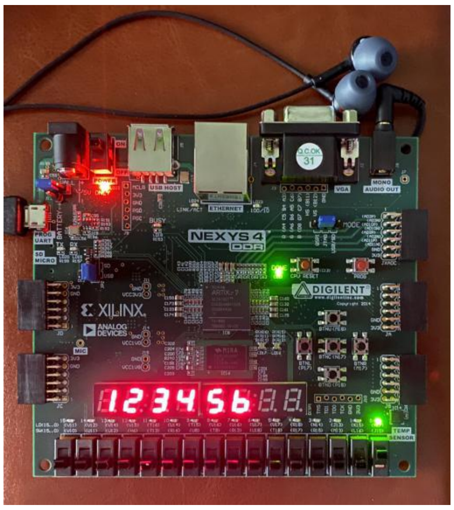
FPGA Digital Alarm Clock

CSCE 3730 Project Report

By Chris Herzberg



The purpose of this project was to create a fully functional digital alarm clock on an FPGA board. The clock displays both the current time, and the time that the alarm will sound, with the ability to adjust both of these times on their respective display setting. Additionally, the user may toggle if the alarm is active or inactive. The clock runs off the board's internal 100MHZ clock, brought down to 1Hz.

The numbers for the clock are displayed on six of the eight built in 7-segment displays. These displays show the current time, as well as the time the alarm is set to go off when the switch to set the alarm is toggled on.

If the alarm is active when the current time reaches the set alarm time, then a sound is emitted through the boards audio output pin which can be heard using either headphones or a speaker, and the LEDs underneath the 7-segment displays flash on and off until the user turns the alarm off.

Software / Hardware

This project was coded using VHDL in the Vivado Design Suite.

The hardware used in this project includes the Artix-7 Nexys 4 DDR FPGA board from Xilinx, and an audio output device such as computer speakers or headphones to output a sound when the alarm goes off.

Implementation

As stated above, the alarm clock was designed using VHDL. The design was comprised of multiple functions that were split into various components which were then linked together in a top module via signals.

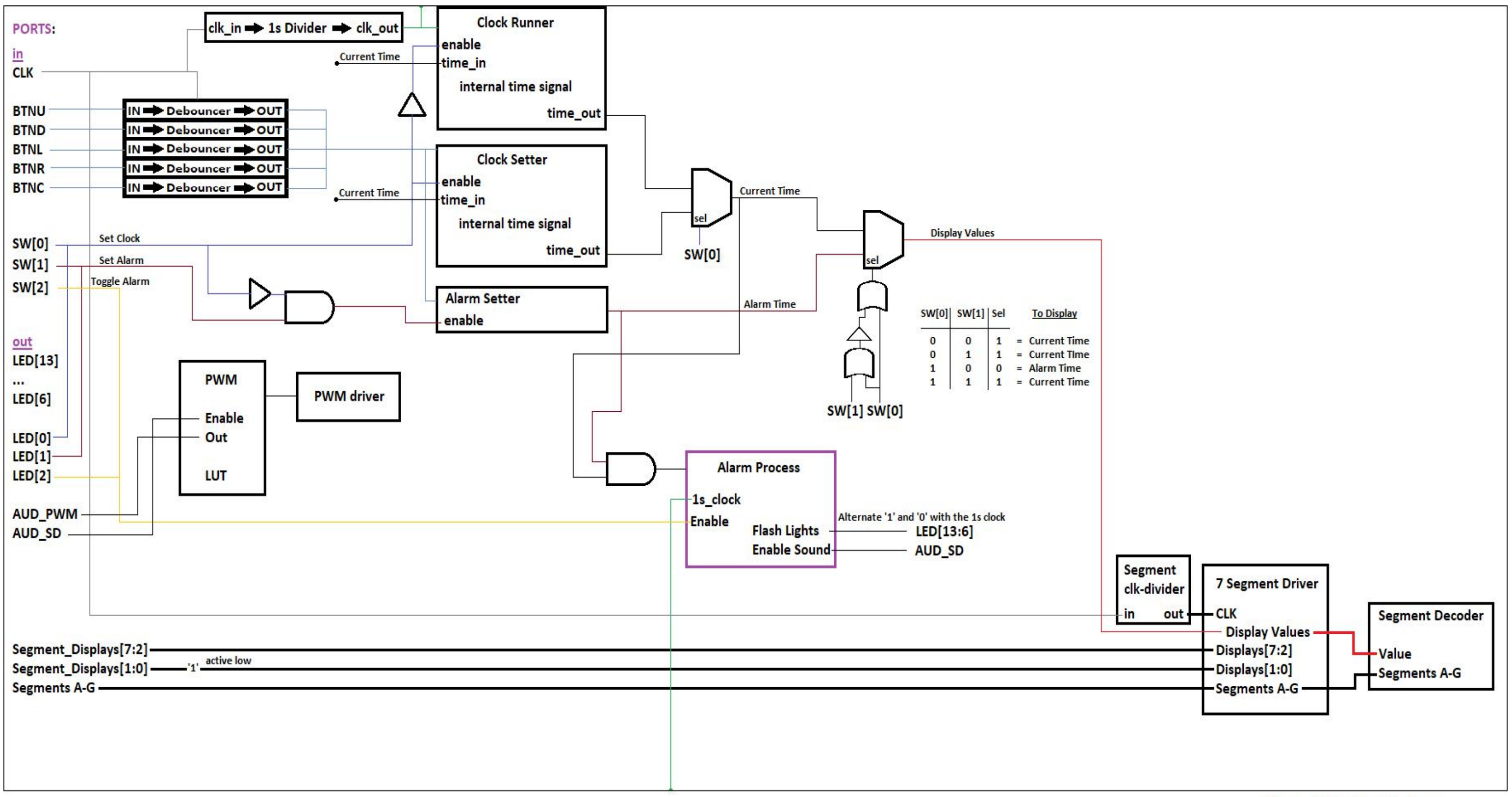


Figure 1. Design Diagram

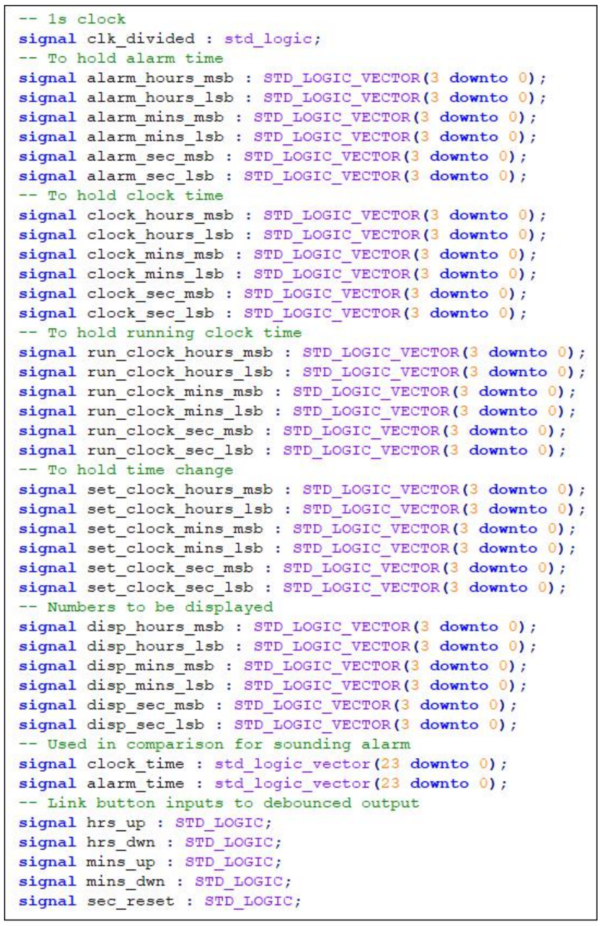
The diagram in Figure 1., above shows an overview of how the clock was designed. The input and output ports of the top module are mapped to the pins on the board as shown on the left of the diagram. These were then connected to the various components using signals in the top module indicated by the colored lines. Each component within a black box in Figure 1. is a separate component instanced within the top module, while the *Alarm Process* inside the purple box was coded within the top module.

Alarm Process

The *Alarm Process* compares the values stored in the *clock_time* and *alarm_time* signals at each rising edge of the clock to determine if they match. If the times match, and the alarm is toggled on, then it will set boolean variables that are local to the process to true for as long as the alarm is still toggled on. While the boolean flags remain set, the LEDs beneath the active seven segment displays (LED thirteen down to six), and the audio pin (AUD_SD) will toggle on and off with each rising edge of the one second clock.. The conditions driving this behavior will remain true until the Alarm toggle is turned off by setting switch 1 low, at which point the sound and flashing lights will remain off until the alarm is again set, and the current time matches that of the set alarm time.

One-Second Clock Divider

The one-second clock divider takes the boards 100Mhz clock signal as input, and outputs a clock signal at 1Hz. This was implemented with a counter that uses an integer signal called *count*, and a temporary binary signal called *temp*. At every rising edge clock pulse of the 100MHz input clock, *count* is incremented by one until it reaches 50,000,000. At this point



```
-- 1s clock
signal clk_divided : std_logic;
-- To hold alarm time
signal alarm_hours_msb : STD_LOGIC_VECTOR(3 downto 0);
signal alarm_hours_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal alarm_mins_msb : STD_LOGIC_VECTOR(3 downto 0);
signal alarm_mins_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal alarm_sec_msb : STD_LOGIC_VECTOR(3 downto 0);
signal alarm_sec_lsb : STD_LOGIC_VECTOR(3 downto 0);
-- To hold clock time
signal clock_hours_msb : STD_LOGIC_VECTOR(3 downto 0);
signal clock_hours_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal clock_mins_msb : STD_LOGIC_VECTOR(3 downto 0);
signal clock_mins_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal clock_sec_msb : STD_LOGIC_VECTOR(3 downto 0);
signal clock_sec_lsb : STD_LOGIC_VECTOR(3 downto 0);
-- To hold running clock time
signal run_clock_hours_msb : STD_LOGIC_VECTOR(3 downto 0);
signal run_clock_hours_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal run_clock_mins_msb : STD_LOGIC_VECTOR(3 downto 0);
signal run_clock_mins_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal run_clock_sec_msb : STD_LOGIC_VECTOR(3 downto 0);
signal run_clock_sec_lsb : STD_LOGIC_VECTOR(3 downto 0);
-- To hold time change
signal set_clock_hours_msb : STD_LOGIC_VECTOR(3 downto 0);
signal set_clock_hours_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal set_clock_mins_msb : STD_LOGIC_VECTOR(3 downto 0);
signal set_clock_mins_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal set_clock_sec_msb : STD_LOGIC_VECTOR(3 downto 0);
signal set_clock_sec_lsb : STD_LOGIC_VECTOR(3 downto 0);
-- Numbers to be displayed
signal disp_hours_msb : STD_LOGIC_VECTOR(3 downto 0);
signal disp_hours_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal disp_mins_msb : STD_LOGIC_VECTOR(3 downto 0);
signal disp_mins_lsb : STD_LOGIC_VECTOR(3 downto 0);
signal disp_sec_msb : STD_LOGIC_VECTOR(3 downto 0);
signal disp_sec_lsb : STD_LOGIC_VECTOR(3 downto 0);
-- Used in comparison for sounding alarm
signal clock_time : std_logic_vector(23 downto 0);
signal alarm_time : std_logic_vector(23 downto 0);
-- Link button inputs to debounced output
signal hrs_up : STD_LOGIC;
signal hrs_dwn : STD_LOGIC;
signal mins_up : STD_LOGIC;
signal mins_dwn : STD_LOGIC;
signal sec_reset : STD_LOGIC;
```

Figure 2. Top Module Signals

the input clock is halfway to 100MHz, and the *temp* variable will flipped to *not temp*. This leads to *temp* being high for half of the time, and low for the remainder. The result of this is that the output clock signal will now have a positive or negative event every half second, resulting in a full period of one second. The value of *Temp* is then assigned to the output *CLKout* which is then connected to the necessary components in the top module using the clk_divided signal.

Button Debouncer

In order to use the board's push buttons, a button debouncer was required. This is because when the button is pressed the signal is usually is not clean, but instead "bounces" up and down for a short time as illustrated to the right at the bottom of Figure 3. The button debouncing component uses two flip-flops called FF1 and FF2 respectively. The button's input value is assigned to the input of FF1,and the output of FF1 is assigned to the input of FF2. The outputs of both flip-flops are combined using and exclusive or (XOR) and then assigned to an internal signal named *reset*. While the value of *reset* is low, a counter will increment at the positive edge of each clock pulse, and if the value of *reset* is high, the count will reset back to zero. Once that count reaches 1,000,0000 (10 milliseconds), the button's input is considered stable and the output of FF2 is assigned to the *btn_out* output port to be used by both the *set clock* and *set alarm* modules. The logic for the debouncer was derived from the information in source [6]. In the top module, each of the five push buttons are linked to their own instance of a debouncer.

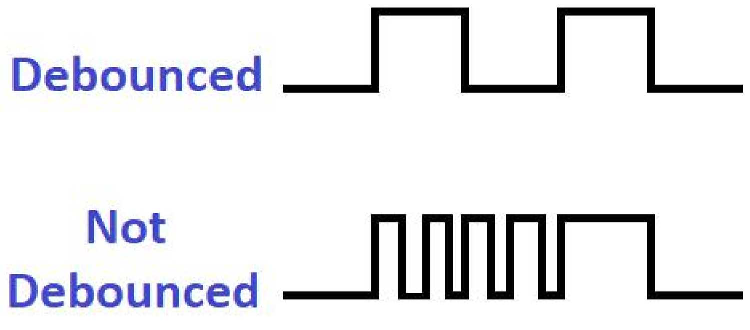


Figure 3. Button Debounce

Toggling Input Modes

The first three switches are used to accept user input for the three following functions respectively; Allow the user to set the time (SW[0]), allow the user to set the time that the alarm will go off (SW[1]), and toggle if the alarm is on (SW[2]). In the event that switch zero and switch 1 are toggled at the same time, SW[0] has priority as shown in Figure 1., where the multiplexer select input is assigned according to the truth table to determine which values are outputted to the seven segment displays.

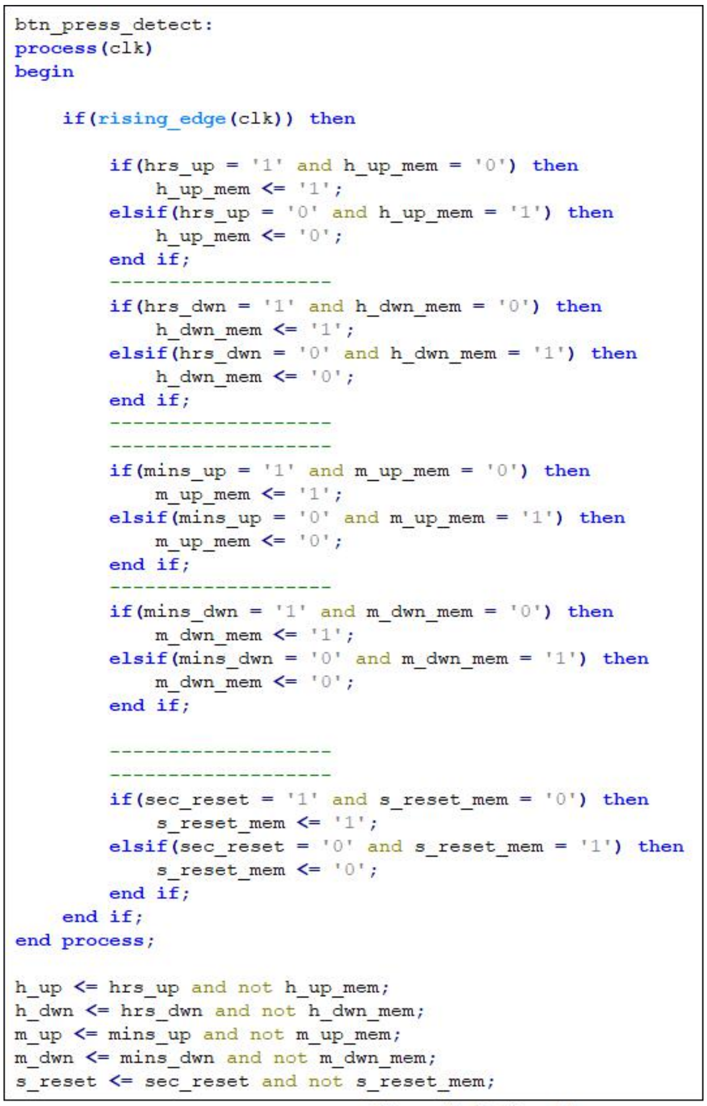
The output LEDs are split into two separate groups. The first group consists of LED[2] down to LED[0]. Each of these three LEDs correspond to switches SW[2] down to SW[0] respectively, and are illuminated when the corresponding switch is toggled. LEDs six through thirteen are linked to the alarm process as described above. These LEDs were chosen because they are directly under the six active seven segment display outputs, i.e., the displays that show the current hours, minutes, and seconds. These LEDs are used in order to give a visual indicator to the user in conjunction with the audio output, that the alarm has been activated.

The three possible display options are split into three separate components, one for running the clock, one for setting the current time, and one for setting the alarm time. The *clock runner* and *clock setter* modules each share the same input and output signal for the time. Both have their own internal clock signal that is assigned to the output value via the *current time* signal in the top module. This allows the current time to be preserved when switching between the active clock and setting mode. Their actual functionality only executes if their *enable* value, which is linked to SW[0], is toggled on each rising edge of the one second clock. The *enable* value for the clock runner is considered set when SW[0] is low, and the *enable* value for the clock setter is set when SW[0] is high. For both the clock setter and clock runner, while they are not enabled, the value of the current time is assigned to the module's respective internal time signal. This ensures that when the component is enabled, it picks up from the current time as determined by the most recently outputted time of the opposite component. If set clock is not enabled, the clock runner process is always running. As a result, even when the *set alarm* module is pushing its time values to the display, the current time runs in the background so that when the mode is toggled back, the time is still current.

For all three time modules, the value of each position is split into six separate 4-bit values to represent the ones and tens position for value of hours, minutes, and seconds respectively. In order to make the comparison of the current time and the set alarm time easier, these values are combined in two 24-bit signals called *clock_time* and *alarm_time* respectively. If the alarm is toggled on, these two values are compared at every rising edge of the 100MHz clock to determine if they equal in the alarm process. These values are assigned outside of the time setter and alarm time setter components as well to ensure they are always up to date.

Button State Memory

The five push buttons are used as inputs for both the *clock setter* and *alarm setter* modules. After being debounced, the buttons are then linked to an internal signal within its own process. Button input is accepted at the rising edge of its debounced signal so that the clock value increments / decrements only once per button press. In order to check for the rising edge of the button's signal without conflicting with the clock signal, a button memory state was required as shown in Figure 4. At every rising edge of the clock if a button input is detected, the signal goes high, and another signal is used to remember the state of the button at the previous clock pulse. If the process detects a high button press signal, and the memory is not high (i.e. it doesn't "remember" the button still being pressed down) the memory signal will go high and will remain so until the button is released. The signal which the other process uses to determine button input is then assigned the value of the button press logically anded with the inverted value stored in its 'memory.' This allows the process to detect the rising edge of the button press at the rising edge of every clock pulse. As a result, each button press increments or decrements the value of minutes or hours at the rate the user presses the buttons. For the time setter module, the center button will reset the seconds value back to zero.



```
btn_press_detect:
process(clk)
begin

    if(rising_edge(clk)) then

        if(hrs_up = '1' and h_up_mem = '0') then
            h_up_mem <= '1';
        elsif(hrs_up = '0' and h_up_mem = '1') then
            h_up_mem <= '0';
        end if;
        -------------------
        if(hrs_dwn = '1' and h_dwn_mem = '0') then
            h_dwn_mem <= '1';
        elsif(hrs_dwn = '0' and h_dwn_mem = '1') then
            h_dwn_mem <= '0';
        end if;
        -------------------
        -------------------
        if(mins_up = '1' and m_up_mem = '0') then
            m_up_mem <= '1';
        elsif(mins_up = '0' and m_up_mem = '1') then
            m_up_mem <= '0';
        end if;
        -------------------
        if(mins_dwn = '1' and m_dwn_mem = '0') then
            m_dwn_mem <= '1';
        elsif(mins_dwn = '0' and m_dwn_mem = '1') then
            m_dwn_mem <= '0';
        end if;

        -------------------
        -------------------
        if(sec_reset = '1' and s_reset_mem = '0') then
            s_reset_mem <= '1';
        elsif(sec_reset = '0' and s_reset_mem = '1') then
            s_reset_mem <= '0';
        end if;
    end if;
end process;

h_up <= hrs_up and not h_up_mem;
h_dwn <= hrs_dwn and not h_dwn_mem;
m_up <= mins_up and not m_up_mem;
m_dwn <= mins_dwn and not m_dwn_mem;
s_reset <= sec_reset and not s_reset_mem;
```

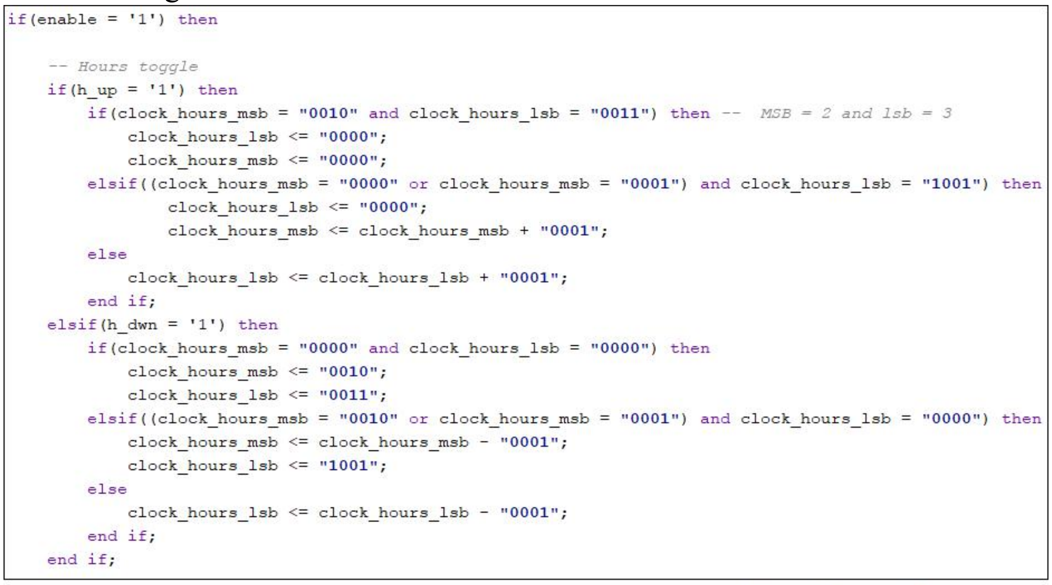
Figure 4. Button Memory

Toggling Which Output to Display

The numbers that will be displayed are determined by the input combination of switches SW[0] and SW[1]. If both are off, the regular clock will be displayed and updated at every rising edge of the one-second clock pulse. As for the *clock setter* and *alarm setter*, the clock setter has priority as shown in Figure 1. If SW[0] is set high, then the clock will display the values as determined by the clock setter component regardless of the state of SW[1]. If SW[1] is high and SW[0] is low, then the values outputted by the *alarm setter* component will be routed to the displays.

Setting the Time

Both the *clock setter* and *alarm setter* are set up the same way in regards to how the time is changed, with the only exception being that the alarm setter doesn't touch the value of seconds. For the alarm, the seconds will always be zero because it is unnecessary to change this value. In order to prevent driver conflicts, the ability to adjust the time is split into three sections. One section to change the hours up or down, one for the minutes, and a final one that resets the seconds down to zero in the case of the clock setter. For each section, an if/else if statement is used to again, prevent the issue of having the different values cause a driver conflict. If the corresponding button to increment the hours or minutes is detected as being high, then the values for each section will increase accordingly with each button press. If the corresponding button to increment values is not detected as high, then the else statement allows for the corresponding button to decrement the value for its time section. When the value is decremented to zero, the next button press will underflow to the max allowed value (23 for hours, 59 for minutes). Likewise, if the value is currently at its max, the value will then overflow back to zero on the next button press. How this is implemented for the clock's hour value can be seen below in Figure 5.



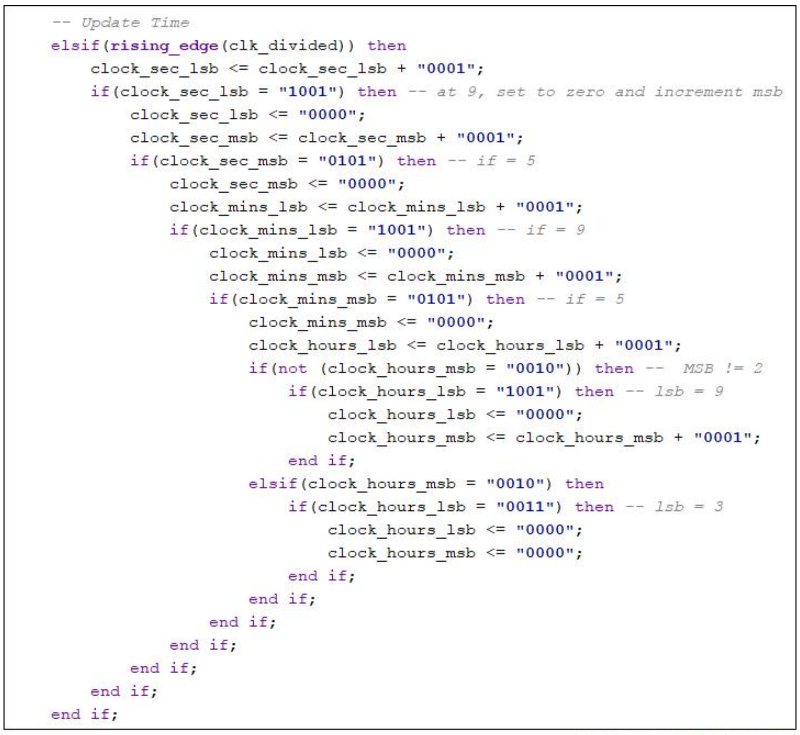
```
if(enable = '1') then

    -- Hours toggle
    if(h_up = '1') then
        if(clock_hours_msb = "0010" and clock_hours_lsb = "0011") then --  MSB = 2 and lsb = 3
            clock_hours_lsb <= "0000";
            clock_hours_msb <= "0000";
        elsif((clock_hours_msb = "0000" or clock_hours_msb = "0001") and clock_hours_lsb = "1001") then
                clock_hours_lsb <= "0000";
                clock_hours_msb <= clock_hours_msb + "0001";
        else
            clock_hours_lsb <= clock_hours_lsb + "0001";
        end if;
    elsif(h_dwn = '1') then
        if(clock_hours_msb = "0000" and clock_hours_lsb = "0000") then
            clock_hours_msb <= "0010";
            clock_hours_lsb <= "0011";
        elsif((clock_hours_msb = "0010" or clock_hours_msb = "0001") and clock_hours_lsb = "0000") then
            clock_hours_msb <= clock_hours_msb - "0001";
            clock_hours_lsb <= "1001";
        else
            clock_hours_lsb <= clock_hours_lsb - "0001";
        end if;
    end if;
```

Figure 5. Setting Time

Running the Clock

The *clock runner* does not run based on user input aside from switch zero being linked to its *enable* value. Because the clock setter is enabled when switch SW[0] is high, the clock runner's *enable* is active low to correspond with SW[0] being toggled off / low. When enabled, the value of the seconds display's least significant four bits will be incremented by one. After this, a series of checks is performed using nested if / else statements as shown in Figure 6. Once the most least significant four bits reaches nine for both seconds and minutes, the least significant four bits are reset to zero, and the four most significant bits are incremented by one. As soon as the most significant four bits for both seconds and minutes are equal to five and their least significant bits are equal to nine, then both are



```
-- Update Time
elsif(rising_edge(clk_divided)) then
    clock_sec_lsb <= clock_sec_lsb + "0001";
    if(clock_sec_lsb = "1001") then -- at 9, set to zero and increment msb
        clock_sec_lsb <= "0000";
        clock_sec_msb <= clock_sec_msb + "0001";
        if(clock_sec_msb = "0101") then -- if = 5
            clock_sec_msb <= "0000";
            clock_mins_lsb <= clock_mins_lsb + "0001";
            if(clock_mins_lsb = "1001") then -- if = 9
                clock_mins_lsb <= "0000";
                clock_mins_msb <= clock_mins_msb + "0001";
                if(clock_mins_msb = "0101") then -- if = 5
                    clock_mins_msb <= "0000";
                    clock_hours_lsb <= clock_hours_lsb + "0001";
                    if(not (clock_hours_msb = "0010")) then --  MSB != 2
                        if(clock_hours_lsb = "1001") then -- lsb = 9
                            clock_hours_lsb <= "0000";
                            clock_hours_msb <= clock_hours_msb + "0001";
                        end if;
                    elsif(clock_hours_msb = "0010") then
                        if(clock_hours_lsb = "0011") then -- lsb = 3
                            clock_hours_lsb <= "0000";
                            clock_hours_msb <= "0000";
                        end if;
                    end if;
                end if;
            end if;
        end if;
    end if;
end if;
```

Figure 6. Clock Runner

reset to zero and the next value in line is incremented by one. For the hours, because this is a twenty four hour clock instead of twelve hours, if the four most significant bits are equal to zero or one, then the most significant four bits are incremented when the four least significant bits are reset to zero after reaching nine. If the hour reaches twenty three at the time they are checked (when minutes and seconds both equal fifty-nine), then they are both reset to zero along with the rest of the clock, and the entire process restarts.

Seven Segment Display
The first 6 seven-segment displays are used to display the current time. As per the official documentation for the Nexys 4 ddr board [1], the seven-segment displays are connected together using a common anode. In order to drive the displays, they must be driven one at a time at a very fast rate. To accomplish this, another clock divider was used to obtain the desired output clock frequency. A case statement was then used within the driver to cycle through the displays sequentially at the desired rate, assigning the correct value to the corresponding display. This all happens fast enough so that while only one seven-segment display is active at one at a time, it happens so fast that to the human eye it appears they are all constantly active at the same time.

Segment / number	A	B	C	D	E	F	G
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	1	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	1	1	0	0

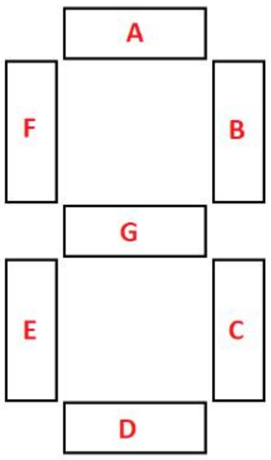
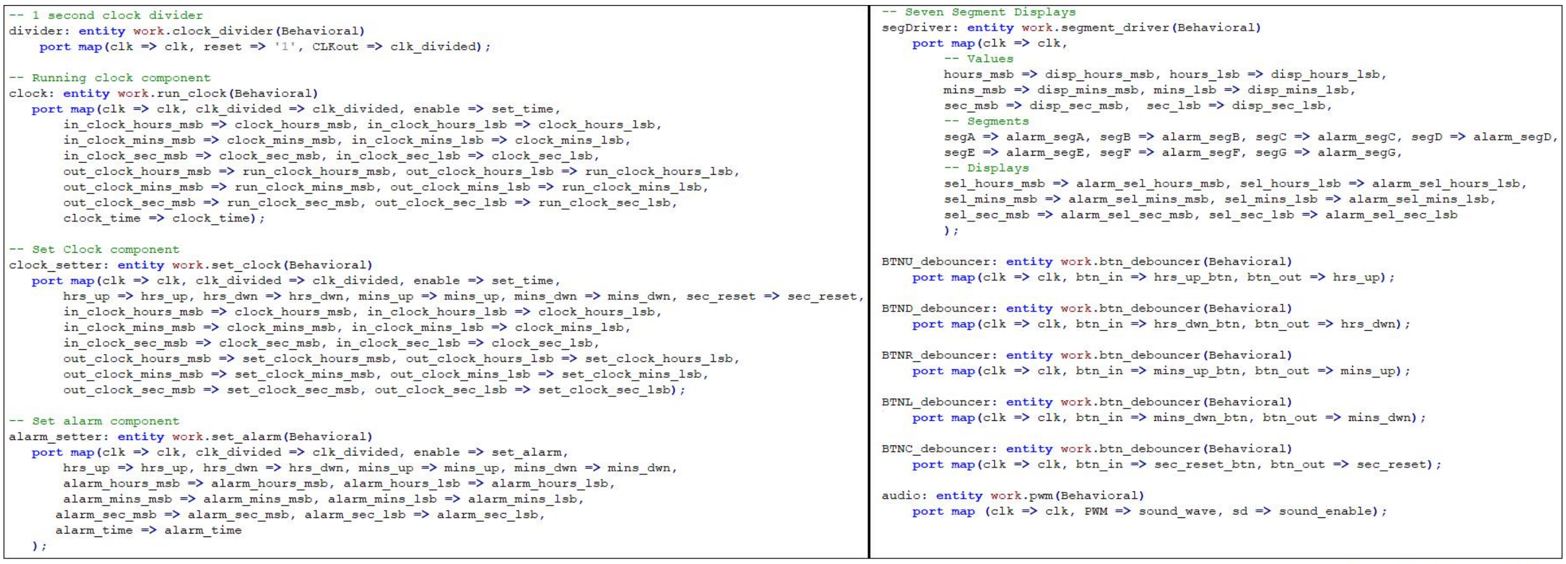


Figure 7. Seven Segment Display

The segments for the displays were configured based on the table and diagram shown above in Figure 7. The segments are active low, and therefore in order to turn off the segments that were not used, a value of '1' was assigned to it. The table was created to determine the binary combination for each of the possible numerical time outputs ranging from zero to nine. They were then coded in the seven-segment decoder module using a case statement such that when the input value matched the specific case, the segments would be driven with the appropriate binary code.

Audible Alarm
The Audio portion was created by generating pulse width modulated waveform. The source for this came from reference [4] which provided the LUT used to generate a sine wave, as well as the VHDL source code to implement the PWM signal. The code was modified slightly from its original design which used switches to generate frequencies and thus different tones. In this design, the frequency was set to a constant value, and the SD pin is driven from the top module as explained earlier in the alarm process.

All of these components were then instanced into the top module as shown below in Figure 8., and connected to either their corresponding input / output pins, or signals which can be better visualized in Figure 1.



```vhdl
-- 1 second clock divider
divider: entity work.clock_divider(Behavioral)
    port map(clk => clk, reset => '1', CLKout => clk_divided);

-- Running clock component
clock: entity work.run_clock(Behavioral)
   port map(clk => clk, clk_divided => clk_divided, enable => set_time,
       in_clock_hours_msb => clock_hours_msb, in_clock_hours_lsb => clock_hours_lsb,
       in_clock_mins_msb => clock_mins_msb, in_clock_mins_lsb => clock_mins_lsb,
       in_clock_sec_msb => clock_sec_msb, in_clock_sec_lsb => clock_sec_lsb,
       out_clock_hours_msb => run_clock_hours_msb, out_clock_hours_lsb => run_clock_hours_lsb,
       out_clock_mins_msb => run_clock_mins_msb, out_clock_mins_lsb => run_clock_mins_lsb,
       out_clock_sec_msb => run_clock_sec_msb, out_clock_sec_lsb => run_clock_sec_lsb,
       clock_time => clock_time);

-- Set Clock component
clock_setter: entity work.set_clock(Behavioral)
   port map(clk => clk, clk_divided => clk_divided, enable => set_time,
       hrs_up => hrs_up, hrs_dwn => hrs_dwn, mins_up => mins_up, mins_dwn => mins_dwn, sec_reset => sec_reset,
       in_clock_hours_msb => clock_hours_msb, in_clock_hours_lsb => clock_hours_lsb,
       in_clock_mins_msb => clock_mins_msb, in_clock_mins_lsb => clock_mins_lsb,
       in_clock_sec_msb => clock_sec_msb, in_clock_sec_lsb => clock_sec_lsb,
       out_clock_hours_msb => set_clock_hours_msb, out_clock_hours_lsb => set_clock_hours_lsb,
       out_clock_mins_msb => set_clock_mins_msb, out_clock_mins_lsb => set_clock_mins_lsb,
       out_clock_sec_msb => set_clock_sec_msb, out_clock_sec_lsb => set_clock_sec_lsb);

-- Set alarm component
alarm_setter: entity work.set_alarm(Behavioral)
   port map(clk => clk, clk_divided => clk_divided, enable => set_alarm,
       hrs_up => hrs_up, hrs_dwn => hrs_dwn, mins_up => mins_up, mins_dwn => mins_dwn,
       alarm_hours_msb => alarm_hours_msb, alarm_hours_lsb => alarm_hours_lsb,
       alarm_mins_msb => alarm_mins_msb, alarm_mins_lsb => alarm_mins_lsb,
      alarm_sec_msb => alarm_sec_msb, alarm_sec_lsb => alarm_sec_lsb,
      alarm_time => alarm_time
   );
```

```vhdl
-- Seven Segment Displays
segDriver: entity work.segment_driver(Behavioral)
    port map(clk => clk,
        -- Values
        hours_msb => disp_hours_msb, hours_lsb => disp_hours_lsb,
        mins_msb => disp_mins_msb, mins_lsb => disp_mins_lsb,
        sec_msb => disp_sec_msb,  sec_lsb => disp_sec_lsb,
        -- Segments
        segA => alarm_segA, segB => alarm_segB, segC => alarm_segC, segD => alarm_segD,
        segE => alarm_segE, segF => alarm_segF, segG => alarm_segG,
        -- Displays
        sel_hours_msb => alarm_sel_hours_msb, sel_hours_lsb => alarm_sel_hours_lsb,
        sel_mins_msb => alarm_sel_mins_msb, sel_mins_lsb => alarm_sel_mins_lsb,
        sel_sec_msb => alarm_sel_sec_msb, sel_sec_lsb => alarm_sel_sec_lsb
        );

BTNU_debouncer: entity work.btn_debouncer(Behavioral)
    port map(clk => clk, btn_in => hrs_up_btn, btn_out => hrs_up);

BTND_debouncer: entity work.btn_debouncer(Behavioral)
    port map(clk => clk, btn_in => hrs_dwn_btn, btn_out => hrs_dwn);

BTNR_debouncer: entity work.btn_debouncer(Behavioral)
    port map(clk => clk, btn_in => mins_up_btn, btn_out => mins_up);

BTNL_debouncer: entity work.btn_debouncer(Behavioral)
    port map(clk => clk, btn_in => mins_dwn_btn, btn_out => mins_dwn);

BTNC_debouncer: entity work.btn_debouncer(Behavioral)
    port map(clk => clk, btn_in => sec_reset_btn, btn_out => sec_reset);

audio: entity work.pwm(Behavioral)
    port map (clk => clk, PWM => sound_wave, sd => sound_enable);
```

Figure 8. Top Components

What is left that could not be implemented

The only originally planned feature that was not implemented is the ability to snooze the alarm. Multiple different approaches using both switch three and the CPU RESET button did not work properly. The idea was that when snooze was toggled, the signal driving the alarm outputs would be toggled off for ten minutes while a count incremented with the rising edge of the one-second clock until the count reached six hundred. Once the count reached this point the LEDs and alarm sound would resume until the alarm was either snoozed again, or completely toggled off. Unfortunately after multiple different approaches and tests, the feature wasn't able to be implemented properly in time to submit this project and return the FPGA board which had been borrowed from the University.

Difficulties / challenges faced

A major challenge encountered while working on this project occurred as a result of initially approaching the design from a software perspective. This resulted in seemingly sound logic not working in practice, and taking up valuable time troubleshooting. When I returned to the drawing board so-to-speak, and instead began by mapping out the hardware logic and their interactions with other elements as shown in Figure 1 before proceeding with the VHDL implementation, the project began to flow much more smoothly.

My initial design was based on the diagram shown below in Figure 9. I was originally supposed to be run with a single clock module that was instanced for each seven segment display. Each clock had an enable input that would be triggered by the previous module as a sort of up-counter. However, issues arose when trying to determine if the value of the clock should change based on the one second clock signal, or by user input.

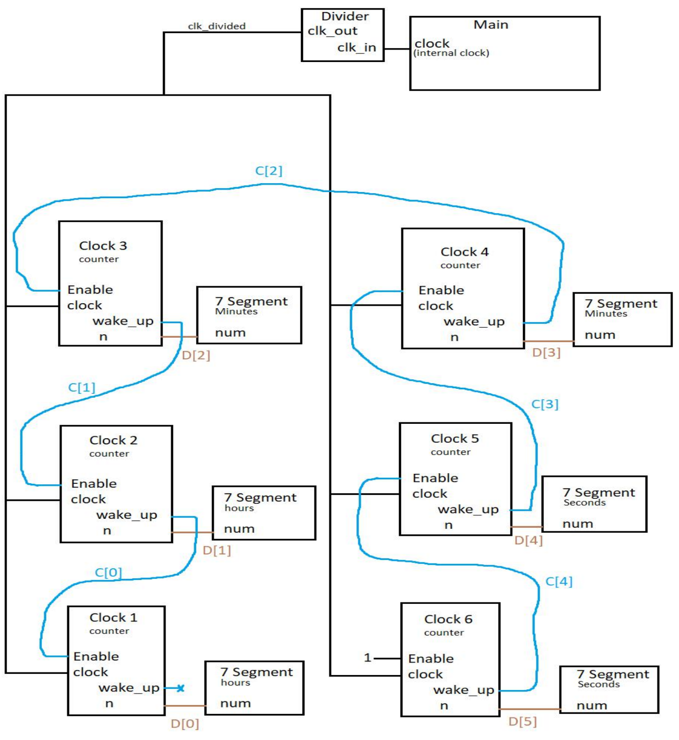


Figure 9. Original clock design

Separating the process for both running the clock, and setting the clock proved to be the solution to resolve the issues and driver conflicts that occurred with this first design approach.

As described in the previous section, the rising edge of button presses are determined in their own processes and linked to a corresponding signal and memory state (Figure 4). This was required because the process that uses these inputs are also based on the state of the 100MHz clock. Synthesis does not allow an else statement to follow the check for the rising edge of a clock signal, so an alternative was required to detect the rising edge of a button press. A memory state was used in conjunction with the actual input signal as described in the previous section. However, initially only one memory state was used. The thinking behind this was that only one button press would be allowed at a time, preventing multiple inputs conflicting. However, this actually caused a conflict the button presses to decrement the time for both minutes and hours worked as intended, but the buttons to increment the time had no effect. To remedy this, it was determined that each button required its own memory state. Once this was implemented, the button presses worked as intended.

The most difficult part of the project was understanding pulse width modulation, or more specifically how to use that to output a tone for the alarm. Reference 4 helped some, but applying the design in VHDL still proved difficult. Fortunately I was able to repurpose the code found in source 5, which was a tone generator. All that was required was to remove the input, and apply a constant frequency for my desired tone. Then the SD output was toggled on/off with each rising edge of the one-second clock in the top module when the time came to sound the alarm.

Another issue initially encountered, was that while the test bench wave forms showed the clock counting up properly, as shown below in Figures 10 and 11, issues arose when testing the design on the actual board. The seconds values would count up as intended, yet when the fifty-nine second mark was reached, the seconds would then reset to zero, but the minutes value would not increment as they were supposed to. This issue was resolved by adding additional connecting signals so that both the clock runner and clock setter components would not conflict. The components each have their own internal input and output ports, which are read into/from internal buffer signals which have their values updated internally within the component.

Additional Figures

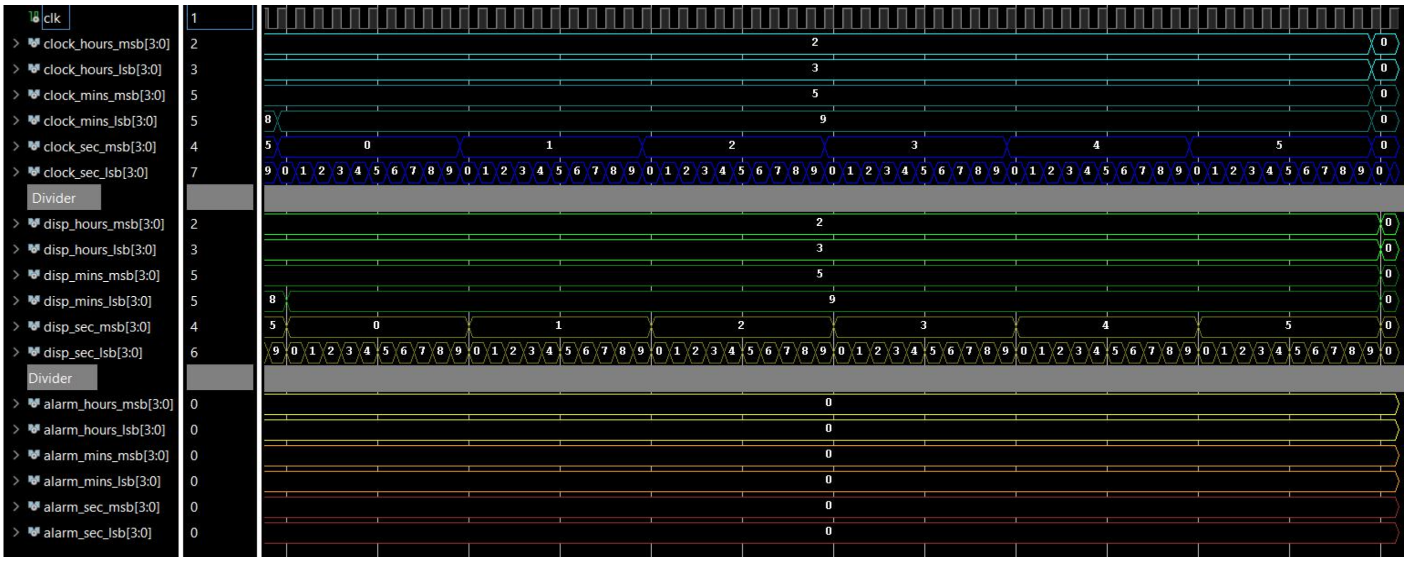


Figure 10. Beginning of test bench

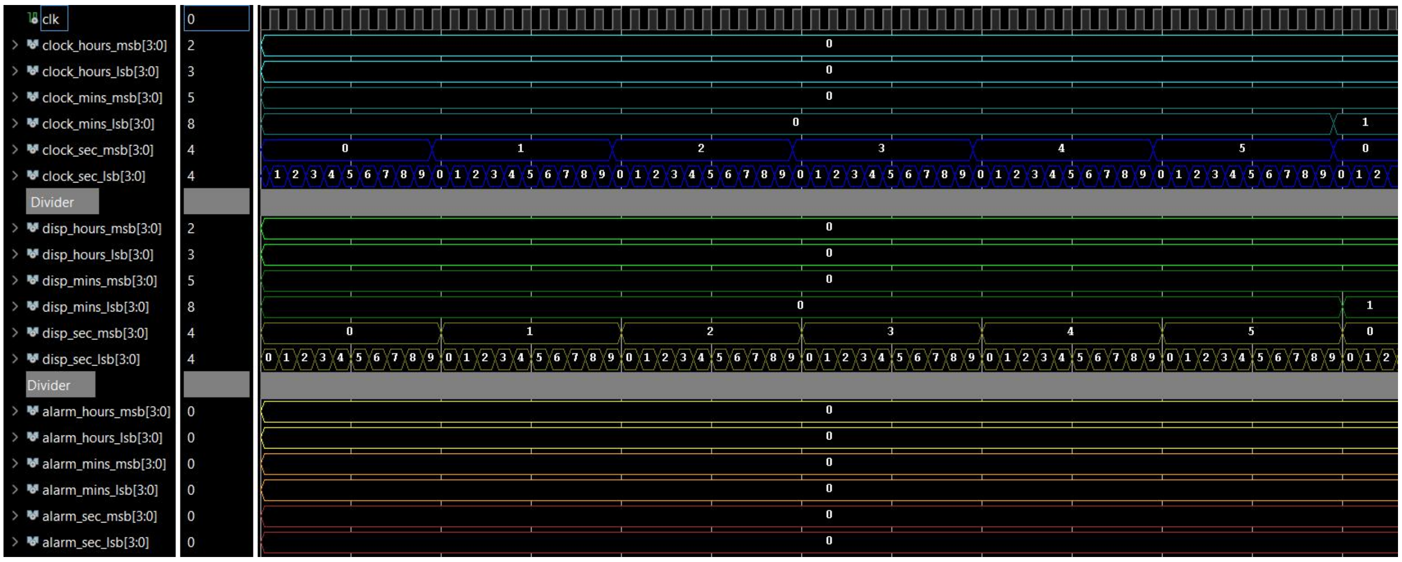
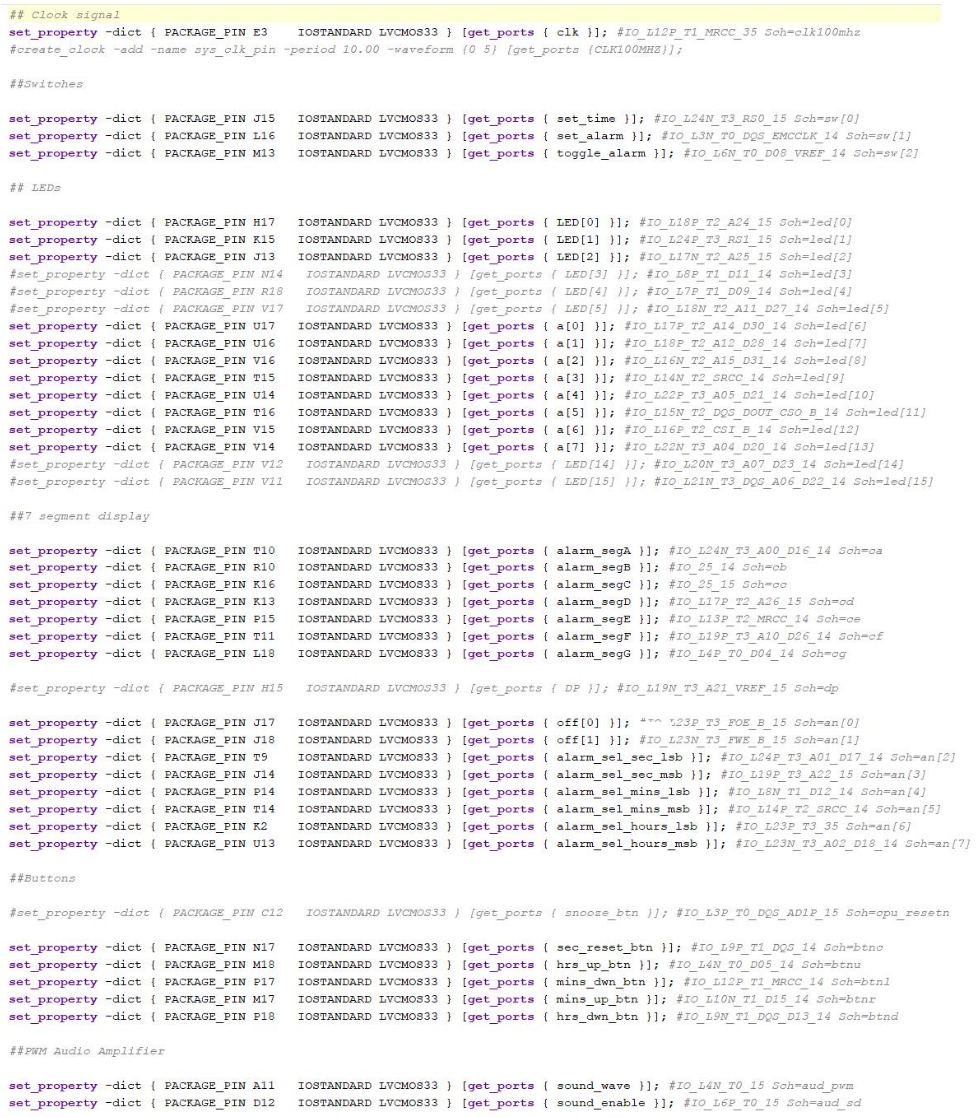


Figure 11. End of test bench



```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { set_time }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { set_alarm }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { toggle_alarm }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]

## LEDs

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { a[0] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { a[1] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { a[2] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { a[3] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { a[4] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { a[5] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { a[6] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { a[7] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

##7 segment display

set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { alarm_segA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { alarm_segB }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { alarm_segC }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { alarm_segD }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { alarm_segE }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { alarm_segF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { alarm_segG }]; #IO_L4P_T0_D04_14 Sch=cg

#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { off[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { off[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_sec_lsb }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_sec_msb }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_mins_lsb }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_mins_msb }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_hours_lsb }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { alarm_sel_hours_msb }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { snooze_btn }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { sec_reset_btn }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { hrs_up_btn }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { mins_dwn_btn }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { mins_up_btn }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { hrs_dwn_btn }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

##PWM Audio Amplifier

set_property -dict { PACKAGE_PIN A11   IOSTANDARD LVCMOS33 } [get_ports { sound_wave }]; #IO_L4N_T0_15 Sch=aud_pwm
set_property -dict { PACKAGE_PIN D12   IOSTANDARD LVCMOS33 } [get_ports { sound_enable }]; #IO_L6P_T0_15 Sch=aud_sd
```

Figure 12. Constrains Used

Conclusion

This project was an extremely beneficial learning experience in that it required me to adapt my thought process while planning and executing my design. Instead of writing my code to describe my desired functionality, I needed to instead plan how the hardware would interact first. Then, and only then, write code to describe that hardware. This project ultimately gave me a much better understanding of why languages like VHDL and Verilog are called hardware description languages.

The flow of the project required initial planning that mapped out the digital logic of the design and how each component would be connected to the others. A great deal was learned over the course of designing and implementing the project. The initial idea seemed straight forward, however the result required the functionality to be divided into several separate components resulting in each function having its own component, and in a few cases even having their own sub-components to properly implement. The final result was a highly modular design.

The final result is a device that was not just created for educational purposes, but something that can be used for practical purposes as well. After the clock was functional, it was tested over a twenty four hour period, and at the end of this period the clock time still matched the actual current time down to the second.

-References-

[1] https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual
- Official reference for the Nexys 4 ddr board

[2] https://insights.sigasi.com/tech/four-and-half-ways-write-vhdl-instantiations/
- VHDL components

[3] https://www.youtube.com/watch?v=2Bvf1GdutFI
- Seven segment display tutorial

[4] https://www.digikey.com/eewiki/pages/viewpage.action?pageId=20939345
- PWM logic

[5] https://github.com/sburg3/pwm-audio
- LUT memory for generating sine wave, as well as vhdl for pwm generation

[6] https://www.digikey.com/eewiki/pages/viewpage.action?pageId=4980758
- Button debouncing