Lab 1 - Task A

VHDL code

Top Level (TOP_LEVEL)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Top Level Circuit Logic:
--  This is the top level part of the code, which controls the layout of the different
--   child components. Centrally, there is the COUNT component, which counts along the
--   different addresses the Fibonacci numbers are located. Between the COUNT and each
--   of the two human-controlled inputs (the enable and the reset) there is a
--   BETTER_DEBOUNCER component which makes sure each button press will only count as
--   an input once to the COUNT component. This is an improvement on the DEBOUNCE
--   component from last years labs. Between the COUNT component and the Display output
--   there is an asynchronous ROM which outputs the Fibonacci number for a given address.

entity TOP_LEVEL is
    generic (DELAY : integer := 50000000);    -- The number of cycles to debounce by
                                              --  (set to fifty million or roughly half
                                              --  a second).
    Port ( clk : in STD_LOGIC;                -- The board's clock.
           en : in STD_LOGIC;                 -- Raw enable button input.
           rst : in STD_LOGIC;                -- Raw reset button input.
           display : out STD_LOGIC_VECTOR (7 downto 0)); -- Current Fibbonacci number
end TOP_LEVEL;

architecture Behavioral of TOP_LEVEL is

-- Intermediate debounced values for the enable and the reset
signal en_debounced, rst_debounced: STD_LOGIC;

-- The current address we have counted to (i.e how many steps we are through the
--  sequence)
signal count: UNSIGNED (3 downto 0);

begin

    -- Enable Debouncer:
    --   This BETTER_DEBOUNCER component debounces the enable button. It takes the raw
    --    button input and turns it into a single pulse for each press, which can then
    --    be fed to COUNT to count once. The time it is debounced over is specified
    --    by the delay generic parameter, which is set to the generic parameter of this
    --    component with the same name.
    en_debouncer: entity work.BETTER_DEBOUNCER
    generic map (DELAY => DELAY)
    port map (
        clk => clk,
        Sig => en,
        Deb_Sig => en_debounced);

    -- Reset Debouncer:
    --   This BETTER_DEBOUNCER component debounces the reset button. It takes the raw
    --    button input and turns it into a single pulse for each press, which can then
    --    be fed to COUNT to do a single reset. The time it is debounced over is
    --    specified by the delay generic parameter, which is set to the generic
    --    parameter of this component with the same name.
    rst_debouncer: entity work.BETTER_DEBOUNCER
    generic map (DELAY => DELAY)
    port map (
        clk => clk,
        Sig => rst,
        Deb_Sig => rst_debounced);

    -- Counter:
    --   This COUNT component takes the debounced inputs and produces a count which is
    --    used to track the progress through the Fibonacci sequence.
    counter: entity work.COUNT
    port map (
        clk => clk,
        en => en_debounced,
        rst => rst_debounced,
        count => count);

    -- Fibonacci Lookup ROM:
    --   This ASYNC_ROM component stores the different elements in the Fibonacci
    --    sequence and allows us to look up the element at a given place.
    fibonacci_lookup: entity work.ASYNC_ROM
    port map (
        Address => count,
        DataOut => display);
end Behavioral;
```

Improved Debouncer (BETTER_DEBOUNCER)

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

-- Improved Debouncer:
--   This is the new debouncer design improving from the one from last years labs. It
--    is based on a parameterisable counter. This counter is designed to increase in
--    count when there is a signal an reset when there is no signal. The debouncer
--    will turn on the debounced output when the counter is reached its max and the
--    signal is still active. The max value of the counter is specified by the DELAY
--    generic argument, which defaults to fifty million.
entity BETTER_DEBOUNCER is
    generic (DELAY : integer := 50000000); -- How many clock cycles should we delay
                                           --  for (default is fifty million or half
                                           --  a second)?
    Port ( CLK : in  STD_LOGIC;            -- The clock signal.
           Sig : in  STD_LOGIC;            -- The signal the debouncer is debouncing.
           Deb_Sig : out  STD_LOGIC);      -- The debounced signal.
end BETTER_DEBOUNCER;

architecture Behavioral of BETTER_DEBOUNCER is

    -- The reset command for the counter. We want to reset when there is no input
    --  signal.
    signal Counter_Input_Rst : STD_LOGIC;
    -- The enable command for the counter. We want to enable the counter whenever
    --  the input signal is there.
    signal Counter_Input_En : STD_LOGIC;
    -- The current count from the counter.
    signal Counter_Output : UNSIGNED (size(DELAY)-1 downto 0);
    -- Is the current count equal to the maximum count.
    signal Counter_Output_Is_Max_Value : STD_LOGIC;
    -- The value of the d-type flip-flop.
    signal DFF_Output : STD_LOGIC;

begin

    -- Enable the counter when the signal is there and it isn't at it's max.
    Counter_Input_En <= Sig and not Counter_Output_Is_Max_Value;
    -- Reset the counter when the signal isn't there.
    Counter_Input_Rst <= not Sig;
    -- Check whether the counter output is at it's max value (the delay - 1).
    Counter_Output_Is_Max_Value <= '1' when Counter_Output = DELAY-1 else '0';
    -- The debounced signal only should last for a single clock cycle so only
    --  output a debounced signal if it is at max value and wasn't last cycle.
    Deb_Sig <= Counter_Output_Is_Max_Value and not DFF_Output;

    -- The counter component is a generic counter. Here it is used to count the
    --  time in the delay.
    counter: entity work.GENERIC_COUNT
    generic map(N => DELAY) -- The size of the counter. We want to set it to delay as
                            --  the counter will go up by one each clock cycle.
    port map (
        clk => clk,
        en => Counter_Input_En,
        rst => Counter_Input_Rst,
        count => Counter_Output);

    -- The process which manages the d-type flip-flop. It sets it self to
    --  Counter_Output_Is_Max_Value with a one cycle delay. This is to ensure the
    --  debouncer only sends a single one-cycle pulse for each press.
    process (CLK) is
    begin
        if RISING_EDGE(clk) then
            DFF_Output <= Counter_Output_Is_Max_Value;
        end if;
    end process;
end Behavioral;
```

Debouncer Internal Generic Counter (GENERIC_COUNT)

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

-- Counter to N:
--   This component functions as a log2(N)-bit counter up to N. It has enable and
--    reset logic, where the enable is expected to be turned on in single clock cycle
--    pulses. Upon reaching fourteen, the next iteration will send it back to zero.

entity GENERIC_COUNT is
    generic (N : integer := 8);
    Port ( clk : in STD_LOGIC;                              -- clock
           en  : in STD_LOGIC;                              -- enable (already debounced)
           rst : in STD_LOGIC;                              -- reset (already debounced)
           count : out UNSIGNED (size(N)-1 downto 0)); -- the current count
end GENERIC_COUNT;

architecture Behavioral of GENERIC_COUNT is

-- Internal count signal that can be mirrored by the external one (because you cannot
--   read outputs)
signal count_internal: UNSIGNED (size(N)-1 downto 0);

begin
    -- Make the external count mirror the internal one
    count <= count_internal;

    -- Counter Process:
    --   Each tick the counter will increase by one if it is enabled. If reset is
    --    equal to 1 it will instead reset to 0. If the current count is N it
    --    will go to zero instead of N+1.
    counter: process(clk)
    begin
        if RISING_EDGE(clk) then
            if rst = '1' then
                count_internal <= (others => '0');
            elsif en = '1' then
                if count_internal /= N - 1 then
                    count_internal <= (count_internal + 1);
                else
                    count_internal <= (others => '0');
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Counter (COUNT)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Counter to thirteen:
--   This component functions as a four-bit counter up to thirteen. It has enable and
--    reset logic, where the enable is expected to be turned on in single clock cycle
--    pulses. Upon reaching fourteen, the next iteration will send it back to zero. We
--    could have easily merged this into the generic counter, but the lab script seemed
--    to imply we shouldn't for some reason.
```

```vhdl
entity COUNT is
    Port ( clk : in STD_LOGIC;                                  -- clock
           en : in STD_LOGIC;                                   -- enable (already debounced)
           rst : in STD_LOGIC;                                  -- reset (already debounced)
           count : out UNSIGNED (3 downto 0));                  -- the current count
end COUNT;

architecture Behavioral of COUNT is

-- Internal count signal that can be mirrored by the external one (because you cannot
--  read outputs)
signal count_internal: UNSIGNED (3 downto 0);

begin
    -- Make the external count mirror the internal one
    count <= count_internal;

    -- Counter:
    --  Each tick the counter will increase by one if it is enabled. If reset is
    --   equal to one it will instead reset to 0. If the current count is thirteen it
    --   will go to zero instead of fourteen.

        counter: process(clk)
        begin
            if RISING_EDGE(clk) then
                if rst = '1' then
                    count_internal <= (others => '0');
                elsif en = '1' then
                    if count_internal = 13 then
                        count_internal <= (others => '0');
                    else
                        count_internal <= (count_internal + 1);
                    end if;
                end if;
            end if;
        end process;
end Behavioral;
```

Fibonacci Lookup ROM (ASYNC_ROM)

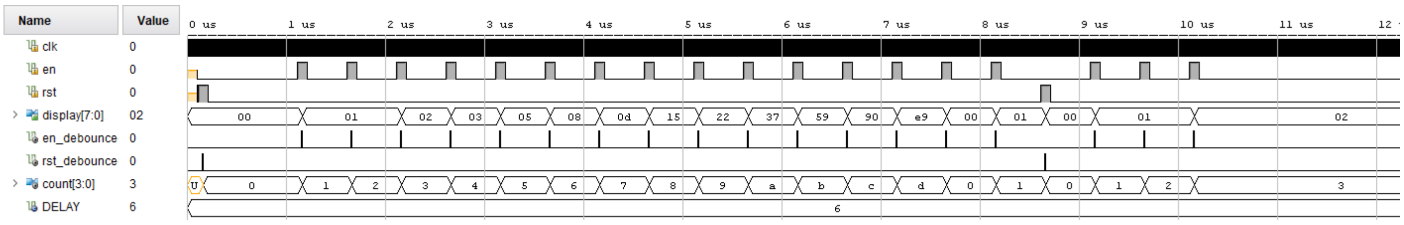
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Async ROM:
--  This ROM stores the values in the Fibonacci sequence up to 233 (as this is the
--   last value which can be stored using 8 bits). The component is given an address
--   and then sets the DataOut accordingly. As it is asynchronous, this happens
--   without waiting for a rising clock edge.

entity ASYNC_ROM is
    Port ( Address : in UNSIGNED (3 downto 0);  -- Address to look up (i.e position
                                                --  in the sequence to look up)
           DataOut : out STD_LOGIC_VECTOR (7 downto 0));  -- Value at address (i.e
                                                          --  element in the sequence
                                                          --  at specified position)
end ASYNC_ROM;

architecture arch of ASYNC_ROM is

type ROM_Array is array (0 to 15) of std_logic_vector(7 downto 0);
    constant Content: ROM_Array := (
        0 => B"00000000", -- 0
        1 => B"00000001", -- 1
        2 => B"00000001", -- 1
        3 => B"00000010", -- 2
        4 => B"00000011", -- 3
        5 => B"00000101", -- 5
        6 => B"00001000", -- 8
        7 => B"00001101", -- 13
        8 => B"00010101", -- 21
        9 => B"00100010", -- 34
        10 => B"00110111", -- 55
        11 => B"01011001", -- 89
        12 => B"10010000", -- 144
        13 => B"11101001", -- 233
        others => B"00000000");
begin
    DataOut <= Content(to_integer(Address));
end arch;
```

Testbench (TOP_LEVEL_tb)

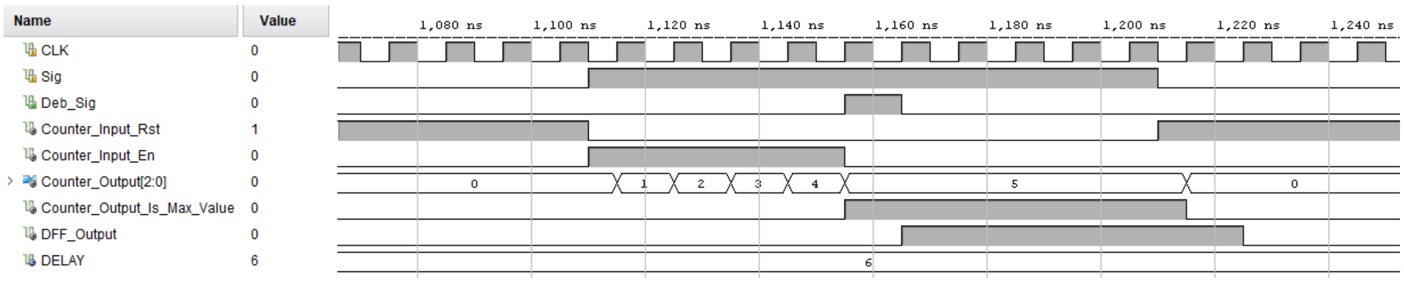
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TOP_LEVEL_tb is
end TOP_LEVEL_tb;

architecture Behavioral of TOP_LEVEL_tb is
    signal clk: STD_LOGIC;                                      -- fake clock signal
    signal en, rst: STD_LOGIC;                                  -- fake button signals
    signal display: STD_LOGIC_VECTOR (7 downto 0);              -- circuit display output
    constant clk_period: time := 10 ns;                         -- how fast the fake clock is
    constant button_press_period: time := clk_period * 10;      -- time button held down for
    constant button_press_wait_period: time := clk_period * 40; -- time between button presses
begin

    UUT : entity work.TOP_LEVEL
    generic map (DELAY => 6)
    port map (
        clk => clk,
        en => en,
        rst => rst,
        display => display);

    -- Clock process
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

-- Tests
    test :process
    begin
        -- Testing Strategy:
        --   Since the combination of inputs are the clock, the enable and the reset, there
        --    are four things which need to be tested:
        --    - Firstly, we need to test that it counts normally when the enable is repeatedly
        --       pressed. This ensures the enable and the basic counting functionality is
        --       working.
        --    - Secondly, we need to test that it does not count when the enable isn't
        --       pressed. This ensures the enable works.
        --    - Thirdly, we need to test that when it reaches the maximum value it correctly
        --       sets to zero.
        --    - Finally, we need to test that it behaves properly when reset, including being
        --       able to start counting again afterwards.

        -- Standard 100 ns delay at start
        wait for 100 ns;

        -- Set inputs to default value
        en <= '0'; rst <= '0';
        wait for clk_period;

        -- Start it off at 0 by resetting
        rst <= '1'; -- user is presses reset
        wait for button_press_period;
        rst <= '0'; -- user releases reset
        wait for button_press_wait_period;

        -- Test what happens if user doesn't press button (large enough time so it is visible
        --  in sim when zoomed out)
        wait for clk_period * 50;
        -- Test the standard counting behaviour and what happens when it reaches the maximum
        --  number
        cycle_through_15: for i in 0 to 14 loop
            en <= '1'; -- user is presses enable
            wait for button_press_period;
            en <= '0'; -- user releases enable
            wait for button_press_wait_period;
        end loop cycle_through_15;

        -- Test reset
        rst <= '1'; -- user is presses reset
        wait for button_press_period;
        rst <= '0'; -- user releases reset
        wait for button_press_wait_period;

        -- Test that counting still works after reset
        cycle_through_3: for i in 0 to 2 loop
            en <= '1'; -- user is presses enable
            wait for button_press_period;
            en <= '0'; -- user releases enable
            wait for button_press_wait_period;
        end loop cycle_through_3;

            -- wait forever
            wait;
        end process;
end Behavioral;
```

Screenshots of the simulation results

View of full run with top level component variables



View of single debounce with improved debouncer component variables



Synthesis report

RTL Component Statistics

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
           2 Input      4 Bit       Adders := 1
+---Registers :
                        4 Bit    Registers := 1
                        1 Bit    Registers := 2
+---Muxes :
           2 Input      4 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 4
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
```

RTL Hierarchical Component Statistics

```
---------------------------------------------------------------------------------
```

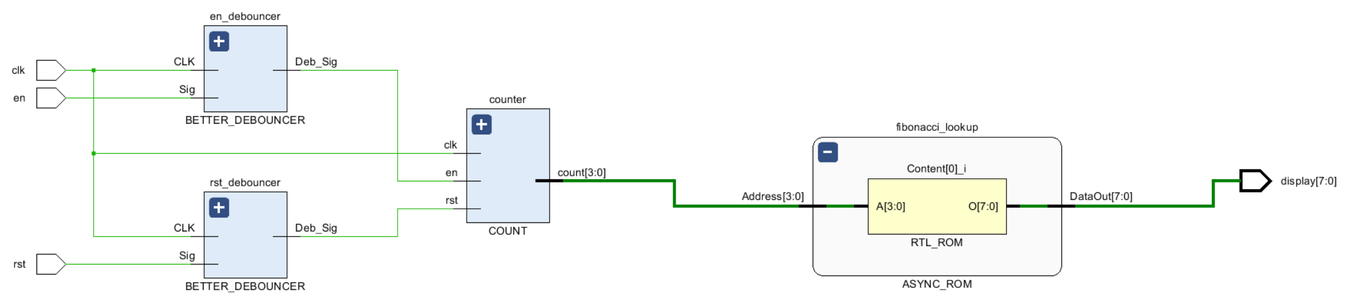
```
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module GENERIC_COUNT
Detailed RTL Component Info :
+---Muxes :
	   2 Input      1 Bit        Muxes := 1
Module BETTER_DEBOUNCER
Detailed RTL Component Info :
+---Registers :
	                1 Bit    Registers := 1
+---Muxes :
	   2 Input      1 Bit        Muxes := 1
Module COUNT
Detailed RTL Component Info :
+---Adders :
	   2 Input      4 Bit       Adders := 1
+---Registers :
	                4 Bit    Registers := 1
+---Muxes :
	   2 Input      4 Bit        Muxes := 1
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
```

Comments

Strangely the generic count component used in the denouncers only has 1 multiplexer according to the rtl statistics. This is contradicted by the schematics. I am not sure why this is.

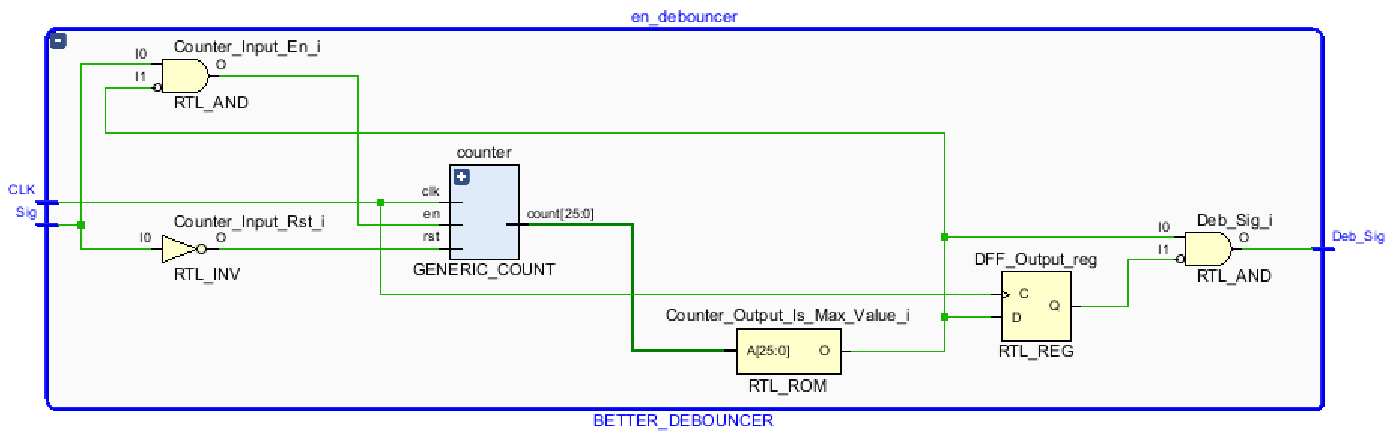
Schematics and commentary

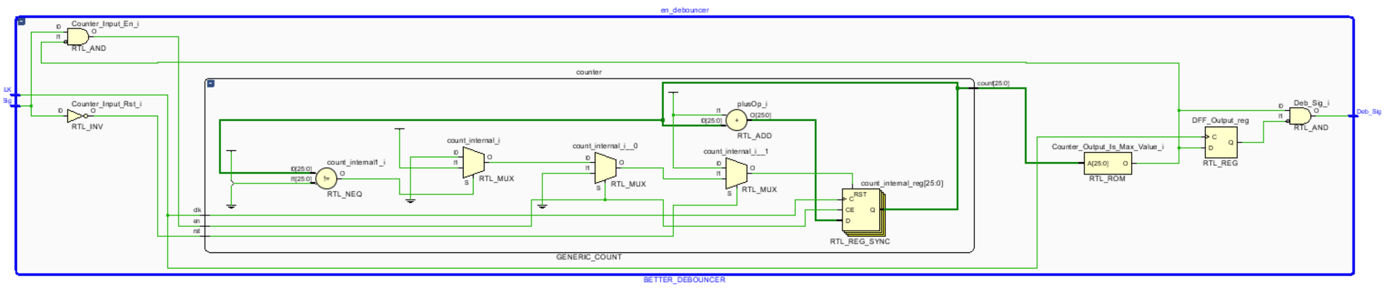
RTL Top Level Schematic



The RTL top level schematic matches the code. It is possible to see the debouncers debounce the buttons that are then fed into the counter. This counter then makes a request to the ROM which supplies the data to display. The ROM is also as expected as it just consists of a single basic ROM component which has the right size.

RTL Improved Debouncer Schematic

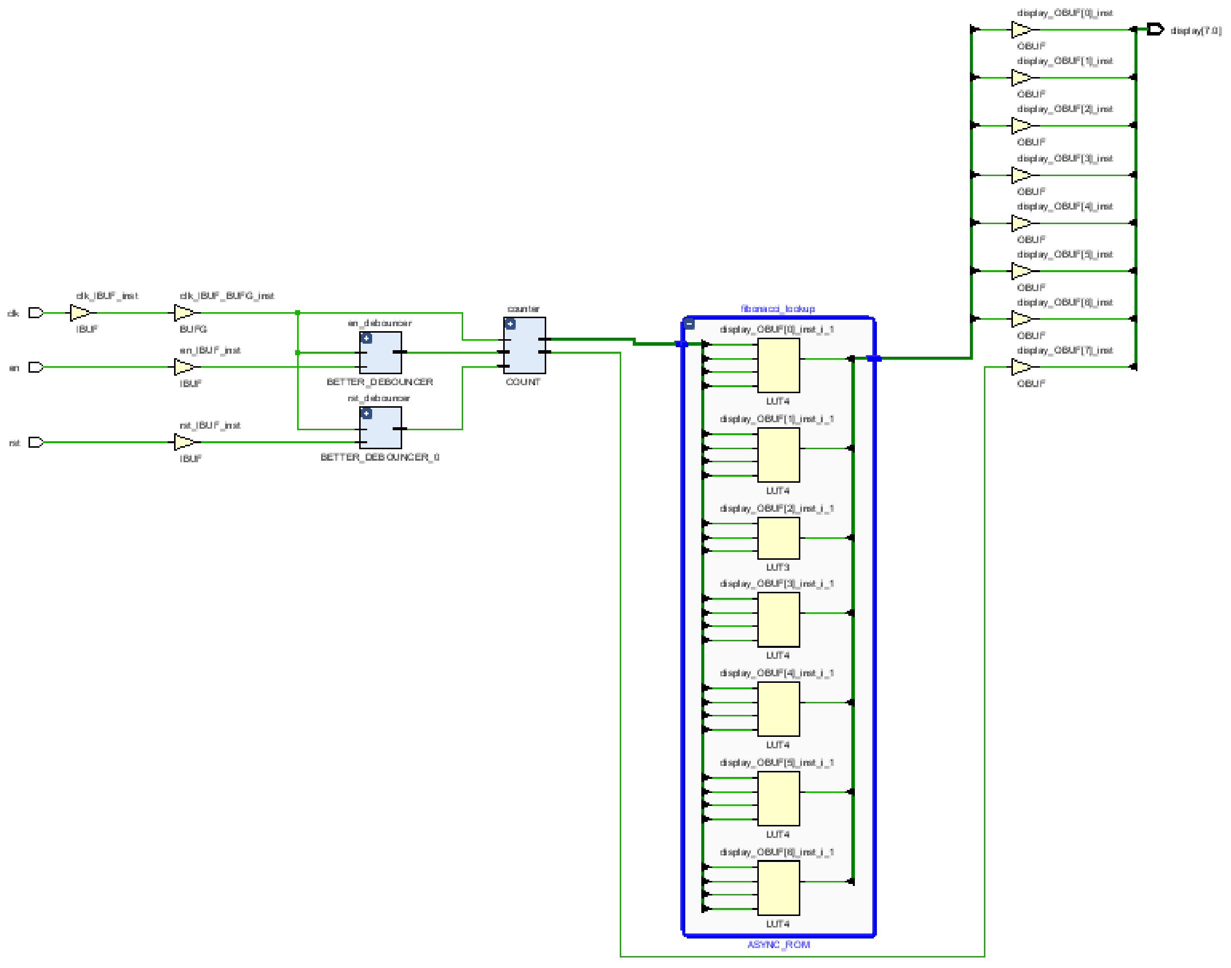




The RTL schematic for the debouncer also matches the code. It is based around a central counter and the inputs to the counter are set by a combination of the debouncer inputs (and in the case of the enable also one of the outputs). The logic for the inputs to the counter also matches what is expected (e.g. en = Sig AND NOT Counter_Output_Is_Max_Value). The outputs and d-type flip-flop are then correctly setup to ensure only a single clock cycle pulse emerges as the debounced signal.

Finally the internals of the counter are also correct, with it consisting of a register and an operation to add 1 each cycle it is enabled, as well as some logic for the enable and reset.

Synthesis Top Level Schematic



The synthesised top-level schematic mostly looks as expected with the basic structure of the circuit matching the design. Notably the ROM has been split up into individual registers this time, which is still as expected but is different to the RTL schematic for the same design. There is an unexpected wire going from the count to the display. I am not sure what that wire is and why it did not appear on the other schematics.