DIGITAL SYSTEMS ELECTRONICS

04OIHNX - A.A. 2024/2025

Prof. G. Masera

Laboratory assignment no. 2 - Switches, Decoders, Numbers and Displays

DUE DATE: 25/03/2025 DELIVERY DATE: 22/03/2025

GROUP 08 - Contributions:

- Daniele Becchero (308299) – 33.3%
- Bohotici Ionut Viorel (300061) – 33.3%
- Simone Viola (310779) – 33.3%

Introduction

This laboratory aims to improve our knowledge and understanding of VHDL by designing some digital circuits which involve the use of the 7-segment displays and the on-board switches.

Section 1: Controlling a 7-segment display

The first section of this laboratory experience is about the design of a decoder for a 7-segment display that must show one the following letters, according to the input combination.

Delivered files

For this project, the reference subfolder is the one named *Ex1*. Two VHDL files have been delivered.

- *decoder7.vhd*, which is the VHDL code of the decoder (top-level entity).
- *decoder7_testbench.vhd*, which is the decoder's testbench.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *decoder7.sof*, and it already includes the assignation file *DE1-SoC.qsf*. For the timing simulation, two additional files have been delivered.

Design entry

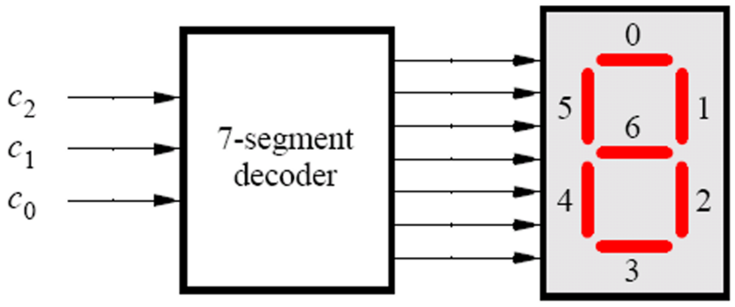


Figure 1 – Circuit structure

The decoder must drive the display to show the letters in Table 1. Where no letter is indicated, the display must be turned off (no segment is lit on). There are five different combinations, hence they can be represented by at least three bits, which are the inputs for the decoder. The display has seven different segments, so the decoder has seven different outputs. Notice that each segment is lit when driven to the logical value '0'. The output of the decoder has been set as a standard logic vector, but the bits order is reversed: the leftmost bit is at index 0, while the rightmost bit is at index 6.

C2 C1 C0	**Character**	**Decoder output**
000	H	1001000
001	E	0110000
010	L	1110001
011	O	0000001
others	blank	1111111

Table 1 - Character selection pattern for exercise 1

In the *port* statement of the entity *decoder7*, there are both the input port (which is connected to the first three switches), and the HEX0 output port (which is connected to the on-board 7-segment display number 0). This last port has a different syntax, as discussed above:

```
HEX0 : out STD_LOGIC_VECTOR(0 to 6)
```

The architecture description is based on a conditional signal assignment. The decoder output is set depending on the input selection combination. The assignment statement is shown below.

```
HEX0 <=   "1001000" when SW = "000" else     -- Letter H
          "0110000" when SW = "001" else     -- Letter E
          "1110001" when SW = "010" else     -- Letter L
          "0000001" when SW = "011" else     -- Letter O
          "1111111";                         -- Blank
```

Functional simulation

The testbench for the previous section entity is quite straightforward. First, an empty entity has been declared. Then, the previous entity was included as a component with its ports in the architecture and two new signals have been created for simulation purposes. These two signals have the same dimensions as the ports of the *decoder7* entity, and they have been used in a process to simulate different input patterns for the decoder. The statement of the *process* is provided below.

```
process
begin
     SEL <= "000";         -- Set SEL to 000
     wait for 20 ns;
     SEL <= "001";         -- Set SEL to 001
     wait for 20 ns;
     SEL <= "010";         -- Set SEL to 010
     wait for 20 ns;
     SEL <= "011";         -- Set SEL to 011
     wait for 20 ns;
     SEL <= "100";         -- Set SEL to 100
     wait;
end process;
```

The final step involves utilizing the port map statement to virtually associate the simulation signals with the decoder's ports. It is essential to note that the testbench is not designed for synthesis; thus, these connections are established virtually.

The simulation yields the results shown in Figure 2. By analysing the various input-output pairs, we can conclude that the VHDL code for the decoder functions as intended.

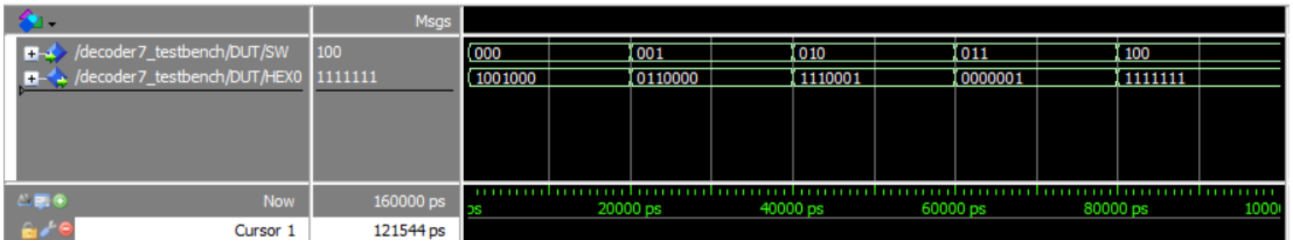

Msgs
/decoder7_testbench/DUT/SW 100
/decoder7_testbench/DUT/HEX0 1111111
000 001 010 011 100
1001000 0110000 1110001 0000001 1111111
Now 160000 ps
Cursor 1 121544 ps
ps 20000 ps 40000 ps 60000 ps 80000 ps 1000

Figure 2 – Exercise 1 simulation results

Synthesis

The VHDL code for *decoder7* is ready to be opened in a new Quartus Prime project and used to program the FPGA. First, we need to import the assignment file *DE1-SoC.qsf*. Next, we start the compilation of the entire project. Once the compilation is complete, we check for errors or warnings. As there are no critical errors or warnings, we connect the board to the PC, launch the auto-detect tool, and upload the *.sof* file to the FPGA.

At this point, the FPGA on the DE1-SoC board is correctly programmed, and we are ready to verify its functionality. Using the first three switches, the user can evaluate all the possible input combinations and confirm whether the character is displayed according to the specifications. Two combinations are shown in the figure below.

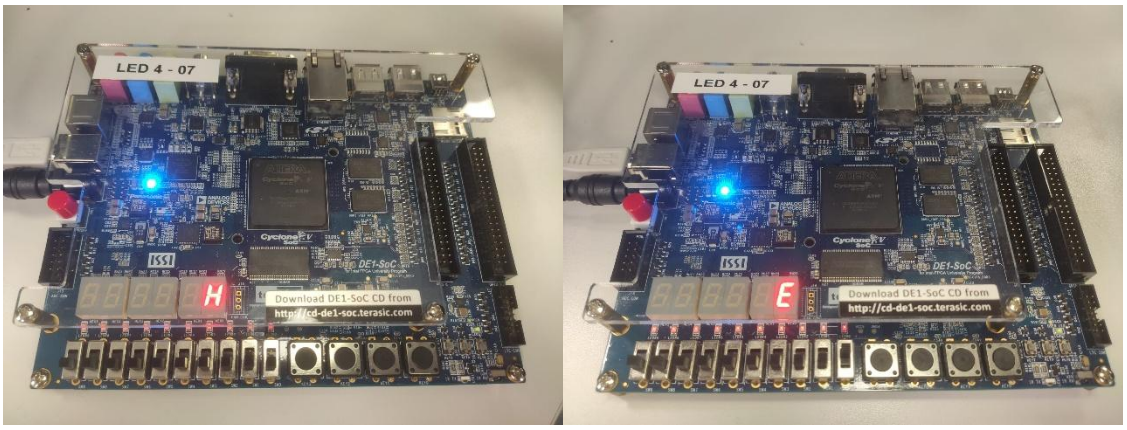


Figure 3 – Exercise 1 test on FPGA. Test combinations (SW_2 SW_1 SW_0): 000 (left side) and 001 (right side)

Timing simulation

The timing simulation is required for this section. Following the instructions in the appendix of the assignment, we first re-compiled the project in Quartus for a Cyclone IV FPGA, as the Cyclone V family is not supported. We selected “ModelSim-Altera” under the “Simulation” option in EDA tool settings, specifying VHDL as the output format. The synthesis generated two files, as described in the assignment, which were imported into the ModelSim project. All other VHDL files were removed from the project. After running the simulation, we observed results differing from the standard simulation.

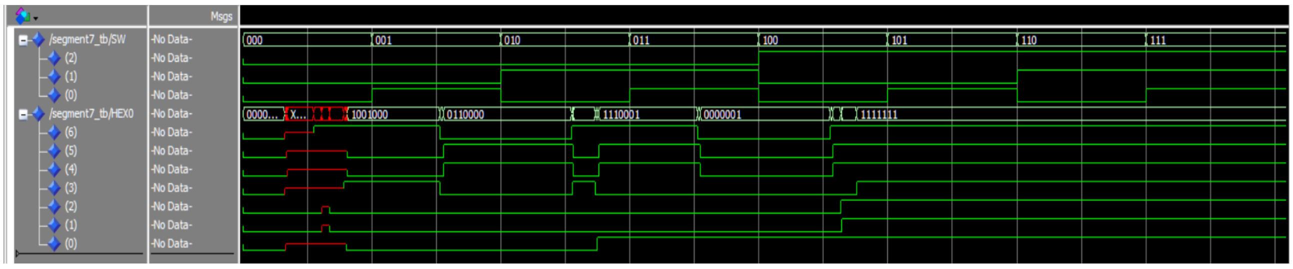


Figure 4 – Timing simulation results for Exercise 1

In the waveform viewer, at the moment of input/output value changes, we noticed artifacts lasting approximately 10 ns at the start of the simulation. Additionally, delays were observed between input value changes and the corresponding output value changes. The timing simulation is considering also the delays between the elementary logic gates that the synthesis tool has fitted onto the FPGA. These delays prevent signals from propagating instantaneously through the logic gates, resulting in a brief period where the output is incorrect. Once the signals propagate along the longest path for the delay (known as the critical path), the correct output values are achieved. This means the accurate output is provided at least 15 ns after the input values change.

In our circuit, the delays last approximately 15 ns, so the maximum operating frequency for the circuit is around 60 MHz. If the input switching frequency exceeds this threshold, the output may not be correctly evaluated due to these delays. However, our circuit is designed to be controlled by mechanical switches, which cannot operate at such high speeds, especially when handled by human fingers. Therefore, the propagation delays between the input and output are negligible for our use case, as they are too brief to be noticeable to the human eye. The considerations mentioned above are primarily relevant when high-speed digital circuits are interconnected, where propagation delays become significant compared to the operational frequency.

Section 2: Multiplexing the 7-segment display output

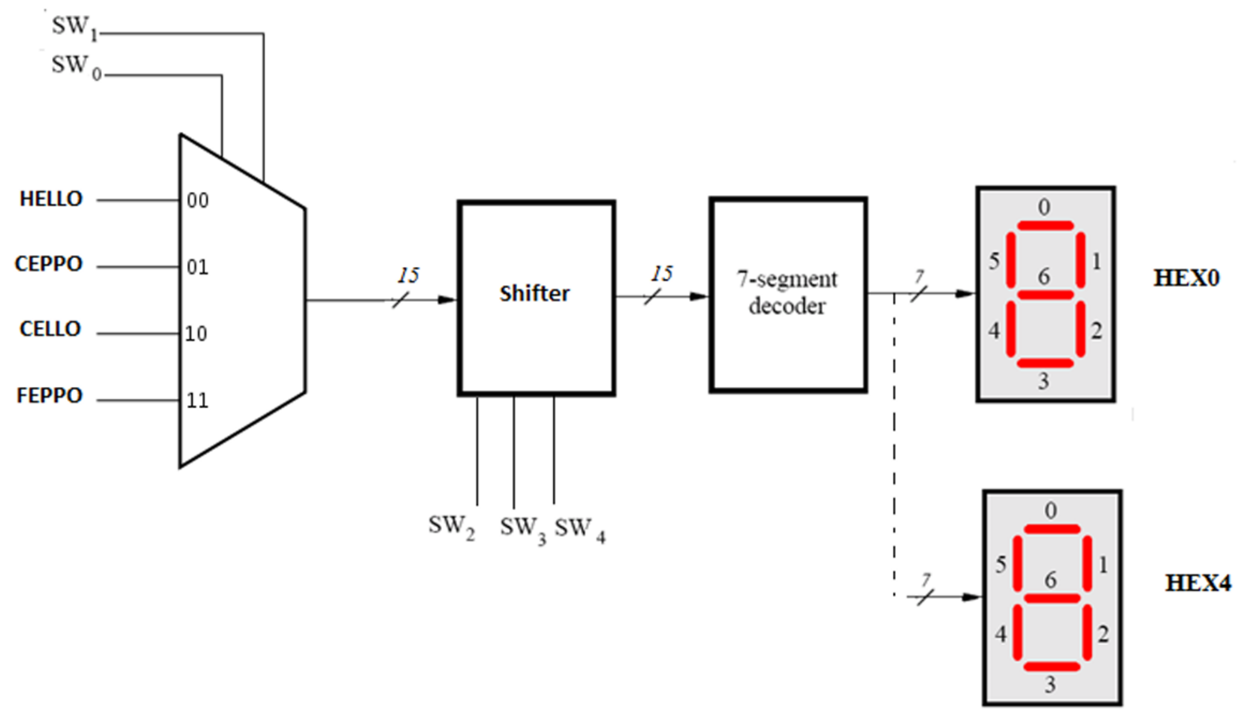


Figure 5 - Exercise 2 circuit structure

In this section, the objective is to design a circuit that shows a five-letter word on five 7 segment displays. The word is selected from a set of four words, determined by the first two switches. It can be displayed shifted by a selected number of positions, controlled by the switches 2 to 4. The structure of the circuit is shown in Figure 5. On the left side, a multiplexer selects the desired word to be sent to the shifter. The available word options and their corresponding selection codes are presented in Table 2. The word rotation pattern for the word “HELLO” is shown in Table 3.

The shifting pattern differs slightly from the one proposed in the assignment because we implemented VHDL code to perform a left-to-right rotation of the characters instead of the intended right-to-left rotation. We only realized this mistake while drafting this report, so we decided to adjust the shifting pattern rather than repeating all the project steps, including simulations, synthesis, and documentation. This adjustment has not impacted the overall functionality.

Word	Code (SW_1 SW_0)
HELLO	00
CEPPO	01
CELLO	10
FEPPO	11

Table 2 - Word selection

Code (SW4 SW3 SW2)	Word pattern (intended)	Word pattern (adjusted)
000	HELLO	HELLO
001	ELLOH	OHELL
010	LLOHE	LOHEL
011	LOHEL	LLOHE
100	OHELL	ELLOH

Table 3 - Character rotation pattern. The word pattern used in this project is in the right column (adjusted pattern)

Each character is represented by 3 bits, making the output of the multiplexer 15-bits wide (3 bits for each of the five characters). The output of the shifter is then connected to the 7-segment decoders. Each decoder requires 3 input bits to identify every letter; therefore, the first three bits of the shifter's output are sent to the first decoder, and similarly for the subsequent decoders. The character code for this section is the following one. It is important to notice that every digit is lit on while the corresponding bit is ‘0’.

Character	Code	Output pattern (leftmost bit drives the segment #0)
H	000	1001000
E	001	0110000
L	010	1110001
O	011	0000001
C	100	0110001
F	101	0111000
P	110	0011000
blank	111	1111111

Table 4 - Character code of decoder7

Delivered files

For this project, the reference subfolder is the one named *Ex2*. Five VHDL files have been delivered.

- *decoder7.vhd*: VHDL code for the 7-segment display decoder.
- *mux.vhd*: VHDL code for the multiplexer.
- *shifter.vhd*: VHDL code for the shifter.
- *part2.vhd*: VHDL code for the entire circuit (top-level entity).
- *part2_tb.vhd*: VHDL testbench.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *part2.sof*, and it already includes the assignation file *DE1-SoC.qsf*. For the timing simulation, two additional files have been delivered.

Design entry

The main entity has been named *part2*, and it is defined hierarchically. As discussed before, there are three fundamental components that are used in the entire circuit: a multiplexer, a shifter and a decoder. For every component, one different VHDL model has been created. The decoder model is the same used in the Section 1 but adapted to display other letters too.

The multiplexer model is the same as the one used in the previous laboratory experience, with the addition of some constants to enhance the comprehension of the code. These constants represent the different words and specifically store the five three-bit sequences (one sequence per character).

The shifter has been the most difficult part to be designed. It has one input port, which comes from the multiplexer, another input port which is connected to the switches 2 to 4, and one output port which will be connected to the decoders. We decided to represent the shifter component architecture using a process statement, as shown below.

```
process (SEL, INPUT)
begin
     if SEL = "000" then
          OUTPUT <= INPUT;
     elsif SEL = "001" then
          OUTPUT(14 downto 3) <= INPUT(11 downto 0);
          OUTPUT(2 downto 0)  <= INPUT(14 downto 12);
     elsif SEL = "010" then
          OUTPUT(14 downto 6) <= INPUT(8 downto 0);
          OUTPUT(5 downto 0)  <= INPUT(14 downto 9);
     elsif SEL = "011" then
          OUTPUT(14 downto 9) <= INPUT(5 downto 0);
          OUTPUT(8 downto 0)  <= INPUT(14 downto 6);
     elsif SEL = "100" then
          OUTPUT(14 downto 12) <= INPUT(2 downto 0);
          OUTPUT(11 downto 0)  <= INPUT(14 downto 3);
     else
          OUTPUT <= INPUT;
     end if;
end process;
```

There are many `if..else` statements which adjust the output vector based on the selection vector. Bit rotation to the left shifts all bits in a vector left by a specified number of positions, with the bits that are shifted out reintroduced at the end of the vector, preserving the total bit sequence. An example is shown in Figure 6: the word HELLO is shifted to the left by one position, according to Table 3.

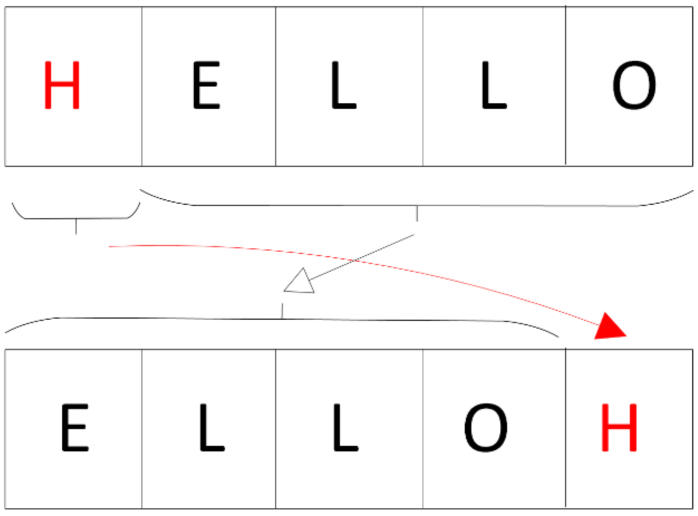


Figure 6 – Bit shifting process example

The last component to discuss is the *decoder*, which is very similar to the one of Section 1. The only difference is in the character that can be displayed. The entity is exactly the same, while the architecture has been edited, as shown.

```
-- DISPLAY code depends on the C code
DISPLAY <=      "1001000" when C = "000" else      -- Letter H
                "0110000" when C = "001" else      -- Letter E
                "1110001" when C = "010" else      -- Letter L
                "0000001" when C = "011" else      -- Letter O
                "0110001" when C = "100" else      -- Letter C
                "0111000" when C = "101" else      -- Letter F
                "0011000" when C = "110" else      -- Letter P
                "1111111";                         -- Blank
```

Five independent decoders are arranged to control five distinct displays. The output of the shifter is divided among the decoders: the leftmost three bits of the shifter's output vector are fed into the decoder driving the leftmost display (HEX4), and this pattern continues sequentially for the other decoders and displays.

Functional simulation

This waveform result requires a lot of effort to understand what is happening in the simulation, because of the complex data structures. The first point to remember is about the SW vector, which can be splitted in two. The first sub-vector, which contains the bits in position from 0 to 1, is the vector that selects the word from the set of four. This sub-vector is directly connected to the selection input of the multiplexer. The other sub -vector contains the bits in position from 2 to 4 and it is connected to the shifter component to select the shifting pattern.

The testbench retains its usual structure. Two signals have been defined within the architecture to provide better control over the simulation process.

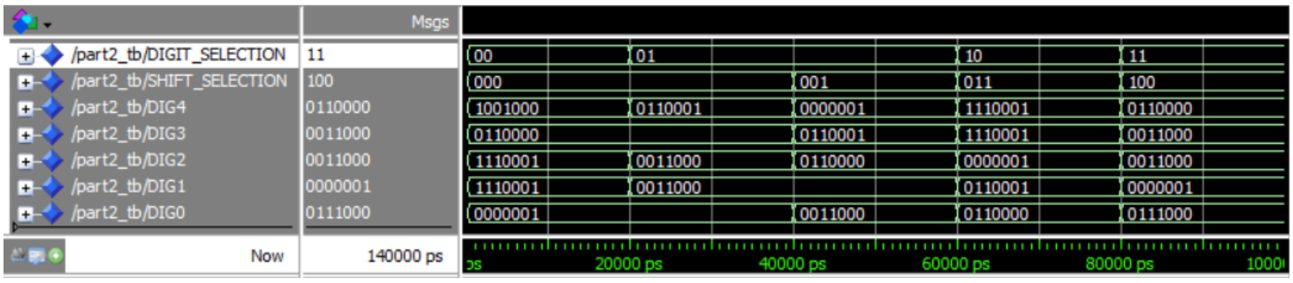
```
signal DIGIT_SELECTION: STD_LOGIC_VECTOR(1 downto 0);
signal SHIFT_SELECTION: STD_LOGIC_VECTOR(2 downto 0);
```

In the architecture, these signals are connected to the input/output ports of the unit under test using the port map statement. Within the process statement, their values are independently modified every 20 ns to explore various input combinations. The VHDL code for the process statement in the testbench is provided below.

```vhdl
process
begin
     DIGIT_SELECTION <= "00";      -- 'HELLO' selected
     SHIFT_SELECTION <= "000";     -- No shifting
     wait for 20 ns;
     DIGIT_SELECTION <= "01";      -- 'CEPPO' selected
     SHIFT_SELECTION <= "000";     -- No shifting
     wait for 20 ns;
     DIGIT_SELECTION <= "01";      -- 'CEPPO' selected
     SHIFT_SELECTION <= "001";     -- 'EPPOC' showed
     wait for 20 ns;
     DIGIT_SELECTION <= "10";      -- 'CELLO' selected
     SHIFT_SELECTION <= "011";     -- 'LLOCE' showed
     wait for 20 ns;
     DIGIT_SELECTION <= "11";      -- 'FEPPO' selected
     SHIFT_SELECTION <= "100";     -- 'EPPOF' showed
     wait;
end process;

DUT : part2
     port map(SW(1 downto 0) => DIGIT_SELECTION,
          SW(4 downto 2) => SHIFT_SELECTION,
          HEX0 => DIG0,
          HEX1 => DIG1,
          HEX2 => DIG2,
          HEX3 => DIG3,
          HEX4 => DIG4);
```

The simulation result is shown in the Figure 7.



```
                               Msgs
/part2_tb/DIGIT_SELECTION  11        00                 01                                     10                 11
/part2_tb/SHIFT_SELECTION  100       000                                  001                  011                100
/part2_tb/DIG4             0110000   1001000            0110001           0000001              1110001            0110000
/part2_tb/DIG3             0011000   0110000                              0110001              1110001            0011000
/part2_tb/DIG2             0011000   1110001            0011000           0110000              0000001            0011000
/part2_tb/DIG1             0000001   1110001            0011000                                0110001            0000001
/part2_tb/DIG0             0111000   0000001                              0011000              0110000            0111000
Now                        140000 ps  ps        20000 ps        40000 ps        60000 ps        80000 ps        1000
```

Figure 7 – Exercise 2 simulation results

To assist with interpreting the results, Tables 2, 3, and 4 are presented once more below.

Word	Code (SW_1 SW_0)
HELLO	00
CEPPO	01
CELLO	10
FEPPO	11

Code (SW_4 SW_3 SW_2)	Word pattern (adjusted)
000	HELLO
001	OHELL
010	LOHEL
011	LLOHE
100	ELLOH

Character	Output pattern (leftmost bit drives the segment #0)
H	1001000
E	0110000
L	1110001
O	0000001
C	0110001
F	0111000
P	0011000
blank	1111111

Synthesis

A new Quartus project has been created, including the VHDL files for `part2`, `decoder7`, `shifter`, and `mux`. The appropriate device was selected, and its pin assignments were imported for compilation. After completing this setup, the DE1-SoC board is ready to be programmed, allowing us to test its functionality. Given the numerous possible combinations this time, the testing process required more time. However, we can confidently conclude that the device operates as per the specifications.

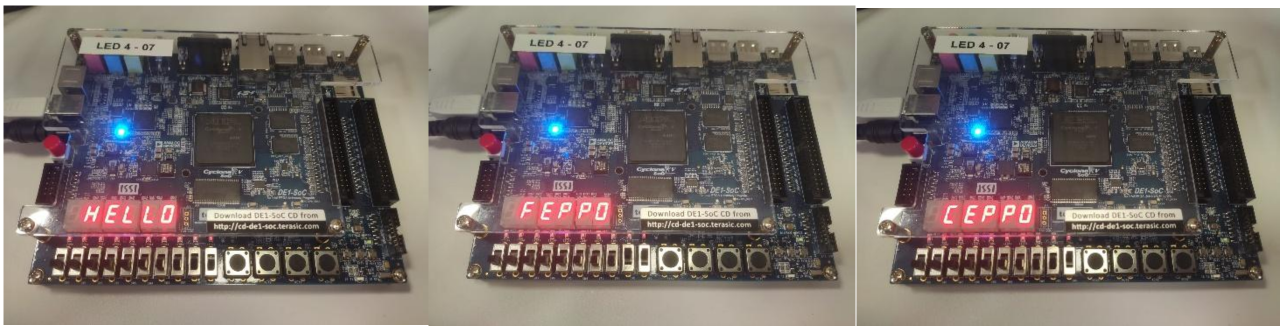


Figure 8 - Exercise 2, word selection test. From left to right, the combination of SW1 and SW0 for each case are 00, 11 and 01

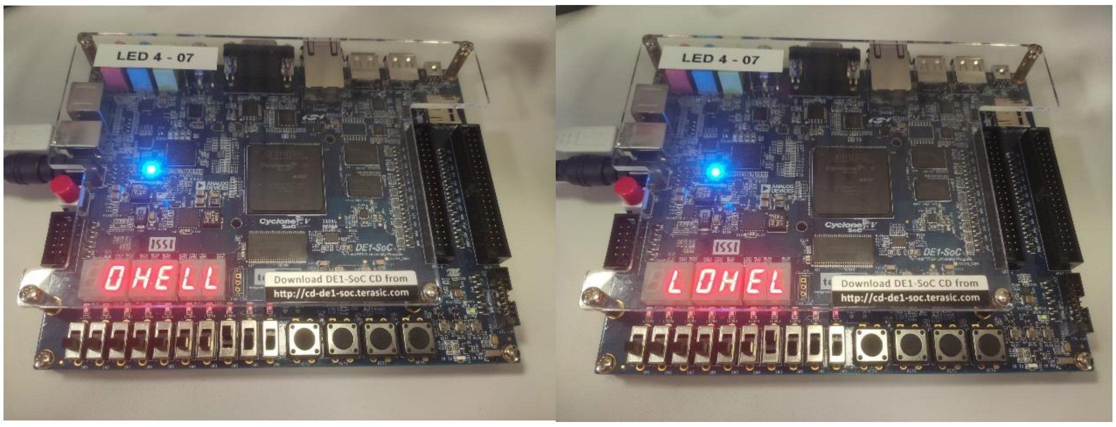


Figure 9 – Exercise 2, shifting pattern test. From left to right, the combination of SW4, SW3 and SW2 for each case are 001 and 010

Timing simulation

We were also asked to run the timing simulation for this section. Following the same steps as in the previous section, we obtained the waveforms shown below. The time interval has been reduced compared to the standard simulation to increase the time resolution.

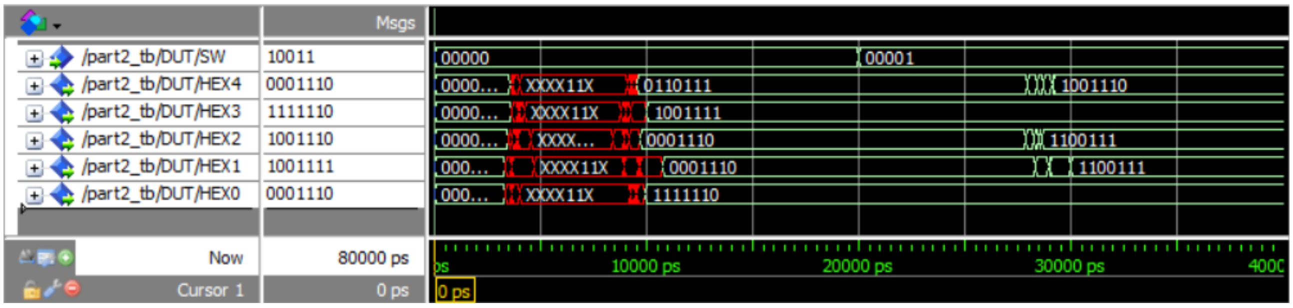


Figure 10 - Timing simulation results for Exercise 2

At the beginning of the simulation, artifacts appear in the output values, lasting around 10 ns. During this period, the output values are incorrect due to physical-level delays. The signals that are displayed in red are indicating undefined values. This undesired behaviour occurs every time the input combination changes, as each input variation induces corresponding artifacts in the output values due to these delays. As the signals propagate through the circuit, the output values stabilize and align with the correct values observed in a standard simulation.

Section 3: Binary to Decimal converter

Section 3 describes the design of a 4-bit binary-to-decimal converter, with the result displayed on 7-segment displays. The approach is straightforward: for binary numbers from 0000 (0 in decimal) to 1001 (9 in decimal), the binary input is sent to a decoder that drives the display to show the corresponding decimal digit. For binary numbers equal to or greater than 1010 (10 in decimal), the tens display is triggered to show the digit '1', while the units digit is determined by the combination of the three rightmost bits (the binary number excluding the MSB).

Binary value	Decimal digits	
0000	0	0
0001	0	1
0010	0	2
...	...	...
1001	0	9
1010	1	0
1011	1	1
1100	1	2
1101	1	3
1110	1	4
1111	1	5

Table 5 - Binary-to-Decimal conversion

The circuit is made of a decoder, a comparator, some multiplexers to select the signals to be sent to the decoders, and two additional circuits, named Circuit A and Circuit B. Circuit A generates the patterns for the unit digits of numbers greater than 9, which are selected by the multiplexers and sent to the unit decoder. Circuit B ensures that the digit '1' is displayed on the tens display when activated by the comparator. The structure of the circuit is shown in Figure 11.

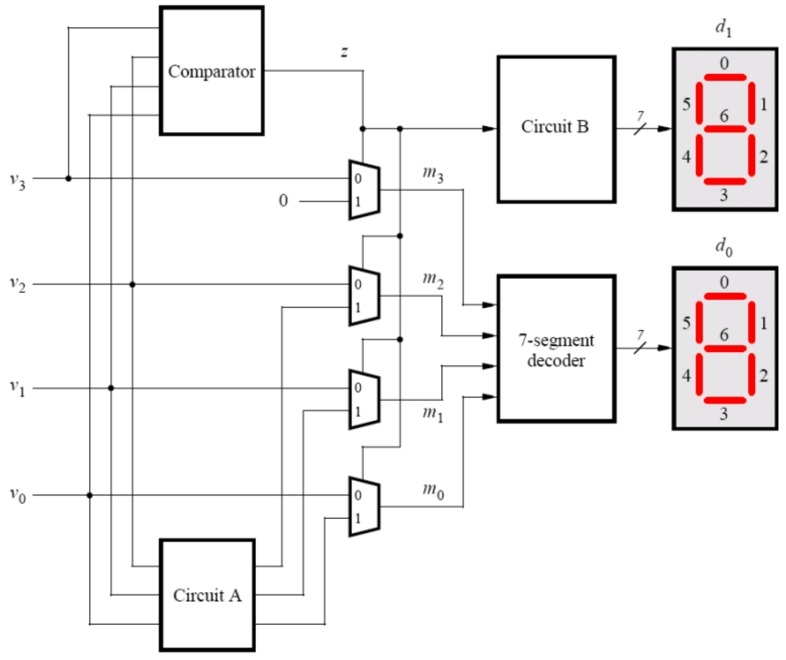


Figure 11 - Block scheme of the system that implements the binary-to-decimal conversion circuit

Delivered files

For this project, the reference subfolder is the one named *Ex2*. Seven VHDL files have been delivered.

- *circuitA.vhd*: VHDL code for the CircuitA block.
- *circuitB.vhd*: VHDL code for the CircuitB block.
- *comparator.vhd*: VHDL code for the comparator.
- *decoder7.vhd*: VHDL code for the decoder for the 7-segment display.
- *mux.vhd*: VHDL code for the multiplexer.
- *part3.vhd*: VHDL code for the entire circuit (top-level entity).
- *part3_tb.vhd*: VHDL testbench.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *part2.sof*, and it already includes the assignation file *DE1-SoC.qsf*.

Design entry

The hierarchy of the circuit is described above. The top-level entity has been named `part3` (*part3.vhd*) and has the switches vector as input port and the first two displays as output ports. The architecture is structural, based on low-level components.

The first component to be described is the 1-bit wide 2-to-1 multiplexer. This component has been described already multiple times in VHDL, and it is a simpler version of the one used in the previous section. Its architecture is defined in dataflow style by means of logic functions.

The second component is the `comparator` (*comparator.vhd*). This component takes a 4-bit wide vector as input, check if the decimal value is greater than nine, and if true, produces a high-level logic output. The architecture behavioural of this entity first performs the cast of the input vector to an unsigned integer value and then check the condition “greater than 9” by an `if…else` statement. If true, the output signal is driven high, else is driven low. The VHDL code of the architecture of the comparator is shown below.

```vhdl
architecture Behavior of comparator is
	signal NUM : INTEGER;
begin
	NUM <= to_integer(unsigned(INPUT));
	process (NUM)
	begin
		if NUM > 9 then
			OUTPUT <= '1';
		else
			OUTPUT <= '0';
		end if;
	end process;
end Behavior;
```

The use of two other libraries was necessary to perform operations with unsigned and numeric type signals.

The third component is the 7-segment decoder (*decoder7.vhd*). This component builds upon the ones from sections 1 and 2, but it has been adjusted to display digits instead of characters. The entity declaration remains unchanged. For simplicity, a cast from a standard logic vector to an unsigned integer has been included in this architecture. Based on the resulting unsigned integer value, the output code to drive the display is determined.

Circuit A has been the least immediate component to describe in VHDL. We spent some time trying to understand its functionality, which is reported in Table 6.

Binary input value	**Decimal value**	**Digit to be displayed (units)**	**Binary code generated**
1010	10	0	000
1011	11	1	001
1100	12	2	010
1101	13	3	011
1110	14	4	100
1111	15	5	101

Table 6 - Circuit A behaviour

Notice that the decision the binary code to be generated is independent of the MSB of the input vector. This circuit is basically a pattern generator based on the input combination (only the three rightmost bits). Its architecture description is the following one.

```
architecture Behavior of circuitA is
begin
     OUTPUT <= "000" when INPUT = "010" else
               "001" when INPUT = "011" else
               "010" when INPUT = "100" else
               "011" when INPUT = "101" else
               "100" when INPUT = "110" else
               "101" when INPUT = "111";
end Behavior;
```

The last component is called Circuit B. It essentially functions as a decoder for the 7-segment display, but its behaviour differs from that of the other decoder defined in the project. This decoder is designed to illuminate the display, showing the digit ‘1’ only when the input value exceeds 9. Circuit B has a single input, which is connected to the comparator's output, and a 7-bit-wide output to drive the display segments.

Functional simulation

For this circuit, the testbench (*part2_tb.vhd*) is going to test some combinations of input values which are in the range 0 to 8. These combinations will focus the test on the decoder and the comparator. Then, the input values are set to 9, 10 and 11 (1001 to 1011 in binary), which is the critical part of the circuit that must be tested with precision, as the transition from 9 to 10 triggers the activation of the second digit. For numbers greater than 10, the testbench will focus the test on *Circuit A* and *Circuit B*.

The VHDL file for the testbench is quite similar to the ones presented before. Three signals have been defined in the architecture for testing purposes. These signals are connected to the unit under test using a `port map` statement. In the process statement, there is the usual alternation between changing the input value and waiting for 20 ns.

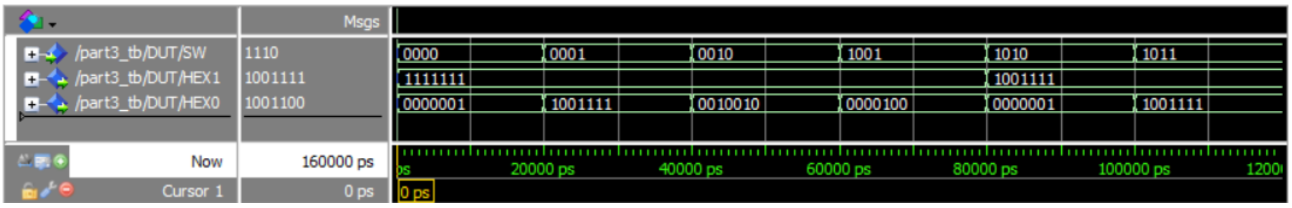


Figure 12 - Simulation results of Exercise 3

Synthesis

The synthesis process is identical to the previous ones. Once a new Quartus project has been created, every VHDL file has been included (apart from the testbench one). The assignment file for the DE1-SoC board has been imported, and the compilation generates the files to be uploaded on the board. The practical results of some input combinations are available in Figure 13.

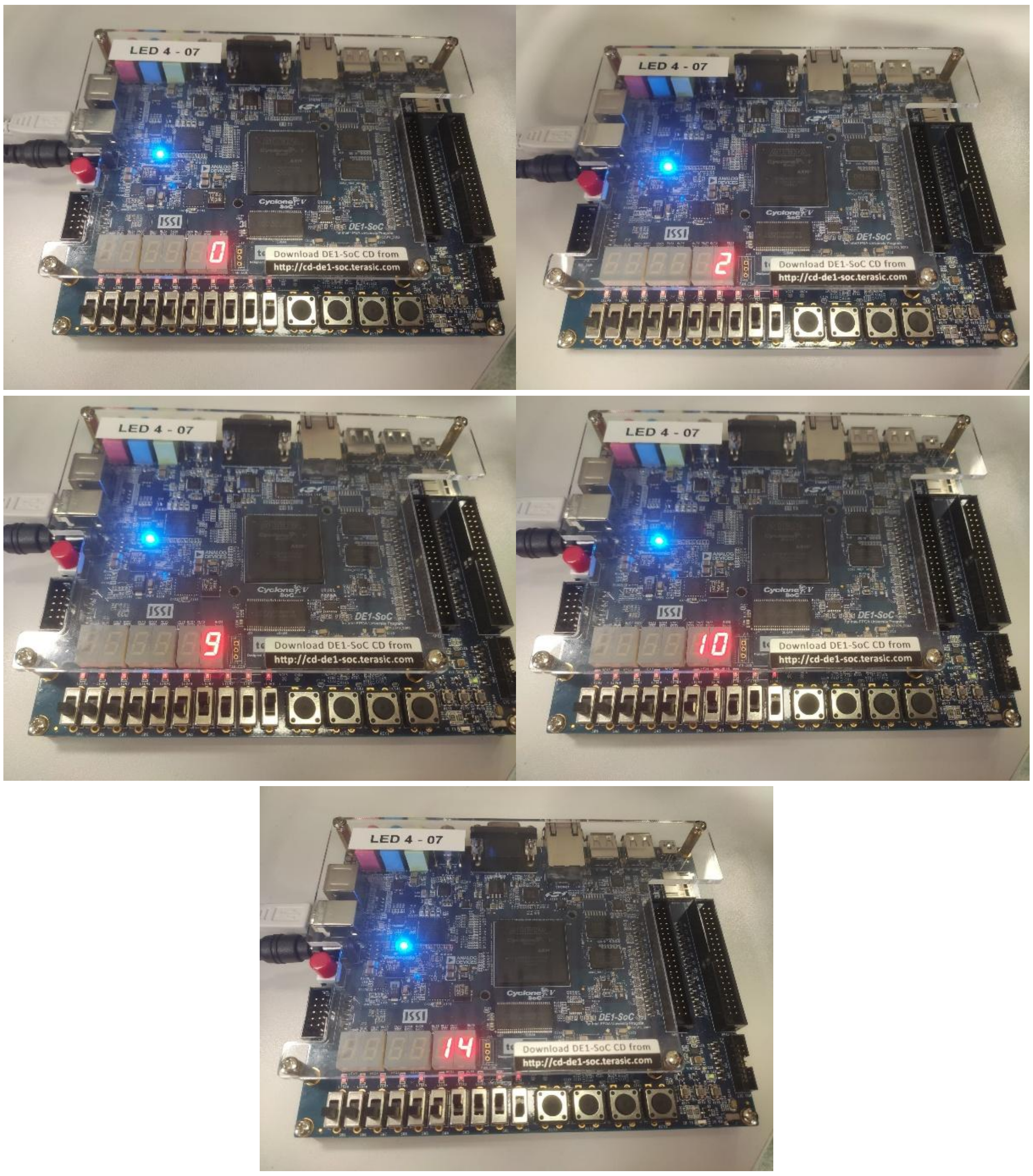


Figure 13 - Exercise 3, circuit test on FPGA. From left to right, and from the top to the bottom, the binary input is 0000, 0010, 1001, 1010, 1110

Section 4: Binary-to-BCD converter

The exercise requires designing a combinational circuit that converts a 6-bit binary number into a two-digit decimal number represented in BCD (Binary Coded Decimal) form. In other words, the circuit must take a binary number as input (using the 6 switches SW5–SW0) and convert it into two decimal digits, each encoded with 4 bits. These two decimal digits - one for the tens and one for the units - must then be displayed on two 7-segment displays (HEX1 for the tens and HEX0 for the units).

Design entry

This converter can be implemented using VHDL code through various approaches. Initial attempts were more software-oriented and not well-optimized for hardware implementation. For instance, an early version—though functional—was discarded due to inefficiency in hardware. This version employed a `for loop` and the summation of 2^n to convert the binary code into decimal without utilizing the ARITH library. It then separated the tens and units and ultimately used a decoder to display the result on a 7-segment display.

Later, it was determined that this approach was suboptimal for VHDL, particularly when targeting FPGA implementation. To design this converter, a structural approach was adopted for connecting components, while a behavioral approach was used to describe the process. The circuit enabling the implementation of this binary-to-BCD converter on an FPGA is outlined below.

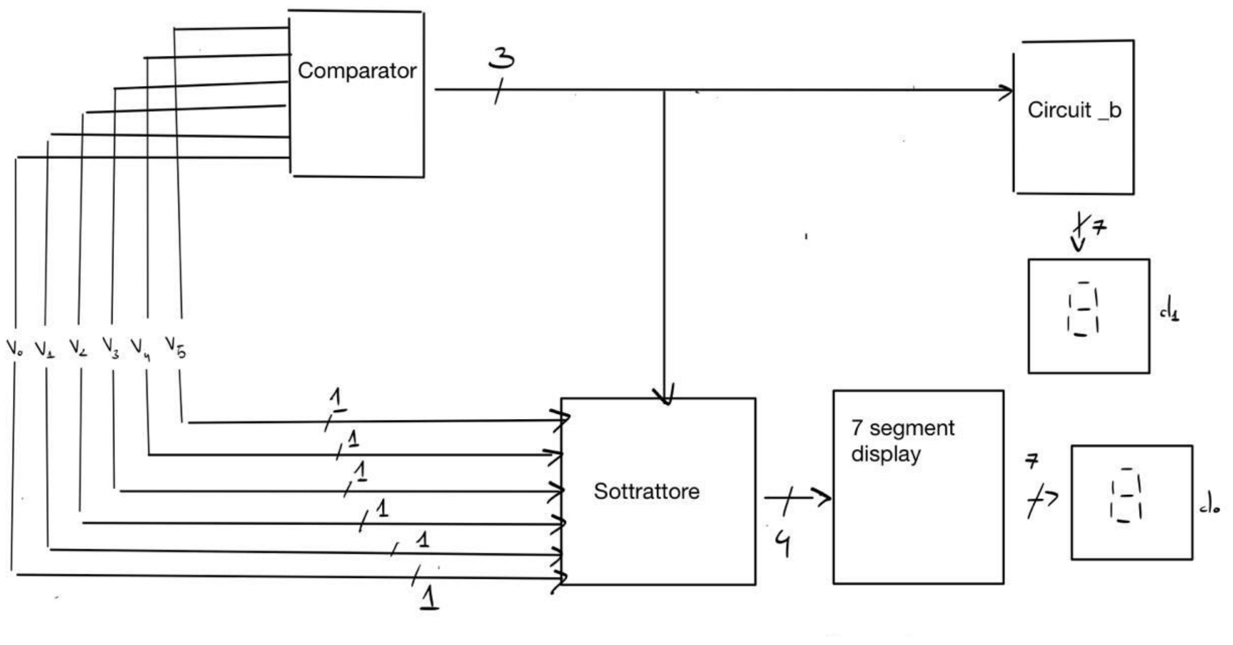


Figure 14 - Exercise 4 structure design

Overview

This design implements a 6-bits comparator that extracts the tens digit from a binary input and drives a 7-segment display accordingly. The system operates as follows:

- Takes a 6-bit input (`in_c`) and produces a 3-bit output (`out_c`), representing the tens digit of the decimal equivalent of `in_c`.
- Uses direct binary comparisons to determine the tens digit.
- Assumes the "<" operator on `std_logic_vector` behaves lexicographically, which aligns with numeric ordering when the MSB is the most significant bit.
- The valid input range is 0 to 60; for invalid values, the display turns off.

Comparator Logic

The comparator maps the 6-bit binary input (`in_c`) to its corresponding tens digit by evaluating against predefined binary thresholds. A subtractor component ("sottrattore") is used to isolate the units digit, ensuring correct display representation.

To ensure proper operation, a "conversion to integer" has been implemented. The `std_logic_vector` input is converted to an integer before arithmetic operations, avoiding incorrect lexicographical comparisons.

If the subtraction operation (`in_sott_1`) does not match expected values, a fallback mechanism subtracts 10.

The circuit will provide output "1111" for values outside 0-9, signalling an undefined/error state.

7-Segment Display Driver (`circuit_b`)

The seven-segment display driver is designed to convert a 3-bit input into a 7-bit active-low output for segment control. The 3-bit input vector corresponds to a digit (ranging from 0 to 6) as provided by the comparator output. The 7-bit output vector labelled "EOX_1", controls the individual segments of the display, with bit 6 representing the most significant bit (MSB). The driver operates using active-low logic, where a segment is turned ON when the corresponding bit is set to '0' and turned OFF when the bit is set to '1'. For any unrecognized input, the driver defaults to turning all segments off, effectively serving as an error indicator.

7-Segment Display Conversion

The 4-bit to 7-bit conversion is designed to generate the appropriate outputs for driving the seven-segment display. The input is a 4-bit binary value, assembled from separate bits, while the output is a 7-bit signal that controls the seven-segment display with active-low logic. This structured approach ensures efficient decoding and display functionality. Additionally, robust error-handling mechanisms are implemented to ensure proper behavior in undefined states.

Delivered files

For this project, the reference subfolder is the one named *Ex2*. Seven VHDL files have been delivered.

- *BTD.vhd*: main VHDL code for the entire circuit (top-level entity).

```
-- Subtractor (sottrattore): combines the tens
-- digit and the units to produce the final output
-- for the units display.

component sottrattore is
    port (
        in_sott_1 : in  std_logic_vector(2 downto 0); -- tens
digit input (LSB is bit 0)
        v_in      : in  std_logic_vector(5 downto 0); -- 6-bit
input (LSB is bit 0)
        out_dis   : out std_logic_vector(3 downto 0)  -- 4-bit
output for units display
    );
end component;

-- Circuit_B: drives the tens display. Active low with bit 6 as
MSB.
component circuit_b is
    port (
        in_b  : in std_logic_vector(2 downto 0);
        EOX_1 : out std_logic_vector(6 downto 0);               -
- 7-segment display output for tens digit
    );
end component;

-- 7-Segment Display: converts a 4-bit input to the display
segments.
component segment_display is
    port(
        v_0, v_1, v_2, v_3 : in  std_logic;
        EOX                : out std_logic_vector(6 downto 0)
    );
end component;
```

- *1_comparator.vhd*: VHDL code for the comparator block.

```
process(in_c)
begin
    -- Direct comparisons: it is assumed that the "<" operator
on std_logic_vector (with order 5 downto 0)
   -- works in a "lexicographic" way consistent with the
   numerical order, in fact there is a correspondence between
MSBs.

    if in_c < "001010" then  -- in_c < 10 decimal the comparison
considers 00 in MSB position                               out_c
<= "000";      -- tens 0
    elsif in_c < "010100" then  -- 10 <= in_c < 20
        out_c <= "001";       -- lo 0 is the bit in position 2
    elsif in_c < "011110" then  -- 20 <= in_c < 30
        out_c <= "010";       -- tens 2
    elsif in_c < "101000" then  -- 30 <= in_c < 40
        out_c <= "011";       -- tens 3 is the bit position 2
elsif in_c < "110010" then  -- 40 <= in_c < 50
        out_c <= "100";       -- tens 4
    elsif in_c < "111100" then  -- 50 <= in_c < 60
        out_c <= "101";       -- tens 5
    elsif in_c <= "111111" then  -- in_c <= 60
        out_c <= "110";       -- tens 6
    else
          out_c <= "111";      --error
    end if;
end process;
end behavioural;
```

- *3_7_segment_decoder.vhd*: VHDL code for the decoder of the 7 segment.

```
process (v_0, v_1, v_2, v_3) -- any change in the inputs
activates the process
   variable bit_series : STD_LOGIC_VECTOR(3 downto 0);
   begin
         -- concatenate the bits to form the binary number (v_3 is
the MSB)
bit_series := v_3 & v_2 & v_1 & v_0;
    if (bit_series = "0000") then
           EOX <= "0000001";   -- 0
     elsif (bit_series = "0001") then
           EOX <= "1001111";   -- 1
     elsif (bit_series = "0010") then
           EOX <= "0010010";   -- 2
     elsif (bit_series = "0011") then
           EOX <= "0000110";   -- 3
     elsif (bit_series = "0100") then
           EOX <= "1001100";   -- 4
     elsif (bit_series = "0101") then
           EOX <= "0100100";   -- 5
     elsif (bit_series = "0110") then
           EOX <= "0100000";   -- 6
     elsif (bit_series = "0111") then
           EOX <= "0001111";   -- 7
     elsif (bit_series = "1000") then
           EOX <= "0000000";   -- 8
     elsif (bit_series = "1001") then
           EOX <= "0001100";   -- 9
     else
           EOX <= "1111111";   -- for invalid values: all segments
off
     end if;
end process;
```

- *4_circuit_B.vhd*: VHDL code to implement the circuit B.
  The code used for the circuit b is approximately the same as the one used for *3_7_segment_decoder*

- *6_sottrattore.vhd*: VHDL code to implement the "Sottrattore" block.

```
process(v_in, in_sott_1)
variable sottrazione, minuendo, sottraendo : integer;
begin                                                          --
Convert the input vector to an integer value.
minuendo := to_integer(unsigned(v_in));
   -- Default assignment: no subtraction.
    sottrazione := minuendo;

    -- Determine the subtraction amount based on the tens digit
(in_sott_1)
    if in_sott_1 = "000" then
 -- 0 range: No subtraction
        null;
    elsif in_sott_1 = "001" then        -- 10-20 range: subtract
10
        sottraendo := 10;
        sottrazione := minuendo - sottraendo;
    elsif in_sott_1 = "010" then        -- 20-30 range: subtract
20
        sottraendo := 20;
        sottrazione := minuendo - sottraendo;
    elsif in_sott_1 = "011" then        -- 30-40 range: subtract
30
        sottraendo := 30;
        sottrazione := minuendo - sottraendo;
    elsif in_sott_1 = "100" then        -- 40-50 range: subtract
40
        sottraendo := 40;
        sottrazione := minuendo - sottraendo;
    elsif in_sott_1 = "101" then        -- 50 range: subtract 50
        sottraendo := 50;
        sottrazione := minuendo - sottraendo;
    elsif in_sott_1 = "110" then  -- Special case: no
subtraction for this branch
        sottraendo := 60;
            sottrazione := minuendo - sottraendo;
    else
        sottrazione := 10;  -- Error fallback value
    end if;
end process;
```

The rest of the code implements a part where the units is associated with 4 bits there will be send to the *7_segment_decoder*

- *TESTBENCH_4.vhd*: VHDL testbench.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *BTD.sof*, and it already includes the assignation file *DE1-SoC.qsf*.

Functional simulation

The simulation applies a series of stimuli to verify the correct conversion of a binary input into two decimal digits, which are displayed on the HEX1 and HEX0 signals. Each change in the SW signal is followed by a 20 ns delay, enabling observation of the circuit's response to each value sequentially.

The process block includes a sequence of assignments to the SW signal, each accompanied by a pause (`wait for 20 ns;`) after each change. This approach facilitates the analysis of the converter's behaviour for various input values, ensuring accurate functionality. The testbench's process statement is shown below.

```
process
begin
    -- range 0-9: 0, 4, 9
    SW <= "000000";  -- 0
    wait for 20 ns;
    SW <= "000100";  -- 4
    wait for 20 ns;
    SW <= "001001";  -- 9
    wait for 20 ns;

    -- range 10-19: 19
    SW <= "010011";  -- 19
    wait for 20 ns;

    -- range 20-29: 20, 29
    SW <= "010100";  -- 20
    wait for 20 ns;
    SW <= "011101";  -- 29
    wait for 20 ns;

    -- range 30-39: 30
    SW <= "011110";  -- 30
    wait for 20 ns;

    -- range 40-49: 44,

    SW <= "101100";  -- 44
    wait for 20 ns;
```

```
        -- range 50-59: 50, 59
        SW <= "110010";  -- 50
        wait for 20 ns;

        SW <= "111011";  -- 59
        wait for 20 ns;

        -- range 60-63: 60, 61, 63
        SW <= "111100";  -- 60
        wait for 20 ns;
        SW <= "111101";  -- 61
        wait for 20 ns;

        wait;  -- end test
    end process;
```

Range	Tested Values (Decimal)	Tested Values (Binary)
0 - 9	0, 4, 9	"000000", "000100", "001001"
10 - 19	19	"010011"
20 - 29	20, 29	"010100", "011101"
30 - 39	30	"011110"
40 - 49	44	"101100"
50 - 59	50, 59	"110010", "111011"
60 - 63	60, 61, 63	"111100", "111101", "111111"

Table 7 - Exercise 4 tested combinations

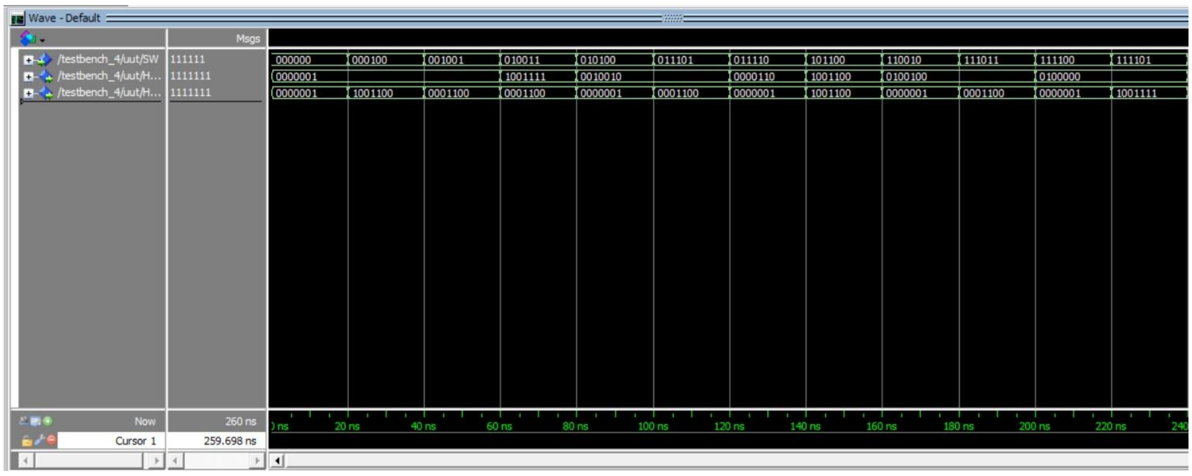


Figure 15 – Exercise 4 simulation results

Synthesis

Following the usual steps, the project has ben compiled and synthesized in Quartus to be uploaded to the FPGA. We confirm that the circuit works as intended. The output coming from some combinations that were tested in lab are available in the figures below.

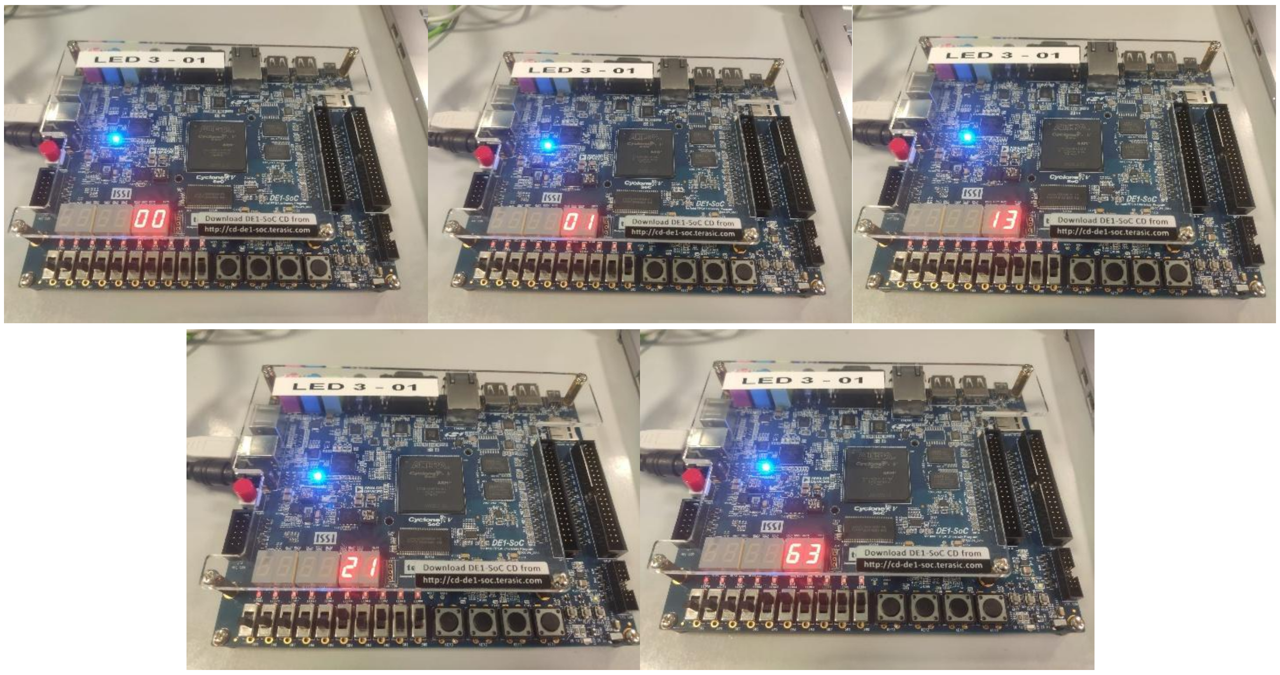


Figure 16 - Exercise 4, circuit test on FPGA. From left to right, and from the top to the bottom, the binary input is 000000, 000001, 001101, 010101, 111111

Conclusions

This laboratory experience has been particularly engaging due to the practical implementation of the circuit we designed. Additionally, we gained a deeper understanding of the hierarchical model in VHDL, which is useful for handling complex circuits that can be naturally divided into simpler building blocks. Throughout the experiment, no significant difficulties arose in coding, simulation, synthesis, or FPGA testing, except in the final exercise.

The circuit design in Section 4 initially followed a microcontroller-style algorithm. It performed binary-to-decimal conversion by dividing the decimal number by ten to determine the tens digit and using the MOD operation to extract the ones digit. While functional, this approach proved suboptimal for VHDL and hardware implementation. A considerable amount of time was spent adapting the microcontroller-style algorithm into a hardware-appropriate model.

These difficulties were confined to the modelling stage of the elementary building blocks that compose the circuit. However, the subsequent coding, simulation, and testing on the FPGA progressed smoothly without any issues.

Sources and notes

For all the VHDL files provided, an online tool has been used to format the code to ensure coherence between all the different files. The tool is free to use, and it is available at this link: VHDL Beautifier.

The VHDL files were developed using course materials produced by Prof. G. Masera (available on the course webpage) and the following eBooks recommended by the professor:

1. Mealy, Bryan, and Fabrizio Tappero. *Free Range VHDL*. 2016.
2. Ashenden, Peter J. *The VHDL Cookbook*. 1990.