University of Strathclyde Engineering

VHDL Project Report

Daniel Conaghan | Liam Tang | Ross Cathcart

202113405 | 202135591 | 202139210

Electronic and Electrical Engineering

Introductory VHDL and FPGA Design

EE270

March 2023

Contents

1 Abstract

Thermostats are an essential piece of technology in the family home, monitoring the temperature and heating the house when the temperature drops below a critical value.

The temperature at which the thermostat activates the heating is a personal decision for the residents. Elderly people often prefer to raise the thermostat threshold temperature to keep them comfortable, whilst younger cost-aware families may lower the preset value to cut back on costs during the cost-of-living crisis.

As such, it is pertinent for a thermometer to offer such customisation, motivating the design task of this report - to design a thermometer with a user-defined preset temperature available.

The VHDL language is used to both simulate the design for the thermostat, but also to synthesise it onto physical hardware; the Baysys3 board manufactured by Digilent is utilised.

This report guides the reader through the entire design flow of:

1. Identifying the main entities and constructing a hierarchy
2. Writing synthesisable VHDL code to realise these entities, with their corresponding architectures
3. Testbenching and simulating the designs to ensure correctness of operation
4. Synthesising the design and deploying the bitstream onto the FPGA

2 Introduction

This project tasks groups to use VHDL to design a thermostat. This thermostat must support the following features:

- Allow the user to set a minimum "switch-on" temperature which will cause the heating system to turn on.
- Keep track of (count) the current temperature and respond to changes automatically where appropriate by switching on or off the heat.
- Display both the current temperature, and the pre-set "switch-on" temperature on a 7-segment display.

To tackle this problem, the functionality of the system was broken down into sub-systems, forming a *hierarchy* which facilitates easier testing and makes the program more modular. Such as design philosophy encourages code reuse in compliance with DRY coding principles. This report presents each system individually and then moves on to discuss their integration to form the overall design.

To support the design discussion, this report will also discuss how the sub-systems were tested through the use of testbenches and thoughtfully designed processes to ensure correct functionality.

Lastly, the report will detail the practical implementation of the design on the Basys3 board with an emphasis on how writing synthesisable VHDL necessitates careful coding standards to ensure correctness of operation. This section provides a discussion on the key differences between hardware and software programming, with the implications this has for the designer being addressed.

3 Background Theory

3.1 BCD and the Double Dabble Algorithm

To keep track of the current temperature in the program, a binary counter is used. The data type for this is a 6-bit unsigned `std_logic_vector` which can track temperatures ranging from zero to sixty-three degrees Celsius. The range of an n bit binary number can be calculated using Equation 1.

$$M(n) = 2^n - 1 \tag{1}$$

Where $\{n \in \mathbb{Z} \mid n \geq 1\}$ and the range of the number is given by [0,M].

Whilst the binary format is sufficient for counting numbers, driving a 7-segment display requires the conversion of the number to a format known as BCD (Binary-coded-decimal). BCD takes the binary number and divides it into 4-bit nibbles which represent the value of each digit in the denary equivalent value. For example, a binary number of 10001 (17 in denary) would be represented as 00010111 in BCD. The first nibble represents the value 1, and the second nibble corresponds to the value 7. Concatenating these digits, this gives the required value of 17.

To perform the conversion of binary to BCD, a famous computer science algorithm known as the "Double Dabble" algorithm [1] can be employed. This algorithm uses a scratch space (empty vector register) with enough room to store the BCD representation of the maximum denary value which can be assumed by the input value to convert. The steps for executing the algorithm are as follows:

1. Consider the empty scratch space being placed to the left of the binary representation of the number to convert.
2. Analyse each nibble of the scratch space. If it has a value greater than or equal to 5, add 3 to the nibble.
3. Shift the entire scratch space and adjacent binary number left by one bit. This pushes the MSB of the binary number into the LSB of the scratch space. Append a 0 onto the end of the scratch space.
4. Repeat steps 2-3 n times, where n is the width of the binary number to convert.

In VHDL, the process shown in Listing 1 implements the pseudocode version described above to convert the input binary number to a BCD output consisting of a tens and ones 4-bit `std_logic vector`. The algorithm operates in $O(n)$ time, where n is the width of the input number to convert, and thus number of iterations of the main loop.

```
convert : process(temp_in)
    variable intermediate : STD_LOGIC_VECTOR(5 downto 0);
    variable scratch_space : unsigned(7 downto 0) := (others=>'0');
begin
    scratch_space := (others => '0');
    intermediate := temp_in;
    --n iterations
    for i in 0 to 5 loop
    --add 3 to each nibble of scratch space if it is >=5
    if scratch_space(3 downto 0) >= 5 then
        scratch_space(3 downto 0) := scratch_space(3 downto 0) + 3;
    end if;

    if scratch_space(7 downto 4) >= 5 then
        scratch_space(7 downto 4) := scratch_space(7 downto 4) + 3;
    end if;

    --shift the intermediate left into scratch space by 1 bit
    scratch_space(7 downto 0) := scratch_space(6 downto 0) & intermediate(5);
    intermediate := intermediate(4 downto 0) & '0';

    end loop;
    --push the respective portion of the scratch space into each BCD digit
    temp_out_tens <= STD_LOGIC_VECTOR(scratch_space(7 downto 4));
    temp_out_ones <= STD_LOGIC_VECTOR(scratch_space(3 downto 0));

end process convert;
```

Listing 1: A process which implements the Double Dabble Algorithm

The reason for adding the seemingly arbitrary value of 3 to all nibbles greater than or equal to 5 is because this corrects the carry digit when the shift takes place. As such, the Double Dabble algorithm is sometimes called the "Shift add-three" algorithm.

3.2 7-segment displays

Following on from the prior section which discussed *how* to convert from binary to BCD, this section will now motivate *why* the BCD format is so useful for applications. The circuitry operation of the 7-segment display will also be discussed. In the context of this project, using a BCD value is mandatory for driving a 7-segment display. Figure 1 displays a four-digit 7-segment display with the circuitry system adjacent.

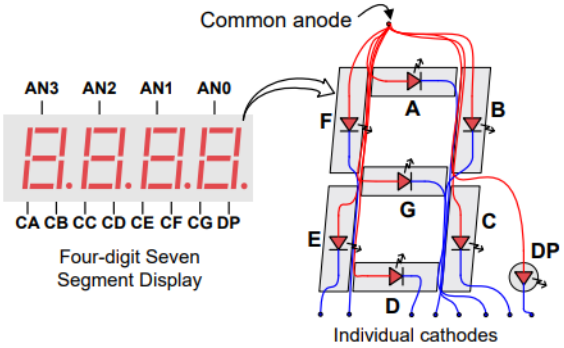


Figure 1: 7-segment display [2]

AN0-3 denotes the anodes of each digit and CA-DP denotes the cathodes of each digit. Each 7-segment display, representing a single digit, contains a common anode and discrete cathodes which are specific for each segment of the display. Since the anode is common to all segments, it can be considered an enable gate to the 4-digit representation. Distinct segments across all 4 digits are tied together. These are represented by CA - DP as shown in Figure 7, where CA represents the A segment for all 4 digits. The configuration of the circuitry connections multiplexes the display meaning that the activation of the anode governs which digit can light up. Each cathode is connected to an electronic control circuit which features current-limiting resistors. These serve to protect the LEDs and prevent them from burning out.

In order for an LED to illuminate, the common anode should have a high potential with the cathode having a low potential. Since the Basys3 Board uses transistors, the enable inputs are always inverted and hence are deemed **active low**.

The required logic levels at each cathode segment to light the corresponding bar of the 7-segment display shown in Figure 1 are summarised in Table 1.

s	BCD	G	F	E	D	C	B	A
0	0000	0	1	1	1	1	1	1
1	0001	0	0	0	0	1	1	0
2	0010	1	0	1	1	0	1	1
3	0011	1	0	0	1	1	1	1
4	0100	1	1	0	0	1	1	0
5	0101	1	1	0	1	1	0	1
6	0110	1	1	1	1	1	0	1
7	0111	0	0	0	0	1	1	1
8	1000	1	1	1	1	1	1	1
9	1001	1	1	0	1	1	1	1

Table 1: Truth table for the 7-segment display

3.3 Scanning Techniques for 7-segment Displays

Due to the multiplexed nature of the four-digit 7-segment display, a distinct segment can only be illuminated once across all 4 digits. This implies that the display will not be continuously illuminated for all 7-segment digits.

The inclusion of a scanning display system can be used to resolve this. The system repeats the desired anode and cathode inputs at a high refresh rate such that it appears continuous to the naked eye. This refresh rate should be greater than 45 Hz, implying each digit should have a period of 4 ms [2]. If this threshold period is not achieved, a flash will be noticed, which is detrimental to the performance of the project design.

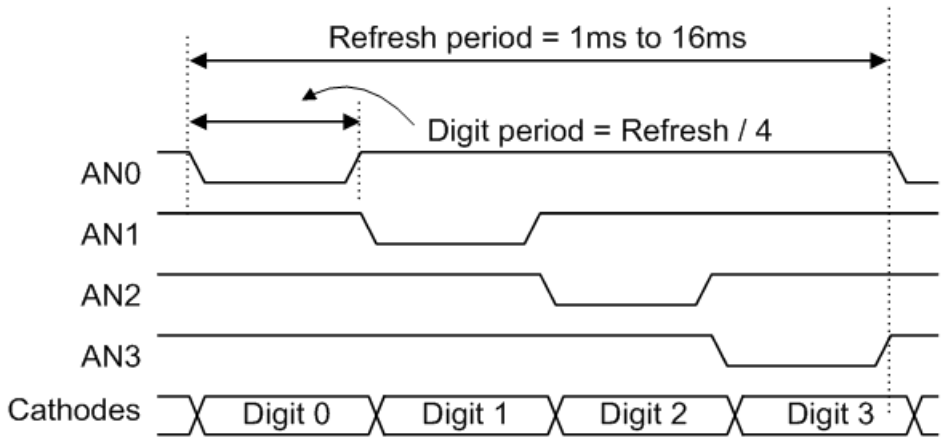


Figure 2: Scanning display timing diagram [2]

Figure 2 above shows the logic for anodes and cathodes as a function of time. It clearly indicates the refresh period and individual digit period required for smooth LED transmission. It is imperative that the anode and cathode pair is correct, and scaled to the correct period, otherwise display errors would occur.

3.4 Frequency division

Frequency Division (or clock division) is the process whereby an input frequency is taken and scaled down to a desired frequency value. Such a process was required to ensure that the clock signal, for both counting cases, was translated appropriately for hardware implementation.

An in-built cloc of frequency 100 Mhz is present in the Basys3 board [2]. This was used as the foundation clock signal and was remodeled to synthesise the desired 12-second count-up clock signal and the 20-second count-down clock signal. Using knowledge of the relationship between the frequency and period of a wave, the following formula can be used to calculate the divider constant required to generate the desired clock signals. This equation is shown in Equation 2.

$$f = 1/T \tag{2}$$

Where f denotes *frequency* measured in Hertz (Hz) and T denotes the *period* measured in seconds (s).

Since a conventional clock waveform has a duty cycle of 50%, the duration where the clock is logic high is given by half the period. To generate the low logic regions of the clock, the output of the clock is inverted for a duration that is equal to half of the wave period.

According to the aims of the project specification, the thermometer is required to toggle every 12 or 20 seconds, so 2 variables are required to keep track of the max count for each signal - one that tracks the count for 12 seconds and one for 20 seconds. Since the frequency of the wave is defined as the number of occurrences per second, the max value will be multiplied by either 12 or 20, accordingly, to correctly scale the delay in the counting sequence.

An alternative, and more industrially favoured method of frequency division, is to introduce a delay after updating the temperature value. This method omits the need for a count variable and changes the ratio of logic high and logic low duration. The introduction of this technique effectively achieves an 8.3% duty cycle for 12 seconds and 0.05% duty cycle for 20 seconds.

4 Program Design

4.1 Assumptions

In the design of the thermometer, a number of assumptions were made which would allow for the simplification of the design.

Firstly, it was assumed that the temperature would always be greater than zero degrees Celsius. The implication of this is that the preset temperature is lower bound at this minimum temperature. It is deemed a reasonable assumption that the heating will always activate before freezing point is reached, with the "most efficient temperature for your thermostat being between 18°C and 20°C" [3]. The reason for making this assumption is that only four 7-segment displays are available on the Basys3 and this project requires that two 2-digit BCD numbers are to be displayed, representing both the current and preset minimum temperature. This leaves no displays available to represent the sign bit needed to express negative numbers.

In addition to the restriction on only allowing positive numbers, the thermostat only supports a current temperature ranging from 0 - 63 $^{\circ}C$ inclusive, stored in a 6-bit unsigned logic vector. For most climates where a thermometer is likely to be placed, this would be sufficient to cover the entire range of temperatures experienced. By limiting the range of the logic vector to this size, memory and resources are saved resulting in a more optimised program that can be targeted towards less powerful hardware.

To accurately model a dynamic environment, the assumption was made that the temperature of the room would drop by 1 $^{\circ}C$ every 20 seconds when the heating is off, and increase by 1 $^{\circ}C$ every 12 seconds when the heating is on. Of course, in reality, external temperatures are unlikely to change this quickly, but for completeness in the testing procedure, making this assumption allows the designer to consider how the system will respond over a wider range of temperatures to ensure the correctness of operation for all conditions the end user may face.

The final assumption made was that the user preset should always be greater than or equal to one degrees celsius. This assumption prevents the wrap-around condition which is present when the current temperature fluctuates at one degree lower than the preset temperature as the heating is intermittently toggled on and off. If a preset of zero degrees was set, the current temperature

would toggle between 0 and 63 degrees at the equilibrium point which is nonsensical. Thus, this small assumption of a preset value which is greater than or equal to 1-degree celsius avoids this undesired behaviour.

4.2 Top Level Hierarchy

Prior to the coding process, a relatively high-level model of the system was developed. This involved reading the design specification to extract the sub-problems and then associating entities to tackle each of these challenges. The advantage of taking such an approach to design is that each of the units can be individually tested, making the testing process more straightforward and enabling more effective diagnosis of problems. Only when a suitable degree of confidence in the operation of each *individual* subsystem is achieved should they be integrated together to form the larger design. Figure 3 shows the interconnections between the sub-systems of the top-level design, and the input/output ports.

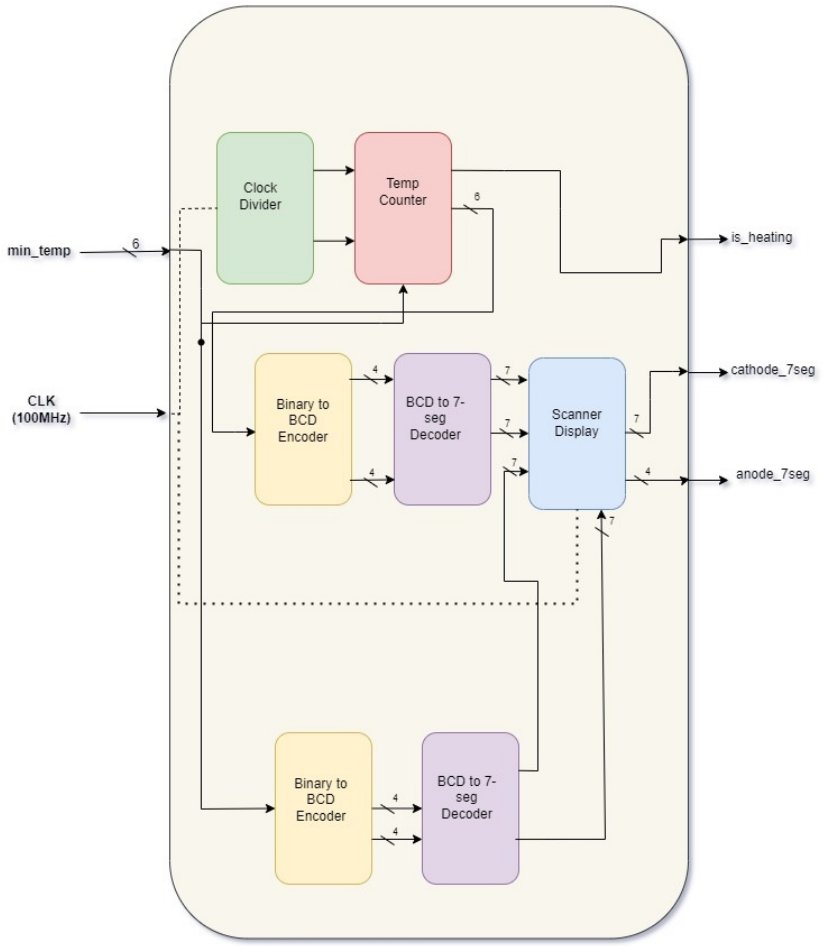


Figure 3: Top level design of thermostat

The main entities identified, and their responsibilities are summarised in Table 2.

Entity	Function
Clock Divider	Generates 2 output clocks of period 12s and 20s. These control when temperature changes occur.
Binary to BCD Encoder	Converts a binary value into a BCD representation, consisting of a tens and one's digit.
BCD to 7-segment Decoder	Converts the BCD value into 7-segment representation (illuminating required segments of the display).
Temperature Counter	Keeps track of the current ambient temperature and switches the heating on/off as required.
Scanner Display	Multiplexes each 7-segment value at a high refresh rate (1kHz) to make it appear as though the four 7-segment values are constantly illuminated.

Table 2: Entities and Functions

The report will now consider a more detailed analysis of each of these entities, breaking down the design approach taken to implement the required functionality which was identified.

4.3 Clock Divider

The design brief of the project states that the temperature of the room should decrease by 1 $^{\circ}C$ every 20 seconds when the heating is off, and increase by 1 $^{\circ}C$ every 12 seconds when the heating is on. In order to track the times at which the temperature changes should occur, it would be useful to have two clocks; one that produces a rising clock edge every 12 seconds and another that produces a rising clock edge every 20 seconds. This would inform the temperature counter that it needs to adjust its temperature at these pre-defined intervals.

In order to accomplish this in VHDL, a frequency division algorithm can be employed. Section 3.4 introduced how such a technique could be used. The Basys3 board has an in-built clock that has a frequency of 100 MHz [2] which can be divided into slower clocks corresponding with a much slower period. This enables the counting to take place at the 20-second and 12-second intervals specified in the problem statement.

4.4 Binary to BCD Encoder

The double dabble method to encode binary to BCD, as was introduced in the background theory, forms the basis of the architecture for the binary to BCD encoder.

This entity takes a 6-bit `std_logic_vector` and converts this into the corresponding binary digits for the tens and one's digits. This I/O interface is shown in the block diagram of Figure 4.

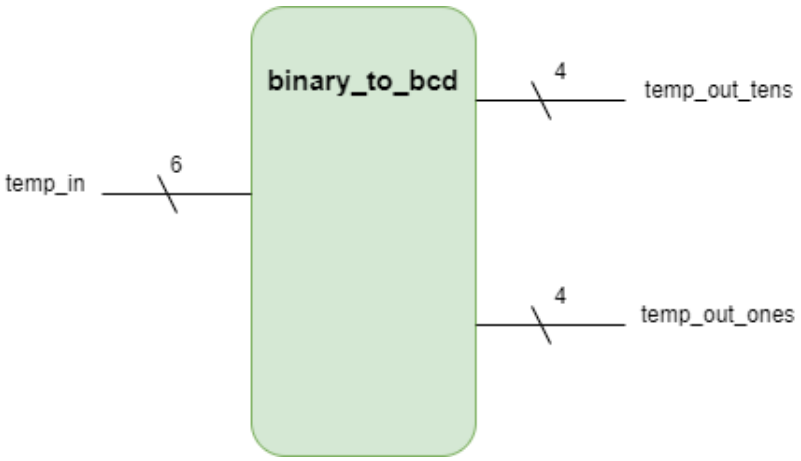


Figure 4: Entity Declaration for Binary to BCD encoder

The double dabble approach for converting between binary to BCD offers a reasonable time complexity for this operation and can be easily extended to accommodate a larger input size by increasing the size of the scratch space within the conversion process of the architecture (shown in Listing 1 earlier).

4.5 BCD to 7-segment Decoder

The signal output of the binary to BCD conversion then undergoes a process of decoding to convert the BCD value to a format supported by the 7-segment display. The high-level block diagram for this entity is shown in Figure 5.

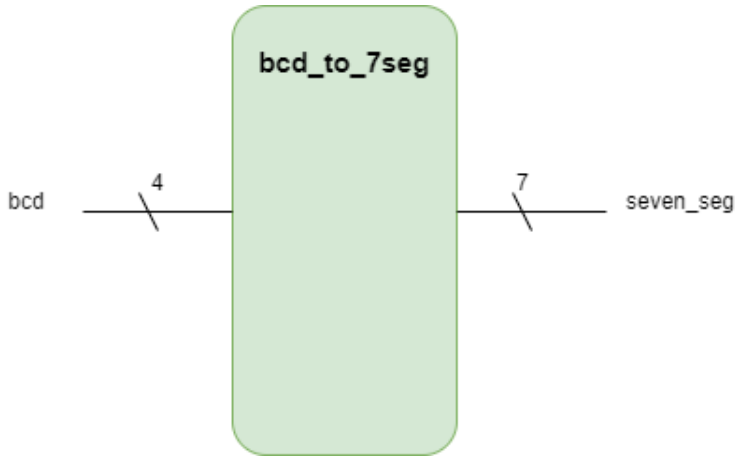


Figure 5: Entity Declaration for bcd to 7 segment

The process to convert the BCD value to a 7-segment value employs a case statement to act as a multiplexer and select the correct (matching) 7-segment sequence for the BCD value. Some sample cases of the larger statement are shown in Listing 2. The lack of priority implied in the case statement is ideal in this application, where a single value maps to a single result, which is analogous to a hash map data structure in software programming.

```
process(bcd)
begin
--case statement illuminates correct segments based on bcd value (active-low)
case bcd is
    when "0000" =>
        seven_seg <= "0000001"; --0
    when "0001" =>
        seven_seg <= "1001111"; --1
    when "0010" =>
        seven_seg <= "0010010"; --2
    when "0011" =>
        seven_seg <= "0000110"; --3
    ...
end case;

end process;
```

Listing 2: Multiplexer case statement for BCD to 7-segment decoding

4.6 Temperature Counter

The temperature counter component models the current temperature as part of the prototype. It is an integral part of the final design as it monitors the behaviour of the thermostat in response to the activation/deactivation of the heater.

This element of the design should:

1. Increase the temperature by 1 degree every 12 seconds if the heater is switched ON
2. Decrease the temperature by 1 degree every 20 seconds if the heater is switched OFF

The IO ports for the temperature counter are shown below in Figure 6.

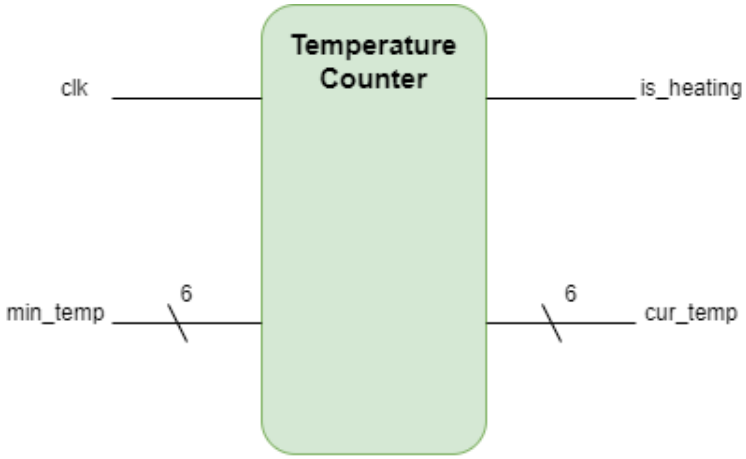


Figure 6: Entity Declaration for Temperature Counter

Following the key aforementioned design features, it was evident that a process involving conditional statements was required in the VHDL code. More subtly, the inclusion of a sensitivity list was deemed mandatory in tracking the decrement and increment triggers. Furthermore, it was decided that the use of a variable would be helpful in tracking the current output temperature.

The dynamism of this counter is controlled by the following process shown in Listing 3.

```
1  --process to either increment or decrement temp in response to time changes
2  delta_t : process(slow_clk_12, slow_clk_20)
3
4  --op_temp tracks current temperature and is modified to maintain equilibrium
5  variable op_temp : unsigned(5 downto 0) := (others=>'0');
6
7  begin
8     --heat up process if the current temp is below minimum
9     if (rising_edge(slow_clk_12)) then
10         if (op_temp <= unsigned(min_temp)) then
11             op_temp := op_temp + 1;
12             is_heating <= '1';
13         end if;
14    --cool down process if the current temp is above minimum
15    elsif (rising_edge(slow_clk_20)) then
16         if (op_temp > unsigned(min_temp)) then
17             op_temp := op_temp - 1;
18             is_heating <= '0';
19         end if;
20    end if;
21    --assign the local op_temp variable to the global output temperature
22    cur_temp <= std_logic_vector(op_temp);
23
24 end process delta_t;
```

Listing 3: Temperature Counting Process

The process, `delta_t`, manages the task of incrementing or decrementing the temperature in response to a change in either of the signals in the sensitivity list which is given as `slow_clk_12` and `slow_clk_20`, respectively. The members of the sensitivity list are clock input signals and are generated by a separate clock divider entity. A variable named `op_temp` of type `unsigned` is created with the intention of tracking the current temperature and is modified to maintain the equilibrium based on the user-defined minimum temperature.

If a `slow_clk_12` rising edge is detected and the current temperature is less than or equal to the pre-set user temperature, then the current temperature is incremented by 1 and `is_heating` is driven high. This condition commences the heating-up process while the current temperature is below the minimum temperature.

Similarly, when a rising edge is detected for `slow_clk_20`, the code checks if the current temper-

ature, stored as a variable, is above the minimum (user preset) temperature. If so, the current temperature decrements by 1, and the heating is toggled to low.

The penultimate line of code concurrently assigns the local variable value `opt_temp` to the output, `cur_temp`. This ensures that changes to the internal temperature state are reflected instantly (ignoring gate delays) to the outputs of the entity.

4.7 Scanner Display

The first step in designing the scanner display involved the identification of the interface ports to allow the entity declaration to be written. The scanner must have an input of the 4 logic vectors which correspond to each of the 7-segment digits to display. In addition, even though the entity must generate its own internal clock, the default 100 MHz clock should also be included as an input to allow the entity to divide this into a more appropriate (slower) frequency of 1 kHz within the architecture body. The output ports should include a 7-bit cathode logic vector which corresponds to the currently selected input 7-segment digit, and a 4-bit anode logic vector which corresponds to the current digit of the display which should be illuminated. These IO ports are illustrated in Figure 13, which provides a visual representation of the entity declaration.

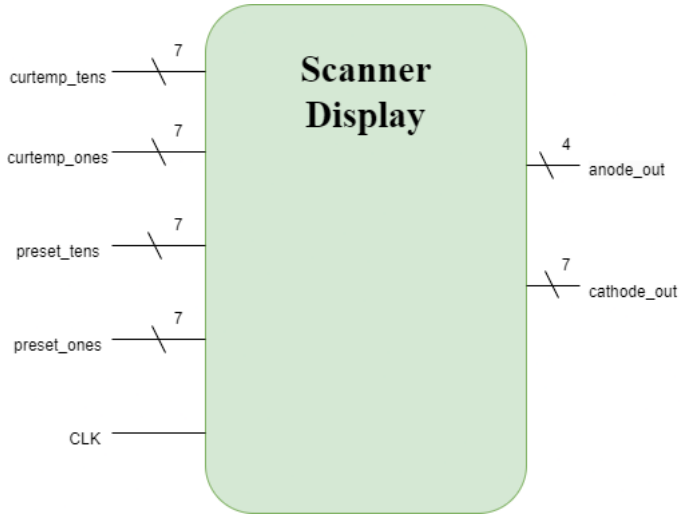


Figure 7: Entity Declaration for the Scanner Display

To implement the scanner display successfully, two key design criteria were identified:

1. The scanner must multiplex between each of the 7-segment input digits at a suitable refresh rate to avoid “flickering”.
2. The scanner must correctly output the anode sequence to activate a specific 7-segment display, and then repeat this for each of the 4 displays in a pre-defined order (tens digits, followed by ones).

From the requirements listed above it was clear that a number of VHDL processes would be required to accomplish these goals. Namely, a process would be needed to generate an internal clock with a relatively high frequency to control the switching between each of the 4 digits to

display. Another process would be required to sequentially increment a counter to drive the anode sequence corresponding with each 7-segment display. Lastly, a process that is sensitive to this anode sequence would be necessary to transfer the correct 7-segment input sequence to the `anode_out` output, thus achieving the desired multiplexer behaviour.

As was indicated in the theory discussion, a refresh rate of at least 60 Hz is suitable to trick the human eye into believing that an LED is constantly illuminated. For the sake of emphasising this effect, a refresh rate of 1 kHz was selected in this application. To divide the 100 MHz clock pre-built into the FPGA into a frequency of 1 kHz, the counting trick introduced previously can be employed. In this case, the quotient of the 100 MHz default clock and the desired frequency of 1 kHz is 100,000. Therefore an input rising clock edge count of 50,000 is required to control when the internal clock should be toggled to produce a 50% duty cycle waveform. The VHDL code for this process is shown in Listing 4.

```
1  clk_gen : process(clk)
2  --count to 50,000 then invert clk to divide f to 1kHz
3  variable count : INTEGER := 0;
4  begin
5      if (rising_edge(clk)) then
6              count := count + 1;
7      end if;
8
9      if (count = maxval-1) then
10         clk_1kHz <= not clk_1kHz; --invert clock
11         count := 0; --reset count
12     end if;
13 end process clk_gen;
```

Listing 4: Frequency division to create a 1 kHz internal clock

To display the correct anode sequence required to select each 7-segment input in a cyclical fashion, a rudimentary counter was used. While the value of the counter was less than 3, it was incremented. Otherwise, the count was reset to 0. This forms a circular count sequence. The enumerated value provides a value to "switch" on as the subject in a case statement. Each case is mapped to a different anode sequence, which are all of a 4-bit `std_logic_vector` type.

The nibbles selected by the case statement have a single `0` in a different location, with all other values assuming a value of `1`. These correspond to the active-low enable inputs of the anode on the seven-segment display architecture found on the Basys3 board. For example, the anode sequence of `0111` would select the input `curtemp_tens` to pass to the `cathode_out`. This process is shown in Listing 5 which illustrates the counting technique and case statement discussed above.

```
anode_scanner : process(clk_1kHz)
    variable multiplex_count : integer := 0;
begin
    if (rising_edge(clk_1kHz)) then
        multiplex_count := multiplex_count + 1;
        if (multiplex_count = 4) then
            multiplex_count := 0;
        end if;
    end if;

    case multiplex_count is
        when 0 =>
            anode <= "0111";
        when 1 =>
            anode <= "1011";
        when 2 =>
            anode <= "1101";
        when 3 =>
            anode <= "1110";
        when others =>
            anode <= "1111";
    end case;
end process anode_scanner;
```

Listing 5: Anode scanner process

Finally, to use the anode sequence which is generated in the process above, another process that is sensitive to changes in the anode can be created. Once again, a case statement is employed to use this anode sequence to pass the appropriate input value to the `cathode_out` of the entity. This behaviour is analogous to a multiplexer and is likely how it is synthesised into hardware. Listing 6 demonstrates how this logic can be translated into VHDL code.

```
cathode_scanner : process (anode)
begin
    --multiplexer selects cathode based on anode
    case anode is
        when "0111" =>
            cathode <= curtemp_tens;
        when "1011" =>
            cathode <= curtemp_ones;
        when "1101" =>
            cathode <= preset_tens;
        when "1110" =>
            cathode <= preset_ones;
        when others => --should never occur
            cathode <= cathode;
    end case;
end process cathode_scanner;
```

Listing 6: Cathode scanner multiplexer process

4.8 Top level

The top-level design source is a file that declares an entity that encapsulates all of the sub-entities which have been discussed previously in this section. The file defines the interconnecting signals needed to connect these entities together and is analogous to physically wiring up the circuit on a breadboard. The entity declaration shown in listing 7 defines the top-level interface ports which will be mapped to the IO blocks of the FPGA in the XDC constraints file.

```
entity top_level is
    Port ( clk : in STD_LOGIC;
           heat_on : out STD_LOGIC; -- boolean to light LED
           temp_preset : in STD_LOGIC_VECTOR (5 downto 0);
           cathode_out : out STD_LOGIC_VECTOR (6 downto 0);
           --the current 7-seg digit being displayed
           anode_out : out STD_LOGIC_VECTOR (3 downto 0));
end top_level;
```

Listing 7: Entity declaration of top level

Following this top-level entity declaration, the region before `begin` in the architecture contains the declarations for all the sub-components of the design. Indirect component instantiation was favoured for the sake of readability in this code, but direct component instantiation is an equally valid approach that could reduce the code verbosity.

These components match the declarations given in the entity declarations for the individual entities,

which specify the input and output ports. A sample component declaration for the temperature counter is shown in Listing 8 below. It should be noted that a configuration specifier statement to create each of the design entities was not required due to the naming of the components (and interfaces) exactly matching the corresponding entities in the working directory.

```
1 component temp_counter is
2 Port ( slow_clk_20 : in std_logic;
3            slow_clk_12 : in std_logic;
4            min_temp : in std_logic_vector (5 downto 0);
5            cur_temp : out std_logic_vector (5 downto 0);
6            is_heating : out std_logic);
7 end component temp_counter;
```

Listing 8: Component declaration of the temperature counter sub-process

Succeeding the component declarations, the internal signals are then created to link the components and allow information to flow within the top level of the hierarchy.

```
1 signal clk_20, clk_12 : STD_LOGIC;
2 signal curtemp_binary : STD_LOGIC_VECTOR (5 downto 0);
3 signal bcd_preset_ones, bcd_preset_tens,
4 bcd_cur_ones, bcd_cur_tens : STD_LOGIC_VECTOR(3 downto 0);
5 signal scanner_ct_ones, scanner_ct_tens, scanner_pst_ones,
6 scanner_pst_tens : STD_LOGIC_VECTOR (6 downto 0);
```

Listing 9: Internal signal creation

Presented above in Listing 9, line 1 displays the creation of 2 signals - `clk_12` will be set to 1 when the thermostat is counting up every 12 seconds and 0 when the thermostat is counting down. Inversely, `clk_20` will be set to 1 when the thermostat is counting down and will be set to 0 when the thermostat is counting up. As explained previously in (4.3), these boolean values can be used to send the correct internal signal to the status LED to indicate what state the thermostat is in.

To *instantiate* the BCD to 7-segment components within the architecture, a series of generate statements were used to avoid code duplication and allow these repeating hardware structures to be succinctly created. The code shown Listing 10 demonstrates how these `generate` statements can be written to create the four instances of the 7-segment component.

```
SEG_GEN: for i in 0 to 3 generate

    SEG_GEN0 : if i = 0 generate --bcd -> 7 seg for 'ones' curtemp
        CUR_ONES : bcd_to_7seg
        Port Map (bcd=>bcd_cur_ones, seven_seg=>scanner_ct_ones);
    end generate;

    SEG_GEN1 : if i = 1 generate --bcd -> 7 seg for 'tens' of curtemp
        CUR_TENS : bcd_to_7seg
        Port Map (bcd=>bcd_cur_tens, seven_seg=>scanner_ct_tens);
    end generate;

    SEG_GEN2 : if i = 2 generate --bcd -> 7 seg for 'ones' of preset
        PRE_ONES : bcd_to_7seg
        Port Map (bcd=>bcd_preset_ones, seven_seg=>scanner_pst_ones);
    end generate;

    SEG_GEN3 : if i = 3 generate --bcd -> 7 seg for 'tens' of preset
        PRE_TENS : bcd_to_7seg
        Port Map (bcd=>bcd_preset_tens, seven_seg=>scanner_pst_tens);
    end generate;
end generate;

```

Listing 10: Generate statements for component instantiation

5 Simulation and Testing

5.1 Testing the Clock Divider

The functionality of the clock divider is very simple; it takes a high-frequency input clock and then divides this into two slower clocks. As such, the test specification for this entity and the corresponding testbench is short. The areas tested include:

1. The input clock produces two clock outputs.
2. The duty cycle of the produced clocks is 50%.
3. The slower of the two clocks produced has a frequency which two-thirds (66.7%) faster than the other slow clock.

It may have been noted that the criteria above avoids explicitly mentioning the exact period of each of the clocks to be produced. This is because, for the purposes of simulation, the generated clocks are of a much higher frequency than those used for synthesis to allow for shorter and more

rapid testing. In synthesis, the period of the clocks generated is 12 seconds and 20 seconds, respectively. Running simulations for this long is impractical (typically simulations are around 10 μs in duration). The frequency of the output clocks is controlled by the constant `default_clk` in the `slow_clocks.vhd` allowing the designer to quickly modify this.

The results of simulating the clock divider are shown in Figure 8.

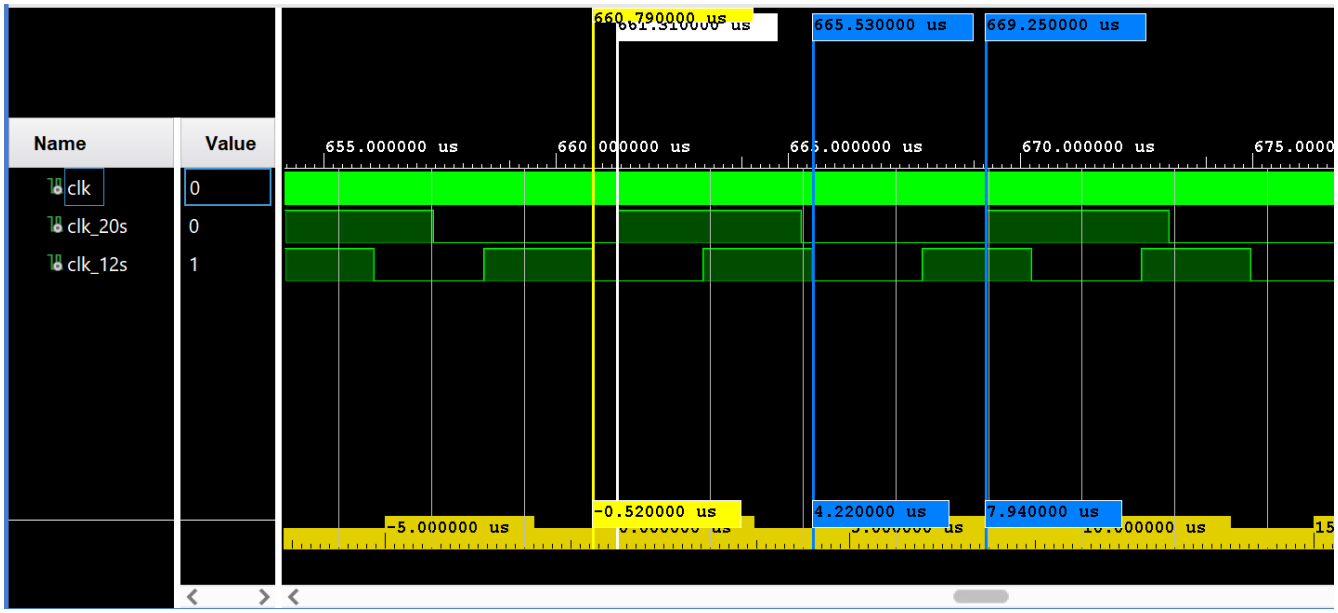


Figure 8: Clock Divider test waveform

Clearly, two slower frequency clocks are being generated at the outputs. By inspection, the duty cycle of the generated clocks can be determined as 50% but this was further confirmed by measuring the Δt values between the logic-low and logic-high portions of a cycle on the graph using cursors. Likewise, by measuring the period of the two clocks using the cursors and calculating the percentage increase in speed, it was verified that the faster of the two generated clocks is indeed 66.7% faster than the other slow clock.

All the test cases identified for this entity were passed so it can be confidently concluded that it should behave as expected when integrated into a larger design.

5.2 Testing the Binary to BCD Encoder

To test that the double dabble algorithm has been implemented correctly, and was successfully converting the input binary number to BCD, a range of test binary values were prepared in the testbench file. These were selected to span across a wide range of the possible values the 6-bit `std_logic_vector` input temperature could assume to ensure a degree of thoroughness. The waveform viewer was then used to verify that the binary number was being converted into the correct `temp_out_tens` and `temp_out_ones`. The results are shown in Figure 9.

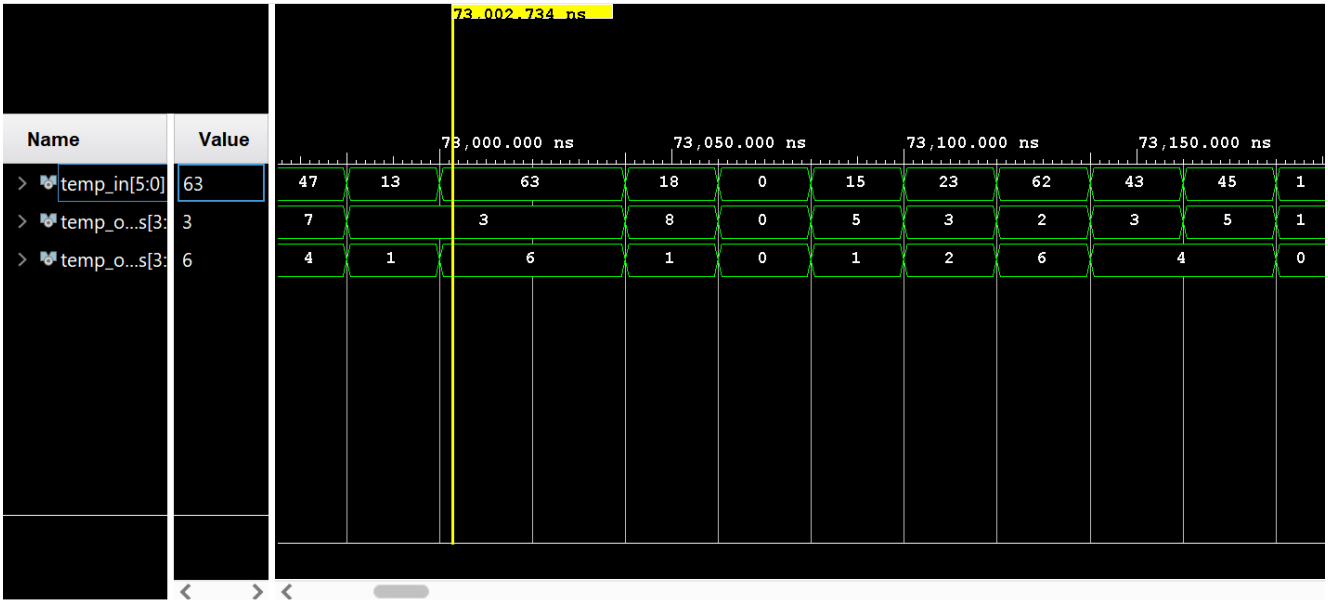


Figure 9: Binary to BCD test waveform

From the test results above, the entity architecture and double dabble algorithm appear to work as intended. All of the test inputs (binary) have been successfully decomposed into their respective tens and one's digits (4-bit `std_logic_vectors`). Note that these appear as integers on the waveform, but this is due to the Radix mode of the simulation tools which offers type conversions post-simulation to facilitate easier comprehension of the results on the waveform viewer - the underlying data types are unchanged.

5.3 Testing the BCD to 7-Segment Decoder

The architecture of the BCD to 7-segment decoder is a simple case statement that uses the 4-bit input BCD digit as a switchable value to decide which of the segments needs to be illuminated to display the correct 7-segment pattern. As such, the testbench for this is very simple and can be exhaustive (covering all of the possible BCD values). This assumes a valid BCD digit is passed to this entity which is a reasonable assumption to make because the source of these BCD digits was from the binary to BCD encoder which was thoroughly tested in the previous section. The output waveform with the simulation results is shown in 10.

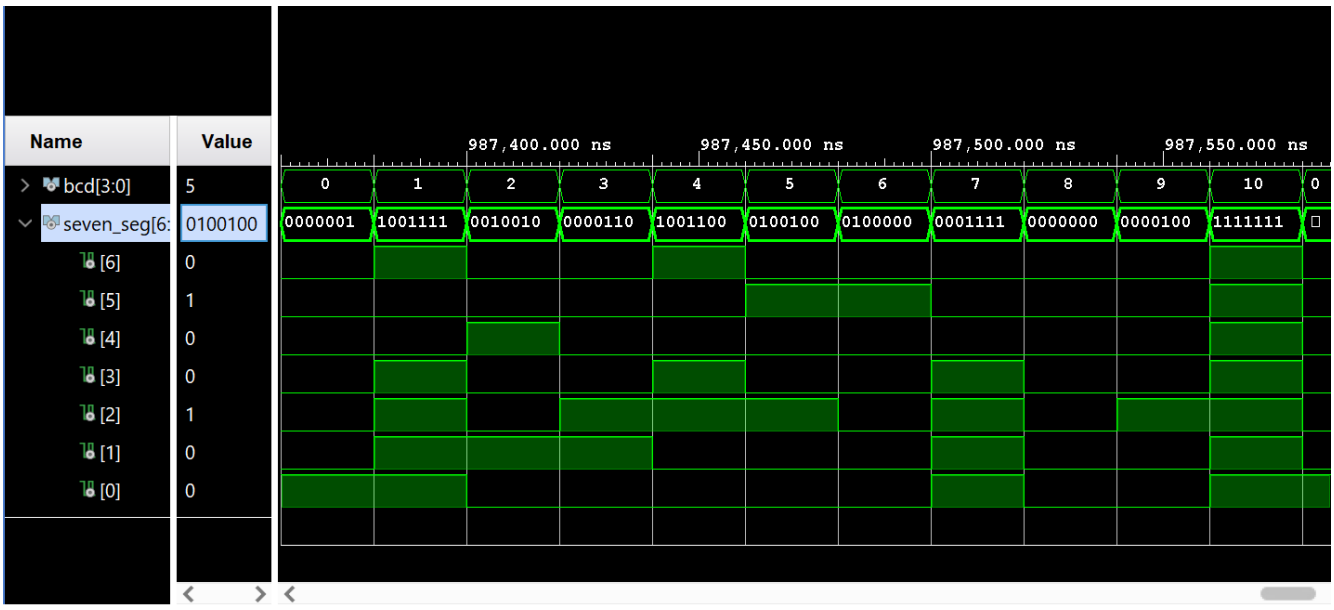


Figure 10: BCD to 7-segment decoder test waveform

Comparing the BCD number and corresponding segments illuminated on the waveform with those found in Table 1, the results appear to be correct. Note that segment G is the final value in each of the `std_logic_vector` objects in the simulation, but it is the first column in Table 1. When this is taken into account, the segments illuminated in the simulation perfectly match those predicted in the theory for all valid input 7-segment input combinations. This provides a degree of confidence that this component of the larger design entity will behave correctly when integrated together with the other components.

5.4 Testing the Scanner Display

To test the behaviour of the scanner display for correctness of operation the following criteria was assessed:

1. The clock division internal to the entity divides the incoming clock of 100 MHz into a slower value.
2. The internal clock correctly triggers changes to the `anode` on rising clock edges.
3. Each anode sequence should be displayed, and for the same duration.
4. The anode sequence should pass the correct input to the `cathode_out`. For example, `cathode_out` should assume the value of `curtemp_tens` when the anode sequence is `0111`.
5. The anode sequence should never become `1111` (the exception case to cover undefined behaviour) in its case statement.

The testbench was constructed to test the above criteria by creating 4 mock 7-segment input values, and a `clock_gen` process to drive the input clock at a frequency of 100 MHz. It is important to note that the refresh rate used in the simulation was 12.5 MHz, which is much higher than the

refresh rate used for synthesis. This was done to reduce the duration of the simulation and prevent needlessly long wait times. The operation of the entity still remains the same, with the only difference in operation being the internal clock time so this is still a valid simulation. The results of the simulation are shown on the waveform in Figure 11.

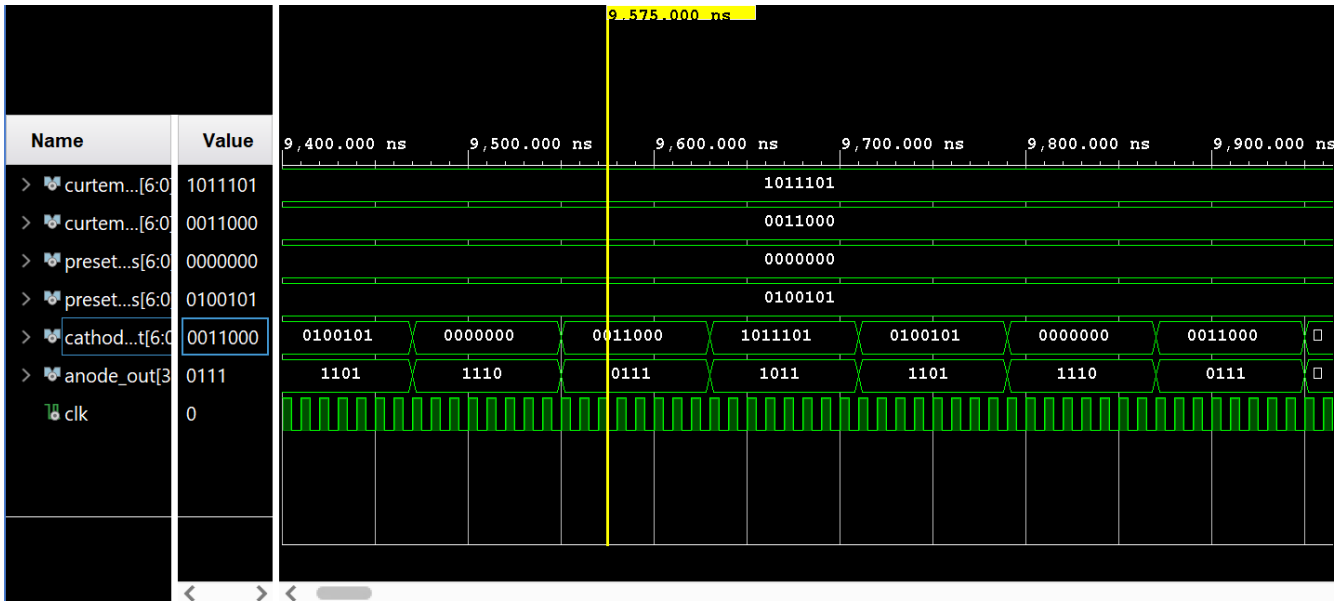


Figure 11: Scanner Testbench Waveform

From the waveform above, conditions 1 and 2 of the test specification are successfully met - the anode sequence changes every 4 clock edges. This indicates that the internal clock which controls the anode sequence has been successfully divided. Furthermore, condition 3 is met as each anode sequence is displayed for the same duration. The practical implication of this is that each of the 4 digits on the 7-segment display will be illuminated for the same duration which is advantageous for stable performance. Condition 4 can be verified by analysing the `cathode_out` value for each `anode_out` to ensure the multiplexer is selecting the correct input value to pass to the output. Table 3 summarises the results of the anode multiplexing demonstrated on the waveform.

Anode Sequence	**Selected input**
0111	curtemp tens
1011	curtemp ones
1101	preset tens
1110	preset ones

Table 3: Anode Multiplexing

Clearly, the correct input value is being passed to the cathode output for a given anode sequence, thus the 7-segments in a linear order.

The final test area states that the anode sequence of `1111` should never appear under normal operation. From what is displayed on the waveform (and outwith the portion shown) it was verified that this sequence is never displayed, indicating that undefined behaviour does not occur.

Overall, the testing procedure for the Scanner display was largely successful and provides a high degree of confidence that the synthesised design should operate as intended.

5.5 Testing the Temperature Counter

To ensure that the temperature counter entity was functioning correctly, the following critical functionality was identified as subject areas for the testing:

1. The temperature counts up (or down) to the preset value, and stabilises at this value.
2. When the temperature is increasing, the `is_heating` boolean value should be logic-high (true). Likewise, the `is_heating` value should be logic-low when the temperature is decreasing.
3. Increases in temperature should be synchronised with rising clock edges of `slow_clk_12`, whereas decreases should be synchronised with rising edges of `slow_clk_20`.

To testbench the design and exercise these areas, a mock pre-set temperature was set and two input clocks with the correct duty cycle and frequency ratio with respect to one another were created. The testing should analyse the state of the `is_heating` and `cur_temp` to ensure that the criteria given are satisfied.

The results of carrying out this testing are shown on the waveform in Figure 12.

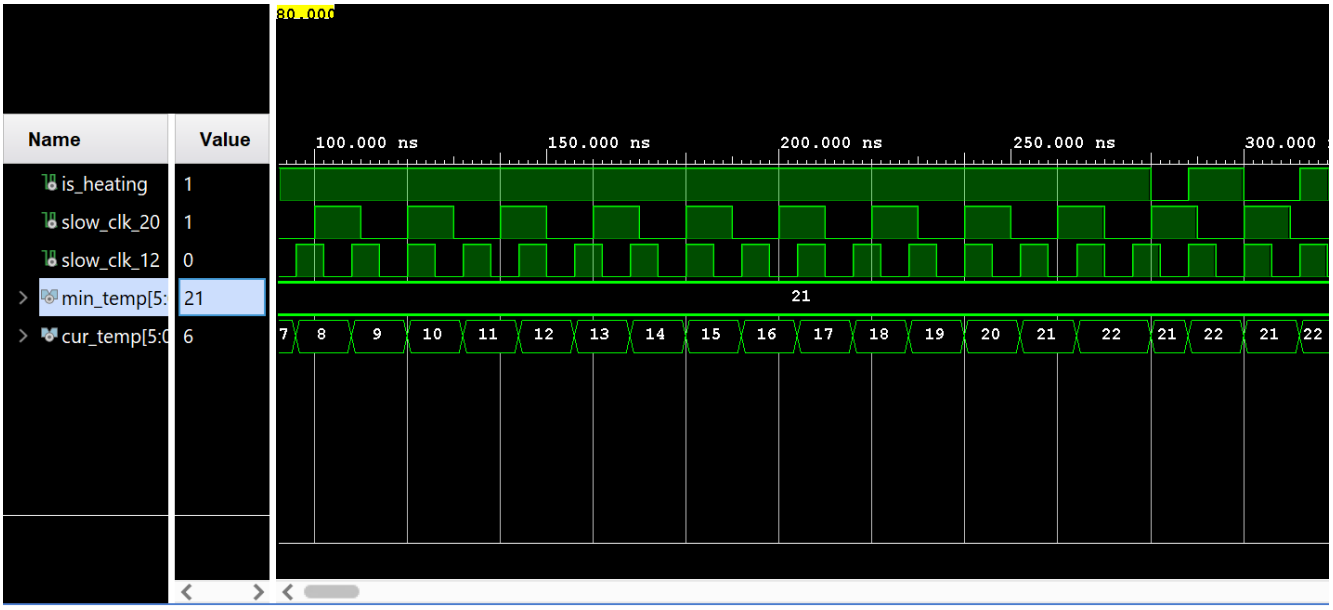


Figure 12: Temp counter test results

Item 1 of the test criteria is clearly demonstrated on the waveform, as the `cur_temp` climbs from its initial value of 0 up to the preset temperature of 21, stabilising at $+1\ ^{\circ}C$ of this value as the heating is toggled on and off to maintain the ambient equilibrium temperature.

Furthermore, the `is_heating` boolean is correctly toggling on and off to symbolise the activation of the heating. For the initial phase of the simulation, the heating is on as the temperature approaches

the preset value. After this point, the heater toggles on and off to maintain this temperature which is reflected by the status of this boolean.

The final areas of testing specify that changes in the current temperature should be controlled by the correct clocks, and occur on rising edges of these only. Whilst the temperature is below the preset value, the increases in current temperature are synchronised with the rising edges of `slow_clk_12`, as required. Above the preset temperature, the drops in temperature are synchronised with the rising edges of `slow_clk_20`, once again satisfying the requirements and design rubric.

As all the test areas were met successfully, it would be reasonable to assume that the temperature counter should behave correctly when synthesised. However, as will be discussed during the "Implementation on the Basys3" section to follow, this was not the case.

5.6 Testing the Top Level Design

To ensure that all the sub-components of the temperature counter functioned correctly when integrated into the larger design, a testbench was written for the overall system. This testbench provided a mock preset temperature and a clock of 100 MHz to the entity as inputs to accurately reflect the stimulus which would be experienced by the design when used in a practical setting. The anode out, cathode out, and LED were monitored as outputs to the system on the waveform generator. This generated the results shown in the waveform of Figure 13.

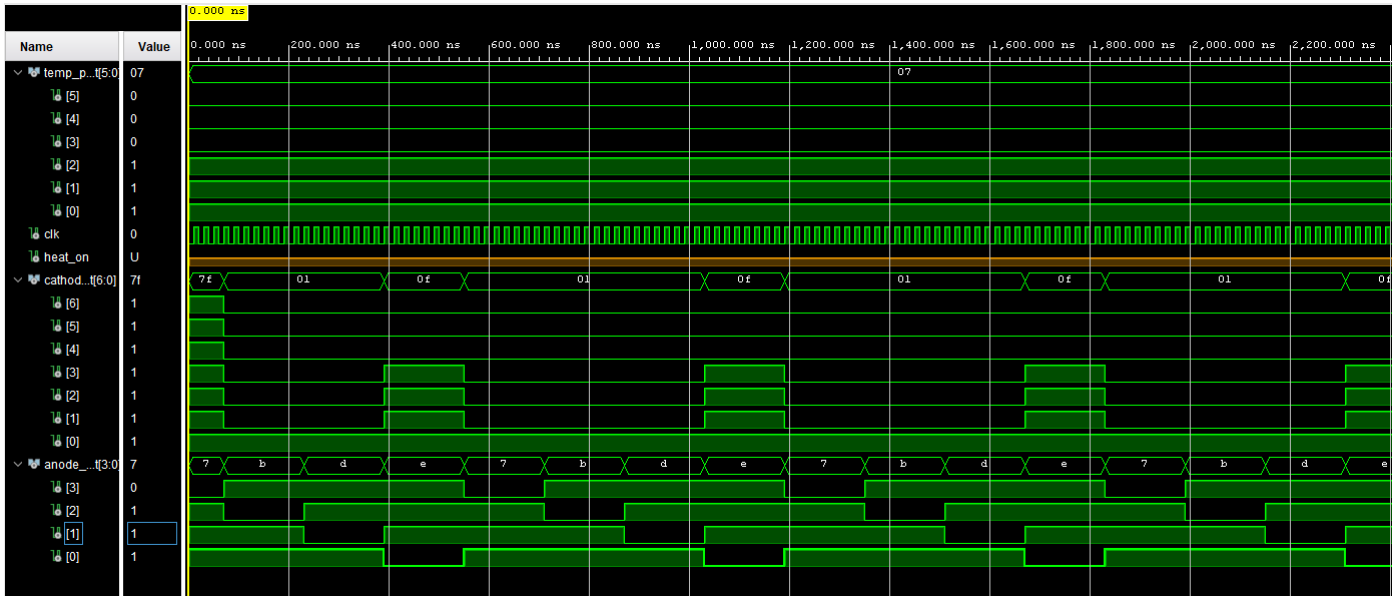


Figure 13: Entity Declaration for the Scanner Display

From the simulation waveform above, the clock signal `(clk)` contains a frequency of 100 MHz taken from the periodical count every 10 nanoseconds. Furthermore, the preset temperature is not changing `(temp_preset)`, which demonstrates the static behaviour which is expected from this value. The preset should only change when the user changes this value manually using the preset switches provided on the Basys3 board.

From `cathode_out`, sub-signals 6,5 and 4 stay high until approximately 70 `(ns)` nanoseconds and then remain low. The same goes for sub-signals 3,2 and 1 except after 70 `ns` they remain low for

a period of 325 `ns` and then up after the 70 `ns` pulse. The sub-signals all pulse at 1 for 70 `ns`, every 480 `ns`. Signal 0 remains high for the duration of the simulation. This combined behaviour described above demonstrates that the cathode is indeed correctly cycling to assume a 7-segment array value corresponding to each of its four inputs.

The anode signal `anode_out` represents a count every 160 `ns`, shown in sub-signals 3,2,1 and 0. It starts with sub-signal 3 at value 0 then 2,1,0 and back to 3. Once again, this cyclical behaviour demonstrates the expected clocked multiplexing behaviour of the scanner display, implying that it is functioning correctly as a sub-component of the larger design entity.

The observations for the `cathode_out` and `anode_out` combine to suggest that the correct multiplexing behaviour has occurred, shown from the signal outputs changing as a function of the sub-signals change.

6 Synthesis

By taking active steps to ensure that the VHDL code written was synthesisable, avoiding common pitfalls such as wait statements and infinite loops, the design successfully passed the synthesis, implementation, and bitstream generation stages of the FPGA workflow.

Phase one of the implementation procedure involved *synthesising* the design into a hardware netlist which details the resources the board must use to execute the design. An advantage of FPGA development when compared with other similar digital devices such as ASICs is that they do not require manual component placement or routing. This is handled automatically, and optimised by the synthesis tools provided by Xilinx. This raises the abstraction level and allows the designer to focus on the core functionality of the design rather than on aesthetics and tedious placement. The generated schematic created by the design tools is shown in Figure 14. This is very similar to the block diagram introduced in Figure 3 previously, but shows a more detailed breakdown of the signal interconnects between components.

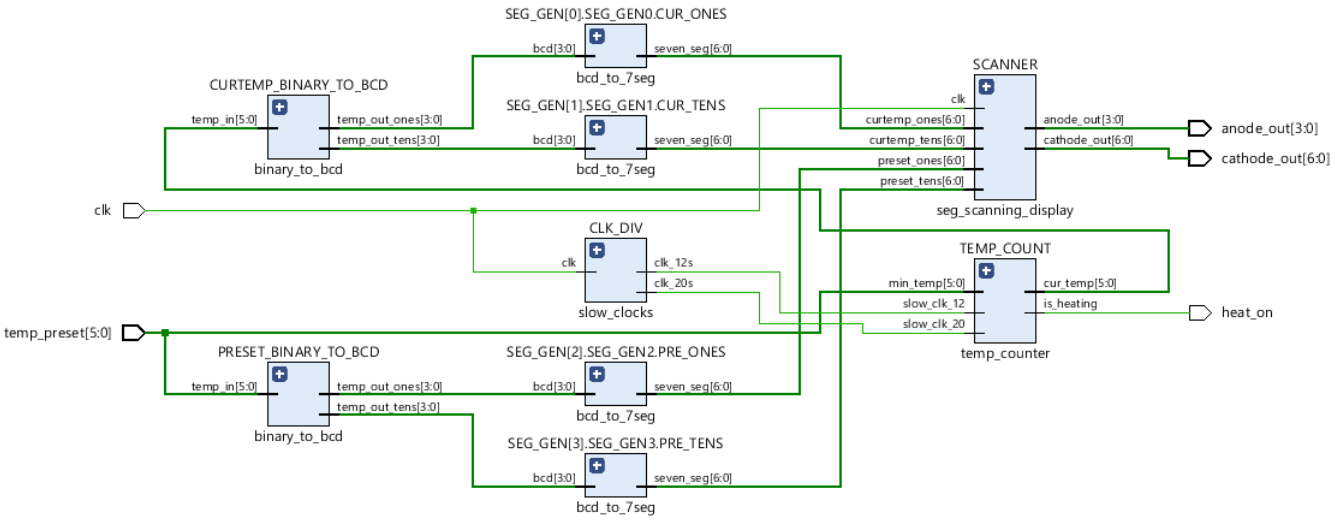


Figure 14: Automatically generated schematic

The physical placement of components on the FPGA is shown in Figure 15. Most of the logic has been implemented using look-up tables (LUTS), which are merely abstracted truth tables describing a logic function. The highly parallel structure of the FPGA is apparent when the bottom right corner of the design is closely inspected. This inherent parallelism is what makes FPGA's very fast and suitable for high-speed applications.

It is also worth noting how little of the overall real estate available on the FPGA has been used by this design; this shows that FPGA's have the capability to implement far more complex logic than that which is found in this program.

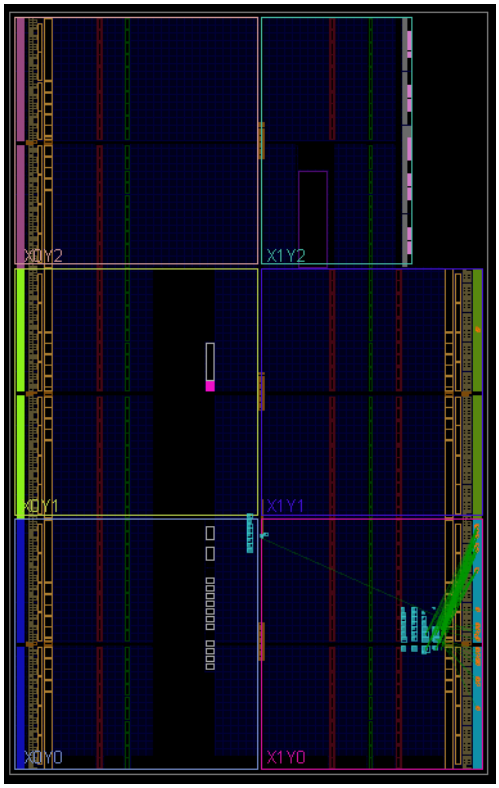


Figure 15: Hardware realisation

7 Implementation on the Basys3 board

From the successful simulations and lack of synthesis errors, it would be expected that the design should translate properly onto hardware. However, this was not the case. As such, this section will consider the issues which were encountered, and discuss the efforts made to solve these problems.

Firstly, to consider what *was* working. The design correctly displays values on the 7-segment display without flickering issues. Furthermore, the tens and ones numbers displayed for the preset correctly display the user-preset selected using the switches along the bottom of the Basys3 board. These observations suggest that

- The binary to BCD converter is correctly working (the preset is a binary value)
- The BCD to 7-segment value is correctly working (the preset is being displayed correctly on the 7-segment)

- The scanner display is working as intended to illuminate each of the 7-segments at a suitable refresh rate

Now to consider the problems. On the first effort to implement the design, the first two 7-segment numbers on the display (representing the `cur_temp`) would count up beyond the `temp_preset` to the maximum possible value of 63, then wrap around and repeat this process indefinitely. All the while, the heater LED stayed on which was correct to indicate that the heater was on. Additionally, the increase in temperature occurred every 12 seconds, as was expected.

This strange and unexpected behaviour does not reflect that of the simulation shown in the waveform of. As this waveform shows, the temperature should count up to the preset and stabilise at this value - only reaching a temperature of $1°C$ greater than the preset as the heating is toggled off and on. Despite efforts to make minor edits to the code of Listing 3, there was no success in making the count stop at the preset.

In consulting with teaching staff during the lab session, it was discovered that the source of the problem was due to the use of two clocks in the sensitivity list of the `delta_t` process. Unknown to group participants, and beyond the scope of this course, was the knowledge that using two clocks in a process sensitivity list is bad practice and an error-prone strategy for writing VHDL code. The reason for this is that the flip-flops and other sequential components typically only support a single clock input. As such, these processes do not map to hardware as intended. The advice was given to the group to reconsider and refactor the existing code to avoid the use of two clocks being used in a single sensitivity list.

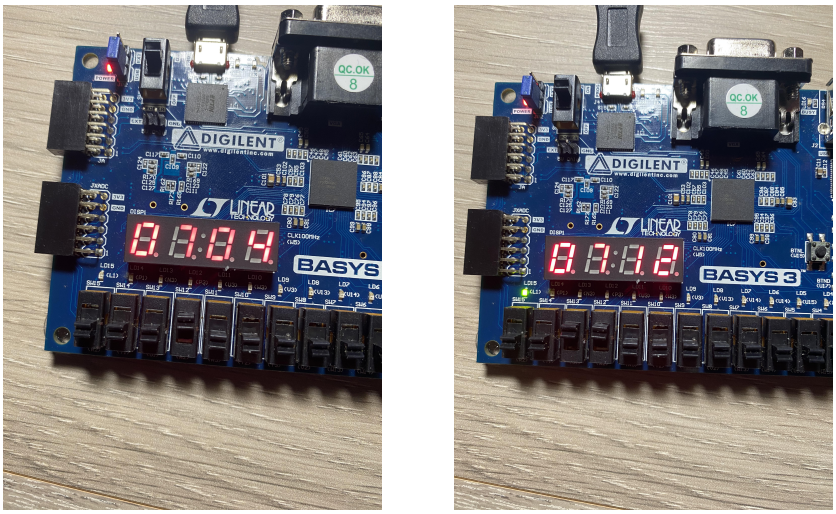
To begin tackling this problem, the `slow_clocks` entity was removed from the hierarchy and the clock division process was integrated *into* a single temperature counter process. This new approach creates a single process that only has the 100 MHz clock in its sensitivity list. On every rising clock edge, this would increment a counter variable for both the 20-second and 12-second internal count. If these counters reach their maximum value, declared as a constant in the architecture declaration region, then conditional branching allows the current temperature to be adjusted accordingly by increasing (if the current temperature is equal to or below the preset) or decreasing it (if the current temperature is above the preset). The VHDL code to reflect the changed architecture is shown in Listing 11.

In this code listing, note the key change: the `temp_change` process only contains a single clock in its sensitivity list. The workaround for maintaining two separate counts without feeding these as inputs to the entity is to delegate the counting responsibility to the process which is also responsible for adjusting the current temperature. This adds to the complexity of the process which is not ideal but is a suitable workaround to avoid the issues associated with crossing clock domains that were present in this first iteration of the design.

```
entity temp_counter is
    Port ( clk : in std_logic;
           min_temp : in std_logic_vector (5 downto 0);
           cur_temp : out std_logic_vector (5 downto 0);
           is_heating : out std_logic);
end temp_counter;

architecture Behavioral of temp_counter is
constant default_clk : integer := 100000000; -- synth clk
constant max_count_12s : integer := default_clk * 12;
constant max_count_20s : integer := default_clk * 20;
signal internal_temp : unsigned(5 downto 0):= "000101";
begin

temp_change : process(clk)
variable count12 : INTEGER := 0;
variable count20 : INTEGER := 0;

begin
    if (rising_edge(clk)) then   --increment both the counters
        count12 := count12 + 1;
        count20 := count20 + 1;

        if (count12 = max_count_12s) then
            count12 := 0;
            if(internal_temp <= unsigned(min_temp)) then
                internal_temp <= internal_temp + 1;
                is_heating <= '1';
            end if;
        end if;

        elsif (count20 = max_count_20s) then
            count20 := 0;
            if(internal_temp > unsigned(min_temp)) then
                internal_temp <= internal_temp - 1;
                is_heating <= '0';
            end if;
        end if;
    end if;

end process temp_change;

cur_temp <= std_logic_vector(internal_temp);

end Behavioral;
```

Listing 11: Revised temp counter file

The revised solution shown in Listing 11 contains the changes made towards eradicating the dual feature representation of clock signals within a sensitivity list. The robustness of the revised code was tested through the use of a testbench where the clock signals, preset temperature, and current temperature were viewed and analysed. After inspection of these waveforms, the correct functionality of the code was confirmed and a bitstream file was created. The group commenced synthesis of this design on the Basys3 board where the following output, shown in Figure 16, was yielded.



(a) Counting Down (b) Counting Up

Figure 16: Count Down and Count up test cases

The `temp_preset` is displayed on the two 7-segment displays farthest to the right. As demonstrated in Figure 16, the `temp_preset` is operating properly as the two correctly displayed digits align with the bit inputs which are dictated by the first 6 switches at the bottom of the Basys3 board. The switches in Figure 16(a) have a configuration that can be represented as "000100" in binary which directly translates to "4" in denary. The switches in Figure 16(b) have a configuration of "001100" which yields a denary output of "12. The correct operation of this is confirmed by the 7-segment display.

It is worth noting that when the `cur_temp` remains at the same value when it reaches the `temp_preset`, with the heater active. Only after the `temp_preset` changes does the `cur_temp` continue to increment or decrement as required.

This final design can be considered successful as it satisfies each element of the project specification as stated in the Introduction.

8 Conclusions

The design of the thermometer in this project has reinforced the concepts introduced in the laboratory sessions to provide the developers with a better understanding of the VHDL language and HDL workflow.

By breaking down the larger design into sub-entities that could be individually tested, the usefulness

of a divide-and-conquer strategy to development was evident. The large and difficult task could be broken down into smaller pieces which facilitated the division of tasks in the programming stages of the design. Separate developers could focus solely on their assigned sub-system which reduced merge conflicts on Git/Github and encouraged more productive collaboration.

The project has provided valuable experience in all stages of the FPGA design process, from simulation, synthesis, implementation, and bitstream generation.

The difficulties faced when implementing the design onto hardware highlighted a unique challenge to hardware programming; it is possible to successfully simulate a design yet experience issues when it is translated onto a physical device. This was not something developers in the team were used to, coming from a background in software programming. However, this emphasises the importance of treating VHDL as a separate entity (pun intended) from a traditional software language. Particular care has to be taken when writing processes with respect to the signals included in the sensitivity lists - as was the root of the problem experienced in this project.

By ultimately resolving the issues experienced when implementing the design on hardware (after lots of edits and revisions to the code), the merits of FPGA devices compared to ASICs were made abundantly clear. The numerous hours spent attempting to remedy the issues by reprogramming the device paled in comparison to the expense which would have been involved in the redesign and manufacturing of a failed ASIC device as these have very high NREs (Non-recurrent engineering costs). This provided developers with a better understanding of the use cases for FPGAs and advantages compared with other digital integrated circuits.

Overall, this experience has highlighted the usefulness of FPGAs in the electronics industry and provided insight into how they can be used to realise a design from conception, all the way to implementation on physical hardware.

References

[1] Conversion of 4210 from binary (101010) to BCD, David F. Brailsford, Online Resource (Accessed 16/03/2023), Available at: http://www.eprg.org/computerphile/fortytwo.pdf

[2] Reference Manual (Basys3), Online Resource (Accessed 16/03/2023), Available at: https://digilent.com/reference/programmable-logic/basys-3/reference-manual

[3] Temperature contols and central heating, Online Resource (Accessed 17/03/2022), Available at: https://www.greatrads.co.uk/blog/what-temperature-should-the-central-heating-thermostat-be-set-at-2