

PSDE5029 Assignment 1

by Joseff Jones

FILE	2874_JOSEFF_JONES_PSDE5029_ASSIGNMENT_1_155114_1907082508.DOCX (11.75M)		
TIME SUBMITTED	18-JAN-2018 04:23AM (UTC+0000)	WORD COUNT	6610
SUBMISSION ID	81980312	CHARACTER COUNT	36333

HND Electrical and Electronic Engineering

PSDE5029 – Virtual Instrumentation and
Control

Assignment 1 – Binary/BCD Conversion in VHDL

Joseff Jones

Date of Submission: 18/1/18

Word count: 6642

Contents

PSDE5029 – Virtual Instrumentation and Control Assignment 1 – Binary/BCD Conversion in VHDL	1
1. Introduction	4
2. Research.....	5
3. Design of Assignment Requirements	6
4. Analysis of Xilinx Vivado Simulated Results	13
5. Analysis of Nexys4 FPGA Practical Results	18
6. Adapting Design to Suit New Purposes or Applications	20
7. Analysis of Xilinx Vivado Simulated Results	33
8. Analysis of Nexys 4 DDR FPGA Practical Results	38
9. Conclusions	40
10. Bibliography	41
References	41

Table of Figures

Figure 1 - 6-bit Binary-to-BCD Converter Design Source (Part 1/4).....	6
Figure 2 - 6-bit Binary-to-BCD Converter Design Source (Part 2/4).....	7
Figure 3 - 6-bit Binary-to-BCD Converter Design Source (Part 3/4).....	8
Figure 4 - 6-bit Binary-to-BCD Converter Design Source (Part 4/4).....	8
Figure 5 - 6-bit Binary-to-BCD Converter TB File (1/3)	9
Figure 6 - 6-bit Binary-to-BCD Converter TB File (2/3)	10
Figure 7 - 6-bit Binary-to-BCD Converter TB File (3/3)	11
Figure 8 - 8-bit Binary-to-BCD Schematic	12
Figure 9 - 8-bit Binary-to-BCD Converter Constraints.....	12
Figure 10 - Function Table	13
Figure 11 - Simulation of 6-bit Binary-to-BCD - 26/27	14
Figure 12 - Simulation of 6-bit Binary-to-BCD 44/45	14
Figure 13 - Simulation of 6-bit Binary-to-BCD 60/61	15
Figure 14 - Simulation of 8-bit Binary-to-BCD - 234	16
Figure 15 - Simulation of 8-bit Binary-to-BCD - 135	16
Figure 16 - Simulation of 8-bit Binary-to-BCD - 67.....	17
Figure 17 - 234 in Binary and BCD.....	18
Figure 18 - 135 in Binary and BCD.....	19
Figure 19 - 67 in Binary and BCD.....	19
Figure 20 - Multiplexer and Demultiplexer Design Source (1/3)	20
Figure 21 - Multiplexer and Demultiplexer Design Source (2/3)	21
Figure 22 - Multiplexer and Demultiplexer Design Source (3/3)	21
Figure 23 - Multiplexer and Demultiplexer Test Bench (1/3)	22
Figure 24 - Multiplexer and Demultiplexer Test Bench (2/3)	23
Figure 25 - Multiplexer and Demultiplexer Test Bench (3/3)	24
Figure 26 - 5KHz Clock Design Source	25
Figure 27 - 5KHz Clock Test Bench File (1/2).....	26
Figure 28 - 5KHz Clock Test Bench File (2/2).....	26

Figure 29 - BCD to 7-Segment Converter Design Source (1/2)	27
Figure 30 - BCD to 7-Segment Converter Design Source (2/2)	28
Figure 31 - BCD to 7-Segment Converter Test Bench File (1/4)	28
Figure 32 - BCD to 7-Segment Converter Test Bench File (2/4)	29
Figure 33 - BCD to 7-Segment Converter Test Bench File (3/4)	29
Figure 34 - BCD to 7-Segment Converter Test Bench File (4/4)	30
Figure 35 - BCD to 7-Segment Converter Constraints	30
Figure 36 - 8-bit Binary-to-BCD with 3*7-Segment LED Display Design	31
Figure 37 - Final Design Constraints	31
Figure 38 - 3*4-bit Mux_Demux Output Code	33
Figure 39 - 3*4-bit Mux_Demux Output - Units	33
Figure 40 - 3*4-bit Mux_Demux Output - Tens	34
Figure 41 - 3*4-bit Mux_Demux Output - Hundreds	34
Figure 42 - 5KHz Clock Simulation	35
Figure 43 - 5KHz Clock Simulation - 200ms Difference	35
Figure 44 - BCD to 7-Segment Simulation - 6	36
Figure 45 - 7-Segment for the value 6	36
Figure 46 - BCD to 7-Segment Simulation - 9	37
Figure 47 - 7-Segment for the value 9	37
Figure 48 - BCD to 7-Segment Simulation - 10	37
Figure 49 - 234 on 7-Segment	38
Figure 50 - 135 on 7-Segment	38
Figure 51 - 67 on 7-Segment	39
Figure 52 - 255 on 7-Segment	39

1. Introduction

This assignment involved the design of a VHDL file, which would be used to emulate a 6-bit binary-to-BCD converter (specifically a 74185A). This VHDL file would then be adapted and amalgamated with other VHDL files to eventually create a final VHDL file which could be used to output a range of results from 0 to 255 to three seven-segment displays based on the input from an 8-bit range. This final VHDL file could also be uploaded to a Nexys 4 DDR FPGA board in order to function as a physical 8-bit binary-to-BCD converter, which also allows testing to be carried out. This report covers the process of design and adaptation for every aspect of the assignment and also contains independent sections dedicated to research, analysis and conclusions.

Brief, concise introduction given...

2. Research

Prior to carrying out the tasks of the assignment, research was carried out in order to gain an understanding of what the assignment might entail – research into the structure of VHDL files, binary to BCD conversion, and IP block design was carried out.

Structure of VHDL Files

According to Cohen, “VHDL is a hardware description language with a vocabulary rich enough to span a very wide range of design descriptions” (1995, p. 22). A VHDL file is made up of two primary bodies – the entity and the architecture. The entity is where components of the VHDL are declared and is where it is determined whether they are inputs or outputs. The architecture can be described as a description of the entity, where the components are assigned functions and where those functions are assigned limitations and conditionals such as if statements and case statements. The behaviour of the VHDL can be written and re-written using the architecture of files – outputs and inputs can have their names changed, or only be inputs/outputs at instantaneous times based on internal clocks.

Not enough Detail

Binary to BCD Conversion

Binary to BCD conversion is a straight forward concept and simply requires understanding how each notation functions. With binary, a number of “bits” are used to represent numbers, where each additional bit is double the size of the bit before it – an example of this is where $111 = 4+2+1$, but 1111 would equal $8+4+2+1$. With BCD, 4-bit sequences are used to represent units, tens, hundreds etc. and the sequence can just be scaled by further adding 4-bit sequences. A BCD value of 67 would look like 01100111 where the four left-most bits represent the tens and the the four right-most bits represent the units. BCD can be advantageous over basic binary because it is easily scalable and is very easy to be deciphered by humans, although binary is typically easier for machines to work with, as most are programmed with binary in mind.

Not enough detail

IP Block Design

IP block design is a function of Vivado which aids with the design of intricate circuits by turning them into much more comprehensible and concise circuits. They are developed using a number of IP blocks which are VHDL files that have had their entities and architectures established and have then been “packaged”, which essentially means they have been turned into templates to be used for IP block design or other uses within Vivado. IP blocks typically come in the form of a simple block with the entity’s inputs on the left side and the entity’s outputs on the right side, allowing a number of IP blocks to be connected with ease - resulting in simple circuit design for what would be otherwise quite complex.

An example would have been

useful at this point

Very little research carried out with only one citation given.... very poor!

3. Design of Assignment Requirements

Designing a fully functional 3*7-segment LED display which outputs a BCD value determined by switch inputs involved combining a number of smaller components, such as:

- A 6-bit Binary-to-BCD Converter
- A 3*4-bit Mux_Demux (Multiplexer and Demultiplexer)
- A 5KHz clock (based on the 100MHz clock provided by the Nexys 4 DDR FPGA)
- BCD to 7-segment Converter

These components were all developed using Vivado software, which allows the development of virtual circuits and devices down to an architectural level. These virtual circuits and devices can then either be used for simulation or uploaded to FPGA boards to be used for practical applications; they can also be saved and stored as templates to be used in any number of other designs within Vivado.

6-bit Binary-to-BCD Converter

The first component to be designed was the 6-bit binary-to-BCD converter. This served the purpose of behaving as a test binary-to-BCD converter for developing an understanding of the operation of the device and also to be stored as a template (known as packaging an IP) to be used in other VHDL files, for example in the 8-bit binary-to-BCD converter. The first step in designing the 6-bit binary-to-BCD converter was to create a project on the Vivado software, this project is used to contain the data written by the user concerning file directory. The next step was to create a VHDL design source – this file contains the entity and architecture of the 6-bit binary-to-BCD converter. The entity and architecture of this particular file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Assignment_1_BCD is
4  Port ( A,B,C,D,E : in STD_LOGIC;
5        Y1,Y2,Y3,Y4,Y5,Y6 : out STD_LOGIC);
6  end Assignment_1_BCD;
7  architecture Behavioral of Assignment_1_BCD is
8  signal Inp : std_logic_vector (4 downto 0);
9  signal Q : std_logic_vector (5 downto 0);
10 begin
11  Inp(0) <= A;
12  Inp(1) <= B;
13  Inp(2) <= C;
14  Inp(3) <= D;
15  Inp(4) <= E;
16  process (Inp)
17  begin
18  case Inp is
19  WHEN "00000" => -- 0/1
20  Q <= ("000000");
21  WHEN "00001" => -- 2/3
22  Q <= ("000001");
23  WHEN "00010" => -- 4/5
24  Q <= ("000010");
25  WHEN "00011" => -- 6/7
26  Q <= ("000011");
27  WHEN "00100" => -- 8/9
28  Q <= ("000100");
29  WHEN "00101" => -- 10/11
30  Q <= ("001000");

```

Figure 1 - 6-bit Binary-to-BCD Converter Design Source (Part 1/4)

The first 30 lines of the code contain the entity and part of the architecture. The entity states the inputs and outputs of the file, and in the case of the 6-bit binary-to-BCD converter, the entity has five inputs (A, B, C, D, E) and six outputs (Y1, Y2, Y3, Y4, Y5, Y6). The architecture contains the code that determines what the output is based on the input, this is achieved using case statements. A specific example of this is line 19 & 20, where the output (or Q) is set to 000000 when the input is 00000.

Good detail

```

31 WHEN "00110" => -- 12/13
32 Q <= ("001001");
33 WHEN "00111" => -- 14/15
34 Q <= ("001010");
35 WHEN "01000" => -- 16/17
36 Q <= ("001011");
37 WHEN "01001" => -- 18/19
38 Q <= ("001100");
39 WHEN "01010" => -- 20/21
40 Q <= ("010000");
41 WHEN "01011" => -- 22/23
42 Q <= ("010001");
43 WHEN "01100" => -- 24/25
44 Q <= ("010010");
45 WHEN "01101" => -- 26/27
46 Q <= ("010011");
47 WHEN "01110" => -- 28/29
48 Q <= ("010100");
49 WHEN "01111" => -- 30/31
50 Q <= ("011000");
51 WHEN "10000" => -- 32/33
52 Q <= ("011001");
53 WHEN "10001" => -- 34/35
54 Q <= ("011010");
55 WHEN "10010" => -- 36/37
56 Q <= ("011011");
57 WHEN "10011" => -- 38/39
58 Q <= ("011100");
59 WHEN "10100" => -- 40/41
60 Q <= ("100001");

```

Figure 2 - 6-bit Binary-to-BCD Converter Design Source (Part 2/4)

The next 30 lines carry on with the architecture and the case statements, further determining what the outputs are based on the inputs.

```

60 Q <= ("100001");
61 WHEN "10101" => -- 42/43
62 Q <= ("100001");
63 WHEN "10110" => -- 44/45
64 Q <= ("100010");
65 WHEN "10111" => -- 46/47
66 Q <= ("100011");
67 WHEN "11000" => -- 48/49
68 Q <= ("100100");
69 WHEN "11001" => -- 50/51
70 Q <= ("101000");
71 WHEN "11010" => -- 52/53
72 Q <= ("101001");
73 WHEN "11011" => -- 54/55
74 Q <= ("101010");
75 WHEN "11100" => -- 56/57
76 Q <= ("101011");
77 WHEN "11101" => -- 58/59
78 Q <= ("101100");
79 WHEN "11110" => -- 60/61
80 Q <= ("110000");
81 WHEN "11111" => -- 62/63
82 Q <= ("110001");
83 When Others =>
84 Q <= ("000000");
85 END CASE;
86 END PROCESS;

```

Figure 3 - 6-bit Binary-to-BCD Converter Design Source (Part 3/4)

The next 26 lines carry on with the case statements, but an important output that's included here is that when the input is "others", meaning anything that isn't defined as input in the rest of the program, the output will be 000000.

```

87 :
88 :      Y1 <= Q(0);
89 :      Y2 <= Q(1);
90 :      Y3 <= Q(2);
91 :      Y4 <= Q(3);
92 :      Y5 <= Q(4);
93 :      Y6 <= Q(5);
94 :
95 end Behavioral;

```

Figure 4 - 6-bit Binary-to-BCD Converter Design Source (Part 4/4)

The final part of the design source is stating the names of the outputs, where Q(0) is defined as Y1, Q(1) is defined as Y2 etc.

Good detail

The next part in the design process was to add a simulation source, or “test bench” file. This test bench file serves the purpose of copying the information from the VHDL design source and determining how it is simulated. Simulation is an important aspect of the design process as it helps determine whether the VHDL program is behaving correctly. The test bench file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Assignment_1_tb is
4  -- Port ( );
5  end Assignment_1_tb;
6  architecture Behavioral of Assignment_1_tb is
7  -- Component Declaration for the Unit Under Test (UUT)
8  COMPONENT Assignment_1_BCD
9  Port ( A,B,C,D,E : in std_logic;
10 Y1,Y2,Y3,Y4,Y5,Y6 : out std_logic
11 );
12 END COMPONENT;
13
14 --Inputs
15 signal A : std_logic := '0';
16 signal B : std_logic := '0';
17 signal C : std_logic := '0';
18 signal D : std_logic := '0';
19 signal E : std_logic := '0';
20 --Outputs
21 signal Y1 : std_logic;
22 signal Y2 : std_logic;
23 signal Y3 : std_logic;
24 signal Y4 : std_logic;
25 signal Y5 : std_logic;
26 signal Y6 : std_logic;
27 --Clocks
28 constant A_period : time := 1 us;
29 constant B_period : time := 2 us;
30 constant C_period : time := 4 us;

```

Figure 5 - 6-bit Binary-to-BCD Converter TB File (1/3)

Alike the design source file, the TB file contains an entity and an architecture, but the entity does not play that much of a role as the inputs and outputs are copied from the VHDL file into the architecture. Another part in the architecture of the TB file for the 6-bit binary-to-BCD is where clocks are defined – clocks are defined in constant periods of 1 us, 2 us, 4 us etc. for the signals to produce all possible outputs as they will not be overlapping all of the time.

Good Detail

```

30 constant C_period : time := 4 us;
31 constant D_period : time := 8 us;
32 constant E_period : time := 16 us;
33 BEGIN
34 -- Instantiate the Unit Under Test (UUT)
35 uut: Assignment_1_BCD PORT MAP (
36   A => A,
37   B => B,
38   C => C,
39   D => D,
40   E => E,
41   Y1 => Y1,
42   Y2 => Y2,
43   Y3 => Y3,
44   Y4 => Y4,
45   Y5 => Y5,
46   Y6 => Y6
47 );
48 -- Clock process definitions
49 A_process :process
50 begin
51   A <= '0';
52   wait for A_period/2;
53   A <= '1';
54   wait for A_period/2;
55 end process;
56 B_process :process

```

Figure 6 - 6-bit Binary-to-BCD Converter TB File (2/3)

The next 27 lines of the TB file include instantiating the UUT and definitions of the clock process. Instantiating the UUT is there to map the inputs and outputs for the simulation to display information correctly to the reader and the clock process definitions are there to set the periods for each input.

What does UUT mean?

```

58   begin
59   B <= '0';
60   wait for B_period/2;
61   B <= '1';
62   wait for B_period/2;
63   end process;
64   C_process :process
65   begin
66   C <= '0';
67   wait for C_period/2;
68   C <= '1';
69   wait for C_period/2;
70   end process;
71   D_process :process
72   begin
73   D <= '0';
74   wait for D_period/2;
75   D <= '1';
76   wait for D_period/2;
77   end process;
78   E_process :process
79   begin
80   E <= '0';
81   wait for E_period/2;
82   E <= '1';
83   wait for E_period/2;
84   end process;
85   END;

```

Figure 7 - 6-bit Binary-to-BCD Converter TB File (3/3)

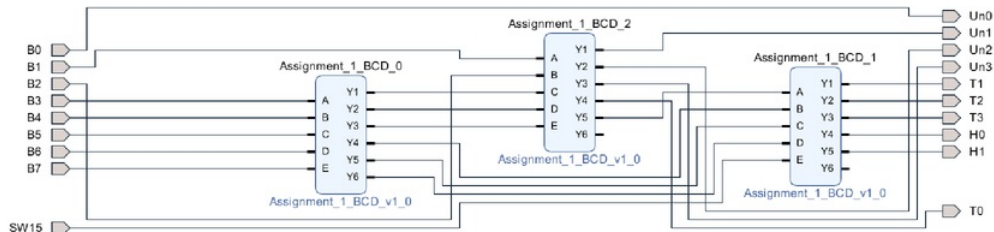
The final part of the TB file for the 6-bit binary-to-BCD converter is a continuation of the clock process definitions.

Not explained fully

Once simulating has been completed, normally, the next step is to add a constraints file in preparation for uploading the VHDL file to the FPGA board for practical testing. In the case of the 6-bit binary-to-BCD file, the constraints file was not needed because the 6-bit binary-to-BCD program wasn't uploaded to the FPGA, as the 6-bit isn't needed for the final design; instead, the 6-bit binary-to-BCD file was turned into an IP block to be used for the 8-bit binary-to-BCD file, which was uploaded to the board. *Good*

8-bit Binary-to-BCD Converter

Creating the 8-bit binary-to-BCD converter involved taking three 6-bits and connecting them together, the figure below shows how it was all connected.



What about the setup procedure?

Figure 8 - 8-bit Binary-to-BCD Schematic

SW15 was connected to ground the circuit in order for the board to show the correct values.

Because the 8-bit binary-to-BCD converter only involved connecting three IP blocks of the 6-bit binary-to-BCD converters to buttons and outputs, there was no need to write anything into the design source or the test bench file as it was already all there; there was however a constraints file that needed to be developed in order to upload the VHDL file to the FPGA. The constraints file that was developed is shown in the figure below.

```

1  ##Switches
2
3  set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports { B0 }]; #IO_L24N_T3_RSO_15 Sch=sw[0]
4  set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 } [get_ports { B1 }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
5  set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 } [get_ports { B2 }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
6  set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 } [get_ports { B3 }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
7  set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 } [get_ports { B4 }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
8  set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 } [get_ports { B5 }]; #IO_L7N_T1_D10_14 Sch=sw[5]
9  set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 } [get_ports { B6 }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
10 set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 } [get_ports { B7 }]; #IO_L5N_T0_D07_14 Sch=sw[7]
11 set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 } [get_ports { SW15 }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
12
13 ##LED's
14
15 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 } [get_ports { Un0 }]; #IO_L18P_T2_A24_15 Sch=led[0]
16 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 } [get_ports { Un1 }]; #IO_L24P_T3_RS1_15 Sch=led[1]
17 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 } [get_ports { Un2 }]; #IO_L17N_T2_A25_15 Sch=led[2]
18 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 } [get_ports { Un3 }]; #IO_L8P_T1_D11_14 Sch=led[3]
19 set_property -dict { PACKAGE_PIN R18    IOSTANDARD LVCMOS33 } [get_ports { T0 }]; #IO_L7P_T1_D09_14 Sch=led[4]
20 set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 } [get_ports { T1 }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
21 set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 } [get_ports { T2 }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
22 set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 } [get_ports { T3 }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
23 set_property -dict { PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [get_ports { H0 }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
24 set_property -dict { PACKAGE_PIN T15    IOSTANDARD LVCMOS33 } [get_ports { H1 }]; #IO_L14N_T2_SRCC_14 Sch=led[9]

```

Figure 9 - 8-bit Binary-to-BCD Converter Constraints

The constraints file is there to define the ports for the FPGA to understand the VHDL file's coding for the inputs and outputs. The inputs in this case are the buttons (there's also a SW15 which is zero, to ensure the board works correctly) and the outputs are the units, tens and hundreds which are displayed on the FPGA as LEDs.

Very little explanation on how the IP blocks were obtained

4. Analysis of Xilinx Vivado Simulated Results

In order to ensure that the developed VHDL files actually worked, simulations were carried out from within the Vivado software, which allowed analyses of the written code; those simulations are shown below.

6-bit Binary-to-BCD Converter

The first simulation to be generated was of the 6-bit binary-to-BCD converter; this analysis involved making use of a function table to determine the BCD output based on a binary input. The function table that was used to determine the output is shown in the figure below.

Function Tables															
Binary Words		Inputs					Outputs								
		Binary Select				Enable G	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	
E	D	C	B	A											
0	1	L	L	L	L	L	H	H	L	L	L	L	L	L	L
2	3	L	L	L	L	H	H	H	L	L	L	L	L	L	H
4	5	L	L	L	H	L	H	H	L	L	L	L	H	L	L
6	7	L	L	L	H	H	H	H	L	L	L	L	H	H	H
8	9	L	L	H	L	L	L	H	L	L	L	H	L	L	L
10	11	L	L	H	L	H	L	H	L	L	H	L	L	L	L
12	13	L	L	H	H	L	L	H	L	L	H	L	L	H	H
14	15	L	L	H	H	H	L	H	L	L	H	L	H	L	L
16	17	L	H	L	L	L	L	H	H	L	L	H	L	H	H
18	19	L	H	L	L	H	L	H	H	L	L	H	H	L	L
20	21	L	H	L	H	L	L	H	H	L	H	L	L	L	L
22	23	L	H	L	H	H	L	H	H	L	H	L	L	L	H
24	25	L	H	H	L	L	L	H	H	L	H	L	L	H	L
26	27	L	H	H	L	H	L	H	H	L	H	L	L	H	H
28	29	L	H	H	H	L	L	H	H	L	H	L	H	L	L
30	31	L	H	H	H	H	L	H	H	L	H	H	L	L	L
32	33	H	L	L	L	L	L	H	H	L	H	H	L	L	H
34	35	H	L	L	L	H	L	H	H	L	H	H	L	H	L
36	37	H	L	L	H	L	L	H	H	L	H	H	L	H	H
38	39	H	L	L	H	H	L	H	H	L	H	H	H	L	L
40	41	H	L	H	L	L	L	H	H	H	L	L	L	L	L
42	43	H	L	H	L	H	L	H	H	H	L	L	L	L	H
44	45	H	L	H	H	L	L	H	H	H	L	L	L	H	L
46	47	H	L	H	H	H	L	H	H	H	L	L	L	H	H
48	49	H	H	L	L	L	L	H	H	H	L	L	H	L	L
50	51	H	H	L	L	H	L	H	H	H	L	H	L	L	L
52	53	H	H	L	H	L	L	H	H	H	L	H	L	L	H
54	55	H	H	L	H	H	L	H	H	H	L	H	L	H	L
56	57	H	H	H	L	L	L	H	H	H	L	H	L	H	H
58	59	H	H	H	L	H	L	H	H	H	L	H	H	L	L
60	61	H	H	H	H	L	L	H	H	H	H	L	L	L	L
62	63	H	H	H	H	H	L	H	H	H	H	L	L	L	H
All		X	X	X	X	X	H	H	H	H	H	H	H	H	H

Figure 10 - Function Table

Where does this table come from?

To use the function table, the value of the binary needs to be determined by looking at the “binary select” column for the binary value that has been input (where H represents a 1 and L represents a 0). The next step is to determine the expected BCD output by looking at the outputs column along the same row as the binary input.

When the 6-bit binary-to-BCD converter is simulated, a screen with a number of inputs (labelled A to E) is compared to the outputs (labelled Y1 to Y6), which makes reading it very easy. An example of this can be seen in the figure below, where the mouse was dragged to somewhere along the middle of the spectrum.



Figure 11 - Simulation of 6-bit Binary-to-BCD - 26/27 *Good Analysis*

Where the yellow marker is is where the inputs and outputs are being measured – the top five rows indicate the inputs, and the bottom six indicate the outputs; where the row is green is where the input/output is high i.e. 1, and where the row isn't is where the input/output is low i.e. 0. This is made easier by looking at the box on the left, which shows the binary values and BCD values clearly in 1s and 0s (where A and Y1 are LSB and E and Y6 are MSB). The binary value measured on the simulation is 01101, and the BCD value measured on the simulation is 010011. Using the function table on the last page, it can be seen that the value 01101 in binary is converted to 010011 in BCD, this represents a value of either 26/27, depending on whether the clock is rising or falling.

Another example of using the simulation to determine the BCD value is shown in the figure below.

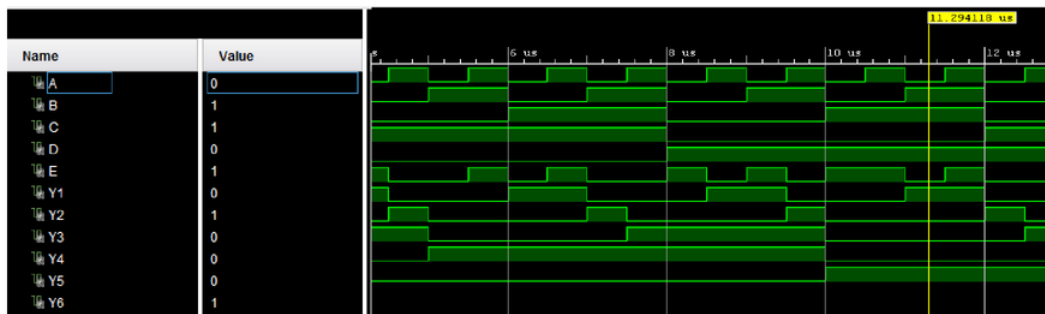


Figure 12 - Simulation of 6-bit Binary-to-BCD 44/45

The above figure shows that a binary value of 10110 is being input and a BCD value of 100010 is being output; looking at the function table on the last page, it can be seen that that is the exact expected output based on the input, and represents a value of 44/45.

Good Analysis

Another example of using the simulation to determine the output BCD value based on the input binary value is shown in the figure below.

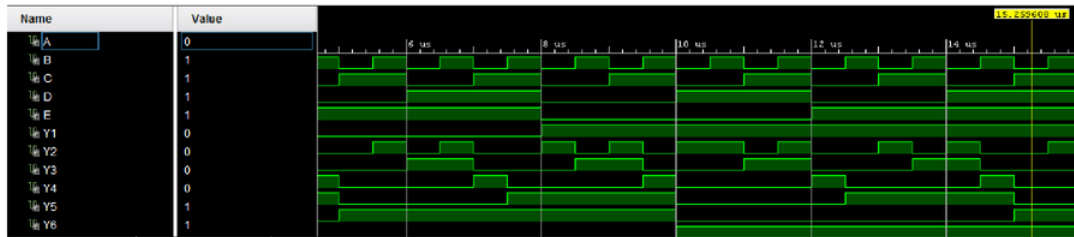


Figure 13 - Simulation of 6-bit Binary-to-BCD 60/61

The above figure shows that a binary value of 11110 is being input and a BCD value of 110000 is being output; looking at the function table, it can be seen that that is the exact expected output based on the input, and represents a value of 60/61.

Good Analysis

8-bit Binary-to-BCD Converter

Unlike the 6-bit binary-to-BCD converter, this converter's analysis did not involve making use of a function table to determine the BCD output based on a binary input. Determining the BCD output of the 8-bit binary-to-BCD converter simply involved comparing the binary value (B7 as MSB, B0 as LSB) to the BCD value (H = hundreds, T = tens, Un = units). An example of this is shown in the figure below, where the output was programmed to be 234.



Figure 14 - Simulation of 8-bit Binary-to-BCD - 234

At the point of the cursor, the binary input is shown to be 11101010, and the BCD output is shown to be 1000110100. In binary, 11101010 works out to be the value 234 ($128+64+32+8+2$). In BCD, the output is also worked out to be 234, as:

- The hundreds are 10, resulting in 2
 - The tens are 0011, resulting in 3
 - The units are 0100, resulting in 4
- Very Good

Another example of this is shown in the figure below, where the output was programmed to be 135.

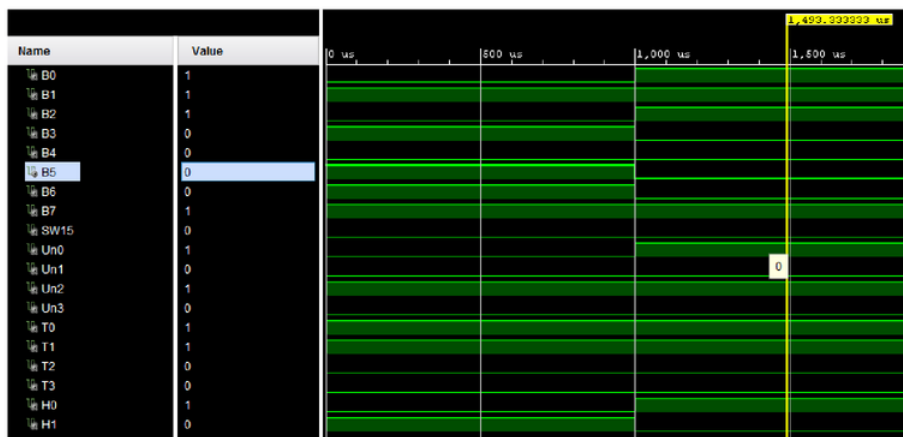


Figure 15 - Simulation of 8-bit Binary-to-BCD - 135

At the point of the cursor, the binary input is shown to be 10000111, and the BCD output is shown to be 0100110101. In binary, 10000111 works out to be the value 135 ($128+4+2+1$). In BCD, the output is also worked out to be 135, as:

- The hundreds are 01, resulting in 1
- The tens are 0011, resulting in 3 *Again, Good analysis*
- The units are 0101, resulting in 5

Another example of this is shown in the figure below, where the output was programmed to be 67.

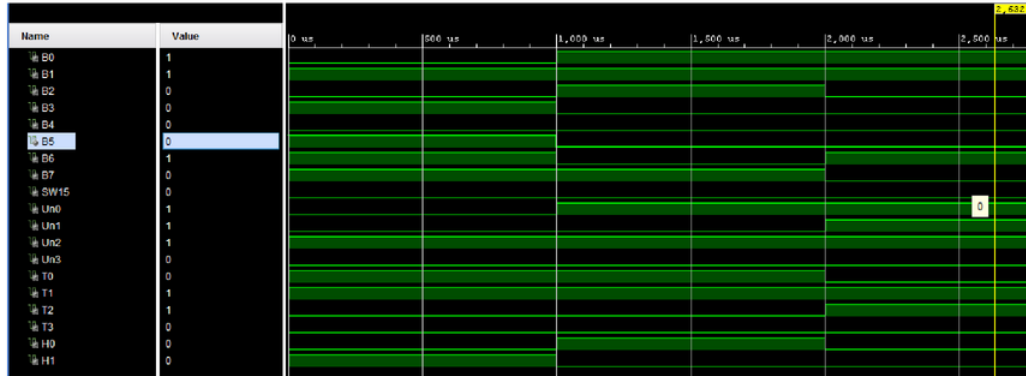


Figure 16 - Simulation of 8-bit Binary-to-BCD - 67

At the point of the cursor, the binary input is shown to be 01000011, and the BCD output is shown to be 0001100111. In binary, 01000011 works out to be the value 67 ($64+2+1$). In BCD, the output is also worked out to be 67, as:

- The hundreds are 00, resulting in 0
- The tens are 0110, resulting in 6
- The units are 0111, resulting in 7

8 bit analysis very good, 6 bit analysis could have been more detailed

5. Analysis of Nexys4 FPGA Practical Results

The process for uploading the 8-bit binary-to-BCD converter involves running the synthesis, running the implementation, generating the bitstream, opening the hardware manager from within Vivado and then connecting the device and programming. Once the 8-bit binary-to-BCD converter was uploaded to the Nexys 4 DDR board, the testing could begin. Testing the program involved using the eight switches to the bottom right of the board as binary to produce a BCD output on the LEDs, three arbitrary values were chosen to be tested – 234, 135 and 67. The first value, 234, was tested and an image of the test is in the figure below.

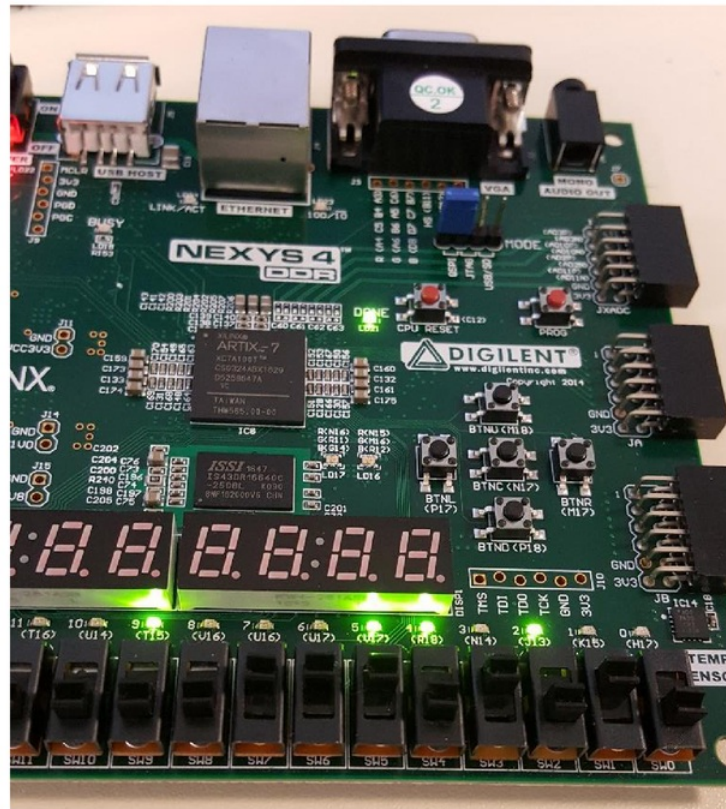


Figure 17 - 234 in Binary and BCD

The binary input 11101010 was input to acquire 234, and the BCD output was 1000110100, which also results in 234; this means that the first value that was tested worked correctly.

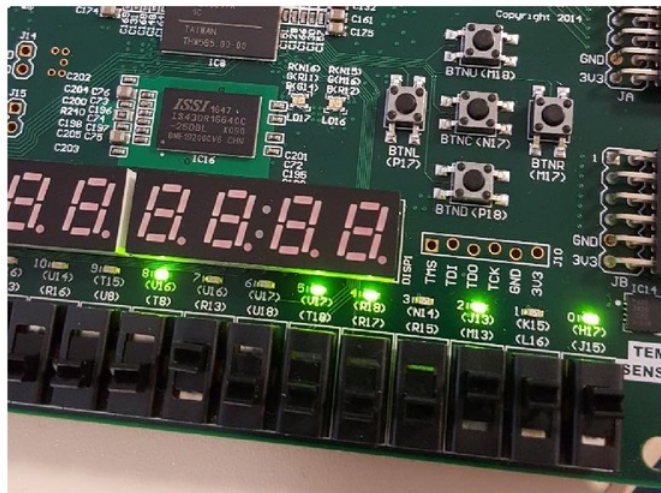


Figure 18 - 135 in Binary and BCD

The next value to be tested was 135 – the binary input was 10000111, and the LEDs produced the right BCD output of 0100110101.

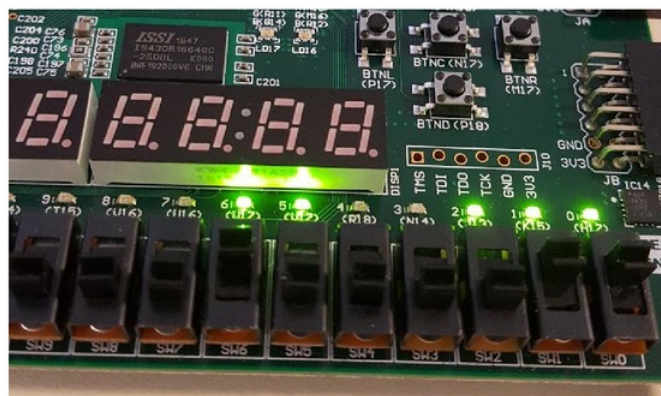


Figure 19 - 67 in Binary and BCD

The final value to be tested on the board was 67 – the binary input for this value was 01000011, and that produced the correct BCD output of 0001100111.

Very good use made of pictorial evidence and correct analysis shown

6. Adapting Design to Suit New Purposes or Applications

3*4-bit Multiplexer and Demultiplexer

The next component to be designed was the 3*4 bit Mux_Demux, this device was used to send multiple signals, one at a time; at a very rapid speed, giving the illusion that all signals are being sent and received at the same time. The first step in designing the 3*4 bit mux_demux was to create a project on the Vivado software, this project is used to contain the data written by the user concerning file directory. The next step was to create a VHDL design source – this file contains the entity and architecture of the 3*4 bit mux_demux. The entity and architecture of this particular file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.std_logic_unsigned.all;
5  entity Assignment1_MD is
6  Port ( FiveKHz : in STD_LOGIC;
7  Rst : in STD_LOGIC;
8  AN0,AN1,AN2 : out STD_LOGIC;
9  Un0,Un1,Un2,Un3 : in STD_LOGIC;
10 T0,T1,T2,T3 : in STD_LOGIC;
11 H0,H1 : in STD_LOGIC;
12 A,B,C,D : out STD_LOGIC
13 );
14 end Assignment1_MD;
15 architecture Behavioral of Assignment1_MD is
16 signal cnt : std_logic_vector (1 downto 0);
17 signal Q : std_logic_vector (2 downto 0);
18 begin
19 process (FiveKHz, Rst)
20 begin
21 if Rst = '1' then
22 cnt <= "11";
23 elsif rising_edge (FiveKHz) then
24 cnt <= cnt + "01";
25 end if;
26 end process;
27 process (cnt(1), cnt(0))
28 begin
29 if (cnt(1)='0' and cnt(0)='0') then Q <= "110";
30 elsif (cnt(1)='0' and cnt(0)='1') then Q <= "101";

```

Figure 20 - Multiplexer and Demultiplexer Design Source (1/3)

The entity of the mux_demux has 12 inputs (5KHz, reset, Un0, Un1, Un2, Un3, T0, T1, T2, T3, H0, H1) and 7 outputs (AN0, AN1, AN2, A, B, C, D). The architecture for the mux_demux states that the signal counts from 1 down to 0, and the output Q goes from 2 down to 0. The process in the architecture produces a count of "11" if the reset is on. Otherwise, once the program sees the rising edge of the 5KHz clock, the count increments by 1. The process for when the count is either 1 or 0 begins at line 27, and this sets the output to specific values e.g. when the cnt(1) and the cnt(0) are both 0, the output is 110.

```

31   elsif (cnt(1)='1' and cnt(0)='0') then Q <= "011";
32   else Q <= "111";
33   end if;
34   end process;
35   AN0 <= Q(0);
36   AN1 <= Q(1);
37   AN2 <= Q(2);
38   process (Un0,Un1,Un2,Un3,T0,T1,T2,T3,H0,H1,Q(0),Q(1),Q(2))
39   begin
40     if (Q(0) = '0' and Un0 = '0') then D<='0';
41     elsif (Q(0) = '0' and Un0 = '1') then D<='1';
42     elsif (Q(1) = '0' and T0 = '0') then D<='0';
43     elsif (Q(1) = '0' and T0 = '1') then D<='1';
44     elsif (Q(2) = '0' and H0 = '0') then D<='0';
45     elsif (Q(2) = '0' and H0 = '1') then D<='1';
46   end if;
47   if (Q(0) = '0' and Un1 = '0') then C<='0';
48   elsif (Q(0) = '0' and Un1 = '1') then C<='1';
49   elsif (Q(1) = '0' and T1 = '0') then C<='0';
50   elsif (Q(1) = '0' and T1 = '1') then C<='1';
51   elsif (Q(2) = '0' and H1 = '0') then C<='0';
52   elsif (Q(2) = '0' and H1 = '1') then C<='1';
53   end if;
54   if (Q(0) = '0' and Un2 = '0') then B<='0';
55   elsif (Q(0) = '0' and Un2 = '1') then B<='1';
56   elsif (Q(1) = '0' and T2 = '0') then B<='0';
57   elsif (Q(1) = '0' and T2 = '1') then B<='1';
58   elsif (Q(2) = '0') then B<='0';

```

Figure 21 - Multiplexer and Demultiplexer Design Source (2/3)

The process that begins in the figure above determines what output is being generated at the units, tens and hundreds; there is also a section which sets the output names as AN0, AN1 and AN2.

```

59   end if;
60   if (Q(0) = '0' and Un3 = '0') then A<='0';
61   elsif (Q(0) = '0' and Un3 = '1') then A<='1';
62   elsif (Q(1) = '0' and T3 = '0') then A<='0';
63   elsif (Q(1) = '0' and T3 = '1') then A<='1';
64   elsif (Q(2) = '0') then A<='0';
65   end if;
66   end process;
67   end Behavioral;

```

Figure 22 - Multiplexer and Demultiplexer Design Source (3/3)

The final part of the design source is a continuation of what output is being generated at the units, tens and hundreds. It finishes by ending the if, process and behavioural statements.

Good understanding shown on how the Mux/Demux has been designed

The next part in the design process was to add a simulation source, or “test bench” file. This test bench file serves the purpose of copying the information from the VHDL design source and determining how it is simulated. Simulation is an important aspect of the design process as it helps determine whether the VHDL program is behaving correctly. The test bench file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Assignment1_MD_tb is
4  -- Port ( ); No need to add info here as this is a testbench
5  end Assignment1_MD_tb;
6  architecture Behavioral of Assignment1_MD_tb is
7  component Assignment1_MD -- Enter file to be tested
8  -- Copy the Port information from the VHDL file
9  Port ( FiveKHz : in STD_LOGIC;
10       Rst : in STD_LOGIC;
11       AN0,AN1,AN2 : out STD_LOGIC;
12       Un0,Un1,Un2,Un3 : in STD_LOGIC;
13       T0,T1,T2,T3 : in STD_LOGIC;
14       H0,H1 : in STD_LOGIC;
15       A,B,C,D : out STD_LOGIC
16     );
17  end component;
18  -- input signals
19  -- Example signal A : std_logic := '0';
20  signal FiveKHz : std_logic := '0';
21  signal Rst : std_logic := '1';
22  signal Un0 : std_logic := '1';
23  signal Un1 : std_logic := '0';
24  signal Un2 : std_logic := '0';
25  signal Un3 : std_logic := '1';
26  signal T0 : std_logic := '1';
27  signal T1 : std_logic := '0';
28  signal T2 : std_logic := '0';
29  signal T3 : std_logic := '1';
30  signal H0 : std_logic := '1';

```

Figure 23 - Multiplexer and Demultiplexer Test Bench (1/3)

The entity for the TB file is ported from the VHDL file; the architecture also ports elements from the VHDL file such as the inputs and outputs. The next part of the architecture defines the input signals such as 5KHz, Rst, Un0 etc. and applies a logic value to them to be tested. The values that were applied to the program should result in 199 (based on the hundreds, H, tens, T, and units, Un).

```

31 : signal H1 : std_logic := '0';
32 -- should equate to 199
33 -- output signals
34 -- Example signal Q1 : std_logic := '0';
35 : signal AN0 : std_logic;
36 : signal AN1 : std_logic;
37 : signal AN2 : std_logic;
38 : signal A : std_logic;
39 : signal B : std_logic;
40 : signal C : std_logic;
41 : signal D : std_logic;
42 -- clock period definitions
43 -- Example constant A_period : time := 8ns;
44 : constant FiveKHz_period : time := 200us;
45 : begin
46 : -- Instantiate the UUT
47 : UUT : Assignment1_MD -- Enter file to be tested
48 : Port Map (
49 : -- Example A => A,
50 : FiveKHz => FiveKHz,
51 : Rst => Rst,
52 : AN0 => AN0,
53 : AN1 => AN1,
54 : AN2 => AN2,
55 : Un0 => Un0,
56 : Un1 => Un1,
57 : Un2 => Un2,
58 : Un3 => Un3,
59 : T0 => T0,
60 : T1 => T1,

```

Figure 24 - Multiplexer and Demultiplexer Test Bench (2/3)

The next part of the test bench file takes the input value of 199 and then turns it into output signals along A, B, C and D and also declares AN0, AN1 and AN2. The period for the mux_demux is also defined in this section as 200us and the ports are all mapped for the simulation.

```

61 : T2 => T2,
62 : T3 => T3,
63 : H0 => H0,
64 : H1 => H1,
65 : A => A,
66 : B => B,
67 : C => C,
68 : D => D
69 : );
70 : -- clock process definitions
71 : clk_process :process
72 : begin
73 :   FiveKHz <= '0';
74 :   wait for FiveKHz_period/2; --for 5 ns signal is '0'.
75 :   FiveKHz <= '1';
76 :   wait for FiveKHz_period/2; --for next 5 ns signal is '1'.
77 : end process;
78 : -- Stimulus process
79 : stim_proc: process
80 : begin
81 :   wait for 1ms;
82 :   Rst <= '0';
83 :   wait for 5ms;
84 :   Rst <= '1';
85 :   wait;
86 : end process;
87 : end Behavioral;

```

Figure 25 - Multiplexer and Demultiplexer Test Bench (3/3)

The last part of the TB file continues mapping the ports. There is also another section which defines the clock process and a stimulus process.

No constraints file was made for the 3*4-bit multiplexer and demultiplexer as it did not need to be uploaded to the FPGA.

Good understanding shown in how the TB file interacts with the Design file to give a simulated result which can be analysed to ensure correct operation.

5KHz Clock

The next component to be designed was the 5Khz clock, which is based on the 100Khz clock already implemented on the FPGA board. The purpose of the 5Khz clock is to produce the 5Khz frequency required for the mux_demux. The first step in designing the 5Khz clock was to create a project on the Vivado software, this project is used to contain the data written by the user concerning file directory. The next step was to create a VHDL design source – this file contains the entity and architecture of the 5Khz clock. The entity and architecture of this particular file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.std_logic_unsigned.all;
5  entity Assignment_1_5khz is
6  Port ( CLK100MHZ : in STD_LOGIC;
7        Rst : in std_logic;
8        FiveKHz : out STD_LOGIC
9  );
10 end Assignment_1_5khz;
11 architecture Behavioral of Assignment_1_5khz is
12 signal cnt1 : integer := 0;
13 signal temp1 : std_logic := '0';
14 begin
15 process(CLK100MHZ,Rst)
16 begin
17   if Rst = '1' then
18     temp1 <= '0';
19     cnt1 <= 0;
20   elsif rising_edge (CLK100MHZ) then
21     if(cnt1= 10000) then
22       temp1 <= not temp1;
23       cnt1 <= 0;
24     else
25       cnt1 <= cnt1 +1 ;
26     end if;
27   end if;
28 end process;
29 FiveKHz <= temp1;
30 end Behavioral;

```

Figure 26 - 5KHz Clock Design Source

The entity of the 5KHz clock has two inputs (100MHz clock, reset) and a single output (5KHz). The architecture for the 5KHz clock begins when there is a count of zero; the process states that the reset will return the temp to zero and the count to zero. If the reset is not on, then the temp changes, but not the count.

Where is the maths involved? Why if cnt1 = 10000?

Much more detail required!

The next part in the design process was to add a simulation source, or “test bench” file. This test bench file serves the purpose of copying the information from the VHDL design source and determining how it is simulated. Simulation is an important aspect of the design process as it helps determine whether the VHDL program is behaving correctly. The test bench file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.std_logic_unsigned.all;
5  -- Uncomment the following library declaration if instantiating
6  entity Assignment_1_5Khz_tb is
7  -- Port ( );
8  end Assignment_1_5Khz_tb;
9  architecture Behavioral of Assignment_1_5Khz_tb is
10 Component Assignment_1_5Khz
11 Port ( CLK100MHZ : in STD_LOGIC;
12       Rst : in STD_LOGIC;
13       FiveKHz : out STD_LOGIC
14 );
15 end Component;
16 --Inputs
17 signal CLK100MHZ : STD_LOGIC := '0';
18 signal Rst : STD_LOGIC := '1';
19 --Outputs
20 signal FiveKHz : STD_LOGIC;
21 --Clock definitions
22 constant CLK100MHZ_period : time := 10ns;
23 begin
24 --Instantiate the UUT
25 UUT : Assignment_1_5Khz
26 Port Map ( CLK100MHZ => CLK100MHZ,
27           Rst => Rst,
28           FiveKHz => FiveKHz
29 );
30 --Clock Process Definitions

```

Figure 27 - 5KHz Clock Test Bench File (1/2)

The architecture for the TB file for the 5KHz clock imports the inputs and outputs from the VHDL file; the inputs are given standard logic values – the 100MHz clock is given a standard logic of 0 and the reset is given a standard logic of 1. The 100MHz period is declared as 10ns to keep the frequency constant. The next lines of the TB file include instantiating the UUT; instantiating the UUT is there to map the inputs and outputs for the simulation to display information correctly to the reader.

```

31 CLK100MHZ_process :process
32 begin
33 CLK100MHZ <= '0';
34 wait for CLK100MHZ_period/2;
35 CLK100MHZ <= '1';
36 wait for CLK100MHZ_period/2;
37 end process;
38 --Stimulus process definitions
39 stim_proc: process
40 begin
41 wait for 1ms;
42 Rst <= '0';
43 wait for 2ms;
44 Rst <= '1';
45 wait;
46 end process;
47 end Behavioral;

```

Figure 28 - 5KHz Clock Test Bench File (2/2)

In the above figure, the clock process definitions are there to set the periods for each input.

No constraints file was made for the 5KHz clock as it did not need to be uploaded to the FPGA.

[Good detail on the TB file](#)

BCD to 7-segment Converter

The next component to be designed was the BCD to 7-segment converter. The BCD to 7-segment converter is in place to complete the final part of the assignment, which was to represent the binary value input on a 7-segment display; it achieves this by converting the BCD value produced by the binary-to-BCD converter into values which can be used by the on-board 7-segment. The first step in designing the BCD to 7-segment converter was to create a project on the Vivado software, this project is used to contain the data written by the user concerning file directory. The next step was to create a VHDL design source – this file contains the entity and architecture of the BCD to 7-segment converter. The entity and architecture of this particular file is shown below.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity BCD_7seg is
5
6      Port ( A,B,C,D          : in std_logic;
7            sa,sb,sc,sd,se,sf,sg : out std_logic
8            );
9
10 end BCD_7seg;
11
12 architecture Behavioral of BCD_7seg is
13
14     signal Q      : std_logic_vector (3 downto 0);
15     signal seg     : std_logic_vector (6 downto 0);
16
17     begin
18
19         Q(0) <= D;
20         Q(1) <= C;
21         Q(2) <= B;
22         Q(3) <= A;
23
24     process (Q)
25     begin
26         case Q is
27             WHEN "0000" =>          -- 0
28                 seg <= ("0000001");
29             WHEN "0001" =>          -- 1
30                 seg <= ("1001111");

```

Figure 29 - BCD to 7-Segment Converter Design Source (1/2)

The entity of the BCD to 7-segment converter design source contains four inputs (A, B, C and D) and seven outputs (sa, sb, sc, sd, se, sf and sg). The architecture features two signals – Q, which varies from 3 to 0, as there are four inputs; and signal seg, which varies from 6 down to 0, as there are seven outputs. The next part of the architecture assigns the inputs A, B, C and D to the signals Q(3), Q(2), Q(1) and Q(0) respectively. The process for (Q) is to output a value of 0000001 when the input is 0000, this is because zero is represented as such on a 7-segment display (where 0 = on and 1 = off). This is then repeated for each value from 0-9.

Good understanding shown

```

31 WHEN "0010" =>      -- 2
32   seg <= ("0010010");
33 WHEN "0011" =>      -- 3
34   seg <= ("0000110");
35 WHEN "0100" =>      -- 4
36   seg <= ("1001100");
37 WHEN "0101" =>      -- 5
38   seg <= ("0100100");
39 WHEN "0110" =>      -- 6
40   seg <= ("0100000");
41 WHEN "0111" =>      -- 7
42   seg <= ("0001111");
43 WHEN "1000" =>      -- 8
44   seg <= ("0000000");
45 WHEN "1001" =>      -- 9
46   seg <= ("0001100");
47
48 WHEN Others =>      -- 'blank'
49   seg <= ("1111111");
50 END CASE;
51 END PROCESS;
52
53 sa <= seg(6);
54 sb <= seg(5);
55 sc <= seg(4);
56 sd <= seg(3);
57 se <= seg(2);
58 sf <= seg(1);
59 sg <= seg(0);
60 end Behavioral;

```

Figure 30 - BCD to 7-Segment Converter Design Source (2/2)

A major part of the design source are lines 48 and 49, which turns all segments off when there is an input greater than 9 or less than 0. The final part of the software assigns segments to sa through to sg, from 6 to 0 respectively. [Good](#)

The next part in the design process was to add a simulation source, or “test bench” file. This test bench file serves the purpose of copying the information from the VHDL design source and determining how it is simulated. Simulation is an important aspect of the design process as it helps determine whether the VHDL program is behaving correctly. The test bench file is shown below.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  -- Uncomment the following library declaration if using
5  -- arithmetic functions with Signed or Unsigned values
6  --USE ieee.numeric_std.ALL;
7
8  ENTITY BCD_7seg_tb IS
9  END BCD_7seg_tb;
10
11 ARCHITECTURE behavior OF BCD_7seg_tb IS
12
13   -- Component Declaration for the Unit Under Test (UUT)
14
15   COMPONENT BCD_7seg
16   PORT(
17     A : IN std_logic;
18     B : IN std_logic;
19     C : IN std_logic;
20     D : IN std_logic;
21     sa : OUT std_logic;
22     sb : OUT std_logic;
23     sc : OUT std_logic;
24     sd : OUT std_logic;
25     se : OUT std_logic;
26     sf : OUT std_logic;
27     sg : OUT std_logic
28   );
29   END COMPONENT;
30

```

Figure 31 - BCD to 7-Segment Converter Test Bench File (1/4)

Similarly to other test bench files covered, the architecture for the BCD to 7-segment converter test bench file begins by stating the numerous inputs and outputs – in this case, there are four inputs and seven outputs.

```

31
32 --Inputs
33 signal A : std_logic := '0';
34 signal B : std_logic := '0';
35 signal C : std_logic := '0';
36 signal D : std_logic := '0';
37
38 --Outputs
39 signal sa : std_logic;
40 signal sb : std_logic;
41 signal sc : std_logic;
42 signal sd : std_logic;
43 signal se : std_logic;
44 signal sf : std_logic;
45 signal sg : std_logic;
46 -- No clocks detected in port list. Replace <clock> below with
47 -- appropriate port name
48
49 constant A_period : time := 80 ns;
50 constant B_period : time := 40 ns;
51 constant C_period : time := 20 ns;
52 constant D_period : time := 10 ns;
53
54 BEGIN
55
56 -- Instantiate the Unit Under Test (UUT)
57 uut: BCD_7seg PORT MAP (
58     A => A,
59     B => B,
60     C => C,

```

Figure 32 - BCD to 7-Segment Converter Test Bench File (2/4)

In the next part of the test bench file, the inputs and outputs are declared as signals following standard logic, and the input signals are given periods that are defined in constant periods of 10 ns, 20 ns, 40 ns and 80ns for the signals to produce all possible outputs as they will not be overlapping perpetually. Instantiating the UUT is there to map the inputs and outputs for the simulation to display information correctly to the reader.

```

61     D => D,
62     sa => sa,
63     sb => sb,
64     sc => sc,
65     sd => sd,
66     se => se,
67     sf => sf,
68     sg => sg
69 );
70
71 -- Clock process definitions
72 A_process :process
73 begin
74     A <= '0';
75     wait for A_period/2;
76     A <= '1';
77     wait for A_period/2;
78 end process;
79
80 B_process :process
81 begin
82     B <= '0';
83     wait for B_period/2;
84     B <= '1';
85     wait for B_period/2;
86 end process;

```

Figure 33 - BCD to 7-Segment Converter Test Bench File (3/4)

The clock process definitions for each process A and B are here to make the signal fluctuate from high to low, 1 to 0.

```

87
88     C_process :process
89     begin
90         C <= '0';
91         wait for C_period/2;
92         C <= '1';
93         wait for C_period/2;
94     end process;
95
96     D_process :process
97     begin
98         D <= '0';
99         wait for D_period/2;
100        D <= '1';
101        wait for D_period/2;
102    end process;
103
104 end behavior;
105

```

Figure 34 - BCD to 7-Segment Converter Test Bench File (4/4)

The clock process definitions for each process C and D are here to make the signal fluctuate from high to low, 1 to 0. [Good detail shown](#)

The constraints for the BCD to 7-Segment Converter are shown in the figure below.

```

1  ##Switches
2
3  set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { D }]; #IO_L24H_T3_R50_16 Sch=sv[0]
4  set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { C }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sv[1]
5  set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { B }]; #IO_L6N_T0_D08_VREF_14 Sch=sv[2]
6  set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { A }]; #IO_L13N_T2_MRCC_14 Sch=sv[3]
7
8  set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { SW0 }]; #IO_L24H_T3_34 Sch=sv[8]
9  set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { SW1 }]; #IO_25_34 Sch=sv[9]
10 set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { SW2 }]; #IO_L15P_T2_DQS_R0WR_B_14 Sch=sv[10]
11 set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { SW3 }]; #IO_L23P_T3_A03_D19_14 Sch=sv[11]
12 set_property -dict { PACKAGE_PIN R6 IOSTANDARD LVCMOS33 } [get_ports { SW4 }]; #IO_L24P_T3_35 Sch=sv[12]
13 set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { SW5 }]; #IO_L20P_T3_A05_D24_14 Sch=sv[13]
14 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { SW6 }]; #IO_L19P_T3_A05_D25_VREF_14 Sch=sv[14]
15 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW7 }]; #IO_L21P_T3_DQS_14 Sch=sv[15]
16
17  ##7 segment display
18
19 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { sa }]; #IO_L24H_T3_A00_D16_14 Sch=ce
20 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { sb }]; #IO_25_14 Sch=cb
21 set_property -dict { PACKAGE_PIN W16 IOSTANDARD LVCMOS33 } [get_ports { sc }]; #IO_25_18 Sch=cc
22 set_property -dict { PACKAGE_PIN W13 IOSTANDARD LVCMOS33 } [get_ports { sd }]; #IO_L17P_T2_A26_15 Sch=cd
23 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { se }]; #IO_L13P_T2_MRCC_14 Sch=ce
24 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { sf }]; #IO_L19P_T3_A10_D24_14 Sch=cf
25 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { sg }]; #IO_L4P_T0_D04_14 Sch=cg
26
27 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { AN0 }]; #IO_L23P_T3_F0E_B_15 Sch=an[0]
28 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { AN1 }]; #IO_L23N_T3_F0E_B_15 Sch=an[1]
29 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { AN2 }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
30 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { AN3 }]; #IO_L19P_T3_A22_15 Sch=an[3]
31 set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { AN4 }]; #IO_L8N_T1_D12_14 Sch=an[4]
32 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN5 }]; #IO_L14P_T2_MRCC_14 Sch=an[5]
33 set_property -dict { PACKAGE_PIN W2 IOSTANDARD LVCMOS33 } [get_ports { AN6 }]; #IO_L23P_T3_35 Sch=an[6]
34 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { AN7 }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

```

Figure 35 - BCD to 7-Segment Converter Constraints

The constraints file is there to define the ports for the FPGA to understand the VHDL file's coding for the inputs and outputs. The inputs in this case are the switches from 0-7 and the and the outputs are the individual segments of the 7-segment display. Inputs D, C, B and A along with outputs AN0, AN1, AN2, AN3, AN4, AN5, AN6 and AN7 are there in preparation for the final part of the assignment.

[What about how this design file is converted into an IP Block?](#)

8-bit Binary-to-BCD with 3*7-Segment LED Display

The final design for the assignment was created by combining the former pieces together. The figure below shows the finished design.

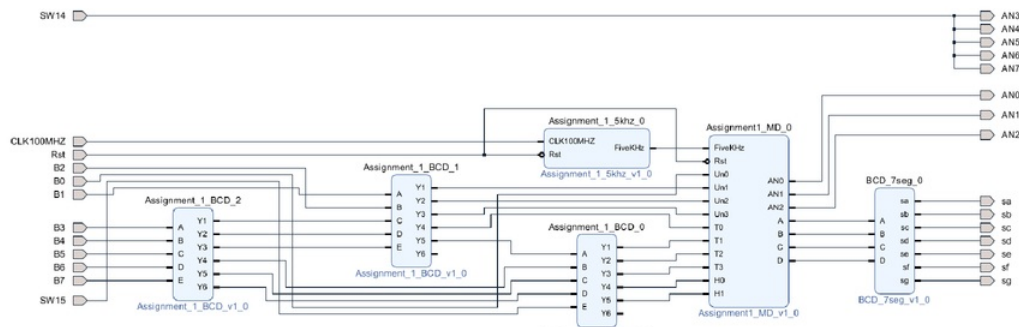


Figure 36 - 8-bit Binary-to-BCD with 3*7-Segment LED Display Design

In total, the final design featured a 5KHz clock, a mux_demux, a BCD to 7-segment converter, and three 6-bit binary-to-BCD converters (which combined make an 8-bit binary-to-BCD converter). The design works by multiplexing units when AN0 is active, tens when AN1 is active, and hundreds when AN2 is active. The final design didn't have a design source or test bench file because the whole project was made up of multiple other files which already had their design sources and test benches; there was, however, a constraints file which was written for the final design, shown below. [Good](#)

```
1 1 // Clock signal
2 set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { CLK100MHz }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
3 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { CLK100MHz }];
4
5
6 ##Switches
7
8 set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { B0 }]; #IO_L24N_T3_R50_15 Sch=sv[0]
9 set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { B1 }]; #IO_L3N_T0_D05_SMCCLK_14 Sch=sv[1]
10 set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { B2 }]; #IO_L6N_T0_D06_VREF_14 Sch=sv[2]
11 set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { B3 }]; #IO_L12N_T2_MRCC_14 Sch=sv[3]
12 set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { B4 }]; #IO_L12N_T1_MRCC_14 Sch=sv[4]
13 set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { B5 }]; #IO_L7N_T1_D10_14 Sch=sv[5]
14 set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { B6 }]; #IO_L17N_T2_A13_D29_14 Sch=sv[6]
15 set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { B7 }]; #IO_L5N_T0_D07_14 Sch=sv[7]
16 set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { Rat }]; #IO_25_34 Sch=sv[9]
17 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { SW14 }]; #IO_L12N_T0_A09_D26_VREF_14 Sch=sv[14]
18 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW15 }]; #IO_L21P_T3_D05_14 Sch=sv[15]
19
20
21
22 ##7 segment display
23
24 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { a }]; #IO_L24N_T3_A00_D16_14 Sch=ca
25 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { ab }]; #IO_25_14 Sch=cb
26 set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { ac }]; #IO_25_15 Sch=cc
27 set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { ad }]; #IO_L17P_T2_A26_15 Sch=cd
28 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { ae }]; #IO_L13P_T2_MRCC_14 Sch=ce
29 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { af }]; #IO_L19P_T3_A10_D26_14 Sch=cf
30 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { ag }]; #IO_L4P_T0_D04_14 Sch=cg
31
32 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { AN0 }]; #IO_L25P_T3_F0R_B_15 Sch=an[0]
33 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { AN1 }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
34 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { AN2 }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
35 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { AN3 }]; #IO_L18P_T3_A22_16 Sch=an[3]
36 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN4 }]; #IO_L8N_T1_D12_14 Sch=an[4]
37 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN5 }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
38 set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { AN6 }]; #IO_L23P_T3_35 Sch=an[6]
39 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { AN7 }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

Figure 37 - Final Design Constraints

The inputs for the constraint file for the final design are switches 0-7, which dictate the binary value being calculated by the board. There is also a reset switch, which when enabled will stop the 7-

segment display from presenting anything. SW15 is included to be used as a ground, as the values displayed would not display correctly without ground. The segments of the display are denoted by sa, sb, sc, sd, se, sf and sg in the constraints file.

Good design shown for full application and explained well

7. Analysis of Xilinx Vivado Simulated Results

3*4-bit Multiplexer and Demultiplexer

The way the 3*4-bit multiplexer and demultiplexer simulation works is by going through the units, then the tens and then the hundreds. When making the test bench file for the mux_demux, a value of 199 was coded in to be outputted, allowing the mux_demux to be tested; that code can be seen in the figure below.

```

22 : signal H0 : std_logic := '1';
23 : signal Un0 : std_logic := '1';
24 : signal Un1 : std_logic := '0';
25 : signal Un2 : std_logic := '0';
26 : signal Un3 : std_logic := '1';
27 : signal T0 : std_logic := '1';
28 : signal T1 : std_logic := '0';
29 : signal T2 : std_logic := '0';
30 : signal T3 : std_logic := '1';
31 : signal H0 : std_logic := '1';
32 : signal H1 : std_logic := '0';
33 : -- should equate to 199
34 : -- output signals

```

Figure 38 - 3*4-bit Mux_Demux Output Code

In order to make sure that a correct output is being generated, AN0, AN1 and AN2 need to be measured on the simulation (shown in the figures below) individually.



Figure 39 - 3*4-bit Mux_Demux Output - Units

In the above figure, AN0 is representing a value of zero at the marker; AN0 is used here to indicate that the value of the units is being measured, and A, B, C and D are showing the binary value of the units. In this case, a binary value of 1001 is being produced by the units which is 9, as expected.

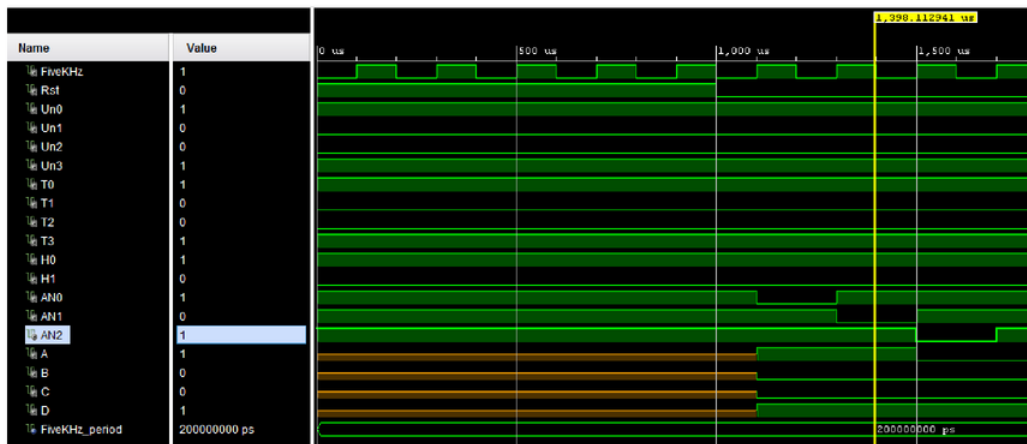


Figure 40 - 3*4-bit Mux_Demux Output – Tens

In the figure above, AN1 is 0, and so the tens are being measured. A binary value of 1001 is being produced from the A, B, C and D outputs, which is the expected output of 9.

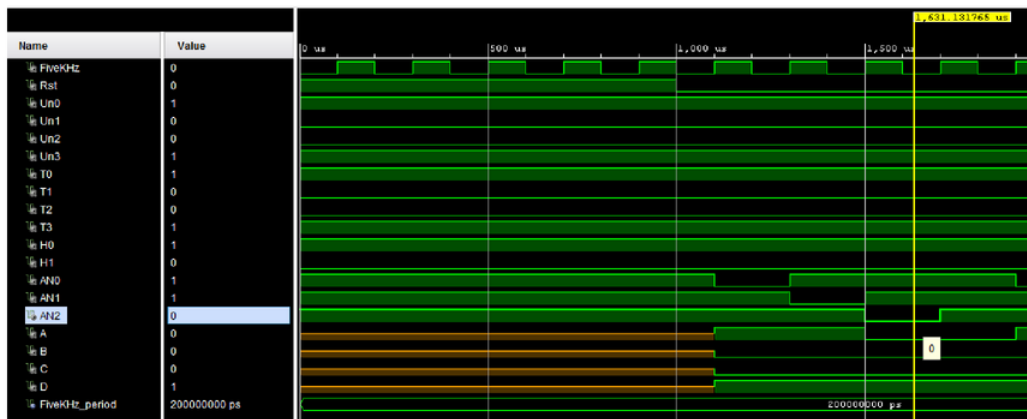


Figure 41 - 3*4-bit Mux_Demux Output – Hundreds

In the figure above, AN2 is 0, and so the hundreds are being measured. A binary value of 0001 is being produced from the A, B, C and D outputs, which is the expected output of 1.

Good example well analysed

5KHz Clock

Simulating the 5KHz clock resulted in three results – the 100MHz clock, the reset and the 5KHz clock; this can be seen on the figure below.



Figure 42 - 5KHz Clock Simulation

As can be seen from above, the 100MHz clock is always running, this is because it is unaffected by the reset because it will be produced by the FPGA. The 5KHz clock however is effected by the reset, as it can be seen that no value is being produced by the 5KHz clock when the reset is on. When the reset is turned off, the 5KHz clock produces an intermittent value; the intermittency is called the period, and could be observed by placing two markers between two rising-edges of the clock pulses, shown clearly in the figure below.

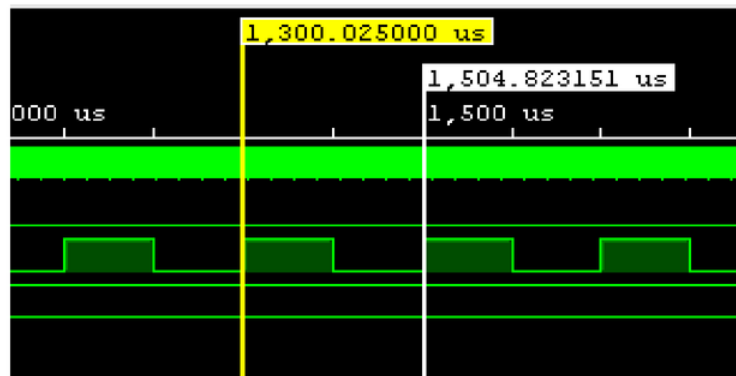


Figure 43 - 5KHz Clock Simulation - 200ms Difference

The difference between the two rising edges is 200ms, and so there is a period of 200ms. This value can be proven mathematically by using the equation $F = \frac{1}{p}$.

$$F = \frac{1}{200 * 10^{-6}} = \frac{10^{-6}}{200} = 5KHz$$

Again good example, well analysed

BCD to 7-Segment Display

In order to test the BCD to 7-segment display, the simulation produced a number of either on or off states represented as A, B, C and D; moving the cursor to create a binary value between 1-9 out of those A, B, C, and D values would then turn on the corresponding sa-sg values to represent the number of the A, B, C and D values in 7-segment format. The A value represents the MSB and the D value represents the LSB. An example of this is shown in the figure below, where the number 6 is generated.



Figure 44 - BCD to 7-Segment Simulation - 6

In the above figure, the A, B, C and D inputs create the binary output 0110, which in decimal is the number 6. In this simulation, for sa-sg, a 0 would represent an on state and a 1 would represent an off state. With that in mind, looking at the states on the left, it can be deduced that:

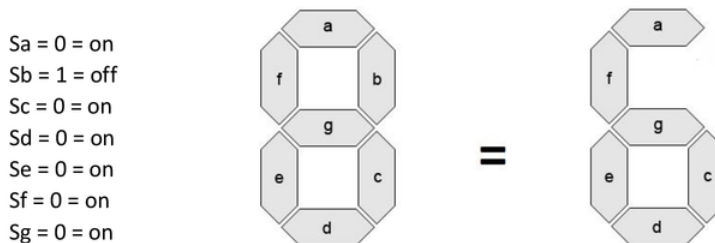


Figure 45 - 7-Segment for the value 6

Using a 7-segment display (shown in the above figure) for reference, the output of the circuit would turn on every segment apart from b, which would produce a 6. This process could be replicated to simulate what output would be produced on the 7-segment output board for any value between 0-9.

In the figure below, the A, B, C and D inputs create the binary output 1001, which in decimal is the number 9.

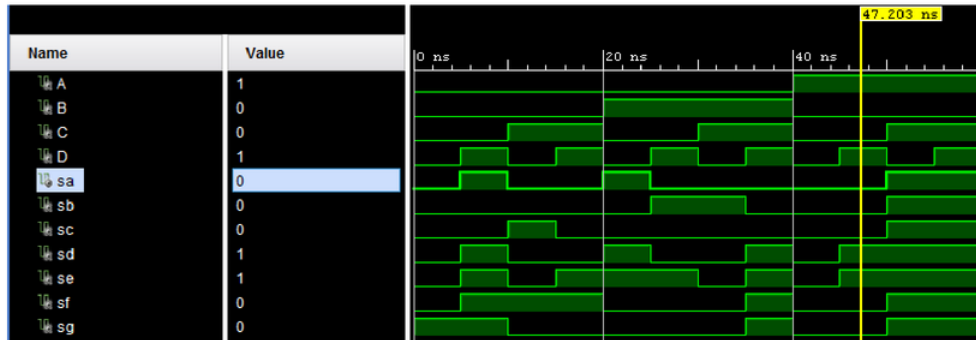


Figure 46 - BCD to 7-Segment Simulation - 9

In this simulation, for sa-sg, a 0 would represent an on state and a 1 would represent an off state. With that in mind, looking at the states on the left, it can be deduced that:

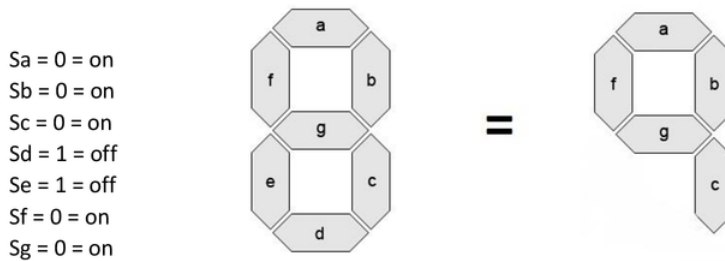


Figure 47 - 7-Segment for the value 9

Using a 7-segment display for reference, the output of the circuit would turn on every segment apart from sd and se, which would produce the aspect of the value 9.

For inputs > 9, the output simply turns off all segments to show nothing; this can be seen in the figure below, where the input is the binary value 1010, representing the decimal value 10, and none of the outputs are on.

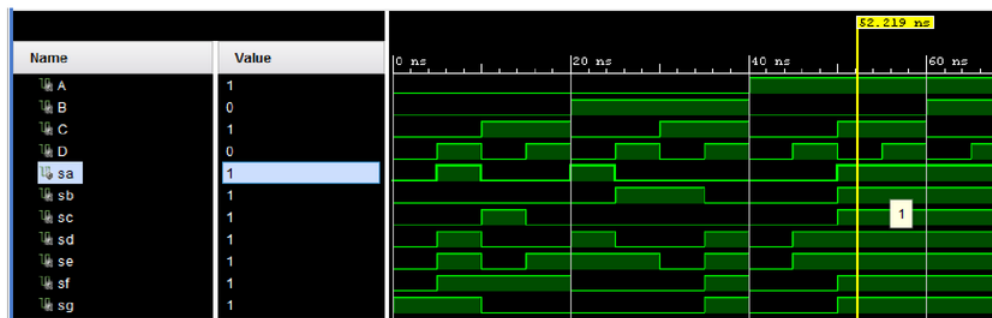


Figure 48 - BCD to 7-Segment Simulation - 10

Excellent analysis shown of 7 segment decoder

8. Analysis of Nexys 4 DDR FPGA Practical Results

The process for uploading the 8-bit binary-to-BCD converter on 3*7-segment display involves running the synthesis, running the implementation, generating the bitstream, opening the hardware manager from within Vivado and then connecting the device and programming. Once the 8-bit binary-to-BCD converter on 3*7-segment display was uploaded to the Nexys 4 DDR board, the testing could begin. Testing the program involved using the eight switches to the bottom right of the board as binary to produce a BCD output which could be presented on the 7-segment displays to present the value of the input binary value. The same three values from the last practical test were chosen to be tested – 234, 135 and 67. The first value, 234, was tested and an image of the test is in the figure below.

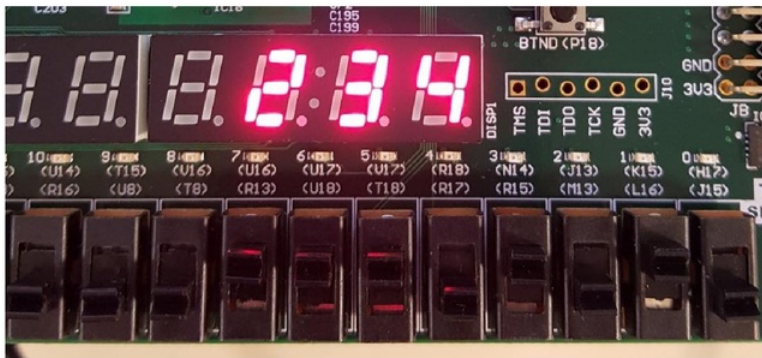


Figure 49 - 234 on 7-Segment

The binary input 11101010 was input to acquire 234, the three seven-segment displays showed the expected value.

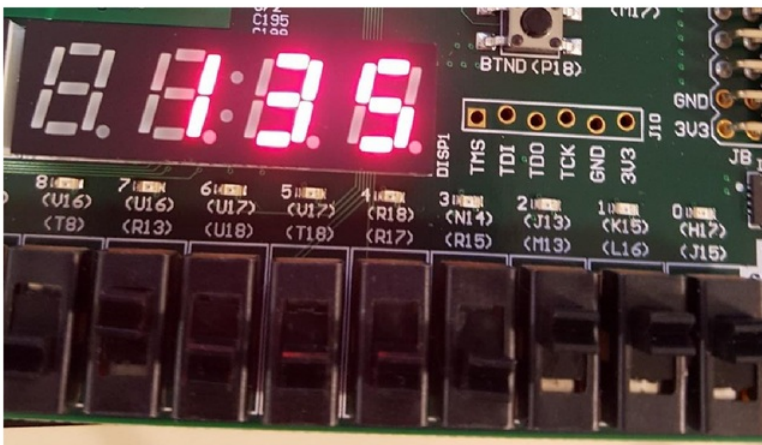


Figure 50 - 135 on 7-Segment

The next value to be tested was 135 – the binary input was 10000111, and the three seven-segment displays produced the right output.

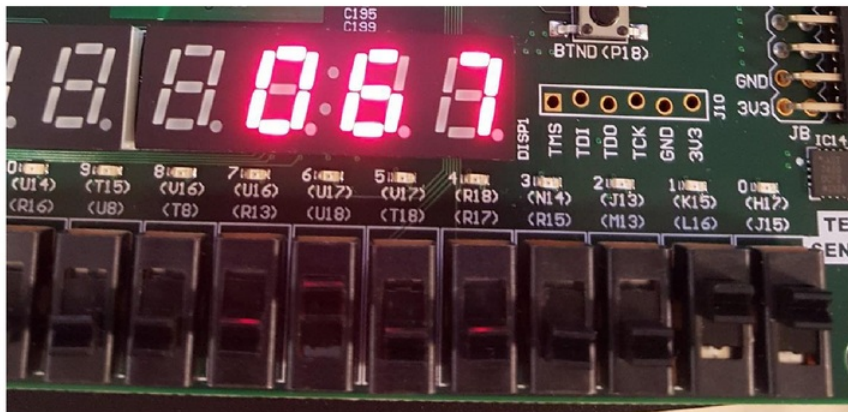


Figure 51 - 67 on 7-Segment

Another value to be tested on the board was 67 – the binary input for this value was 01000011, and that produced the correct three seven-segment values.

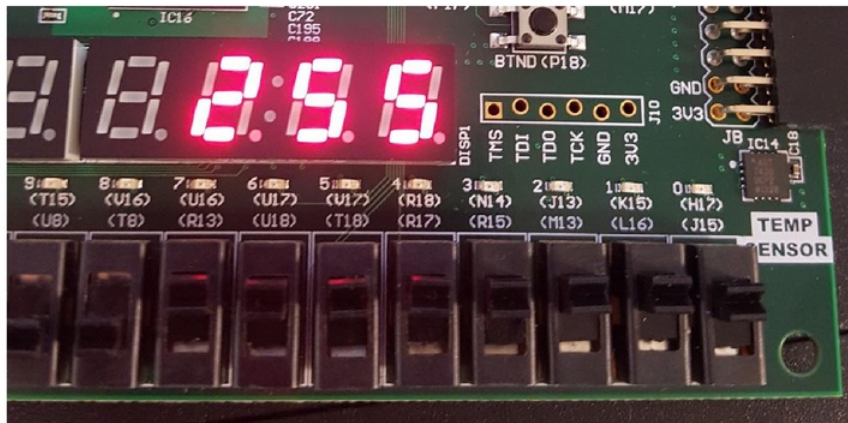


Figure 52 - 255 on 7-Segment

Finally, to ensure that all binary inputs would work, a value of 255 was input. All three tests passed, and so did the additional test to determine that all inputs worked correctly.

Very good set of practical results shown which have been analysed well

9. Conclusions

In the few tasks involved in this assignment, VHDL has proven to be an efficient tool for developing the necessary components and circuits that have been used to achieve a multitude of outcomes. I have learnt that the applications of VHDL technology are vast, and that those applications can be developed and expanded upon with minimal friction when VHDL is utilised properly. The Vivado software, which was used to develop and simulate the VHDL files as well as communicate with the Nexys 4 DDR FPGA, proved functional and intuitive throughout its many uses – an example of the intuitiveness of the software is when an error occurred when the filename of a project featured too many characters; the Vivado software provides a comprehensible log of any errors that occur from within projects, and the error log specified exactly what the problem was as the error occurred, helping to amend the error quickly and without having to look up error codes.

The assignment aims involved designing:

- A 6 bit Binary to BCD converter (later extended to become an 8-bit Binary-to-BCD converter)
- A 3*4 bit Mux_DeMux
- A 5 KHz clock from the internal 100MHz clock on the Nexys 4 DDR FPGA
- A BCD to 7 segment converter

Each of these files would then need to be compounded to produce the final design, which would have a binary input which would then produce a BCD value to be displayed on three 7-segment LED displays. The assignment aims have all been met, and I feel sufficient evidence is present throughout the report to reflect that. *I agree!*

Good overall conclusion given but little reflection noted

10. Bibliography

References

Cohen, B. (1995). *VHDL Coding Styles and Methodologies*. New York: Kluwer Academic Publishers.

*Citing
list
leave
s a lot
to be
desired!*

PSDE5029 Assignment 1

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

65/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

Text Comment. Brief, concise introduction given...

PAGE 5

Text Comment. Not enough Detail

Text Comment. Not enough detail

Text Comment. An example would have been useful at this point

Text Comment. Very little research carried out with only one citation given.... very poor!

PAGE 6

PAGE 7

Text Comment. Good detail

PAGE 8

Text Comment. Good detail

PAGE 9

Text Comment. Good Detail

PAGE 10

Text Comment. What does UUT mean?

PAGE 11

Text Comment. Not explained fully

Text Comment. Good

PAGE 12

Text Comment. What about the setup procedure?

Text Comment. Very little explanation on how the IP blocks were obtained

PAGE 13

Text Comment. Where does this table come from?

PAGE 14

Text Comment. Good Analysis

Text Comment. Good Analysis

PAGE 15

Text Comment. Good Analysis

PAGE 16

Text Comment. Very Good

PAGE 17

Text Comment. Again, Good analysis

Text Comment. 8 bit analysis very good, 6 bit analysis could have been more detailed

PAGE 18

PAGE 19

Text Comment. Very good use made of pictorial evidence and correct analysis shown

PAGE 20

PAGE 21

Text Comment. Good understanding shown on how the Mux/Demux has been designed

PAGE 22

PAGE 23

PAGE 24

Text Comment. Good understanding shown in how the TB file interacts with the Design file to give a simulated result which can be analysed to ensure correct operation.

PAGE 25

Text Comment. Where is the maths involved? Why if $\text{cnt1} = 10000$? Much more detail required!

PAGE 26

Text Comment. Good detail on the TB file

PAGE 27

Text Comment. Good understanding shown

PAGE 28

Text Comment. Good

PAGE 29

PAGE 30

Text Comment. Good detail shown

Text Comment. What about how this design file is converted into an IP Block?

PAGE 31

Text Comment. Good

PAGE 32

Text Comment. Good design shown for full application and explained well

PAGE 33

PAGE 34

Text Comment. Good example well analysed

Text Comment. Again good example, well analysed

Text Comment. Excellent analysis shown of 7 segment decoder

Text Comment. Very good set of practical results shown which have been analysed well

Text Comment. I agree!

Text Comment. Good overall conclusion given but little reflection noted

Text Comment. Citing list leaves a lot to be desired!

GRADING FORM: CSG-RUBIC

JOSEFF JONES

65

RESEARCH-INFORMED LITERATURE



Structure of a typical VHDL file: Could have been much more detailed. (1/5) Binary to BCD conversion: Again, could have been much more detailed. (2/5) IP Block Design in Vivado: An example would have been useful at this point. (2/5) Very little to no research cited... Total (5/15)

5

KNOWLEDGE AND UNDERSTANDING



Design of 6/8 bit Binary to BCD Converter: No reference made to the Data sheet on the DM74185, otherwise good explanation shown of design and testbench files for both the 6 bit (4/5) and the 8 bit design. (3/5) Re-design to suit new applications: Mux/Demux design good (2/3); 5KHz clock design not explained well (1/3); 7 segment decoder design shows understanding. (2/3). Full design adaptation clear and correct. (4/6) Total (16/25)

16

ANALYSIS



Analysis of Xilinx Vivado simulation results: Good analysis of the 8 bit Binary to BCD results (4/5), but the 6 bit results could have been more detailed. (2/5) Analysis of adapted application results: Very good analysis given on all results shown. Mux/Demux (3/5); 5KHz clock (4/5); 7 segment decoder. (4/5) Total (17/25)

17

PRACTICAL APPLICATION AND DEP.



Downloading and testing to specification on Nexys 4 DDR FPGA board: Good, clear set of results obtained which have been analysed well for the 8 bit Binary to BCD converter. (8/10) Downloading and testing of new application results on the Nexys 4 DDR FPGA board: Very good set of practical results shown for the adapted design which have been analysed well. (8/10) Total (16/20)

16

SKILLS FOR PROFESSIONAL PRAC.



Deployment of appropriate media: Very good use made of screen shots and pictorial evidence when designing and analysing results. ((4/5) Report layout and readability: The report flowed well with all aspects of the assignment covered well to a large extent. (4/5) Appropriate conclusion: Good understanding shown on how the results met the specifications within the assignment. Good reflection. (3/5) Total (11/15)

11