;
        end if;
    end if;
end process switch_register;

--  Sets the state as IDLE (reset state) when the reset input is set high.
--  At each clock cycle if the reset isn't high, the state is set to the next
--  state.
state_assignment : process (clk) is
begin
    if rising_edge(clk) then
        if (rst = '1') then
            state <= IDLE;
        else
            state <= next_state;
        end if;
    end if;
end process state_assignment;

fsm_process : process (state, en, rst, LIMT_CNT_OUT, FIFO_FULL, FROM_OUTPUT, LIMT_REG_OUT)
begin
    case state is
        when IDLE =>
            -- Start computing values as soon as the enable button is pressed and
            -- the output has finished displaying.
            if en = '1' and FROM_OUTPUT = '0' then
                next_state <= COMP;
            else
                next_state <= state;
            end if;
        when COMP =>
            -- When the FIFO is full, we want to wait until the OUTPUT logic
            -- pops some values and frees some space.
            if FIFO_FULL = '1' then
                next_state <= HOLD;
            -- As soon as the limit set by the switches have been reached, the
            -- logic should stop.
            elsif LIMT_CNT_OUT = to_integer(unsigned(LIMT_REG_OUT))-1 then
                next_state <= IDLE;
            else
                next_state <= state;
            end if;
        when HOLD =>
            -- In this state, we wait until the OUTPUT logic pops values and
            -- frees up some space from the FIFO. Once the FIFO is no longer full,
            -- more values can be computed until the switch limit is reached.
            if FIFO_FULL = '0' then
                next_state <= COMP;
            else
                next_state <= state;
            end if;
    end case;
end process fsm_process;

-- We want to reset the max value counter to 0 only in IDLE state.
```

```
LIMT_CNT_RST <= '1' when state = IDLE else
                '0';
-- The counter should only increment when in COMP state so it can
-- synchronise with the next outputs being generated. FIFO must not
-- full as well.
LIMT_CNT_EN <= '1' when state = COMP and FIFO_FULL = '0' else
               '0';
-- The source is enabled and generates the next value only when in
-- the compute state (COMP) and when the FIFO is not full. We need
-- the source to be enabled as soon as the enable button is clicked
-- during the IDLE state so that the first value can be computed
-- as soon as FIFO write is enabled.
EN_SOURCE <= '1' when state = COMP and FIFO_FULL = '0' else
             '1' when state = IDLE and en = '1' and FROM_OUTPUT = '0' else
             '0';
-- FIFO write needs to be enabled in the COMP state so that the
-- newly computed values are stored, FIFO should not be full.
FIFO_WR_EN <= '1' when state = COMP and FIFO_FULL = '0' else
              '0';
-- We store the value of SWITCHES when the state is IDLE and when the
-- user enables the logic, this way the logic will not change when the
-- SWITCH values are changed during operation.
LIMT_REG_EN <= '1' when state = IDLE and en = '1' else
               '0';
LIMT_REG_RST <= '1' when (state = IDLE and en = '0') or RST = '1' else
                '0';

end Behavioral;
```

SOURCE_CTRL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.all;

entity OUTPUT_CTRL is
    Generic (disp_delay : natural := 100000000);
    Port ( CLK : in  STD_LOGIC;
           USER_PB : in  STD_LOGIC_VECTOR (4 downto 0);
           SWITCHES : in  STD_LOGIC_VECTOR (7 downto 0);
           FROM_SOURCE : in  STD_LOGIC;
           TO_SOURCE : out  STD_LOGIC;
           DATA_FROM_FIFO : in  STD_LOGIC_VECTOR (7 downto 0);
           LEDS : out  STD_LOGIC_VECTOR (7 downto 0);
           FIFO_RD_EN : out  STD_LOGIC;
           FIFO_EMPTY : in  STD_LOGIC);
end OUTPUT_CTRL;

architecture Behavioral of OUTPUT_CTRL is

signal RST, EN : STD_LOGIC;
type state_type is (IDLE, DISP);        -- The FSM states
signal state, next_state: state_type;   -- The states as signals

-- Internal signals for the counters enable and resets, this is the
-- LED delay logic, it holds the value for a number of clock periods.
signal DISP_CNT_OUT : UNSIGNED (log2(disp_delay)-1 downto 0);
signal DISP_CNT_EN, DISP_CNT_RST : STD_LOGIC;
```

```
begin

RST <= USER_PB(1);
EN <= USER_PB(3);

-- Port map for the display delay counter
DISP_CNT: entity work.parameterizable_counter
GENERIC MAP (LIMIT => disp_delay)
PORT MAP(
        clk => clk,
        rst => DISP_CNT_RST,
        enable => DISP_CNT_EN,
        count_out => DISP_CNT_OUT
);

--  Sets the state as IDLE (reset state) when the reset input is set high.
--  At each clock cycle if the reset isn't high, the state is set to the next
--  state.
state_assignment : process (clk) is
begin
    if rising_edge(clk) then
        if (rst = '1') then
            state <= IDLE;
        else
            state <= next_state;
        end if;
    end if;
end process state_assignment;

fsm_process : process (state, FIFO_EMPTY)
begin
    case state is
        -- Start displaying as soon as the FIFO is no longer empty.
        when IDLE =>
            if FIFO_EMPTY = '0' then
                next_state <= DISP;
            else
                next_state <= state;
            end if;
        -- As soon as the FIFO is empty we can stop displaying.
        when DISP =>
            if FIFO_EMPTY = '1' then
                next_state <= IDLE;
            else
                next_state <= state;
            end if;
    end case;
end process fsm_process;

-- We can reset the value of the counter in the IDLE state so
-- when the logic resumes, the delays will be constant.
DISP_CNT_RST <= '1' when state = IDLE else
                '0';
-- The counter needs to be enabled in the DISP state, so that
-- the logic knows when to enable FIFO read to pull the new
-- values in, as the counter will roll-over.
DISP_CNT_EN <= '1' when state = DISP else
               '0';
-- When the logic is not displaying, we need the LEDs to be off.
LEDS <= DATA_FROM_FIFO when state = DISP else
        "00000000";
```

```
-- We only want the FIFO to be read from when the state is DISP
-- and when the counter is at 0.
FIFO_RD_EN <= '1' when state = DISP and DISP_CNT_OUT = disp_delay - 1 else
              '0';
-- This OUTPUT logic will always be active as long as the FIFO is not
-- empty, we can invert the FIFO_EMPTY signal so that when OUTPUT is
-- active (displaying all values via LEDs) this OUTPUT_ACTIVE signal
-- will be high. This signal can be passed to the SOURCE logic.
TO_SOURCE <= '1' when state = DISP else
             '0';

end Behavioral;
```

3 - VHDL Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

ENTITY TOP_LEVEL_tb IS
END TOP_LEVEL_tb;

-- TEST STRATEGY
--
--  Global Reset & Initialisation:
--      Outside of this vector, all btn inputs will initialise
--      to zeros, followed by a reset button click.
--
--  TEST 1:
--      Cycle through the first 5 values, this is the ensure that
--      all logic within this circuit is functioning properly.
--
--  TEST 2:
--      Following from Test 1, once the last byte has been output,
--      we need to verify that the LEDs output zeros.
--
--  TEST 3:
--      Following from TEST 1, a reset will be inputted via the
--      pushbutton. This is the verify that the circuit resets
--      properly and goes back to the initial state.
--
--  TEST 4:
--      Cycle through the first 5 values just like in TEST 1, but
--      the enable button will be pressed while the logic is in
--      operation (after the 1st input). This test will verify
--      that the enable input will be ignored.
--
--  TEST 5:
--      Continue from TEST 4, the reset button will be toggled after
--      the 4th input, this will verify that the circuit can be reset
--      while the FSM is in operation.
--
--  TEST 6:
--      Cycle through the first 5 values just like in TEST 4, but
--      the switches will be changed while the FSM is operation
--      (after the 2nd input). This test will verify that changing
--      the switches will be ignored when the FSM is in operation.
--
--  TEST 7:
```

```
--      Cycle through as many inputs as it takes to cause the FIFO
--      to become full, verify that the circuit can still function
--      as designed and handle the FIFO becoming full.

ARCHITECTURE behavior OF TOP_LEVEL_tb IS
   --Inputs
   signal GCLK : std_logic;
   signal BTN : std_logic_vector(4 downto 0);
   signal SW : std_logic_vector(7 downto 0);

         --Outputs
   signal LD : std_logic_vector(7 downto 0);

   -- Clock period definitions
   constant GCLK_period : time := 10 ns;

   -- Defining a record of valid outputs to verify the circuit.
   -- The inputs to this circuit (via pushbuttons) will be done
   -- outside of the test vector array.
   type valid_output_array is array (natural range <>) of STD_LOGIC_VECTOR(7 downto 0);
   constant valid_outputs : valid_output_array := (
       -- OUTPUT 1 (4800C00B)
       (X"48"),
       (X"00"),
       (X"C0"),
       (X"0B"),
       -- OUTPUT 2 (0000CA5F)
       (X"00"),
       (X"00"),
       (X"CA"),
       (X"5F"),
       -- OUTPUT 3 (0000570E)
       (X"00"),
       (X"00"),
       (X"57"),
       (X"0E"),
       -- OUTPUT 4 (0380DFAD)
       (X"03"),
       (X"80"),
       (X"DF"),
       (X"AD"),
       -- OUTPUT 5 (0000006C)
       (X"00"),
       (X"00"),
       (X"00"),
       (X"6C")
    );

BEGIN

         -- Instantiate the Unit Under Test (UUT)
   uut: entity work.TOP_LEVEL
   GENERIC MAP (disp_delay => 30)
   PORT MAP (
          GCLK => GCLK,
          BTN => BTN,
          SW => SW,
          LD => LD
   );

   -- Clock process definitions
```

```
GCLK_process :process
begin
            GCLK <= '0';
            wait for GCLK_period/2;
            GCLK <= '1';
            wait for GCLK_period/2;
end process;


-- Stimulus process
set_inputs: process
begin
 -- hold reset state for at least 2000 ns.
 wait for 2500 ns;
     wait until falling_edge(GCLK);

 -- TEST 0:
 --  Global Reset & Initialisation
 --  Duration: 248 clock periods
 BTN <= "00000";
 SW <= "00000000";
 wait for GCLK_period*18;
 BTN(1) <= '1';
 wait for GCLK_period*30;
 BTN(1) <= '0';
 -- Wait for at least 200 clock periods after
 -- each reset.
 wait for GCLK_period*200;

 -- TEST 1:
 --  Duration: 761.5 clock periods
 SW <= "00000101";
 BTN(3) <= '1';
 wait for GCLK_period*30;
 BTN(3) <= '0';

 wait for GCLK_period*765;

 -- TEST 2:
 --  Duration: 150 clock periods
 wait for GCLK_period*150;

 -- TEST 3:
 --  Duration: 230 clock periods
 BTN(1) <= '1';
 wait for GCLK_period*30;
 BTN(1) <= '0';
 wait for GCLK_period*200;

 wait for GCLK_period*150;

 -- TEST 4:
 --  Duration: 761.5 clock periods
 SW <= "00000101";
 BTN(3) <= '1';
 wait for GCLK_period*30;
 BTN(3) <= '0';
 -- wait until the end of first output display for enable toggle.
 wait for GCLK_period*162.51;
 BTN(3) <= '1';
 wait for GCLK_period*30;
```

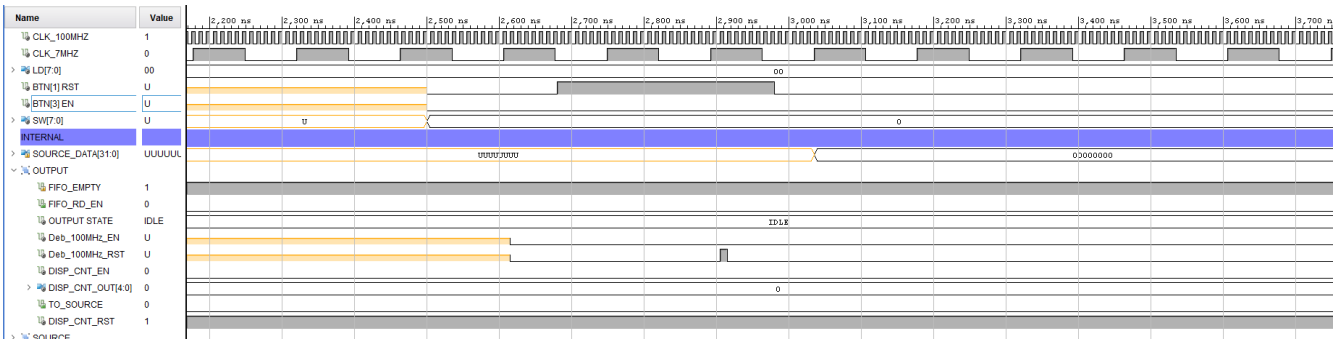
```
    BTN(3) <= '0';

    -- TEST 5:
    -- wait until the end of third output display for reset toggle.
    wait for GCLK_period*210;
    BTN(1) <= '1';
    wait for GCLK_period*30;
    BTN(1) <= '0';

    wait for GCLK_period*200;

    -- TEST 6:
    SW <= "00000101";
    BTN(3) <= '1';
    wait for GCLK_period*30;
    BTN(3) <= '0';
    -- wait until 2nd output has been fully displayed
    wait for GCLK_period*304.5;
    SW <= "00000011";

    -- TEST 7
    wait for GCLK_period*800;
    BTN(1) <= '1';
    wait for GCLK_period*30;
    BTN(1) <= '0';
    wait for GCLK_period*200;
    SW <= "00100011";
    BTN(3) <= '1';
    wait for GCLK_period*30;
    BTN(3) <= '0';

    wait;
end process;


check_outputs : process
begin
    -- hold reset state for at least 2000 ns.
    wait for 2500 ns;
    wait until falling_edge(GCLK);
    -- Wait for Test 0 (resets) to complete
    wait for GCLK_period*248;

    -- TEST 1:
    --  The outputs start displaying as soon as the
    --  OUTPUT FSM goes to DISP state, which takes 69.5
    --  clock periods. We will start checking from 72
    --  clock cycles so that the LEDs would've output
    --  2.5 clock periods before.
    wait for GCLK_period*72;
    test_1_check_loop : for i in 0 to 19 loop
        -- Check if the LED output matches the array of known valid output at
        -- the end of every enable btn toggle, as soon as the btn is depressed.
        -- Notify if it's a pass or fail. I'm using severity warning to ensure
        -- that the simulation doesn't halt as soon as a vector fails like it
        -- does for severity failure.
        assert (( LD = valid_outputs(i) ))
        report "TEST 1 VECTOR " & integer'image(i) &
        " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
        " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(i))))
        severity warning;
```

```
        report "TEST 1 VECTOR " & integer'image(i) & " PASS."
        severity note;
        -- LEDs display for 30 clk cycles,
        wait for GCLK_period*30;
    end loop;
    wait for GCLK_period*94;

    -- TEST 2:
    --  The sequence has finished, the LEDs should be outputting 0, we need to check this.
    assert (( LD = x"00" ))
    report "TEST 2 " &
    " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
    " Expected LED Output = 0"
    severity warning;
    report "TEST 2 PASS."
    severity note;

    wait for GCLK_period*149;

    -- TEST 3:
    --  The LEDs should stay at zeros after the reset. We need to wait for
    --  230 clock periods for the reset to complete and an additional 5
    --  periods so we can read the output.
    wait for GCLK_period*235;
    assert (( LD = x"00" ))
    report "TEST 3 " &
    " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
    " Expected LED Output = 0"
    severity warning;
    report "TEST 3 PASS."
    severity note;

    wait for GCLK_period*175;


    -- TEST 4:
    --  Check if the enable button is ignored when the logic is in operation.
    --  It takes 72.5 clk periods to go from en high to an output, we can wait
    --  an additional 2.5 periods then measure.
    wait for GCLK_period*75;
    test_4_check_loop : for i in 0 to 11 loop
        -- Check if the LED output matches the array of known valid output at
        -- the end of every enable btn toggle, as soon as the btn is depressed.
        -- Notify if it's a pass or fail. I'm using severity warning to ensure
        -- that the simulation doesn't halt as soon as a vector fails like it
        -- does for severity failure.
        assert (( LD = valid_outputs(i) ))
        report "TEST 4 VECTOR " & integer'image(i) &
        " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
        " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(i))))
        severity warning;
        report "TEST 4 VECTOR " & integer'image(i) & " PASS."
        severity note;
        -- LEDs display for 30 clk cycles,
        wait for GCLK_period*30;
    end loop;

    -- TEST 5
    wait for GCLK_period*30;
    assert (( LD = x"00" ))
    report "TEST 5 " &
```

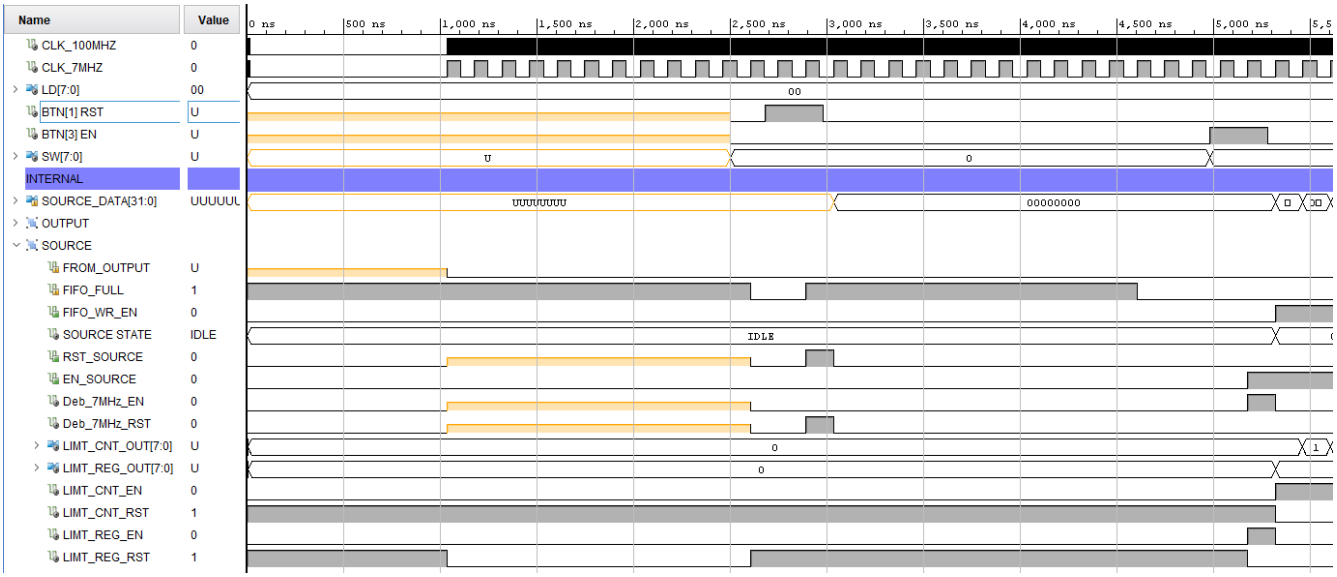
```
    " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
    " Expected LED Output = 00"
    severity warning;
    report "TEST 5 PASS."
    severity note;

    wait for GCLK_period*200;


    -- TEST 6
    wait for GCLK_period*67;
    test_6_check_loop : for i in 0 to 19 loop
        -- Check if the LED output matches the array of known valid output at
        -- the end of every enable btn toggle, as soon as the btn is depressed.
        -- Notify if it's a pass or fail. I'm using severity warning to ensure
        -- that the simulation doesn't halt as soon as a vector fails like it
        -- does for severity failure.
        assert (( LD = valid_outputs(i) ))
        report "TEST 6 VECTOR " & integer'image(i) &
        " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
        " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(i))))
        severity warning;
        report "TEST 6 VECTOR " & integer'image(i) & " PASS."
        severity note;
        -- LEDs display for 30 clk cycles,
        wait for GCLK_period*30;
    end loop;

    wait;
end process;


END;
```

3.1 Behavioural Simulation

Waveform 1: Global Reset & Initialisation Showing Output

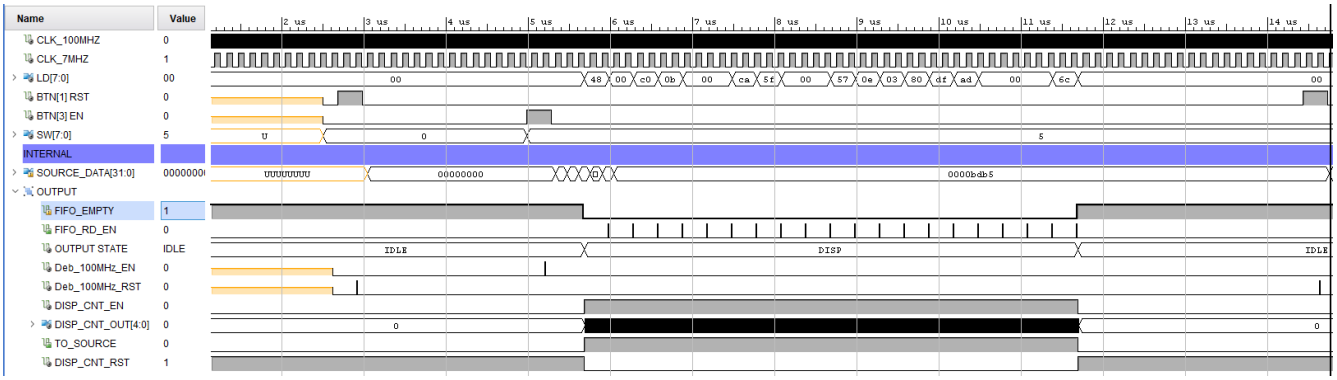


Waveform 2: Global Reset & Initialisation Showing Source



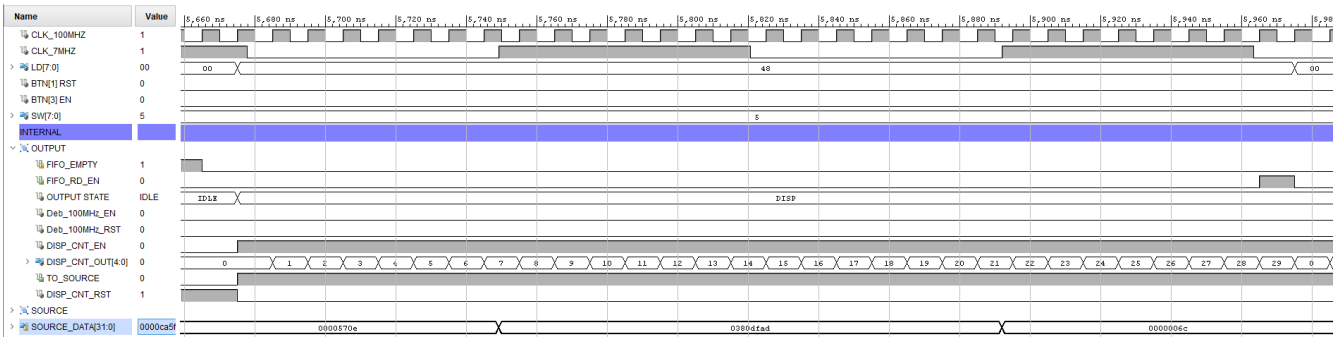
TEST 1: involves cycling through the first 5 values, this is the ensure that all logic within this circuit is functioning properly.

Waveform 3: Test 1.1



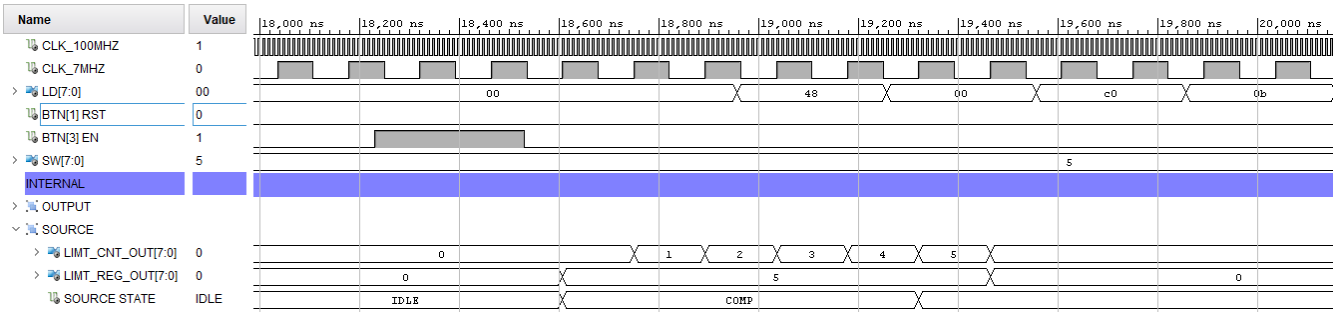
Zoomed out showing full cycle of first 5 values from btye ‘48’ to ‘6C’.

Waveform 4: Test 1.2



Waveform 4 shows the display counter incrementing from 0-29 as soon as 'DISP_CNT_OUT' goes high. This ensures that the LED's flash a value for a second at a time.

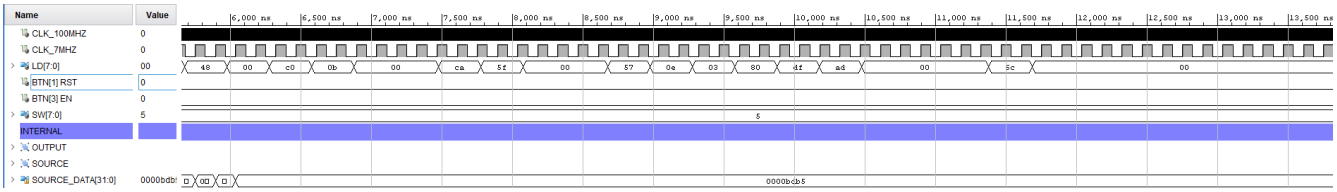
Waveform 5: Test 1.3



Waveform 5 shows the source state entering 'Comp' mode and storing the value of the switches so that a change mid operation doesn't affect the circuit

TEST 2: Following from Test 1, once the last byte has been output, we need to verify that the LEDs output zeros.

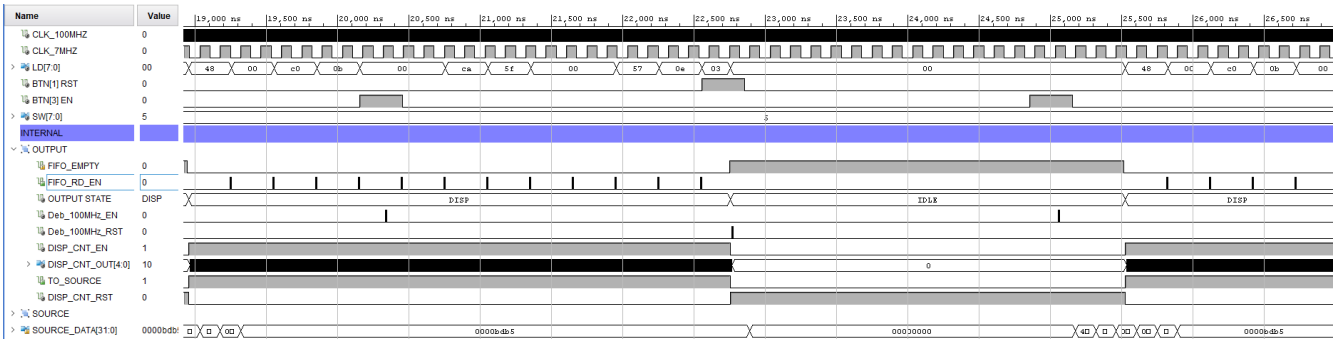
Waveform 6: Test 2



Above waveform shows that the output returns to 0 after a full cycle.

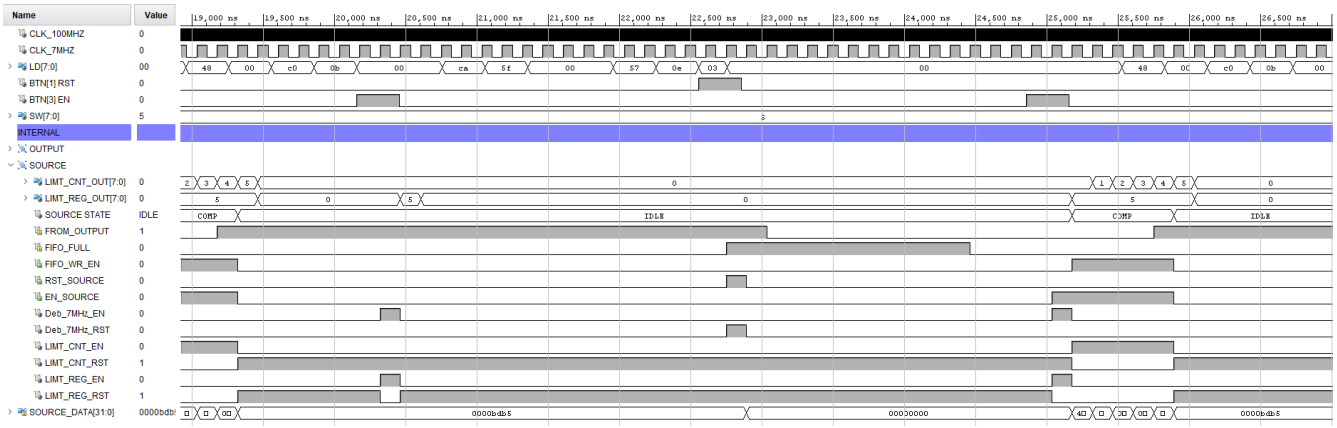
TEST 3: Following from TEST 1, a reset will be inputted via the push-button. This is the verify that the circuit resets properly and goes back to the initial state.

Waveform 7: Test 3.1



Above shows output signals being reset after operation.

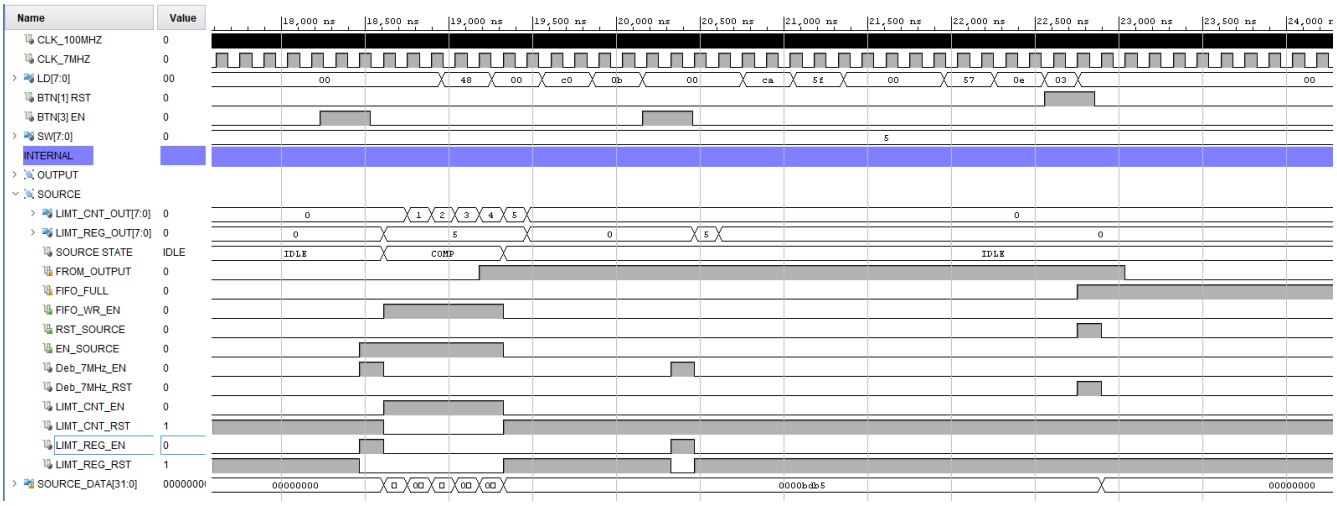
Waveform 8: Test 3.2



Above shows source signals being reset after operation.

TEST 4: Cycle through the first 5 values just like in TEST 1, but the enable button will be pressed while the logic is in operation (after the 1st input). This test will verify that the enable input will be ignored.

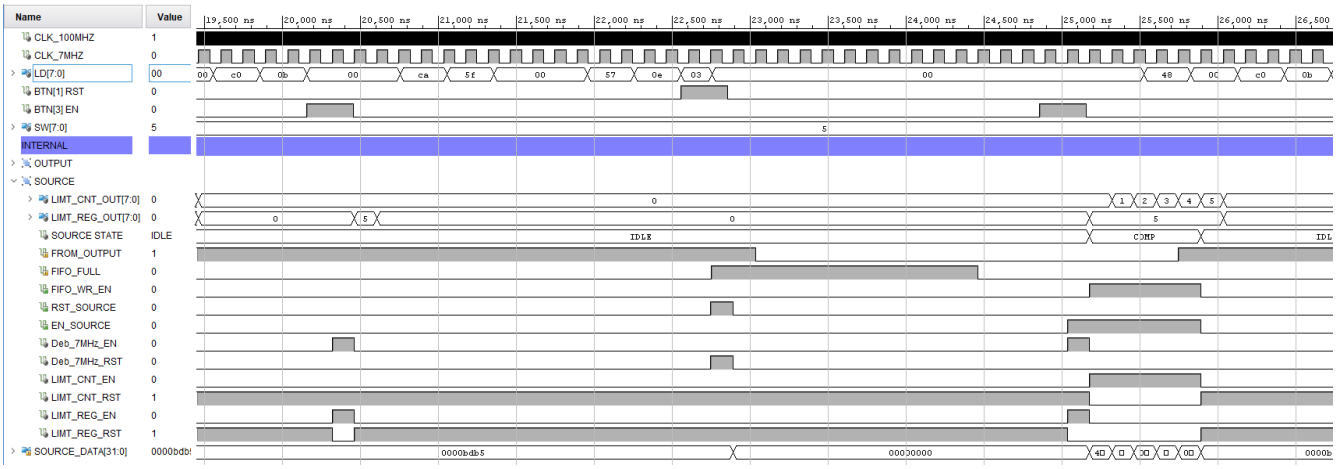
Waveform 9: Test 4



Above shows enable pressed mid-operation the value of 'LMT_REG_OUT' changes to 5 for one clock cycle then returns back to zero. This is the expected output as you can see that the source state never changes from idle.

TEST 5: Continue from TEST 4, the reset button will be toggled after the 4th input, this will verify that the circuit can be reset while the FSM is in operation.

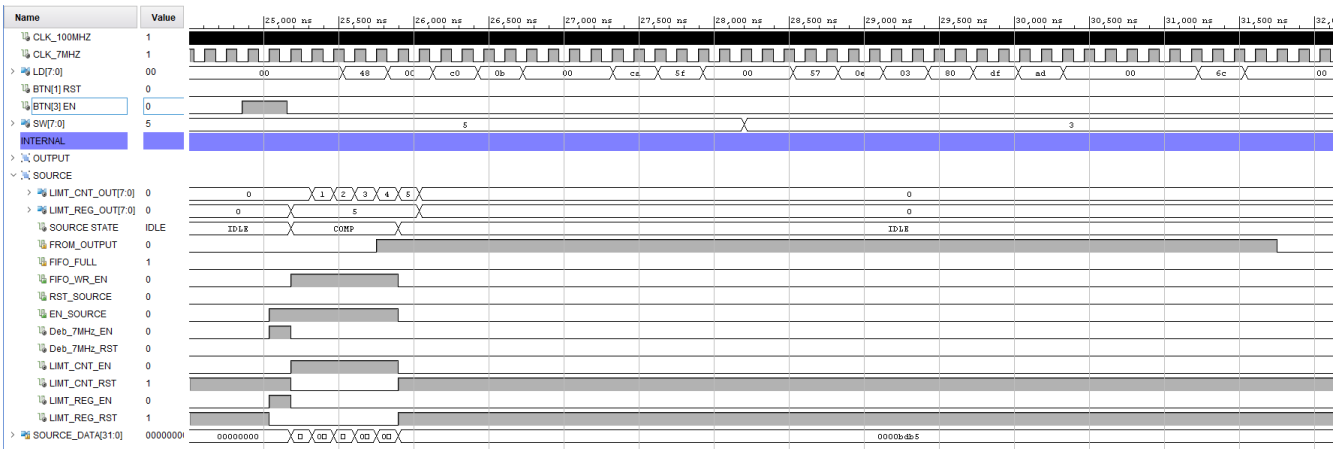
Waveform 10: Test 5



Waveform 10 shows a reset mid way through operation. The state goes back to idle and the LED outputs zeros.

TEST 6: Cycle through the first 5 values just like in TEST 4, but the switches will be changed while the FSM is operation (after the 2nd input). This test will verify that changing the switches will be ignored when the FSM is in operation.

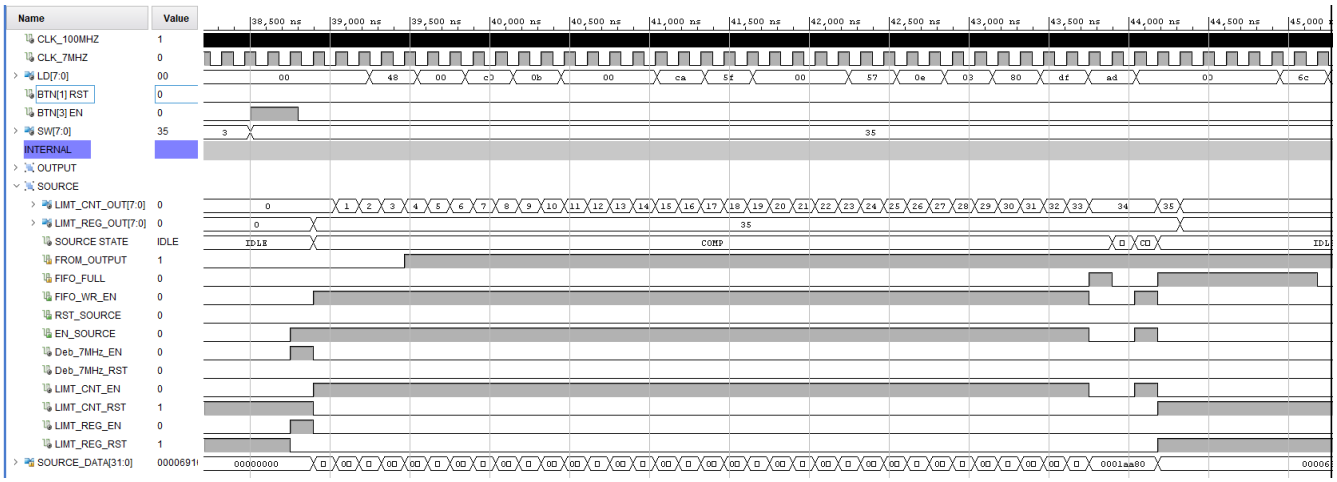
Waveform 11: Test 6



Waveform 11 shows that changing the switch value from 5 to 3 doesn't have any effect on the output

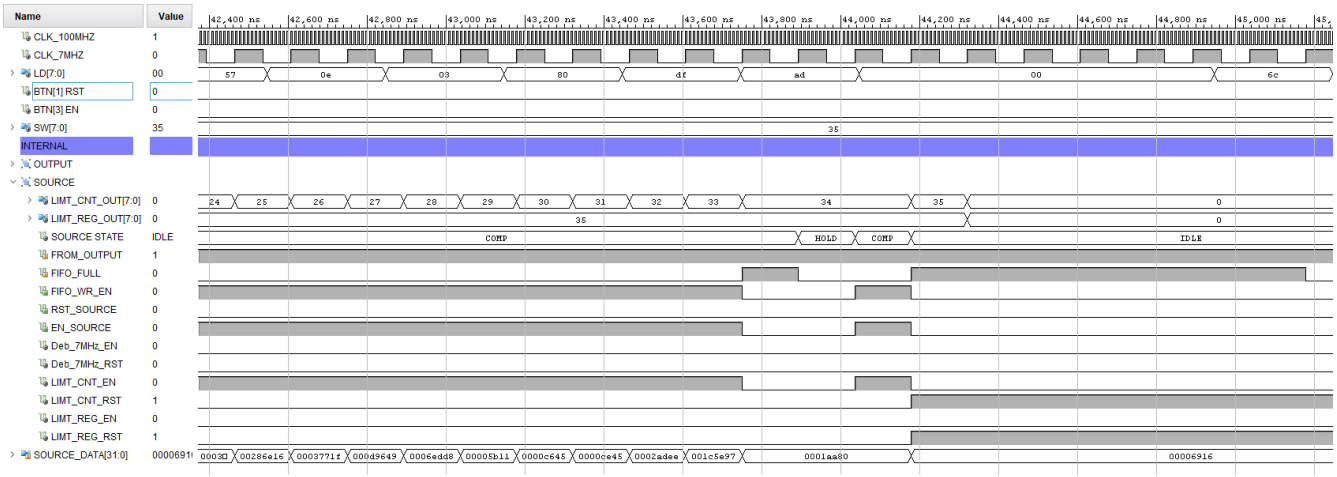
TEST 7: Cycle through as many inputs as it takes to cause the FIFO to become full, verify that the circuit can still function as designed and handle the FIFO becoming full.

Waveform 12: Test 7.1



Shows full cycle from source side

Waveform 13: Test 7.2



Shows the source state going into hold after 'FIFI_FUll' goes to logic high,

RTL Component Statistics

```
           2 Input     27 Bit       Adders := 1
           2 Input      9 Bit       Adders := 1
           2 Input      8 Bit       Adders := 1
           2 Input      6 Bit       Adders := 1
+---Registers :
                     32 Bit    Registers := 1
                     27 Bit    Registers := 1
                      8 Bit    Registers := 2
                      6 Bit    Registers := 1
                      2 Bit    Registers := 2
                      1 Bit    Registers := 33
+---Muxes :
           2 Input     27 Bit        Muxes := 1
```

```
          64 Input     16 Bit        Muxes := 2
           2 Input      8 Bit        Muxes := 2
           2 Input      6 Bit        Muxes := 1
           5 Input      2 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 3
           4 Input      1 Bit        Muxes := 1
------------------------------------------------------------------------------
Finished RTL Component Statistics
------------------------------------------------------------------------------
```

RTL Hierarchical Component Statistics

```
+---Registers :
                       1 Bit    Registers := 2
Module xpm_cdc_single
Detailed RTL Component Info :
+---Registers :
                       2 Bit    Registers := 1
Module parameterizable_counter
Detailed RTL Component Info :
+---Adders :
           2 Input      8 Bit       Adders := 1
+---Registers :
                       8 Bit    Registers := 1
+---Muxes :
           2 Input      8 Bit        Muxes := 1
Module SOURCE_CTRL
Detailed RTL Component Info :
+---Adders :
           2 Input      9 Bit       Adders := 1
+---Registers :
                       8 Bit    Registers := 1
                       2 Bit    Registers := 1
+---Muxes :
           5 Input      2 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 3
           4 Input      1 Bit        Muxes := 1
Module parameterizable_counter__parameterized0
Detailed RTL Component Info :
+---Adders :
           2 Input     27 Bit       Adders := 1
+---Registers :
                      27 Bit    Registers := 1
+---Muxes :
           2 Input     27 Bit        Muxes := 1
Module OUTPUT_CTRL
Detailed RTL Component Info :
+---Registers :
                       1 Bit    Registers := 1
+---Muxes :
           2 Input      8 Bit        Muxes := 1
Module Param_Counter
Detailed RTL Component Info :
+---Adders :
           2 Input      6 Bit       Adders := 1
+---Registers :
                       6 Bit    Registers := 1
+---Muxes :
           2 Input      6 Bit        Muxes := 1
Module MEM_A
```

```
Detailed RTL Component Info :
+---Muxes :
          64 Input     16 Bit        Muxes := 1
Module MEM_B
Detailed RTL Component Info :
+---Muxes :
          64 Input     16 Bit        Muxes := 1
Module DATA_SOURCE
Detailed RTL Component Info :
+---Adders :
           3 Input     32 Bit       Adders := 1
+---Registers :
                       32 Bit    Registers := 1
Module Debouncer
Detailed RTL Component Info :
+---Registers :
                        1 Bit    Registers := 3
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
```