Digital Engineering
Lab 1

Y3890959
Y3878784

24th January 2023

Task A: Debouncer implementation and simulation by parameterization

VHDL Code

Top-Level Entity

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

entity fibonacci_8bit_sequence is
    -- Adjustable value for the # of clock cycles the debouncer counts for.
    -- Default set to 50x10^6 for a debounce of approx. 0.5 seconds for a
    -- 100MHz processor.
    generic (LIMIT : NATURAL := 50000000);
    Port ( clk : in STD_LOGIC;
           --  Count raw user input, translates to 'enable' signal close to the counter
           count : in STD_LOGIC;
           --  Raw user input that when presses will reset the sequence
           reset : in STD_LOGIC;
           --  Fibonacci output that is read from ROM
           output : out STD_LOGIC_VECTOR (7 downto 0));
end fibonacci_8bit_sequence;

architecture Behavioral of fibonacci_8bit_sequence is

-- Internal signals containing the debounced signals for the raw user inputs
signal reset_debounced : STD_LOGIC;
signal enable_debounced : STD_LOGIC;

-- Internal signal containing the output of the counter
-- This output points to a specific location in ROM
signal counter_output_address : UNSIGNED (3 downto 0);

begin
    --  This debouncer is used to debounce the count button input.
    --  When the user clicks the count button, it is debounced and
    --  inputted into the counter as enable.
    enable_input_debouncer : entity work.efficient_debouncer
    generic map (
        LIMIT => LIMIT
    )
    port map (
        clk => clk,
        -- Raw signal is inputted
        input_raw => count,
        -- Debounced signal outputted
        output_debounced => enable_debounced
    );


    -- This debouncer is used to debounce the reset button input.
    -- When the user clicks the reset button, it is debounced and
    -- inputted to the counter as reset.
    reset_input_debouncer : entity work.efficient_debouncer
    generic map (
        LIMIT => LIMIT
    )
    port map (
        clk => clk,
        -- Raw user input is inputted
        input_raw => reset,
        -- DEbounced signal is outputted
        output_debounced => reset_debounced
    );


    -- The debounced reset and count signals are inputted as reset and count.
    -- When count is pressed, the debounced signal comes in as enable and enables
    -- the counter allowing it to increment. When the debounced reset is inputted,
    -- the counter resets back to zero.
    sync_enable_4bit_counter : entity work.sync_enable_4bit_counter
    port map (
        clk => clk,
        -- The debounced signals are inputted (not raw user inputs)
        rst => reset_debounced,
        en => enable_debounced,
        -- The output is mapped to the counter address internal signal
        cnt_out => counter_output_address
    );

    -- The ROM stores the first 14 elements of the Fibonacci sequence.
    -- The output from the counter is inputted as the address into the ROM,
    -- the data at that address is then outputted by the ROM into the output bus.
    fibonacci_8bit_sync_read_rom : entity work.fibonacci_8bit_async_read_rom
    port map (
        -- Output of 4-bit counter is used to access memory item
        address => counter_output_address,
        -- Memory item is outputted into DataOut bus (8-bit Fibonacci sequence)
        DataOut => output
    );

end Behavioral;
```

Efficient (Updated) Debouncer Entity

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

```vhdl
-- The efficient debouncer implements a counter which counts to a defined limit
-- whilst the raw input signal is high and doesn't bounce to low, and the limit
-- hasn't been reached. By defining a limit (# of clock cycles), this design can
-- enforce a duration for the raw input signal to be high before the debounced
-- output of high is generated.
entity efficient_debouncer is
    -- Adjustable value for the # of clock cycles until the counter resets.
    generic (LIMIT : NATURAL := 17);
    Port (
           clk : in STD_LOGIC;
           -- The raw user input is mapped
           input_raw : in STD_LOGIC;
           -- The debounced signal output is mapped
           output_debounced : out STD_LOGIC
    );
end efficient_debouncer;

architecture Behavioral of efficient_debouncer is

-- Internal signal for the inverted raw user input.
signal input_raw_inverted : STD_LOGIC;
-- Internal signal bus for the output from the counter.
signal count_out_int : UNSIGNED (log2(LIMIT)-1 downto 0);
-- Internal signal for the output from the counter max value detector.
signal max_detector : STD_LOGIC;
--Internal signal for the counter enable value.
signal counter_enabler : STD_LOGIC;
-- Internal signal for the output from the flip flop.
signal flip_flop_output : STD_LOGIC;

begin
    -- We need to make sure that the counter resets as soon as the raw input
    -- signal bounces from high to low. An inverter is created such that when
    -- the raw input bounces to low, the 'input_raw_inverted' internal signal
    -- will be high, this signal can then be mapped to the reset port of the
    -- counter.
    input_raw_inverted <= not input_raw;

    -- The other counter reset condition is when the counter reaches the value
    -- defined by 'LIMIT'. The 'max_detector' internal signal will be set high
    -- as soon as the LIMIT is reached, mapping this signal to the counter's
    -- reset port will complete this logic.
    max_detector <= '1' when count_out_int = LIMIT-1 else
                    '0';

    -- As per the debouncer's logic, the counter need to be enabled when the
    -- raw input signal is high and the 'max_detector' internal signal is low
    -- (when the counter's LIMIT hasn't been reached). The counter needs to be
    -- disabled at all other times. To acheive this logic, the 'max_detector'
    -- signal needs to be inverted, and compared with the raw input signal with
    -- an AND gate.
    counter_enabler <= input_raw AND (not max_detector);

    -- The debouncer needs to wait for a number of clock cycles defined by LIMIT
    -- and should only output a debounced high output if there is a high output
    -- from 'max_detector' and a low from the 'max_detector' from the last rising
    -- edge. This is to prevent the debouncer from outputting high due to reaching
    -- the LIMIT value (setting 'max_detector' to high) for two consecutive clock
    -- cycles. This logic is implemented with an AND gate comparing the output from
    -- 'max_detector' and the inverted output from the flip flop.
    output_debounced <= max_detector AND (not flip_flop_output);
    -- The parameterizable counter has been implemented as a separate entity, this
    -- componenent is used to count the number of clock cycles elapsed since the raw
    -- input was set to high.
    parameterizable_counter : entity work.parameterizable_counter
    generic map (
        -- The limit is a parameterizable value allowing for an adjustable number
        -- of clock cycles to count.
        LIMIT => LIMIT
    )
    port map (
        -- (Self explanatory) The clock signal is mapped
        clk => clk,
        -- The counter needs to reset as soon as the raw input bounces to a low.
        rst => input_raw_inverted,
        -- The counter needs to be enabled as per logic defined (and explained)
        -- in the 'counter_enabler' internal signal definition above.
        enable => counter_enabler,
        -- The counter's output needs to be mapped to an internal signal to implement
        -- further logic.
        count_out => count_out_int
    );

    --
    d_type_flip_flop : entity work.d_type_flip_flop
    port map (
        -- (Again, self explanatory)
        clk => clk,
        -- The output from the 'max_detector' signal is mapped to the data input
        -- of the flip flop, this is stored until the next clock rising edge.
        data => max_detector,
        -- The data that was stored in the last clock rising edge is outputted
        -- from the flip flop to the 'flip_flop_output' internal signal to
        -- facilitate additional logic for the 'output_debounced' internal signal
        -- as mentioned above.
        output => flip_flop_output
    );

end Behavioral;
```

Counter Entity

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

-- counter to LIMIT (0 to LIMIT-1) with synchronous reset and enable
entity parameterizable_counter is
    -- Adjustable value to count to before the counter resets.
    -- The counter will count from 0 to LIMIT-1.
    generic (LIMIT : NATURAL := 17);
    port (
        -- 'rst' to reset counter when set high.
        -- 'enable' must be high for counter to count.
        clk, rst, enable : in STD_lOGIC;
        -- A data bus to facilitate an output, bus width is dynamic
        -- based on the 'LIMIT' defined above.
        count_out : out UNSIGNED (log2(LIMIT)-1 downto 0)
    );
```

```
end parameterizable_counter;


architecture Behavioural of parameterizable_counter is
    -- Internal signal for the internal output of the counter
    -- used to count and reset.
    signal count_int : UNSIGNED (log2(LIMIT)-1 downto 0);

begin
counter: process (clk)
begin
    -- When there's a rising clock edge:
    --  If the reset pin is high;
    --      the counter resets to 0
    --  If enable is high:
    --      if the counter output is less than the LIMIT - 1:
    --          then the counter increments.
    --      if the counter output is equal to LIMIT - 1;
    --          the counter resets (rolls-over) to 0
    if rising_edge(clk) then
        if (rst = '1') then                     -- Reset condition
            count_int <= (others => '0');
        elsif (enable = '1') then               -- Enable condition:
            if (count_int = LIMIT-1) then
                count_int <= (others => '0');       -- Roll-over
            else
                count_int <= count_int + 1;         -- Count up
            end if;
        end if;
    end if;
end process counter;

-- The internal signal bus is mapped to the output signal bus.
count_out <= count_int;

end Behavioural;
```

ROM Entity

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

entity fibonacci_8bit_async_read_rom is
    Port ( Address : in UNSIGNED (3 downto 0);  -- Address of data to be retrieved
           DataOut : out STD_LOGIC_VECTOR (7 downto 0)); -- Data that is retrieved
end fibonacci_8bit_async_read_rom;
```

```
architecture Behavioral of fibonacci_8bit_async_read_rom is
--  With 8-bits, a maximum of 14 sequence elements can be stored,
--  any larger (144 + 233) requires more bits.
type ROM_Array is array (0 to 15) of std_logic_vector(7 downto 0);
    constant Content: ROM_Array := (
    0 =>  X"00",                 --  0
    1 =>  X"01",                 --  1
    2 =>  X"01",                 --  1
    3 =>  X"02",                 --  2
    4 =>  X"03",                 --  3
    5 =>  X"05",                 --  5
    6 =>  X"08",                 --  8
    7 =>  X"0D",                 --  13
    8 =>  X"15",                 --  21
    9 =>  X"22",                 --  34
    10 => X"37",                 --  55
    11 => X"59",                 --  89
    12 => X"90",                 --  144
    13 => X"E9",                 --  233
    others => X"00");            --  0

begin
    DataOut <= Content(to_integer(Address));
end Behavioral;
```

Testbenches and Simulation

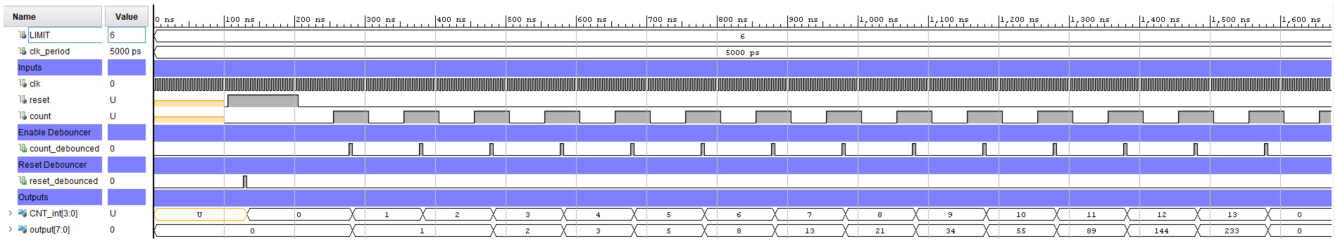
Testbech VHDL Code

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

entity fibonacci_8bit_sequence_tb is
    -- Adjustable value for the # of clock cycles the debouncer counts for.
    -- Default set to 6 for a much quicker debounce, essential for testbench
    -- simulations. The debouncer will activate for a pulse with at least
    -- 5 rising clock edges.
    generic (LIMIT : NATURAL := 6);
end fibonacci_8bit_sequence_tb;

architecture Behavioral of fibonacci_8bit_sequence_tb is

signal clk, reset, count : STD_LOGIC;
signal output : STD_LOGIC_VECTOR (7 downto 0);

constant clk_period : time := 5ns;

begin

    UUT : entity work.fibonacci_8bit_sequence
        GENERIC MAP (
            LIMIT => LIMIT
        )
        PORT MAP (
            clk => clk,
            count => count,
            reset => reset,
            output => output
        );
```

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

--  TEST STRATEGY
--      TEST 1:
--      After a reset, a 'for' loop will be used to toggle the enable
--      signal 19 times, allowing the counter to count to 16, roll-over
--      then count up twice. This is to test the counting behaviour
--      and the roll-over transition.
--
--      TEST 2:
--      We can send a reset pulse lasting for 2 clock rising edges (less
--      than the ~5 needed for the deboucer to activate) to verify that
--      the debouncer does indeed work as intended.
--
--      TEST 3:
--      Following from test 2, a reset signal will be sent to the counter
--      (the reset pulse lasting >5 clock rising edges), the counter
--      should reset back to 0, and the first value of the Fibonacci
--      sequence should output.
--
--      TEST 4:
--      Following from test 3, the counter will be toggled with another
--      'for' loop 10 times, this is to verify whether the counter can
--      count up even after a reset.
--
--      TEST 5:
--      Following from test 4, this test will toggle the reset and count
--      inputs at the same time.

test : process
begin
    wait for 100ns;
    wait until falling_edge(clk);

    --  Initialise counter
    reset <= '0';
    count <= '0';

    wait for clk_period;

    -- Reset counter
    reset <= '1';
    count <= '0';

    wait for clk_period*20;

    --  TEST 1
    count_to_18 : for i in 0 to 18 loop
        reset <= '0';
        count <= '0';
        wait for clk_period*10;
        reset <= '0';
        count <= '1';
        wait for clk_period*10;
        end loop count_to_18;

        --  TEST 2
        count <= '0';
        wait for clk_period*10;
        reset <= '1';
        wait for clk_period*2;
        reset <= '0';

        wait for clk_period*10;

        --  TEST 3
        reset <= '1';
        wait for clk_period*10;

        --  TEST 4
        count_to_12 : for i in 0 to 12 loop
            reset <= '0';
            count <= '0';
            wait for clk_period*10;
            reset <= '0';
            count <= '1';
            wait for clk_period*10;
        end loop count_to_12;

        --  TEST 5
        reset <= '0';
        count <= '0';

        wait for clk_period*5;

        reset <= '1';
        count <= '1';

        wait for clk_period*20;

        reset <= '0';
        count <= '0';

        wait;

    end process;


end Behavioral;
```

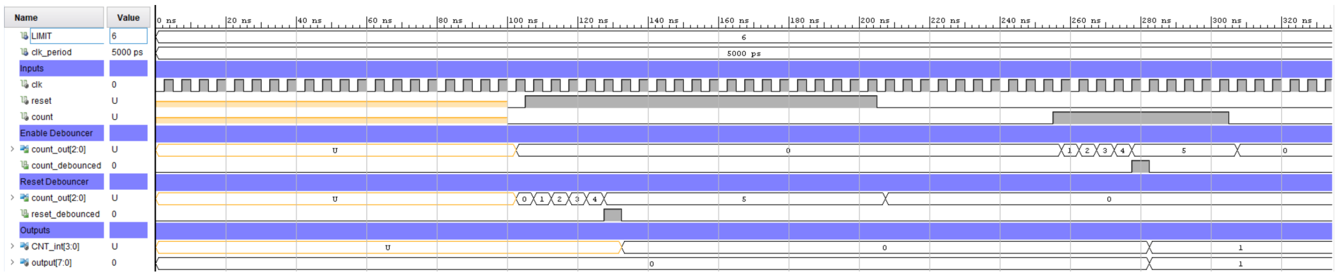
Testbench Waveforms

Entire Sequence



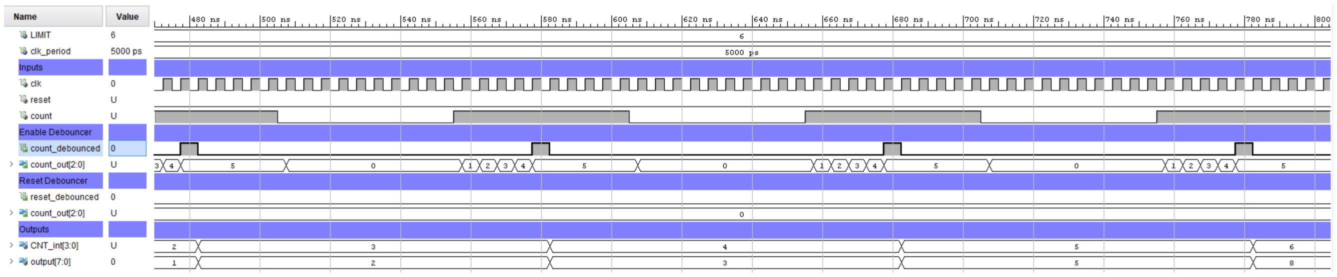
The above waveform screenshot shows the entire sequence from time 0 to output 1 after a roll-over. This waveform is intended to verify that the overall output is correct.

Test 1, Waveform 1 - Initial reset and counting



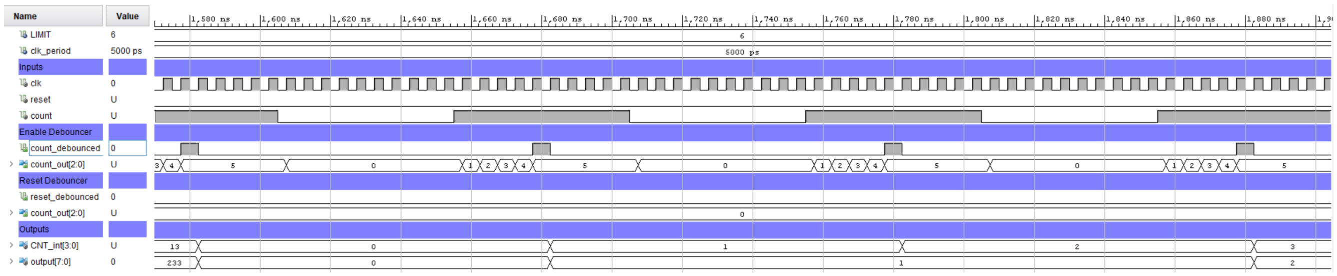
The above waveform shows the initial global reset, verifying a successful activation of both the 'reset' and 'count' debouncers and a successful reset and increment of the overall circuit.

Test 1, Waveform 2 - Counting up



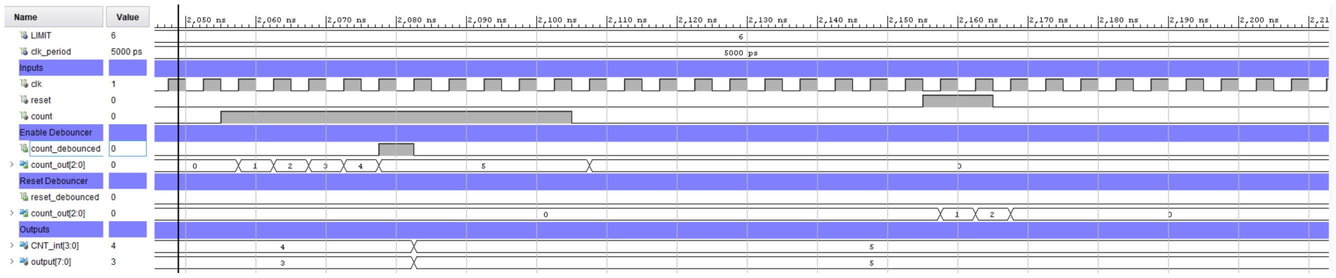
This waveform (a continuation of Test 1, Waveform 1) verifies that the debouncer is activating and the counter increments as designed, the Fibonacci sequence is shown (1,2,3,5,8).

Test 1, Waveform 3 - Roll-over and count up



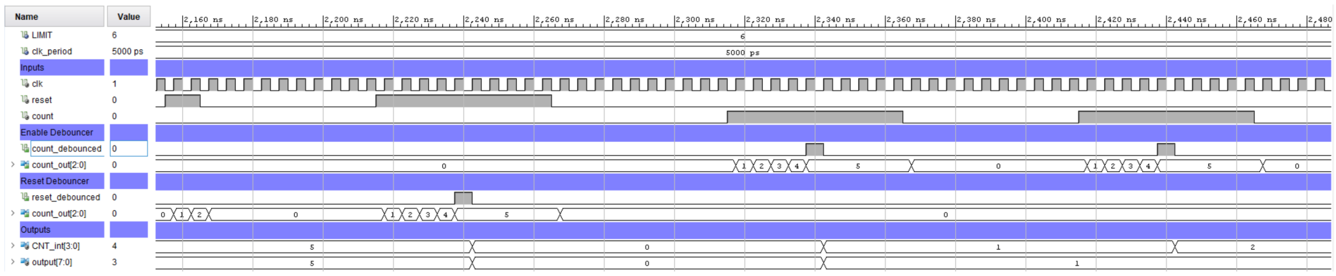
This waveform verifies that the maximum Fibonacci sequence value is reached, and the sequence rolls over to 0 without a reset signal being generated by the user and can resume counting up.

Test 2 - Quick reset pulse



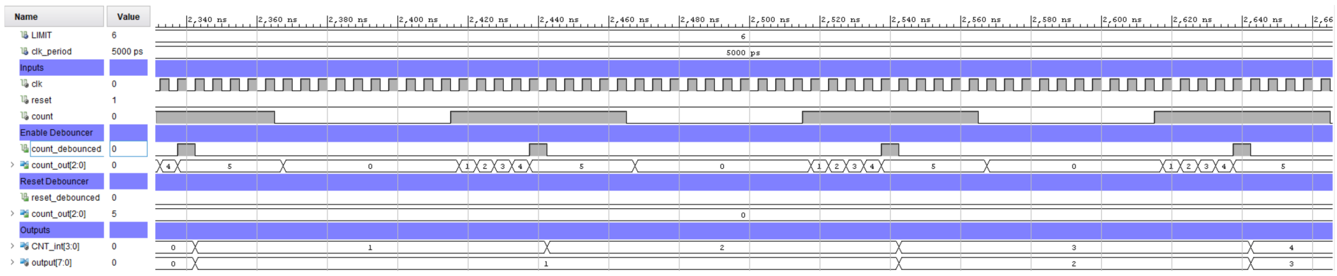
This waveform verifies that the debouncer is functioning as intended by showing the parameterizable counter on board the debouncer incrementing when the reset signal is high but resetting as soon as it goes low before the 'LIMIT' is reached, thus invalidating the reset input.

Test 3 - Longer reset pulse



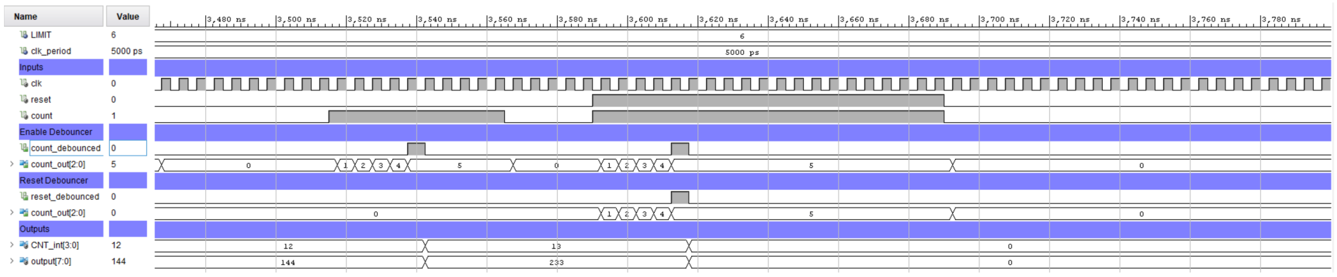
This waveform verifies that the debouncer successfully activates with a longer reset pulse of a valid clock duration. The counter in the debouncer increments with each clock cycle while the reset is high, and outputs a high debounced output once 'LIMIT' is reached.

Test 4 - Resume after mid-operation reset



This waveform verifies that the circuit successfully resumes counting after a mid-operation reset following Test 3.

Test 6 - Holding reset and count



Circuit Analysis and Synthesis

RTL Componenet Statistics

```
Detailed RTL Component Info :
+---Adders :
           2 Input     26 Bit       Adders := 2
           2 Input      4 Bit       Adders := 1
+---Registers :
                       26 Bit    Registers := 2
                        4 Bit    Registers := 1
                        1 Bit    Registers := 2
+---Muxes :
           2 Input     26 Bit        Muxes := 2
           2 Input      4 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 2
```

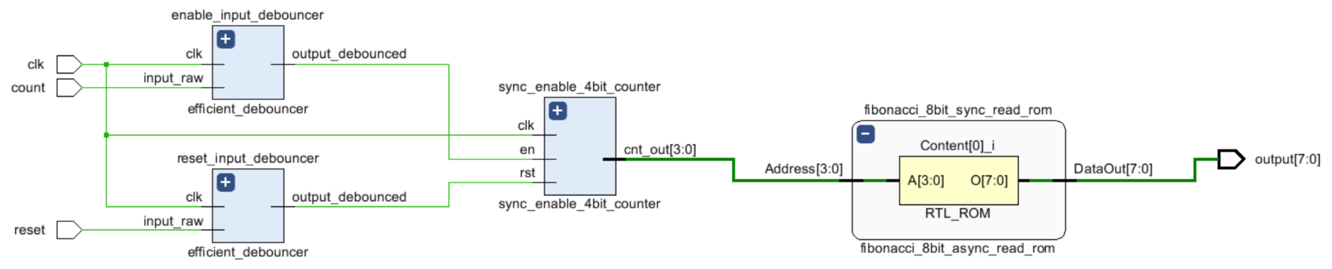
RTL Hierarchical Component Statistics

```
Hierarchical RTL Component report
Module parameterizable_counter
Detailed RTL Component Info :
+---Adders :
           2 Input     26 Bit       Adders := 1
+---Registers :
                       26 Bit    Registers := 1
+---Muxes :
           2 Input     26 Bit        Muxes := 1
Module d_type_flip_flop
Detailed RTL Component Info :
+---Registers :
                        1 Bit    Registers := 1
Module efficient_debouncer
Detailed RTL Component Info :
+---Muxes :
           2 Input      1 Bit        Muxes := 1
Module sync_enable_4bit_counter
Detailed RTL Component Info :
+---Adders :
           2 Input      4 Bit       Adders := 1
+---Registers :
                        4 Bit    Registers := 1
+---Muxes :
           2 Input      4 Bit        Muxes := 1
```

From the circuit design phase of this lab, we developed the ‘LIMIT’ parameter to have a size of 50,000,000 so that when running on a 1MHz FPGA, inputs lasting less than 0.5 seconds would be ignored. As the 'parameterizable_counter' entity on board each debouncer will have to count up to the 'LIMIT' parameter, the entity will need to be 26 bits in size ($\log_2(50,000,000) = 25.58 \approx 26$). So it makes sense that the counter has been synthesised as 26 bits in size.

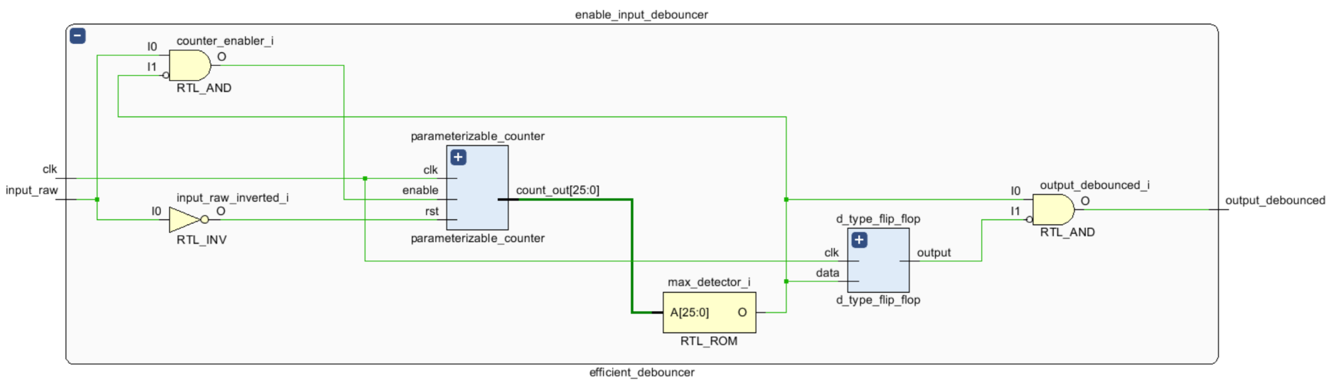
Schematics

RTL Top Level (ROM Expanded)



As designed, there's a debouncer each for the 'reset' and 'count' raw input signals. These debounced signals are what control the 4-bit counter which accesses the Fibonacci ROM.

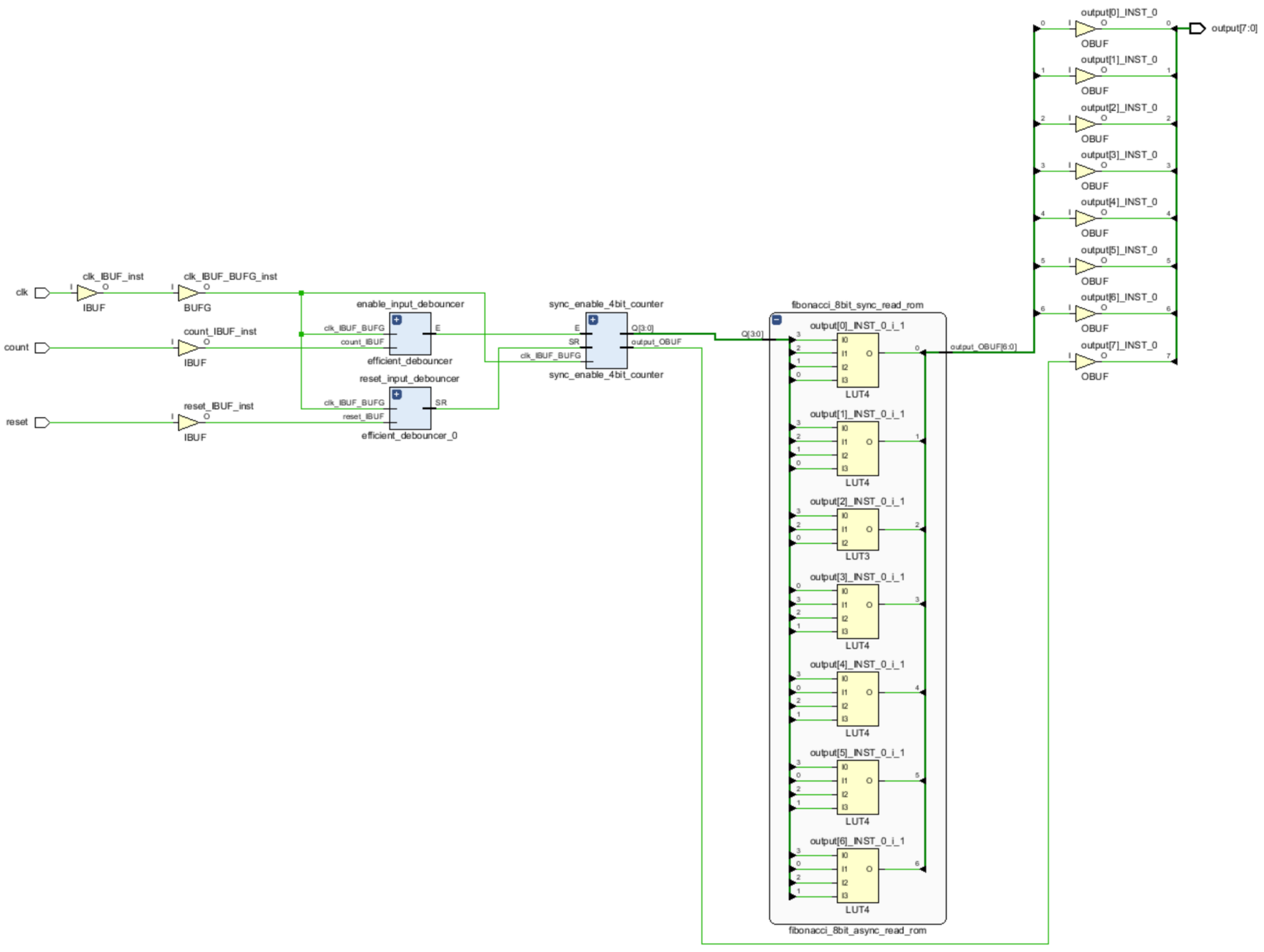
RTL Debouncer



The clock signal is input along with the raw input for debouncing. Whilst the raw input is high and the max value ('LIMIT') hasn't been reached, the counter will increment. Once the 'LIMIT' has been reached, 'enable' goes low, which stops the counter from incrementing. The counter is reset once the raw input goes low.

When the 'LIMIT' has been reached, a high signal output from the 'max_detector' is stored in the d-type flip flop. The same output is compared with an AND gate with an inverted output from the d-type flip flop, this is to ensure the debouncer doesn't activate for more than one clock cycle.

Synthesis Top Level (ROM Expanded)



There are input buffers for each input pin. The clock pin has an additional buffer (BUFG) which is used to distribute high fan-out clock signals. The ROM is synthesised as an array of look-up tables. And finally, each signal in the output bus has an output buffer.