DSD gruppe 45
Exercise 2-4

Lasse Østerberg Sangild
20114274

Simon Alexander Alsing
201304202

April 6, 2016

Contents

2 Exercise 2

2.1 The half-adder

2.1.1 Dataflow style

```
1  ------------------ Dataflow_HA ------------------------
2   -------------- Library statements -------------------
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.all;
5
6  -- Entity declaration half_adder_dataflow --
7  entity half_adder_dataflow is
8     port ( a, b       : in std_logic;
9            sum, carry : out std_logic
10          );
11 end half_adder_dataflow;
12
13 -- Architecture dataflow --
14 architecture dataflow of half_adder_dataflow is
15 begin
16    sum   <= a xor b;
17    carry <= a and b;
18
19 end dataflow;
```

Figure 1: Half adder in dataflow style.

The RTL view of the dataflow style implementation of the half adder is shown in figure 2.

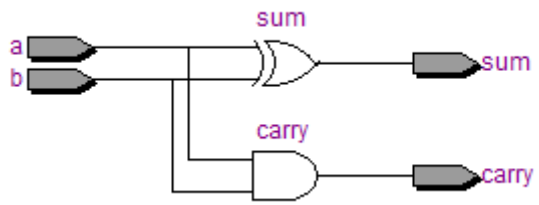


Figure 2: RTL view of Halfadder in dataflow style.

2.1.2 Behavioural style

```
------------------ Behavioural_HA ---------------------
 -------------- Library statements ------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Entity declaration half_adder--
entity half_adder_behavioural is
   port ( a, b        : in std_logic;
          sum, carry  : out std_logic
         );
end half_adder_behavioural;

-- architecture behavioural --
architecture behavioural of half_adder_behavioural is
begin
process(a,b)
begin
   if a = '1' then
      if b = '1' then
         sum   <= '0';
         carry <= '1';
      else
         sum   <= '1';
         carry <= '0';
      end if;
   else
      carry <= '0';
      sum   <= b;
   end if;
end process;
end behavioural;
```

Figure 3: Code used for implementation of a half-adder using the behavioural coding style.

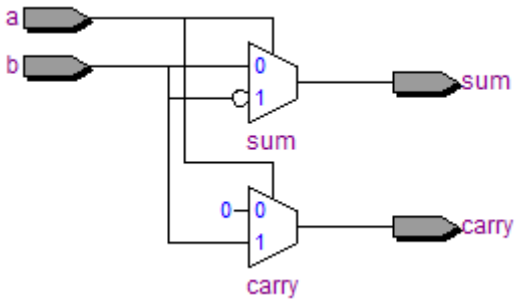


Figure 4: RTL view of half-adder in Behavioural style.

The RTL view of the Behavioural style implementation of the half-adder is shown in fig 4.

2.1.3 Structural style

```
1  ----------------- half_adder_structrual ----------------------
2   -------------- Library statements -------------------
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.all;
5
6  -- Entity declaration half_adder--
7  entity half_adder_structrual is
8     port ( A, B         : in std_logic;
9            SUM, CARRY   : out std_logic
10          );
11 end half_adder_structrual;
12
13 -- architecture structrual --
14 architecture structrual of half_adder_structrual is
15 begin
16 u1: entity work.and_2 port map(a=>A ,b=>B, carryS=>CARRY);
17 u2: entity work.xor_2 port map(a=>A, b=>B, sum=>SUM);
18 end structrual;
```

Figure 5: Code used for implementing the half-adder using structural coding style.

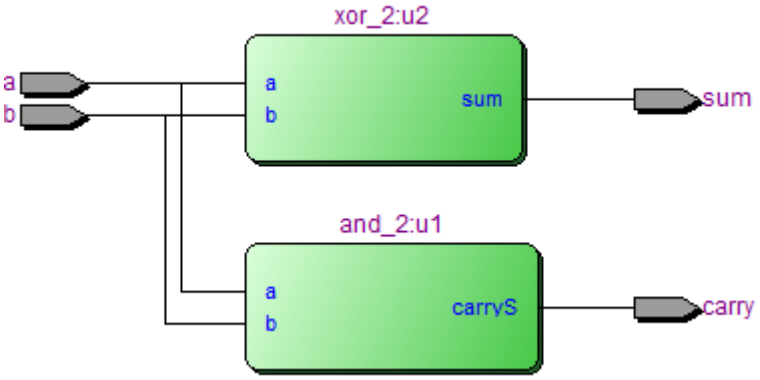


Figure 6: RTL view of half-adder in Structural style.

The RTL view of the Structural style implementation of the half adder is shown in fig 6.

2.1.4 RTL views and differences

The RTL view of a dataflow implementation of the half-adder is shown on figure 2. Compared to the Behavioural style seen on figure 4 and the structural style seen on figure 6, we can see that the dataflow tells us directly how the implementation is done, whereas the others are box implementations. The RTL view of the half-adder gives some information as to which values are to be expected, but not which gates are used to do the calculation. The structural style is a complete box implementation, it gives no information of what a or b could be, or how sum and carry are calculated. It demands the user to accept that in an earlier stage the used parts have been implemented correctly.

In figure 8 you can see the Technology Map View(TMV) of the half-adder. This is the same for dataflow and behavioural style, hence only one picture for them. The TMV for the structural style is seen on figure 7

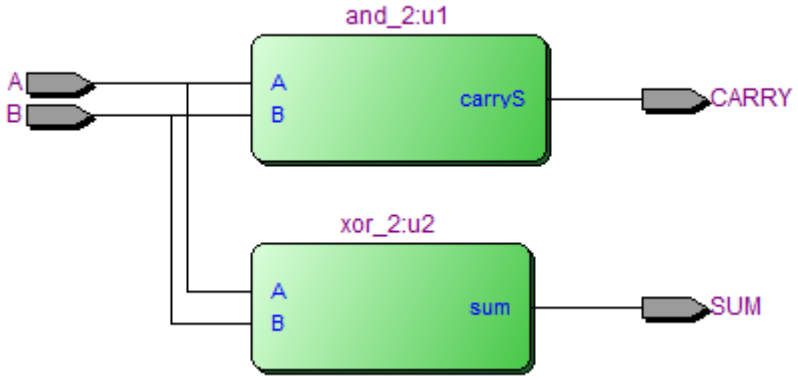


Figure 7: Technology Map View of structural style.

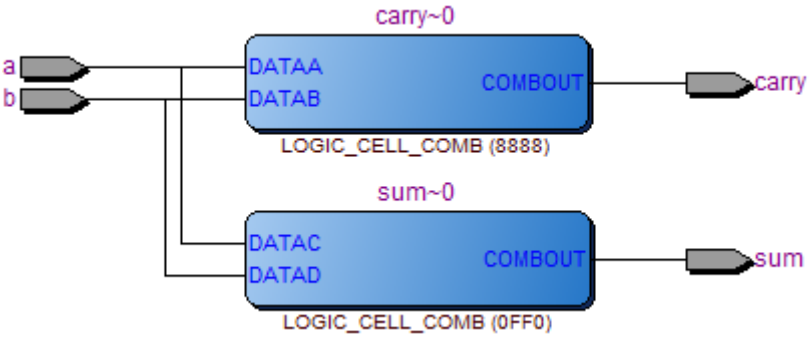


Figure 8: Technology Map View of dataflow and behavioural style.

2.1.5 Timing Simulation

The timing simulation of the different styles are shown in figures 9, 12 and 15. It is apparent that both static-1-hazard and static-0-hazard are present. They are all having hazards at the same time, so even though it is programmed in different styles the outcome is the same. Looking at these three styles in the functional simulation, shown in figures 10, 13 and 16, it is apparent that the timing hazards are not present in this type of diagram. The simulation is the same, except that the timing simulation take static hazards into account. In figure 11, 14 and 17 the resource usage is shown for the different coding styles. As expected the resource usage is the same for all of them.

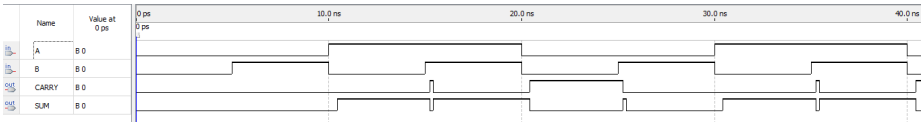


Figure 9: Timing simulation of structural style.

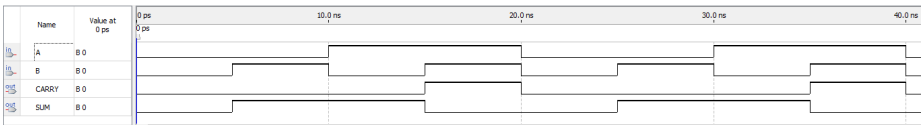


Figure 10: Functional simulation of structural style.

Total logic elements	2 / 33,216 (< 1 %)
Total combinational functions	2 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 11: Resource usage for the structural half adder implementation.

2.1.5.1 Structural

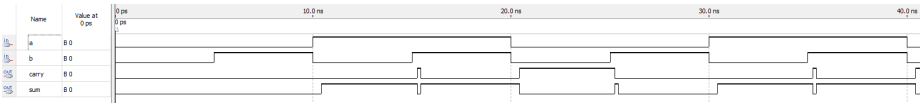


Figure 12: Timing Simulation of behavioural style.

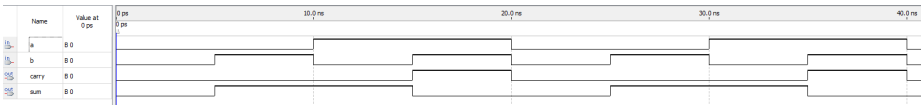


Figure 13: Functional Simulation of behavioural style.

Total logic elements	2 / 33,216 (< 1 %)
Total combinational functions	2 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 14: Resource usage for the behavioural half adder implementation.

2.1.5.2 Behavioural

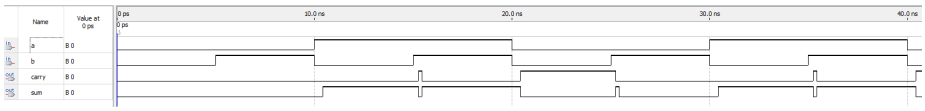


Figure 15: Functional simulation of dataflow style.

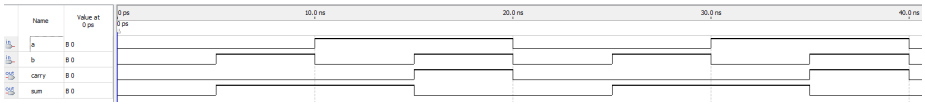


Figure 16: Functional simulation of dataflow style.

Total logic elements	2 / 33,216 (< 1 %)
Total combinational functions	2 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 17: Resource usage for the dataflow half-adder implementation.

2.1.5.3 Dataflow

2.2 The full-adder

2.2.1 Design and implementation

The design and implementation, as seen in figure 18 was done using structural style (lines 15 and 16), where a structural and a dataflow half-adder is used, and dataflow style (line 17).

```
1  -------------- Library statements ------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4
5  -- Entity declaration full_adder--
6  entity full_adder is
7     port (FAa, FAb, FAcin : in std_logic;
8           FAsum, FAcout   : out std_logic;
9           FAc1, FAc2, s1  : buffer std_logic
10          );
11 end full_adder;
12
13 architecture FA of full_adder is
14 begin
15 h1: entity work.half_adder_structrual port
   ↪  map(a=>FAa,b=>FAb,sum=>s1,carry=>FAc1);
16 h2: entity work.half_adder_dataflow port
   ↪  map(a=>s1,b=>FAcin,sum=>FAsum,carry=>FAc2);
17 FAcout <= FAc1 or FAc2;
18 end FA;
```

Figure 18: Implementation of a full-adder using a combination of dataflow and structural coding styles.

2.2.2 Verification

In figure 19 the RTL view is shown. It shows that the design is synthesized with two half-adders and an OR-gate at the end to determine the carry out. Just as written in figure 18 lines 15-17.

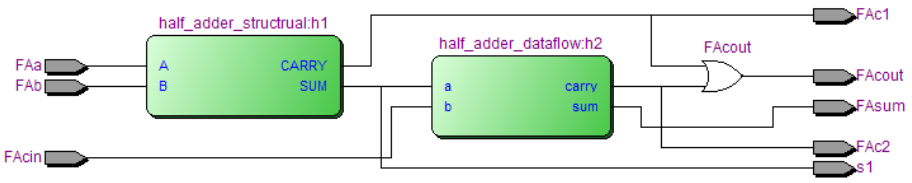


Figure 19: RTL view of the full-adder.

The flattened RTL view in figure 20 shows the same. That the structural half-adder is implemented using an XOR- and an AND-gate structurally, whereas the dataflow half-adder implements them directly.

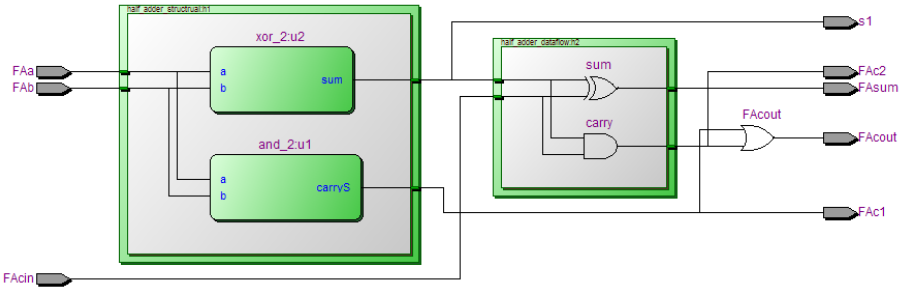


Figure 20: RTL Flat view of the full-adder.

Figure 21 shows the usage of dataflow and behavioural. The blue logic-cells are the dataflow style inputs and outputs, whereas the green ones are structural style. The full-adder is implemented as two logic cells (dataflow style) and two structurals.

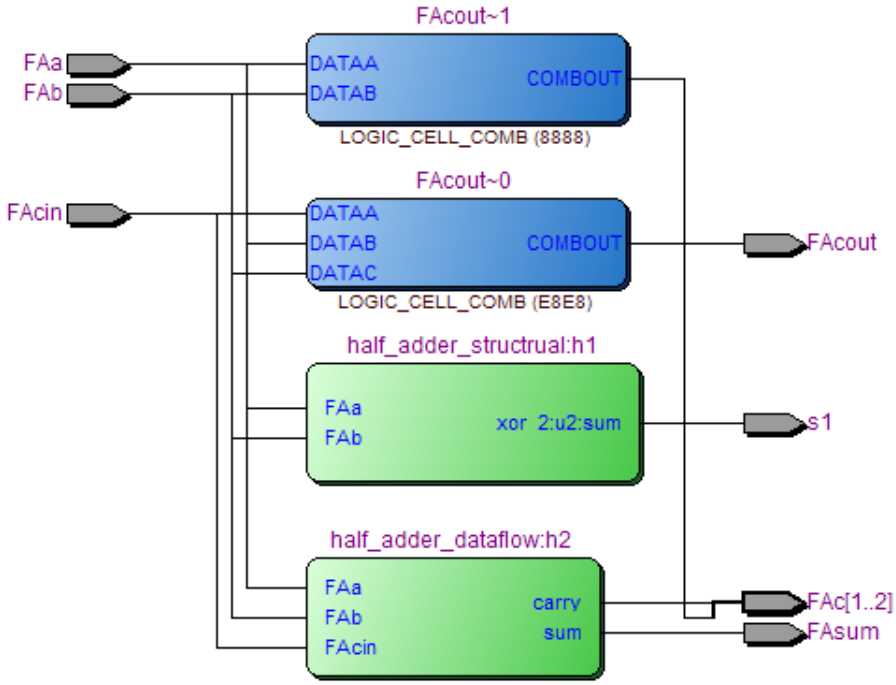


Figure 21: TM Post Mapping view of the full-adder.

The implementation uses five logical elements (figure 22). The two half-adder each uses an XOR- and an AND-gate and the full-adder then uses an OR-gate to calculate the carry.

Total logic elements	5 / 33,216 (< 1 %)
Total combinational functions	5 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	8 / 475 (2 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 22: Resource usage of the full-adder.

2.2.3 Validation and simulation

Figure 23 shows a simulation on the basis of table 1. The interesting parts here are FAcout and FAsum. As seen in the simulation FAcout is high whenever at least two of the inputs are high. FAsum is high when one or three inputs are high.

cin\FAa FAb	00	01	10	11
0	000	001	010	011
1	100	101	110	111

Table 1: Table showing all the possible input combinations for the full-adder.

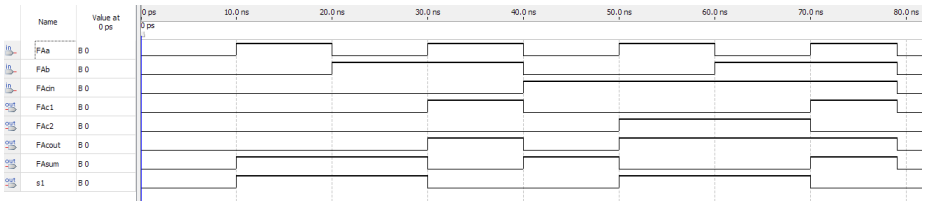


Figure 23: Simulation of the full-adder showing all possibilities.

2.3 Test

Figure 24: Picture of the full adder implementation on the DE2 board with the input carry =1, a=0, b=1 and output 010

2.4 The Four-Bit-Adder

As seen on figure 25 the four-bit-adder is implemented in a structural coding style. Figure 26 is a timing simulation with simultaneous change of all of the nine inputs. The static-0- and 1-hazards still exists because of the delay from the carries. Note that sum[0] is not affected by this because the first gate has it's carry input hardware-wise and hence does not depend on the output from another gate. The other sums are delayed with an additional gate delay, so that sum[0] has 1 gate delay, sum[1] has two gate delays (it depends on the first gate to finish) and so on. In total a four bit adder is delayed with 4 gate delays. From the timing simulation an estimated 9.5 ns is the total gate delay and hence the total time for the outputs to stabilize. The four-bit-adder is limited by the structure of dependence, since it has to wait on several gatedelays to process.
A weird thing is that the cout and sum[3] responds to the changing inputs faster than c2 and c3 and sum[1] and sum[2]. We are not fully aware as to why, but this might be cause by the placements of the gates on the chip.
In figure 27 the implementation on the DE2 board is shown. SW0 is carry in, while sw[1-4] is a inputs and sw[5-8] is b inputs. As expected with the inputs cin $= 1_2$, A[0-3] $= 0010_2 = 2_{10}$, B[0-3] $= 1101_2 = 11_{10}$, we get LED $= 01110_2 = 14_{10}$.

```
1  ------------------------ four bit adder -----------
2  -------------- Library statements ------------------------
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.all;
5
6  -- Entity declaration four_bit_adder--
7  entity four_bit_adder is
8     port ( cin        : in std_logic;
9            A, B       : in std_logic_vector(3 downto 0);
10           sum        : out std_logic_vector(3 downto 0);
11           cout       : out std_logic;
12           c1, c2, c3 : buffer std_logic
13       );
14 end four_bit_adder;
15
16 architecture fba of four_bit_adder is
17 begin
18 fa1: entity work.full_adder port map(FAa=>A(0), FAb=>B(0), FAcin
   ↪  => cin, FAsum => sum(0), FAcout => c1);
19 fa2: entity work.full_adder port map(FAa=>A(1), FAb=>B(1), FAcin
   ↪  => c1, FAsum => sum(1), FAcout => c2);
20 fa3: entity work.full_adder port map(FAa=>A(2), FAb=>B(2), FAcin
   ↪  => c2, FAsum => sum(2), FAcout => c3);
21 fa4: entity work.full_adder port map(FAa=>A(3), FAb=>B(3), FAcin
   ↪  => c3, FAsum => sum(3), FAcout => cout);
22 end fba;
```

Figure 25: Implementation of the four-bit-adder using structural coding style.

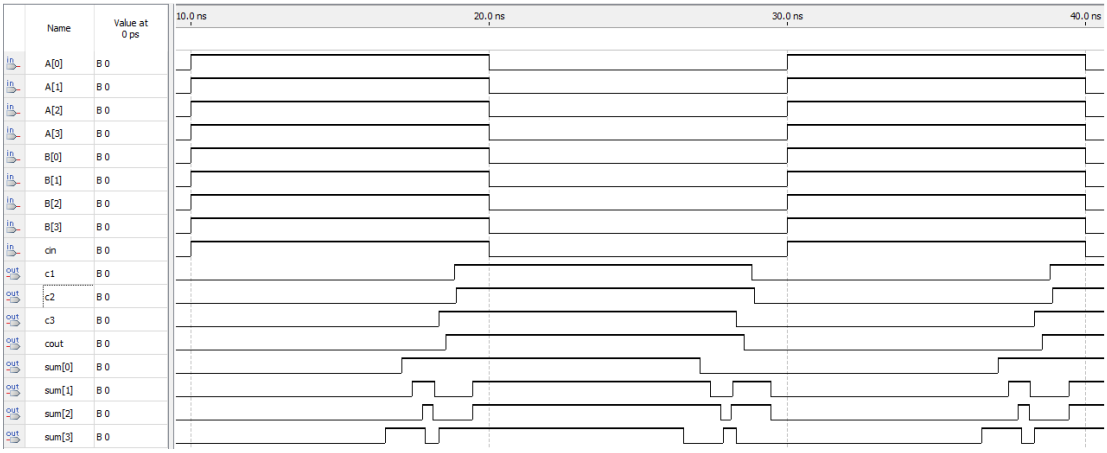


Figure 26: Timing simulation of the four-bit-adder with simultaneous change of all inputs

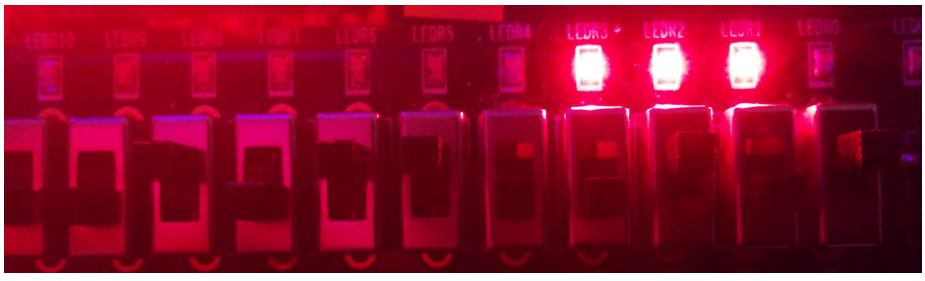


Figure 27: The four-bit-adder on the DE2 board. c = 1_2, A[0-3]=0010_2 = 2_{10}, B[0-3]=1101_2 = 11_{10}, LED = 01110_2 = 14_{10}

3 Exercise 3

3.1 Signed and unsigned arithmetic

3.1.1 Simple four-bit unsigned adder

Typecasting is used in line 15 in figure 28.

```
1  ----- Simple implementation of a four_bit_adder -----
2
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.all;
5  USE ieee.numeric_std.all;
6
7  entity four_bit_adder_simple is
8     port( A,B : in std_logic_vector(3 downto 0);
9           sum : out std_logic_vector(3 downto 0)
10          );
11 end four_bit_adder_simple;
12
13 architecture unsigned_impl of four_bit_adder_simple is
14 begin
15    sum <= std_logic_vector(unsigned(A)+unsigned(B));
16 end unsigned_impl;
```

Figure 28: Code used for implementation of a simple four-bit unsigned adder.

3.1.1.1 Verification The truth table for one half-adder is shown in table 2 (the result is equal to that of an XOR-gate, because we disregard the carry). The simulation in figure 29 follows the truth table.

A\B	0	1
0	0	1
1	1	0

Table 2: Truth Table(TT) for the simple four-bit-adder.

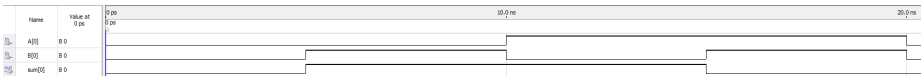


Figure 29: Simulation of one half-adder from the implementation of the four-bit-adder.

3.1.1.2 Test Testing for the same possibilities as in figure 29 is shown in figure 31 and the used code is seen in figure 30.

```
----- Tester class -----

LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity FBA_Simple_Tester is
   port( a,b  : in std_logic_vector(3 downto 0);
         SW   : in std_logic_vector(8 downto 0);
         LEDR : out std_logic_vector(3 downto 0);
         sum  : out std_logic_vector(3 downto 0)
   );
end FBA_Simple_Tester;

architecture test of FBA_Simple_Tester is
begin
   FA1: entity work.four_bit_adder_simple(unsigned_impl) port map(A(0)=>a(0), B(0)=>b(0), sum(0)=>sum(0));
   FA2: entity work.four_bit_adder_simple(unsigned_impl) port map(A(1)=>a(1), B(1)=>b(1), sum(1)=>sum(1));
   FA3: entity work.four_bit_adder_simple(unsigned_impl) port map(A(2)=>a(2), B(2)=>b(2), sum(2)=>sum(2));
   FA4: entity work.four_bit_adder_simple(unsigned_impl) port map(A(3)=>a(3), B(3)=>b(3), sum(3)=>sum(3));
end test;
```

Figure 30: Code used for the tester.

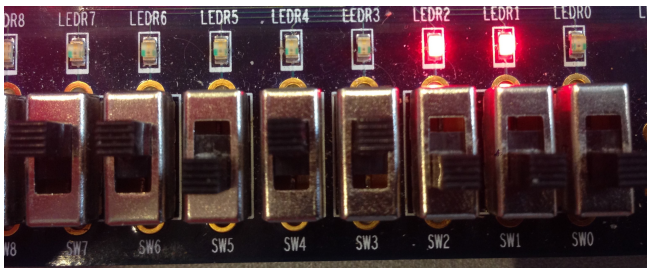


Figure 31: Test of all of the possibilities for the half-adder (see TT in figure 2).

3.1.2 Simple four-bit signed adder

```
architecture signed_impl of four_bit_adder_simple is
begin
   sum <= std_logic_vector(signed(A)+signed(B));
end signed_impl;
```

Figure 32: Code added in the end of figure 28 to implement the signed version.

3.1.2.1 Test The tester for the four-bit-adder is implemented by editing the architecture in figure 30 to the code shown in figure 33. The testing result on the DE2-board is shown in figure 34.

```
14 architecture test of FBA_Simple_Tester is
15 begin
16    --FA1: entity work.four_bit_adder_simple(unsigned_impl) port
      ↪  map(A(0)=>a(0), B(0)=>b(0), sum(0)=>sum(0));
17    --FA2: entity work.four_bit_adder_simple(unsigned_impl) port
      ↪  map(A(1)=>a(1), B(1)=>b(1), sum(1)=>sum(1));
18    --FA3: entity work.four_bit_adder_simple(unsigned_impl) port
      ↪  map(A(2)=>a(2), B(2)=>b(2), sum(2)=>sum(2));
19    --FA4: entity work.four_bit_adder_simple(unsigned_impl) port
      ↪  map(A(3)=>a(3), B(3)=>b(3), sum(3)=>sum(3));
20    i1:  entity work.four_bit_adder_simple(signed_impl)
21     port map(   A=>SW(4 downto 1),
22       B=>SW(8 downto 5),
23       sum=>LEDR(3 downto 0));
24 end test;
```

Figure 33: Code added in the end of figure 28 to implement the signed version.

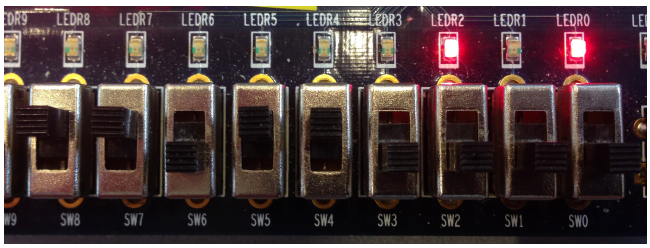


Figure 34: Test of all of the possibilities for the half-adder. A = SW(1-4) = 1000_2, B = SW(5-8) = 1101_2. This still follows the TT in table 2 as expected.

The reason for the same output in figures 31 and 34 is because an unsigned number 1111_2 is equal to 15_{10}, whereas a signed $1111_2 = 0000_2 + 1_2 = -1_{10}$ (turns negative because MSB = 1). But when any number is added to this it gives the same result due to the lack of carries (overflow).
The unsigned and signed architectures are implemented under the same entity (figures 28 and 32). This is done because they use the same port mapping and performs tasks which fall under the "umbrella" four-bit-adder-simple.

3.1.3 Four-bit-adder with carry

The code for the complete four-bit-adder with carry is shown in figure 35. It is still called four_bit_adder_simple because it is built upon the groundwork laid earlier in this exercise. Lines 12-20 shows the unsigned implementation, and the lines 22-30 shows the signed implementation.

```vhdl
----- Simple implementation of a four_bit_adder -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity four_bit_adder_simple is
   port(A,B : in std_logic_vector(3 downto 0);
        sum : out std_logic_vector(4 downto 0);
        cin : std_logic);
end four_bit_adder_simple;

architecture unsigned_impl of four_bit_adder_simple is
-- create 5 bit signals to ensure correct addition --
signal sum_5b, cin_5b : std_logic_vector(0 to 4); -- initialized
 ↪  to left most value hence both signals are 00000
constant sum_4b : std_logic_vector(0 to 3) := "0000";
begin
   cin_5b <= sum_4b & cin;
   sum_5b <= std_logic_vector(resize(unsigned(A),5) +
    ↪  resize(unsigned(B),5) + unsigned(cin_5b));
   sum <= sum_5b;
end unsigned_impl;

architecture signed_impl of four_bit_adder_simple is
-- create 5 bit signals to ensure correct addition --
signal sum_5b, cin_5b : std_logic_vector(0 to 4); -- initialized
 ↪  to left most value hence both signals are 00000
constant sum_4b : std_logic_vector(0 to 3) := "0000";
begin
   cin_5b <= sum_4b & cin;
   sum_5b <= std_logic_vector(resize(signed(A),5) +
    ↪  resize(signed(B),5) + signed(cin_5b));
   sum <= sum_5b;
end signed_impl;
```

Figure 35: Code used for implementing a four-bit-adder with carry.

3.1.3.1 Test The results of the unsigned and signed tests are shown in figures 36 and 37. The results are for figure 36 $1101_2 + 1000_2 + 0 = 13_{10} + 8_{10} = 21_{10} = 10101_2$ because the overflow of our adder is shown as carry out (LEDR(4)). The signed implementation shown in figure 37 shows $1101_2 + 1000_2 + 1_2 = (-3_{10}) + (-8_{10}) + 1_{10} = -10_{10} = 10110_2$ as the LED's show. Note that the output, in contrast to subsection 3.1.2, is different for the signed and unsigned versions. This is due to the resize function, which keeps the original number, but adds a 1 as MSB if it is signed and a 0 as MSB if it is unsigned. Comparing figures 37 and 38 the signed and unsigned implementation both result in the same output. Note that the two outputs are actually showing the decimal values of -10(signed) and 22(unsigned). In the signed version it is also important to note that the MSB gives the sign of the number. This could

have been implemented better by putting the MSB at the left most LED of the DE2-board.

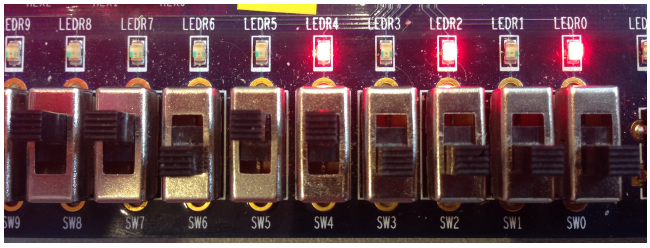


Figure 36: Test of the four-bit-adder with carry, showing the same result on LEDR[0-3] as figure 34 with the switches set to the same (A = 1101, B = 1000, c = 0), but with the added functionality of LEDR[4], giving the full result.

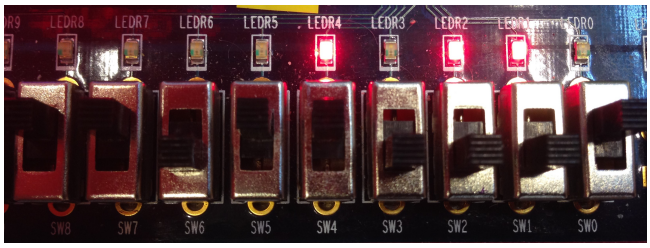


Figure 37: Test of the signed four-bit-adder with carry, using the same set-up as figure 36, but now with the carry (c) set to 1 (SW[0]).

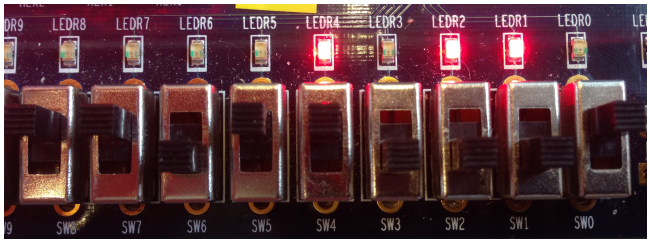


Figure 38: Test of the unsigned four-bit-adder with carry, using the same set-up as figure 37.

3.2 Concatenation

3.2.1 Basic concatenation

As seen in figure 39 the shift once left is implemented in lines 16-17, shift twice right in lines 18-19 and the rotate thrice right in lines 20-21. The if-statement is controlled by three switches,SW(7 downto 5), and the input entered using SW(4 downto 0) is displayed and manipulated by activating one or none of the switches (SW(7 downto 5)). This is done to make the output easier to understand.

```
1  ----- LIBRARIES -----
2  LIBRARY ieee;
3  use ieee.std_logic_1164.all;
4  USE ieee.numeric_std.all;
5
6  entity shift_div is
7  port ( SW   : in std_logic_vector(7 downto 0);
8         LEDR : out std_logic_vector(4 downto 0)
9         );
10 end shift_div;
11
12 architecture shift of shift_div is
13 begin
14 process(SW)
15    begin
16    if(SW(5) = '1')then --Shift once to the left
17       LEDR(4 downto 0) <= std_logic_vector(SW(3 downto 0) &
          ↪  '0');
18    elsif(SW(6) = '1')then --Shift twice to the right
19       LEDR(4 downto 0) <= std_logic_vector("00" & SW(4 downto
          ↪  2));
20    elsif(SW(7) = '1')then --Rotate thrice to the right
21       LEDR(4 downto 0) <= SW(2) & SW(1) & SW(0) & SW(4) & SW(3);
22    else --Show original input
23       LEDR(4 downto 0) <= SW(4 downto 0);
24    end if;
25 end process;
26 end shift;
```

Figure 39: Code used for implementing the three operations using the concatenation operator.

3.2.2 Implementation

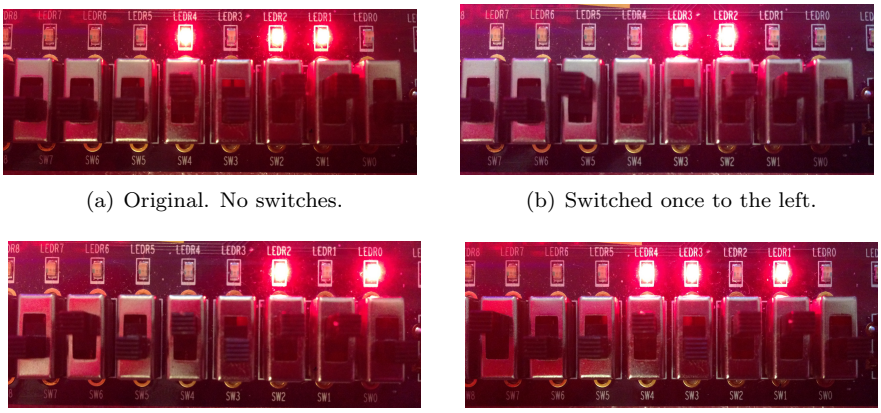
Implementation of the single functions to investigate the number of LE's used for each was done by commenting out every line between line 14 and 26 in figure 39 except for the one in question(Leaving lines 17, 19 and 21 one at a time). The three possibilities all use zero LE's.
These implementations are done by routing the inputs to outputs using wires and no LE's as no manipulation is to be done.

3.2.3 Test

In figure 4.2.2 all of the possible outcomes are shown. Using the starting configuration of "10110" the outputs are expected to be as shown in table 3. The expected and realized outcomes match.

	Output
Start	10110
Shift once left	01100
Shift twice right	00101
Rotate thrice right	11010

Table 3: Expected outputs



(a) Original. No switches.

(b) Switched once to the left.

(c) Switched twice to the right.

(d) Rotated thrice to the right.

Figure 40: The original and altered states as coded in figure 39.

3.3 Multiplication

The code used for creating the 8 bit multiplier is seen in figure 41. The output will be of a maximum size of $(2^8 - 1) \times (2^8 - 1) = (256 - 1) \times (256 - 1) = 65025_{10} = 1111\ 1110\ 0000\ 0001_2$ using a maximum of 16 bits for the output.

3.3.1 Multiplication component

```
1  ----- Libraries -----
2  LIBRARY ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  -----Entity----
7  entity multiplication is
8     port(
9     SW   : in std_logic_vector(15 downto 0);
10    LEDR : out std_logic_vector(15 downto 0)
11    );
12 end multiplication;
13
14 ----Architecture-----
15 architecture multiplier of multiplication is
16 begin
17    LEDR(15 downto 0) <= std_logic_vector(unsigned(SW(15 downto
       ↪  8)) * unsigned(SW(7 downto 0)));
18 end multiplier;
```

Figure 41: Code used for implementing the 8 bit multiplier.

3.3.2 Test

The multiplier is working as intended as seen in figure 42. It uses a staggering amount of 103 LE's.

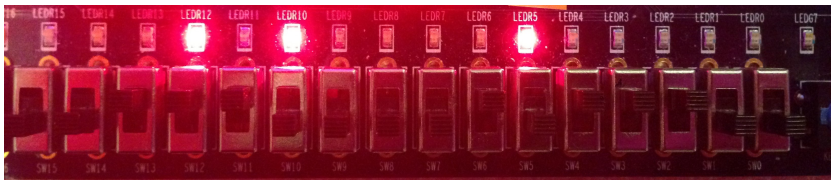


Figure 42: Test of the multiplier. $0011\ 1000 \times 0101\ 1100 = 0001\ 0100\ 0010\ 0000$.

3.3.3 Number of bits vs. number of LE's

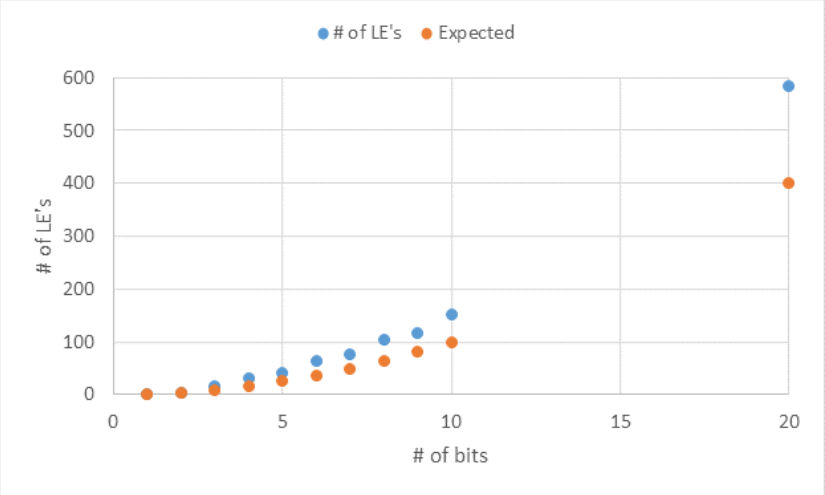


Figure 43: Graph of the number of bits vs. the number of used LE's

Input bit size	Max output bit size	# of LE's	Expected # of LE's	Diff
1	1	1	1	0
2	4	4	4	0
3	6	17	9	8
4	8	31	16	15
5	10	41	25	16
6	12	63	36	27
7	14	75	49	26
8	16	103	64	39
9	18	116	81	35
10	20	151	100	51
20	40	583	400	183
100	200	13 867	10 000	3 867
1000	1 000 000	342 999	1 000 000	-657 001

Table 4: Table showing # LE's used and expected $\left(n_{bits}^2\right)$

Table 3.3.3 shows how the algorithm in Quartus works. It is not very good at low number of input bits (if DSP is disabled) below probably an input bit size of around 200, but as we ramp up, the algorithm is really strong as seen in the example with an input size of 1000 bits.
The number of bits correlate with a power series of n_{bits}^2.
Allowing DSP uses 328 718 LE's with multiplication of 1000 bits.

3.3.4 Multiplication with constants

In this part, the code diverges a bit from what is understood by the assignment. In line 18 on figure 44, the number is typecast into an unsigned 4-bit ($15_{10} = 1111_2$). This allows the program to use a lot less resources (LED's) to output it's result. If the typecasting was not used, the program would expect the constant to be able to have a size equal to the input size (8 bits) resulting in an output using a maximum of 16 bits. Using this method, the output is allowed to only use the necessary number of bits so in stead of using 16 bits to visualize the code shown in figure 44, it only uses 12 bits.

```
1  ----- Libraries -----
2  LIBRARY ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  -----Entity----
7  entity multiplication is
8     port(
9     SW   : in std_logic_vector(7 downto 0);
10    LEDR : out std_logic_vector(11 downto 0)
11    );
12 end multiplication;
13
14 ----Architecture-----
15 architecture multiplier of multiplication is
16 constant tal : integer := 4;
17 begin
18    LEDR <= std_logic_vector(unsigned(SW) * TO_UNSIGNED(tal,4));
19 end multiplier;
```

Figure 44: Code used for implementing the constant

As seen in table 5 multiplication by any number of 2^n results in no LE's used. Any of the constants being made up from an addition of two numbers which use no LE's will use 9 LE's. From there on it becomes more complicated and no clear pattern is found.

Table 5: Multiplication with a constant

Constant	# of LE's
1	0
2	0
3	9
4	0
5	9
6	9
7	17
8	0
9	9
10	9
11	22
12	9
13	20
14	16
15	16

4 Exercise 4

4.1 Binary to 7-segment decoder using "with-select"

4.1.1 The coding

```
-----LIBRARIES-----
library ieee;
use ieee.std_logic_1164.all;

-----ENTITY-----
entity Binary_7_Segment is
   port (SW   : in std_logic_vector(3 downto 0);
         HEX0 : out std_logic_vector(0 to 6)
   );
end Binary_7_Segment;

architecture Test of Binary_7_Segment is
begin
   with SW select
   HEX0 <=  "0000001" when "0000",  -- 0
            "1001111" when "0001",  -- 1
            "0010010" when "0010",  -- 2
            "0000110" when "0011",  -- 3
            "1001100" when "0100",  -- 4
            "0100100" when "0101",  -- 5
            "0100000" when "0110",  -- 6
            "0001111" when "0111",  -- 7
            "0000000" when "1000",  -- 8
            "0001100" when "1001",  -- 9
            "0001000" when "1010",  -- A
            "1100000" when "1011",  -- b
            "0110001" when "1100",  -- C
            "1000010" when "1101",  -- d
            "0110000" when "1110",  -- E
            "0111000" when "1111",  -- F
            "1111111" when others;
end Test;
```

Figure 45: Code used for implementing the binary to seven segment display.

4.1.2 Test

Figure 46: Test of the DE2 showing a 4 as expected.

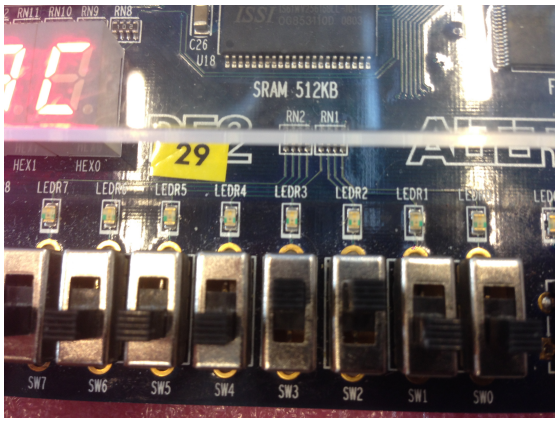


Figure 47: Test of the DE2 showing a C (12) as expected.

As seen in figure 48 there is used seven multiplexers, one for each segment.

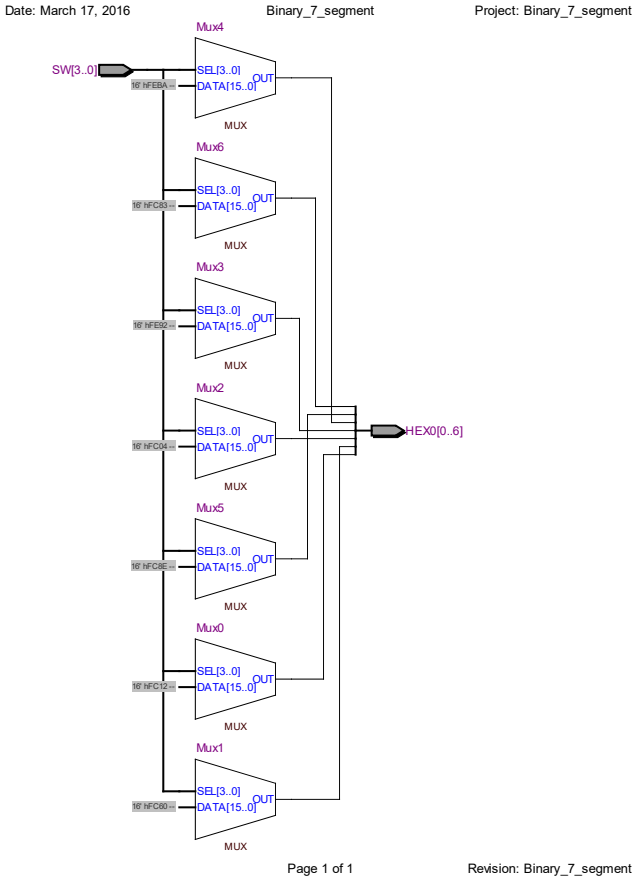


Figure 48: RTL-view of the binary to seven segment.

4.2 Demultiplexing using "when"

Implemented using a multiplexer, a demultiplexer, a slightly altered code from the seven segment exercise and a tester.

4.2.1 The code

```
1  -----Libraries-----
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  -----Entity-----
7  entity tester is
8     port( KEY : in std_logic_vector(1 downto 0);
9           SW  : in std_logic_vector(11 downto 0);
10          HEX0, HEX1, HEX2, HEX3   : out std_logic_vector(6
            ↪ downto 0)
11    );
12 end tester;
13
14 -----Architecture-----
15 architecture TEST of tester is
16 signal data_val : std_logic_vector(20 downto 0) :=
   ↪ "11111111111111111111";
17 begin
18 HEX3 <= "1111111";
19 MX : entity work.Multiplexer      port map (data => data_val,
   ↪ bin_in => SW, ab => KEY);
20 DMX : entity work.Demultiplexer   port map (data_in => data_val,
   ↪ disp0 => HEX0, disp1 => HEX1, disp2 => HEX2);
21 end TEST;
```

Figure 49: Code used for testing the demultiplexer. HEX3 is added in lines 10 and 18 to blank out the display.

```
-----LIBRARIES-----
library ieee;
use ieee.std_logic_1164.all;

-----ENTITY-----
entity bin2sevenseg is
   port (bin : in std_logic_vector(3 downto 0);
         seg : out std_logic_vector(6 downto 0)
   );
end bin2sevenseg;

architecture Test of bin2sevenseg is
begin
   with bin select
   seg <=   "1000000" when "0000",  -- 0
            "1111001" when "0001",  -- 1
            "0100100" when "0010",  -- 2
            "0110000" when "0011",  -- 3
            "0011001" when "0100",  -- 4
            "0010010" when "0101",  -- 5
            "0000010" when "0110",  -- 6
            "1111000" when "0111",  -- 7
            "0000000" when "1000",  -- 8
            "0011000" when "1001",  -- 9
            "0001000" when "1010",  -- A
            "0000011" when "1011",  -- b
            "1000110" when "1100",  -- C
            "0100001" when "1101",  -- d
            "0000110" when "1110",  -- E
            "0001110" when "1111",  -- F
            "1111111" when others;
end Test;
```

Figure 50: Code used for implementing the seven segment displays. The combinations to be selected in the "with-select" are reversed comparing to the earlier exercise.

```
1  -----LIBRARIES-----
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  -----ENTITY-----
7  entity Multiplexer is
8     port( data   : out std_logic_vector(20 downto 0);
9           bin_in : in std_logic_vector(11 downto 0);
10          ab     : in std_logic_vector(1 downto 0)
11    );
12 end Multiplexer;
13
14 -----ARCHITECTURE-----
15 architecture Mu_test of Multiplexer is
16 constant sig_On  : std_logic_vector(20 downto 0) :=
   ↪ "10000000101010111111111";
17 constant sig_Err : std_logic_vector(20 downto 0) :=
   ↪ "00001100101111010101111";
18 signal   sig_val : std_logic_vector(20 downto 0) :=
   ↪ "11111111111111111111";
19 begin
20 val_1 : entity work.bin2sevenseg port map(bin => bin_in(3 downto
   ↪ 0), seg => sig_val(6 downto 0));
21 val_2 : entity work.bin2sevenseg port map(bin => bin_in(7 downto
   ↪ 4), seg => sig_val(13 downto 7));
22 val_3 : entity work.bin2sevenseg port map(bin => bin_in(11
   ↪ downto 8), seg => sig_val(20 downto 14));
23 data <= sig_On    when ab = "11" else -- When no keys pushed
   ↪ show "On " ("11")
24        sig_Err   when ab = "01" else -- When Key1 is pushed
          ↪ show "Err" ("01")
25        sig_val   when ab = "10";     -- When Key0 is pushed
          ↪ show value of SW(11 downto 0) ("10")
26 end Mu_test;
```

Figure 51: Code used for implementing the multiplexer.

```
1  -----LIBRARIES-----
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  -----ENTITY-----
6  entity Demultiplexer is
7     port( data_in : in std_logic_vector(20 downto 0);
8           disp0, disp1, disp2 : out std_logic_vector(6 downto 0)
9           );
10 end Demultiplexer;
11
12 -----Architecture-----
13 architecture demuxer of Demultiplexer is
14 begin
15 disp0 <= data_in(6 downto 0);
16 disp1 <= data_in(13 downto 7);
17 disp2 <= data_in(20 downto 14);
18 end demuxer;
```

Figure 52: Code used for implementing the demultiplexer.

After compilation the synthesizer gave a lot of inferred latches, which was due to a path not being specified in figure 51 lines 23-25. This was solved by assigning the theoretical paths to "Err" as seen in figure 53.

```
23 data <= sig_On  when ab = "11" else
24 -- When no keys pushed show "On " ("11")
25         sig_Err when ab = "01" else
26           -- When Key1 is pushed show "Err" ("01")
27         sig_val when ab = "10" else
28           -- When Key0 is pushed show value of SW(11 downto 0)
             ↪  ("10")
29         sig_Err;
30           -- Making sure that all other possible values are set
             ↪  to error
```

Figure 53: Code used to remove inferred latches.

4.2.2 Test

(a) Showing the "On" state.

(b) Showing the "Err" state.

(c) Showing the numbers.

Figure 54: All of the possible results of pressing the keys.

4.3 Table lookup

4.3.1 The code

```
-----Libraries-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----Entity-----
entity table_lookup_vector is
   port ( SW    : in std_logic_vector(2 downto 0);
          LEDR0 : out std_logic
         );
end table_lookup_vector;

-----Architecture-----
architecture lookup of table_lookup_vector is
constant output : std_logic_vector(0 to 7) := "11010--1";
begin
   LEDR0 <= output(to_integer(unsigned'(SW(2), SW(1), SW(0))));
    ↪  --(a,b,c)
end lookup;
```

Figure 55: Code used to remove inferred latches.

4.3.2 Simulation

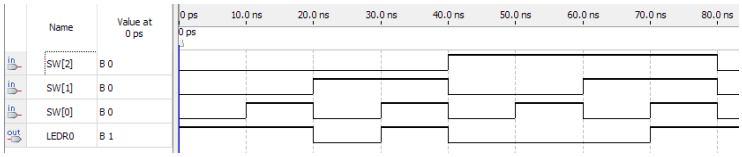


Figure 56: Functional simulation of the table lookup

4.3.3 Test

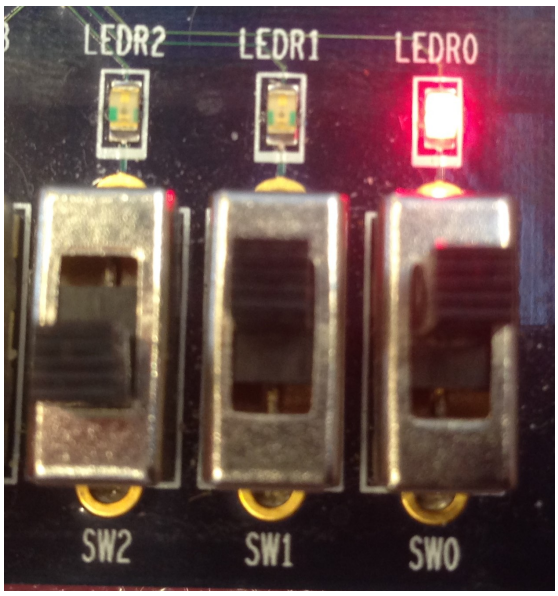


Figure 57: Test of the Table Lookup option ”011”.

4.4 Bidirectional ports (optional)

4.4.1 The code

```
library ieee;
use ieee.std_logic_1164.all;

entity bidir_test is
   port( KEY    : in std_logic_vector(1 downto 0);
         SW     : in std_logic_vector(0 downto 0);
         LEDR   : out std_logic_vector(1 downto 0);
         GPIO_1 : inout std_logic_vector(1 downto 0));
end bidir_test;

architecture bidir of bidir_test is
begin
   GPIO_1(0) <= NOT KEY(0) when SW(0) = '0' else 'Z';
   LEDR(0)   <= NOT KEY(0) when SW(0) = '0' else 'Z';
   LEDR(1)   <= GPIO_1(1)  when SW(0) = '0' else 'Z';
   GPIO_1(1) <= KEY(1)     when SW(0) = '1' else 'Z';
   LEDR(1)   <= KEY(1)     when SW(0) = '1' else 'Z';
   LEDR(0)   <= GPIO_1(0)  when SW(0) = '1' else 'Z';
end bidir;
```

Figure 58: Code used for implementing the bidirectional ports.

4.4.2 Test

Table 6: Table without the GPIO pins connected

SW(0)	KEY(0)	KEY(1)	LEDR(0)	LEDR(1)
0	0	0	0	1
0	0	1	0	1
0	1	0	1	1
1	0	0	1	1
1	0	1	1	0
1	1	0	1	1

Table 7: Table with the GPIO pins connected

SW(0)	KEY(0)	KEY(1)	LEDR(0)	LEDR(1)
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1

It is easy to see in table 6 that pushing a key, does not affect the other LEDR (KEY(0) interacts with LEDR(0) and KEY(1) with LEDR(1)). On the other hand, in tabel 7 the LEDR's are always the same, because they are linked through the GPRIO ports and both keys affect both LEDR's.

4.4.3 Simulation

If we were to simulate this functionality the std_logic library supports values besides 0 and 1. Actually it supports additionally 7 values, all of which could be used, but the "Don't care" value '-' would be our choice. This is due to the logic in reading and using it. The others indicate a range of different logic values that wouldn't be easy understandable for others to read when reading the source code.

If we were to implement this, it would simply be setting the don't care in the direction we wouldn't want the information to flow. This way the code is readable and easy to understand.