

Virtual Instrumentation & Control

Assignment 1 FINAL Scott

Dedoncker

by Scott Dedoncker

FILE	12968_SCOTT_DEDONCKER_VIRTUAL_INSTRUMENTATION__CONTROL_ASSIGNMENT_1_FINAL_SCOTT_DEDONCKER_155114_1660704928.PDF (1.94 M)		
TIME SUBMITTED	18-JAN-2018 09:42AM (UTC+0000)	WORD COUNT	7380
SUBMISSION ID	81987247	CHARACTER COUNT	41141

HND Electrical & Electronic Engineering
PSDE 5029: Virtual Instrumentation & Control
Systems



Assignment 1

Binary/BCD Conversion in VHDL

Scott Dedoncker

Valero Energy Electrical Apprentice

Date of Submission: 18/1/18

Word Count: 7458

Contents

Introduction	3
1. Research.....	4
1.1 What is VHDL?.....	4
1.2 Structure of a Typical VHDL File	4
1.3 Binary to BCD conversion.....	5
1.4 IP's in Vivado Block Design.....	6
2. Design of Assignment Requirements	7
2.1 6 Bit Binary to BCD Converter	7
2.1.1 VHDL File	8
2.1.2 VHDL Testbench (Simulation) File.....	10
2.2 Analysis of 6 Bit Binary to BCD Simulation Results	12
2.3 Adapting Design for an 8 Bit Converter	14
3. Analysis of Xilinx Vivado Simulated Results	15
4. Analysis of Nexys4 FPGA Practical Results	18
5. Adapting Design to Suit New Purposes.....	20
5.1 Five KHz Clock Design.....	21
5.1.1 Five KHz Clock VHDL File	21
5.1.2 Five KHz Clock Test Bench (Simulation) File.....	22
5.2 Multiplexer Design.....	24
5.2.1 Multiplexer VHDL File	24
5.2.2 Multiplexer Test Bench (Simulation) File	26
5.3 7-Segment Display Design.....	28
5.3.1 7-Segment Display VHDL File	28
5.3.2 7-Segment Display Test bench (Simulation) File.....	30
6. Analysis of Xilinx Vivado Simulated Results	32
6.1 5 KHz Clock Simulation.....	32
6.2 Multiplexer Simulation.....	33
6.3 BCD to 7-Segment Display Simulation	37
7. Analysis of Nexys 4 DDR FPGA Practical Results	39
7.1. Building & Testing the First Block Design.....	39
7.2. Building & Testing the Second Block Design.....	41
7. Conclusions	44
8. Appendices.....	45
Bibliography	46

Figure 1 - Project Summary	7
Figure 2 - VHDL File Code 1.....	8
Figure 3 - VHDL File Code 2.....	9
Figure 4 - Testbench File Code 1.....	10
Figure 5 - Testbench File Code 2.....	11
Figure 6 - 6 Bit Sim 1	12
Figure 7 - 6 Bit Sim 2	12
Figure 8 - 6 Bit Sim 3	13
Figure 9 - 8 Bit Cinverter Code.....	14
Figure 10 - 232 Binary Value.....	15
Figure 11 - 232 Binary to BCD Conversion.....	15
Figure 12 - 119 Binary Value.....	16
Figure 13 - 119 Binary to BCD Conversion.....	16
Figure 14 - 43 Binary Value.....	17
Figure 15 - 43 Binary to BCD Conversion.....	17
Figure 16 - 232 BCD FPGA Value.....	18
Figure 17 - 119 BCD FPGA Value.....	19
Figure 18 - 43 BCD FPGA Value.....	19
Figure 19 - Five KHz Clock VHDL Code.....	21
Figure 20 - Five KHz Clock Testbench Code 1	22
Figure 21 - Five KHz Clock Testbench Code 2	23
Figure 22 - Multiplexer VHDL Code 1	24
Figure 23 - Multiplexer VHDL Code 2	25
Figure 24 - Multiplexer Testbench Code 1.....	26
Figure 25 - Multiplexer Testbench Code 2.....	27
Figure 26 - 7-Segment Display VHDL Code	28
Figure 27 - 7-Segment Display Examples.....	29
Figure 28 - 7-Segment Display Testbench Code 1	30
Figure 29 - 7-Segment Display Testbench Code 2	31
Figure 30 - 5 KHz Clock Simulation	32
Figure 31 - 1 Period of the Clock.....	32
Figure 32 - Multiplexer Testbench Code & Simulation	33
Figure 33 - Multiplexer Simulation Values 1.....	34
Figure 34 - Multiplexer Simulation Values 2.....	35
Figure 35 - Multiplexer Simulation Values 3.....	36
Figure 36 - 7-Segment Display Simulation	37
Figure 37 - 7-Segment Display Test Value of 10	37
Figure 38 - 7-Segment Display Test Value of 8	38
Figure 39 - 7-Segment Display Test Value of 2	38
Figure 40 - First Test Block Design.....	39
Figure 41 - Test Block Design Constraints.....	39
Figure 42 - 257 Test	40
Figure 43 - 125 test.....	40
Figure 44 - 39 test.....	40
Figure 45 - Second Block Design.....	41
Figure 46 - Second Constraints	41
Figure 47 - Test for 255.....	42
Figure 48 - Test for 119.....	42
Figure 49 - Test for 232.....	42
Figure 50 - Test for 43.....	43

Introduction

The overall purpose for this assignment is to investigate, by researching and performing practical experiments, the Nexys4 DDR FPGA board, the coding language VHDL, Vivado design suite, and how they all link together. This will involve research into the subject that will cover key areas including:

- The VHDL language
- The method of converting binary values to BCD values
- What an IP is
- How Vivado utilises IP's for system design

This research will need to be well structured and coherent, as well as giving a good understanding for all of the points raised.

The knowledge gained from this research will then need to be applied to a set of practical's and simulations using a Nexy4 DDR FPGA. This involves building the following programs:

- 6 bit binary to BCD converter
- Adapting the 6 bit converter to accommodate 8 bits (1 byte)
- 5 KHz clock
- Multiplexer/De-multiplexer
- BCD to 7-segment display

All of these programs will involve creating a VHDL file, as well as a test bench file that will be simulated in Vivado prior to the files being downloaded to the FPGA.

Good introduction to the assignment which clearly shows what is required to do to complete

1. Research

1.1 What is VHDL?

There are many variations of language when it comes to programming. One key language used to program electronics, design systems and express different methods of connecting devices and how they behave is VHDL. VHDL is an IEEE recognised and standardised language that is used across the board in programming and on a broader scale in engineering as a whole. The reason for this is the fact that the language is so powerful and versatile while also not being too heavily taxing on software or hardware alike. Its bread and butter is basic combinations and sequential when designing a simple digital circuit.

(James, 2002) (Stephen D. Brown, 2009)

James states that there are 3 different techniques when it comes to designing a file using VHDL:

1. "Structural: Specifies how components such as gates are interconnected."
2. "Data flow: Specifies how data flows from input signals to output signals."
3. "Behavioural: Specifies in terms of a sequential algorithm how outputs are determined from inputs"

(James, 2002)

VHDL takes Boolean functions and presents them as a single statement that can be recognised by the FPGA. For example a Boolean statement of:

$X = B + AC$

And represent as the following:

$X \leq B \text{ or } (A \text{ and } C)$

VHDL code usually consists of large numbers of statements combined together and declared using architecture. VHDL is not a case sensitive language and uses the standardised ";" expression to end lines.

1.2 Structure of a Typical VHDL File

The basic foundations for a VHDL file are found at its very beginning. The first declarations in a VHDL file are for the entity and the architecture. The entity is a list of methods in which the file can establish connections with modules within the program by means of using interface signals. They can also be used to establish connections with external signals by using a port statement. It can also be the place where all the inputs and outputs for that VHDL file can be listed.

The architecture of the file sets the functions for the design. It specifies how the system will operate and includes declarations for all of the signals, components, functions and procedures. This typically includes a behavioural model that begins in the architecture and ends when the program does.

(James, 2002) (EuroElectronica, 2016) (Pedroni, 2008)

Good

1.3 Binary to BCD conversion

Binary to BCD conversion is the process of taking a binary value, usually an 8 bit byte, and converting it into a binary converted decimal, or BCD. This BCD is a set of digits where each digit represents a fixed number of bits between 1 and 8 per digit usually, although this can differ.

A Binary number is a number that is expressed in the base 2 numeral system. This is the system where each number increment (bit) doubles in size to give the value of the next bit. Each bit is symbolised as either a 1 or a 0 and it is standard to read a binary number from right to left (this in some cases is not always the case though). Example of binary numbers can be seen below:

1 = 0 0 0 1
2 = 0 0 1 0
4 = 0 1 0 0
5 = 0 1 0 1
9 = 1 0 0 1
20 = 0 0 0 1 0 1 0 0
84 = 0 1 0 1 0 1 0 0

A BCD number is a number where each digit is represented by its own set of 1 – 4 bits. Whereas binary systems usually use a full byte (8bits) to symbolise a number, BCD systems take advantage of the fact that 4 bits can display the numbers 0-9, and so a byte can in theory give information for two individual digits if the two sets of 4 bits are recognised as separate entities. An example of BCD numbers can be seen below:

6 = 0 0 0 0 0 1 1 0 = (0 0 0 0) – 0, (0 1 1 0) – 6.
67 = 0 1 1 0 0 1 1 1 = (0 1 1 0) – 6, (0 1 1 1) – 7.
333 = 0 0 1 1 0 0 1 1 0 0 1 1 = (0 0 1 1) – 3, (0 0 1 1) – 3, (0 0 1 1) – 3.

The algorithm typically used to convert binary values into BCD values is commonly referred to as the “double dabble” algorithm. This algorithm takes the inputted binary value and stores it. It then takes what number the binary value portrays, and breaks each digit into its own 4bits. Each set of 4 bits is commonly referred to as (from the least significant figure up) units, tens, hundreds, thousands. The beauty of this system is that the order of which the digit is displayed dictates whether it is a multiple of 10, 100, 1000, 10,000, and so there is no need to store unnecessary data for the 0’s. An example can be seen below:

Binary value of 142 = 1 0 0 0 1 1 1 0

16 bit BCD value of 142 = 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0

Sectioned out = (0 0 0 0) – thousands, (0 0 0 1) – hundreds, (0 1 0 0) – tens, (0 0 1 0) – units

As the number does not reach 1000, the first set of bits equal 0. The second 4 bit set represent the hundreds, and as the number is in the region of 100, the 4bits will equal 1. The third set of 4bits equal the tens, so to represent 40 the 4bits equal 4. The final set of bits represent the single units, this time being 2, so the binary value for these 4 bits equals 2.

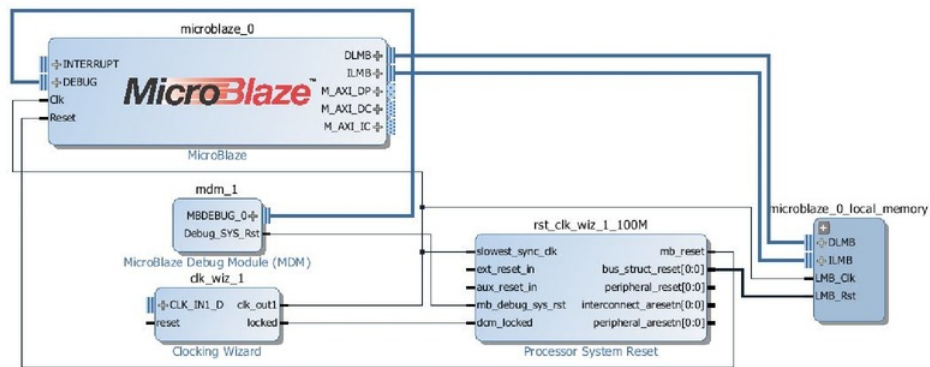
(Jones, 1998)

Very good detail shown

1.4 IP's in Vivado Block Design

Vivado's methods of reusing older VHDL files come in the form of intellectual properties, or IP's for short. This is done using the Vivado IP integrator. This is the process of taking a design, and inserting it into a block design canvas. These designs are typically generated at interface level but can be manipulated at port level.

The purpose of the IP is so that programs can be scaled down and code can be reused in different programs with VHDL files being adapted for new incoming code. The interface used for this system is the block design system. An example of this system can be seen below:



(Digilent: A National Instruments Company)

Each VHDL has been packaged as an IP and imported into the system as a system module. The modules are then interconnected to form a circuit. It is also capable to connect inputs and outputs into the system that can be mapped to external ports on an FPGA, say for example a screen or buttons. This is done by editing the constraints in the XDC file. A constraints file is a method of telling the block design that there are physical inputs from the real world that will affect how the program performs. These physical usually have effects that would be highs or lows in simulation. They can also be displays such as a 7 segment display that is an output to the system.

(Churiwala, 2016) (Digilent: A National Instruments Company)

Good detail

2. Design of Assignment Requirements

2.1 6 Bit Binary to BCD Converter

The first practical part of this assignment was to design and simulate a VHDL file using the Xilinx Vivado design suite. The file would be a 6 bit binary to BCD converter that would take an inputted binary value and convert it to a BCD value. When setting up the file the Artix-7 family was selected and the part named “xc7a100tcsq324” is the on-board FPGA for the Nexys 4 DDR board, so that was chosen in the setup wizard. The language was also changed from Verilog to VHDL as a VHDL file was being created. The file was also placed in a suitable directory. *Good*

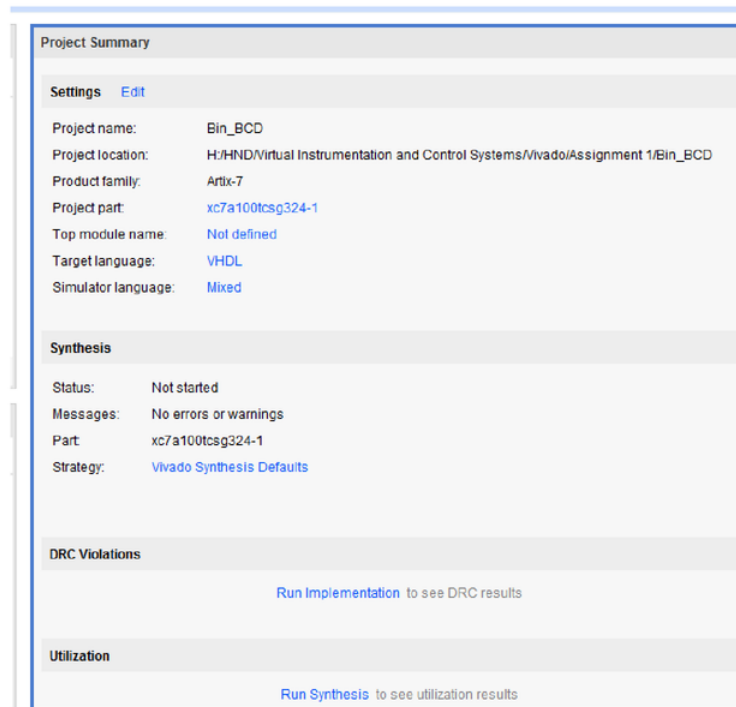


Figure 1 - Project Summary

Once the project had been created the program was then written. There were two programs, the VHDL file and the test bench file. The VHDL file is for the main program while the test bench file is for simulation purposes on the software prior to it being loaded onto an external device. An explanation of each piece of code from each file can be seen below.

2.1.1 VHDL File

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Bin_BCD is
4      Port ( A,B,C,D,E : in STD_LOGIC;
5            Y1,Y2,Y3,Y4,Y5,Y6 : out STD_LOGIC);
6
7  end Bin_BCD;
8  architecture Behavioral of Bin_BCD is
9      signal Inp : std_logic_vector (4 downto 0);
10     signal Q: std_logic_vector (5 downto 0);
11     begin
12         Inp(0) <= A;
13         Inp(1) <= B;
14         Inp(2) <= C;
15         Inp(3) <= D;
16         Inp(4) <= E;
17     process (Inp)
18     begin
19         case Inp is
20         WHEN "00000" => -- 0/1
21             Q <= ("000000");
22         WHEN "00001" => -- 2/3
23             Q <= ("000001");
24         WHEN "00010" => -- 4/5
25             Q <= ("000010");
26         WHEN "00011" => -- 6/7
27             Q <= ("000011");
28         WHEN "00100" => -- 8/9
29             Q <= ("000100");
30         WHEN "00101" => -- 10/11
31             Q <= ("001000");
32         WHEN "00110" => -- 12/13
33             Q <= ("001001");
34         WHEN "00111" => -- 14/15
35             Q <= ("001010");
36         WHEN "01000" => -- 16/17
37             Q <= ("001010");
38         WHEN "01001" => -- 18/19
39             Q <= ("001100");
40         WHEN "01010" => -- 20/21
41             Q <= ("010000");
42         WHEN "01011" => -- 22/23
43             Q <= ("010001");
44         WHEN "01100" => -- 24/25
45             Q <= ("010010");
```

Figure 2 - VHDL File Code 1

The image to the left here shows each line of code from the VHDL file. The first five lines Set the entity diagram for the file and outline all ports that will be used, the inputs being A to E, and the Outputs being Y1 to Y6.

The next step is to determine the architecture for the file. This is done from line 8 onwards where all the inputs from 0/1 to 63/64 are inputted into the system. This is set so that the input signal is set from 4 – 0 and Q is set from 5 – 0 with Q5 being the most significant figure of Y6 and Q0 being the least significant figure of Y1. Each set of numbers is paired as the least significant figure can determine which number of that pair it can be, by changing the least significant figure to either a 1 or 0.

Good detail

```

46 WHEN "01101" => -- 26/27
47   Q <= ("010011");
48 WHEN "01110" => -- 28/29
49   Q <= ("010100");
50 WHEN "01111" => -- 30/31
51   Q <= ("011000");
52 WHEN "10000" => -- 32/33
53   Q <= ("011001");
54 WHEN "10001" => -- 34/35
55   Q <= ("011010");
56 WHEN "10010" => -- 36/37
57   Q <= ("011011");
58 WHEN "10011" => -- 38/39
59   Q <= ("011100");
60 WHEN "10100" => -- 40/41
61   Q <= ("100000");
62 WHEN "10101" => -- 42/43
63   Q <= ("100001");
64 WHEN "10110" => -- 44/45
65   Q <= ("100010");
66 WHEN "10111" => -- 46/47
67   Q <= ("100011");
68 WHEN "11000" => -- 48/49
69   Q <= ("100100");
70 WHEN "11001" => -- 50/51
71   Q <= ("101000");
72 WHEN "11010" => -- 52/53
73   Q <= ("101001");
74 WHEN "11011" => -- 54/55
75   Q <= ("101010");
76 WHEN "11100" => -- 56/57
77   Q <= ("101011");
78 WHEN "11101" => -- 58/59
79   Q <= ("101100");
80 WHEN "11110" => -- 60/61
81   Q <= ("110000");
82 WHEN "11111" => -- 62/63
83   Q <= ("110001");
84 WHEN OTHERS =>
85   Q <= ("000000");
86 END CASE;
87 END PROCESS;
88
89 Y1 <= Q(0);
90 Y2 <= Q(1);
91 Y3 <= Q(2);
92 Y4 <= Q(3);
93 Y5 <= Q(4);
94 Y6 <= Q(5);
95
96 end Behavioral;

```

Figure 3 - VHDL File Code 2

The code to the left demonstrated the continuation of all pairs of binary inputted numbers up until 63. Line 84 shows that when anything else is inputted other than a binary value displayed above, the program will display the binary value of "Q = 0" shown in line 85.

The code for Q0 = the least significant figure and Q5 = the most significant figure can be seen on lines 89 – 94. The behavioural is then ended.

*Ag
ai
n,
go
od
de
tail*

2.1.2 VHDL Testbench (Simulation) File

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Bin_BCD_tb is
4  -- Port ( );
5  end Bin_BCD_tb;
6  architecture Behavioral of Bin_BCD_tb is
7  -- Component Declaration for the Unit Under Test (UUT)
8      COMPONENT Bin_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;
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: Bin_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  
```

Figure 4 - Testbench File Code 1

The test bench file begins in similar fashion to the VHDL, setting up the entity diagram and architecture as well as all the same input and output ports.

Lines 14 to 19 are where the simulated input binary values were inputted in order to test that the file converts the binary values to BCD values. Signal A would be the least significant and signal E would be the most.

Lines 27 to 32 display the clock time periods for each signal. A being the least significant figure would have 1 microsecond, and E would have 16 microseconds.

Good

```

49  -- Clock process definitions
50  A_process :process
51  begin
52  A <= '0';
53  wait for A_period/2;
54  A <= '1';
55  wait for A_period/2;
56  end process;
57  B_process :process
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 5 - Testbench File Code 2

Good understanding shown of how the test bench file works

The remainder of the lines of code in the test bench file are used for the clock process definitions. This is where a clock is created in the program with a 50% duty cycle that generates two values over a set time period

Good detail

An example of this working can be seen from lines 52 to 56. Line 52 sets A to 0, then line 53 tells the program to wait for 0.5 us. Then line 54 tells the program to switch A to 1, then line 55 tells the program to wait for another 0.5 us. This equates to 1 us overall which is what A_period was set to on line 28.

This means that as E_period is set to 16 us, E process will have each value of 0 and 1 displayed for 8 us respectively.

Once this occurs for all processes A, B, C, D & E the program ends.

2.2 Analysis of 6 Bit Binary to BCD Simulation Results

The test bench file was then simulated with 3 different binary values in order to see if the file would convert then from a binary value to a BCD value. The graphs accompanying the conversion show the simulated conversion. It shows that a high = 1 and a low = 0. These are shown for each test.

The first test for the file was for 6/7. It can be seen from figure 6 below that the binary value for 6/7 is displayed from A to E and converted to the BCD value from Y1 to Y6 successfully.

Binary value of: 0 0 0 1 1 0/1

BCD value of: 0 0 0 0 1 1 0/1



Figure 6 - 6 Bit Sim 1

Good analysis

The second test for the file was for 34/35. It can be seen in figure 7 that A – E displays the Binary value, and Y1 – Y6 displays the converted BCD value correctly.

Binary value of: 1 0 0 0 1 0/1

BCD value of: 0 1 1 0 1 0 0/1

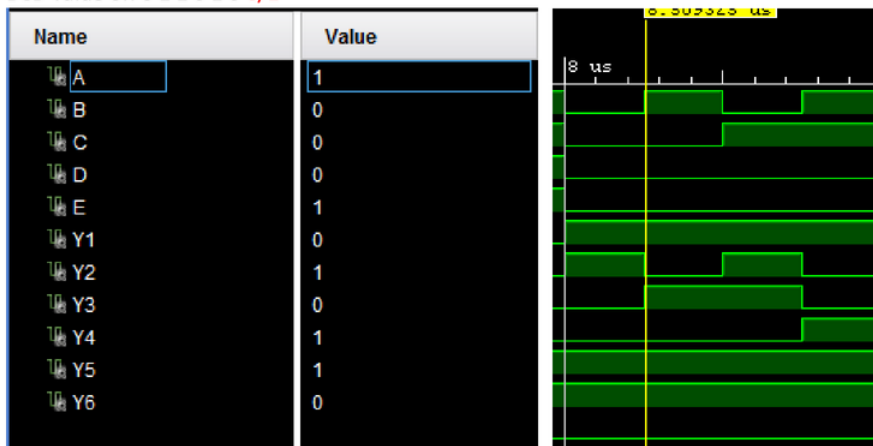


Figure 7 - 6 Bit Sim 2

Good analysis

The third test was for 56/57. Figure 8 shows the inputted Binary value from A – E, and the outputted BCD value from Y1 to Y6 displaying 56/57 correctly.

Binary value of: 1 1 1 0 0 0/1

BCD value of: 1 0 1 0 1 1 0/1



Figure 8 - 6 Bit Sim 3

Good detail shown

2.3 Adapting Design for an 8 Bit Converter

Now that this VHDL file had been tested and worked successfully it needed to be adjusted to convert 8 bit binary values instead of 6. This meant adjusting the inputs and outputs in the code to accommodate the new extended values. B0 – B7 was now used for the 8 bit binary value, with SW15 being the switch that could be manually used to determine whether the least significant value is either 1 or 0. The BCD outputs were also altered to give a better depiction of the BCD value in the simulation screen. Un0, Un1, Un2 & Un3 would be used to display the values of 1, 2, 4, 8 respectively. T0, T1, T2 & T3 would be used to display the same singular values, but defined in their multiples of 10 counterparts. H0 & H1 would be used to display the values 1 and 2 but would be used to define the hundreds values. The simulated values would be tested using lines 15 to 22 similar to the 6 bit test, but with the SW15 value being inputted on line 24 this time.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Bin_BCD_8bit_tb is
4  -- Port ( );
5  end Bin_BCD_8bit_tb;
6  architecture Behavioral of Bin_BCD_8bit_tb is
7  -- Component Declaration for the Unit Under Test (UUT)
8  COMPONENT Bin_BCD_8Bit_wrapper
9  Port ( B0,B1,B2,B3,B4,B5,B6,B7,SW15 : in std_logic;
10 Un0,Un1,Un2,Un3,T0,T1,T2,T3,H0,H1 : out std_logic
11 );
12 END COMPONENT;
13
14 --Inputs
15 signal B0 : std_logic := '0';
16 signal B1 : std_logic := '0';
17 signal B2 : std_logic := '0';
18 signal B3 : std_logic := '1';
19 signal B4 : std_logic := '0';
20 signal B5 : std_logic := '1';
21 signal B6 : std_logic := '1';
22 signal B7 : std_logic := '1';
23 -- First test @ 232
24 signal SW15 : std_logic := '0';
25
26 --Outputs
27 signal Un0 : std_logic;
28 signal Un1 : std_logic;
29 signal Un2 : std_logic;
30 signal Un3 : std_logic;
31 signal T0 : std_logic;
32 signal T1 : std_logic;
33 signal T2 : std_logic;
34 signal T3 : std_logic;
```

Figure 9 - 8 Bit Converter Code

Where is the 8 bit IP block design? How was it obtained from the 6 bit design?

3. Analysis of Xilinx Vivado Simulated Results

The new test bench file was simulated and tested for 3 different 8bit binary values in order to see if the file would correctly convert them into BCD values.

The first test was for the value of 232. Figure 10 shows the value being inputted into the test bench file with B7 being the least significant figure and B0 being the most. Figure 11 shows the successful conversion of 232 going from the binary value to a BCD. H0 – H1 display the value of 2 (200), T0 – T3 display the value of 3 (30), and Un0 – Un3 display the value of 2, showing collectively the value 232.

```

12 |      END COMPONENT;
13 |
14 |      --Inputs
15 |      signal B0 : std_logic := '0';
16 |      signal B1 : std_logic := '0';
17 |      signal B2 : std_logic := '0';
18 |      signal B3 : std_logic := '1';
19 |      signal B4 : std_logic := '0';
20 |      signal B5 : std_logic := '1';
21 |      signal B6 : std_logic := '1';
22 |      signal B7 : std_logic := '1';
23 |      -- First test @ 232
24 |      signal SW15 : std_logic := '0';
25 |

```

Figure 10 - 232 Binary Value

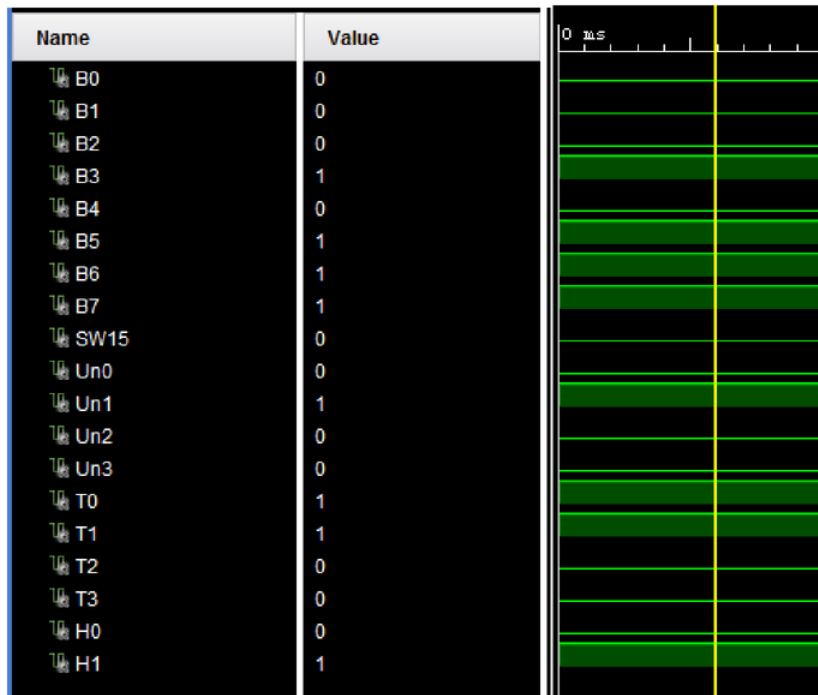


Figure 11 - 232 Binary to BCD Conversion

The second test performed was for the value of 119. Figure 12 shows the value being inputted into the test bench file. Figure 13 shows the successful conversion of the 119 binary value to a BCD value. H0 – H1 displayed 1 (100), T0 – T3 displayed 1 (10) and Un0 – Un3 displayed 9, collectively displaying the value of 119.

Good detail

```

○ wait for 1ms;
  -- Second test @ 119
○ B0 <= '1';
○ B1 <= '1';
○ B2 <= '1';
○ B3 <= '0';
○ B4 <= '1';
○ B5 <= '1';
○ B6 <= '1';
○ B7 <= '0';
○ wait for 1ms;

```

Figure 12 - 119 Binary Value

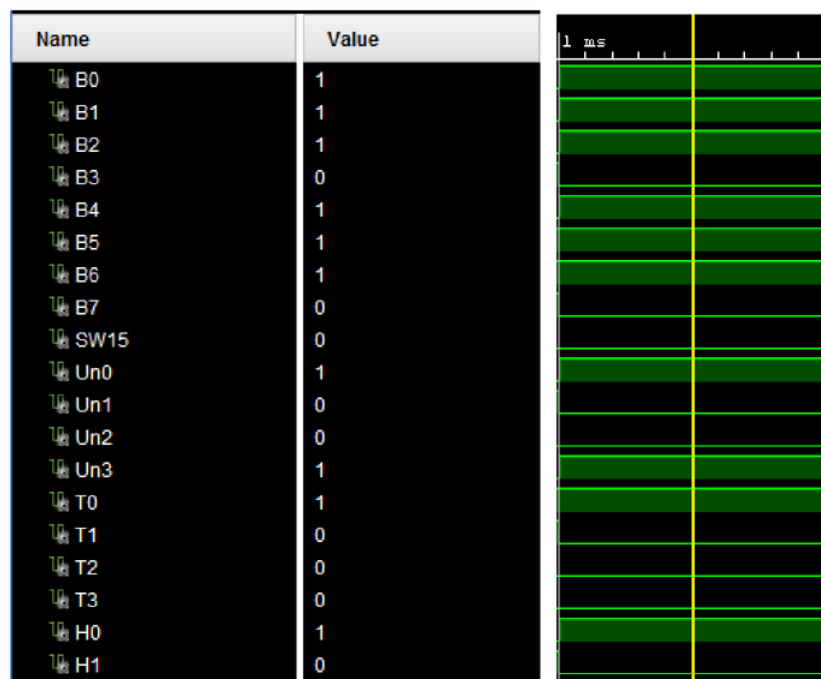


Figure 13 - 119 Binary to BCD Conversion

The third and final test was for the value of 43. Figure 14 shows the binary value inputted into the test bench file. Figure 15 shows the correct conversion of the binary value to a BCD value. H0 – H1 displays 0 as there is no hundreds value for this binary input, T0 – T3 displays the value of 4 (40), and Un0 – Un3 displays the value of 3, collectively displaying the BCD converted value for 43.

```

82      -- Third test @ 43
83      B0 <= '1';
84      B1 <= '1';
85      B2 <= '0';
86      B3 <= '1';
87      B4 <= '0';
88      B5 <= '1';
89      B6 <= '0';
90      B7 <= '0';

```

Good analysis given

Figure 14 - 43 Binary Value

Name	Value	2 ms
B0	1	
B1	1	
B2	0	
B3	1	
B4	0	
B5	1	
B6	0	
B7	0	
SW15	0	
Un0	1	
Un1	1	
Un2	0	
Un3	0	
T0	0	
T1	0	
T2	1	
T3	0	
H0	0	
H1	0	

Figure 15 - 43 Binary to BCD Conversion

4. Analysis of Nexys4 FPGA Practical Results

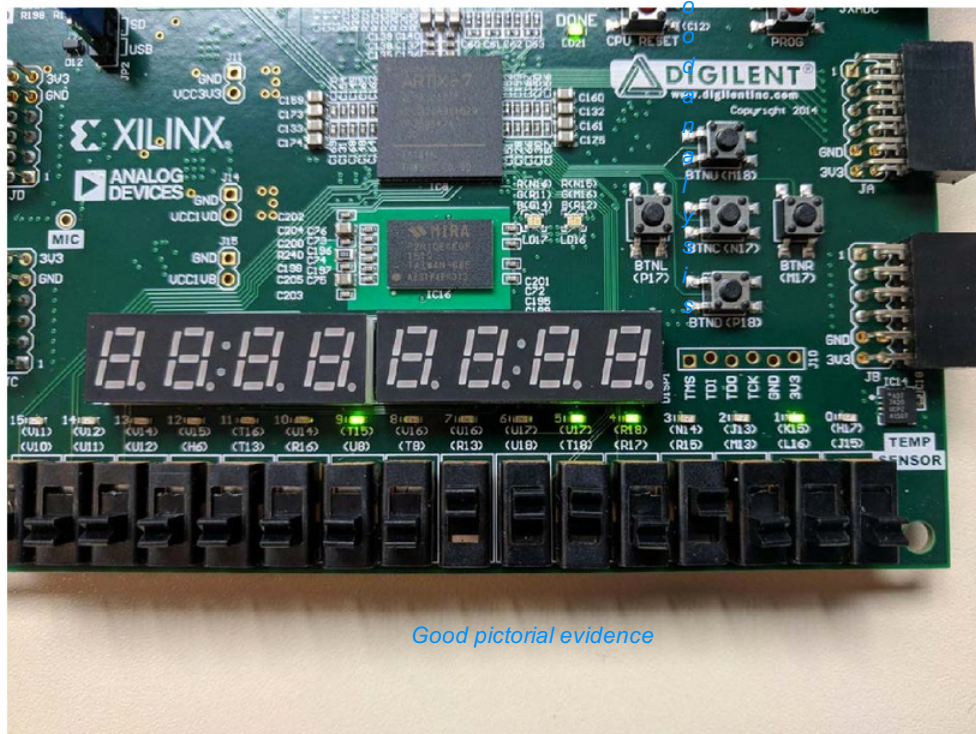
Now that the file had been simulated using the test bench file and proven to work, it was time to download the VHDL file onto the Nexys4 FPGA board and test it practically. In order to do this the program had to run an implementation as well as generate the bit stream. Once these two tasks had been done the hardware manager could be opened and the file downloaded onto the FPGA board. The board would display values running from Un0 all the way to H1, with the least significant character being on the farthest right LED0, and the most significant character being the furthest left LED9 being used in the system. The Binary value would then be inputted onto the switches SW0 – SW7 and if the VHDL file performed as expected, the board's LEDs would display the correct BCD value.

The first test was to see if the board would once again display the value to 232.

The binary value of: $0\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0 = 232$ was entered into the switches and it displayed the correct BCD value on the LEDs of: $1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0$

To break up and check: $(1\ 0) = 200$ $(0\ 0\ 1\ 1) = 30$ $(0\ 0\ 1\ 0) = 2$

G



Good pictorial evidence

Figure 16 - 232 BCD FPGA Value

The second test was for the binary value of 119.

Once again the binary value of: $0\ 1\ 1\ 1\ 0\ 1\ 1\ 1 = 119$ was entered onto the switches of the FPGA board once more, and it displayed the correct BCD value on the LEDs of: $0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1$

To break up and check: $(0\ 1) = 100$ $(0\ 0\ 0\ 1) = 10$ $(1\ 0\ 0\ 1) = 9$

Go
od

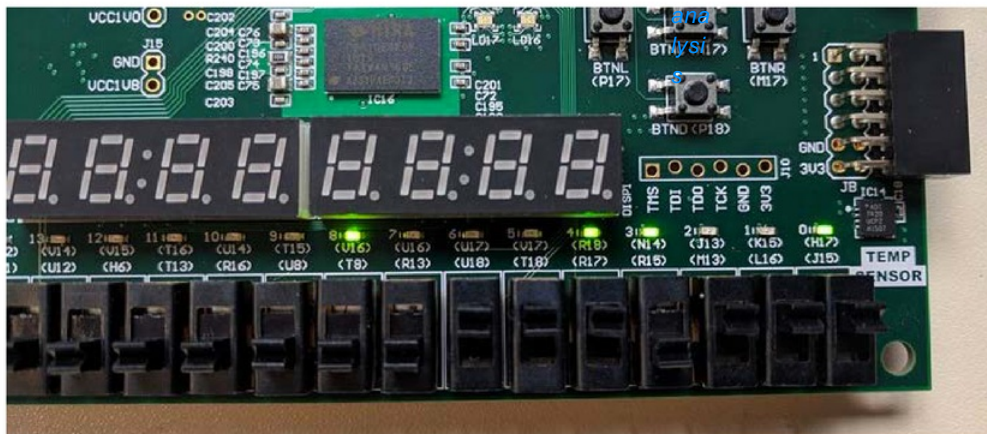


Figure 17 - 119 BCD FPGA Value

The third and Final test was for the binary value of 43.

Once again the value of: $0\ 0\ 1\ 0\ 1\ 0\ 1\ 1 = 43$ was entered onto the switches of the FPGA board, and it displayed the correct BCD Value on the LEDs of: $0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1$

To break up and check: $(0\ 0) = 0$ $(0\ 1\ 0\ 0) = 40$ $(0\ 0\ 1\ 1) = 3$

Good
analysis

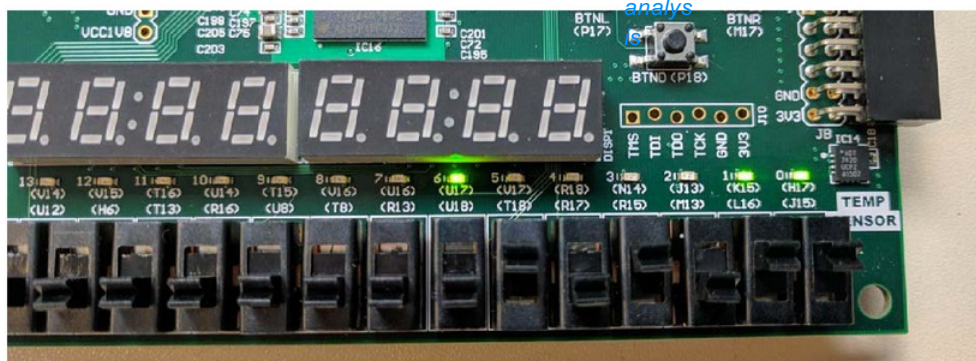


Figure 18 - 43 BCD FPGA Value

5. Adapting Design to Suit New Purposes

Now the original VHDL file needed to be adapted to output a result of 0 – 255 representing the 8 bit binary range to 3 7 segment displays. This meant that not only would the 8 Bit Binary to BCD converter VHDL would be used, but it would need to be packaged into an IP and combined with 2 other VHDL files packaged into IPs:

- Five KHz Clock Pulse (using the internal 100MHz clock on the FPGA board)
- 4 Bit Multiplexer
- BCD to 7 Segment display

These would all need to be combined in a test block design before they could be loaded onto the Nexys4 FPGA board.

5.1 Five KHz Clock Design

The 5KHz clock was the first VHDL file to be created for this new adapted design. It used the same Artix 7 family and part "xc7a100tcs324". The language was also changed from verilog to VHDL and the project was saved in the same directory as the Binary to BCD converter project.

5.1.1 Five KHz Clock VHDL File

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.std_logic_unsigned.all;
5
6  entity FiveKHz_Clock is
7
8      Port ( CLK100MHZ : in    STD_LOGIC;
9            Rst       : in    std_logic;
10           FiveKHz   : out   STD_LOGIC
11
12           );
13
14 end FiveKHz_Clock;
15
16 architecture Behavioral of FiveKHz_Clock is
17
18     signal cnt1      : integer := 0;
19
20     signal temp1     : std_logic := '0';
21
22     begin
23
24
25
26     process (CLK100MHZ,Rst)
27     begin
28         if Rst = '1' then
29             temp1 <= '0';
30             cnt1 <= 0;
31         elsif rising_edge (CLK100MHZ) then
32             if(cnt1= 10000) then
33                 temp1 <= not temp1;
34                 cnt1 <= 0;
35             else
36                 cnt1 <= cnt1 +1 ;
37             end if;
38         end if;
39
40     end process;
41     FiveKHz <= temp1;
42
43 end Behavioral;
```

The code begins by setting the entity for the project with port having two inputs of the 100 MHz clock, and a reset function, and an output of the 5 KHz clock pulse.

The architecture is then set for file, setting the count to an integer of 0, and the signal temp to a standard logic of 0.

The process determines how the clock will work. The process begins by looking to see if reset has been set to 1. If so then both temp and count are set to 0 as the clock has been reset. However if the program detects a rising edge from the 100 MHz clock, a new if statement will be assessed. This statement will see if the count has reached 10,000. As this is the limit of the count, the count resets 0. The else statement here would be for when the clock is at any number other than 10,000, and so would instead increment the count by 1.

Figure 19 - Five KHz Clock VHDL Code

Why 10000? Where is the maths?

5.1.2 Five KHz Clock Test Bench (Simulation) File

```

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
6
7  entity FiveKHz_Clock_tb is
8  -- Port ( );
9  end FiveKHz_Clock_tb;
10
11  architecture Behavioral of FiveKHz_Clock_tb is
12
13  Component FiveKHz_Clock
14  Port ( CLK100MHZ      : in  STD_LOGIC;
15        Rst             : in  STD_LOGIC;
16        FiveKHz         : out STD_LOGIC
17
18        );
19  end Component;
20
21  --Inputs
22  signal CLK100MHZ      : STD_LOGIC := '0';
23  signal Rst            : STD_LOGIC := '1';
24
25  --Outputs
26  signal FiveKHz        : STD_LOGIC;
27
28
29  --Clock definitions
30
31  constant CLK100MHZ_period : time := 10ns;
32
33  begin
34
35  --Instantiate the UUT
36
37  UUT : FiveKHz_Clock
38
39  Port Map ( CLK100MHZ      => CLK100MHZ,
40            Rst             => Rst,
41            FiveKHz         => FiveKHz
42
43            );
44  --Clock Process definitions
45
46  CLK100MHZ_process :process
47  begin
48  CLK100MHZ <= '0';
49  wait for CLK100MHZ_period/2;
50  CLK100MHZ <= '1';
51  wait for CLK100MHZ_period/2;
52  end process;

```

Figure 20 - Five KHz Clock Testbench Code 1

The test bench file sets the entity and architectures similar to the VHDL file with the same number of inputs and outputs.

The inputs on lines 21 – 23 are then set to standard logic 0 for the 100 MHz clock and 1 for the reset.

The output on line 26 set the 5 KHz clock to just standard logic.

The clock period is also set to 10 nanoseconds on line 31.

The clock process definitions are then set on lines 44 to 51. Similar to the processes on the Binary to BCD converter, the clock gets set to 0 for half the period, which in this case is half of 10 nanoseconds equalling 5 nanoseconds. It is then set to 1 for the other 5 nanoseconds. This process repeats.

G
o
o
d

```

53 |
54 | --Stimulus process definitions
55 |
56 | stim_proc: process
57 |
58 |     begin
59 |         wait for 1ms;
60 |
61 |         Rst    <= '0';
62 |         wait for 2ms;
63 |         Rst    <= '1';
64 |
65 |         wait;
66 |     end process;
67 |
68 | end Behavioral;

```

The last few lines of the testbench code for the five KHz clock are used to set the stimulus process definitions. It begins with the clock waiting for 1ms, before setting reset to 0, then waiting for 2ms before setting reset to 1. The program then waits again before ending the process.

Good

Figure 21 - Five KHz Clock Testbench Code 2

5.2 Multiplexer Design.

The purpose of the Multiplexer was to take the BCD converted values from the binary inputs and carries them through a multiplexing system to the 7-segment display as a single line input. The VHDL used the same Artix 7 family component and part "xc7a100tcsg324". The language was once again changed from Verilog to VHDL and the file was saved in a similar directory to the previous projects.

5.2.1 Multiplexer VHDL File

```
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 Mux_Demux 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 Mux_Demux;
15 architecture Behavioral of Mux_Demux 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
28 process (cnt(1), cnt(0))
29 begin
30   if (cnt(1)='0' and cnt(0)='0') then Q <= "110";
31   elsif (cnt(1)='0' and cnt(0)='1') then Q <= "101";
32   elsif (cnt(1)='1' and cnt(0)='0') then Q <= "011";
33   else Q <= "111";
34   end if;
35 end process;
36
37 AN0 <= Q(0);
38 AN1 <= Q(1);
39 AN2 <= Q(2);
```

Figure 22 - Multiplexer VHDL Code 1

The code begins by setting the entity of the file. This VHDL file has 12 inputs and 7 outputs. The inputs are set for the Five KHz in standard logic, the Reset, Un0 – Un3, T0 – T3 and H0 – H1. The outputs were set for AN0 – AN2 and A – D.

The architecture is then set from line 16 to line 18.

The reset is set similar to the other files in lines 23 – 27 where it resets to 11, or continues the increment in steps of 1 in the else function.

The process for the units, tens and hundreds is then set from line 30, through to the end of the file at line 82. This is a series of processes where the multiplexer takes the BCD converted values from the 3 Binary to BCD converters that would be in the block design and converts them into one single line of code to be displayed by the 7-segment display.

Not clearly understood

```

44
45 process (Un0,Un1,Un2,Un3,T0,T1,T2,T3,H0,H1,Q(0),Q(1),Q(2))
46 begin
47
48 if (Q(0) = '0' and Un0 = '0') then D<='0';
49 elsif (Q(0) = '0' and Un0 = '1') then D<='1';
50 elsif (Q(1) = '0' and T0 = '0') then D<='0';
51 elsif (Q(1) = '0' and T0 = '1') then D<='1';
52 elsif (Q(2) = '0' and H0 = '0') then D<='0';
53 elsif (Q(2) = '0' and H0 = '1') then D<='1';
54
55 end if;
56
57 if (Q(0) = '0' and Un1 = '0') then C<='0';
58 elsif (Q(0) = '0' and Un1 = '1') then C<='1';
59 elsif (Q(1) = '0' and T1 = '0') then C<='0';
60 elsif (Q(1) = '0' and T1 = '1') then C<='1';
61 elsif (Q(2) = '0' and H1 = '0') then C<='0';
62 elsif (Q(2) = '0' and H1 = '1') then C<='1';
63
64 end if;
65
66 if (Q(0) = '0' and Un2 = '0') then B<='0';
67 elsif (Q(0) = '0' and Un2 = '1') then B<='1';
68 elsif (Q(1) = '0' and T2 = '0') then B<='0';
69 elsif (Q(1) = '0' and T2 = '1') then B<='1';
70 elsif (Q(2) = '0') then B<='0';
71
72 end if;
73
74 if (Q(0) = '0' and Un3 = '0') then A<='0';
75 elsif (Q(0) = '0' and Un3 = '1') then A<='1';
76 elsif (Q(1) = '0' and T3 = '0') then A<='0';
77 elsif (Q(1) = '0' and T3 = '1') then A<='1';
78 elsif (Q(2) = '0') then A<='0';
79
80 end if;
81 end process;
82 end Behavioral;

```

Figure 23 - Multiplexer VHDL Code 2

5.2.2 Multiplexer Test Bench (Simulation) File

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity Mux_Demux_tb is
4  -- Port ( ); No ned to add info here as this is
5  end Mux_Demux_tb;
6  architecture Behavioral of Mux_Demux_tb is
7  component Mux_Demux -- 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 := '1';
24  signal Un2 : std_logic := '0';
25  signal Un3 : std_logic := '0';
26  signal T0 : std_logic := '1';
27  signal T1 : std_logic := '1';
28  signal T2 : std_logic := '1';
29  signal T3 : std_logic := '0';
30  signal H0 : std_logic := '1';
31  signal H1 : std_logic := '0';
32
33  -- output signals
34  -- Example signal Q1 : std_logic := '0';
35
36  signal AN0 : std_logic;
37  signal AN1 : std_logic;
38  signal AN2 : std_logic;
39  signal A : std_logic;
40  signal B : std_logic;
41  signal C : std_logic;
42  signal D : std_logic;
43
44

```

Figure 24 - Multiplexer Testbench Code 1

The test bench file for this program would be used to simulate the units, tens and hundreds of a value being taken through the system over the course of 3ms. For this the BCD value of 173 was inputted on lines 20 – 31. This would then be simulated to run a test to see if the value is displayed correctly at the other end. This can be seen in the simulation testing for the Binary to BCD 7 segment display.

The output signals are all set to standard logic.

Good

```

46 -- clock period definitions
47 -- Example constant A_period : time := 8ns,
48 constant FiveKHz_period : time := 200us;
49 begin
50 -- Instantiate the UUT
51 UUT : Mux_Demux -- Enter file to be tested
52 Port Map (
53 -- Example A => A,
54 FiveKHz => FiveKHz,
55 Rst => Rst,
56 AN0 => AN0,
57 AN1 => AN1,
58 An2 => AN2,
59 Un0 => Un0,
60 Un1 => Un1,
61 Un2 => Un2,
62 Un3 => Un3,
63 T0 => T0,
64 T1 => T1,
65 T2 => T2,
66 T3 => T3,
67 H0 => H0,
68 H1 => H1,
69 A => A,
70 B => B,
71 C => C,
72 D => D
73 );
74
75 -- clock process definitions
76 clk_process : process
77 begin
78 FiveKHz <= '0';
79 wait for FiveKHz_period/2; --for 5 ns signal
80 FiveKHz <= '1';
81 wait for FiveKHz_period/2; --for next 5 ns
82 end process;
83
84 -- Stimulus process
85
86 stim_proc: process
87 begin
88
89 wait for 1ms;
90 Rst <= '0';
91 wait for 5ms;
92 Rst <= '1';
93 wait;
94 end process;
95 end Behavioral;

```

Figure 25 - Multiplexer Testbench Code 2

A constant time period is given to the 5 KHz time period of 200us, which is used in the clock process same as before on lines 76 to 82.

All the ports are mapped to the correct locations on lines 54 to 72.

The stimulus process can then be found on lines 86 to 94 before the end of the test bench file.

5.3 7-Segment Display Design

The 7 segment display is what will give a visual display on the FPGA board of the processed values from the multiplexer. It can display any number between 0-9. For numbers greater than a single digit multiple displays are used. The VHDL used the same Artix 7 family component and part "xc7a100tcs324". The language was once again changed from Verilog to VHDL and the file was saved in a similar directory to the previous projects.

5.3.1 7-Segment Display VHDL File

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity BCD_7seg is
4  Port ( A,B,C,D : in std_logic;
5         sa,sb,sc,sd,se,sf,sg : out std_logic
6  );
7  end BCD_7seg;
8  architecture Behavioral of BCD_7seg is
9  signal Q : std_logic_vector (3 downto 0);
10 signal seg : std_logic_vector (6 downto 0);
11 begin
12 Q(0) <= D;
13 Q(1) <= C;
14 Q(2) <= B;
15 Q(3) <= A;
16 process (Q)
17 begin
18 case Q is
19 WHEN "0000" => -- 0
20 seg <= ("0000001");
21 WHEN "0001" => -- 1
22 seg <= ("1001111");
23 WHEN "0010" => -- 2
24 seg <= ("0010010");
25 WHEN "0011" => -- 3
26 seg <= ("0000110");
27 WHEN "0100" => -- 4
28 seg <= ("1001100");
29 WHEN "0101" => -- 5
30 seg <= ("0100100");
31 WHEN "0110" => -- 6
32 seg <= ("0100000");
33 WHEN "0111" => -- 7
34 seg <= ("0001111");
35 WHEN "1000" => -- 8
36 seg <= ("0000000");
37 WHEN "1001" => -- 9
38 seg <= ("0001100");
39 WHEN Others => -- 'blank'
40 seg <= ("1111111");
41 END CASE;
42 END PROCESS;
43 sa <= seg(6);
44 sb <= seg(5);
45 sc <= seg(4);
46 sd <= seg(3);
47 se <= seg(2);
48 sf <= seg(1);
49 sg <= seg(0);
50 end Behavioral;
```

The first lines of code for this VHDL file once again set the entity for the program and list all the inputs and outputs. In this case the inputs are A, B, C & D and the outputs are sa – sg.

The architecture is then setup. This here shows that signal Q will be the incoming binary signal for a value between 0 – 9, with 3 down to 0 representing the 4 bits of 1, 2, 4 & 8. The signal seg then determines which segments of the 7-segment display will illuminate.

Lines 18 – 41 then convert each binary number into the correct digit displayed by the 7 segments. A 1 represents a segment being off, and a 0 represents a segment being on. There is also a setting for when an unrecognised value is sent to be processed and displayed. This is shown on rails 39 – 40 and in this circumstance the display is set to straight 1's meaning all the segments are off.

G
o
o
d
d
et
ail

Figure 26 - 7-Segment Display VHDL Code

An example of how the display uses this code to process and display each digit can be seen below. Figure 27 shows a 7 segment display with each segment labelled a – g. If the number 3 was too be displayed by the VHDL file, lines 25 – 26 in figure 26 of the code show an input binary value of 0 0 1 1 and an output segment value of 0 0 0 0 1 1 0. The 0's will place segments a, b, c, d & g on, while the 1's will keep segments e & f off. This can be seen in figure 27's centre image. Another example can be the segments displaying the number 4. Lines 27-28 in figure 26 show a binary input value of 0 1 0 0, and an outputted segment value of 1 0 0 1 1 0 0. This means that segments a, d & e are set to 1 and therefore off, and segments b, c, f & g are set to 0 and therefore on, collectively displaying the digit 4. This can also be seen in figure 27's furthest right image.

Clear understanding shown

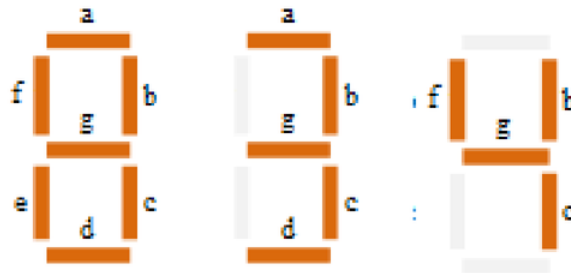


Figure 27 - 7-Segment Display Examples

5.3.2 7-Segment Display Test bench (Simulation) File

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  -- Uncomment the following library declaration if using
4  -- arithmetic functions with Signed or Unsigned values
5  --USE ieee.numeric_std.ALL;
6  ENTITY BCD_7seg_tb IS
7  END BCD_7seg_tb;
8  ARCHITECTURE behavior OF BCD_7seg_tb IS
9  -- Component Declaration for the Unit Under Test (UUT)
10 COMPONENT BCD_7seg
11 PORT(
12   A : IN std_logic;
13   B : IN std_logic;
14   C : IN std_logic;
15   D : IN std_logic;
16   sa : OUT std_logic;
17   sb : OUT std_logic;
18   sc : OUT std_logic;
19   sd : OUT std_logic;
20   se : OUT std_logic;
21   sf : OUT std_logic;
22   sg : OUT std_logic
23 );
24 END COMPONENT;
25 --Inputs
26 signal A : std_logic := '0';
27 signal B : std_logic := '0';
28 signal C : std_logic := '0';
29 signal D : std_logic := '0';
30 --Outputs
31 signal sa : std_logic;
32 signal sb : std_logic;
33 signal sc : std_logic;
34 signal sd : std_logic;
35 signal se : std_logic;
36 signal sf : std_logic;
37 signal sg : std_logic;
38 -- No clocks detected in port list. Replace <clock> below with
39 -- appropriate port name
40 constant A_period : time := 80 ns;
41 constant B_period : time := 40 ns;
42 constant C_period : time := 20 ns;
43 constant D_period : time := 10 ns;
44 BEGIN
45 -- Instantiate the Unit Under Test (UUT)
46 uut: BCD_7seg PORT MAP (

```

Figure 28 - 7-Segment Display Testbench Code 1

The test bench file is also used to simulate the 7 segment display. It begins by setting the files entity and architecture. All of the inputs and outputs are set as standard logic for the unit under test.

The clock processes for this file are set as constants, with A being the most significant figure of 80ns, and D being the least at 10ns.

Good

```

46 uut: BCD_7seg PORT MAP (
47   A => A,
48   B => B,
49   C => C,
50   D => D,
51   sa => sa,
52   sb => sb,
53   sc => sc,
54   sd => sd,
55   se => se,
56   sf => sf,
57   sg => sg
58 );
59 -- Clock process definitions
60 A_process :process
61 begin
62   A <= '0';
63   wait for A_period/2;
64   A <= '1';
65   wait for A_period/2;
66 end process;
67 B_process :process
68 begin
69   B <= '0';
70   wait for B_period/2;
71   B <= '1';
72   wait for B_period/2;
73 end process;
74 C_process :process
75 begin
76   C <= '0';
77   wait for C_period/2;
78   C <= '1';
79   wait for C_period/2;
80 end process;
81 D_process :process
82 begin
83   D <= '0';
84   wait for D_period/2;
85   D <= '1';
86   wait for D_period/2;
87 end process;
88 end behavior;

```

Figure 29 - 7-Segment Display Testbench Code 2

Lines 46 to 58 map all the ports to be put under test by the simulation, these being the inputs and outputs.

Following this are the clock process definitions which work the same as previous clock processes in the other test bench files. These are set for values A, B, C & D.

6. Analysis of Xilinx Vivado Simulated Results

Now that all the VHDL files for each part of the program had been created, their counterpart test bench files needed to be simulated to make sure they could work properly.

6.1. 5 KHz Clock Simulation

The first file put under test was the 5 KHz clock file. The file was simulated over a period of 4ms. The simulation gave the following graph in figure 30. The yellow and white markers demonstrate 1 period of the clock which should equate to 5 KHz.

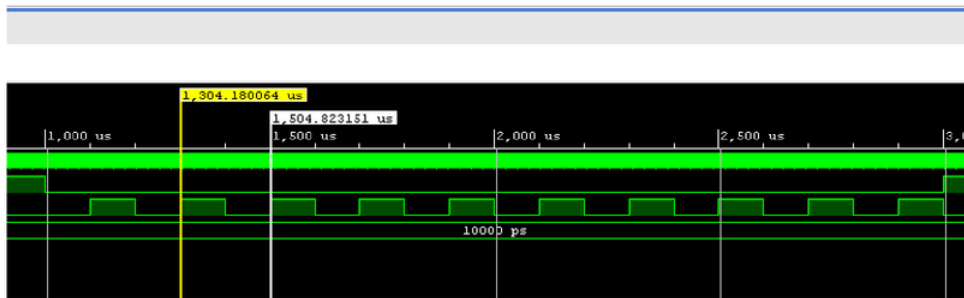


Figure 30 - 5 KHz Clock Simulation

The difference of 200us between the two points in the simulation equates to 1 period. The frequency = $\frac{1}{\text{period}}$ which is $\frac{1}{200 \times 10^{-6}} = 5000 \text{ Hz}$. Therefore the difference between the 2 points equals 5 KHz. This can be seen in figure 31. *Good*

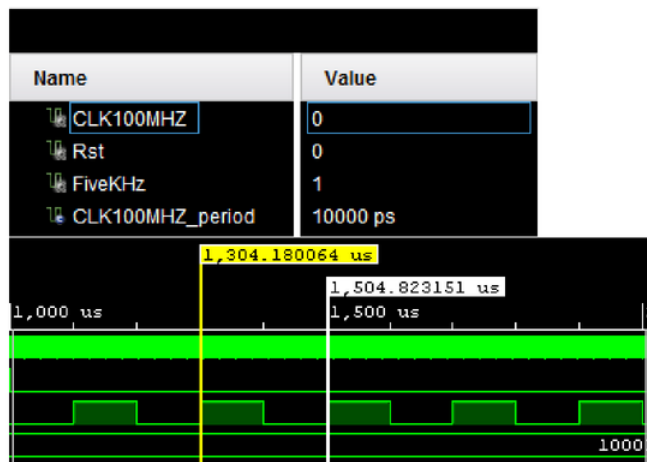


Figure 31 - 1 Period of the Clock

6.2 Multiplexer Simulation

The simulation for the multiplexer needed to take what would be considered a binary value that was now in BCD form, and see if it could be processed by the file.

Lines 20 – 31 were used to set the test bench file to take 173 as a converted BCD value in binary

```
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 := '1';  
24 : signal Un2 : std_logic := '0';  
25 : signal Un3 : std_logic := '0';  
26 : signal T0 : std_logic := '1';  
27 : signal T1 : std_logic := '1';  
28 : signal T2 : std_logic := '1';  
29 : signal T3 : std_logic := '0';  
30 : signal H0 : std_logic := '1';  
31 : signal H1 : std_logic := '0';
```

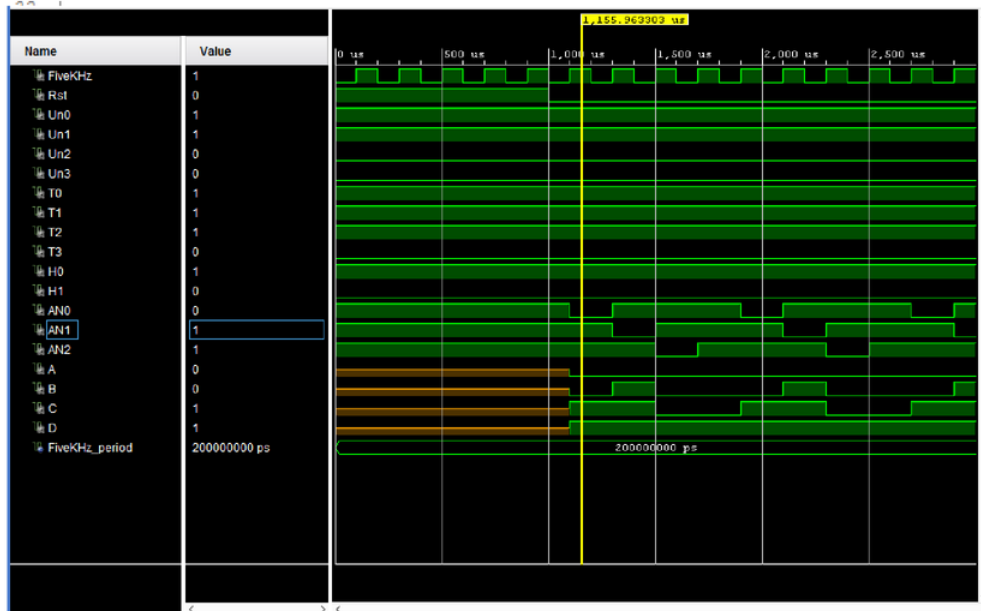


Figure 32 – Multiplexer Testbench Code & Simulation

form. The simulation was then ran over 3ms.

H1-H2, T0-T3 & Un0-Un3 show the binary value being simulated. The A, B, C & D values show the value being sent through in first units, then tens and then hundreds. This first screenshot in figure 33 shows D and C equalling 1, and B and A equalling 0 which displays the unit 3. This can be seen on the graph where C & D line up with the digital highs of 1, and A & B line up with the digital lows of 0. There is also the clock period at the very top of the graph that shows the values lining up with one clock period.

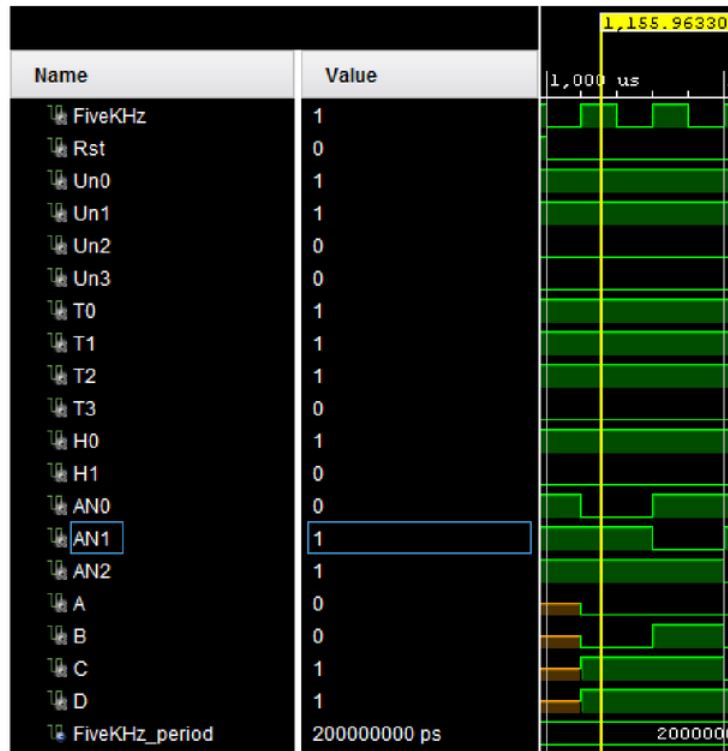


Figure 33 - Multiplexer Simulation Values 1

This second screenshot in figure 34 shows D, C and B equalling 1 and A equalling 0, depicting the tens value of 7 for 1 clock period.

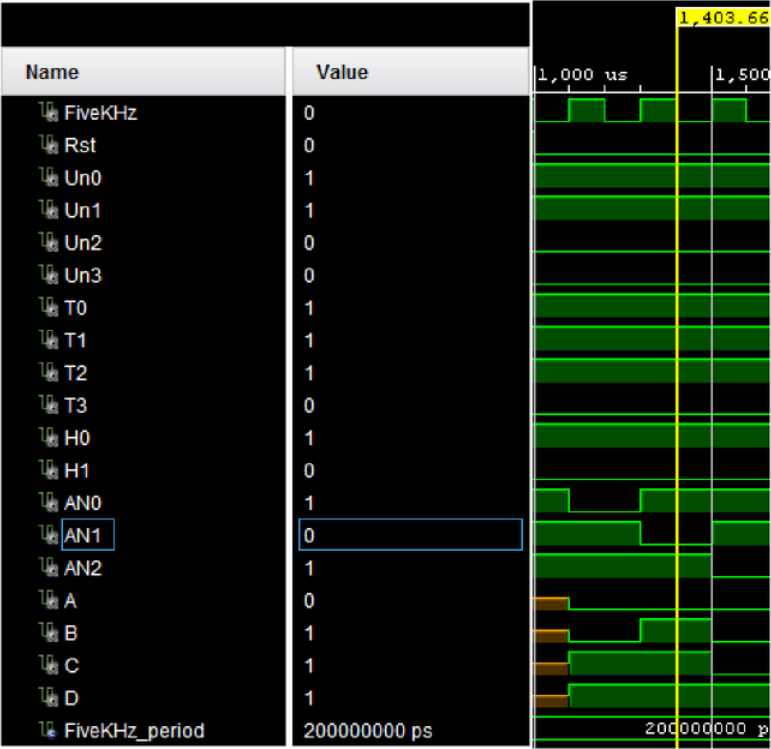


Figure 34 - Multiplexer Simulation Values 2

The third screenshot in figure 35 shows D equalling 1 and C, B and A equalling 0, depicting the hundreds value of 1.

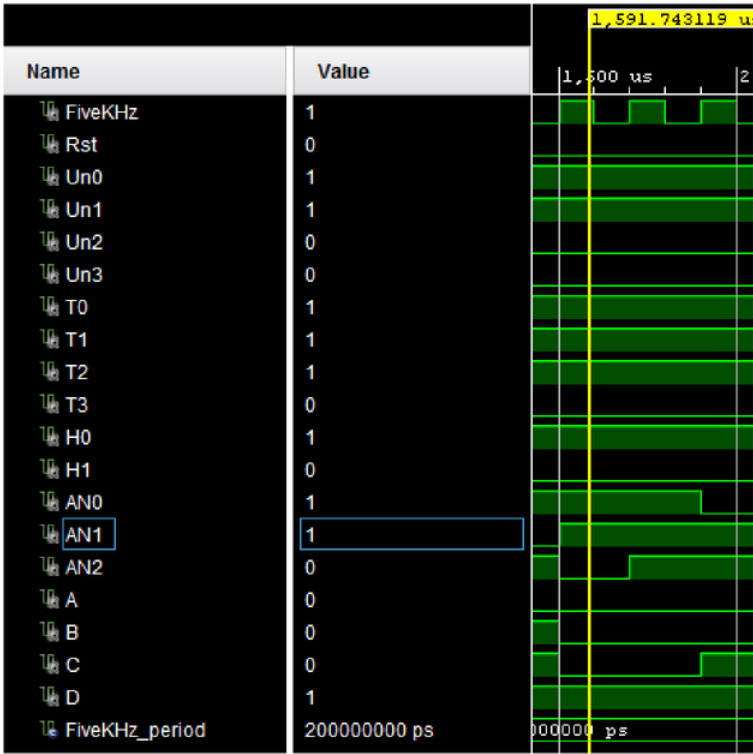


Figure 35 - Multiplexer Simulation Values 3

Very good analysis of 3 sets of simulated tests carried out

6.3. BCD to 7-Segment Display Simulation

In order to test the 7 segment display under simulation, 3 binary values were inputted into the system; 10, 8 and 2. The simulation was then ran to see if the graph would show the correct data based off of the number inputted. In this case A = most sig fig while D = least sig fig.

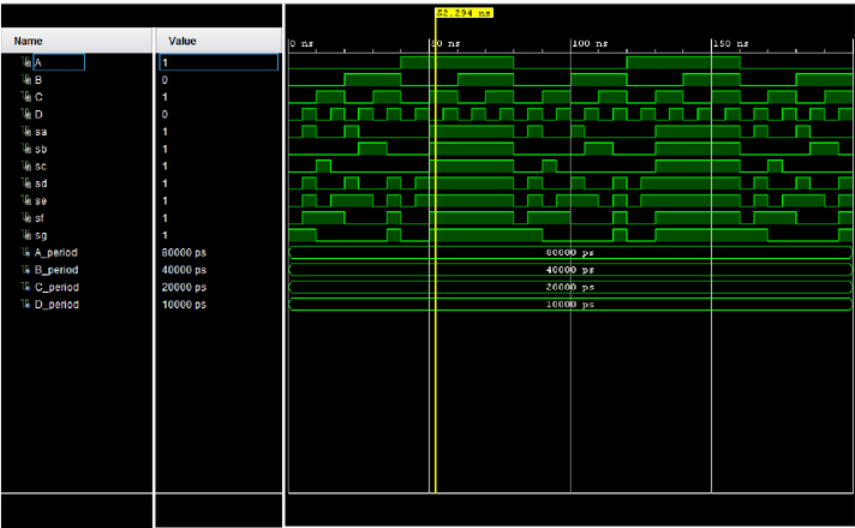


Figure 36 - 7-Segment Display Simulation

The first value enter was the binary value = $8 + 2 = 10$. This was to test if the display would recognise a value that wasn't in the region of 0-9 and therefore not illuminate any of the segments. Figure 37 shows A to D displaying the binary value of 10, and sa to sg showing all segment display values being off as the segment can only display 0 – 9. This is also seen on the graph with digital highs matching

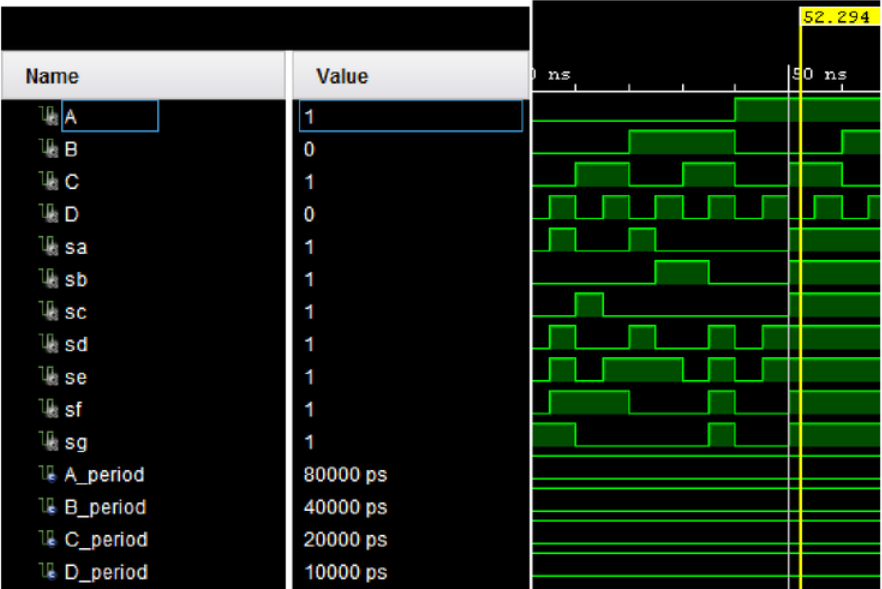


Figure 37 - 7-Segment Display Test Value of 10

up with the 1's, meaning the segments would be off.

The second test performed was for a binary value of 8. This was to see if all segments of the display would illuminate. Figure 38 shows A to D inputting a binary value of 8, and sa to sg all being set to 0 by the program, meaning that all these segments would in turn be on. This shows that the file has

once again performed correctly.

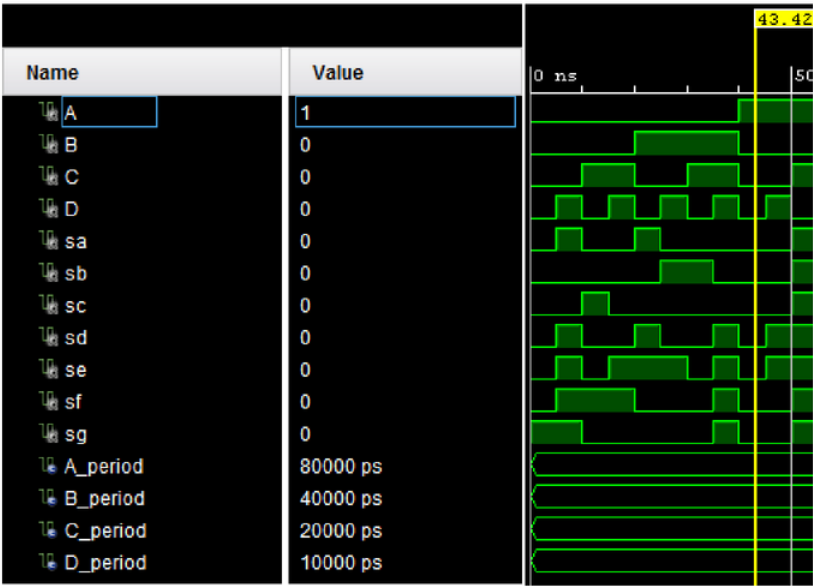


Figure 38 - 7-Segment Display Test Value of 8

Good analysis given

The third test was for a binary value of 2 displayed here in figure 39 by A to D. Sa, sb, sd, se & sg then all display due to being set to 0, while sc & sf don't display due to them being set to 1. This means that segments c and f would not illuminate on the display, therefore the remaining segments would

display the correct value of 2.

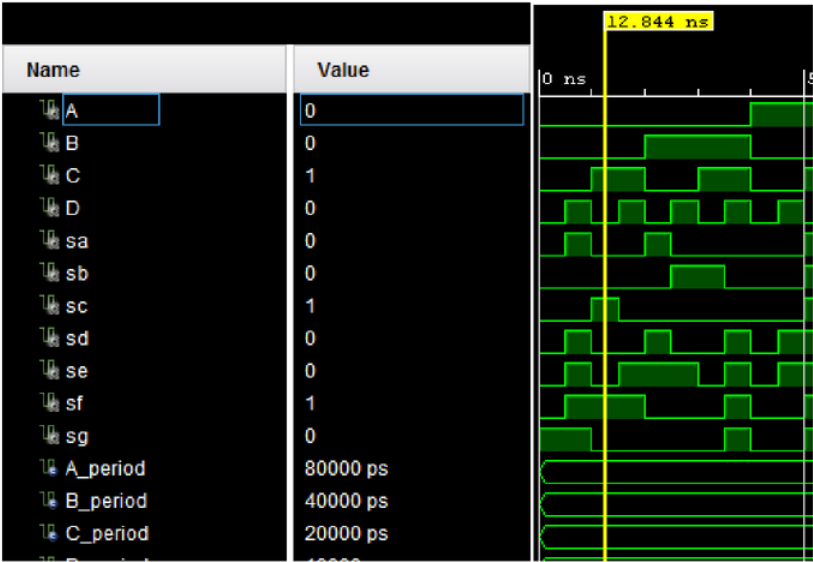


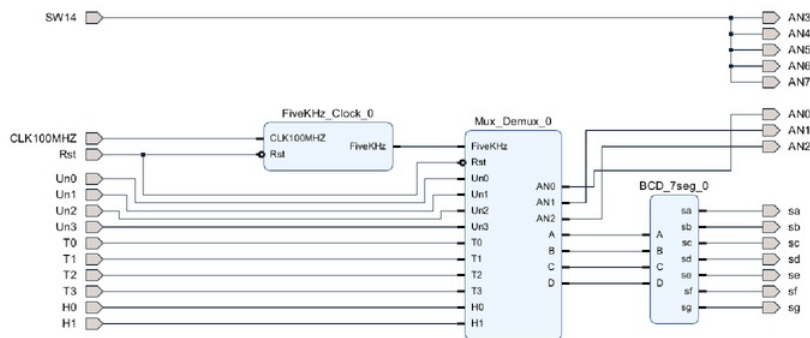
Figure 39 - 7-Segment Display Test Value of 2

Good set of test results analysed correctly

7. Analysis of Nexys 4 DDR FPGA Practical Results

7.1. Building & Testing the First Block Design

Now that all the files had been tested and proven to work, they all needed to be combined together to form the final program of a binary to BCD 7-segment display. This was done by opening up each individual project and packaging them as an IP (Intellectual Property). Once this was done, a new project was opened, once again choosing the Artix-7 family and the part named "xc7a100tcs324". The language was also once again changed from Verilog to VHDL. Now a test block design was opened up and all of the IP's for each VHDL file were imported into this one program. They were then connected together with suitable inputs and outputs. This can be seen in figure 40 below. The initial testing here did not involve the binary to BCD converters as the system needed to be tested just converting BCD values into the 7-segment display before the binary to BCD conversion could be implemented.



Good detail

Figure 40 - First Test Block Design

Constraints for the program were then assigned using a master XDC file. These were mapped for each switch and each segment of the display so that the switches could be used to determine which digits would display on the FPGA. The units, tens and hundreds values were assigned switches so that a manual BCD value could be inputted, and that value could then be displayed on the segments.

```
1 ## Clock signal
2
3 set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform { 0 5 } [get_ports { CLK100MHZ }];
5
6 ## Switches
7
8 set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { Un0 }]; #IO_L24N_T3_R0Q_15 Sch=sv[0]
9 set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { Un1 }]; #IO_L3N_T0_DQS_SMCCLK_14 Sch=sv[1]
10 set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { Un2 }]; #IO_L6N_T0_D08_VREF_14 Sch=sv[2]
11 set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { Un3 }]; #IO_L13N_T2_MRCC_14 Sch=sv[3]
12 set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { T0 }]; #IO_L12N_T1_MRCC_14 Sch=sv[4]
13 set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { T1 }]; #IO_L7N_T1_D10_14 Sch=sv[5]
14 set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { T2 }]; #IO_L17N_T2_A13_D29_14 Sch=sv[6]
15 set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { T3 }]; #IO_L5N_T0_D07_14 Sch=sv[7]
16 set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 } [get_ports { H0 }]; #IO_L24N_T3_34 Sch=sv[8]
17 set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports { H1 }]; #IO_25_34 Sch=sv[9]
18 set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports { SW10 }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sv[10]
19 set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports { SW11 }]; #IO_L23P_T3_A03_D19_14 Sch=sv[11]
20 set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMOS33 } [get_ports { SW12 }]; #IO_L24P_T3_35 Sch=sv[12]
21 set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports { SW13 }]; #IO_L20P_T3_A08_D24_14 Sch=sv[13]
22 set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports { SW14 }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sv[14]
23 set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 } [get_ports { Rat }]; #IO_L21P_T3_DQS_14 Sch=sv[15]
24
25 ## 7 segment display
26
27 set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { sa }]; #IO_L24N_T3_A00_D16_14 Sch=sa
28 set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports { sb }]; #IO_25_14 Sch=cb
```

Figure 41 - Test Block Design Constraints

The synthesis and implementation was then ran, before the bit stream was then generated. After this the program was then downloaded to the Nexys4 FPGA using the hardware manager and 3 tests were performed.

The first test was for a value of 257. This was done by entering the BCD value of 257 on the FPGA board using the switches. It gave the resulting display seen in figure 42 which was correct.

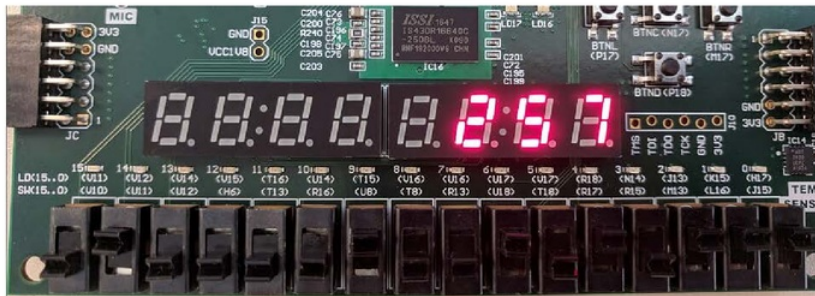


Figure 42 - 257 Test

How do I know the switches are set to 257?

The second test was for a value of 125. This was once again entered on the FPGA switches and gave the following results in figure 43, which is correct.

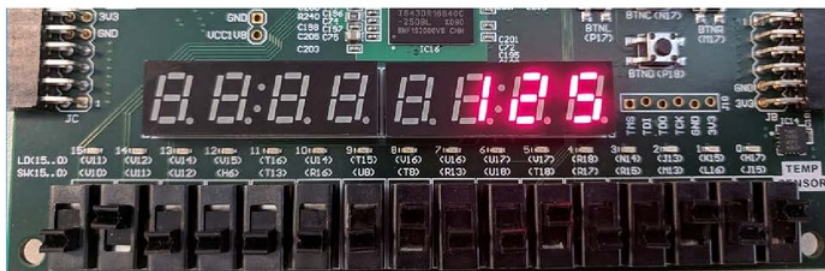


Figure 43 - 125 test

The third and final test was for 39. This gave the final result shown in figure 44 which is correct.

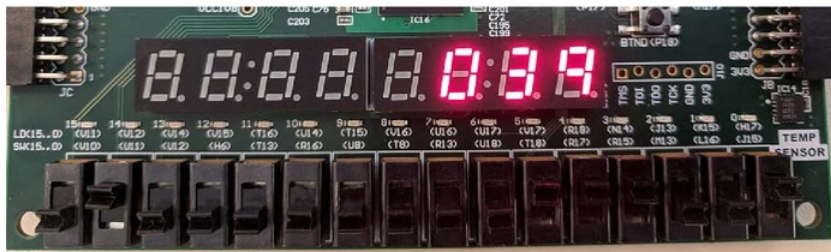
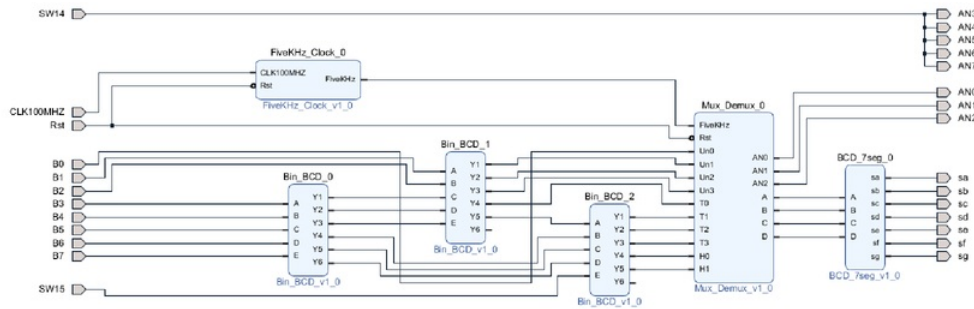


Figure 44 - 39 test

Results are correct but not clearly explained

7.2. Building & Testing the Second Block Design

Now that the BCD to 7-Segment display had been proven to function correctly, the binary to BCD converters needed to be added in so that an inputted binary value using the FPGA switches on the Nexys4 DDR, could be taken all the way through and processed as a digit(s) on the 7-segment display(s) at the end. For this a new test block design was created with the binary to BCD IP being imported 3 times. Suitable inputs and outputs were then assigned to this design. The master XDC file once again supplied the constraints for the switches and 7-segment displays. This meant that now a binary value would be inputted using the switches rather than a BCD value. Switch 0 was set as being the least significant figure, while switch 7 was set as being the most significant figure.



Good design shown

Figure 45 - Second Block Design

```

1  ## Clock signal
2
3  set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
4  create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
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_L3W_T0_DQS_EMCCLK_14 Sch=sv[1]
10 set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { B2 }]; #IO_L6H_T0_D06_VREF_14 Sch=sv[2]
11 set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { B3 }]; #IO_L13W_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 U18      IOSTANDARD LVCMOS33 } [get_ports { B6 }]; #IO_L17N_T2_A13_D20_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 T8       IOSTANDARD LVCMOS18 } [get_ports { B0 }]; #IO_L24N_T3_34 Sch=sv[8]
17 set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports { B1 }]; #IO_25_34 Sch=sv[9]
18 #set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L16P_T2_DQS_RDWR_B_14 Sch=sv[10]
19 #set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sv[11]
20 #set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sv[12]
21 set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports { Ret }]; #IO_L20P_T3_A06_D24_14 Sch=sv[13]
22 set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports { SW14 }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sv[14]
23 set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 } [get_ports { SW15 }]; #IO_L21P_T3_DQS_14 Sch=sv[15]
24
25 ## 7 segment display
26
27 set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { sa }]; #IO_L24N_T3_A00_D16_14 Sch=ca
28 set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports { sb }]; #IO_25_14 Sch=cb
29 set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports { sc }]; #IO_25_15 Sch=cc
30 set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports { sd }]; #IO_L17P_T2_A26_15 Sch=cd

```

Figure 46 - Second Constraints

Good .xdc file shown

The program was then downloaded again to the FPGA board and this time 4 tests were performed, three of which used the same values from the initial Nexys4 practical results. The first test was to make sure the board displayed a maximum value of 255. This was done by switching all the switches on. This gave the following result in figure 47 which was correct. [Go](#)

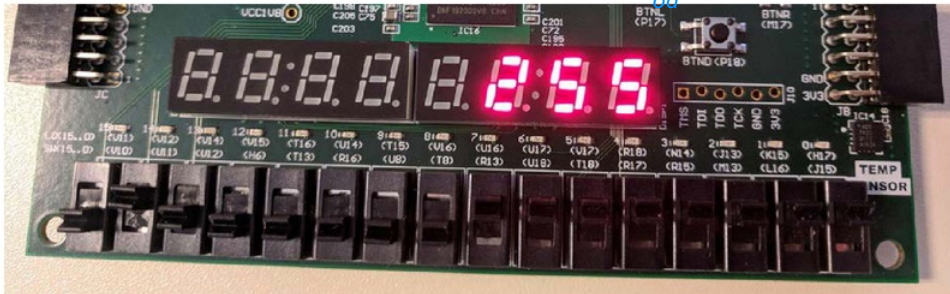


Figure 47 - Test for 255

The second test was for the value of 232. This involved its binary value of 1 1 1 0 1 0 0 0 being entered onto the switches. This gave the following result in figure 49 which proved to be correct.

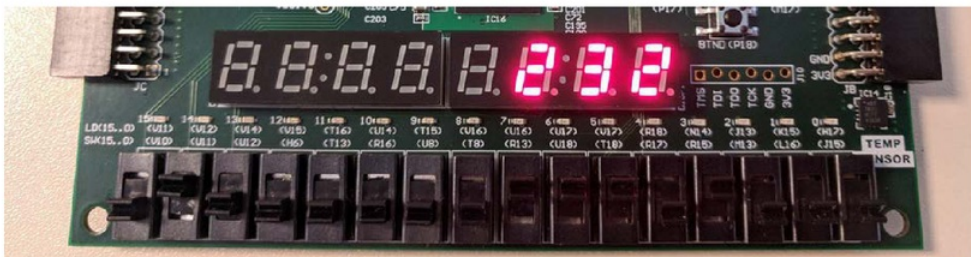


Figure 49 - Test for 232

[Good analysis](#)

The third test was for a value of 119. This involved a binary value of 0 1 1 1 0 1 1 1 being inputted into the switches. This gave the following result shown in figure 48, which was the correct display.

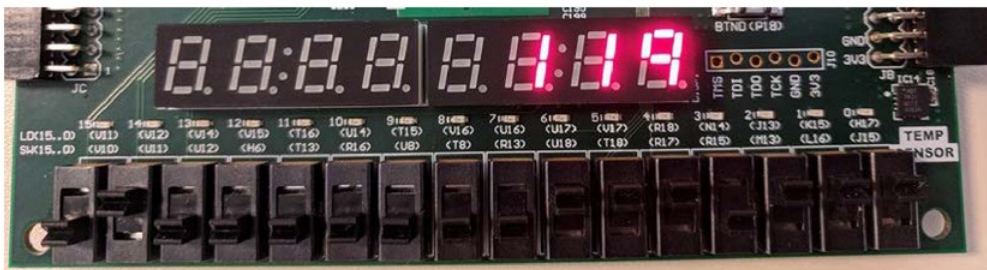


Figure 48 - Test for 119

[Good analysis](#)

The fourth and final test was for the value of 43. This involved entering the binary value of 0 0 1 0 1 0 1 1 being entered onto the switches. This gave the result shown in figure 50 which was the correct display for the board.

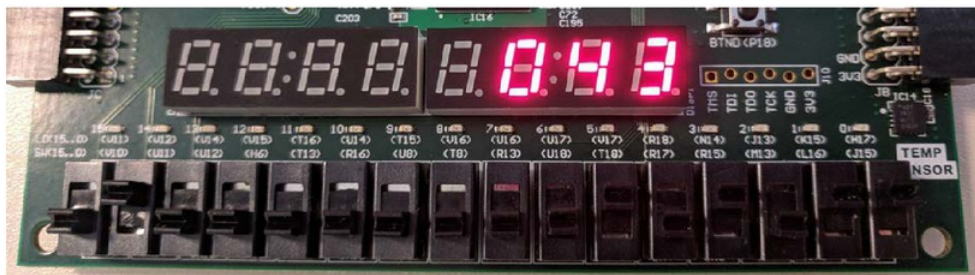


Figure 50 - Test for 43

*Good analysis
Good use made of pictorial evidence*

7. Conclusions

Overall the aim of this assignment was to investigate, practically and through research, the Nexys4 DDR FPGA board, the coding language VHDL, and the Vivado design suite. For this research was carried out into 3 key fields; VHDL code, the methods of converting binary to BCD, and how Vivado adopts the IP design method. Each section was thoroughly research with quotes and references throughout to reliable sources, both book and Internet based. The VHDL code was broke down to its foundations, how it adapts Boolean expressions as its own, with the design also being well explained. The method in which a binary value becomes a BCD value, and the algorithm known as the “double dabble” algorithm was thoroughly explained giving clear broken down examples of how a binary value becomes the new BCD value, including the uses of units, tens, hundreds and thousands up. The final bit of research was into intellectual properties known as IP’s and how Vivado adopts these to make a more flushed design for programming.

Once the research had been carried out, various VHDL files needed to be written, simulated and then downloaded for further testing on a Nexys4 DDR FPGA board. The tests went as follows:

- 6 bit binary to BCD converter – this file was only simulated, and displayed the correct conversion of 3 binary values to new BCD values.
- 8 bit binary to BCD converter – this file was adapted from the 6 bit file. 3 values were simulated in the Vivado design suite before 3 more values were tested on the FPGA using switches and LEDS. All of these tests gave the correct results.
- 5 KHz clock – the 5 KHz clock was simulated and gave the correct period length using the Vivado design suite simulation.
- Multiplexer – the multiplexer VHDL file was built and simulated in software and gave the correct results.
- BCD to 7-segment display – this was initially built and simulated in software before being loaded onto the FPGA in the first block test design with the 5 KHz clock to make sure that BCD values converted to 7-segment digits correctly, which they did.
- The final test was combining all the VHDL files as IP’s and importing them into a block design. This block design was then downloaded onto the FPGA for the final test. Three binary values were inputted using the switches 0 – 7 on the FPGA board, and the correct 7-segment digits were displayed for each one.

Good general conclusion shown but very little reflection

8. Appendices

Bibliography

Churiwala, S. (2016). *Designing with Xilinx® FPGAs: Using Vivado* (illustrated ed.). Springer.

Digilent: A National Instruments Company. (n.d.). *Advanced Microblaze Design using Memory Interface Generator (MIG), Ethernet, UART & GPIOs*. Retrieved from Digilent: A National Instruments Company: <https://reference.digilentinc.com/nexys4-ddr/advmb>

EuroElectronica. (2016, 4 26). *Basic Structure of a VHDL File*. Retrieved from EuroElectronica: <http://www.euroelectronica.ro/3-basic-structure-of-a-vhdl-file/>

James, M. (2002). *VHDL*. Retrieved from anu.edu.au: <http://users.cecs.anu.edu.au/~Matthew.James/engn3213-2002/notes/combvhdlNode1.html>

Jones, C. T. (1998). *Programmable Logic Controllers: The Complete Guide to the Technology* (illustrated ed.). Brilliant-Training.

Pedroni, V. A. (2008). *Digital Electronics and Design with VHDL*. Morgan Kaufmann.

Stephen D. Brown, Z. G. (2009). *Fundamentals of Digital Logic with VHDL* (3 ed.). McGraw-Hill.

*Good but could have been more
extensive*

Virtual Instrumentation & Control Assignment 1 FINAL Scott Dedoncker

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

62/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

Text Comment. Good introduction to the assignment which clearly shows what is required to do to complete

PAGE 5

Text Comment. Good

PAGE 6

Text Comment. Very good detail shown

PAGE 7

Text Comment. Good detail

PAGE 8

Text Comment. Good

PAGE 9

Text Comment. Good detail

PAGE 10

Text Comment. Again, good detail

Text Comment. Good

Text Comment. Good detail

Text Comment. Good understanding shown of how the test bench file works

Text Comment. Good analysis

Text Comment. Good analysis

Text Comment. Good detail shown

Text Comment. Where is the 8 bit IP block design? How was it obtained from the 6 bit design?

Text Comment. Good

Text Comment. Good detail

Text Comment. Good analysis given

Text Comment. Good analysis

Text Comment. Good pictorial evidence

Text Comment. Good analysis

Text Comment. Good analysis

PAGE 21

PAGE 22

Text Comment. Why 10000? Where is the maths?

PAGE 23

Text Comment. Good

PAGE 24

Text Comment. Good

PAGE 25

Text Comment. Not clearly understood

PAGE 26

PAGE 27

Text Comment. Good

PAGE 28

PAGE 29

Text Comment. Good detail

PAGE 30

Text Comment. Clear understanding shown

PAGE 31

Text Comment. Good

PAGE 32

PAGE 33

Text Comment. Good

PAGE 34

PAGE 35

Text Comment. Very good analysis of 3 sets of simulated tests carried out

Text Comment. Good analysis given

Text Comment. Good set of test results analysed correctly

Text Comment. Good detail

Text Comment. How do I know the switches are set to 257?

Text Comment. Results are correct but not clearly explained

Text Comment. Good design shown

Text Comment. Good .xdc file shown

Text Comment. Good

Text Comment. Good analysis

Text Comment. Good analysis

Text Comment. Good analysis

Text Comment. Good use made of pictorial evidence

Text Comment. Good general conclusion shown but very little reflection

Text Comment. Good but could have been more extensive

GRADING FORM: CSG-RUBIC

SCOTT DEDONCKER

62

RESEARCH-INFORMED LITERATURE



Structure of a typical VHDL file: Satisfactory / good research carried out. (2/5) Binary to BCD conversion: Good detail shown in the report. (3/5) IP Block Design in Vivado: Good detail. (3/5) Total (8/15)

8

KNOWLEDGE AND UNDERSTANDING



Design of 6/8 bit Binary to BCD Converter: 6 bit design clearly understood. (3/5) No IP Block diagram for 8 bit design with the design not explained well. (2/5) Re-design to suit new applications: More detail required on 5KHz clock design. (1/3) Mux/Demux not clearly understood. (1/3) 7 segment decoder explained well. (2/3). Full design adaptation clear and correct. (4/6) Total (13/25)

13

ANALYSIS



Analysis of Xilinx Vivado simulation results: 6 bit analysis clear and correct. (4/5) 8 bit analysis very clear and correct. (4/5) Analysis of adapted application results: Very good analysis of all simulated results including 5KHz clock (3/5), Mux/Demux (3/5) and 7 segment decoder. (3/5)
Total (17/25)

17

PRACTICAL APPLICATION AND DEP.



Downloading and testing to specification on Nexys 4 DDR FPGA board: Good set of test results which have been analysed correctly for the 8 bit Binary to BCD converter. (7/10) Downloading and testing of new application results on the Nexys 4 DDR FPGA board: Very good pictorial evidence shown and analysed correctly for the adapted design. (7/10)
Total (14/20)

14

SKILLS FOR PROFESSIONAL PRAC.



Deployment of appropriate media: Good use made of screen shots and pictorial evidence in design and analysis. (4/5) Report layout and readability: Report was easily understood and covered most aspects required to complete the assignment. (3/5)
Appropriate conclusion: Good conclusion given which confirms assignment completion but with little reflection. (3/5) Total (10/15)

10