Introductory VHDL and FPGA Design

Design Project Report

Sports Venue Attendance Counter

Group 01

Group member: Registration Number

Muhammad Taha Shahid: 202104809

Lena Krabbe: 202104728

Kieran Frost: 201924874

Submission date: 29th March 2021

Abstract

This report outlines the VHDL logic used and implemented in order to satisfy the conditions set by a design brief. This brief required a stadium counter system to be created where the attendance of each individual stand should be monitored. If a stand exceeds 90% capacity, an LED should light. Furthermore, the overall attendance of the stadium should then be displayed (in decimal) on 7-segment displays. This functionality was achieved using a 3-tier hierarchy system. This resulted in the project being split in two sections: a stadium counter section (utilizing individual counters) and a display entity.

The main conclusions drawn from this report are how specific VHDL statements and design approaches may be used in order to satisfy the user specifications. Furthermore, it is shown how a suitable set of test inputs may be created so that the functionality of the design can be verified. It is also shown how this project may be implemented on a Basys-3 FPGA board for real-world application.

Contents

1. Introduction

Keeping track of the attendance in situations like a game in a sports venue is an important and critical task. Most importantly one can guarantee that there are not more people entering a certain room or section than there are seats or space available. This is relevant for general safety or emergency situations. Moreover, the information of attendance can be used for statistics or for optimizing aspects like advertising, selling drinks and food or allocating security staff.

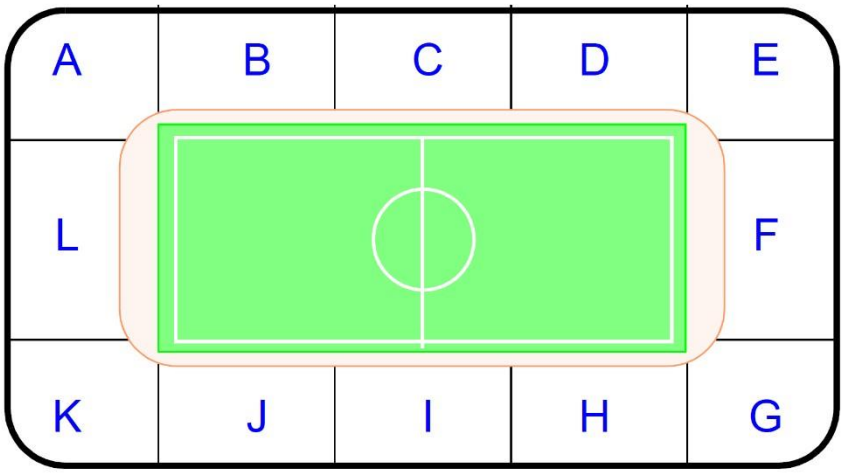

Figure 1: schematic drawing of sports venue

In this work the attendance in a sports venue is observed by tracking the number of people entering over time. For that task, a counter is designed and simulated in VHDL using Vivado. Although it is verified that the circuit could be implemented into actual hardware on an FPGA device, this framework is rather theoretical and does not include programming or using the FPGA itself.

The design approach comprises of one counter for each of the 12 sections A to L in the venue, as seen in Figure 1, so that the attendance in each respective section can be tracked (as entry to the venue is by section). The maximum attendance per section is set to 250 and if 90% capacity is obtained for a section, a corresponding LED is illuminated. The total attendance is obtained by summing the individual 12 counters and should then be displayed on a set of four 7-segment displays over time. After a game in the sport venue is over, the attendance is reset to zero, as everyone will have left soon after the game end.

2. Design Approach

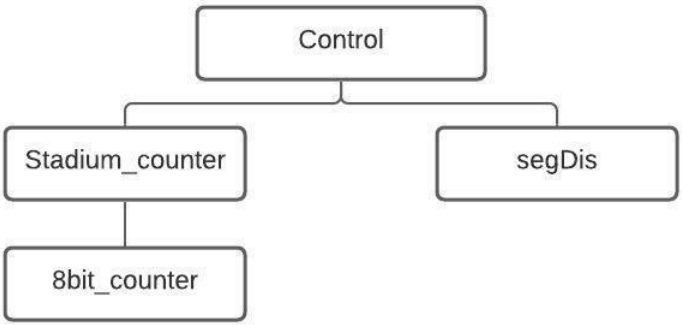

Figure 2: Hierarchy diagram for design approach

Figure 2 shows the design approach of the system. “Control” is the top level of hierarchy as it is dependent on the output of the actual counter “Stadium_counter” and the display “SegDis”. “Control” feeds the output of the stadium counter to the 7-segment display entity. Furthermore, we have a lower level component in the project design labelled “8bit_counter”: it is a common 8-bit counter that can increment an 8-bit binary number. This component is used inside “Stadium_counter” to count the number of people entering the stadium.

3. Counter

3.1 8-bit Counter from Lab 8

It became apparent from the project brief that the stadium counter would rely heavily on the 8-bit counter design entity generated in Lab 8. This counter has an entity declaration as shown in Figure 3.

```
entity counter_8bit is
    Port ( CLK : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Enable : in STD_LOGIC;
           Load : in STD_LOGIC;
           UpDn : in STD_LOGIC;
           Data : in STD_LOGIC_VECTOR(7 downto 0);
           Q : out STD_LOGIC_VECTOR(7 downto 0));
end counter_8bit;
```

Figure 3: 8-bit Counter Entity

As evident from this declaration, the counter has five standard logic inputs as well as an input standard logic vector of size 8 called Data. In turn the counter returns an 8-bit standard logic vector called Q.

In the Lab 8 exercises, each of these inputs had the following specifications:

- Reset – Asynchronous where if '1' then Q should be set to zero
- CLK – synchronous clock input so rest of inputs only affect output on rising edge
- Enable – If '0' then no matter what, the value of Q would remain unchanged if Reset=0
- Load – If '1' then Q should equal Data
- UpDn -If '1' and Load is '0' then Q should increment by 1, and if UpDn is '0' and Load is '0' then Q should decrease by 1

It can be seen from this functional specification that there are some redundant features in this design. For example, "Load" would not be used and "UpDn" would always equal '1' as the design brief did not state that the counter should also count down. However, the counter could be run by varying the Enable and Reset. It was therefore determined that if a person walked into a stadium section, then the Enable would be high. Furthermore, when the match has ended then the global stadium counter reset can be fed to the individual 8-bit counter reset input ports, in order to set the stadium sections back to an attendance of 0.

After considering the design brief, it became clear that an additional if statement should be added at the end of the architecture body so that the 8-bit number "Q" would remain 8-bit. This means that if the counter counts to "11111111", it would be reset to "11111110" before the next clock cycle as without this, the variable would have returned to 0.

3.2 Using 8-bit Counters to implement Stadium Counter

From the design brief, it was specified that there were 12 stadium sections (Stand A through L) and in each of these sections there was a maximum attendance of 250. This means that the stadium counter would have to individually count each section rather than having an all-encompassing attendance counter. It is plain to see that there would therefore need to be 12 individual 8-bit counters produced (i.e. one for each section) as well as an input std_logic_vector array called Enable and length 12.

In VHDL these counter component instantiations can be accomplished by using direct component instantiation or for-generate statements. The difference between the two is that for direct component instantiations, each counter would have to be explicitly declared and initialised. On the contrary, forgenerate statements utilise 2 key VHDL aspects: a for loop and a generate statement. A general forgenerate statement can be seen below:

```
Component_gen: FOR i in 0 to (**number of components desired** -1) generate
      Component_element_gen :**Name of component being instantiated** port
      map(**link local ports to global ones**);
end generate;
```

As seen in the above code, "i" is called an iterator and is initialised to be 0. Every time the procedure executes the value of "i" increases by 1, and hence a new component instance is generated for every executed loop iteration. This type of generate statement only works when the port to port/signal mapping is constant for every desired component. This therefore causes difficulty if an output value of the component is stored in a different location for every component instance. As a direct consequence, it was

decided that in order for a for-generate statement to perform adequately for the stadium counter design, a new type had to be created.

3.3 Type Creation

Since each counter for each section of the stadium will produce an output value "Q" which is an 8-bit binary number, a bespoke array type needed to be created. Simply put, it was desirable to create an array type which could store 12 standard logic vectors of size 8. This was done by the code segment shown in Figure 4:

```
type storer is array(0 to 11) of std_logic_vector(7 downto 0);
signal dl : storer; --container to store data from sections
```

Figure 4: Bespoke Storage Container

A signal is created called "dl" which is of type "storer". The type "storer" is an array type of size 12, with each array element being a standard logic vector. The element locations of "dl" are defined to be dl[0] up to dl[11] and each of these elements are themselves standard logic arrays of size 8. In the design process, it was determined that dl[0] would correspond to the total number of people in stand A and dl[11] would correspond to the total number of people in stand L.

This could therefore be incorporated into the for-generate statement and the output of the component instance would write to dl[i] where "i" is the component instance number.

3.4 Stadium Counter For-Generate

With the relevant tools now being able to be utilized, it can now be seen in Figure 5 how the for-generate statement shown below could be created:

```
section_counter_gen: FOR i in 0 to 11 generate

    count_elem_gen:counter_8bit
        port map(CLK=>CLK,Reset=>Reset,Enable=>Enable(i),Load=>'0',UpDn=>'1',Data=>dl(i),Q=>dl(i));

end generate;
```

Figure 5: For-Generate Statement

The left-hand ports correspond to that of the 8-bit counter, and the port/signals on the right correspond to global stadium counter ports and signals. It can be seen that, for example, the first counter(A) created would have an enable equal to that of Enable[0] and "Q" would write to dl[0], whereas for the second counter(B) created, the enable would be determined by Enable[1] and "Q" would output to dl[1]. This process would repeat until all 12 counters are generated.

3.5 Checking Counter Values

It then became apparent that following these component instantiations, each element of "dl" would have to be checked to verify that it does not exceed 250. It was found that a VHDL process should be used to best complete this functionality. Since this process only needs to be executed when there is a change in one (or more) elements of "dl" it was decided that the signal "dl" should be included in the process sensitivity list as in VHDL, a sensitivity list is effectively a list of watched signals and the process will execute

when a sensitivity list variable changes. Following this, it then needed to be determined what functionality should be used to satisfy the check.

It should be noted that a variable (of type “storer”) within the process should be set equal to the signal “dl”. This is because of the unique properties of variables inside processes. Variable values get updated immediately within a process, whereas a signal value only gets updated after the process has executed (i.e. completed or reached a “wait” statement). Since it is desired that if a section has more than 250 people in it after the counter stage, the stand count should be immediately set to 250 therefore a process variable should be created.

A for loop should then be used to iterate through each element of the variable and if the integer value of the unsigned binary number is greater than 250, it should be reset back to “11111010” (i.e. 250 in binary). This exact functionality can be seen in the highlighted segment of code in Figure 6, where “max_seats”

```
stad_check:process(dl) --process to check that all are b
    variable total_attendance: unsigned(11 downto 0);
    variable temp_init :unsigned (7 downto 0);
    variable temp_resize: unsigned(11 downto 0);
    variable stad_area: storer; --needed to not cause dr

begin
    total_attendance:="000000000000";
    temp_init:="00000000";
    temp_resize:="000000000000";

    for j in 0 to 11 loop
        if to_integer(unsigned(dl(j)))>max_seats then
            stad_area(j):="11111010"; --so number of sea
        else
            stad_area(j):=dl(j); --else if less that 250
        end if;
```

Figure 6: Part of Stadium Check Process

has been defined in the architecture body to be a constant integer of value 250.

Following on from this, additional functionality can be attached to this process. One of the specifications of the design brief is that a total stadium count can be displayed on 7-segment displays. Therefore, in this stadium counter VHDL file, it should be noted that there should be an output called “total_count” which is the unsigned value of the sum of all the individual sections. This “total_count” could then be passed to the display section of the design. Since each section can only hold 250 people, and the stadium contains 12 sections then the maximum value that “total_count” could hold as an integer would be 3000. In unsigned binary, 3000 is a 12-bit number and so it was determined that “total_count” should be a 12-bit unsigned output. It can therefore be seen that within this “stad_check” process, an unsigned addition section could be added such as that shown in Figure 7.

```
stad_check:process(dl) --process to check that all are belo
    variable total_attendance: unsigned(11 downto 0);
    variable temp_init :unsigned (7 downto 0);
    variable temp_resize: unsigned(11 downto 0);
    variable stad_area: storer; --needed to not cause drive

begin
    total_attendance:="000000000000";
    temp_init:="00000000";
    temp_resize:="000000000000";

    for j in 0 to 11 loop
        if to_integer(unsigned(dl(j)))>max_seats then
            stad_area(j):="11111010"; --so number of seats
        else
            stad_area(j):=dl(j); --else if less that 250 it
        end if;

        temp_init := unsigned(stad_area(j)); --initialising
        temp_resize := resize(temp_init,12); --resizing as
        total_attendance:=(total_attendance+temp_resize); --
    end loop;
    total_count<=total_attendance; --mapping total_count to

end process;
```

Figure 7: Full Stadium Check Process

As evident in the above process, the unsigned addition section has 4 distinct stages:

- A) “temp_init” is assigned to be the unsigned binary number of “stad_area” for that iteration
- B)”temp_resize” converts “temp_init” into a 12-bit unsigned number
- C)”total_attendance” is set to be a compound variable were every element of “stad_area” and hence every “temp_resize” value is added to it
- D) Finally, the “total_count” output of the stadium counter is assigned to be equal to the final “total_attendance” value

The resize stage is fundamental to unsigned addition in VHDL, as if two unsigned number of different sizes are summed an error will occur.

3.6 Lighting LED

From the design brief, one of the user specifications outlines that if a stadium section exceeds 90% capacity, an LED should light. For our stadium design, the total number of seats in each section is 250 therefore if 225 people entered the stand, an LED should light. A constant integer variable was therefore declared in the architecture body called “led lit” and set to this value.

Since there are 12 sections within the stadium, a standard logic vector of size 12 was created as an output in the entity declaration where each element corresponds to an individual section (e.g., LED[0] corresponds to A, LED[1] corresponds to B etc.). It also became clear that a separate process should be created for this to minimise the time taken for an LED to light.

Once again, a for-loop was utilised inside the process to iterate through every element of “dl” being casted to a “storer” variable called “light_check” and if this element exceeds this “led_lit” value then the corresponding LED turns high. This process can be seen in Figure 8.

This process “Light” will execute concurrently to “stad_check” and hence will mean that an LED would light quicker than if this for-loop was incorporated into the “stad_check”. This is since processes execute sequential statements.

```
Light:process(dl) -- process to light corresponding LED when over
variable light_check: storer; --needed to not cause driver error o
begin

    for j in 0 to 11 loop
        if to_integer(unsigned(dl(j)))>max_seats then
            light_check(j):="11111010"; --so number of seats canno
        else
            light_check(j):=dl(j); --else if less that 250 it stay
        end if;

        if to_integer(unsigned(light_check(j)))>led_lit then --if
            LED(j)<='1';
        elsif to_integer(unsigned(light_check (j)))<led_lit then --
            LED(j)<='0';
        end if;
    end loop;
end process;
```

Figure 8: Lighting Process

3.7 Counter Design Block Diagram

It was therefore determined that the stadium counter structure should look similar to that in the block diagram of Figure 9. An Enable array of size 12 would be passed so that each element corresponds to the ‘Enable’ of the relevant counter (e.g. Counter A would receive Enable [0], B would receive Enable[1] etc.) as well as the clock input and reset. The output of these counters would then be contained in the container “dl” before being subject to two processes: one to light an LED and one to sum the terms to produce a final attendance unsigned number.

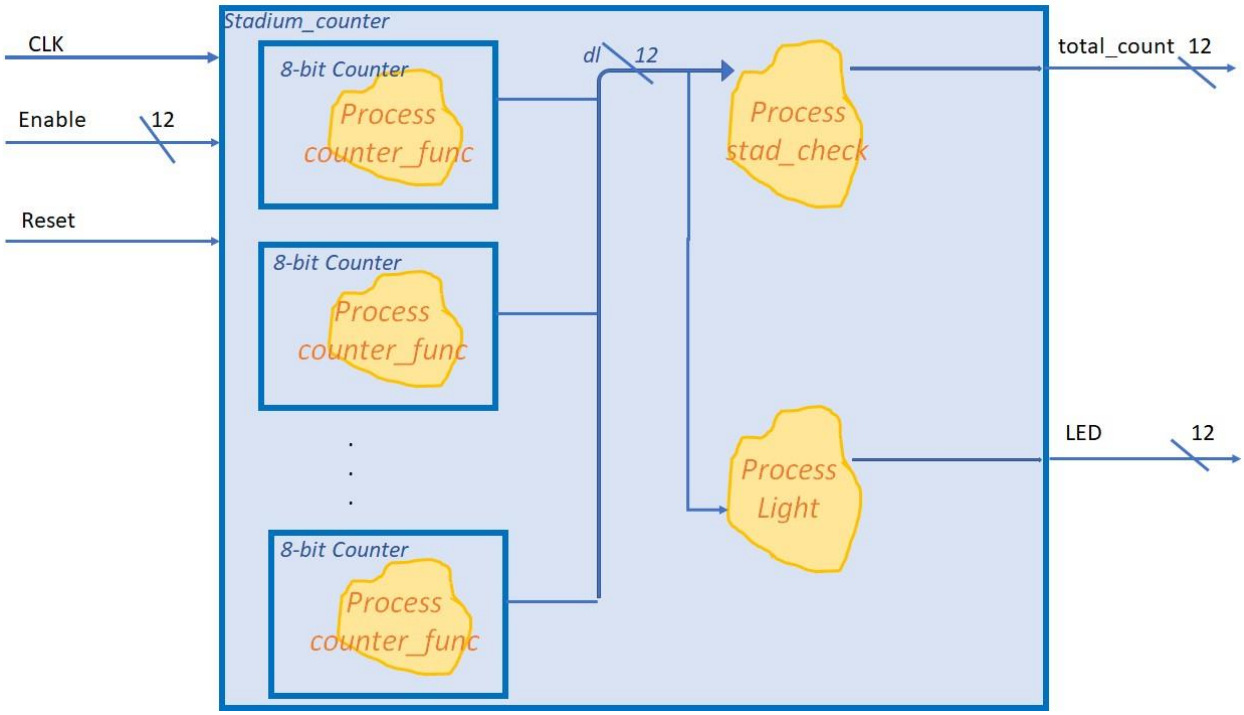

Figure 9: Stadium Counter Block Diagram

3.8 Stadium Counter Initial Testing

Before the stadium counter code was incorporated into the larger hierarchy, a simple testbench was constructed in order to verify that all the necessary functionality is achieved. In this testbench, the signal Enable was initialised as "00000000000" so that a warm-up phase is achieved by the entity. Following this, on the next rising clock edge, Enable was changed to"100000000000" for 60ns. It was expected that therefore on every rising clock edge that the total-count should increase by 1. Therefore, after 80ns the total stadium count should be 3. Following this, Enable was again changed to "100100001000" for 20ns. This would correspond to only one clock cycle so following this, the total stadium count should be 6. Finally, following this "Reset" is set to '1' and therefore it is expected that the total-count should be reset back to 0. The results of this simulation are shown in Figure 10. It is also important to note that the attendance of each individual section was monitored by using the scope/object feature of the simulation. This therefore meant that the storage container "dl" could be tracked, and each section's count could be verified.

As evident in Figure 10 the simulation results correspond to the expected, theoretical results. However, a further testbench needed to be created so that the LED lighting procedure could be analysed. Since an LED would only light when there are 225 people in a stand, the simulation needed to run for approximately 4500ns to validate that an LED turn high. For this simulation, Enable was initialised at "000000000000" and following the first clock cycle it was then changed to "100100001000" for the entirety of the remaining simulation. Therefore, it was expected that after 226 clock cycles LED[0], LED[3] & LED[8] should all turn high. The results corresponding to this testbench are shown in Figure 11.

From the traces of Figure 11 it can be seen that the LED array stays at “000000000000” until around 4.5µs. However, after this, a change occurs. The trace around this point was then expanded to produce Figure 12. Therefore, at the t=4520ns it can be seen that LED[0], LED[3] & LED[8] turn high which is the desired result.

The stadium counter was then incorporated into the “Control” parent VHDL code in order for the total stadium count to be displayed on 7-segment displays.

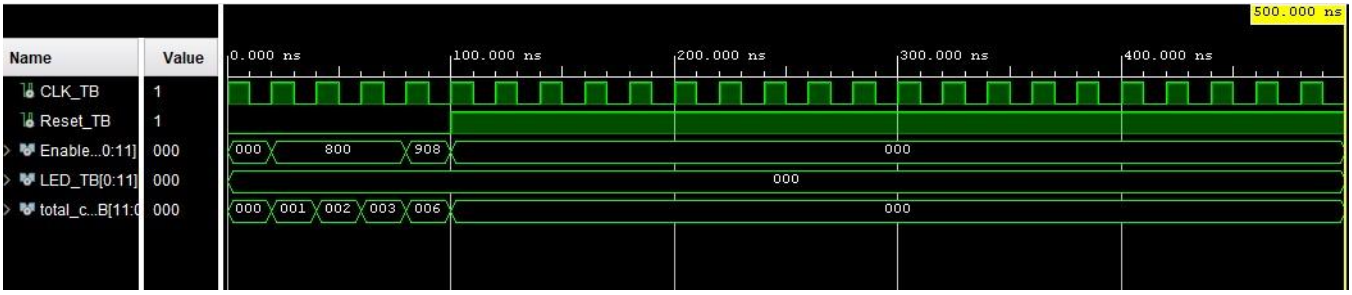

Figure 10: Waveform of Stadium Counter Testbench

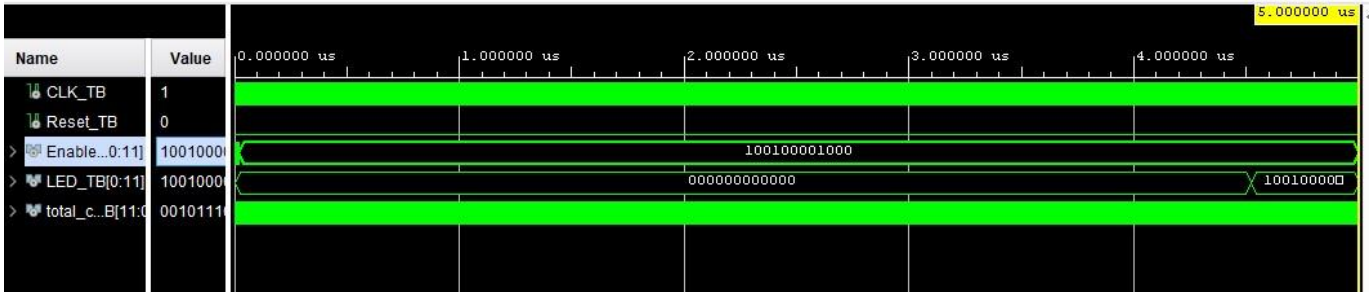

Figure 11: LED Testbench Results

Figure 12: Expanded LED Results

3.9 Troubleshooting

Initially in the design project, the stadium counter entity caused several issues. Firstly, in the “Light” process each element of “dl” was simply converted to “unsigned” type before being compared with an integer. It was quickly realised that this cause the LED array output to be undefined as this sort of comparison cannot be made. This was therefore rectified by utilising the “to_integer” casting in VHDL. Following this, the LED operated as required. However, there was another, deeper issue where the total stadium count in the testbench returned “XXXXXXXXXXXX”. It was realised that an ‘X’ output was

commonly the result of a driver error in VHDL. Therefore, the initial code was consolidated. From this investigation it was found that, rather than creating a new “storer” type variable in the “stad_check” process, the signal “dl” was changed explicitly if the count value was larger than 250. This was therefore causing a driver error, as the variable “total_attendance” had a direct correlation with the “dl” and thus since signal values only update when the process suspends, this was causing an undefined output. To rectify this, a “storer” type variable was created in the process called “stad_area” which was initialised to be a mirror variable of “dl”. When the initial, simple testbench was then run, all outputs of the stadium counter entity were declared, and appropriate values were obtained.

4. 7-Segment Display

4.1 General information

To display all possible attendance values in the range of 0-3000, all four 7-segment display units of the FPGA are used. Figure 13 shows the FPGA with the four 7-segment displays in the bottom left corner. Originally, the segment display design is implemented driving cathodes and anodes of each segment using a specific scanning timing as described in the reference manual (see [1]). However, for this project framework, the FPGA board is not available and therefore each 7-segment display will be described as a 7-bit array of logic values that describe the status of segments a to g (see Figure 14): 0 is low or inactive (grey) and 1 is high or active (blue). Therefore, all four digits of the decimal number to be displayed can each be described using a variable of type std_logic_vector of 7 bits.

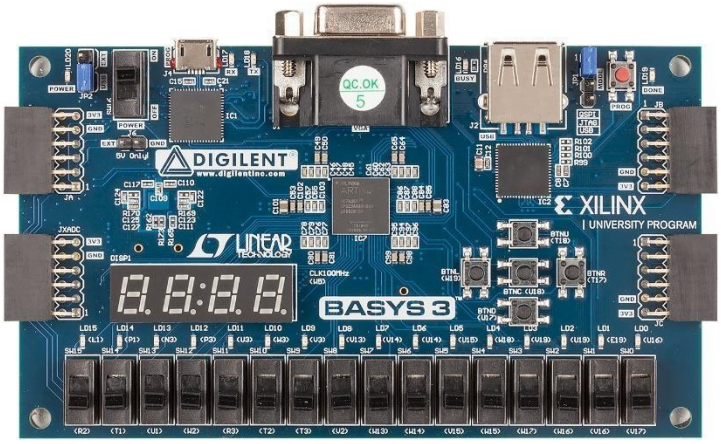

Figure 13: Basys3 FPGA Board with 7-segment display in bottom left part

4.2 Display Interface

The input of this display design is a 12-bit unsigned binary number called “attendance”. This satisfies the project brief since only positive numbers are considered and because 12 bits are sufficient to display a maximum of 3000 attendees in the stadium. As indicated previously, there are four outputs of type std_logic_vector each containing 7 bits to drive the four individual 7-segment displays. Therein, “seg1” is describing the most significant decimal digit (i.e. referring to the most left 7-segment display in Figure 13),

and “seg4” represents the least significant decimal digit as the most right 7-segment display. This design interface is also shown in Figure 15.

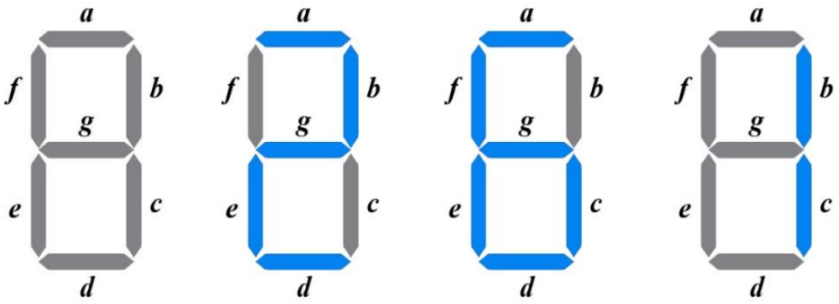

Figure 14: 7-segment displays describing decimal numbers: all segments are inactive (left), active segments display a decimal two (second from left), a decimal six (second from right) and a decimal one (right)

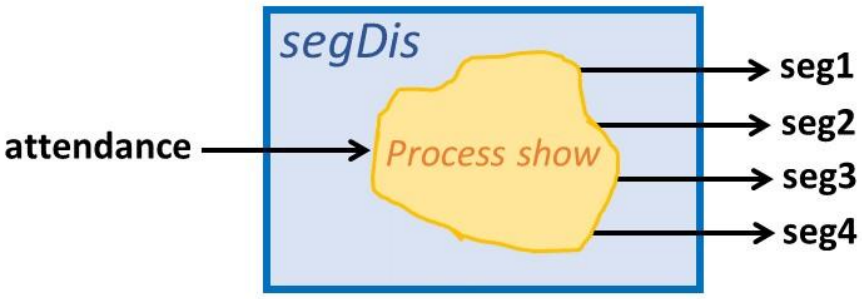

Figure 15: Design entity of 7-segment display showing the interfacing

4.3 Display Functionality

The declaration region of the architecture contains the definition of constants of type “std_logic_vector” that describe which segments of each 7-segment display are active or not active to generate the decimal numbers of zero to nine. Thereby, the order is alphabetic, i.e. the first bit refers to segment a, the second to b etc., as depicted in Figure 14 for the decimal numbers two, six and one. The code snippet in figure 16 also clarifies this.

```
constant one: std_logic_vector := "0110000";
constant two: std_logic_vector := "1101101";
constant three: std_logic_vector := "1111001";
constant four: std_logic_vector := "0110011";
constant five: std_logic_vector := "1011011";
constant six: std_logic_vector := "1011111";
```

Figure 16: definition of constant vector to represent decimal numbers on a 7-segment display

Moreover, a subtype "intSmall" is defined that covers only integer value in the range of [0,3000] to reduce storage for following variables in the process.

The central process "*show*" in this design's architecture includes several steps to generate the four outputs. To catch every change in attendance, the input attendance number is included in the process sensitivity list. With every change in attendance, the process wakes up and the statements are executed sequentially. Therefore, it is not necessary to include a clock signal as an input to the design nor to the process.

After declaring and initializing several process variables, there are essentially two steps that are performed inside a for-loop. This for-loop has four iterations (as there are four digits to assign) and starts with addressing the most significant decimal digit for loop variable $i = 0$. Therein, the first step is to calculate the respective single decimal digit from the integer attendance number "*att_int*" (which had been obtained before the for-loop by converting the input attendance of type unsigned to an integer number). This is done by employing the modulo operator. For better understanding, this first step of the "*show*" process is shown for an example attendance number of $att\_int = 4832$ with variable names matching the actual VHDL code: the underlying equation to calculate the current digit "$noTemp$" is

$$noTemp = \frac{att\_int - dif - (att\_int \bmod fac)}{fac} .$$

Initially, the variables "dif" and "fac" are set to 0 and 1000 respectively; after each calculation of "$noTemp$" both are updated according to

$$dif \rightarrow dif + noTemp \cdot fac$$

$$fac \rightarrow fac \,/\, 10$$

As the variable name suggests, "$noTemp$" is overwritten for each iteration. Table 1 shows the four iterations to obtain the four digits of the example decimal number 4832.

Table 1: Calculating the single digits of decimal number 4832

iteration	$noTemp$	dif	fac
0	$\frac{4832 - 0 - 832}{1000} = 4$	0	1000
1	$\frac{4832 - 4000 - 32}{100} = 8$	$0 + 4 \cdot 1000 = 4000$	100
2	$\frac{4832 - 4800 - 2}{10} = 3$	$4000 + 8 \cdot 100 = 4800$	10
3	$\frac{4832 - 4830 - 0}{1} = 2$	$4800 + 3 \cdot 10 = 4830$	1

Subsequently, the second step is a case statement that compares the current "$noTemp$" value with the possible decimal numbers of zero to nine and then assigns the corresponding constant "std_logic_vector" that had been defined in the declaration region of the architecture (see code snippet in Figure 16). Thereby it is important to assign the constant to the right segment output ("seg1", "seg2", "seg3" or "seg4"). To do so, the helper variable "*segAll*" of the same type ("std_logic_vector") but quadruple size is

introduced (before the for-loop) as a concatenation of the four outputs. As the loop variable i refers directly and unambiguously to a certain digit, the "*segAll*" array is filled with the corresponding constant array at the positions

$$(i \cdot 7 \text{ to } i \cdot 7 + 6)\,.$$

After the for-loop, the four outputs "seg1" to "seg4" are assigned to the values of the respective partial vector, which is also shown in the code snippet of Figure 17.

```
--divide concatenated segAll into 4 7-segment displays (vectors)
seg1 <= segAll(0 to 6);   --representing most significant digit
seg2 <= segAll(7 to 13);
seg3 <= segAll(14 to 20);
seg4 <= segAll(21 to 27); --representing least significant digit
```

Figure 17: assignment of outputs from concatenated variable "segAll"

4.4 Display Testbench

To check if the 7-segment display shows the desired functionality, a testbench is created for this part of the total attendance counter design. In doing so, a variety of attendance inputs have been tested as stimulus to assure that the created design can handle one- to four-digit attendance numbers. Verifying the outputs in their original type std_logic_vector in the waveform is not applicable which leads to converting the logic arrays to a different format. This can e.g. be the conversion to the equivalent unsigned decimal number by interpreting the std_logic_vector as binary number and calculating the decimal equivalent. An example is the conversion of seg1 = "0110011" representing the segment decimal four according to

$$0110011_2 = 51_{10}$$

which enables to unambiguously identify if the 7-segment display works as indicated. Table 2 outlines this conversion for all possible decimal segment digits from zero to nine.

Table 2: Conversion of segment outputs from binary vectors to unsigned decimals for better visualization

Number on segment display	Binary vector (abcdefg)	Unsigned decimal
zero	1111110	126
one	0110000	48
two	1101101	109
three	1111001	121
four	0110011	51
five	1011011	91
six	1011111	95
seven	1110000	112
eight	1111111	127
nine	1111011	123

Figure 18 shows the waveform for a series of stimulus inputs. Therein it is visible that the attendance (first line in waveform) is correctly represented in the four 7-segment displays. This is especially important for this framework, as the debugging of the total design including the counter and the 7-segment display would be very difficult if one cannot assure that the single components are functioning as desired. Further, another considerable option to visualize the output vectors would be the conversion to hexadecimal numbers.

In order to further save on power consumption and therefore optimize this design, it should be considered to completely deactivate one 7-segment display if it is displaying a leading zero.

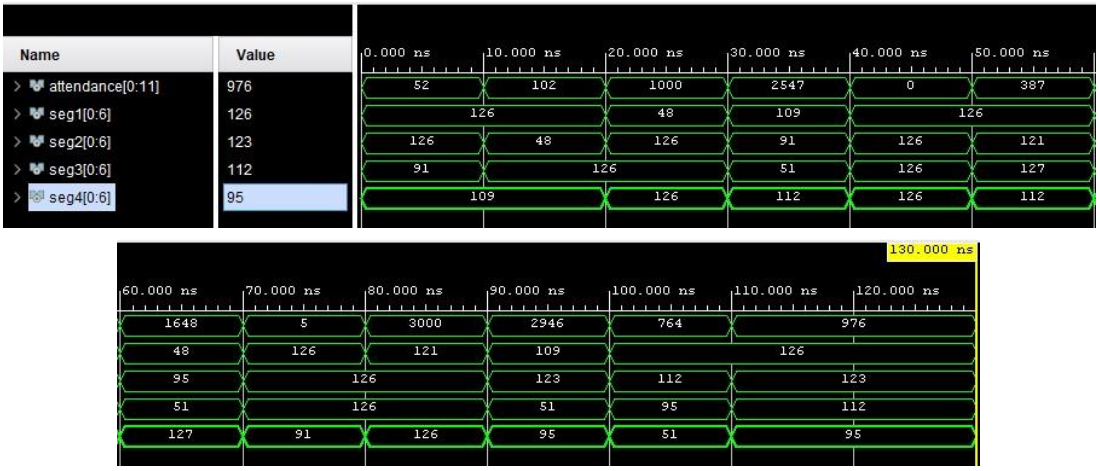

Figure 18: Waveform of 7-segment design testbench to verify functionality

5. Hierarchy

Figure 19 shows the Block diagram for the system. It consists of three main components: Control, Stadium_counter and segDis. The stadium counter entity receives the clock, enable and reset signals and returns both the LED and Total_Count outputs. Total_count is also fed to the Attendance port of segDis, which in turn produces the drivers of four 7-Segment displays (seg1, seg2, seg3 and seg4). This whole system is controlled by the control component which connects the component Stadium_Counter and the component segDis.

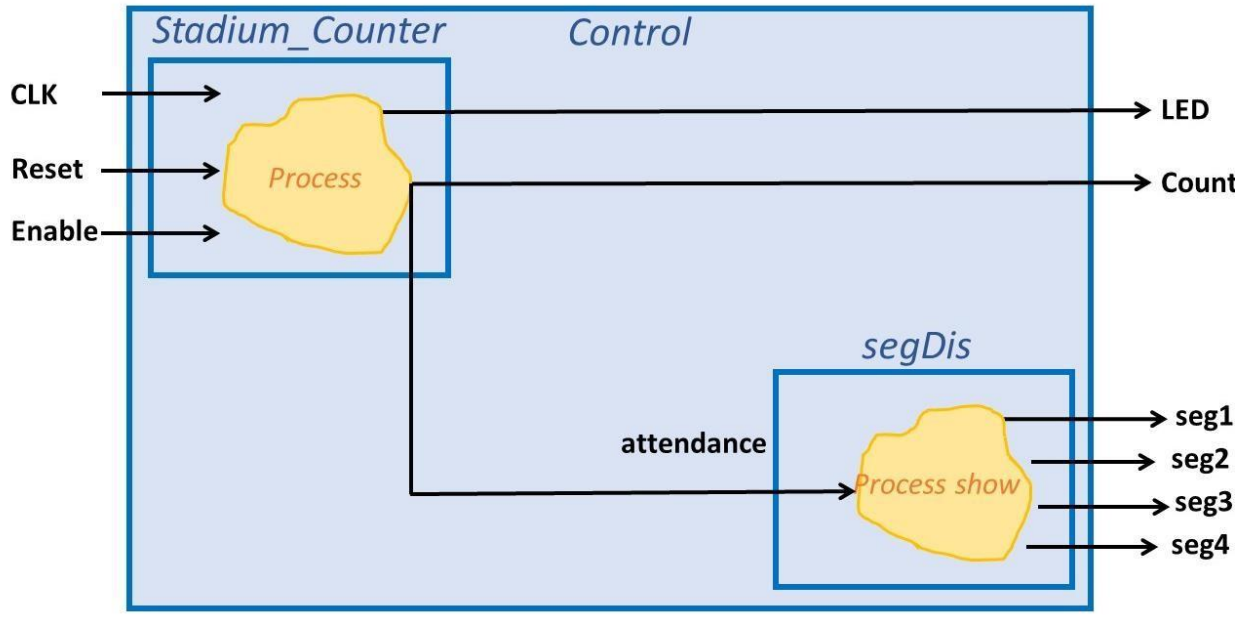

Figure 19: Block diagram for control

```
COMPONENT Control
PORT(
    CLK : IN std_logic;
    Enable : IN std_logic_vector(0 to 11);
    Reset : IN std_logic;
    total_count : INOUT unsigned(11 downto 0);
    LED : OUT std_logic_vector(0 to 11);
    seg1 : OUT std_logic_vector(0 to 6);
    seg2 : OUT std_logic_vector(0 to 6);
    seg3 : OUT std_logic_vector(0 to 6);
    seg4 : OUT std_logic_vector(0 to 6)
    );
END COMPONENT;
```

Figure 20: Ports for control

Figure 20 shows the component control entity declaration. Inside the testbench three inputs are present: CLK, Enable and Reset. CLK represents a clock input which is used to control the synchronous actions of the counter. Furthermore, reset is also fed to stadium_counter and is used to reset the stadium "total_count" at the end of the game. Finally, Enable (which is an array of 12 bits) is passed to the stadium counter. Each bit in that array corresponds to whether a person is walking into a stand and can therefore be used to turn each individual counter on and off. If a '0' input is applied, the counter will remain idle but if a '1' input is applied, the counter will increment. In this Control entity, "total_count" is an "Inout" port. This is because it is an output from stadium_counter but is also an input to segDis. It's value represents the total number of people in stadium which is then displayed to 4 7-sagment displays which are defined as output seg1, seg2, seg3, seg4. Furthermore, there is an LED output which is produced by stadium_counter and is a 12 bits array where each element corresponds to the individual section. The LED only turns high when the section exceeds 90% capacity.

It should be noted that the “total_count” output from the Control is not specified in the design brief however was incorporated in order to verify that the results from the testbench were adequate.

6. Testing of Control Entity

6.1 Justification of Test Inputs

Figure 21 shows the testbench for the clock. It is initialized to ‘0’ but then it enters a loop where its value changes every 0.5ns to not clock. Therefore, if it is ‘0’ it will change to ‘1’ after 0.5ns. Clock process can be used to change the rate of count. The lower the period of the clock, the greater the count will be.

```
Clock: PROCESS
  BEGIN
    CLK<='0';
    for i in 1 to 1000 loop
       wait for 0.5ns;
       CLK<= not CLK;
    END loop;
    wait;
    END PROCESS;
```

Figure 21: Clock generation

Figure 22 shows the test stimuli used to verify the functionality of the control component. It consists of Enable and Reset inputs. This test bench is created to imitate a real game scenario. The counter has a scale where 1ns is equal to 1 minute in real life and clock has delay of 0.5ns which means every 30s some person is entering each section if the Enable bit is ‘1’. The counter is initialised with zero and reset is high, so it is not counting. People start entering the stadium 90 minutes before the game so in order to accommodate this, the reset is changed to ‘0’ and bits in Enable are changed between ‘0’ and ‘1’ to count different sections. In the final line of Figure 22 we can see that all the Enable bits are ‘1’ so that the LED output can be verified. During the game, we can see that there is no one entering the stadium so all the counters are ’0’. After game finishes Reset is turned to ‘1’ to reset the counter.

In this testbench, the Enable of stand A was initially set high and all others low. This was to easily show that the total attendance of the stadium would increase by 1 on every rising edge. Following this, stand B’s Enable was also set high in order to verify that the attendance would only increase by 2 on every rising clock edge. The combination of high Enable signals was then varied every 10ns to verify the attendance and segment displays were producing the desired result. Lastly, the Enable of all stands was then set high for 200ns to verify that once a stand count exceeds 225, an LED would light. Reset was then set high at the end of the simulation to show that this feature is asynchronous.

```
tb : PROCESS
BEGIN
-- Match scale 1ns=1m
Enable<= "000000000000";
Reset<= '1';
--Setting starting values to zero
wait for 5ns;
Enable<= "100000000000";
Reset<= '0';
wait for 10ns; -- counter before 90m of game
Enable<= "110000000000";
wait for 10ns; -- counter before 80m of game
Enable<= "101000000000";
wait for 10ns; -- counter before 70m of game
Enable<= "100010000000";
wait for 10ns;-- counter  before 60m of game
Enable<= "100001100010";
wait for 10ns;-- counter  before 50m of game
Enable<= "100001100011";
wait for 10ns;-- counter  before 40m of game
Enable<= "111001111111";
wait for 10ns;-- counter  before 30m of game
Enable<= "111101111111";
wait for 10ns;-- counter  before 20m of game
Enable<= "111111111111";
wait for 200ns;-- counter  before 10m of game
Enable<= "000000000000";
wait for 10ns;-- counter during the game
Reset<= '1';-- reseting to zero after match ends // counter after the  game
wait;
END PROCESS;
```

Figure 22: Test bench for overall counter design

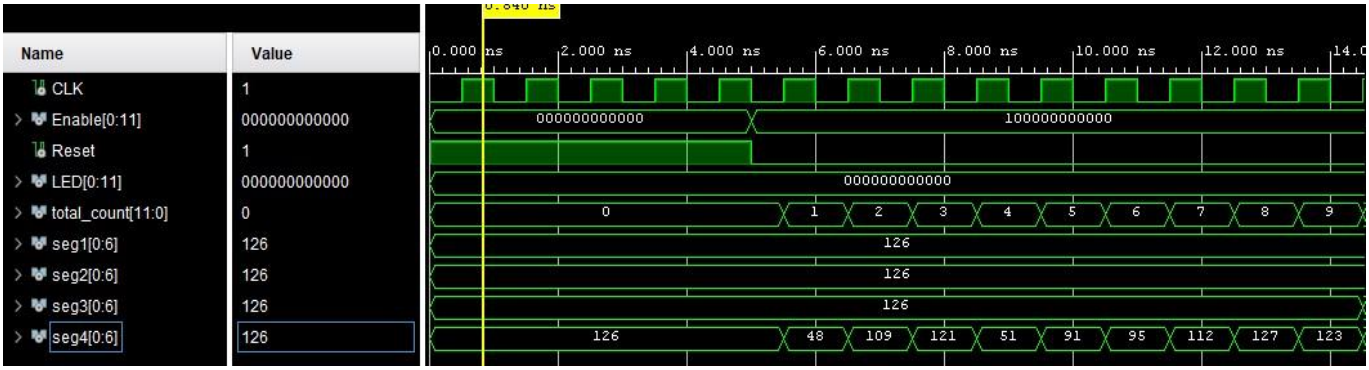

Figure 23: Simulation at start

6.2 Simulation results

Figure 23 shows the first 14ns of the simulation. It is evident that the value of "total_count" does not increase while the Reset is 1 and Enable is an array of 0's. In real life this would be when the stadium is closed to the public. After Reset was changed to '0' (i.e. when the stadium is opened to the public), people start to enter. This was represented as the 1st bit of enable being changed to '1'. Since only one Enable element in the array is high, the "total_count" can be shown to be merely incrementing by '1' on every rising clock edge as desired. This "total_count" would then be shown to the user using 7-segment displays. For the first active clock cycle after Reset at t=6ns, we can see first 3 segments are '126' and segment 4 is 48. Consolidating Table 2 shows that the number displayed by the 7-segment displays would be "0001"

which is as required. This tells the user there is one person in the stadium. Furthermore, all LED outputs are 0 for this time period because no section is 90% full yet.

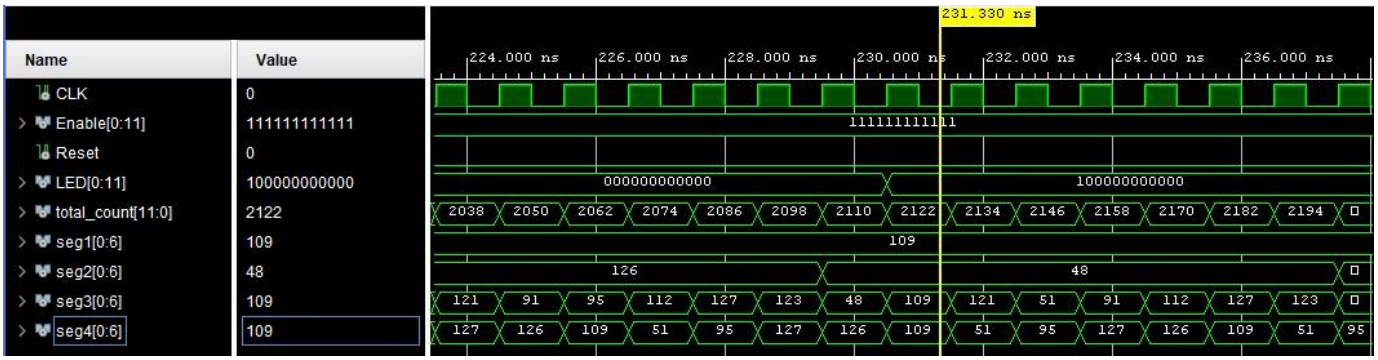

Figure 24: Simulation close to start of game

Figure 24 shows the simulation close to start of game. All Enable elements are at '1' so people are entering in every section. It can be seen that on every rising edge, "total_count" is increasing by 12 as desired. Furthermore, at the courser our count is 2122 which is shown on our segments as seg 1 is "109", seg 2 is "48", seg 3 is "109" and seg 4 is also "109". Therefore, if we look at our conventions in Table 2 this whole number translates to "2122" which is exactly the number reflected by the "total_count". For the LED output, it is evident that the first bit is high which means that counter of section A has exceeded 90% capacity. This was as expected due to the fact that from the very beginning of the simulation, the Enable of the first element is the only one to have remained active high, therefore it is logical that after 225 active clock cycles, this LED would light and be the only one turned on at that point.

Figure 25 shows the simulation during the game is being played and just prior to the game finishing. It is visible that the cursor is pointed at the point where game is being played and during this time no one is entering the stadium. Furthermore, the LED elements 0,1,2,5,6,10 & 11 are all active high. The test inputs were then consolidated, and it was realized that this was the desired results as these corresponded to the counters whose Enable had been '1' most often. Once the game has finished, Reset is turned to '1' since the stands will return to their empty state and it can be seen that the "total_count" for the stadium is 0.

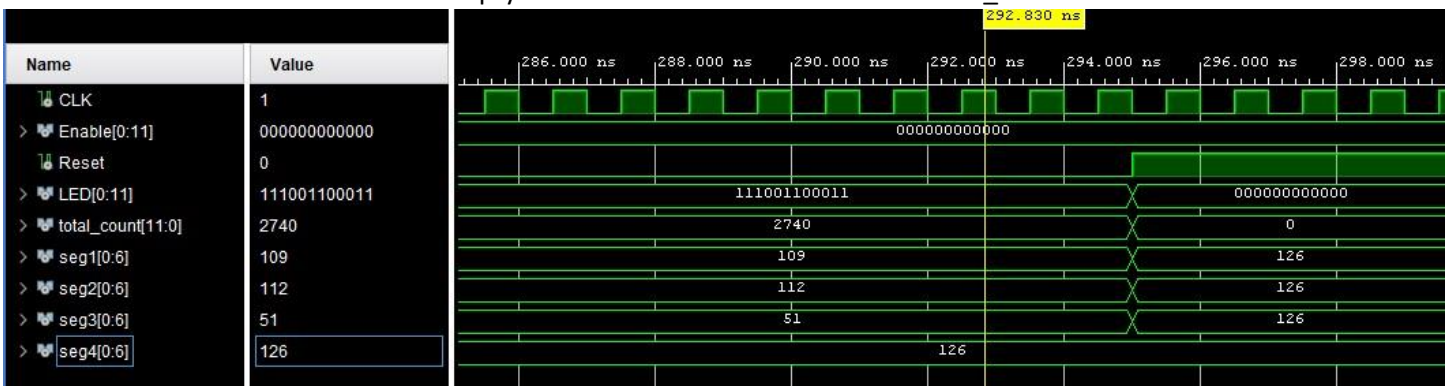

Figure 25 :Simulation in the end

It can also be seen that this Reset function is asynchronous to the clock which is as desired. This reset can also be seen on the 7-segment displays as all segments have “126” which equates to an output of ‘0’. Furthermore, all LED’s element values are returned to ‘0’.

7. FPGA Resources and floorplan

Following the design and simulation of the Stadium Counter project code, it became apparent that this design should be synthesized and implemented for a Basys-3 FPGA board. This is due to the fact that simulations on VHDL code could execute without error and produce the desired result however could fail when it comes to the synthesis stage.

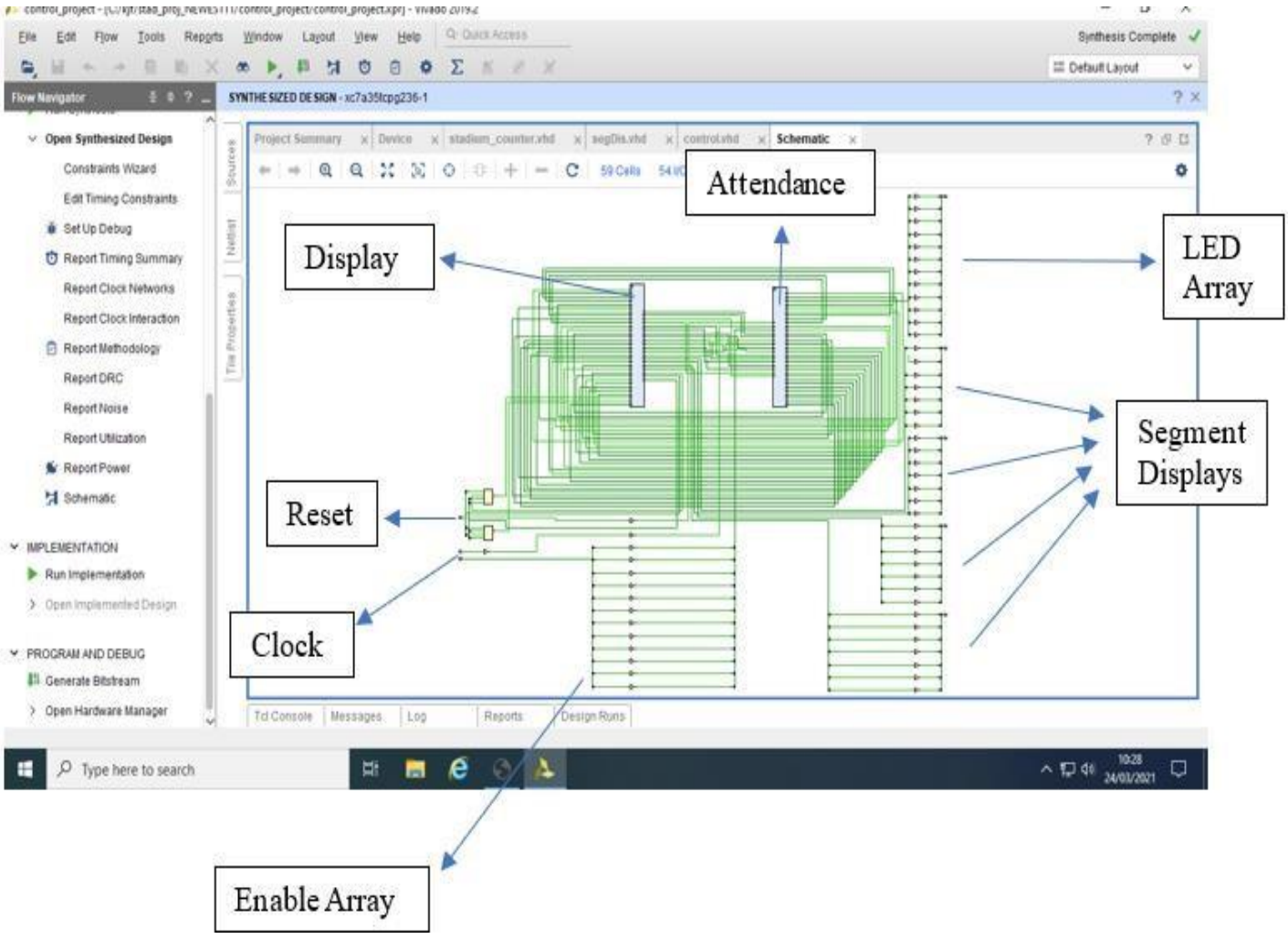

Figure 26: Synthesised Schematic

Firstly, a synthesized design was produced of the “Control.vhd” file which can be seen in Figure 26. It was determined that no errors occurred when producing this design which is as desired. Following on from this, an implementation analysis was run on the project code in order to verify that this schematic could be mapped onto a Basys-3 FPGA board. The implemented design can be seen in the FPGA floorplan in Figure 27.

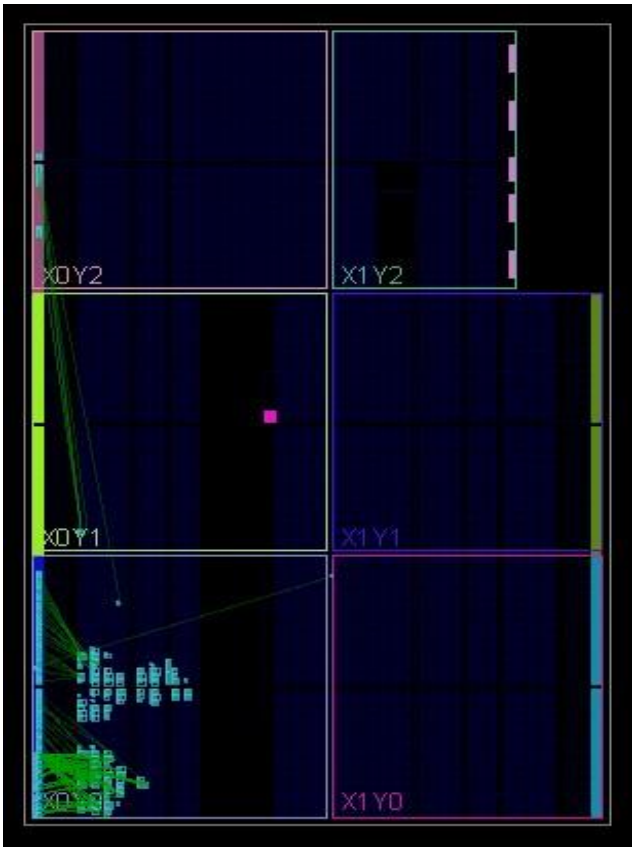

Figure 27: FPGA floorplan implementation

It can therefore be deduced that our Stadium design code could be implemented on such an FPGA, and if a suitable constraints file was placed in the Vivado project, a bitstream file could be created, which in turn could be downloaded onto a Basys-3 board in order to implement the design.

8. Review and reflection

For this work, three different design tasks had been identified and assigned to the project group members (refer to minutes in Appendix E). This proved to be a valuable decision because everyone could work at their own pace and time. To finish the work in limited time, we specifically set deadlines for crucial tasks. However, it was not possible to make a 100% fair split as further splitting design tasks would have been unrewarding. Therefore, it was very important that we regularly met for updating so that any troubleshooting problems could be solved quickly and efficiently. Furthermore, it was a bonus that

everyone in the group made it possible to spontaneously meet when bugs appeared and could not be solved by a single person. This led to very satisfying results and a good learning experience.

Due to the pandemic, we heavily relied on the VPN and remote desktop to use Vivado, which slowed the design process a little down on a few occasions. Moreover, a more extensive testing on the counter design could have saved some time on troubleshooting when connecting both the counter and the display design to the overall design. However, this is also a side effect of a not 100% fair splitting of tasks.

Overall, the design project worked very well for us, considering that the group members worked and contributed from three different times zones.

9. Discussion and Conclusion

In this work an attendance counter had been designed in VHDL to keep track of attending people in a sports venue using an FPGA. The overall counter consists of 12 8-bit counters, that refer to the attendance in 12 venue sections and were instantiated using a for-generate statement. Inside the 8-bit counter design, small adjustments had to be made to prevent value overflow. Besides counting the entering of people attending the sports game, there is an LED for each section that illuminates if the respective section exceeds 90% of the maximum seat capacity. To display the total attendance as a 4-digit decimal number, a set of four 7-segment displays is established that updates with every change in the attendance number. Therefore, the single decimal digits were obtained from the attendance number and each 7-segment element was set accordingly using a case-statement. To connect the two designs of the attendance counter and the display, an overall design was established thereby producing a hierarchy system. Using this, it was possible to test the full design implementation in order to verify that all design specifications were met. Employing reasonable test inputs, it was possible to verify that the implemented design is capable of successfully tracking and displaying the attendance for applications like events in sport venues. Moreover, it was also examined that it is possible to synthesize the overall design for the use on an FPGA board.

Further work could include expanding the counter by an additional input array to enable counting down when people leave the venue section. As mentioned previously power consumption can be further reduced when deactivating the 7-segment displays that show leading zeros. Another detail could be added by not only illuminating an LED when 90% capacity of the respective section is reached, but rather making the LED blink for a high percentage of seats sold and illuminate when 100% of seats are taken. Furthermore, in the present design if a stand is at its maximum capacity then any change in Enable for this stand would be simply ignored. However, further functionality could be implemented so that this Enable would increase another stands count. This is because in real-life, if a stand is full then the person would then be placed in another, more spacious stand.

References

[1] Digilent, "Basys 3 FPGA Board Reference Manual," 8 April 2019. [Online]. Available: https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf.

Appendix A1: Stadium counter design code

```
----------------------------------------------------------------------------------
-- Company: STRATHLYDE GROUP 1
-- Engineer: KIERAN FROST
--
-- Create Date: 15.03.2021 11:16:23
-- Design Name: STADIUM COUNTER -- Module
Name: stadium_counter - Behavioral -- Project
Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created --
Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity stadium_counter is
   Port ( CLK : in STD_LOGIC;
          Enable : in STD_LOGIC_VECTOR(0 to 11); --enable(0) corresponds to person walking in A etc;
Reset : in STD_LOGIC;
          LED : out STD_LOGIC_VECTOR(0 to 11); --LED(0) corresponds to LED for A;
total_count : out UNSIGNED(11 downto 0));--total binary count of all people end
stadium_counter;

architecture Behavioral of stadium_counter is

   component counter_8bit
      port(CLK,Reset,Enable,Load,UpDn: in std_logic;
           Data                    : in std_logic_vector(7 downto 0);
Q                    :out std_logic_vector(7 downto 0));     end
component;

   constant max_seats : integer:=250;--assigned to us in project brief
constant led_lit : integer:=225; --90% of 250
```

```
   type storer is array(0 to 11) of std_logic_vector(7 downto 0);
signal dl : storer; --container to store data from sections

begin
   --generating 12, 8 bit counters corresponding to the 12 different sections
section_counter_gen: FOR i in 0 to 11 generate

      count_elem_gen:counter_8bit                                                    port
map(CLK=>CLK,Reset=>Reset,Enable=>Enable(i),Load=>'0',UpDn=>'1',Data=>dl(i),Q=>dl(i));

   end generate;

   --dl stores 12 std_logic_vectors for the 12 different areas dl(0) corresponds to A, dl(1) corresponds to
B etc.

   stad_check:process(dl) --process to check that all are below max seats, if not set to max seats and
process executes when dl changes
      variable total_attendance: unsigned(11 downto 0);
variable temp_init :unsigned (7 downto 0);       variable
temp_resize: unsigned(11 downto 0);
      variable stad_area: storer; --needed to not cause driver error on total_count

   begin
      total_attendance:="000000000000";
temp_init:="00000000";
      temp_resize:="000000000000";

      for j in 0 to 11 loop
         if to_integer(unsigned(dl(j)))>max_seats then
            stad_area(j):="11111010"; --so number of seats cannot exceed 250 in each area
else
            stad_area(j):=dl(j); --else if less that 250 it stays the same
end if;

         temp_init := unsigned(stad_area(j)); --initialising a tempoary variable so that addition can be
performed
         temp_resize := resize(temp_init,12); --resizing as the maximum number of seats is 3000 which is a
12 bit unsinged binary number
         total_attendance:=(total_attendance+temp_resize); --this therefore adds every element together
to form a variable total_attendance       end loop;
      total_count<=total_attendance; --mapping total_count to total_attendance

   end process;
```

```
   Light:process(dl) -- process to light corresponding LED when over 90% capacity reached in each
section, occurs when dl changes
   variable light_check: storer; --needed to not cause driver error on total_coun
begin

      for j in 0 to 11 loop
         if to_integer(unsigned(dl(j)))>max_seats then
            light_check(j):="11111010"; --so number of seats cannot exceed 250 in each area
else
            light_check(j):=dl(j); --else if less that 250 it stays the same
end if;

         if to_integer(unsigned(light_check(j)))>led_lit then --if the area is greater than 225, corresponding
led lights
            LED(j)<='1';
         elsif to_integer(unsigned(light_check (j)))<led_lit then --else stays unlit
LED(j)<='0';          end if;       end loop;
  end process;

end Behavioral;
```

Appendix A2: Stadium counter Testbench

```
----------------------------------------------------------------------------------

-- Company:
-- Engineer: Kieran Frost
--
-- Create Date: 17.03.2021 16:26:47 --
Design Name:
-- Module Name: stadium_counter_TB - Behavioral --
Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created --
Additional Comments:
--
```

```
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating --
any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity stadium_counter_TB is
--  Port ( );
end stadium_counter_TB;

architecture Behavioral of stadium_counter_TB is

   component     stadium_counter     is
Port ( CLK : in STD_LOGIC;
           Enable : in STD_LOGIC_VECTOR(0 to 11); --enable(0) corresponds to person walking in A etc;
Reset : in STD_LOGIC;
           LED : out STD_LOGIC_VECTOR(0 to 11); --LED(0) corresponds to LED for A;
total_count : out UNSIGNED(11 downto 0));--total binary count of all people end component;

   signal CLK_TB,Reset_TB : std_logic;
   signal    Enable_TB,LED_TB    :std_logic_vector(0    to    11);
signal total_count_TB:unsigned(11 downto 0);


begin

   clock_gen:process
      begin
         while now<=500 ns loop
CLK_TB <= '1'; wait for 10ns;
CLK_TB <= '0'; wait for 10ns;
         end loop;
wait;
   end process;

   stimuli:process
begin
         Enable_TB<="000000000000";Reset_TB<='0';wait for 20ns; --warm up phase
```

```
        Enable_TB<="100000000000";Reset_TB<='0';wait for 20ns;--one person walking in A
        Enable_TB<="100000000000";Reset_TB<='0';wait for 20ns;--and again
        Enable_TB<="100000000000";Reset_TB<='0';wait for 20ns;--and again
        Enable_TB<="100100001000";Reset_TB<='0';wait for 20ns;
        Enable_TB<="000000000000";Reset_TB<='1';wait     for     20ns;
wait;
   end process;

   --so LED=00000000000,and total_count should be 3

   DUT:stadium_counter                                    port                                    map
(CLK=>CLK_TB,Enable=>Enable_TB,Reset=>Reset_TB,LED=>LED_TB,total_count=>total_count_TB);

end Behavioral;
```

Appendix B: 8-bit counter design code

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer: Kieran Frost
--
-- Create Date: 05.03.2021 11:33:21 --
Design Name:
-- Module Name: counter_8bit - Behavioral --
Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created --
Additional Comments:
library IEEE; use
IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
```

```
-- Uncomment the following library declaration if instantiating --
any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity counter_8bit is    Port
( CLK : in STD_LOGIC;
        Reset : in STD_LOGIC;
        Enable : in STD_LOGIC;
        Load : in STD_LOGIC;
        UpDn : in STD_LOGIC;
        Data : in STD_LOGIC_VECTOR(7 downto 0);
Q : out STD_LOGIC_VECTOR(7 downto 0)); end
counter_8bit;

architecture Behavioral of counter_8bit is

begin
   counter_func:process(CLK,Reset)
   variable REG : unsigned(7 downto 0) :="00000000";
begin
      if Reset='1' then
         REG:="00000000";  -- if reset is equal to 1 then REG gets set to 0

      elsif Reset='0' then  -- if reset is not high then this if statement is enabled
if rising_edge(CLK)then --the following steps only occur synchronously              if
Enable ='1' then  -- if enable is 1 then the clock is enable so stuff gets done

               if Load ='1' then  -- if load is high then REG becomes the data input
REG:=unsigned(Data);

               elsif Load ='0' then  -- if load is not high then this if statement is used
if UpDn = '1' then -- if updn is high then the counter should count up
REG:=REG +1;

                  elsif UpDn = '0' then -- if updn is low then the counter should count down
REG:=REG-1;

end if;
               end if;    --ending load =''

            elsif Enable = '0' then  -- if enable =0 then reg should just equal itself
REG:=REG;
```

```
            end if; --ending enable equal to something statement

         end if; -- ending the rising edge if statement

       end if; --ending the Reset='' statement
       if REG>"11111110" then –checking wont exceed 8 bit on next iteration
Reg:="11111110";
         end if;
       Q<=std_logic_vector(REG); -- assigning Q to equal the conversion of REG to std-logic-vector

    end process; --ending the process

end Behavioral;
```

Appendix C1: 7-Segment display design code

```
----------------------------------------------------------------------------------

-- Engineer: Lena Krabbe

--

-- Create Date: 16.03.2021 09:46:11

-- Design Name: 7-Segment Display

-- Module Name: segDis - Behavioral

-- Project Name: Sports Venue Attendance Counter


----------------------------------------------------------------------------------


--libraries library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.NUMERIC_STD.ALL;

-- arithmetic functions with Signed or Unsigned values


entity segDis is

   Port ( attendance : in unsigned(0 to 11);   --12 bits to represent decimal up to 3000

seg1 : out std_logic_vector(0 to 6); --representing most significant digit          seg2 :

out std_logic_vector(0 to 6);          seg3 : out std_logic_vector(0 to 6);          seg4 :

out std_logic_vector(0 to 6)); --representing least significant digit end segDis;
```

```
architecture Behavioral of segDis is

   --define segment display elements: a-b-c-d-e-f-g, active high
constant zero: std_logic_vector := "1111110";    constant one:
std_logic_vector := "0110000";
   constant two: std_logic_vector := "1101101";
constant three: std_logic_vector := "1111001";
constant four: std_logic_vector := "0110011";
constant five: std_logic_vector := "1011011";
constant six: std_logic_vector := "1011111";
constant seven: std_logic_vector := "1110000";
constant eight: std_logic_vector := "1111111";
constant nine: std_logic_vector := "1111011";

   --define subtype of integer     subtype
intSmall is integer range 0 to 3000;

begin

   show: process (attendance) is --change in attendance causes process to start

   variable att_int: intSmall;   --decimal value of attendance     variable
fac: intSmall;      --factor for modulo operation     variable noTemp:
intSmall;    --one digit of decimal attendance number     variable dif:
intSmall;      --helping variable for calculating noTemp

   variable segAll: std_logic_vector(0 to 27); --concatenation of all 4 7-segment displays

   begin
```

```
        dif    :=
0;
      fac := 1000; --initial factor modulo and division starting at most significant digit
att_int := to_integer(attendance); --convert unsigned to integer        noTemp := 0;

      --get single digits of decimal attendance number, starting at most significant digit
for i in 0 to 3 loop --4 segments

          noTemp := (att_int - dif - (att_int mod fac))/fac;
dif := dif + noTemp*fac;  --refresh dif           fac :=
fac/10;          --get factor fac for next digit

 --select correct lightning std_logic_vector (defined by constants) and assign it to correct position in
segAll          case noTemp is              when 0 =>              segAll(i*7 to ((i*7)+6)) := zero;              when 1
=>              segAll(i*7 to ((i*7)+6)) := one;              when 2 =>              segAll(i*7 to ((i*7)+6)) := two;
when 3 =>              segAll(i*7 to ((i*7)+6)) := three;              when 4 =>              segAll(i*7 to ((i*7)+6))
:= four;              when 5 =>              segAll(i*7 to ((i*7)+6)) := five;              when 6 =>
segAll(i*7 to ((i*7)+6)) := six;              when 7 =>              segAll(i*7 to ((i*7)+6)) := seven;
when 8 =>              segAll(i*7 to ((i*7)+6)) := eight;              when others =>
            segAll(i*7 to i*7+6) := nine;
end case;       end loop;

      --divide concatenated segAll into 4 7-segment displays (vectors)
seg1 <= segAll(0 to 6);   --representing most significant digit
seg2 <= segAll(7 to 13);       seg3 <= segAll(14 to 20);       seg4 <=
segAll(21 to 27); --representing least significant digit     end process;


end Behavioral;
```

Appendix C2: 7-Segment display testbench

```
----------------------------------------------------------------------------------
-- Engineer: Lena Krabbe
--
-- Create Date: 17.03.2021 10:25:32
-- Design Name: 7-segment Display
-- Module Name: segDis_TB - Behavioral
-- Project Name: Sports Venue Attendance Counter
----------------------------------------------------------------------------------

--libraries library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.NUMERIC_STD.ALL;
-- arithmetic functions with Signed or Unsigned values

entity segDis_TB is
--  Port ( );
end segDis_TB;

architecture Behavioral of segDis_TB is

   --declare component
component segDis
      Port ( attendance : in unsigned(0 to 11); --12 bits to represent decimal up to 3000
seg1, seg2, seg3, seg4 : out std_logic_vector(0 to 6));
   end component;

   --create signals to simulate DUT
   signal attendance: unsigned(0 to 11);        signal
seg1,seg2,seg3,seg4: std_logic_vector(0 to 6);
```

```
begin
   --instantiate component
   DUT: segDis port map (attendance => attendance, seg1 => seg1, seg2 => seg2, seg3 => seg3, seg4 =>
seg4);

   --stimulation process: try different number, especially 0 and 3000 (min/max) and all possible digit
values (0 to 9)    stimuli: process is    begin       attendance <= to_unsigned(52,12); wait for 10ns;
attendance <= to_unsigned(102,12); wait for 10ns;       attendance <= to_unsigned(1000,12); wait
for 10ns;       attendance <= to_unsigned(2547,12); wait for 10ns;       attendance <=
to_unsigned(0,12); wait for 10ns;       attendance <= to_unsigned(387,12); wait for 10ns;
attendance <= to_unsigned(1648,12); wait for 10ns;       attendance <= to_unsigned(5,12); wait for
10ns;       attendance <= to_unsigned(3000,12); wait for 10ns;       attendance <=
to_unsigned(2946,12); wait for 10ns;       attendance <= to_unsigned(764,12); wait for 10ns;
attendance <= to_unsigned(976,12); wait for 10ns;
   wait;
   end process;


end Behavioral;
```

Appendix D: Control design code

```
-- Engineer: Muhammad Taha Shahid
--
-- Create Date: 15.03.2021  --
Design Name:
-- Module Name: counter_8bit - Behavioral --
Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
```

```
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE; use
IEEE.STD_LOGIC_1164.ALL; use
IEEE.NUMERIC_STD.ALL;

entity Control is
    Port ( CLK : in STD_LOGIC;-- Clock input for the counter
           Enable : in STD_LOGIC_VECTOR(0 TO 11); -- Enable array for 12 sagments of stadium
           Reset : in STD_LOGIC;-- Reset to set values back to zero after match finishes           LED
: out STD_LOGIC_VECTOR(0 TO 11);-- LED's that shows capacity is more than 90%
total_count: inout unsigned(11 downto 0); -- signal to connect from counter to display
seg1 : out std_logic_vector(0 to 6);--1st display          seg2 : out std_logic_vector(0 to 6);--
2nd display          seg3 : out std_logic_vector(0 to 6);--3rd display          seg4 : out
std_logic_vector(0 to 6));--4th display
end Control;

architecture Behavioral of Control is

component stadium_counter is
    Port ( CLK : in STD_LOGIC;
           Enable : in STD_LOGIC_VECTOR(0 to 11);
           Reset : in STD_LOGIC;          LED : out
STD_LOGIC_VECTOR(0 to 11);          total_count :
out UNSIGNED(11 downto 0));
end component;

component segDis is
```

```
   Port ( attendance : in unsigned(0 to 11);
seg1 : out std_logic_vector(0 to 6);          seg2
: out std_logic_vector(0 to 6);          seg3 : out
std_logic_vector(0 to 6);          seg4 : out
std_logic_vector(0 to 6));
end component;



begin
--creathing hierarchy for counter
Attendance: stadium_counter port
map(
      CLK=>CLK,
      Enable=>Enable,
      Reset=>Reset,
LED=>LED,
total_count=>total_count);
--creathing hierarchy for counter
 Display : segDis  port map(
attendance=>total_count,
seg1=>seg1,       seg2=>seg2,
seg3=>seg3,       seg4=>seg4);



end Behavioral;
```

Appendix D2: Control design Testbench

```
 -- Engineer: Muhammad Taha Shahid
--
-- Create Date: 15.03.2021  --
Design Name:
```

```
-- Module Name: counter_8bit - Behavioral --
Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Additional Comments:
--
----------------------------------------------------------------------------------


LIBRARY ieee;

LIBRARY generics;

USE ieee.std_logic_1164.ALL;

USE ieee.numeric_std.ALL;

ENTITY testbench IS

END testbench;


ARCHITECTURE behavior OF testbench IS


        COMPONENT Control

        PORT(

                CLK : IN std_logic;

                Enable : IN std_logic_vector(0 to 11);
Reset : IN std_logic;                total_count : INOUT
unsigned(11 downto 0);                LED : OUT
std_logic_vector(0 to 11);            seg1 : OUT
std_logic_vector(0 to 6);             seg2 : OUT
std_logic_vector(0 to 6);             seg3 : OUT
```

```
std_logic_vector(0 to 6);                    seg4 : OUT
std_logic_vector(0 to 6)
                    );
        END COMPONENT;

        SIGNAL CLK :  std_logic;
        SIGNAL Enable :  std_logic_vector(0 to 11);
        SIGNAL Reset :  std_logic;
        SIGNAL LED :  std_logic_vector(0 to 11);
        SIGNAL total_count :  unsigned(11 downto 0);
        SIGNAL seg1 :  std_logic_vector(0 to 6);
        SIGNAL seg2 :  std_logic_vector(0 to 6);
        SIGNAL seg3 :  std_logic_vector(0 to 6);
SIGNAL seg4 :  std_logic_vector(0 to 6); BEGIN

        uut: Control PORT MAP(
                CLK => CLK,
                Enable => Enable,
                Reset    =>    Reset,
        LED => LED,
                total_count => total_count,
                seg1 => seg1,
        seg2 => seg2,
seg3 => seg3,               seg4 =>
seg4
        );

 Clock: PROCESS
```

```
   BEGIN     CLK<='0';
for i in 1 to 1000 loop
wait for 0.5ns;
      CLK<= not CLK;
END loop;     wait;
    END PROCESS;

  tb : PROCESS
BEGIN
  -- Match scale 1ns=1m
  Enable<= "000000000000";
  Reset<= '1';
  --Setting starting values to zero
wait for 5ns;
  Enable<= "100000000000";   Reset<= '0';
wait for 10ns; -- counter before 90m of game
Enable<= "110000000000";   wait for 10ns; --
counter before 80m of game    Enable<=
"101000000000";   wait for 10ns; -- counter
before 70m of game    Enable<=
"100010000000";   wait for 10ns;-- counter
before 60m of game    Enable<=
"100001100010";   wait for 10ns;-- counter
before 50m of game    Enable<=
"100001100011";   wait for 10ns;-- counter
before 40m of game    Enable<=
"111001111111";   wait for 10ns;-- counter
before 30m of game    Enable<=
"111101111111";   wait for 10ns;-- counter
```

```
before 20m of game    Enable<=
"111111111111";    wait for 200ns;-- counter
before 10m of game    Enable<=
"000000000000";    wait for 10ns;-- counter
during the game
  Reset<= '1';-- reseting to zero after match ends // counter after the  game
  wait;
  END PROCESS;


END;
```

Appendix E: Minutes

10.03.2021: Kick-Off Meeting (3:30pm-5pm)

Attendees: Taha, Kieran, Lena

Divide design project into three tasks:

1. Counter design: Kieran
2. 7-segment display: Lena
3. Testbench design: Taha

Details on counter design:

- Number of seats per section as constant (250)
- Input: enable array, clock, reset
- Output: overall total attendance, LED array to indicate >= 90% in each section
- 90% = 225 people – calculation

Deadlines for this project:

- Technical design until 20.03.
- Demo 26.03.
- Report Submission 29.03.

17.03.2021: Update Meeting (3pm-4:30pm)

Attendees: Lena, Taha, Kieran

Review Counter Design:

- Explained how the structural VHDL code works
- Changes: counter output is type unsigned (before std_logic_vector)

Review Segment Display:

- Explain how single digits of counter output (unsigned -> integer) are obtained
- Explain case statement
- Explain how functionality of segment displays can be tested: testbench
- Explain process sensitivity list

Decisions:

- connect both components via higher level design -> Introduce hierarchy (Taha)
- Platform for report and minutes writing: Word on OneDrive

Next steps:

- Design high level design & write the testbench to check overall functionality (Taha)
- Write down central design choices and implementation for counter design (Kieran) and 7segment display design (Lena)
- All: Brainstorm for report structure until next meeting

20.03. Troubleshooting Counter Design (11:45am – 12:30am)

Attendees: Lena, Kieran

Fixed bugs in counter design code

- two signals driving same output
- Variable vs signal assignment

Verification via counter testbench

21.03. Update Meeting (11:30am – 1:30pm)

Attendees: Taha, Kieran, Lena

Review Testbench and overall design:

- in/out signal “total_count” could also be an internal signal
- put a reset in the beginning
- include new counter design version from 20.03.
- Account for real time scenario: 1ns = 1min, clock cycle of 5min (= 5ns), overall game time 90min = 18 clock cycles ➡ clock scaling

Demo Slides: Screenshots of different code parts (from inside Vivado or with highlighting text editor like Notepad) with explanations until next meeting via shared PowerPoint. Lena will share PowerPoint during Demo (2 screens).

Report Writing: Split up tasks

- Kieran: Comment on FPGA resources & floorplan
- Lena: Discussion and Conclusion, Review of design problem & specification
- Taha: Block diagram/hierarchy/components, justification of test inputs, discussion of simulation results
- All: Review on implemented design and individual testbench
- Next meeting: abstract; review and references

Display design: Maybe incorporate subtype of integer

All: comment code and make namings meaningful until Tuesday (23.03.) (sent this version to Taha)

Discussion about References:

- Slides of VHDL
- Website referring to signal/variable assignments
- Reference Manual for 7-Segment Display

24.03. Last Meeting before Demo (4:00pm-4:45pm)

Attendees: Lena, Kieran, Taha

Prepare for demo:

- Slides: 3 slides per person
- Show code and execute simulation

Lena writes email to Louise to clarify what is wanted for the demo. -> Answer: only demo, no slides!

Write individual parts of the report until Saturday, meet on Sunday (11am UK time) for finalizing report (especially the reflection part and references).

25.03. Troubleshooting Meeting (7:15pm-8:30)

Attendees: Lena, Kieran, Taha

Fixed overflow in 8-bit counter: insert if-statement to prevent overflowing when counter number get bigger than 255

26.03. After Demo Talk

Attendees: Lena, Kieran, Taha

Show individual section counter in waveform to better check functionality (Kieran)

Adjust clock scaling to make it more realistic (Taha)

28.03. Final Report Meeting (11:00am-12:30pm)

Attendees: Lena, Kieran, Taha

Agree on some aspects:

- Put hierarchy figure and first hierarchy text before counter design
- Make code snippets as screenshots to have good highlighting and with a box
- Block diagrams design: decide on one design for block diagrams (Lena sent PDF template)
- Appendix: as text, everyone puts in his/her own code + minutes
- add page numbers and chapter numbers

ToDo:

- Kieran: put counter discussion into counter section, write something short about the individual counter visualization, final upload of report pdf, add a counter block diagram
- Taha review numbering of figures, Abstract writing, Recreate block diagram for Control
- Lena: Review & reflection, Formatting on minutes

Make all content ready until 28.03. midnight, afterwards: formatting

Short meeting on Monday at 9am to look through final version

29.03. Final Overview Meeting (9:00am-10:00am)

Attendees: Lena, Kieran, Taha

Having final view on report

- Fixed some problems with double quotes
- Rearranged some figures
- Rephrased few sentences