PULSE GENERATOR

Electronic Systems' Project

Abstract

Design and development of a pulse generator hardware using Xilinx FPGA Zync platform

MARCO PETTORALI – A.A. 2019-2020

Index

Introduction 2
Functional requirements 2
Non-functional requirements 2
Possible employments 2
Architectural design 3
Input and output ports 3
Logic scheme 3
Registers 3
Signals used 4
VHDL code 5
dff.vhd – D Flip Flop VHDL description 5
pulsegen.vhd – Pulse Generator VHDL description 6
Testing 9
Test plan 9
Tested cases 11
pulsegen_tb.vhd – Testbench description 12
Results of the tests 16
Synthesis 21
Vivado's elaborated design 21
Timing constraints 21
Synthesis warnings 21
Setup and hold slacks 21
Critical path 22
Utilization report 23
Power report 23
Conclusions 24

Introduction

Functional requirements

The pulse generator that we want to design is a device that, given a single-clock-long pulse on the input, switches between two modes: the first one makes the output to be constant, keeping the previous value; the second one makes the output switch between the high and the low logic value at each clock cycle, producing some pulses.

If a pulse long more than one clock cycle is given to the input, then the output has to go to ‘0’, and an error signal must be raised to ‘1’. This status of the device (including the last mode in which the system was before the error) is kept until the next right pulse (the one during exactly one clock cycle) given to the input.

Non-functional requirements

Since the device has to recognize one-clock-long pulses, the device has to be synchronized by an external clock signal, and a reset signal must be used to initialize it.

For a coherence purpose, we want the two outputs of the system (out and error signals) to respond to the input at the same time.

No particular time constraints are requested for this application; in fact, when an input is given we cannot expect the output to immediately respond. However, we have to design the circuit in a way that this response time is minimized.

Possible employments

A pulse generator can be used for several applications. Since its main functionality is that to provide rectangular pulses, it can be used to build a testing framework for electronic devices, to synchronize a circuit, to halve the clock frequency (given an input clock we obtain a clock signal on the output that has exactly the double of the period of the first one).

Architectual design

Input and output ports

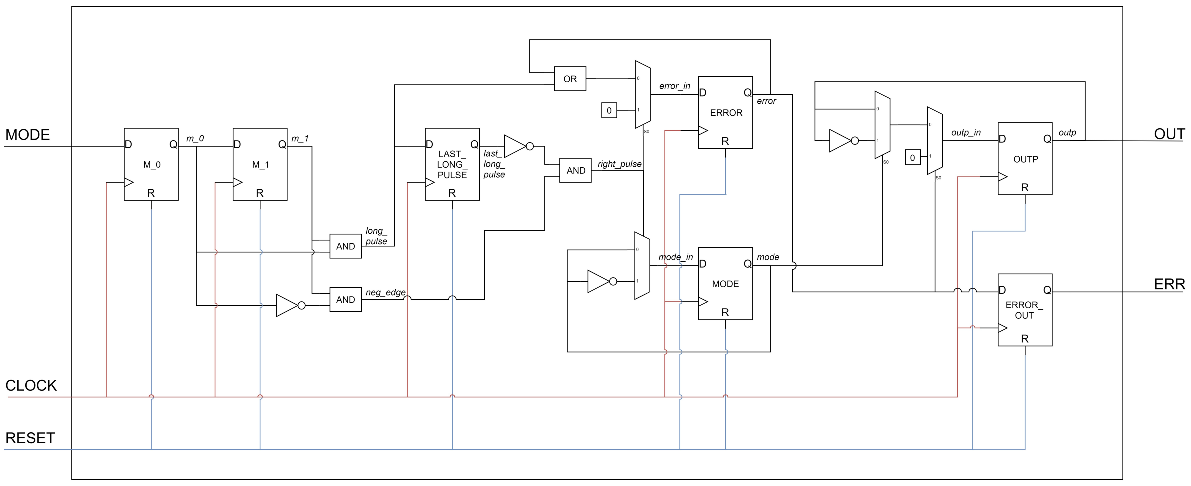
Input ports:

- Mode: it is the main input of the device; pulses are given to this input;
- Clock: external clock signal to synchronize the device;
- Reset: external reset signal to initialize the device.

Output ports:

- Out: it is the main output of the device, it can change each clock cycle or stay constant according to the input given on the 'Mode' port;
- Error: it is the error flag signal; it raises to one whenever a not valid pulse is fed to the 'Mode' port.

Logic scheme



Registers

- ***M_0***: it stores the value of the input given at the current clock cycle;
- ***M_1***: it stores the value of the input given at the previous clock cycle;
- ***LAST_LONG_PULSE***: it contains a '1' if a not valid pulse was detected at the previous clock cycle, '0' otherwise;
- ***ERROR***: it contains '1' if a not valid pulse was detected, and stay to the high logic value until a right pulse is detected, then it contains a '0';

- ***MODE***: it contains ‘0’ if the device is in “constant output” mode, ‘1’ if it’s in the “switching output” mode;
- ***OUTP***: it’s the register holding the ‘Out’ port;
- ***ERROR_OUT***: it is the register holding the ‘Err’ port; it’s introduced to make the Err output go to 1 at the same time at which the Out port goes to ‘0’ after an error is detected.

Signals used

- ***m_0***: it’s the signal at the output of M_0 register;
- ***m_1***: it’s the signal at the output of M_1 register;
- ***long_pulse***: it’s the signal detecting a pulse during more than one clock cycle. In fact, it is the and between m_0 and m_1, so it means that if they’re both to ‘1’, the current and the previous value of the input are ‘1’, so the input is seeing a pulse during at least two clock cycles;
- ***neg_edge***: it’s the signal detecting the negative edge of the pulse given to the input. In fact, it is the and between m_1 and not m_0, so it means that neg_edge is ‘1’ if the previous value of the input is ‘1’ and the current one is ‘0’: a pulse has just finished.
- ***last_long_pulse***: it’s the signal at the output of LAST_LONG_PULSE register;
- ***right_pulse***: it’s the signal detecting each one-clock-long pulse. In fact, it is the and between neg_edge and last_long pulse, so it means that if a negative edge on the input is found (neg_edge is ‘1’) and before that edge no errors were found (last_long_pulse is ‘0’) then the last pulse must be a good pulse;
- ***error_in:*** it’s the input of the ERROR register; when no right pulses are found, the ERROR register must contain the same value as before or ‘1’, if a long pulse is detected. When a right pulse is found, the ERROR register must immediately go to ‘0’;
- ***error***: it’s the signal at the output of the ERROR register;
- ***mode_in***: it’s the signal at the input of the MODE register; when a right pulse is detected, the mode of the system has to be switched, otherwise is kept the same value of the mode;
- ***mode***: it’s the signal at the output of the MODE register;
- ***outp_in***: it’s the signal at the input of the OUTP register; if no error is detected and the mode of the system is ‘0’, then the output stay constant. If no error is detected and the mode of the system is ‘1’, then the output has to switch from high to low value at each clock; otherwise, if an error is detected, the OUTP register must immediately go to ‘0’.

VHDL code

dff.vhd – D Flip Flop VHDL description

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

entity DFF is
    port(
        d: in std_logic;
        clk: in std_logic;
        rst: in std_logic;
        q: out std_logic);
end DFF;

architecture rtl of DFF is
begin
    dff_proc: process(clk, rst)
    begin
--      if the reset signal is '0', the output has to go to '0'
        if(rst='0') then
            q <= '0';
--      if the reset is not '0' and a positive edge of the edge is
found
--      then the output takes the value of the input
        elsif(rising_edge(clk)) then
            q <= d;
        end if;
    end process;
end rtl;
```

pulsegen.vhd – Pulse Generator VHDL description

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

entity pulsegen is
    port(
        m : in std_logic;
        clock : in std_logic;
        reset : in std_logic;
        o : out std_logic;
        err : out std_logic);
end pulsegen;

architecture rtl of pulsegen is

--  including d-flip-flop component
    component dff is
        port(
            d: in std_logic;
            clk: in std_logic;
            rst: in std_logic;
            q: out std_logic);
    end component;

--  defining all the needed signals
    signal m_0 : std_logic;
    signal m_1 : std_logic;
    signal neg_edge : std_logic;
    signal long_pulse : std_logic;
    signal right_pulse : std_logic;
    signal last_long_pulse : std_logic;
    signal error_in : std_logic;
    signal error : std_logic;
    signal mode_in : std_logic;
    signal mode : std_logic;
    signal outp_in : std_logic;
    signal outp : std_logic;
    signal outp_mux_in : std_logic_vector(1 downto 0);

begin
--  mapping each register's port to the signals of the pulse
generator
    m_0_reg:dff
        port map(
            d => m,
            clk => clock,
            rst => reset,
            q => m_0
        );
    m_1_reg:dff
        port map(
            d => m_0,
            clk => clock,
            rst => reset,
            q => m_1
        );

    last_long_pulse_reg:dff
        port map(
            d => long_pulse,
            clk => clock,
            rst => reset,
            q => last_long_pulse
        );

    error_reg:dff
        port map(
            d => error_in,
            clk => clock,
            rst => reset,
            q => error
        );

    mode_reg:dff
        port map(
            d => mode_in,
            clk => clock,
            rst => reset,
            q => mode
        );

    outp_reg:dff
        port map(
            d => outp_in,
            clk => clock,
            rst => reset,
            q => outp
        );

    error_out_reg:dff
        port map(
            d => error,
            clk => clock,
            rst => reset,
            q => err
        );

--  neg_edge detects a negative edge on the input, checking the
current and previous value of the input
    neg_edge <= m_1 and not(m_0);
```

```
--  long_pulse detects a pulse longer than one clock cycle
    long_pulse <= m_0 and m_1;

--  right_pulse checks if the last pulse given to the input is
good or not.
--  It checks if a negative edge is found and if it's not
associated to a long pulse
    right_pulse <= neg_edge and not(last_long_pulse);

--  error_in is the input of error_reg register. It resets the
error register if a good pulse is found,
--  or else keeps the previous value of the error signal or sets
the error register to one if a long pulse is detected
    error_in <= '0' when right_pulse = '1' else
                long_pulse or error;

--  mode_in is the input of mode_reg register. It switches when a
right pulse is detected, else it makes the mode_reg to
--  keep the same value
    mode_in <= not(mode) when right_pulse = '1' else
               mode;

--  outp_mux_in is the input to the multiplexer before the
outp_reg register. The most significant bit is the error signal
    outp_mux_in <= error & mode;

--  outp_in is the input of outp_reg register. If no error is
found: if the mode of the system is '0', then the output is kept
--  constant, otherwise the output is swiched at each clock cycle.
If an error is detected, the output goes to '0'.
    outp_in <= outp when outp_mux_in = "00" else
               not(outp) when outp_mux_in = "01" else
               '0';

--  the output of the device is the output of the outp_reg
register
    o <= outp;

end rtl;
```

Testing

Test plan

The system must respond correctly to different use cases; from the ones in which inputs are given correctly, in which the output has to change accordingly, to the ones with long pulses. In that case, when the situation of error is finished, everything has to appear as it was before the error.

To accurately test each case in which the system can end up, it's necessary to understand which are the possible combination of pulses that can be fed to the input. After enumerating all of them, we can choose which one of them is really interesting to check if the system works as it has to.

There are two kind of pulses: the 'right' pulses (those whose length is one clock cycle) and the 'wrong' ones (the others). At first, we have to test if the right and the wrong pulses alone work as expected; moreover, we have at least to check that a 2-clock-cycles-long pulse works the same as an N-clock-cycles-long one. Those tests are respectively number 1, 4 and 5 of the below table of test cases.

The two kinds of pulses can be put in any sequence, and there are four ways to do this:

1. Every pulse is a right pulse (R-…-R);
2. Every pulse is a wrong pulse (W-…-W);
3. There's at least one wrong pulse after a right pulse (…-W-R-…);
4. There's at least one right pulse after a wrong pulse (…-R-W-…);

With the …-W-R-… sequence we are considering also how the system reacts when it recovers from an error.

Note that every other combination of pulses can be traced back to a combination of the ones listed above. For instance, testing the sequence Right – Wrong – Wrong, it's equivalent to test the sequence R-W and the sequence W-W. This is because this system does not keep trace of the past pulses, except for the one that is being fed to the device at a certain instant, so it's physically impossible to have different behaviors for the two cases listed before in the example. We are going to test one similar case to be sure that everything works fine (test 9).

So, we test the 'W-W' sequence in test 6, the 'R-W' one in test 7, the 'W-R' one in test 8. To test the 'R-R' sequence we take the opportunity to check what happens when we pass from the switching mode to the constant one, and we want to get both a ‘0’ and a ‘1’. So two tests are made for ‘R-R’, in tests 2 and 3; one with an odd number of clock cycles in between the two pulses, the other one with an even number of clock cycles.

Moreover, it’s important to check that everything works fine when multiple pulses are given with a single clock cycle of space between them. We will call those tests *N’*, where *N* is the number of a test in the test plan table. Again, it’s not interesting to check all the possible combinations of spacing in between pulses because this device, as it’s designed, cares only about the state of the system at a certain point and the type of the pulse that’s being fed to it at a certain moment.

Cases 1, 4 and 5 don’t obviously admit the *N’* form, neither do case 2, because it requires an even number of clock cycles in between the two pulses.

#	Sequence	Input	Expected response
1	R	1 single clock pulse	Before the pulse, the output is constant (‘0’ in this case); then it begins switching between ‘0’ and ‘1’.
2	R-R	1 single clock pulse + 1 single clock pulse after an odd number of clock cycles	After the first pulse, the output is switching; then, after the second pulse, the output stabilizes on the ‘0’ value.
3	R-R	1 single clock pulse + 1 single clock pulse after an even number of clock cycles	After the first pulse, the output is switching; then, after the second pulse, the output stabilizes on the ‘1’ value.
4	W	1 pulse during 2 clock cycles	Before the pulse, the output is constant; after the second pulse, the output goes to ‘0’, while the error flag goes to ‘1’. Those changings to the outputs happen at the same time.
5	W	1 pulse during 5 clock cycles	Same behavior as use case #4.
6	W-W	1 pulse during 2 clock cycles + 1 pulse during 5 clock cycles	After the first wrong pulse, the output goes to ‘0’ and the error flag goes to ‘1’. The second pulse is ignored.
7	R-W	1 single clock pulse + 1 pulse during 5 clock cycles	After the first pulse, mode switches to ‘1’ and the output star switching between ‘0’ and one. After

			the wrong pulse output goes to ‘0’ and the error flag raises to ‘1’.
8	W-R	1 pulse during 2 clock cycles + 1 single clock pulse	After triggering an error, the second right pulse makes the error flag go to ‘0’ and the output switch between ‘0’ and ‘1’. In fact, at first the system mode is ‘0’ (constant). The first wrong pulse is ignored. The second pulse switches the mode from ‘0’ to ‘1’ (switching).
9	R-W-W-R	1 single clock pulse + 1 pulse during 3 clock cycles + 1 pulse during 5 clock cycles + 1 single clock pulse	After the first pulse, the output start to switch. The second (wrong) pulse makes the output go to ‘0’ and the error flag go to ‘1’. The next wrong pulse is ignored. The last pulse is correct, so the mode switches to ‘0’, the output is kept constant (to ‘0’) and the error goes to ‘0’.

Tested cases

In the test bench written for testing the pulse generator, only some of the test plan cases will actually be implemented. This is because some of them are really not so interesting after trying some others:

- Testing case 1 is useless if we test at least one between case 2 and 3;
- Testing case 2 is not interesting if we test case 3;
- Testing case 2’ is interesting to see if the system rolls immediately back to its original state; so testing case 3’ is not interesting anymore;
- We can choose one between cases 6 and 6’, because in both cases the second pulse will always be ignored; we choose case 6’ because it’s more restrictive;
- Case 9’, in which every pulse is spaced from the others by a single clock cycle, is not a good test, because it’s even too specific; it’s better to test case 9 mixing some 1 clock cycle spacing with N clock cycle spacing.

In the end we come up with the following implemented test cases: 3, 4, 5, 7, 8, 9, 2’, 6’, 7’, 8’.

pulsegen_tb.vhd – Testbench description

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

entity pulsegen_tb is
end pulsegen_tb;

architecture test of pulsegen_tb is

    constant T_CLK : time := 8 ns;

    signal mode_tb : std_logic;
    signal clk_tb : std_logic := '0';
    signal rst_tb : std_logic := '0';
    signal o_tb : std_logic;
    signal err_tb : std_logic;

    signal end_sim_tb : std_logic := '0';

    component pulsegen is
        port(
            m : in std_logic;
            clock : in std_logic;
            reset : in std_logic;
            o : out std_logic;
            err : out std_logic);
    end component;

    begin

        clk_tb <= not(clk_tb) or end_sim_tb after T_CLK/2;
        end_sim_tb <= '1' after T_CLK*250;

        dut:pulsegen
            port map(
                m => mode_tb,
                clock => clk_tb,
                reset => rst_tb,
                o => o_tb,
                err => err_tb
            );

        test_proc: process(clk_tb)
            variable t : integer := 0;
            begin
            if (rising_edge(clk_tb)) then
                case(t) is
```

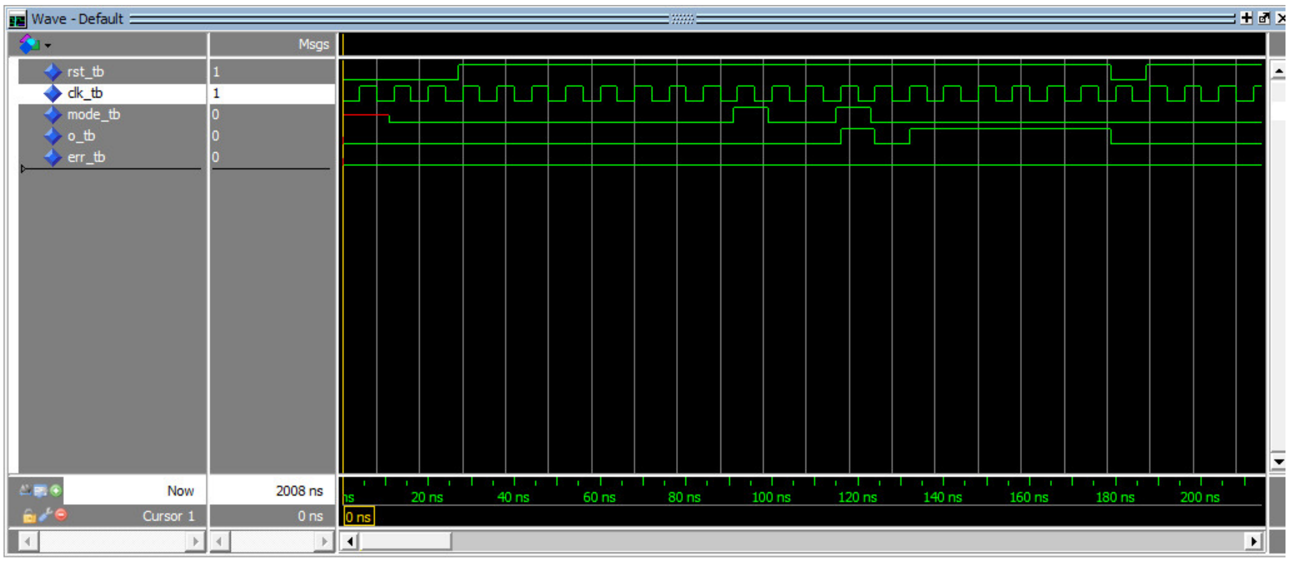
```
-- NOTE: each input is given after 7 ns, without loss of
generality, to better visualize
-- the instant at which a certain input is actually read. This is
because if an input is given
-- precisely on the clock edge, it will be seen only at the next
edge. Moreover, having the signal
-- change at the same instant of the clock is not realistic.

                    when 0 => mode_tb <= '0' after 7 ns;

--                  CASE 3 - R-R
--                  reset
                    when 1 => rst_tb <= '0' after 7 ns;
                    when 2 => rst_tb <= '1' after 7 ns;

--                  right pulse
                    when 10 => mode_tb <= '1' after 7 ns;
                    when 11 => mode_tb <= '0' after 7 ns;
--                  right pulse after even spacing
                    when 13 => mode_tb <= '1' after 7 ns;
                    when 14 => mode_tb <= '0' after 7 ns;

--                  -------------------------------------------

--                  CASE 4 - W
--                  reset
                    when 21 => rst_tb <= '0' after 7 ns;
                    when 22 => rst_tb <= '1' after 7 ns;

--                  wrong pulse - 2 clocks
                    when 30 => mode_tb <= '1' after 7 ns;
                    when 32 => mode_tb <= '0' after 7 ns;

--                  -------------------------------------------

--                  CASE 5 - W
--                  reset
                    when 41 => rst_tb <= '0' after 7 ns;
                    when 42 => rst_tb <= '1' after 7 ns;

--                  wrong pulse - 5 clocks
                    when 50 => mode_tb <= '1' after 7 ns;
                    when 55 => mode_tb <= '0' after 7 ns;

--                  -------------------------------------------

--                  CASE 7 - R-W
--                  reset
                    when 61 => rst_tb <= '0' after 7 ns;
                    when 62 => rst_tb <= '1' after 7 ns;

--                  right pulse
```

```
                    when 66 => mode_tb <= '1' after 7 ns;
                    when 67 => mode_tb <= '0' after 7 ns;

--                  wrong pulse - 5 clocks
                    when 70 => mode_tb <= '1' after 7 ns;
                    when 75 => mode_tb <= '0' after 7 ns;

--                  -------------------------------------------

--                  CASE 8 - W-R
--                  reset
                    when 81 => rst_tb <= '0' after 7 ns;
                    when 82 => rst_tb <= '1' after 7 ns;

--                  wrong pulse - 2 clocks
                    when 90 => mode_tb <= '1' after 7 ns;
                    when 92 => mode_tb <= '0' after 7 ns;

--                  right pulse
                    when 96 => mode_tb <= '1' after 7 ns;
                    when 97 => mode_tb <= '0' after 7 ns;


--                  -------------------------------------------

--                  CASE 9 - R-W-W-R
--                  reset
                    when 105 => rst_tb <= '0' after 7 ns;
                    when 106 => rst_tb <= '1' after 7 ns;

--                  right pulse
                    when 111 => mode_tb <= '1' after 7 ns;
                    when 112 => mode_tb <= '0' after 7 ns;

--                  wrong pulse - 3 clocks
                    when 114 => mode_tb <= '1' after 7 ns;
                    when 117 => mode_tb <= '0' after 7 ns;

--                  wrong pulse - 5 clocks
                    when 119 => mode_tb <= '1' after 7 ns;
                    when 124 => mode_tb <= '0' after 7 ns;

--                  right pulse after 1 clock cycle
                    when 125 => mode_tb <= '1' after 7 ns;
                    when 126 => mode_tb <= '0' after 7 ns;

--                  -------------------------------------------

--                  CASE 2' - R-R
--                  reset
                    when 130 => rst_tb <= '0' after 7 ns;
                    when 131 => rst_tb <= '1' after 7 ns;
```

```
--                   right pulse
                     when 140 => mode_tb <= '1' after 7 ns;
                     when 141 => mode_tb <= '0' after 7 ns;
--                   right pulse after 1 clock cycle
                     when 142 => mode_tb <= '1' after 7 ns;
                     when 143 => mode_tb <= '0' after 7 ns;
--                   -------------------------------------------
--                   CASE 6' - W-W
--                   reset
                     when 150 => rst_tb <= '0' after 7 ns;
                     when 151 => rst_tb <= '1' after 7 ns;

--                   wrong pulse - 2 clocks
                     when 160 => mode_tb <= '1' after 7 ns;
                     when 162 => mode_tb <= '0' after 7 ns;
--                   right pulse - 5 clocks after 1 clock cycle
                     when 163 => mode_tb <= '1' after 7 ns;
                     when 168 => mode_tb <= '0' after 7 ns;
--                   -------------------------------------------
--                   CASE 7' - R-W
--                   reset
                     when 180 => rst_tb <= '0' after 7 ns;
                     when 181 => rst_tb <= '1' after 7 ns;

--                   right pulse
                     when 190 => mode_tb <= '1' after 7 ns;
                     when 191 => mode_tb <= '0' after 7 ns;

--                   wrong pulse - 5 clocks after one clock cycle
                     when 192 => mode_tb <= '1' after 7 ns;
                     when 197 => mode_tb <= '0' after 7 ns;

--                   -------------------------------------------
--                   CASE 8' - W-R
--                   reset
                     when 210 => rst_tb <= '0' after 7 ns;
                     when 211 => rst_tb <= '1' after 7 ns;

--                   wrong pulse - 2 clocks
                     when 220 => mode_tb <= '1' after 7 ns;
                     when 222 => mode_tb <= '0' after 7 ns;

--                   right pulse after 1 clock cycle
                     when 223 => mode_tb <= '1' after 7 ns;
                     when 224 => mode_tb <= '0' after 7 ns;

                     when others => null;
                 end case;
                 t := t+1;
             end if;
         end process;
end test;
```

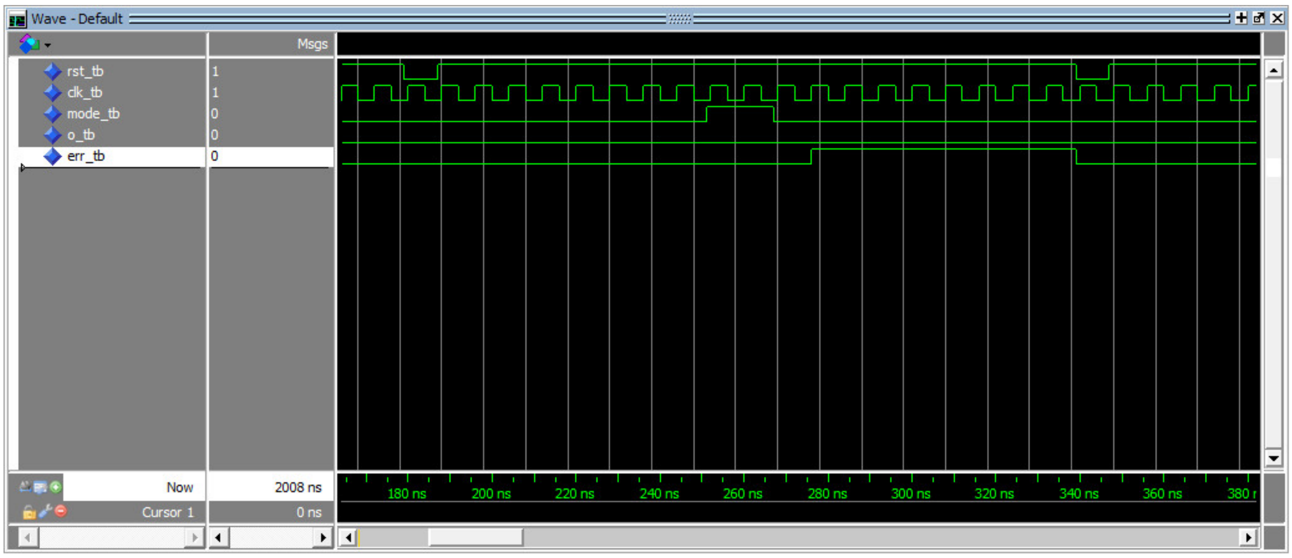
Results of the tests

Case 3



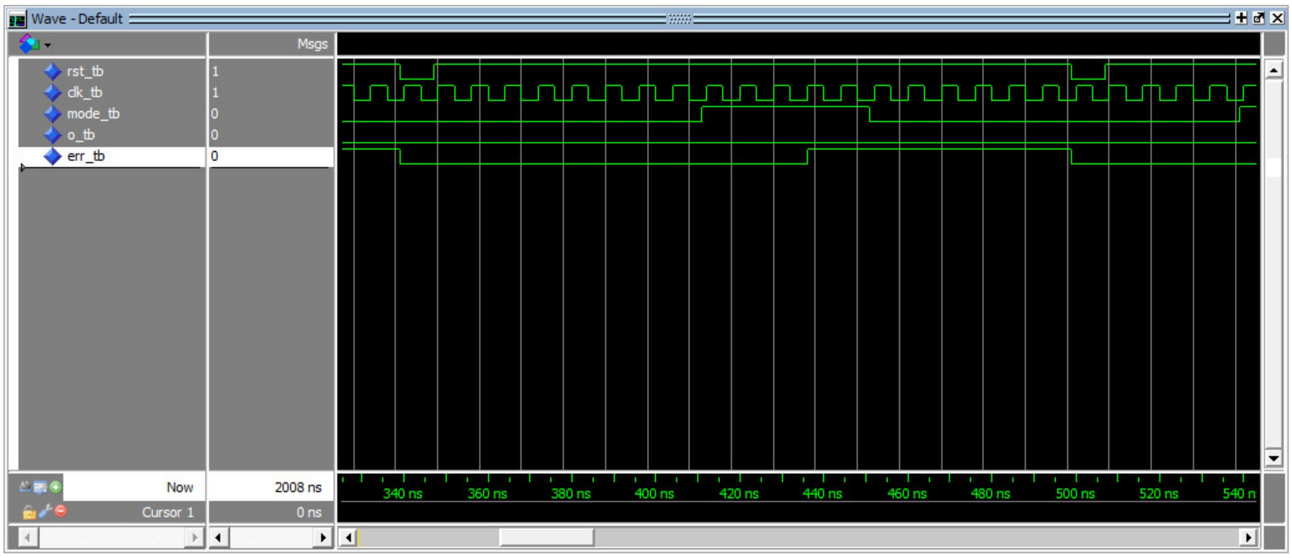
After the first pulse, the output begins to switch from ‘0’ to ‘1’; after the second pulse, the device comes back at mode ‘0’ and the output is constant at level ‘1’.

Case 4



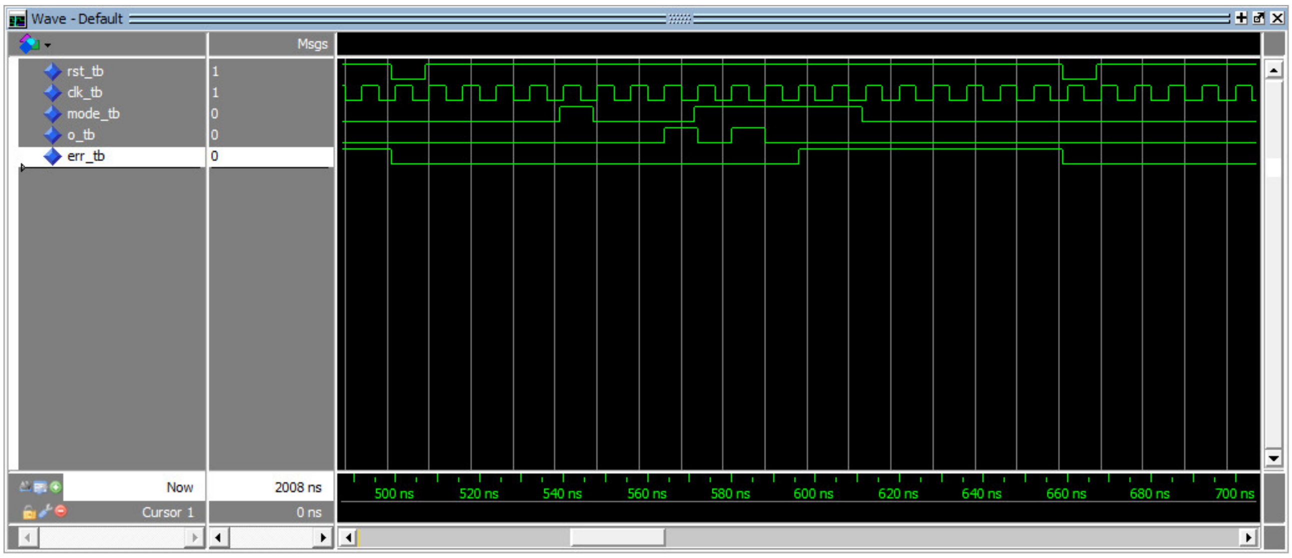
After the two-clock-cycles long pulse, the error flag is raised and the output is constant to ‘0’.

Case 5



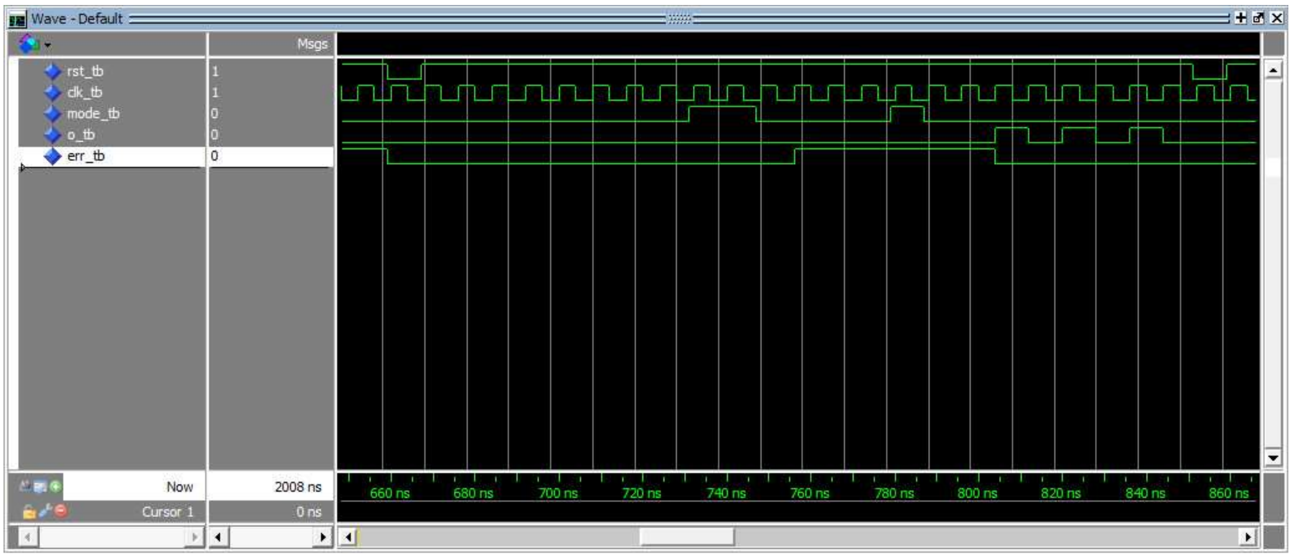
After the five-clock-cycles long pulse, the error flag is raised and the output is constant to ‘0’.

Case 7



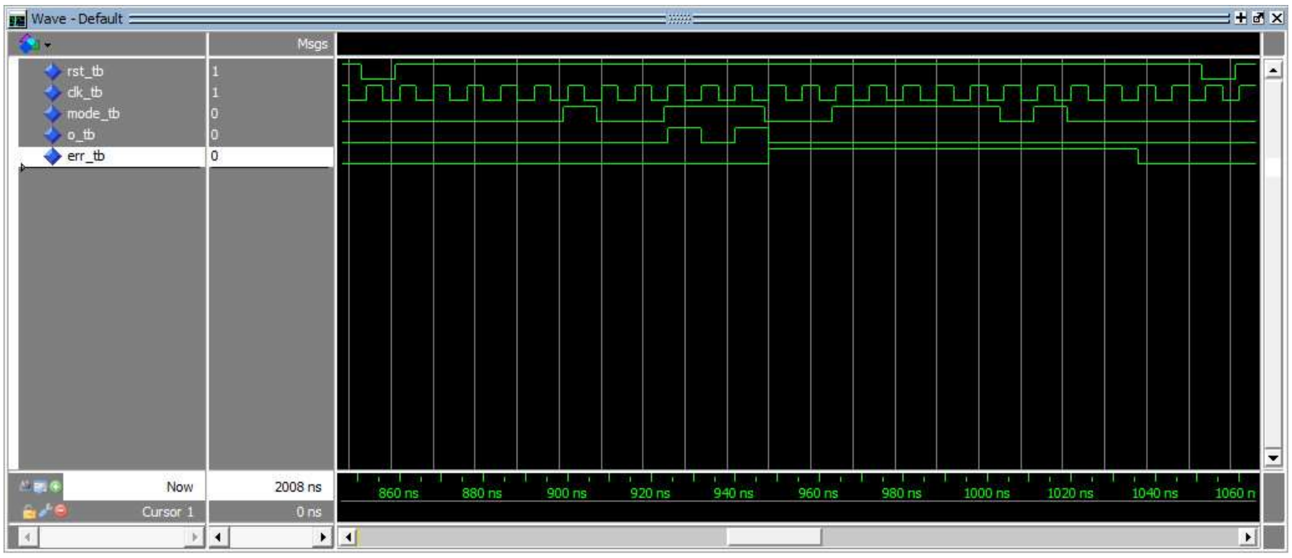
After the first pulse, the output begins to switch; then, after the wrong pulse, an error is raised and the output is kept on ‘0’.

Case 8



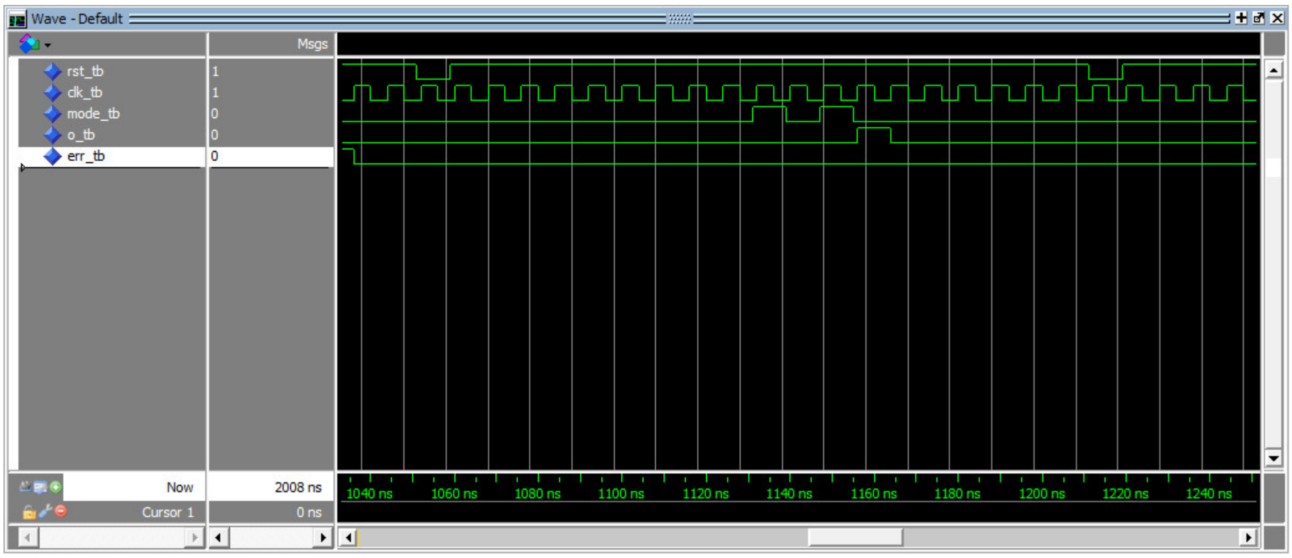
After the first (wrong) pulse, an error is raised; then after the right pulse, the error is recovered and the mode is set to “switching”. Note that in the exact moment at which the output start switching, the error flag is reset.

Case 9



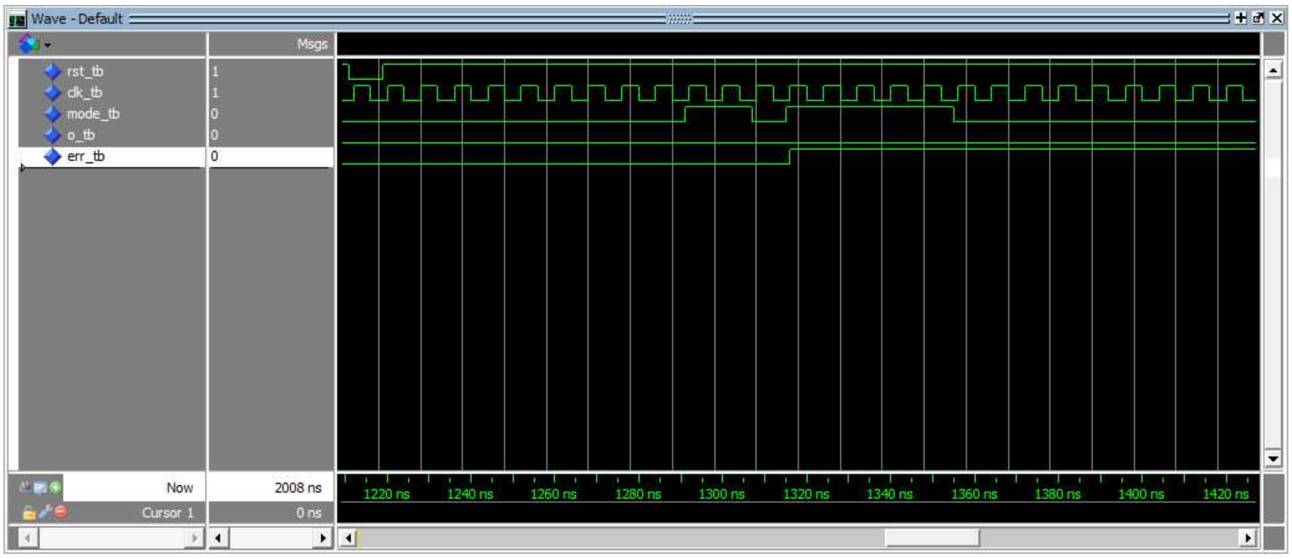
After the first pulse, the output begins to switch. As soon as the error is discovered because of the second pulse, the output is set to ‘0’, and the error flag is set to ‘1’. The second wrong pulse is ignored. The error situation is recovered after the second right pulse, which keeps the output constant to ‘0’.

Case 2’



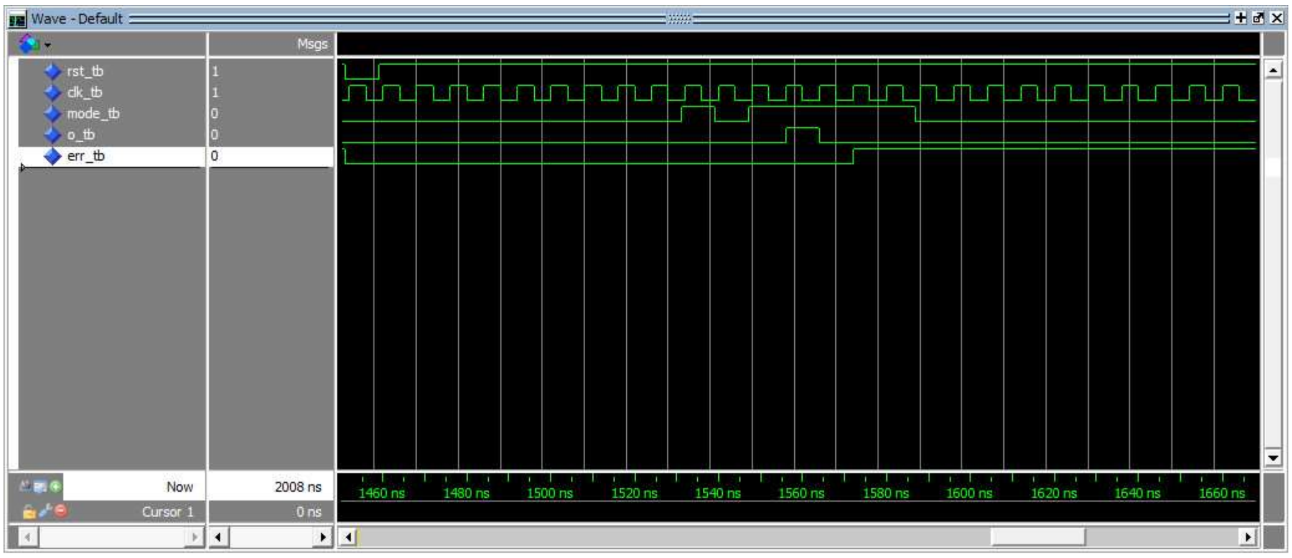
After the first pulse, the output start switching, but after a single clock cycle, the output comes back to ‘0’.

Case 6’



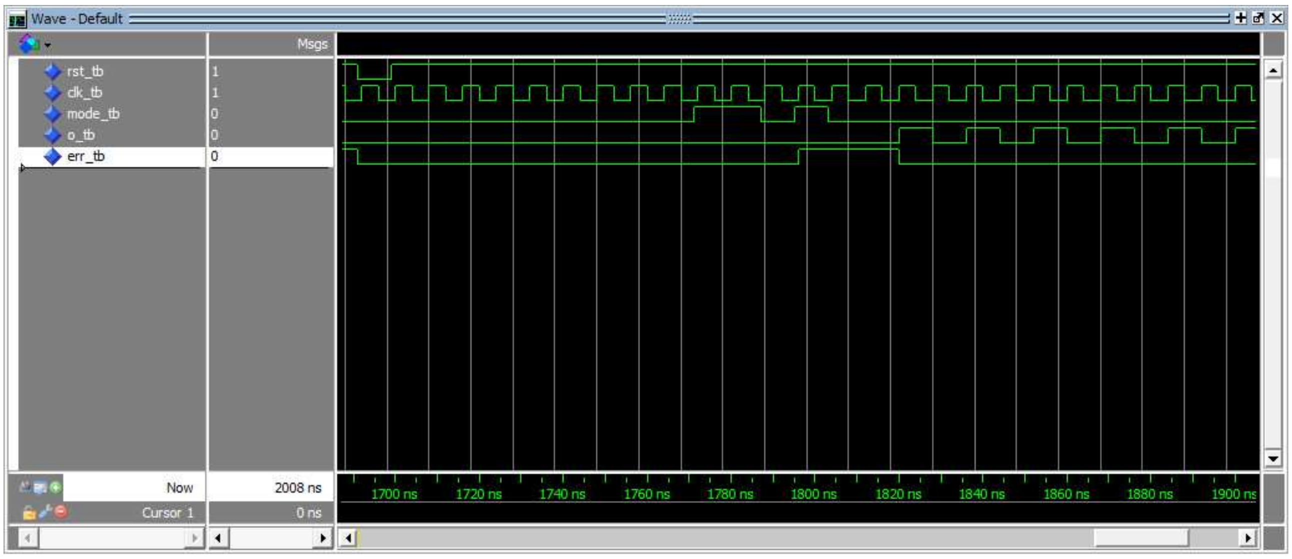
After the first wrong pulse, an error is raised. Since the second pulse is a wrong one, it is ignored.

Case 7’



After the first right pulse, the output start switching, but immediately after an error is raised due to the detection of a wrong pulse.

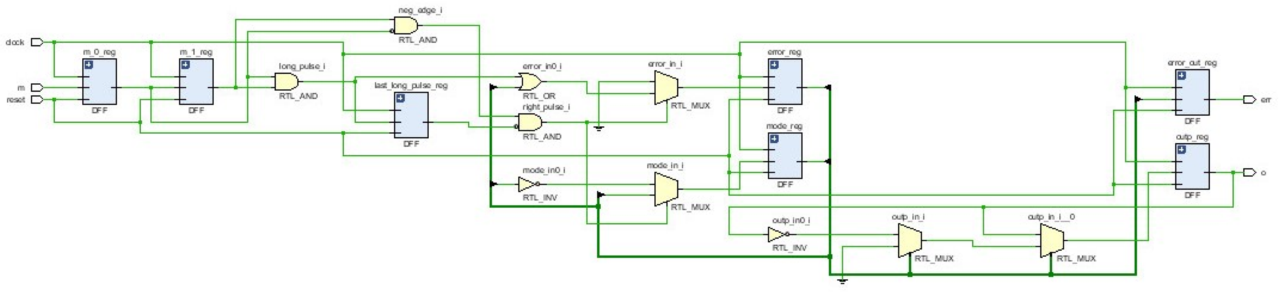
Case 8’



After the first wrong pulse an error is raised, but then it’s recovered after the second pulse, that makes the mode of the system go to “switching”.

Synthesis

Vivado’s elaborated design



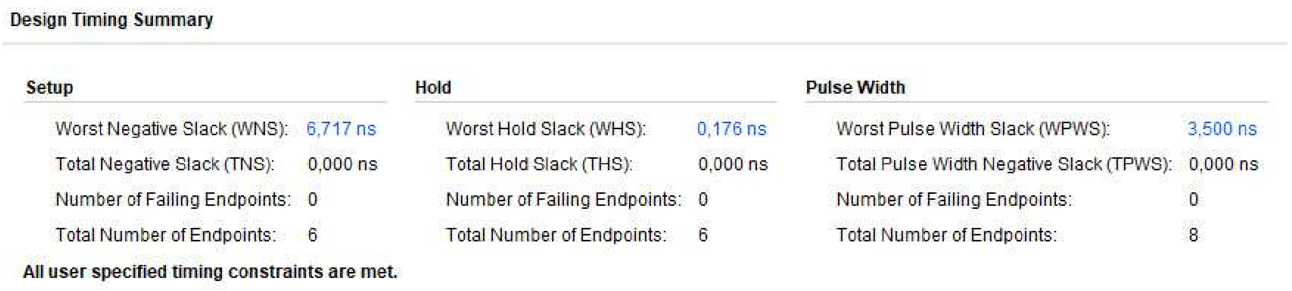
Timing constraints

- Clock @ 125 MHz (period: 8 ns, rise at: 0 ns, fall at: 4 ns)

Synthesis warnings

Warning message	*Reason*
[Constraints 18-5210] *No constraints selected for write.*	This is a warning that comes up for every Vivado’s project because of a bug

Setup and hold slacks



Design Timing Summary

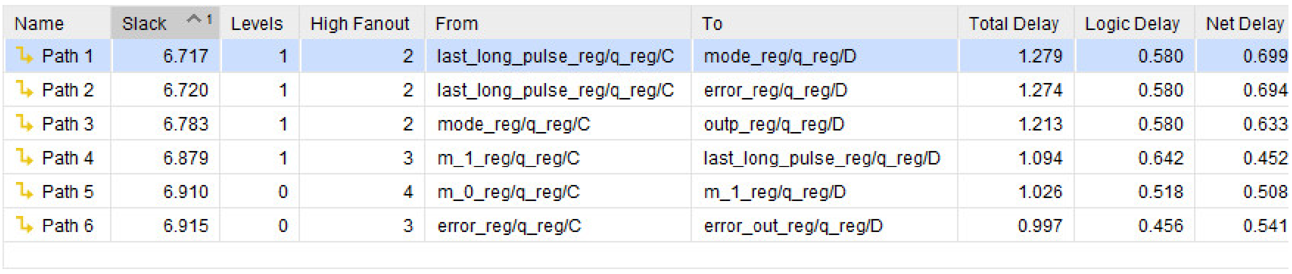
Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	6,717 ns	Worst Hold Slack (WHS):	0,176 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	6	Total Number of Endpoints:	6	Total Number of Endpoints:	8

All user specified timing constraints are met.

As it’s clear from the timing summary of the synthesis, since all the time slacks are positive, there are no setup nor hold time violations.

Critical path

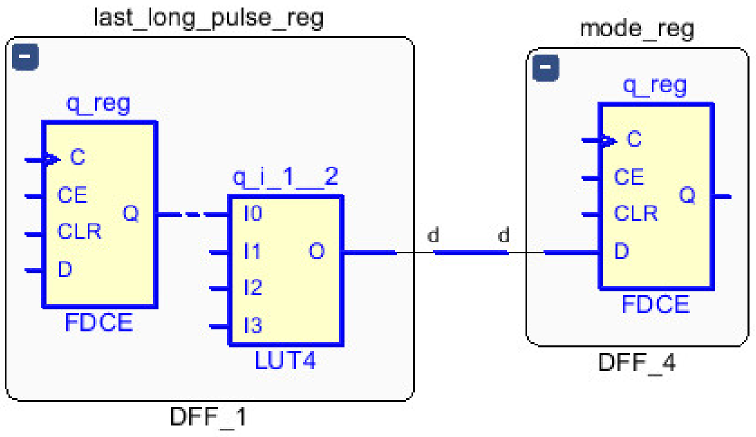
These ones are the register-logic-register paths sorted by t_{setup} slack time; the first one is the critical one, and provides the global setup slack and the maximum clock frequency of the clock for this implementation:



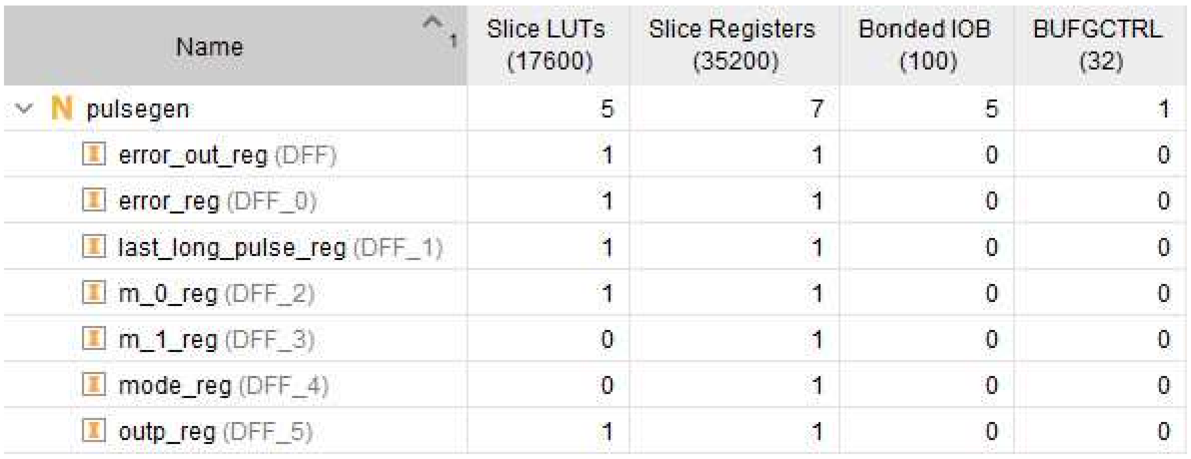
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	6.717	1	2	last_long_pulse_reg/q_reg/C	mode_reg/q_reg/D	1.279	0.580	0.699
Path 2	6.720	1	2	last_long_pulse_reg/q_reg/C	error_reg/q_reg/D	1.274	0.580	0.694
Path 3	6.783	1	2	mode_reg/q_reg/C	outp_reg/q_reg/D	1.213	0.580	0.633
Path 4	6.879	1	3	m_1_reg/q_reg/C	last_long_pulse_reg/q_reg/D	1.094	0.642	0.452
Path 5	6.910	0	4	m_0_reg/q_reg/C	m_1_reg/q_reg/D	1.026	0.518	0.508
Path 6	6.915	0	3	error_reg/q_reg/C	error_out_reg/q_reg/D	0.997	0.456	0.541

Since $Slack = T_{ck} - T_{c-q} - T_{prop} - T_{setup} = 6.717\ ns$, and $T_{ck} = 8\ ns$, it's possible to speed up the clock period to reach $T_{\max ck} = T_{ck} - Slack = 8\ ns - 6.717\ ns = 1.283\ ns$, getting a maximum frequency of $\frac{1}{T_{\max ck}} = \frac{1}{1.283\ ns} \cong 779\ MHz$. This is actually an estimation, an upper boundary of the real maximum frequency value, because in this part of the synthesis we're not considering the delay due to the capacitance of the wires that interconnect the inner blocks of the device.

The following screenshot shows the synthetized registers involved in Path 1:



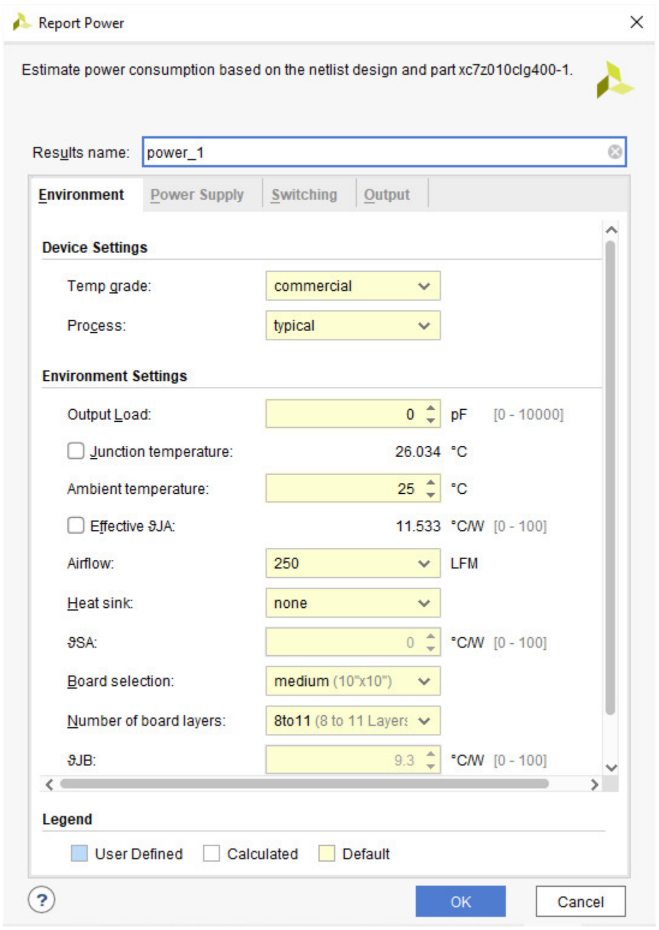
Utilization report



Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (100)	BUFGCTRL (32)
pulsegen	5	7	5	1
error_out_reg (DFF)	1	1	0	0
error_reg (DFF_0)	1	1	0	0
last_long_pulse_reg (DFF_1)	1	1	0	0
m_0_reg (DFF_2)	1	1	0	0
m_1_reg (DFF_3)	0	1	0	0
mode_reg (DFF_4)	0	1	0	0
outp_reg (DFF_5)	1	1	0	0

Power report

Given the default parameters for the environment:



Report Power

Estimate power consumption based on the netlist design and part xc7z010clg400-1.

Results name: power_1

Environment | Power Supply | Switching | Output

Device Settings

Temp grade:	commercial		
Process:	typical		

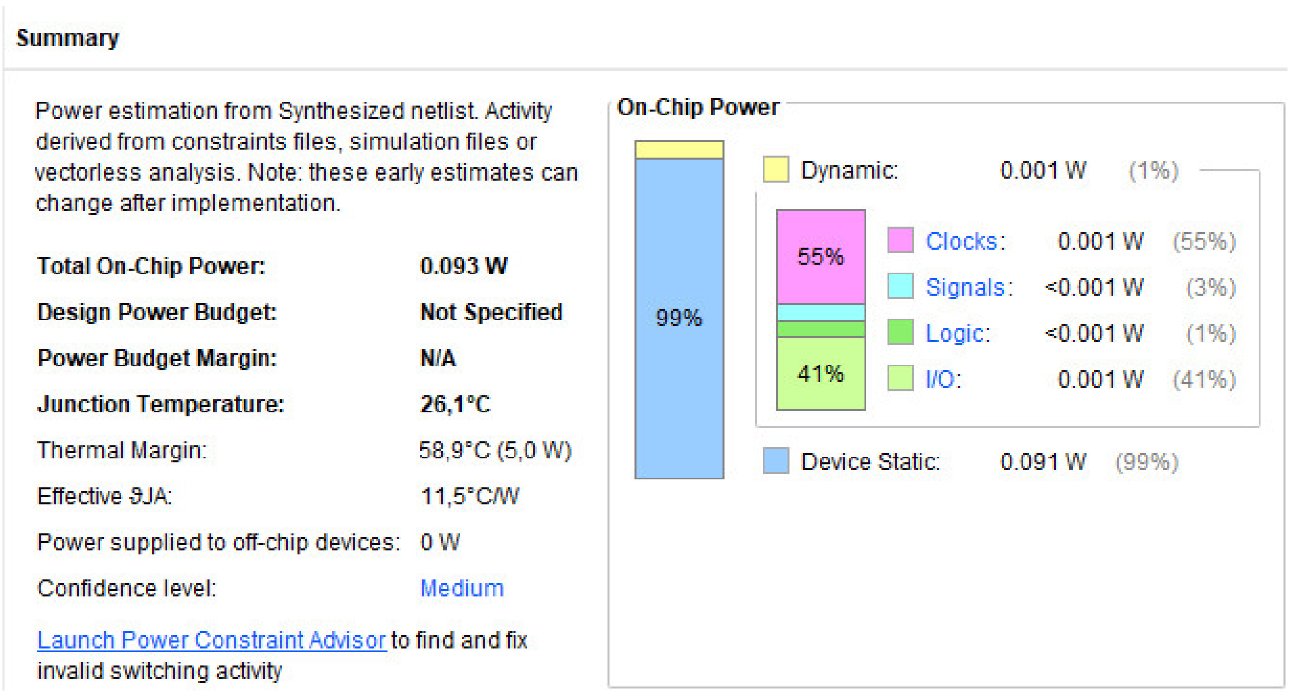
Environment Settings

Output Load:	0	pF	[0 - 10000]
☐ Junction temperature:	26.034	°C	
Ambient temperature:	25	°C	
☐ Effective ϑJA:	11.533	°C/W	[0 - 100]
Airflow:	250	LFM	
Heat sink:	none		
ϑSA:	0	°C/W	[0 - 100]
Board selection:	medium (10"x10")		
Number of board layers:	8to11 (8 to 11 Layers		
ϑJB:	9.3	°C/W	[0 - 100]

Legend

▢ User Defined ☐ Calculated ☐ Default

The synthesis gives back the following power report:



Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.093 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,1°C**
Thermal Margin:	58,9°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		0.001 W	(1%)
	Clocks:	0.001 W	(55%)
	Signals:	<0.001 W	(3%)
	Logic:	<0.001 W	(1%)
	I/O:	0.001 W	(41%)
Device Static:		0.091 W	(99%)

Conclusions

The pulse generator developed during this work meets both the functional and the non-functional requirements.

The simulation of the planned test cases proved that the VHDL code of the device works correctly, and its responses are deterministic and reflect the desired behavior of the system.

After the automatic synthesis stage, we can assert that the final product is a robust, low-power consuming device, and it can be used at high clock frequencies, up to 0.7 GHz.