Laboratory 4

Counter

Digital Systems Design with VHDL
IAS0600

Marcel Cases Freixenet

Tallinn,
October 10, 2018

Tallinna Tehnikaülikool
TalTech

Contents

1 Introduction

1.1 Aim

This project consists in the design and implementation of a digital up-counter on a FPGA board. A signal from a physical push-button is filtered through a debouncer and an edge detector and is used in a counter process to increment one unit each time the button is pushed. The current value of the counter is shown on the board's LEDs in a binary form.

1.2 Background

When in an electronic circuit the input is a physical contact (a switch or a button, for example) the raw signal provided by these components when they are pushed or switched can not be used directly as a digital signal due to the bouncing effect. Thus the signal must be filtered.

In this Lab, the signal goes through two filters: a *debouncing* and *edge detector* components.

1.2.1 Debouncing

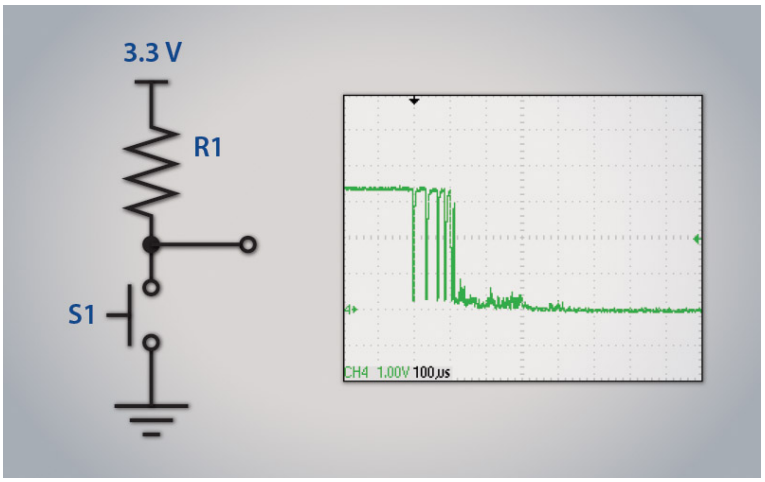

Figure 1.1 *A signal to be debounced*

According to the Lab's manual, *in order to avoid registering several button pushes during a single press it has to be debounced. The problem is that Push Button contains a metal spring and it actually makes contact several times before stabilizing. The high-speed logic circuits may react to the contact bounce as if the*

Push Button has been pressed several times, because some pulses may have a sufficient voltage and long enough duration. Thus, for the hardware implementation to work correctly, contact bounce must be filtered.

For this purpose, a 4-bit shift register is used. It is switched at a frequency of 100Hz (see ⇨ Section 2.1.1).

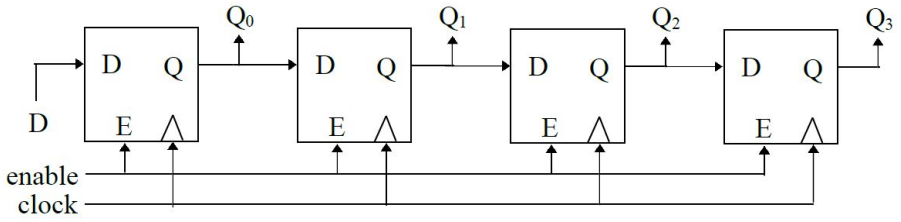

Figure 1.2 *A 4-bit SIPO Shift Register Built from Four D-type Flip-flops*

When the four Q signals of the four bit SIPO are equal to 1, then we can consider that the input signal D has been debounced.

1.2.2 Edge detection

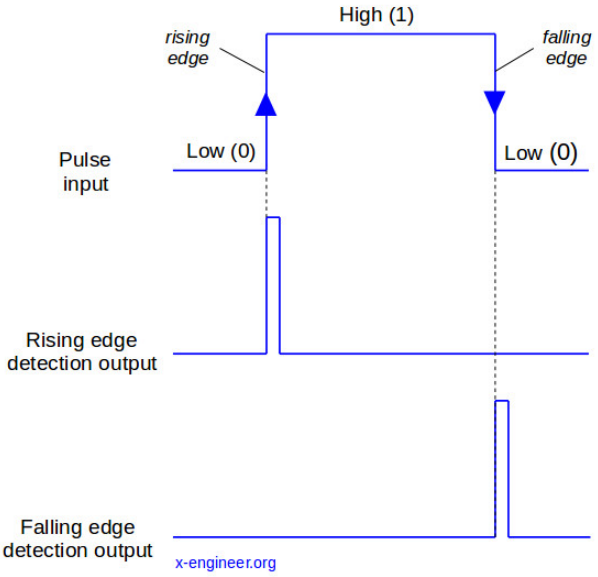

Figure 1.3 *Raw signal and its edge detection filtered signal*

The edge detection component has the purpose to transform the signal from a long pressed button to a single clock time signal. Its purpose is to run a process statement only once and not an uncontrollably number of times. For this purpose, a D-type Flip Flop is used.

2 Workflow

The hardware description of the counter consists in two components (debouncing and edge-detection) and a process (up-counter). They run concurrently one another.

2.1 Debouncing component

This debouncing component uses a 4-bit shift register made from D type flip flops. The input signal is considered raw (pre-filtering) and the output signal is considered filtered. When the four bits of the Q signals equal 1 then the output is triggered as seen in line 29. The *enable* signal has to be fed with a clock-divided signal at a frequency of 100Hz.

```
1  entity debouncing is
2          Port (  clk : in std_logic;
3                  reset : in std_logic;
4                  enable : in std_logic ;
5                  in_raw : in std_logic; -- D input
6                  out_filtered : out std_logic
7                  );
8  end debouncing;
9
10
11 architecture Behavioral of debouncing is
12         signal Q : std_logic_vector (0 to 3);
13         signal D : std_logic ;
14 begin
15
16 --  *4-BIT SHIFT REGISTER*
17 process (clk) begin
18         if rising_edge(clk) then
19                 if reset = '1' then
20                         Q <= (others => '0');
21                 elsif enable = '1' then
22                         Q(0) <= D;
23                         Q(1 to 3) <= Q(0 to 2);
24                 end if;
25         end if;
26 end process;
27
28 D <= in_raw;
29 out_filtered <= '1' when Q = "1111" else '0';
30
31 end Behavioral;
```

2.1.1 Clock divider component

A clock divider is necessary to define the **shifting frequency** of the debouncer. This component transforms the original clock of the Nexys 4 board (100MHz) to a frequency of 100Hz and each new pulse has a duration of a single clock-time (10ns, as the original frequency has).

The *clk_div* signal is routed to the *enable* signal of the debouncing component.

```
entity clock_divider is
        generic (eoc: integer := 999999); --100Hz
        Port (  clk : in STD_LOGIC;
                reset : in STD_LOGIC;
                clk_div : out STD_LOGIC
                );
end clock_divider;


architecture Behavioral of clock_divider is
        signal counter: integer range 0 to eoc;
begin

process (clk) begin
        if rising_edge(clk) then
                clk_div <= '0';
                if reset = '1' then
                        counter <= 0;
                elsif counter = eoc then
                        counter <= 0;
                        clk_div <= '1';
                else counter <= counter + 1;
                end if;
        end if;
end process;

end Behavioral;
```

2.2 Edge-detection component

The edge-detection component uses a single D-type flip flop that moves the signal state of *q0* to *q1* every clock pulse. This process results in a delay. Thus when *q0* is high and *q1* is low then *out_filtered* goes to '1' only for a single clock width time (10ns).

```
entity edge_detection is
        Port (  clk : in std_logic;
                reset : in std_logic;
                in_raw : in std_logic;
                out_filtered : out std_logic
                );
end edge_detection;


architecture Behavioral of edge_detection is
```

```
11          signal q1, q0 : std_logic; -- Flip-Flop D internal signals
12  begin
13
14  process (clk) begin -- Flip-Flop D process
15          if rising_edge(clk) then
16                  if reset = '1' then
17                          q0 <= '0';
18                          q1 <= '0';
19                  else
20                          q0 <= in_raw;
21                          q1 <= q0;
22                  end if;
23          end if;
24  end process;
25
26  out_filtered <= q0 and not q1;
27
28  end Behavioral;
```

2.3 Up-counter process

This process is described in the top level file *counter.vhd*. Its purpose is to sum a unit every time *btnc_post_edge* equals 1. This value is stored to a 16-bit signal (sub_buf) and is routed to the board's LEDs concurrently (*sum*).

```
1   process (clk, reset) begin
2           if rising_edge(clk) then
3                   if reset = '1' then
4                           sum_buf <= (others => '0');
5                   elsif btnc_post_edge = '1' then
6                           sum_buf <= std_logic_vector(unsigned(sum_buf) + 1);
7                   end if;
8           end if;
9   end process;
10  sum <= sum_buf;
```

2.4 Top level file

2.4.1 Libraries

The first lines of the code define the libraries needed for the project. In this project, only one library is needed, which is the standard logic revision 1164 defined by IEEE.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
```

For this project a *IEEE.NUMERIC_STD.ALL* library is necessary to perform arithmetic operations correctly.

2.4.2 Entity

Three input ports are needed: *clk* (100MHz), *reset* (common for the whole system) and *btnc* (to indicate that we want to sum a unit to the counter). The output signal is a 16-bit *sum* to show on the LEDs the current value of the counter.

```
entity counter is
        Port (  clk : in std_logic ; -- Internal clock (100MHz)
                reset : in std_logic ; -- Upper button on Nexys 4
                btnc : in std_logic ; -- Centre Button on Nexys 4
                sum : out std_logic_vector (15 downto 0) -- LEDs binary
                ↪  output
                );
end counter;
```

2.4.3 Architecture

The internal signals defined in the architecture of the project are those that route components and a buffer signal *sum_buf* in order to operate correctly. Then all components are instantiated and all together form the filter.

The physical input signal *btnc* goes to the debouncer, then is transformed to *btnc_post_debouncing* and enters the edge detection component. The latter component outputs a *btnc_post_edge* signal, which is considered completely filtered, and goes to the counter process.

```
architecture Behavioral of counter is

-- [...] Components definition
-- Internal signals
        signal clk_div : std_logic ;
        signal btnc_post_debouncing, btnc_post_edge : std_logic ; -- Filtered
        ↪  signals from counter button
        signal sum_buf : std_logic_vector (15 downto 0); -- 'sum' output
        ↪  signal buffer in order to read and use it as an input
begin

inst_clock_divider : clock_divider
        port map (  clk => clk,
                reset => reset,
                clk_div => clk_div
                );

inst_debouncing : debouncing
        port map (  clk => clk,
                reset => reset,
                enable => clk_div,
                in_raw => btnc,
                out_filtered => btnc_post_debouncing
                );

inst_edge_detection : edge_detection
        port map (  clk => clk,
                reset => reset,
                in_raw => btnc_post_debouncing,
```

```
28                  out_filtered => btnc_post_edge
29                  );
30
31
32  -- [...] Binary Up Counter Process (as described in Section 2.3)
33
34  end Behavioral;
```

2.5 Testbench

The aim of the testbench is to simulate the real behavior of a physical button and to check whether or not the hardware description is capable to debounce the signal and detects the rising edge. The main process of this testbench is the folowing:

```
1   stimulus: process begin
2
3           btnc_tb <= '1' ;    wait for 5 ns;
4           btnc_tb <= '0' ;    wait for 5 ns;
5
6           btnc_tb <= '1' ;    wait for 5 ns;
7           btnc_tb <= '0' ;    wait for 5 ns;
8
9           btnc_tb <= '1' ;    wait for 5 ns;
10          btnc_tb <= '0' ;    wait for 5 ns;
11
12          btnc_tb <= '1' ;    wait for 100 ns;
13          btnc_tb <= '0' ;    wait for 100 ns;
14
15  end process;
```

The results are shown in ⇨ Section 3.

3 Results and discussion

A simulation has been run for this project, showing the behavior of the hardware description for three consecutive pushes.

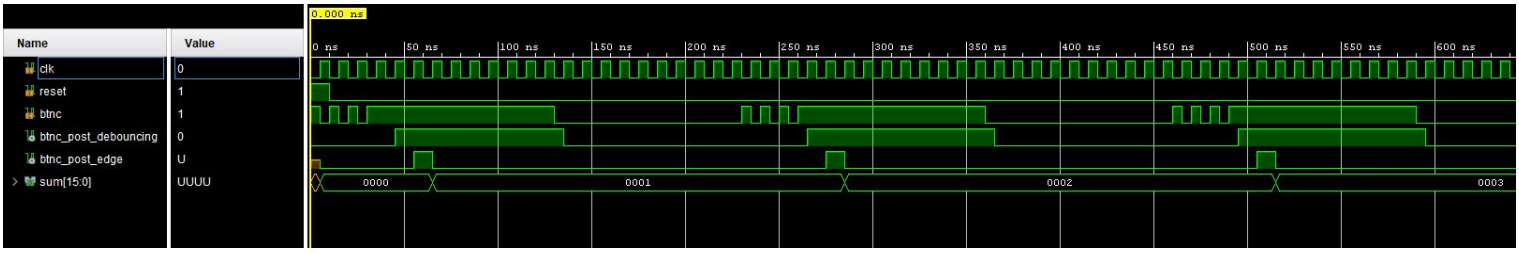

Figure 3.1 *Simulation for three pushes*

For simulation purposes, the *enable* signal of the debouncer has been fed with the original clock (100MHz) instead of the divided clock of 100Hz.

Once the simulation has been performed and checked, multiple implementations have been run, the first of them using a single edge detection without the debouncing filter. In this implementation, although the counter worked, it was difficult to sum only one unit to the internal counter because the signal from the push button was bounced. Sometimes more than one unit were summed.

In the second part of the project, the debouncing filter has been used together with the edge detection. When the debouncing is shifted at the right frequency (100Hz) then the output signal from the edge detector was clean and after several tests it never summed more than one unit per push action.

4 Conclusion

This Lab has been useful to understand how physical signals have to be processed in order to be available to use them in a digital system.

As we have seen, the raw input of a push-button is not valid as it has to be filtered using at least an edge detector, but a debouncer component is necessary to guarantee that no more than one pulse (for every real physical pulse) will be used in the digital system.

We have experienced the necessity to use buffer signals. They are useful when an output signal also has to be used as an input signal.

I consider that this laboratory has been successfully developed as we have observed physically in the Nexys 4 board the binary value of the counter on 16 LEDs.