Introduction

FPGAs can process instructions in parallel, hence making them advantageous to microcontrollers and microprocessors. The purpose of this lab is to program the Nexys4 DDR using the Vivado IDE in VHDL so that a binary flip flop counter can run parallel with a 7 segment counter.

1. **Binary counter**

```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26  -- Uncomment the following library declaration if using
27  -- arithmetic functions with Signed or Unsigned values
28  --use IEEE.NUMERIC_STD.ALL;
29
30  -- Uncomment the following library declaration if instantiating
31  -- any Xilinx leaf cells in this code.
32  --library UNISIM;
33  --use UNISIM.VComponents.all;
34
35  entity counter is
36      Port ( clk : in STD_LOGIC;
37             direction : in STD_LOGIC;
38             count_out : out STD_LOGIC_VECTOR (3 downto 0);
```

Figure 1.1: counter design source with IEEE libraries

To create a binary counter, **a design source** must first be initialised, this will allow us to define a module thus defining I/O ports such as **clk, direction and count_out**. Furthermore, defining a **design source** will allow programmers to code VHDL in a hierarchical manner. As shown in Figure 1.1, using the IEEE libraries, I/O ports inside the **Entity** can be used to declare the module's pinouts. Additionally, it allows us to map parameters to the required ports on the Nexys4 DDR such as the LEDs.

```
45      architecture Behavioral of counter is
46
47      signal count_int : unsigned(3 downto 0) := (others => '0');
48
49      begin
50
51      process (clk) --begin function
52      begin
53         if clk ='1' and clk 'event then -- changes the clock from high to low
54            --allows for the osscillation by adding and subtracting from the internal counter
55            if direction = '1' then
56               count_int <= count_int + 1;
57            else
58               count_int <= count_int - 1;
59            end if;
60         end if;
61      end
62      process;
63
64      count_out <= std_logic_vector(count_int(3 downto 0));
65
66      end Behavioral;
```

Figure 1.2: Copied code from Vivado IDE coding example

As shown in Figure 1.2. the **Coding Example** feature allows programmers to copy commonly used codes to increase coding efficiency. Thus, a **Simple Counter** was copied to create a flip flop that will oscillate between 0 and 1 at the clock frequency. This code is placed inside the **Architecture** (which describes the module's internal structure). Additionally, variable names were changed from the **Coding Example** accordingly to allow the module to function.

This **Simple Counter** is simulated against a test bench (as shown in Figure 1.3) to verify the functionality of the circuit similar to a test case; comparing the circuit's expected inputs and outputs in a software setting. This verifies the HDL language thus preventing synthesis error. The **Counter** is working as the switching matches its testbench, shown when comparing **dir** and **count[3:0]**.

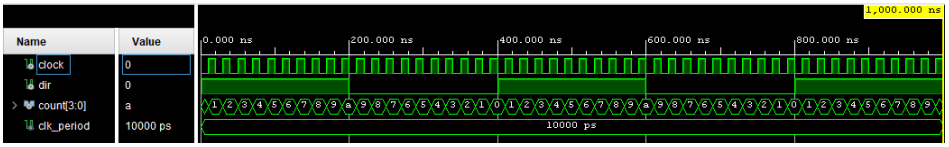

Figure 1.3: Binary counter test bench

2. Synthesis, Implementation and delaying the binary counter

The Counter project will then be synthesised to convert the RTL code to the netlist, subsequently the implementation tool is leveraged to optimise the netlist. This will allow the FPGA to be programmed by generating the bitstream. Binding the LEDs on the FPGAs onto the desired I/O ports was also done manually in the console, this allowed the LED binary counters shown in Figure 2.1 to be displayed. However, this binary counter is switching so fast it appears to be toggled on. Thus, the counter should be slowed down by a bit divider for the human eye.

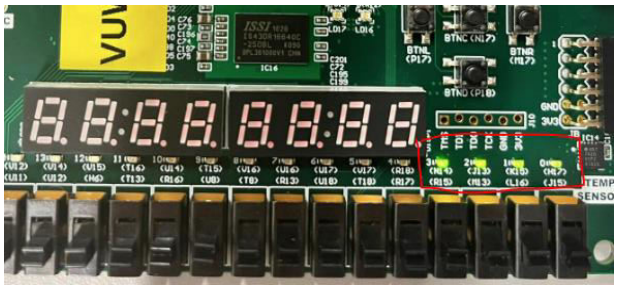

Figure 2.1: Counter output on FPGA LEDs

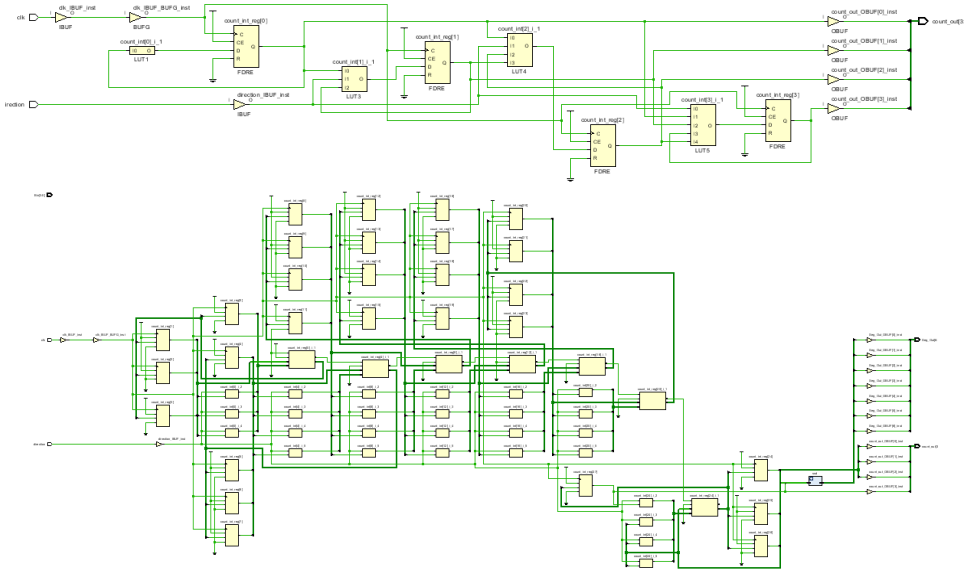

Figure 2.2: Schematic of a binary counters (top) and a binary counter with a bit divider

Figure 2.2 illustrates the schematic difference between a binary counter and a binary counter with a bit divider thus, highlighting the saved production time of implementing a bit divider. This was implemented by altering 2 lines of code in Figure 1.2 to the code in Figure 2.3. This alteration allows the LEDs to output when the 4 MSB is encountered. Furthermore, this would allow the human eye to observe the Nexys4 DDR's binary counter as shown in Figure 2.4 unlike the LEDs shown in Figure 2.1. Unlike high level programming languages, VHDL requires bit dividers to slow down a counter efficiently as an FPGA internal clock rate cannot be altered.

```
signal count_int : unsigned(27 downto 0) := (others => '0');
count_out <= std_logic_vector(count_int(27 downto 24));
```

Figure 2.3: code alteration made for a bit divider

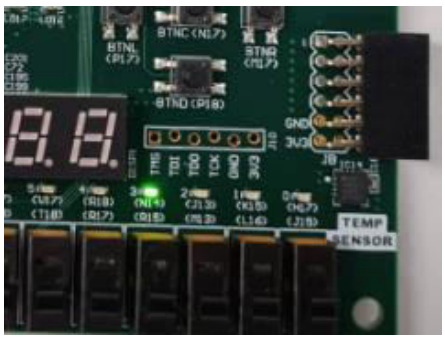

Figure 2.4: Binary counter at "1000" with a bit divider for the human eye to observe.

3. 7-Segment display

Hex Digit	Inputs: 4-bit binary number representing 0-9				Outputs: Seven functions that will drive the individual display segments						
					G	F	E	D	C	B	A
0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	0	0	1
2	0	0	1	0	0	1	0	0	1	0	0
3	0	0	1	1	0	1	1	0	0	0	0
4	0	1	0	0	0	1	1	0	0	0	1
5	0	1	0	1	0	0	1	1	0	1	0
6	0	1	1	0	0	0	0	0	0	1	0
7	0	1	1	1	1	1	1	1	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	1	0	0	0	0
A	1	0	1	0	0	0	0	1	0	0	0
B	1	0	1	1	0	0	0	0	0	1	1
C	1	1	0	0	1	0	0	0	1	1	0
D	1	1	0	1	0	1	0	0	0	0	1
E	1	1	1	0	0	0	0	0	1	1	0
F	1	1	1	1	0	0	0	1	1	1	0

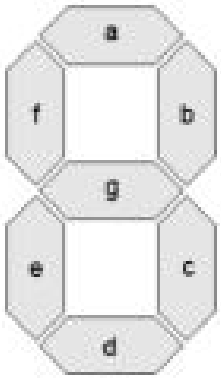

Figure 3.1 Truth table of the 7-segment display

Figure 3.1 shows the truth table of the 7-segment display (7SD) counter that would be processed parallel to the binary counter. This truth table will be applied onto another **source design** called "ssd_decoder". This will be called into the counter **source design** so that it can be processed in parallel with the binary counter as observed in Figure 3.2. Notably, the Nexys4 DDR uses inverted logic thus 0 = on toggle.

Figure 3.2: Binary counter and 7-segment display counter running in parallel

Similarly to how the binary counter is made in section 1 and 2, all inputs, outputs, vectors and the ssd component etc. was declared for the 7SD counter inside the counter **design source** which can be observed in the appendix. Furthermore, a testbench wasn't necessary for the 7SD as it ticked in parallel with the binary counter output e.g. "1111001" when "0001" hence, synchronised.

Questions

Q1 What is the name of the Xilinx chip we are using and how many pads/pins does it have?

- The Artix-7 FPGA has 324 pins

Q2 What frequency does our chip run at?

- 100 Mhz, but can be capable of generating internal clock speeds exceeding 450 MHz.

Q3 What is the frequency of bit 24 of count_int (bit 0 of count_out)?

- 3 Hz

Q4 If you wanted a clock frequency of just over 3kHz, how many bits long would your count have to be?

- $\frac{10^8}{3000} = 2^{+1}$.: = 14 bits; assuming the original clock frequency is 100 MHz.

Q5 What is the purpose of the following line in the counter.vhd file: Count_out <= count_int;

- Assigning the value of "count_in" to the "count_out" variable

Q6 What is the purpose of a testbench? Explain how this is useful.

- A test bench is used when comparing the circuit's expected inputs and outputs in a software simulation; thus analogous to "test cases" in software development. This verifies the HDL language which aims to prevent synthesis error, consequently allows the circuit to be programmed onto a FPGA.

Code for the binary counter with bit divider running in parallel with 7SD counter

```
20
21
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26  -- Uncomment the following library declaration if using
27  -- arithmetic functions with Signed or Unsigned values
28  --use IEEE.NUMERIC_STD.ALL;
29
30  -- Uncomment the following library declaration if instantiating
31  -- any Xilinx leaf cells in this code.
32  --library UNISIM;
33  --use UNISIM.VComponents.all;
34
35  entity counter is
36      Port ( clk : in STD_LOGIC;
37             direction : in STD_LOGIC;
38             count_out : out STD_LOGIC_VECTOR (3 downto 0);
39             Bin : in STD_LOGIC_VECTOR (3 downto 0);
40             Seg_Out : out STD_LOGIC_VECTOR (6 downto 0)
41             );
42
43  end counter;
44
45  architecture Behavioral of counter is
46
47  Component ssd_decoder is
48  Port (
49      Bin : in STD_LOGIC_VECTOR (3 downto 0);
50      Seg_Out : out STD_LOGIC_VECTOR (6 downto 0)
51      );
52  end
53  Component;
54
55  signal count_int : unsigned(27 downto 0) := (others => '0');
56
57  begin
58
59
60
61  ssd: ssd_decoder PORT MAP(
62      Bin => std_logic_vector(count_int(27 downto 24)),
63      Seg_Out => Seg_Out
64      );
65
66
67  process (clk) --begin function
68  begin
69     if clk ='1' and clk 'event then -- changes the clock from high to low
70        --allows for the osscillation by adding and subtracting from the internal counter
71        if direction = '1' then
72           count_int <= count_int + 1;
73        else
74           count_int <= count_int - 1;
75        end if;
76     end if;
77  end
78  process;
79
80  count_out <= std_logic_vector(count_int(27 downto 24));
81
82  end Behavioral;
83
```

7SD code

```
34  entity ssd_decoder is
35      Port ( Bin : in STD_LOGIC_VECTOR (3 downto 0);
36             Seg_Out : out STD_LOGIC_VECTOR (6 downto 0));
37  end ssd_decoder;
38
39  architecture Behavioral of ssd_decoder is
40
41  begin
42
43  --HEX-to-seven-segment decoder
44  --    HEX:    in    STD_LOGIC_VECTOR (3 downto 0);
45  --    LED:    out   STD_LOGIC_VECTOR (6 downto 0);
46  --
47  -- segment encoinputg
48  --       0
49  --      ---
50  --  5 |   | 1
51  --      ---   <- 6
52  --  4 |   | 2
53  --      ---
54  --       3
55
56    with Bin SELect
57     Seg_Out<= "1111001" when "0001",    --1
58           "0100100" when "0010",    --2
59           "0110000" when "0011",    --3
60           "0011001" when "0100",    --4
61           "0010010" when "0101",    --5
62           "0000010" when "0110",    --6
63           "1111000" when "0111",    --7
64           "0000000" when "1000",    --8
65           "0010000" when "1001",    --9
66           "0001000" when "1010",    --A
67           "0000011" when "1011",    --b
68           "1000110" when "1100",    --C
69           "0100001" when "1101",    --d
70           "0000110" when "1110",    --E
71           "0001110" when "1111",    --F
72           "1000000" when others;    --0
73
74  end Behavioral;
```