Ben Linam
Dylan Hall
EEL4712L
20 Sept. 2018
Lab 3

EEL4712L

Lab 3 – Introduction to VHDL

Prelab

Preliminary Setup

Overview

*For this lab, we will be orienting ourselves with the new **IDE** used for **VHDL** in this course, **Quartus Prime.** Also, we will be learning some coding conventions and syntax for **VHDL**. We will be learning how to write logic designs and form them into one project which will enable us to test the logic on an **Altera DE1-SoC FPGA.***

Materials

- An Altera DE1-SoC kit
- A computer with Quartus Prime

Procedure

*We will be using **ENTITY** designs in this laboratory. We will be designing a **BCD-to-10-Line Decoder** as well as a **4-to-2 Priority Encoder.** Once designed, we will be using the **Pin Assignment** utility provided by **Quartus Prime** to determine the **Software → Hardware** implementations.*

*Such that, we will be setting the logic inputs/outputs from our **VHDL** designs to physical pins on our **Altera DE1-SoC** boards to test and simulate the designs.*

*From this, we will be able to use our previously designed **8-bit-up-counter** to test the combinational logic of our new designs.*

Part 1: *BCD-to-10-Line Decoder*

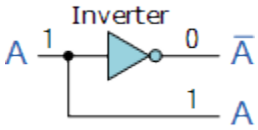
- Background:

BCD: *Binary-coded Decimal*

- *Standard logic of 0000 → 1111, this bit vector holds a length of 4 bits.*

Description:

*A decoder is a common component used to convert input's into output signals. A **Line Decoder** takes an n-digit binary number and creates an 2^n output signal. Usually larger scaled **Decoders** are just an assortment of **1-to-2 Line Decoders** cascaded into one another. A simple **1-to-2 Line Decoder** is an inverter. An inverter takes in one input and produces 2^1 outputs. The architecture involved inside common decoders is just several input **AND** gates whose inputs are coordinated with the input and inverted input lines of the decoder.*



*From this image, one will see that the input **A**, once propagated, will produce a 2-signal output $(\boldsymbol{A}, \overline{\boldsymbol{A}})$.*

Figure 1. Screen capture displaying an **Inverter**

*From the description of the **BCD-to-10-Line Decoder,** one can see that our implementation will hold **4 input signals,** with an $\mathbf{2^4}$, or **16**, output signals.*

*But, our implementation is described to be a **BCD-to-10-Line Decoder,** this simply means the last 5 possible output signals will always be invalid, or disabled output lines. **(See Table 1.)***

Producing Boolean Output Expressions:

*To produce Boolean output expressions, one would use **SOP (Sum of Products: minterms, 1's)**. This is because of the nature of the **Decoder**, which propagates the inputs through **AND** gates, so all you must do is **OR** the outputs together to reproduce the proper output functions. But, if the implementation requires the outputs to be **ACTIVE-LOW**, one must use **NOR** gates instead to produce the proper output functions.*

- Logic/Truth Tables –

*Below you will see the logic truth-table for a **BCD-to-10-Line Decoder**. For this implementation the inputs are registered with **logic-high (positive logic, active-high, 1)** and our outputs are triggered with **logic-low (negative logic, active-low, 0).***

Key (Table 1):

- Rows this color are designed as valid inputs.

- Rows this color are designed as invalid inputs.

- Cells this color represents valid outputs

#	$A_3A_2A_1A_0$	D_{15}'	D_{14}	D_{13}'	D_{12}'	D_{11}'	D_{10}'	D_9'	D_8'	D_7'	D_6'	D_5'	D_4'	D_3'	D_2'	D_1'	D_0'
0	0 0 0 0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0 0 0 1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0 0 1 0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0 0 1 1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
4	0 1 0 0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
5	0 1 0 1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
6	0 1 1 0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
7	0 1 1 1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
8	1 0 0 0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
9	1 0 0 1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
10	1 0 1 0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1 0 1 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
12	1 1 0 0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
13	1 1 0 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	1 1 1 0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15	1 1 1 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 1.** Truth table for a **BCD-to-10-Line Decoder

- Logic/Truth Tables Cont. -

*Because this design is **ACTIVE-LOW** we will be using **NOT** and **NAND** gates to produce the proper output functions. We use **NAND** gates to represent the **NEGATIVE-LOGIC**, but we complement the input to **AND** gates to adjust for inverted inputs/outputs.*

Note: $[D_5 \rightarrow D_0] = 0$

I.e.

$$D_{15} = A_3 + A_2 + A_1 + A_0 \rightarrow De\ Morgan's$$

$$D_{15} = (A_3'A_2') + (A_1'A_0') \rightarrow De\ Morgan's$$

$$\boldsymbol{D_{15} = \left((A_3'A_2')(A_1'A_0')\right)' \rightarrow (1)}$$

*Now, simply take **(1)** and **NOT** each term incrementally in the same fashion as one would count in binary.*

i.e.

$$D_{14} = \left((A_3'A_2')(A_1'A_0)\right)'$$

$$D_{13} = \left((A_3'A_2')(A_1A_0')\right)'$$

$$D_{12} = \left((A_3'A_2')(A_1A_0)\right)'$$

$$D_{11} = \left((A_3'A_2)(A_1'A_0')\right)'$$

…

$$D_6 = \left((A_3'A_2)(A_1A_0')\right)'$$

- Realization/Simulation –

*Below you will see a screen capture of the **Quartus Prime logic design** of the **BCD-to-10-Line Decoder**.*

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5   ENTITY bcd_decoder IS
6       GENERIC(N : INTEGER:=4);
7       PORT(
8       EN : IN STD_LOGIC;
9       A  : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10      D  : OUT STD_LOGIC_VECTOR((N**2)-1 DOWNTO 0));
11  END bcd_decoder;
12
13  ARCHITECTURE logic OF bcd_decoder IS
14  BEGIN
15
16  D(15) <= ( NOT A(3) AND NOT A(2) ) NAND (NOT A(1) AND NOT A(0));
17  D(14) <= ( NOT A(3) AND NOT A(2) ) NAND (NOT A(1) AND     A(0));
18  D(13) <= ( NOT A(3) AND NOT A(2) ) NAND (    A(1) AND NOT A(0));
19  D(12) <= ( NOT A(3) AND NOT A(2) ) NAND (    A(1) AND     A(0));
20  D(11) <= ( NOT A(3) AND     A(2) ) NAND (NOT A(1) AND NOT A(0));
21  D(10) <= ( NOT A(3) AND     A(2) ) NAND (NOT A(1) AND     A(0));
22  D(9)  <= ( NOT A(3) AND     A(2) ) NAND (    A(1) AND NOT A(0));
23  D(8)  <= ( NOT A(3) AND     A(2) ) NAND (    A(1) AND     A(0));
24  D(7)  <= (     A(3) AND NOT A(2) ) NAND (NOT A(1) AND NOT A(0));
25  D(6)  <= (     A(3) AND NOT A(2) ) NAND (NOT A(1) AND     A(0));
26  D(5) <= '1';
27  D(4) <= '1';
28  D(3) <= '1';
29  D(2) <= '1';
30  D(1) <= '1';
31  D(0) <= '1';
32
33  END logic;
34
```

Figure 2. Screen capture displaying the **Quartus Prime logic design** for the **BCD-to-10-Line Decoder**

- Code Logic -

Following Figure 2.

*The **VHDL** code follows the basic format. We include the libraries, we declare an **ENTITY (bcd_decoder)** then we declare that entities **PORTS** (in/out). In this case, we used a **GENERIC** to change the decoders number of input/output bits easier (**See BCD-to-10-Line Decoder Background).** Finally, we declare the **ARCHITECTURE** for the **ENTITY.** We declared out inputs/outputs in **Big Endian,** which holds the **MSB (most-significant-bit)** to the left, and the **LSB (least-significant-bit)** on the right. This allows us to directly follow our truth table. Because this is an **ACTIVE-LOW** decoder, we had to look for zeros. The logic behind this will be noted in the truth table, **(See Table 1.)**.*

*Because we derived the functions in such a way that only one output will active at a time, each output must be a function of each input. i.e. if one is to build a **K-MAP**, one would notice every zero would have no possible connection.*

*It was noted in the design requirements, and implied in the name, that only ten lines will be utilized in this design, hence, terms **5** to **0 (LSB)** will always be set to one.*

- Simulated Waveform Diagram –

*Below you will see the simulated timing/waveform diagram of the **VHDL** design above.*

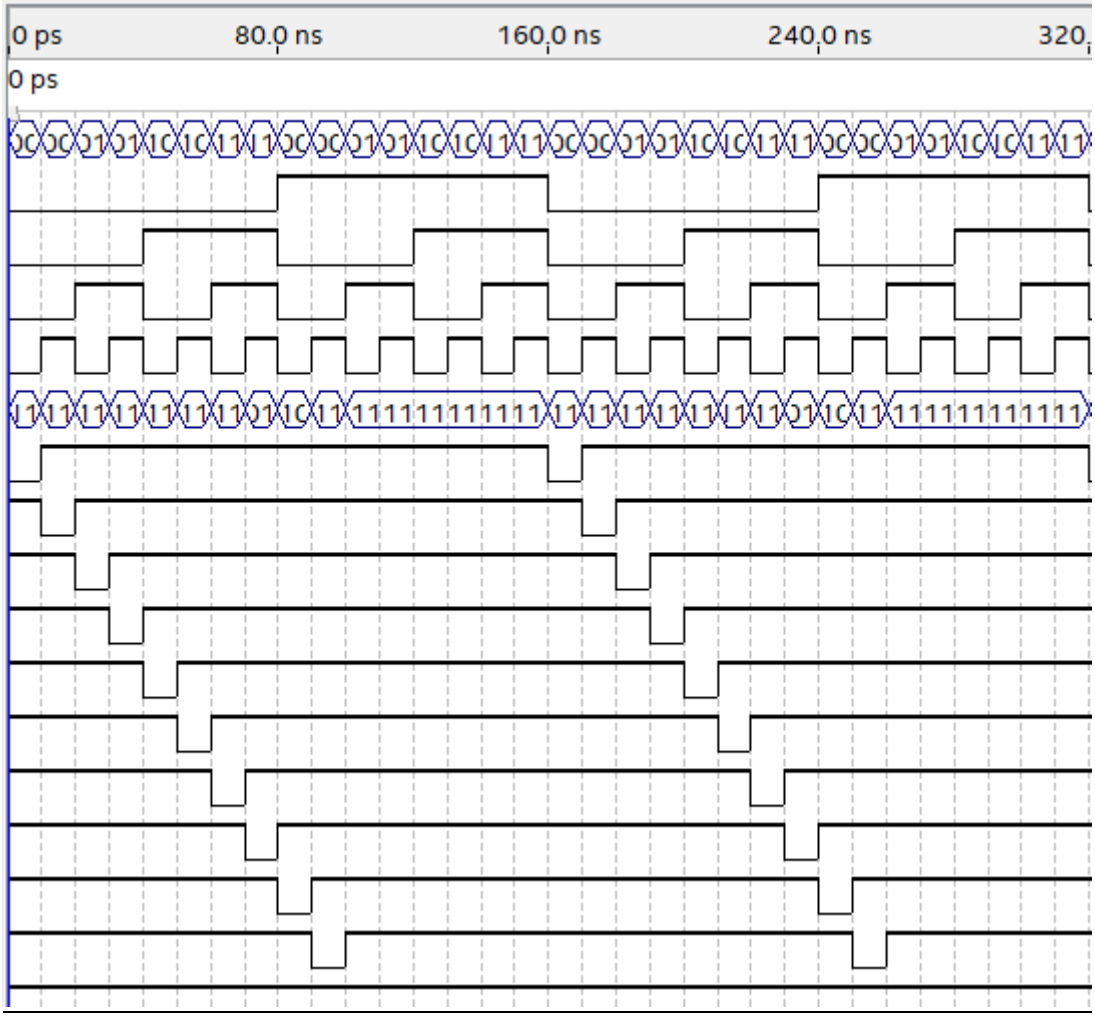


Figure 3. Screen capture displaying the simulation timing/waveform diagram of the logic design

Part 2: *4-to-2 Priority Encoder*

- Background

Description:

*An encoder is a common component used to convert input's into output signals. An **encoder** takes an 2^n binary inputs and creates an n bit output signal. A downfall to this design, is such that, the encoder can produce an incorrect output if more than one input is **logic-high (1)** at one time. To get around this problem, the **Priority Encoder** was designed. This type of encoder, by nature, assigns a priority to each input. This causes the **Priority Encoders** output corresponds to the currently active input which has the highest priority. This will cause all other inputs that are **logic-high** with **lower** priority to be ignored in the output.*

Producing Boolean Output Expressions:

*To produce proper Boolean output functions, one would utilize **POS (Product of Sums: maxterms, 1's)**. This is because the nature of a priority encoder outputs already include **OR** gates. So, one only needs to put the outputs through **AND** gates to represent the functions. But, if the design implementation calls for **ACTIVE-LOW** outputs, one must utilize **NAND** gates, to replicate the proper function behavior.*

- Logic/Truth Tables –

*Below you will see the logic truth-table for a **4 to 2 Priority Encoder** . For this implementation the inputs are registered with **logic-high (positive logic, 1)** and our outputs are triggered with **logic-low (negative logic, 0).***

Key (Table 2):

- Rows this color are designed as valid inputs.

- Cells this color represents valid outputs

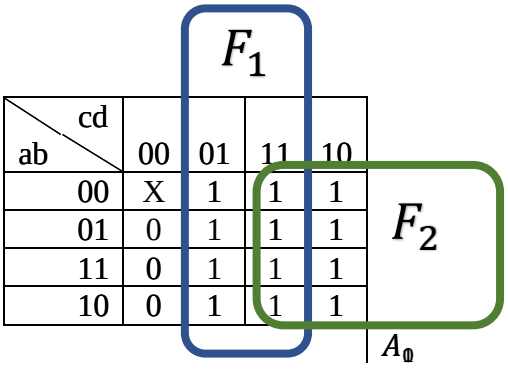
	Input				***Output***		
#	D_3	D_2	D_1	D_0	A_1	A_0	Any
0	0	0	0	0	X	X	0
1	0	0	0	1	0	0	1
2	0	0	1	X	0	1	1
3	0	1	X	X	1	0	1
4	1	X	X	X	1	1	1
…	X	X	X	X	X	X	0
15	X	X	X	X	X	X	0

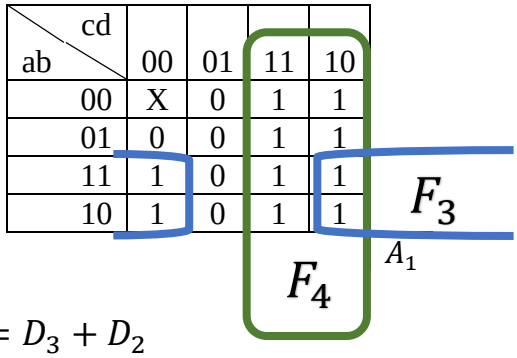
Table 2.** Truth table for a **4-to-2 Priority Encoder

$$Any = \sum(1,2,3,4) = D_3 + D_2 + D_1 + D_0$$

$$A_0 = \sum(2,4)$$

$$A_1 = \sum(3,4)$$





$$A_0 = F_1 + F_2 = D_3 + D_2$$

$$A_1 = F_3 + F_4 = D_3 + D_2' D_1$$

- Realization/Simulation –

*Below you will see a screen capture of the **Quartus Prime logic design** of the **4-to-2 Priority Encoder**.*

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5   -- Declaring an entity which acts as an Object
6   ENTITY prio_encoder IS
7      PORT ( -- PORT: acts as constructor, declaring entity inputs/outputs
8      Din:  IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9      Aout: BUFFER STD_LOGIC_VECTOR(1 DOWNTO 0);
10     V:    OUT STD_LOGIC);
11  END prio_encoder;
12
13  -- Declaring the behavior of the entity; entity logic design
14  ARCHITECTURE Behavior OF prio_encoder IS
15  BEGIN
16
17  Aout(0) <= (NOT Din(2) AND Din(1)) OR Din(3);
18  Aout(1) <= Din(2) OR Din(3);
19
20  V <= Din(0) or Din(1) or Din(2) or Din(3);
21  END Behavior;
```

Figure 4. Screen capture displaying the **Quartus Prime logic design** for the ***4-to-2 Priority Encoder***.

- Code Logic -

Following Figure 4.

*The **VHDL** code follows the basic format. We include the libraries, we declare an **ENTITY (bcd_decoder)** then we declare that entities **PORTS** (in/out). In this case, we used a **GENERIC** to change the decoders number of input/output bits easier (**See BCD-to-10-Line Decoder Background).** Finally, we declare the **ARCHITECTURE** for the **ENTITY.** We declared out inputs/outputs in **Big Endian,** which holds the **MSB (most-significant-bit)** to the left, and the **LSB (least-significant-bit)** on the right. This allows us to directly follow our truth table. Because this is an **ACTIVE-LOW** decoder, we had to look for zeros. The logic behind this will be noted in the truth table, **(See Table 1.)**.*

*Because we derived the functions in such a way that only one output will active at a time, each output must be a function of each input. i.e. if one is to build a **K-MAP**, one would notice every zero would have no possible connection.*

- Simulated Waveform Diagram –

*Below you will see the simulated timing/waveform diagram of the **VHDL** design above.*

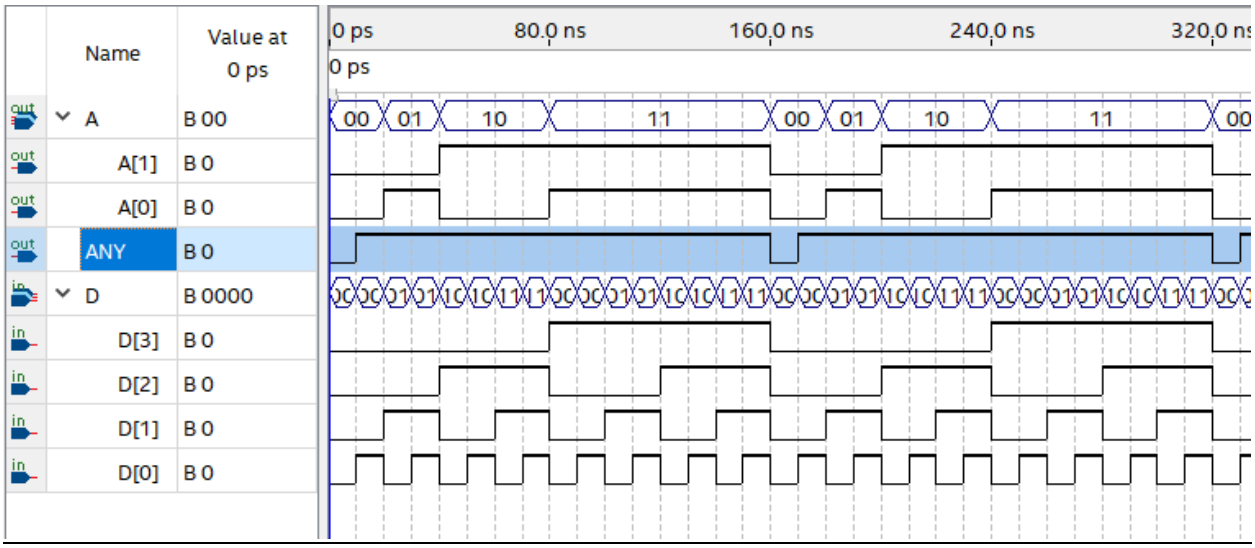


Figure 5. Screen capture displaying the simulation timing/waveform diagram of the logic design

Conclusion:

*In summation, this lab was another good introduction to both **VHDL** and the **DE1-SoC** board. IT was also very enjoyable to implement the logic behind components in which we have learned previously. Being able to programmatically replicate the logic behind these components is something of beauty. It was a really good refresher on some basic combinational logic and a good opportunity to review previous concepts.*