Documentation

Team Paris

24.06.2022

Team members

- Aditya Kumar
- Wiktor Kochanek
- Gordan Konevski

Introduction

Our project is a prototype of an alarm clock implemented with VHDL and a PCB design corresponding to the model. We used VHDL as a way to program our code as that allowed us to describe the system behaviour and simulate it, and then use synthesis tools in order to translate the software model into a real hardware model. We then used Xilinx Vivado to design our PCB complimenting the VHDL implementation.

Description

Our prototype is an alarm clock intended to use, for example for waking up. It counts time from the moment it is on and a time can be set on it for an alarm. When the running clock time and the set alarm time are equal, the alarm clock will buzz.

Project/Team management

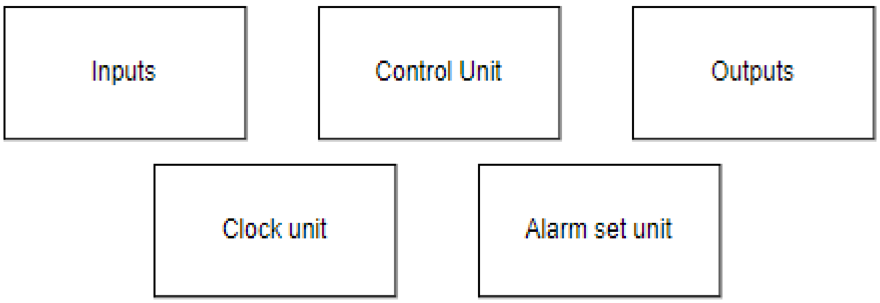
We decided to each focus on a specific field within the project, but at all points during the implementation we had communication with each other to ensure continuity of our implementations. In other words, the actual realisation of the divided tasks was done on separate workstations, but the project itself was developed in unison with the group. In the end, the vast majority of the work in the VHDL code implementation was done by Aditya. Majority of the work with Vivado and the PCB design was done by Gordan. The Xilinx synthesis and majority of documentation writing work has been done by Wiktor.

Technologies

For the purposes of VHDL and FPGA implementation we used ModelSIM, as included in Xilinx ISE. ModelSIM has a very efficient debugging environment and very efficient display of data, so it is very useful for a beginner project like this, to get used to the working environment. Xilinx ISE synthesis tool was then used to obtain schematics and move the implementation towards the FPGA direction. It was the ideal tool to use as it supported our target hardware, which is obsolete and not compatible with many modern synthesis tools. For the purposes of generating a gate and connection schematic, we used Vivado. Vivado also allows for development with VHDL, but, for reasons that are elaborated further below, its use was limited to schematic generation.

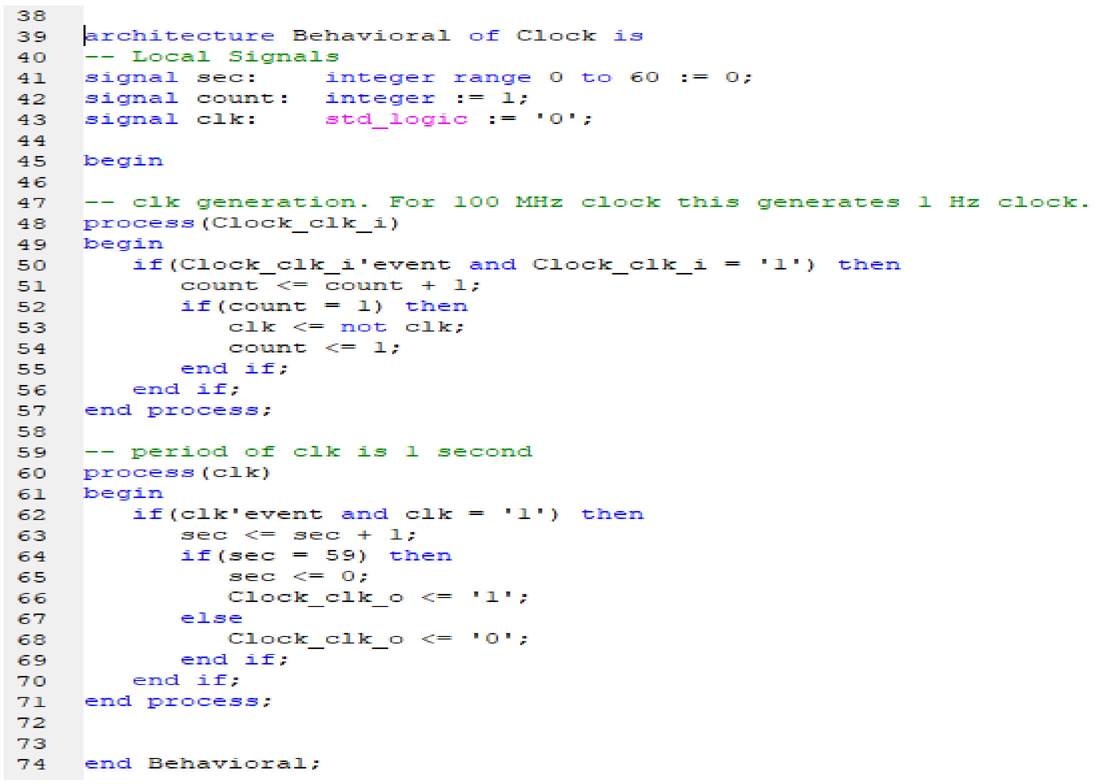
VHDL Implementation

The block diagram of our model can be seen above. Each block depicts a different component of our model. “Inputs” in this case actually references our main implementation Digital_clock.vhdl which ports and maps all other units. Our real inputs in this evaluation are as follows:



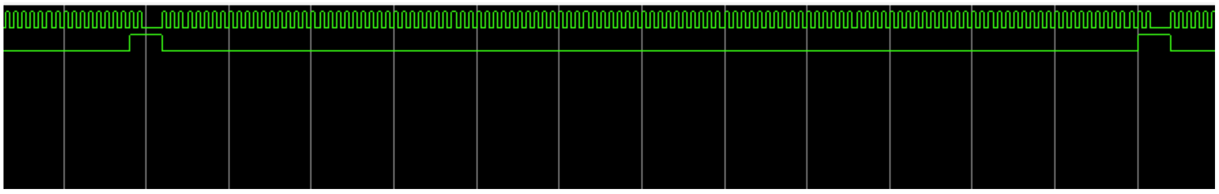
- Clock input
- Input for turning on the alarm
- Input to manually set the current clock time
- 4 inputs to set alarm time

Our clock unit refers to the program component Clock.vhdl. Very simply this program implements our second clock, but creating a counter of a period 1s. After reaching 59s on the counter, next time it were to increase, it instead sends a pulse, which then is used within the Controller.vhdl to count current time. All of our time related operations and representations are based on this clock implementation. A relevant code snippet as well as the testbench result can be seen below.

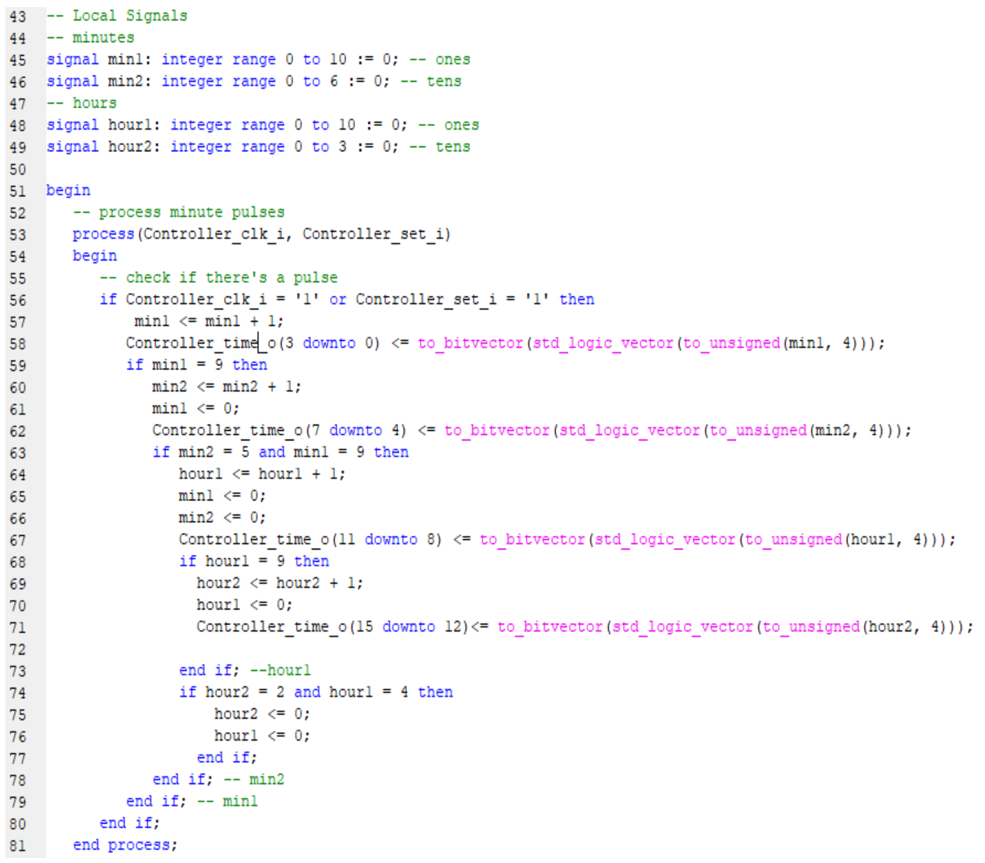


```vhdl

architecture Behavioral of Clock is
-- Local Signals
signal sec:     integer range 0 to 60 := 0;
signal count:   integer := 1;
signal clk:     std_logic := '0';

begin

-- clk generation. For 100 MHz clock this generates 1 Hz clock.
process(Clock_clk_i)
begin
   if(Clock_clk_i'event and Clock_clk_i = '1') then
      count <= count + 1;
      if(count = 1) then
         clk <= not clk;
         count <= 1;
      end if;
   end if;
end process;

-- period of clk is 1 second
process(clk)
begin
   if(clk'event and clk = '1') then
      sec <= sec + 1;
      if(sec = 59) then
         sec <= 0;
         Clock_clk_o <= '1';
      else
         Clock_clk_o <= '0';
      end if;
   end if;
end process;


end Behavioral;
```

VHDL CODE:
CLOCK

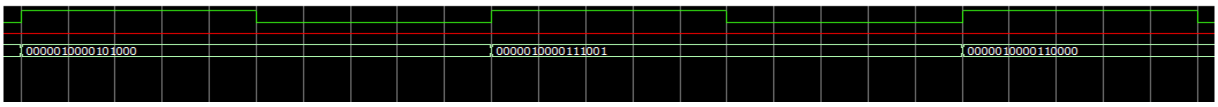

TESTBENCH : CLOCK

Our control unit refers to the program component Controller.vhdl. This program counts the current time. Following a pulse input from clock.vhdl component, the program starts counting up by minute using a counter inside an if statement. By utilising if statements within if statements triggered by counters reaching specific numbers - 10 for single minute counter, 6 for ten-minute counter, and 10 for single hour counter, as well as defining the hour 24 as a reset point, we are able to cover all the hours in a 24 hour clock. The specific values of min1, min2, hour1 and hour2 (which represent single minutes, tens minutes, single hours and tens hours respectively) are stored as a 16 bit signal, which is then used for display and for alarm checking. A relevant snippet of the code, as well as the testbench result can be seen below.



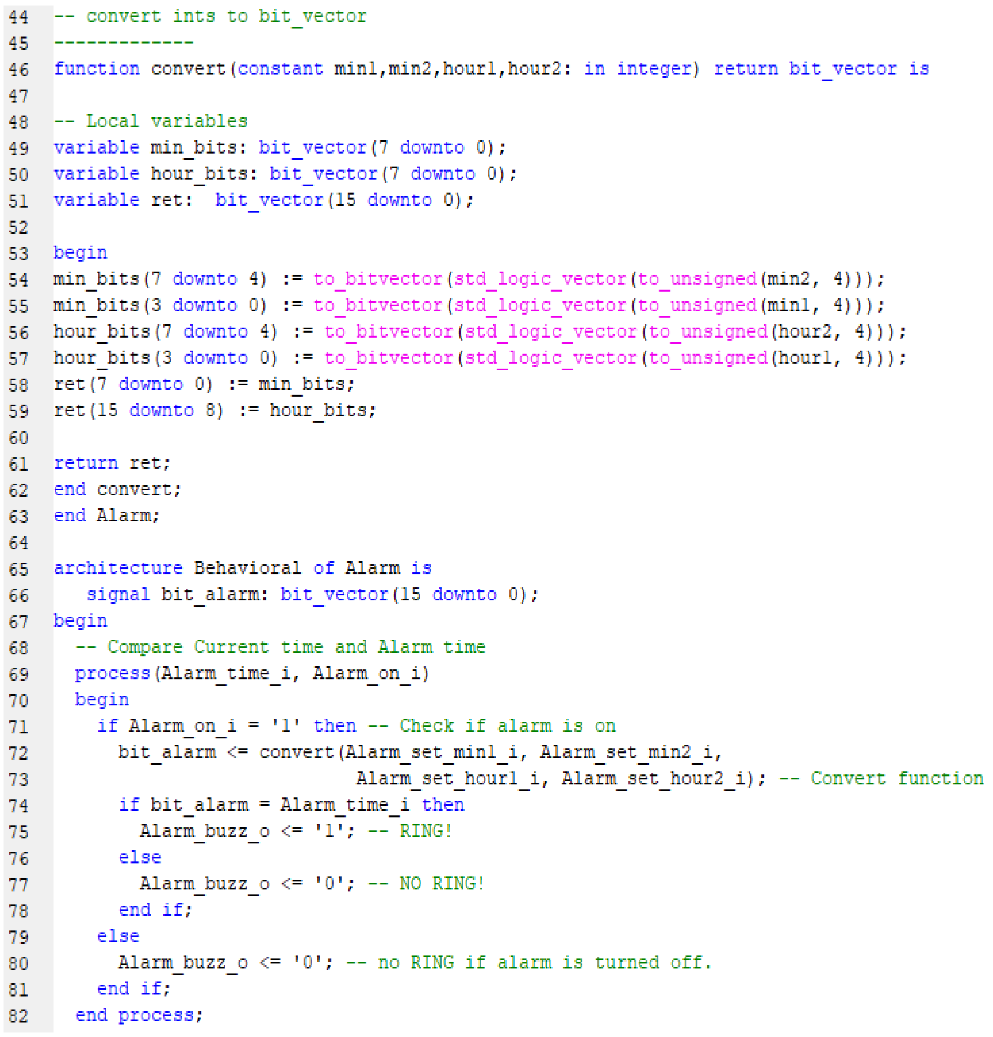
```vhdl
-- Local Signals
-- minutes
signal min1: integer range 0 to 10 := 0; -- ones
signal min2: integer range 0 to 6 := 0; -- tens
-- hours
signal hour1: integer range 0 to 10 := 0; -- ones
signal hour2: integer range 0 to 3 := 0; -- tens

begin
   -- process minute pulses
   process(Controller_clk_i, Controller_set_i)
   begin
      -- check if there's a pulse
      if Controller_clk_i = '1' or Controller_set_i = '1' then
          min1 <= min1 + 1;
         Controller_time_o(3 downto 0) <= to_bitvector(std_logic_vector(to_unsigned(min1, 4)));
         if min1 = 9 then
            min2 <= min2 + 1;
            min1 <= 0;
            Controller_time_o(7 downto 4) <= to_bitvector(std_logic_vector(to_unsigned(min2, 4)));
            if min2 = 5 and min1 = 9 then
               hour1 <= hour1 + 1;
               min1 <= 0;
               min2 <= 0;
               Controller_time_o(11 downto 8) <= to_bitvector(std_logic_vector(to_unsigned(hour1, 4)));
               if hour1 = 9 then
                 hour2 <= hour2 + 1;
                 hour1 <= 0;
                 Controller_time_o(15 downto 12)<= to_bitvector(std_logic_vector(to_unsigned(hour2, 4)));

               end if; --hour1
               if hour2 = 2 and hour1 = 4 then
                   hour2 <= 0;
                   hour1 <= 0;
                 end if;
            end if; -- min2
         end if; -- min1
      end if;
   end process;
```

VHDL CODE: CONTROLLER



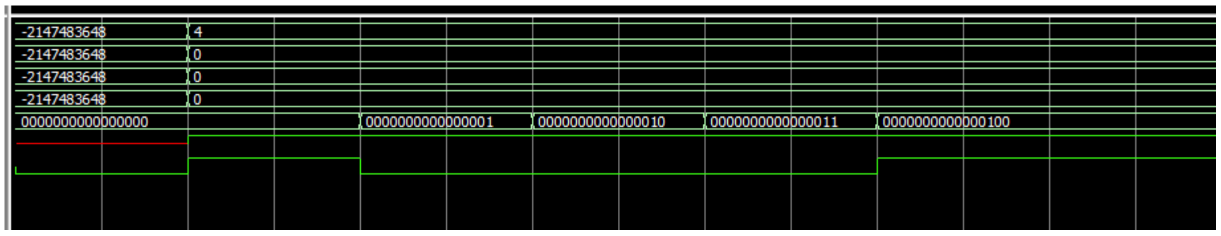
TESTBENCH : CONTROLLER

Our alarm set unit refers to the program component Alarm.vhdl. This program is responsible for setting the alarm time and activating the audio output if current time is equal to alarm time. We set an alarm time using 4 inputs. The program then, using a convert function, changes the vector inputs into bit inputs, which can be compared with current time, an output from the control unit. If the alarm set unit is on and the alarm time equals the current time, an audio output is given. If the times don't match, or if the alarm set module is set to off, no audio output is given. A Relevant snippet of the code as well as the testbench result can be seen below.



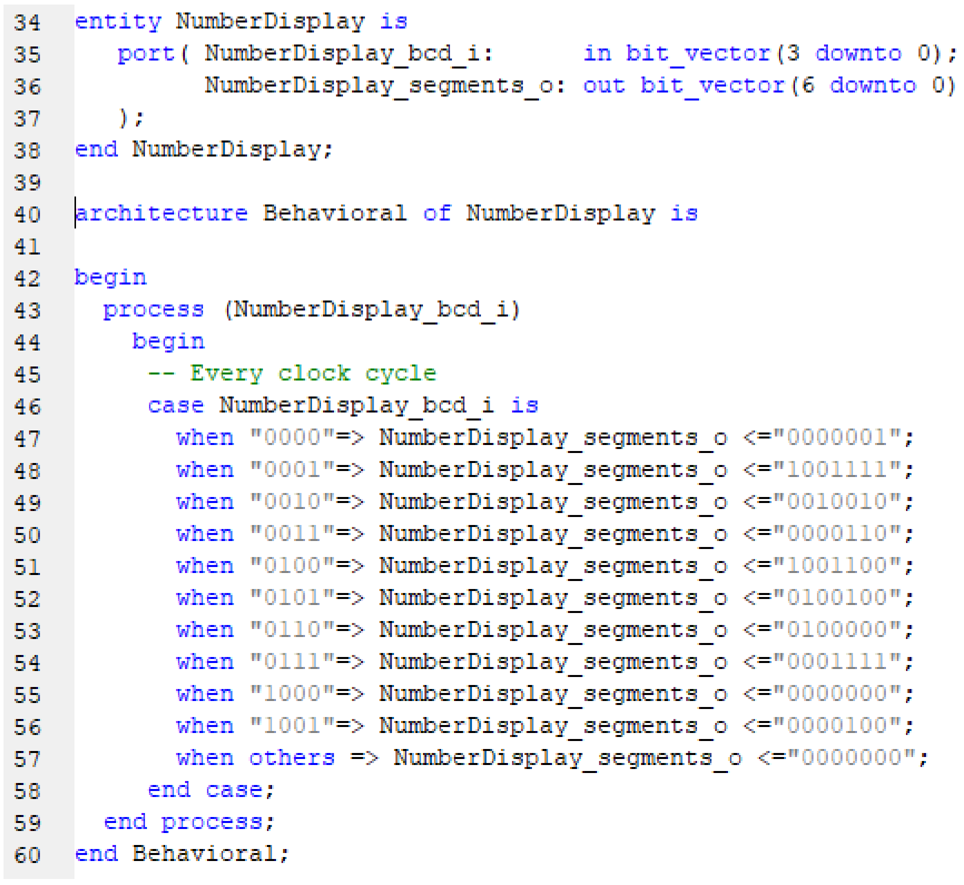
```vhdl
-- convert ints to bit_vector
-------------
function convert(constant min1,min2,hour1,hour2: in integer) return bit_vector is

-- Local variables
variable min_bits: bit_vector(7 downto 0);
variable hour_bits: bit_vector(7 downto 0);
variable ret:  bit_vector(15 downto 0);

begin
min_bits(7 downto 4) := to_bitvector(std_logic_vector(to_unsigned(min2, 4)));
min_bits(3 downto 0) := to_bitvector(std_logic_vector(to_unsigned(min1, 4)));
hour_bits(7 downto 4) := to_bitvector(std_logic_vector(to_unsigned(hour2, 4)));
hour_bits(3 downto 0) := to_bitvector(std_logic_vector(to_unsigned(hour1, 4)));
ret(7 downto 0) := min_bits;
ret(15 downto 8) := hour_bits;

return ret;
end convert;
end Alarm;

architecture Behavioral of Alarm is
   signal bit_alarm: bit_vector(15 downto 0);
begin
  -- Compare Current time and Alarm time
  process(Alarm_time_i, Alarm_on_i)
  begin
    if Alarm_on_i = '1' then -- Check if alarm is on
      bit_alarm <= convert(Alarm_set_min1_i, Alarm_set_min2_i,
                           Alarm_set_hour1_i, Alarm_set_hour2_i); -- Convert function
      if bit_alarm = Alarm_time_i then
        Alarm_buzz_o <= '1'; -- RING!
      else
        Alarm_buzz_o <= '0'; -- NO RING!
      end if;
    else
      Alarm_buzz_o <= '0'; -- no RING if alarm is turned off.
    end if;
  end process;
```

VHDL CODE: ALARM



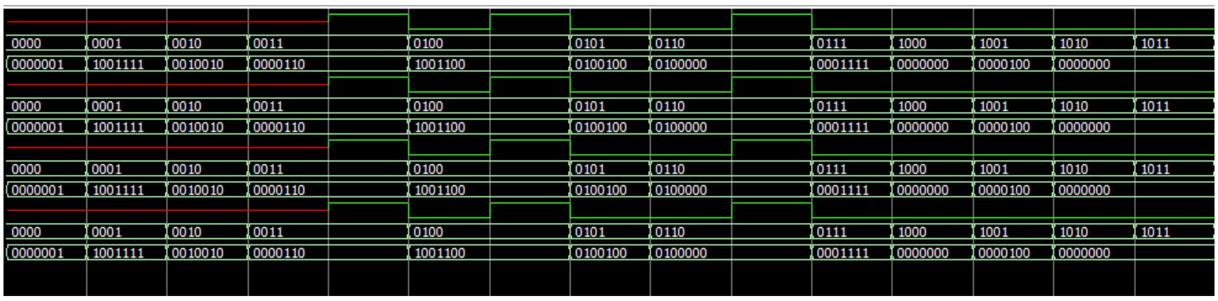
TESTBENCH : ALARM

Our output unit refers to the program component NumberDisplay.vhdl and, partially, to Alarm.vhdl. Indeed the audio output from the Alarm Set unit is considered in this unit as well. The main focus however, is on the NumberDisplay.vhdl code. This component translates the current time given by the controller unit and displays it, using 4 separate 7-segment displays. The current time is sent as a 16 bit stream, seperated into 4 segments - 15 down to 12, 11 down to 8, 7 down to 4 and 3 down to 0 - each of these bit segments are considered an input for a separate display. All displays are coded to be able to display all numbers between 0 and 9, despite not all of them requiring such ability. This is done for simplicity's sake, as a singular implementation for all 4 displays is far easier to code than 4 separate implementations. Each arriving segment of the 16 bit stream is, in effect, a 4 bit signal which is then translated to numbers on the 7-segment display using case-is-when statements. A relevant snippet of the code as well as the testbench result can be seen below.



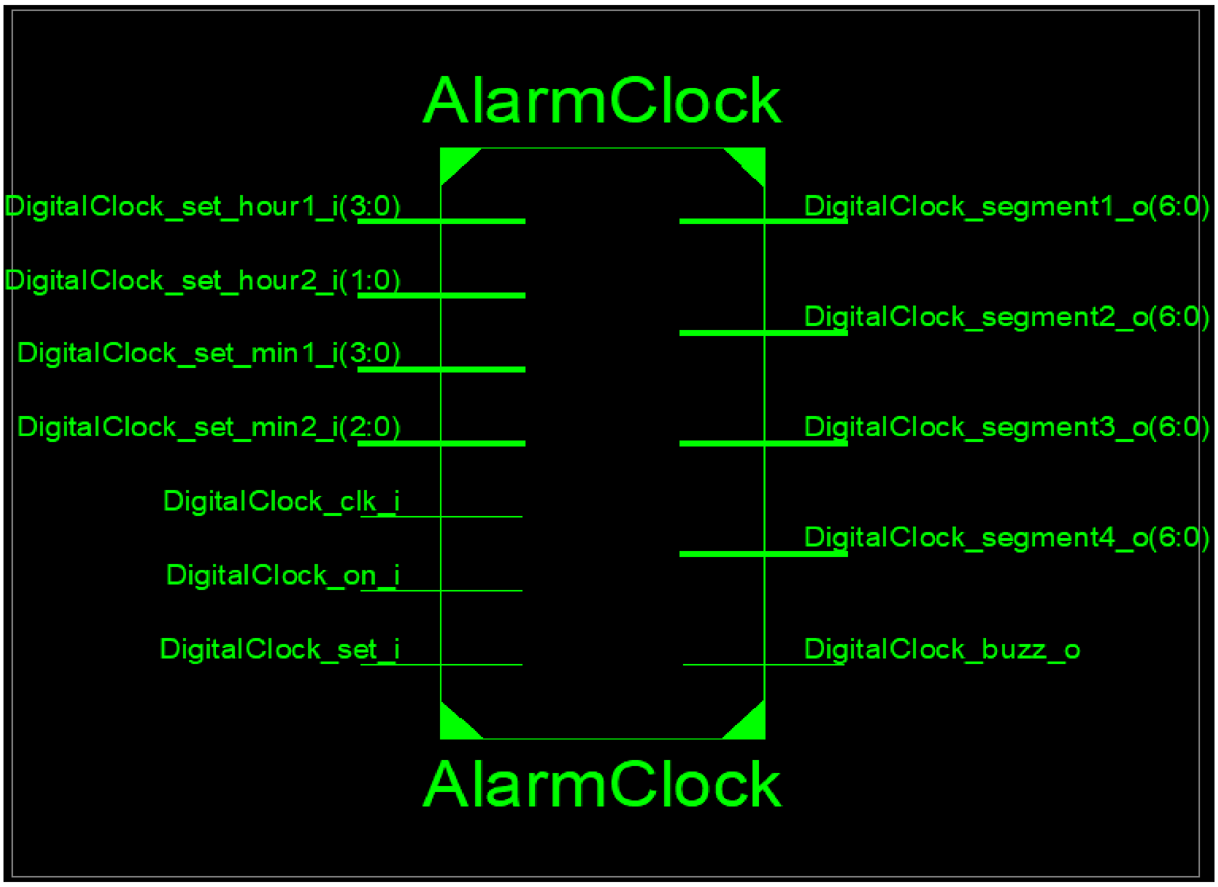
```
entity NumberDisplay is
   port( NumberDisplay_bcd_i:      in bit_vector(3 downto 0);
         NumberDisplay_segments_o: out bit_vector(6 downto 0)
   );
end NumberDisplay;

architecture Behavioral of NumberDisplay is

begin
  process (NumberDisplay_bcd_i)
    begin
     -- Every clock cycle
     case NumberDisplay_bcd_i is
       when "0000"=> NumberDisplay_segments_o <="0000001";
       when "0001"=> NumberDisplay_segments_o <="1001111";
       when "0010"=> NumberDisplay_segments_o <="0010010";
       when "0011"=> NumberDisplay_segments_o <="0000110";
       when "0100"=> NumberDisplay_segments_o <="1001100";
       when "0101"=> NumberDisplay_segments_o <="0100100";
       when "0110"=> NumberDisplay_segments_o <="0100000";
       when "0111"=> NumberDisplay_segments_o <="0001111";
       when "1000"=> NumberDisplay_segments_o <="0000000";
       when "1001"=> NumberDisplay_segments_o <="0000100";
       when others => NumberDisplay_segments_o <="0000000";
     end case;
  end process;
end Behavioral;
```

VHDL CODE: NUMBER DISPLAY

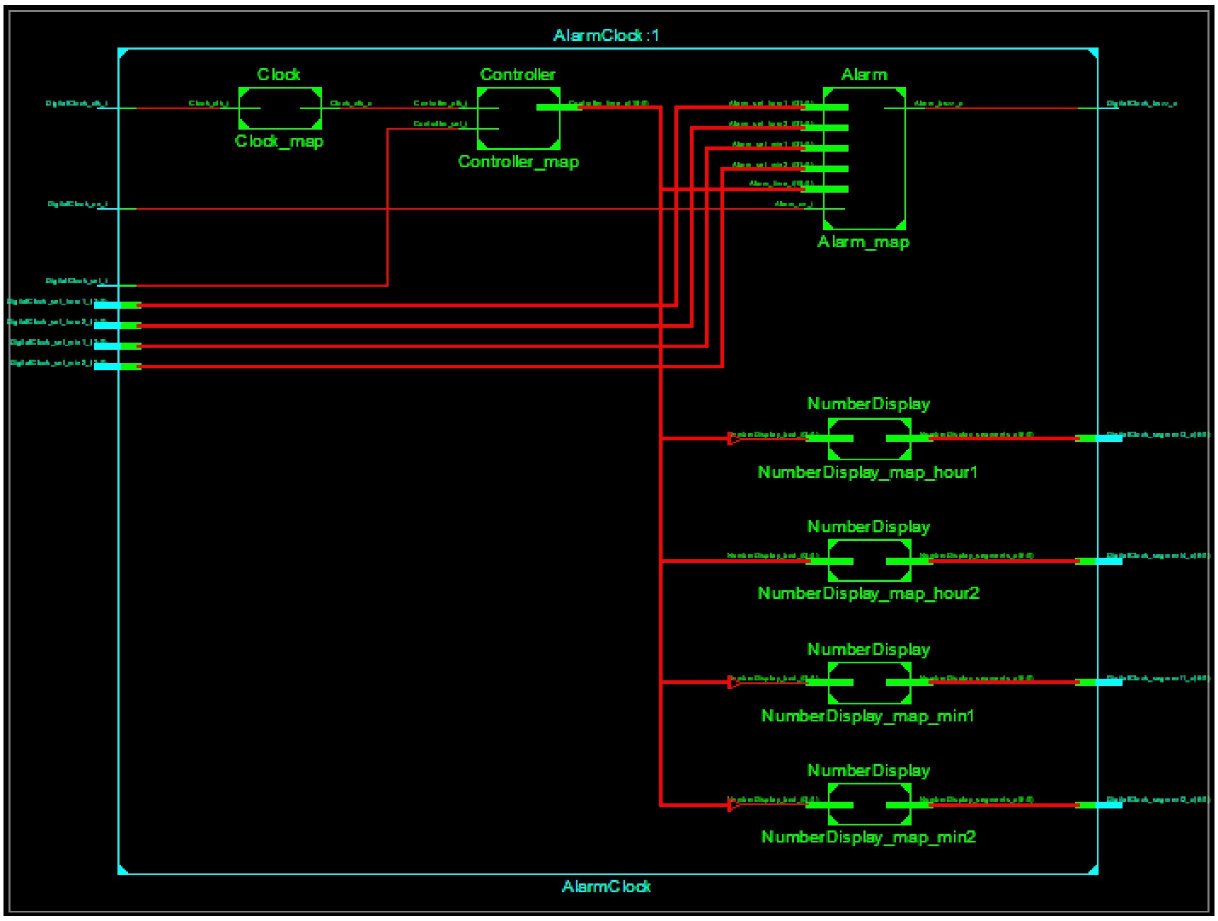


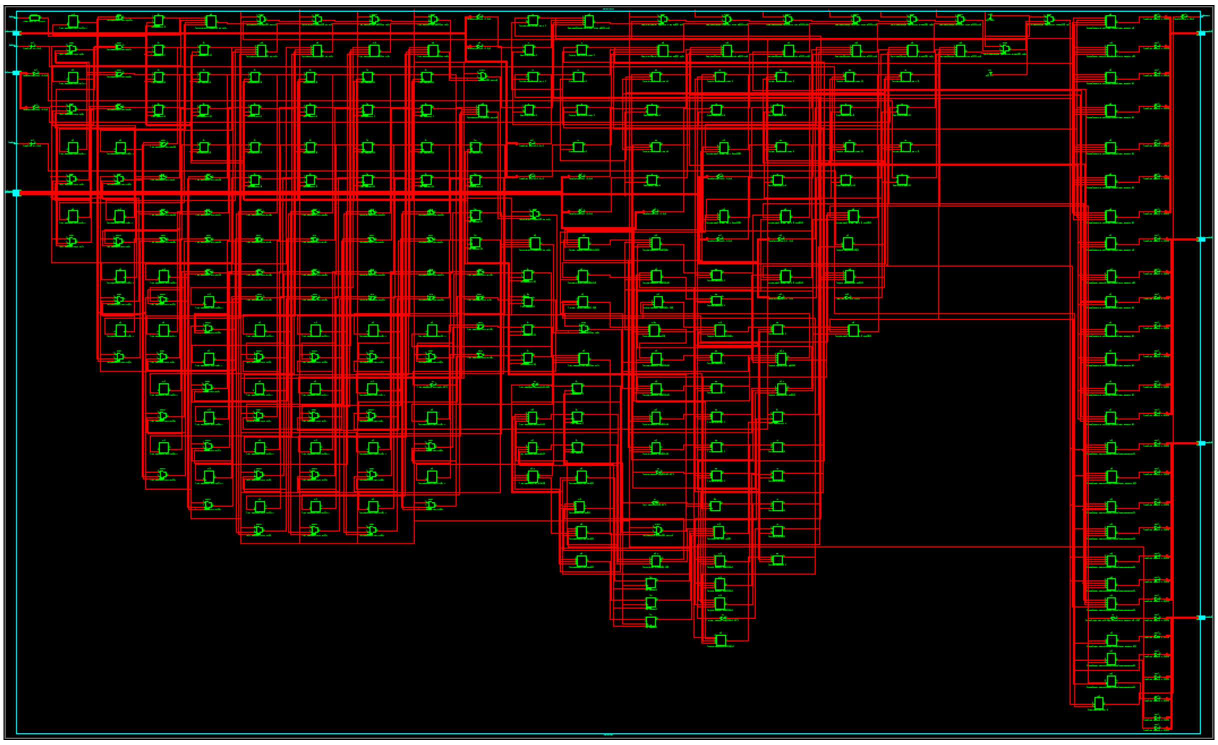
TESTBENCH : NUMBER DISPLAY

We were also able to, using Xilinx, synthesise this VHDL code for the purposes of FPGA implementation. While that implementation is not explicitly covered by our project due to unforeseen circumstances, the RTL and Technology schematics obtained from the synthesis are worth including and examining nonetheless. They are presented below.

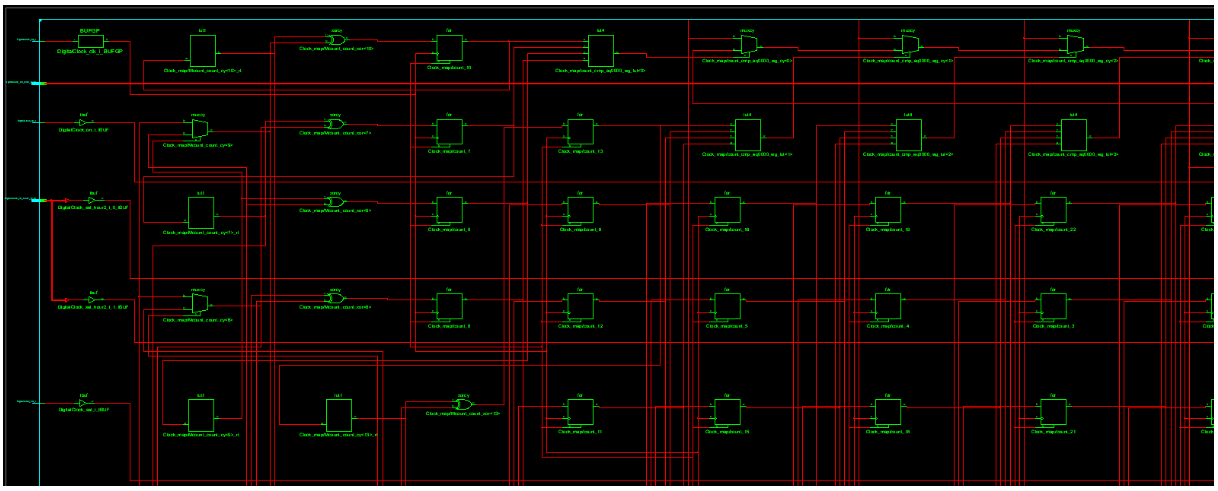

AlarmClock
DigitalClock_set_hour1_i(3:0)
DigitalClock_set_hour2_i(1:0)
DigitalClock_set_min1_i(3:0)
DigitalClock_set_min2_i(2:0)
DigitalClock_clk_i
DigitalClock_on_i
DigitalClock_set_i
DigitalClock_segment1_o(6:0)
DigitalClock_segment2_o(6:0)
DigitalClock_segment3_o(6:0)
DigitalClock_segment4_o(6:0)
DigitalClock_buzz_o
AlarmClock

RTL SCHEMATICS



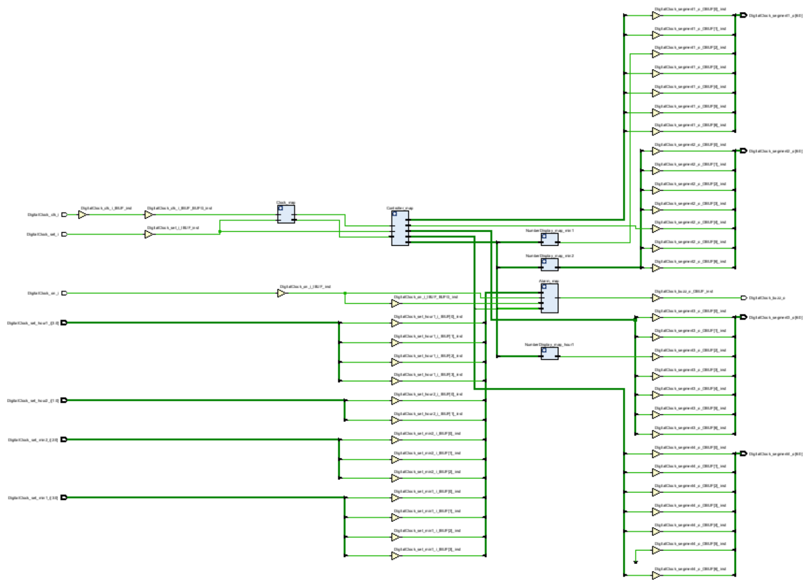


TECHNOLOGY SCHEMATICS, IN FULL AND A ZOOMED IN SEGMENT

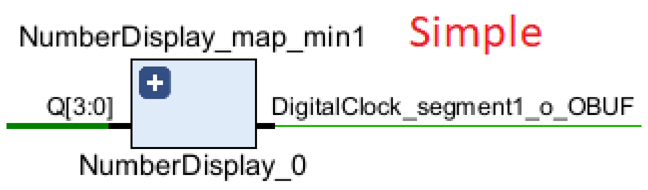


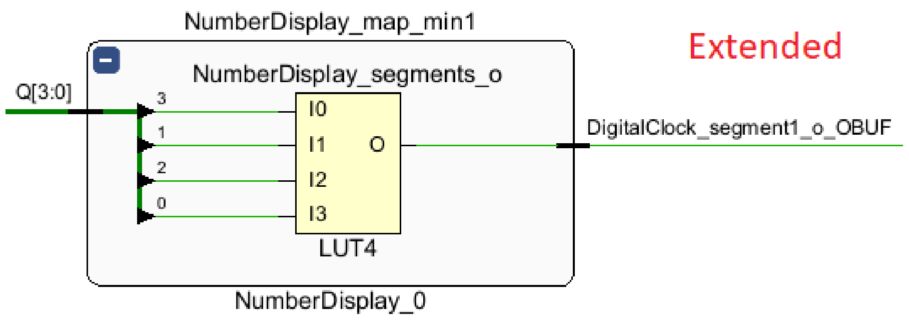
PCB Schematic and Design

After successfully testing the VHDL implementation via ModelSim, we copied the code to Vivado. Vivado is a powerful tool that is capable of auto-generating an optimal schematic for the gates and connections needed in order to develop a system such as ours. Vivado has its own VHDL coding environment; although the application itself has a much broader scope of use in different areas. But, for the purposes of our project, the VHDL environment was the perfect tool for a simple and quick solution. It should be mentioned, however, before anything else is delineated, that Vivado was only used for the purposes of generating the schematic and its necessary elements. It did not fit quite in the direction that the project was taking when the pure VHDL implementation and testbench simulation was developed, as it was more difficult to understand and we could not produce the necessary results that were, otherwise, made simple by Modelsim. Naturally, those results, in all likelihood, are technically achievable through the use of Vivado as well, but the obstacle that obstructed us from developing with it further was, indeed, our difficult experience trying to figure the tool out (especially due to compatibility and driver issues with Vivado that are outside the scope of the project itself). Nevertheless, once the project was imported and properly simulated, the final result came out surprisingly large and somewhat convoluted.

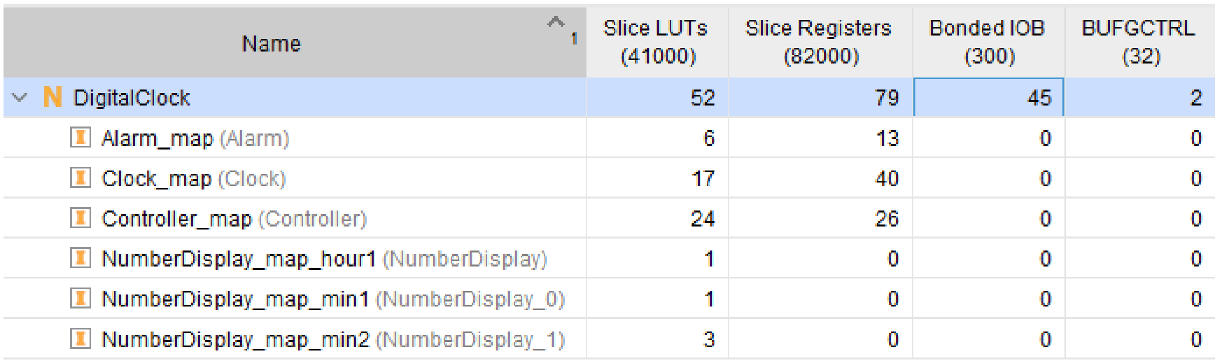


The schematic itself has quite a few elements and connections. Of course, it is not complicated to a fault and is still useful, in its own right. For instance, Vivado has, in a very organised fashion, brought together all of the elements of each and every separate part, and has had them grouped together in their own respective “types”. Every type can be extended and its details can be examined further.





On top of that, Vivado also generates a table of useful information that would be useful in determining the bill of elements. This table shows the required number of registers and LUTs.



Name	Slice LUTs (41000)	Slice Registers (82000)	Bonded IOB (300)	BUFGCTRL (32)
DigitalClock	52	79	45	2
Alarm_map (Alarm)	6	13	0	0
Clock_map (Clock)	17	40	0	0
Controller_map (Controller)	24	26	0	0
NumberDisplay_map_hour1 (NumberDisplay)	1	0	0	0
NumberDisplay_map_min1 (NumberDisplay_0)	1	0	0	0
NumberDisplay_map_min2 (NumberDisplay_1)	3	0	0	0

The LUT (Look-Up Table) is a tiny asynchronous SRAM used to create combinational logic, whereas the FF (Flip-Flop) is a single-bit memory cell used to store the current state. LUTs are typically read-only, with their content changing only during FPGA configuration. However, in Xilinx FPGAs, approximately half of the LUTs may be written to, allowing them to be used to build multiple tiny RAMs (so-called "distributed RAM"). And, in an actual physical implementation of this project, that information would be quite valuable.

Sources/References

Link to our git repository : https://github.com/Konevski/Hardware-Eng-Lab

References used in the project

1. https://github.com/saopayne/Digital-Alarm-Clock
2. https://vhdlguru.blogspot.com/2010/03/vhdl-code-for-bcd-to-7-segment-display.html