Department of Electrical and Computer Engineering

University of Canterbury

ENEL 373 Project Report

Designing a Programmable Down-Counter and PWM Waveform Generator on an FPGA

by

Kate Chamberlin (54384616)

Jesse Baxter (83051957)

Report submitted on 27/05/18

Contents

1. Introduction

A field programmable gate array (FPGA) is a device in which the hardware can be configured and reconfigured by a user. By use of a hardware descriptive language such as VHDL or Verilog, the user can construct circuits as simple as an AND gate to systems as complicated as a CPU. The aim of this project was to design a 16-bit, programmable down-counter and PWM (Pulse-Width Modulation) waveform generator on an FPGA.

For this project, the design was implemented on a Nexys-4DDR board, and VHSIC Hardware Description Language (VHDL) was used as the hardware descriptive language. It was implemented using a program called Vivado 2016.2 by Xilinx [1]. The main features of the FPGA used for this project were:

- 16 slide switches
- five push buttons
- USB host connector
- LEDs

It was required that the user could switch between to output waves – one generated by the programmable down-counter, and one generated as a PWM waveform. The programmable down-counter had three main clock rates that the user could toggle through, one of which could be reprogrammed using the 16 slider switches. These clock rates then had to create an output that either toggled high/low, asserted high when the down-counter reached zero, or asserted low. The user could also set and reset the period and duty cycle of the PWM waveform output using the slider switches. The final design needed to implement at least one finite state machine (FSM) to demultiplex the buttons, switches and output waveforms.

2. Top-Level Design

It was decided that the final design would consist of three FSMs, shown in **Error! Reference source not found.**. The overarching FSM_3 used the up button to change overall modes – PWM generation and down-counting. Based on the state, it then forwarded relevant signals (such as button signals and switch inputs) to the correct module. These modules were FSM_1 for the PWM generation mode, and FSM_2 for the down-counting mode.

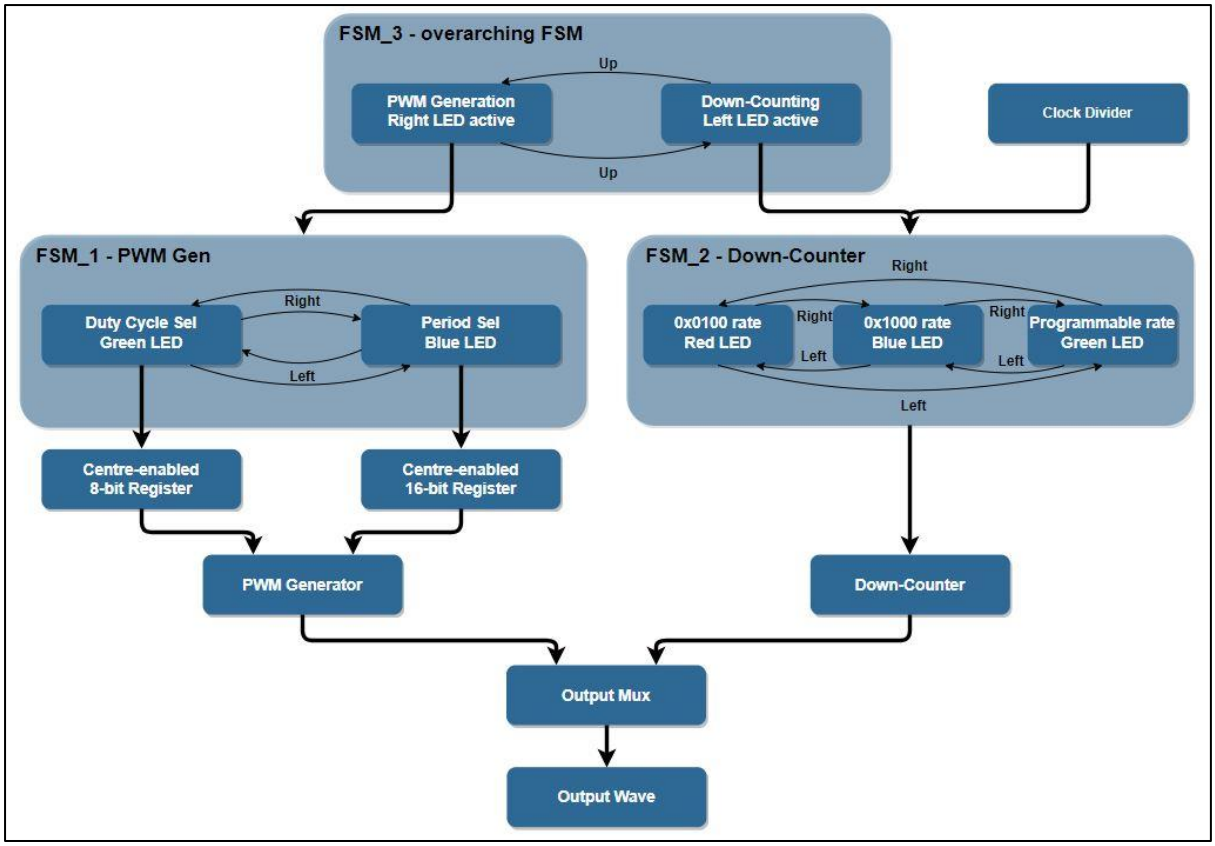


Figure 1 - Top-level block diagram for the timer (state diagrams included for understanding)

FSM_1 then used the left and right buttons to change states for user selection. Depending on the state, the centre button signal was used as an enable for either the 8-bit duty cycle register (allowing the user to input the duty cycle value), or the 16-bit period register (allowing the user to input the period of the PWM waveform). The right LED was used for debugging this FSM by displaying a specific colour depending on the current state of FSM_1.

The second FSM used the left and right button signals from FSM_3 to cycle through the three required main clock signals that were generated by the clock divider. It thus demultiplexed the signals by forwarding the selected clock signal to the down-counter. This FSM used the left LED to display the current state similarly to FSM_1.

Once the PWM generator and down-counter had each generated their respective waveforms, these signals were passed to the output multiplexer which allowed one of the signals to be passed to the header and debugging LED to be measured by the oscilloscope. This was done by reading the state from FSM_3 and selecting the input signal that matched the state.

3. Expanded Design Summary

3.1 16-bit Programmable Down-Counter

The main requirement of the down-counter was that it had to support three main clock rates. The user would then be able to select the clock rate they wished to output to the waveform LED (LED0). There are five programmable push buttons on the Nexys-4DDR board and they are:

- BTNC (centre)
- BTNL (left)
- BTNR (right)
- BTNU (up)
- BTND (down)

Of these, only the left, right, centre and down buttons were forwarded to the FSM using a 4-bit bus since the up button was used exclusively for switching states in FSM_3. The left and right buttons were used to change states, and the centre button was used to load the switch input as the programmed clock rate.

3.1.1 ClockDivider_100k

All 16 slide switches were used to set the value at which the down-counter would count down from. In this mode, BTNC was used as an enable for a 16-bit register, to pass the current configuration of the switches via a 16-bit bus to ClockDivider_100k. The clock divider contained three down-counting processes – one for each clock rate. These rates were arbitrarily chosen as follows:

- rate_low (for the 0x0100 clock rate)
- rate_high (for the 0x1000 clock rate)
- rate_custom (for the clock rate determined by the switches)

Each process toggled a signal once it reached zero. ClockDivider_100k then outputs all three formed clock signals, passing them to FSM_2 for output selection.

3.1.2 FSM_2

FSM_2 took the buttons and the 100MHz clock signal and outputted one of the three clock signals depending on which state the FSM is in. Both buttons BTNL and BTNR were used to

switch between the three main clock rates. The benefit of using two buttons instead of one was that the user was able to navigate in both directions through the various down-counter modes. To achieve this functionality, the three states were created in a type declaration, allowing for the creation of a binary-encoded FSM. It had a format largely similar to the binary encoding in [2], and is shown in Appendix 9.3. Three state signals (previous_s, current_s, and next_s) were created with each one assigned to a different state. A case statement was then used to output the clock rate associated with the state of current_s to the down-counting module. Whenever BTNL or BTNR was pressed, all three state signals were reassigned to different states, and a specific colour was displayed with the left LED, giving FSM_2 the functionality shown in Figure 1.

3.2 PWM Generator

Like the down-counter, the PWM generation section of the design had the use of the left, right, centre and down buttons in a 4-bit bus. The left and right buttons were again used to change states between duty cycle selection and period selection, with the centre button acting as a 'select' button (i.e. enable for the registers). This meant that separate registers were required for the period and duty cycle, which were PeriodRegister and DutyRegister respectively. Generation of the PWM waveform was a similar process to that of the down-counter.

3.2.1 FSM_1

This state machine had to switch between duty cycle selection and period selection. It was also binary encoded, since for a two-state machine this was not complex (shown in Appendix 9.2). Whenever a button was pressed, the state simply toggled, activating the correct LED colour and register. A state change also forwarded the centre button signal to the relevant register as an enable signal.

3.2.2 PWM_Gen

The PWM_Gen module used a 16-bit clock signal (clk_ctr) to count down from a value and toggle the output when the signal reached X"0000". The value that clk_ctr counted down from had to be computed from the values stored in both the period register (reset_period) and the duty register (reset_duty). Dividing reset_duty by the maximum value that could be stored in the duty register gave the desired duty cycle as a percentage. This could then be multiplied by

reset_period to obtain the proportion of time at which the output waveform should be high. The calculation is shown below:

```
compare_val <= to_integer(unsigned(reset_duty) *
      unsigned(reset_period) * (1 / 255));
```

In to perform the calculation, reset_duty and reset_period had to be converted to unsigned integers. This was because reset_period is a 16-bit vector whereas reset_duty is only an 8-bit vector. Multiplying the by (1 / 255) would have given the result as a float, so it was necessary to use the to_integer calculation to covert compare_val to an integer.

3.3 Mode changing

3.3.1 FSM_3

FSM_3 was implemented as a two-state binary encoded state machine, again due to simplicity. This was done as shown in Appendix 9.4. Unlike the other two FSMs however, it received all button signals directly from the debouncers, and forwarded on only the four that were relevant to FSM_1 and FSM_2. It used the up button to change states and outputted a state variable for used by the output_mux module outlined below. It also forwarded the switch signals to the relevant modules, which retrospectively was unnecessary and created more complexity in the design.

3.3.2 Output_Mux

The output_mux module existed in cooperation with FSM_3. Receiving input waveforms and LED signals, it multiplexed the input waveforms, selecting only the one generated by the current mode's modules. It also multiplexed the LED signals so that only one RGB LED set was active at any given time, so it was clear which mode FSM_3 was in.

3.4 Supplementary modules

3.4.1 Registers

Three registers were used in this design. Two 16-bit registers were used to process input from the switches as the PWM period and programmable down-counting period. One 8-bit register was also used to process input from switches 0 to 7 as the duty cycle. These registers were all BTNC enabled on the rising edge of the 100MHz clock signal if their respective states were active.

3.4.2 Debouncers

Button debouncing was implemented to ensure stable state-changing in all the FSMs. Each button signal was fed through a debouncer before being passed to any other module (Figure 2). The debouncer used a counter to count from the button push event up to 50ms before outputting high for one single 100MHz clock cycle.

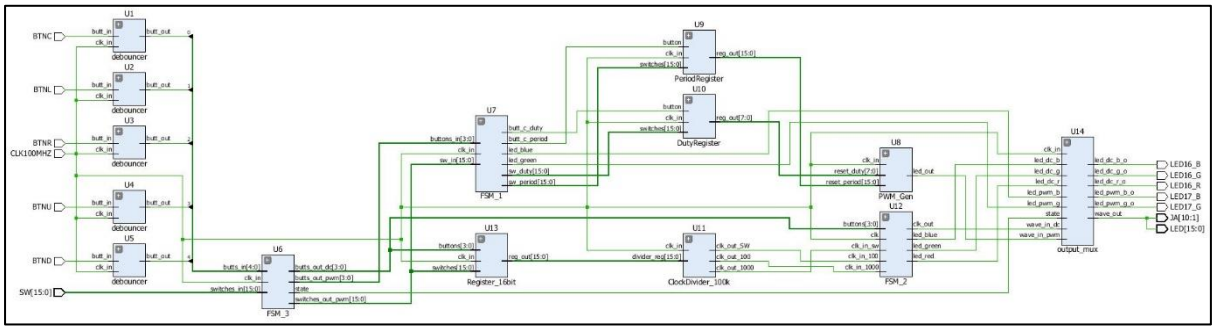


Figure 2 - The RTL schematic for the timer (expanded in appendix 9.1)

4. Testing

Throughout the design process, several modules were tested using the simulator provided by Vivado. One of these has been included for reference in Appendix 9.5. Using this simulator, it was relatively simple to adjust input signals as they were expected and evaluate the output signals. The testbench that has been appended was for FSM_2 (the down-counting FSM) and showed correct state-switching, but the output signal was not completely as expected due to debugging problems outlined in section 5.

In this testbench, the LED outputs changed with each left/right button press, thereby confirming that state-changes were functional. The LED colours were also checked to ensure that they corresponded to the current state. The button presses were coded so that the testbench results showed a complete cycle of states (a series of left/right presses from buttons 1 and 2) between t = 0ns and t = 1,200ns. Displayed from time t = 1,200ns onwards was the lack of change with other presses (from the centre and down buttons 0 and 3) since they had no functionality within the FSM.

The clk_out signal was functional for some states but not for others. For example, while led_blue was active, clk_out was outputting the clk_in_100 signal. Similarly, when led_green was active, clk_out was outputting the clk_in_150 signal. However, when led_red was active, the clk_in_50 signal was not being displayed on the clk_out line as expected, but rather the clk_in_100. This may have been due to a small discrepancy in either FSM_2 or the simulation code.

5. Problems

When implementing the design in final stages of the project, a timing error was encountered. From the error messages obtained, it was discovered that the PWM_Gen module was exceeding the maximum time constraints. It was found that the error was coming from the calculation of the duty cycle since initially compare_val was assigned:

```
to_integer((unsigned(reset_duty) * unsigned(reset_period)) / 255));
```

The division in the code above was taking the CPU too long to execute, hence obtaining a timing violation. By instead multiplying by (1 / 255), the computation executed much faster and the implementation was able to complete successfully. Due to this problem, the design was not functional for the demonstration. Debugging and testing also lagged as a result.

6. Improvements

One minor improvement that could have been made with the design would have been passing the sw_in bus straight into PeriodRegister and DutyRegister bypassing FSM_1. This would have reduced the number of signals going into and out of the FSMs, improving maintainability and reducing code complexity.

Furthermore, the current design had one overarching entity which connected every single module. An improvement to maintainability and complexity could be made in future by having multiple structural entities rather than one. For example, the button debouncers could have been grouped together, reducing the complexity of the overarching entity's code.

7. Conclusion

The aim of this project was to design a 16-bit, programmable down-counter and PWM waveform generator on an FPGA using VHDL. By use of the switches, left, right, and centre

buttons, the user was able to select different clock rates for the programmable down-counter and set the period and duty cycle of the PWM output. The up button provided a means for the user to switch between the two modes, and coloured LEDs were used to indicate the state the FPGA is in. Both modes were implemented with FSMs and an overarching FSM was used for toggling between the down-counter and the PWM waveform generator. Having a design like this allowed for more functionality as more FSMs could have been added to the design through FSM_3. The only drawn back with would be that FSM_3 would have become harder to maintain. Due to issues with timing errors the design was not able to be fully debugged and tested, and so was not completely functional. While the LEDs indicated that state switching was functional for all FSMs, the output waveforms were not correct, and more time spent debugging would have fixed this.

8. References

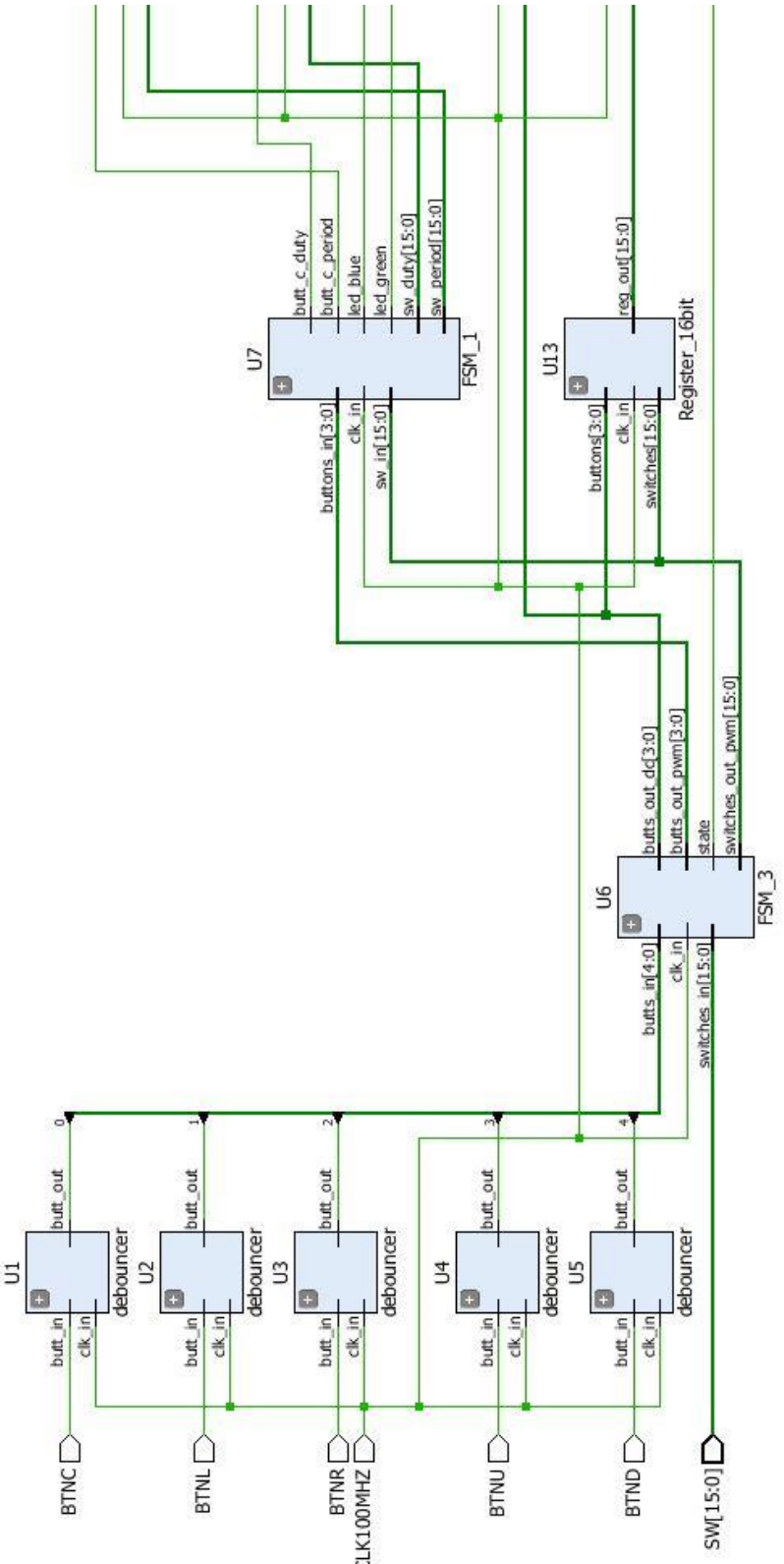
[1] Xilinx, "Vivado Design Suite - HLx Editions," 04 April 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed 27 May 2018].

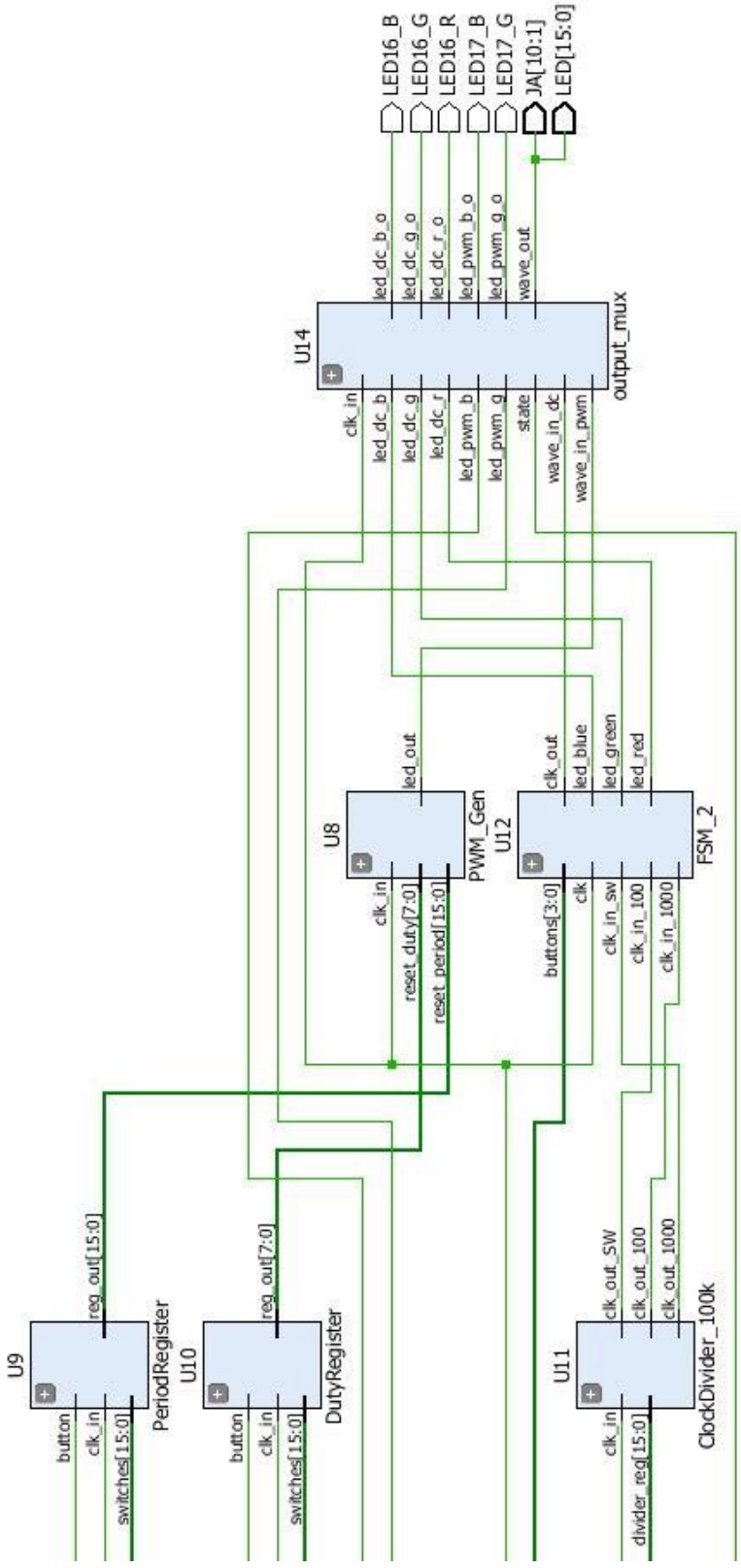
[2] D. S. J. Weddell, "Designing State Diagram Modules using VHDL (Part 2)," University of Canterbury, Christchurch, 2018.

[3] J. Rajewski, "What is an FPGA?," 17 January 2018. [Online]. Available: https://embeddedmicro.com/blogs/tutorials/what-is-an-fpga. [Accessed 20 May 2018].

9. Appendices

9.1 Expanded view of the RTL schematic




LED16_B
LED16_G
LED16_R
LED17_B
LED17_G
JA[10:1]
LED[15:0]
U14
output_mux
clk_in
led_dc_b
led_dc_g
led_dc_r
led_pwm_b
led_pwm_g
state
wave_in_dc
wave_in_pwm
led_dc_b_o
led_dc_g_o
led_dc_r_o
led_pwm_b_o
led_pwm_g_o
wave_out
U8
PWM_Gen
clk_in
reset_duty[7:0]
reset_period[15:0]
led_out
U12
FSM_2
buttons[3:0]
clk
clk_in_sw
clk_in_100
clk_in_1000
clk_out
led_blue
led_green
led_red
U9
PeriodRegister
button
clk_in
switches[15:0]
reg_out[15:0]
U10
DutyRegister
button
clk_in
switches[15:0]
reg_out[7:0]
U11
ClockDivider_100k
clk_in
divider_reg[15:0]
clk_out_SW
clk_out_100
clk_out_1000

9.2 FSM_1 State-switching

```
process (state, sw_in)
begin
    if state = '0' then --period selection
        led_blue <= '1';
        led_green <= '0';
        sw_period <= sw_in;
        butt_c_period <= buttons_in(0);
        butt_c_duty <= '0';
    else                --duty cycle selection
        led_blue <= '0';
        led_green <= '1';
        sw_duty <= sw_in;
        butt_c_duty <= buttons_in(0);
        butt_c_period <= '0';
    end if;
end process;
```

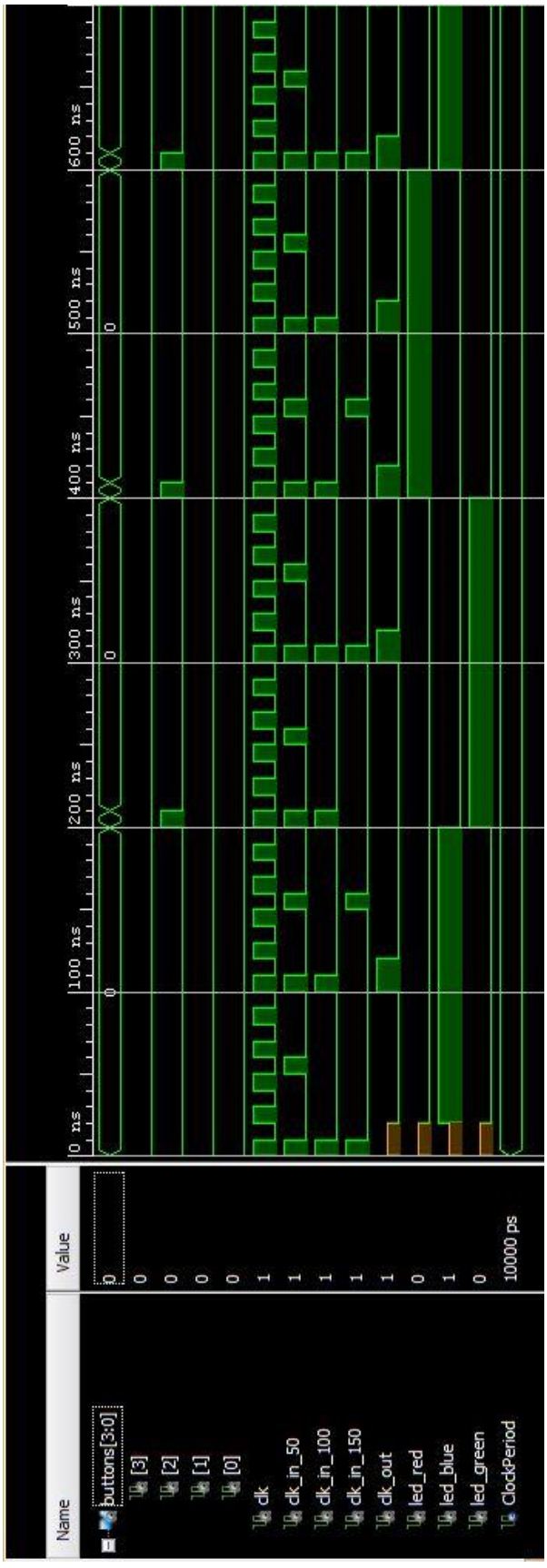
9.3 FSM_2 State-switching for one case

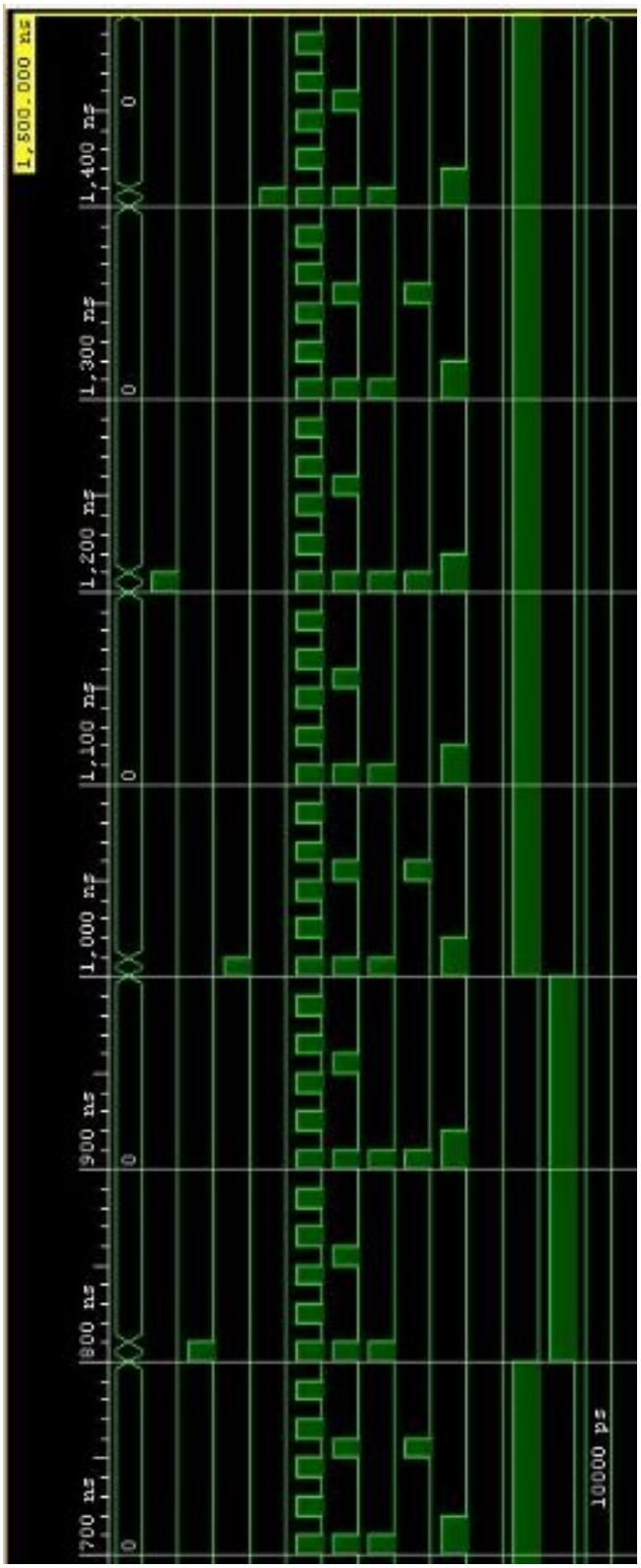
```
process(clk, buttons)
begin
    if rising_edge(clk) then
        case current_s is
            when rate_low =>
            if buttons(1) = '1' then
                clk_out <= clk_in_sw;
                previous_s <= rate_high;
                current_s <= rate_custom;
                next_s <= rate_low;
                led_red <= '0';
                led_blue <= '0';
                led_green <= '1';
            elsif buttons(2) = '1' then
                clk_out <= clk_in_1000;
                previous_s <= rate_low;
                current_s <= rate_high;
                next_s <= rate_custom;
                led_red <= '0';
                led_blue <= '1';
                led_green <= '0';
            else
                clk_out <= clk_in_100;
                led_red <= '1';
                led_blue <= '0';
                led_green <= '0';
            end if;
```

9.4 FSM_3 State-switching

```
process (clk_in)
    begin
        if rising_edge(clk_in) then
            if butts_in(3) = '1' then
                sv <= not sv; --toggle state
            end if;
            if sv = '0' then
                state <= '0';
                switches_out_dc <= switches_in;
                butts_out_dc(2 downto 0) <= butts_in(2 downto 0);
                butts_out_dc(3) <= butts_in(4);
                butts_out_pwm <= X"0";
            elsif sv = '1' then
                state <= '1';
                switches_out_pwm <= switches_in;
                butts_out_pwm(2 downto 0) <= butts_in(2 downto 0);
                butts_out_dc <= X"0";
            end if;
        end if;
    end process;
```

9.5 Testbench output




1,500.000 ns
1,400 ns
1,300 ns
1,200 ns
1,100 ns
1,000 ns
900 ns
800 ns
700 ns
0
10000 ps