ECSE 222 Fall 2022
VDHL Assignment 3 Report: Design and Simulation of Digital Circuits

Part 1 – Participants

Karim Elgammal 260920556

Lutming Wang 261006348

Part 2 – Executive Summary

Throughout this lab, we have been able to implement a 4-bit barrel shifter, half adder, full adder, ripple carry adder and finally a BCD adder. Most of the more abstracted components of this lab have been implemented in both a structural and behavioral style. Moreover, we have been able to transverse through the different design abstractions of components that are assembled with smaller components (namely; the instantiation of the multiplexers within the structural design of the barrel shifter, the instantiation of the adders within the ripple carry adder, and then finally the instantiation of the ripple carry adder within the BCD adder). This allows us to explore the various abstractions within the design process, when synthesizing logic circuits.

Part 3 & 4 –

Implementation of a 4-bit Circular Barrel Shifter

A 4-bit circular barrel shifter is a circuit that can take in an input and shift said input a certain amount/direction of bits. This shift is essentially controlled using the select statement that is also inputted. To grasp a better understanding of what each sequence of the sel signal is able to do to the input, a truth table comprising of the position of the inputs after 'going through' the logic of the 4-bit circular barrel shifter was formulated (Fig. 1). Input: signal X (3 downto 0).

sel(0)	sel(1)	Y(0)	Y(1)	Y(2)	Y(3)
0	0	X(0)	X(1)	X(2)	X(3)
1	0	X(3)	X(0)	X(1)	X(2)
1	1	X(1)	X(2)	X(3)	X(0)
0	1	X(2)	X(3)	X(0)	X(1)

Figure 1 - Truth Table for Circular Barrel Shifter

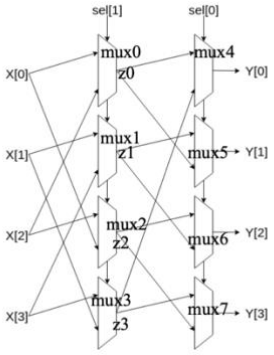

Figure 2.1 Schematic of Circular Barrel Shifter

Structural Implementation:

The structural implementation of the 4-bit circular barrel shifter utilizes two layers of 2-1 multiplexers to shift the bits using the selector signals which essentially serve as a control. As seen in figure 2.1, the barrel shifter uses 8 multiplexers laid parallel to each other in order to achieve its functionality. The structural description of the architecture of the entity 'karim_elgammal_barrel_shifter_structural' instantiates a 2-to-1 multiplexer 8 times as seen in the schematic above and maps the ports of the outputs of the first layer of multiplexers to a carrier signal 'z'. The signals z0, z1, z2, z3, are then used as inputs to the second layer of multiplexers that is seen within the schematic. Labels between the schematic and the source code are the same. The sel signal is constant for each layer of multiplexers; that is, the first layer of multiplexers only use sel(1) and the second layer always use sel(0) as the selector signal of each multiplexer.

```
architecture structBS of karim_elgammal_barrel_shifter_structural is

signal z0 : std_logic;
signal z1 : std_logic;
signal z2 : std_logic;
signal z3 : std_logic;

component myMUX
    port (A : in std_logic;
          B : in std_logic;
          S : in std_logic;
          outY : out std_logic);
          end component;

    begin

    mux0 : myMUX port map(A => X(0),
                          B => X(2),
                          S => sel(1),
                          outY => z0);

    mux1 : myMUX port map(A => X(1),
                          B => X(3),
                          S => sel(1),
                          outY => z1);

    mux2 : myMUX port map(A => X(2),
                          B => X(0),
                          S => sel(1),
                          outY => z2);

    mux3 : myMUX port map(A => X(3),
                          B => X(1),
                          S => sel(1),
                          outY => z3);

    mux4 : myMUX port map(A => z0,
                          B => z3,
                          S => sel(0),
                          outY => Y(0));

    mux5 : myMUX port map(A => z1,
                          B => z0,
                          S => sel(0),
                          outY => Y(1));

    mux6 : myMUX port map(A => z2,
                          B => z1,
                          S => sel(0),
                          outY => Y(2));

    mux7 : myMUX port map(A => z3,
                          B => z2,
                          S => sel(0),
                          outY => Y(3));

end structBS;
```

Figure 2.2 Structural Description of Circular Barrel Shifter

Figure 2.3 displays the simulation plot for the structural description of the circular barrel shifter. As seen for the binary number '1101', we can see the effects of each of the sequences of the selector signals. More specifically, we can see that when the selector signals are both 0; there is no change to the order of the bits that were initially inputted, this also matches the first row in our truth table (Fig. 1). When the selector signals are sel(0) = 1, sel(1) = 0, the bits are shifted to the right as seen within the simulation plot and also the truth table above. Sel(0) = 1, sel(1) = 1, results in the bits to shift to the left as seen within both our representations; and finally sel(0) = 0, sel(1) = 1, results in a shift of two magnitudes, and since this is a four bit input, this shift results in the same output weather it is to the right or to the left.

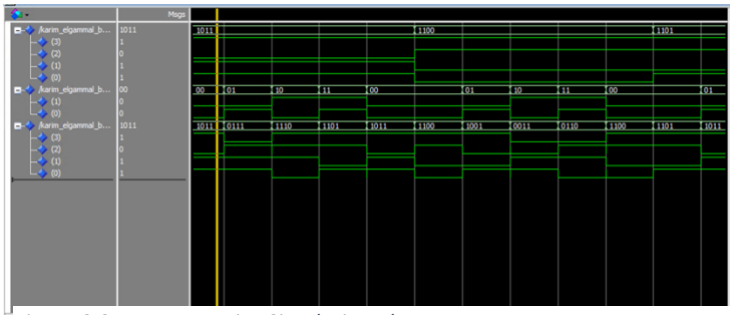

Figure 2.3 Representative Simulation Plot

Behavioral Implementation:

The behavioral implementation of the barrel shifter was completed using a single select statement as seen below in figure. The select statement is based on our logic from the truth table above (fig. 1). The concatenate operator is used to implement the shifts that occur depending on the select signal that is inputted along with the binary number, by combining the single signals of each placement in the input X, in the order described by the table in figure 1.

```
109  architecture behlogicFunc of karim_elgammal_barrel_shifter_behavioral is
110
111  begin
112
113    with sel select
114
115         Y <= X(0) & X(1) & X(2) & X(3) when "00",
116              X(3) & X(0) & X(1) & X(2) when "10",
117              X(1) & X(2) & X(3) & X(0) when "11",
118              X(2) & X(3) & X(0) & X(1) when "01",
119              "XXXX" when others;
120
121
122  end behlogicFunc;
123
124
```

Figure 2.4 Behavioral Implementation of Circular Barrel Shifter

The behavioral implementation results in an equivalent simulation plot (fig. 2.5). There were some confusions on the precedence of signals and signal assignment within the VHDL implementation, however this was resolved after some trial-and-error testing of the sequence of the select statement. The simulation plot below, shows us an input of 0110, and displays the various outputs based on the sequence of the select signal.

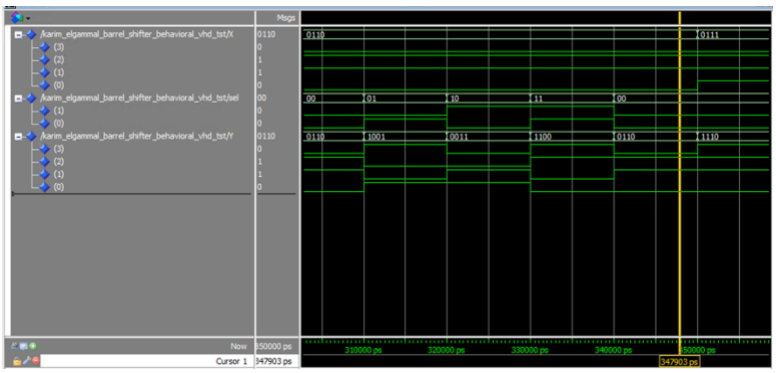

Figure 2.5 Behavioral Implementation Simulation Plot

<u>Implementation of a Half-Adder & a Full-Adder:</u>

Both adders in this section were implemented in a structural style; this allowed for a higher precision when it came to instantiating these adders within the design of our RCA. The half adder is implemented using the truth table of the half adder. From this truth table a circuit is synthesized to complete the addition of 2 bits. This structural implementation uses an XOR gate in order to reach the sum of the inputs, and an AND gate in order to output the carry signal. This results in a

combinational logic circuit that has the functionality of a half adder; adding two bits to each other, with a carry out signal which keeps track of the 'overflow' of the addition process.

```
136  architecture strucHA of half_adder is
137
138  begin
139
140    s <= a XOR b;
141    c <= a AND b;
142
143  end strucHA;
144
```

```
157  architecture strucFA of full_adder is
158
159    signal temp0 : std_logic;
160    signal temp1 : std_logic;
161    signal temp2 : std_logic;
162
163  component half_adder
164      port ( a : in std_logic;
165             b : in std_logic;
166             s : out std_logic;
167             c : out std_logic);
168        end component;
169
170  begin
171        ha_0 : half_adder port map( a => a,
172                                    b => b,
173                                    s => temp0,
174                                    c => temp1);
175
176        ha_1 : half_adder port map( a => c_in,
177                                    b => temp0,
178                                    s => s,
179                                    c => temp2);
180
181        c_out <= temp1 OR temp2;
182
183  end strucFA;
184
```

Figure 3.1 Structural Implementation of Half Adder and Full Adder

The full adder was also implemented in a structural method; more specifically, the half adder that was designed earlier was instantiated twice to realize the combinational logic of a full adder. A full adder logic circuit contains two sets of XOR-AND gates in order to combine the carry in of a previous addition calculation and supply the sum of the two bits. To implement this, temporary signals (temp0, temp1, temp2) were used to reiterate the sum and the count into the second half adder. Then, the count is carried out using an OR gate seen in line 181. Using a structural style within this design specifically, allowed for a minimized cost for the full adder (original cost of 17, 15 with instantiation). The simulation plots below coincide with the truth tables of each entity, hence, allowing us to believe that the implementation is correct.

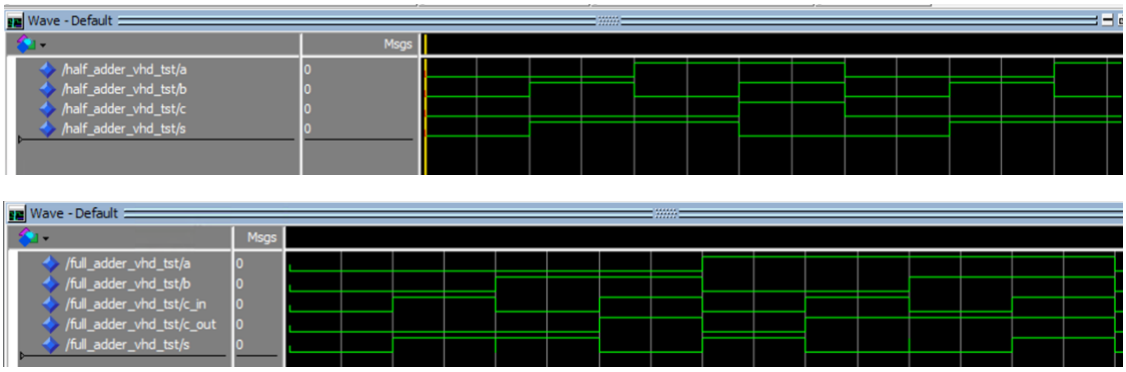

Figure 3.2 Representative Simulation Plots for Half-Adder and Full-Adder

Implementation of 4-Bit Ripple Carry Adder:

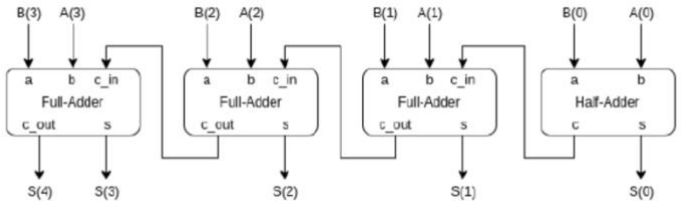

Figure 4.1 Combinational Logic Circuit of a 4-Bit RCA

Structural Implementation:

The structural implementation of the 4-bit ripple carry adder (RCA), was completed based on the combinational logic circuit provided above. By instantiating 3 full adders in sequence and 1 half adder. Using this design, we can see how each carry out signal is 'rippled' into the next adder, allowing for a carry pout to be processed by the result of the previous adders. Temporary signals are also used within this design in order to transfer the carry out signal of each adder to the next one. A half adder was used to process the first index of the input, and this is to account for the fact that when the RCA is being used holistically, there is not an initial carry input. Then, after the instantiation of the adders (which are declared as components of strucRCA), the inputs are mapped to the ports of the components that make up the design seen in figure 4.1.

```
196  architecture strucRCA of rca_structural is
197      signal temp0: std_logic;
198      signal temp1: std_logic;
199      signal temp2: std_logic;
200
201      component half_adder
202      port ( a : in std_logic;
203             b : in std_logic;
204             s : out std_logic;
205             c : out std_logic);
206         end component;
207
208      component full_adder
209      port( a : in std_logic;
210            b : in std_logic;
211            c_in : in std_logic;
212            s : out std_logic;
213            c_out : out std_logic);
214         end component;
215  begin
216      adder_0 : half_adder port map( a => B(0),
217                                     b => A(0),
218                                     s => S(0),
219                                     c => temp0);
220
221      adder_1 : full_adder port map( a => B(1),
222                                     b => A(1),
223                                     c_in => temp0,
224                                     s => S(1),
225                                     c_out => temp1);
226
```

```
227      adder_2 : full_adder port map( a => B(2),
228                                     b => A(2),
229                                     c_in => temp1,
230                                     s => S(2),
231                                     c_out => temp2);
232
233      adder_3 : full_adder port map( a => B(3),
234                                     b => A(3),
235                                     c_in => temp2,
236                                     s => S(3),
237                                     c_out => S(4));
238
239  end strucRCA;
240
241
242
```

Figure 4.2 Structural Implementation of RCA

As seen from our representative simulation plot below, the structural implementation of the 4-bit RCA was successful. The output S(4) is our carry out of the overall RCA addition process; and hence we can look at the cursor selection to validate our design.

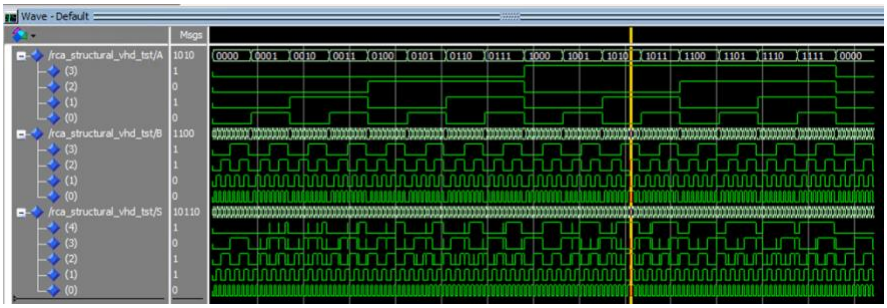

Figure 4.3 Simulation Plot of Structural Implementation of RCA

Behavioral Implementation:

The behavioral Implementation of the RCA was based on the same combinational logic; however, it describes the design on the same abstraction level as the Boolean expression of each of the adders. As seen in figure 4.4, the RCA adder uses XOR, AND and OR gates in order to 'ripple' the carry which is stored into a set of temporary signals, declared at the start of the architecture. This design allows for perhaps less control over the output, as mistakes within the logic expression will directly change the output.

```
254  architecture behRCA of rca_behavioral is
255
256    signal temp0 : std_logic;
257    signal temp1 : std_logic;
258    signal temp2 : std_logic;
259    signal temp3 : std_logic;
260
261  begin
262
263    S(0) <= B(0) XOR A(0);
264    temp0 <= B(0) AND A(0);
265
266
267    S(1) <= (B(1) XOR A(1)) XOR temp0;
268    temp1 <= (B(1) AND A(1)) OR (B(1) and temp0) OR (A(1) and temp0);
269
270    S(2) <= (B(2) XOR A(2)) XOR temp1;
271    temp2 <= (B(2) AND A(2)) OR (B(2) and temp1) OR (A(2) and temp1);
272
273    S(3) <= (B(3) XOR A(3)) XOR temp2;
274    S(4) <= (B(3) AND A(3)) OR (B(3) AND temp2) OR (A(3) AND temp2);
275
276    end behRCA;
277
```

Figure 4.4 Behavioral Implementation of 4-Bit RCA

An examination of the simulation plot gives an insight on how the logical operations are nearly identical to the structural implementation of our design. This allows us to believe that both implementations follow through from our schematic presented in fig 4.1. For example, we can observe that the addition of 0110 (6) and 1100 (12) is 10010 (18); which follows in our representative simulation plot.

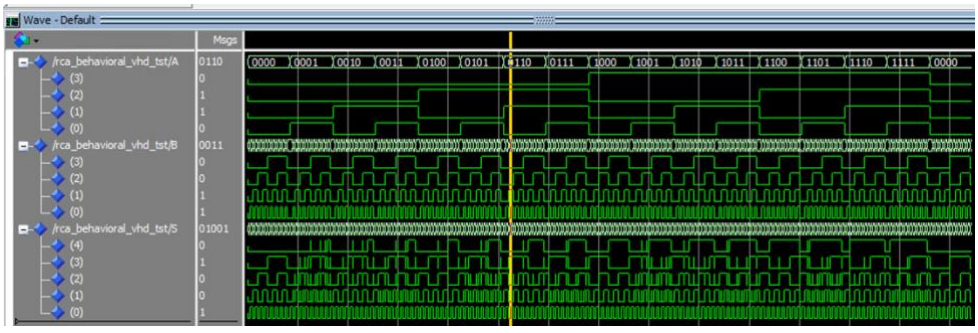

Figure 4.5 Simulation Plot of Behavioral Implementation of RCA

Implementation of BCD Adder:

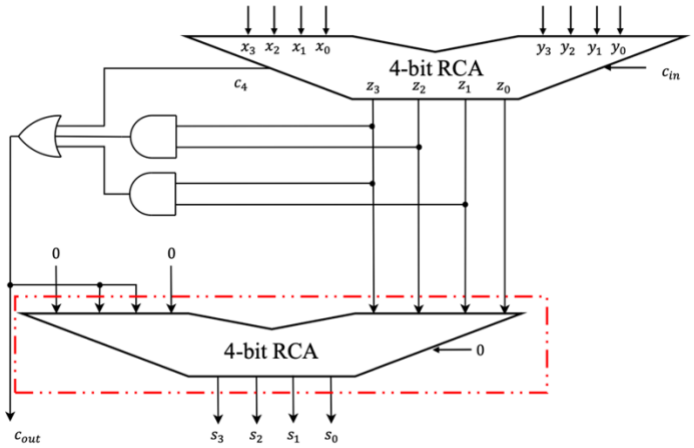

Figure 5.1 Schematic of BCD Adder Using Two RCAs

Structural Implementation:

A BCD adder can be implemented using two 4 bit RCAs as seen in figure 5.1. Following this schematic, the structural implementation of the BCD adder instantiates two RCAs that we had design earlier in the previous section. Hence, the two RCAs were instantiated and the last three digits of the first RCA, are combined using two AND gates and one OR gate with the carry of said RCA instantiation, to create the carry out of the overall BCD. Furthermore, the temporary signals z, which are apparent within our schematic and within our implementation, act as a buffer signal that hold the output of the first RCA.

```
290  architecture strucBCD of bcd_adder_structural is
291
292    signal z0 : std_logic;
293    signal z1 : std_logic;
294    signal z2 : std_logic;
295    signal z3 : std_logic;
296    signal c0 : std_logic;
297    signal c1 : std_logic;
298    signal placeholder :std_logic;
299  component rca_structural
300      port( A: in std_logic_vector (3 downto 0);
301            B: in std_logic_vector (3 downto 0);
302            S: out std_logic_vector (4 downto 0));
303  end component;
304    begin
305  rca1 : rca_structural port map( A => A,
306                                  B => B,
307                                  S(0) => z0,
308                                  S(1) => z1,
309                                  S(2) => z2,
310                                  S(3) => z3,
311                                  S(4) => c0);
312
313
314    c1 <= (z3 AND z1) OR (z2 AND z3) OR c0;
315
316
317  rca2 : rca_structural port map( A(0) => z0,
318                                  A(1) => z1,
319                                  A(2) => z2,
320                                  A(3) => z3,
321                                  A(3) => z3;
322                                  B(0) => '0',
323                                  B(1) => c1,
324                                  B(2) => c1,
325                                  B(3) => '0',
326                                  S(0) => S(0),
327                                  S(1) => S(1),
328                                  S(2) => S(2),
329                                  S(3) => S(3),
330                                  S(4) => placeholder);
331
332    C <= c1;
333
    end strucBCD;
```

Figure 5.2 Structural Implementation of BCD Adder

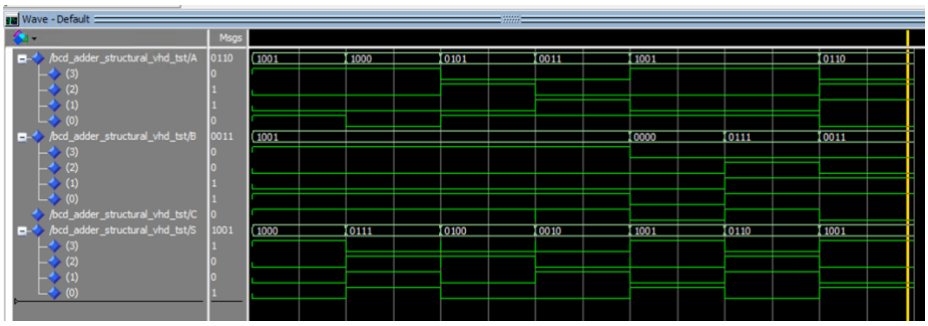

Figure 5.3 Representative Simulation Plot for the Structural Implementation of BCD Adder

Behavioral Implementation:

The behavioral implementation of the BCD adder uses the arithmetic operations of VHDL. Essentially, the overflow of the sum is calculated using the VHDL arithmetic operations and tested using conditional assignment. Whenever the sum of a specific internal operation of the BCD reaches 9, the count is recorded in the adjust signal. This implementation is based on the implementation of a BCD adder within the Stephen Brown and Zvonko Vranesic book; 'Fundamentals of Digital Logic with VHDL Design'.

```
345  ARCHITECTURE behBCD of bcd_adder_behavioral is
346    signal z : std_logic_vector (4 downto 0);
347    signal temp : std_logic_vector (4 downto 0);
348    signal adjust : std_logic;
349  begin
350    z <= ('0'& A) + B;
351    adjust <= '1' when z > 9 else '0';
352    temp <= z when (adjust = '0') else z + 6;
353    S(0) <= temp(0);
354    S(1) <= temp(1);
355    S(2) <= temp(2);
356    S(3) <= temp(3);
357    C <= temp(4);
358
359
360  end behBCD;
361
```

Figure 5.4 Behavioral Implementation of BCD Adder

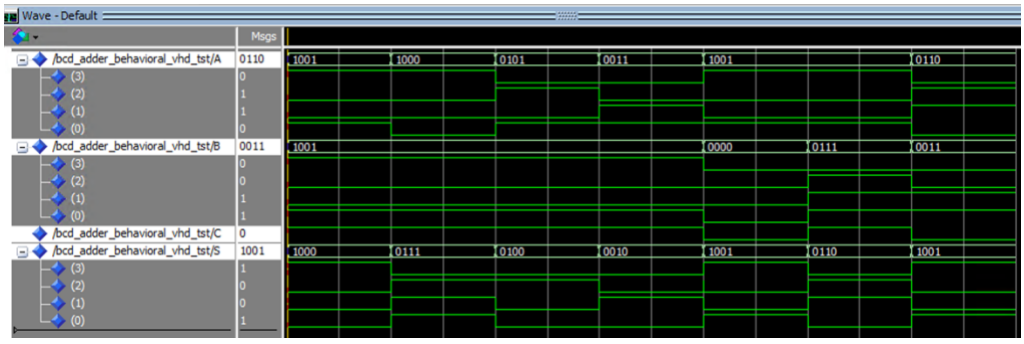

Figure 5.5 Simulation Plot of the Behavioral Implementation of BCD Adder

Part 5 – Conclusions

	4-bit circular barrel shifter		RCA		One-digit BCD adder	
	Structural	Behavioral	Structural	Behavioral	Structural	Behavioral
Logic Utilization (in LUTs)	5 / 32,070 (< 1 %)	5 / 32,070 (< 1 %)	4 / 32,070 (< 1 %)	4 / 32,070 (< 1 %)	7 / 32,070 (< 1 %)	5 / 32,070 (< 1 %)
Total pins	10 / 457 (2 %)	10 / 457 (2 %)	13 / 457 (3 %)	13 / 457 (3 %)	13 / 457 (3 %)	13 / 457 (3 %)

Figure 6 Logic Utilizations and Total Pins Recordings

For most of the designs implemented throughout this lab, the structural implementation and the behavioral implementation are equal in the logic utilizations and the number of pins needed to implement the design. However, this differs within the implementations of the one-digit BCD adder; the structural implementation seems to require a higher logical utilization. This essentially means that the structural implementation needs a higher number of ALMs to implement the design. This leads to a hypothesis that behavioral implementations are more efficient when it comes to ALMs. This is quite logical as the structural implementation requires multiple layers of further structural implementations. Structural implementations require more ‘space’ on an FPGA or in other words; a structural implementation fills more ‘space’ on the board. However, this investigation is quite limited as our scope of investigation is restricted to the implementation of one device that could be an anomaly. Further investigation is required in order to reach a consistent conclusion.