Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 5. Code Converters

This chapter introduces a category of circuits known as "code converters". This type of circuit includes the encoders and the decoders. Encoders are circuits used in the conversion of information in one format to another. Decoders are complementary circuits, used to undo a conversion performed by an encoder. The concepts of code converter circuits are investigated in this chapter through their implementation in VHDL.

In previous chapters, a single LED has been employed in the circuits as their outputs. An LED was a feasible output interface, as so far the case study circuits in this book have been conceived in order to perform single bit functions. In this chapter a more real world case study is developed. It is a basic calculator that performs 8 bits operations, and presents the results in 7-segment displays. A decoder circuit is used to convert the binary result provided by the calculator, into a hexadecimal value to be shown in the 7-segment displays. In order to perform 8 bits operations, the single bit signals are arranged in VHDL as an array of signals, also known as a "bus". For a better understanding of the proposed case study, in this chapter there is also a discussion on "array of signals in VHDL" and "7-segment displays decoders". At the end of the chapter, the students should be able:

- to understand the concepts of encoder and decoder;
- to implement a decoder in VHDL;
- to design, to simulate and to prototype the proposed case study using an FPGA board.

5.1 Arrays of Signals

As shown in Figure 4.8, A, B, C and F are all 1 bit signals. In Figure 5.1 the same signals are represented as arrays of 4 bits signals. This means that A, B, C and F can now hold values in the range from 0 to 15 in decimal or, in binary, from 0000 to 1111.

Considering in Figure 4.8 and in Figure 5.1 that:

- C1 performs the function F1 = A or B or C
- C2 performs the function F2 = B xor C
- In Figure 4.8 (1 bit operands), assume A = 0, B = 1, C = 0
- In Figure 5.1 (4 bits operands), assume A = 0101, B = 1000, C = 0001

Figure 5.2 shows the results obtained considering 1 bit and 4 bits wide operands. For 1 bit operands, F1 and F2 result in 1. For the 4 bits version, F1 results in 1101 (13 in decimal) and F2 results in 1001 (9 in decimal).

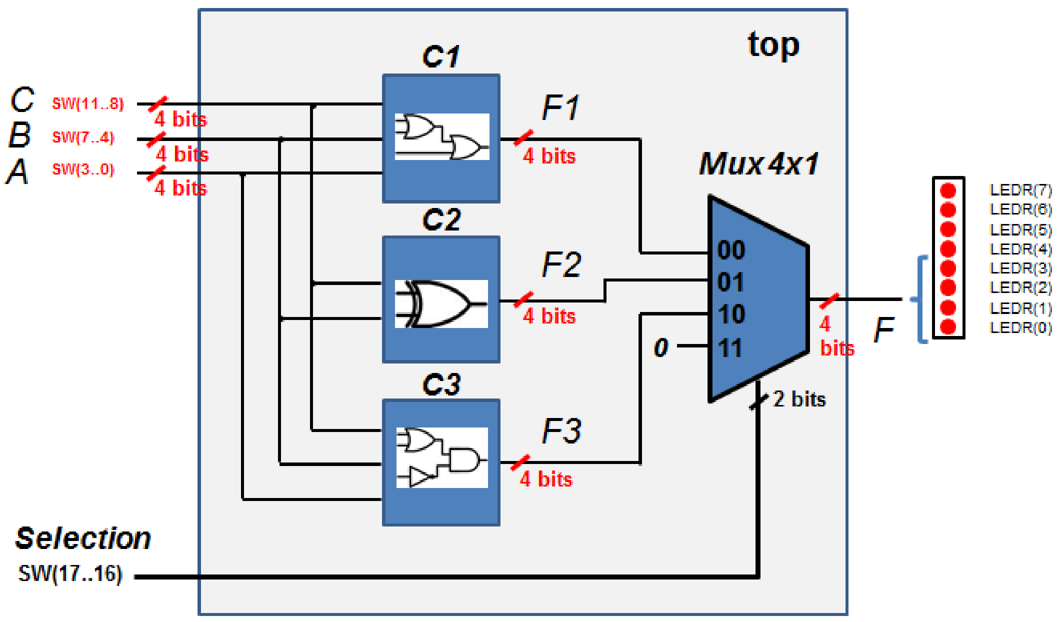


Figure 5.1. Block diagram of a 4 bits circuit.

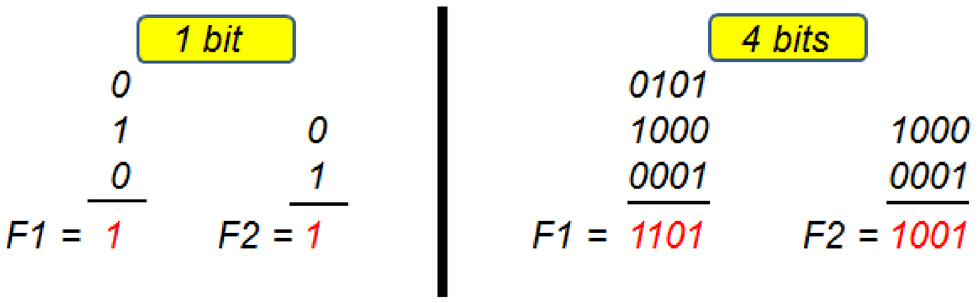


Figure 5.2. C1 and C2 components using 1 bit and 4 bits operators.

In VHDL the array of signals used in Figure 5.1 can be defined through the *std_logic_vector* declaration:

- ***std_logic*** is used to declare a single signal, a single bit, a wire.
- ***std_logic_vector*** is used to declare an array of signals, a set of bits, a bus.

Considering a multiple bit implementation (array of signals), the components shown in Figure 5.1 have been rewritten, and they are listed next. It is important to notice in Figure 5.3, Figure 5.4, and Figure 5.5, that only the entity has changed. The architecture, where the functions are performed, has not been changed.

4 bits

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C1 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      C: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C1;

architecture c1_str of C1 is
begin
  F <= A or B or C;
end c1_str;
```

4 bits

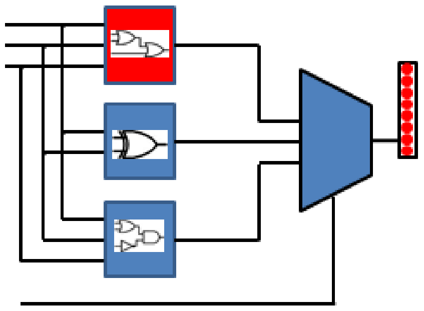


Figure 5.3. VHDL implementation for component C1 (4 bits).

4 bits

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C2 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C2;

architecture c2_str of C2 is
begin
  F <= A xor B;
end c2_str;
```

4 bits

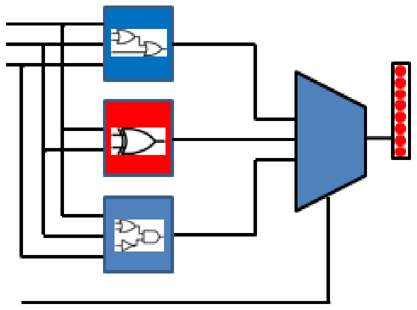


Figure 5.4. VHDL implementation for component C2 (4 bits).

4 bits

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C3 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      C: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C3;

architecture c3_str of C3 is
begin
  -- proposed lab exercise
end c3_str;
```

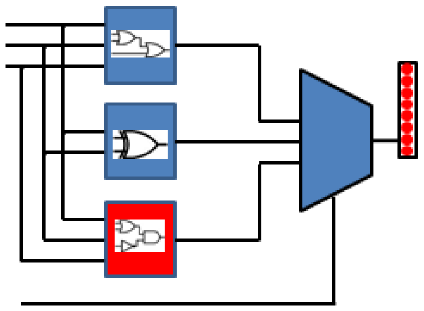


Figure 5.5. VHDL implementation for component C3 (4 bits).

4 bits

2 bits

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity Mux is
port (w: in std_logic_vector(3 downto 0);
      x: in std_logic_vector(3 downto 0);
      y: in std_logic_vector(3 downto 0);
      z: in std_logic_vector(3 downto 0);
      s: in std_logic_vector(1 downto 0);
      m: out std_logic_vector(3 downto 0)
     );
end Mux;

architecture mux_bhv of Mux is
begin
  -- see previous lab.
end mux_bhv;
```

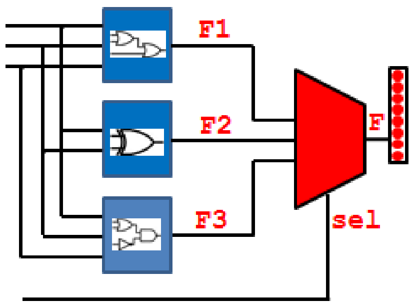
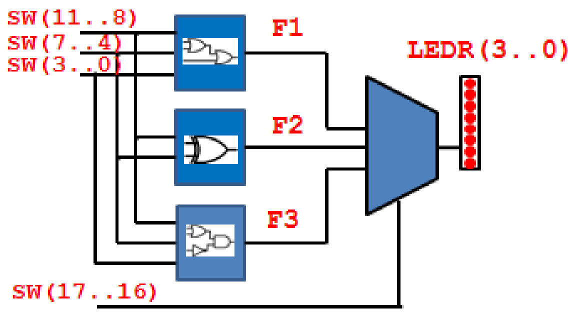


Figure 5.6. VHDL implementation for component Mux (4 bits inputs/outputs).

In Figure 5.7, the A, B and C inputs are now 4 bits signals represented, respectively, by $SW_{3..0}$, $SW_{7..4}$, and $SW_{11..8}$. The output is now also a 4 bits signals represented by $LEDR_{3..0}$.

```
library  ieee;
use  ieee.std_logic_1164.all;
entity top is
 port ( SW : IN STD_LOGIC_VECTOR(17  downto 0);
        LEDR : OUT  STD_LOGIC_VECTOR(17  downto 0)
 );
end top;
architecture top_stru of top is
  signal F1, F2, F3: std_logic_vector(3 downto 0);
  component C1
    port (A : in std_logic_vector(3  downto 0);
          B : in std_logic_vector(3  downto 0);
          C : in std_logic_vector(3  downto 0);
          F : out std_logic_vector(3  downto 0));
  end component;
  component C2
    port (A : in std_logic_vector(3  downto 0);
          B : in std_logic_vector(3  downto 0);
          F : out std_logic_vector(3  downto 0));
  end component;
  component C3
    port (A : in std_logic_vector(3  downto 0);
          B : in std_logic_vector(3  downto 0);
          C : in std_logic_vector(3  downto 0);
          F : out std_logic_vector(3  downto 0)
    );
  end component;
  -- Mux component declaration goes here
```



```
begin

 L0: C1 port map (SW(3 downto 0),
        SW(7 downto 4), SW(11 downto 8), F1);

 L1: C2 port map (SW(3 downto 0),
        SW(7 downto 4), SW(11 downto 8), F2);

 L2: C3 port map (SW(3 downto 0),
        SW(7 downto 4), SW(11 downto 8), F3);

 L3: Mux port map (F1, F2, F3,
        SW(17 downto 16), LEDR(3 downto 0));

end top_stru;   -- END architecture
```

Figure 5.7. VHDL implementation for the top circuit of Figure 5.1.

5.2 Seven Segment Displays

In digital circuits, the 7-segment display is a widely used component. This component has seven LEDs that can be turned on and off independently. As shown in Figure 5.8, there are two types of 7-segment displays:

- **Common cathode** – the LEDs cathode terminals are connected together to the ground and each anode terminal is connected individually to a Vcc source.
- **Common anode** - the LEDs anode terminals are connected together to Vcc, and each cathode terminal is connected individually to ground.

In Figure 5.1, the 4 bits result is shown in four LEDs (4 bits). A 7-segment display could have been used to show the result in another format as, for instance, in decimal or hexadecimal. In this case, a decoder circuit would be necessary in order to convert the binary output into the appropriate format and coding expected by the 7-segment display.

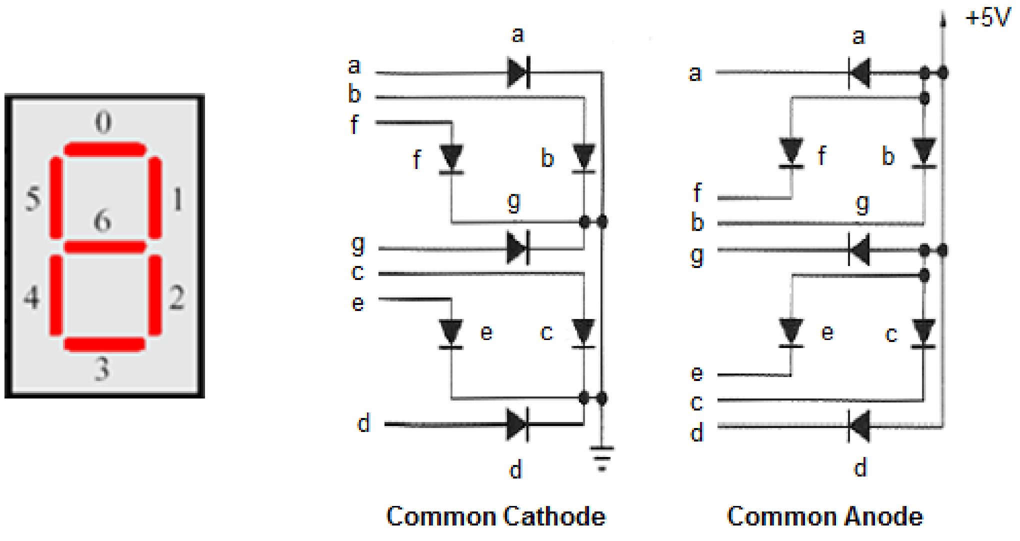
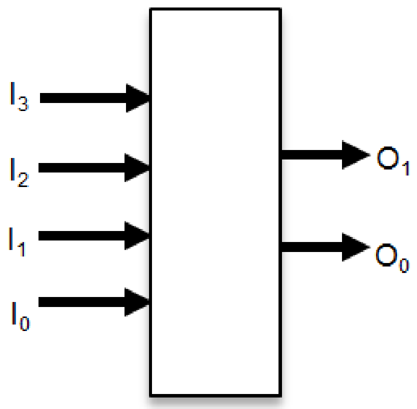


Figure 5.8. Seven segment display.

5.3 Encoders and Decoders

As stated before, encoders are used to convert a piece of information from one format to another, and decoders are used to undo the conversion. Figure 5.9 shows an example of encoder, used to convert the 4-bits input value I to a 2-bits coded output O. In this example, the input shown in Figure 5.9(a) presents just one bit set at a time, which is also known as the "one-hot" code. When comparing to the decoder, an encoder component can be used to reduce the amount of bits required to represent any piece of information.

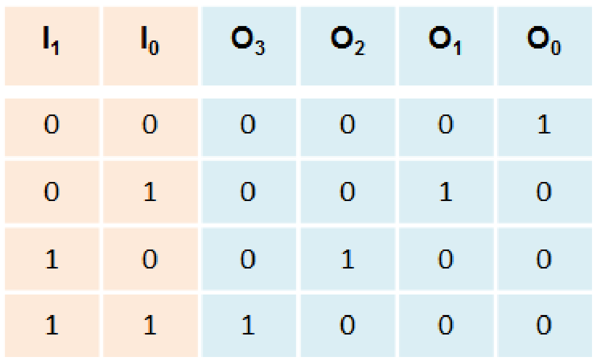
I_3	I_2	I_1	I_0	O_1	O_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



(a) (b)

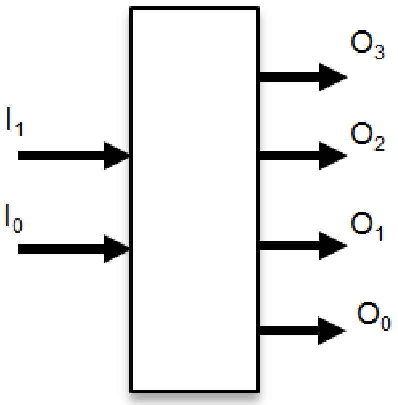
Figure 5.9. Encoder. (a) truth table. (b) symbol.

The decoder shown in Figure 5.10 was designed to convert back the code provided by the encoder shown in Figure 5.9, that is, it receives a 2-bits code on its input, and provides an one-hot output. This decoder activates only one O_i output at a time, according to the combination of the I input values. In this sense, there is a direct relationship between the number of outputs (N_o) and the number of inputs (N_i), i.e., $N_o = 2 \times N_i$.



I_1	I_0	O_3	O_2	O_1	O_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(a)



(b)

Figure 5.10. Decoder. (a) truth table. (b) symbol.

5.4 Designing a Seven Segment Decoder

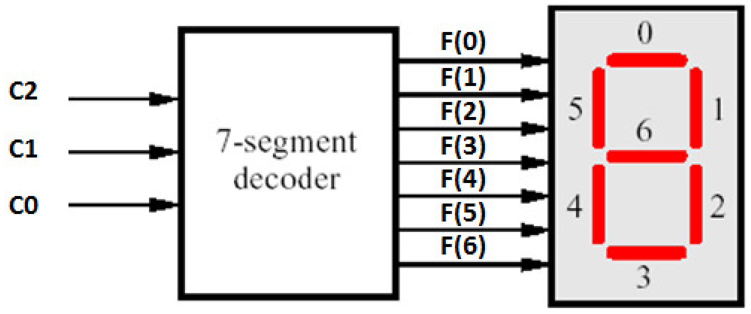
As shown in Figure 5.8, algorisms can be exhibited in a 7-segment display by turned on and off the different segments. For instance, in Figure 5.8 when providing the value “0000000” in binary to the seven inputs (a, b, …, g), all LEDs are lighted on, and the ‘8’ value will show on the display. In this case, in order to display the number ‘8’, the code “1000” (eight in binary) should be decoded into “0000000”. A 7-segment decoder is a very useful component, as it can be used in digital instruments that have a numerical output, for instance, multimeters, frequency-meters, etc.

The DE2 board has seven 7-segment displays, all of them common anode (LEDs are turned on with a logic zero in the inputs). To show a binary value in one of DE2’s displays, a conversion from binary code to 7-segment code should be performed.

As an example of a decoder design, consider a circuit whose functionality is to display the letters ‘U’, ‘F’, ‘S’, and ‘C’ in a 7-segment display. The truth table in Figure 5.11(a) is conceived considering 5 codes: “000” for letter ‘U’; “001” for letter ‘F’; “010” for letter ‘S’; “011” for letter ‘C’; and “111” to clear the display (all LEDs are off). The remaining 3 possible codes to represent in 3 bits (“100”, “101”, and “110”) are not used in this circuit. The 7-segment decoder has to be designed in order to convert the 3-bit input codes into 7-bit output codes shown in the 4th column of Figure 5.11(a), which are used to turn on and off the LEDs of the display. For instance, to show the letter ‘U’, LEDs 0 and 6 (a and g in Figure 5.8) should be turned off. So, in the 4th column of Figure 5.11(a), there is a ‘1’ on this positions (common anode).

C2	C1	C0	7 bits *6543210*	Display
0	0	0	1000001	U
0	0	1	0001110	F
0	1	0	0010010	S
0	1	1	1000110	C
1	1	1	1111111	

(a)



(b)

Figure 5.11. A 7-segment decoder. (a) truth table. (b) block diagram.

Considering the truth table in Figure 5.11(a), a component to implement this 7-segment decoder could be designed using different methods as, for instance:

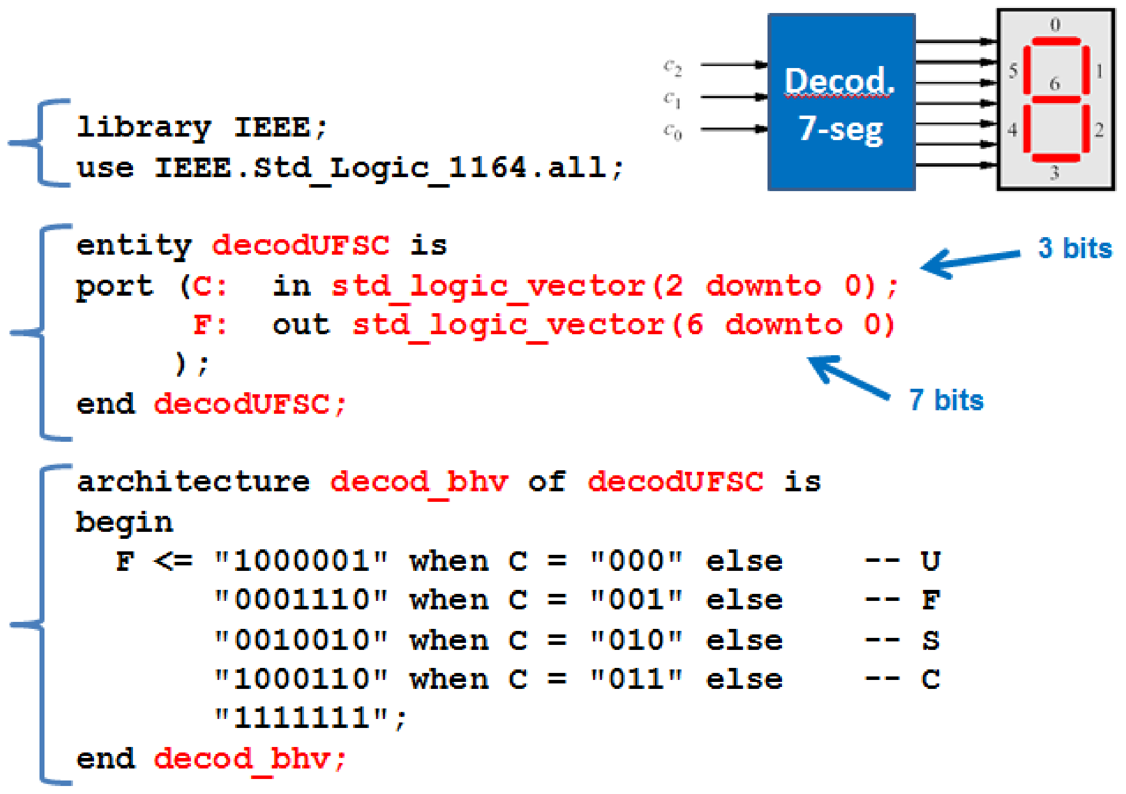
- Using the Sum of Products synthesis method

```
F(0) <= C2' C1' C0' + C2 C1 C0
F(1) <= C2' C1' C0 + C2' C1 C0' + C2' C1 C0 + C2 C1 C0
F(2) <= …
```

- Using behavioral analysis

```
F <= "1000001" when C2C1C0 = "000" else
     "0001110" when C2C1C0 = "001" else
     ...
     else "1111111"
```

The full VHDL implementation for the proposed decoder component, based on the behavioral analysis method is shown in Figure 5.12.



```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity decodUFSC is
port (C:  in std_logic_vector(2 downto 0);
      F:  out std_logic_vector(6 downto 0)
     );
end decodUFSC;

architecture decod_bhv of decodUFSC is
begin
  F <= "1000001" when C = "000" else    -- U
       "0001110" when C = "001" else    -- F
       "0010010" when C = "010" else    -- S
       "1000110" when C = "011" else    -- C
       "1111111";
end decod_bhv;
```

Figure 5.12. VHDL code for the UFSC 7-segment decoder.

5.5 Case Study: A Simple but Fully Functional Calculator

A case study, adapted from previous chapters is presented next. The case study follows the same structure as shown in Figure 4.8, but considering a multi-bit implementation as shown in Figure 5.1. Furthermore, instead of components implementing arbitrary logic functions, in the proposed case study the components are conceived targeting the design of a simple but actual calculator.

It is an 8 bits calculator (operands are 8 bits long) that performs 4 operations, one arithmetic (F1, summation) and three logic functions (F2, F3 and F4), as follows:

- F1 = A + B
- F2 = A or B
- F3 = A xor B
- F4 = not A

Figure 5.13 shows the switches, displays and LEDs used by the calculator in the DE2 board.

The calculator's usage is straightforward:

1. Operand A is provided in switches $SW_{7..0}$
2. Operand B is provided in switches $SW_{15..0}$
3. One of the four operations is provided in switches $SW_{17..16}$
4. Result is shown in hexadecimal on displays HEX_1 and HEX_0 and in binary on $LEDR_{7..0}$ (red LEDs)

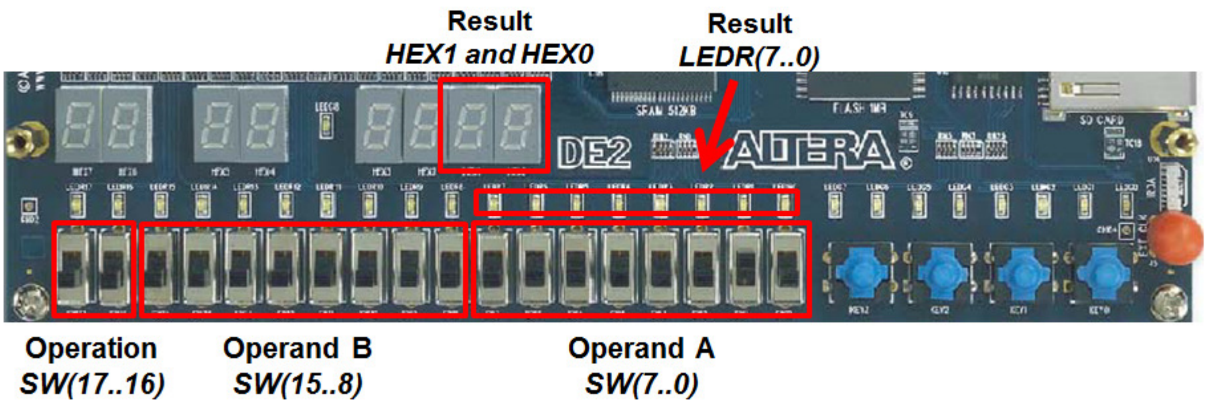


Figure 5.13. Inputs and outputs for the calculator using DE2 board.

In Table 5.1, all possible combinations for the selection bits SW_{17} and SW_{16} are listed. For instance, if the user wants to perform an exclusive or (xor) operation between inputs A (provided in $SW_{7..0}$) and B (provided in $SW_{15..8}$), SW_{17} should be switched on ('1') and SW_{16} should be switched off ('0'). The operation result is shown on the 7-segment displays and LEDs straightaway.

Table 5.1. Operations performed by the calculator.

Input $SW_{17..16}$	Output $LEDR_{7..0}$ and 7-seg displays
0 0	A + B
0 1	A or B
1 0	A xor B
1 1	Not A

The block diagram in Figure 5.14 shows the calculator input/output (I/O) signals, and the top-level component "top_calc", which is used to glue together the internal components. It shows also the two 7-segment displays and the eight LEDs used to output the calculator results. The internal components and all the connections are shown in Figure 5.15.

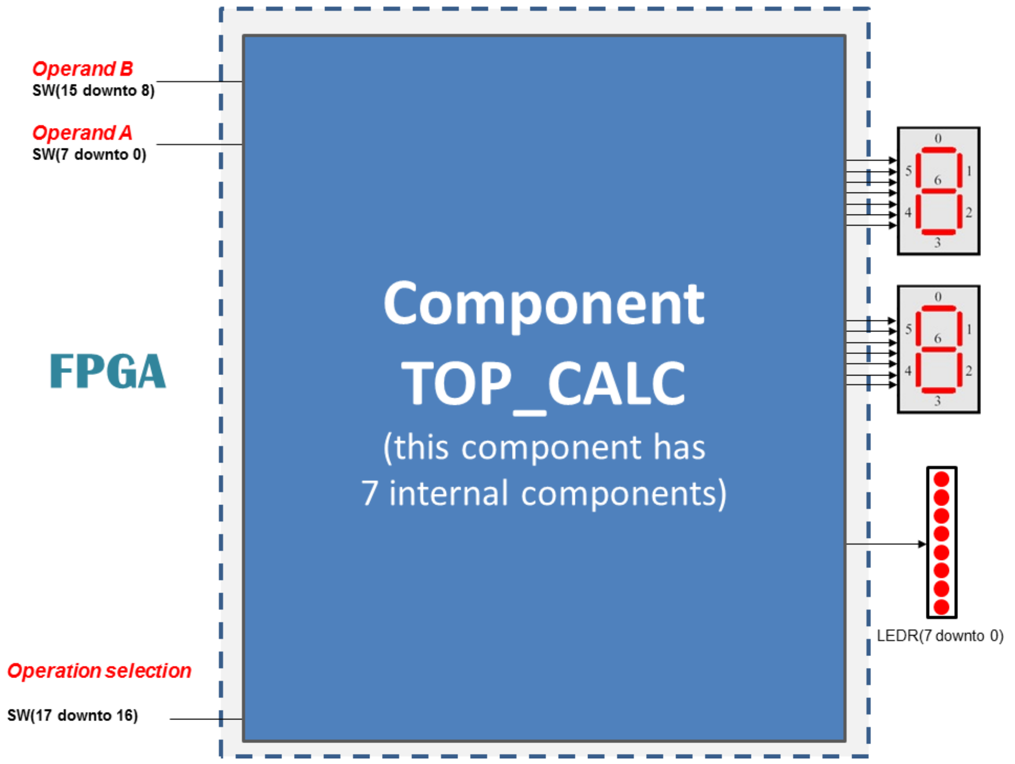


Figure 5.14. Top-level view of the calculator's circuit.

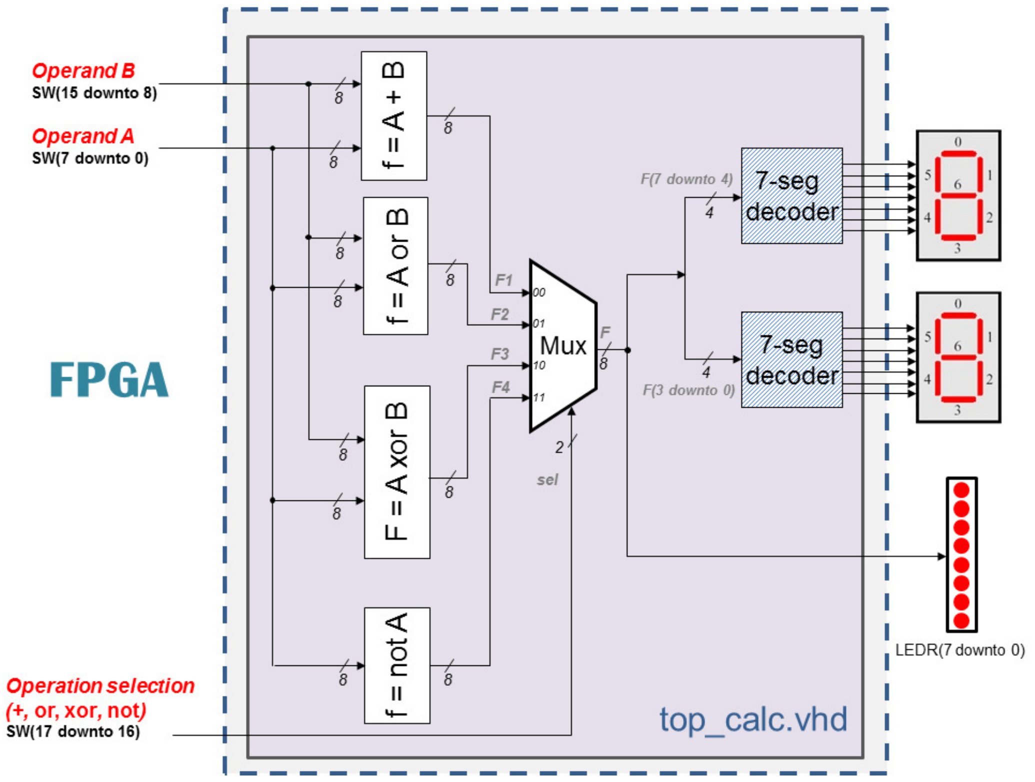


Figure 5.15. Calculator internal components.

The listing in Figure 5.16 is a VHDL implementation for the top-level component shown in Figure 5.14. In this implementation there are some important remarks:

- The “entity” section lists the signals used to identify the switches, 7-segments displays and LEDs, according to the *DE2_pin_assignments.qsf* file, as discussed in Chapter 2, Step 5;
- The internal signals, used to connect the internal components, are declared right below the “architecture” statement;
- Components C1 and C4 are listed next, but the VHDL code for components C2 and C3 are not included, as they are exactly the same as component C1;
- The Decod7seg component declaration is also not included, as this is the task to be achieved in this laboratory session;
- All the connections shown in Figure 5.15 are performed by the VHDL code listed in the body of the architecture (right after the “begin” statement);
- Components C1, C2, C3 and C4 provide signals F1, F2, F3 and F4 as their outputs;
- The mux4x1 component has F1, F2, F3 and F4 as its inputs. $SW_{17..16}$ is the operation selection, and the internal signal F is its output;
- The Decod7seg component was declared just once in the architecture’s declarations section, and in the architecture’s body the two copies of the component are instantiated (see the two pointers to L6 and L7).

```
library ieee;
use ieee.std_logic_1164.all;
entity top_calc is
 port ( SW : in std_logic_vector (17 downto 0);
        HEX0, HEX1: out std_logic_vector (6 downto 0);
        LEDR : out std_logic_vector (17 downto 0)
 );
end top_calc;
architecture top_stru of top_calc is
  signal F, F1, F2, F3, F4: std_logic_vector (7 downto 0);
  component C1
    port (A : in std_logic_vector(7 downto 0);
          B : in std_logic_vector(7 downto 0);
          F : out std_logic_vector(7 downto 0));
  end component;
  -- components C2 and C3, same as C1
  component C4
    port (A : in std_logic_vector(7 downto 0);
          F : out std_logic_vector(7 downto 0)
    );
  end component;
 component mux4x1
   port (w, x, y, z: in std_logic_vector(7 downto 0);
                  s: in std_logic_vector(1 downto 0);
                  m: out std_logic_vector(7 downto 0)
   ) ;
 end component;
 -- Add component Decod7seg here
 -- Attention!! Only one declaration for Decod7seg
```

```
begin

 L1: C1 port map (SW(7 downto 0),
        SW(15 downto 8), F1);

 L2: C2 port map (SW(7 downto 0),
        SW(15 downto 8), F2);

 L3: C3 port map (SW(7 downto 0),
        SW(15 downto 8), F3);

 L4: C4 port map (SW(7 downto 0), F4);

 L5: mux4x1 port map (F1, F2, F3, F4,
        SW(17 downto 16), F);

→L6: Decod7seg port map (F(3 downto 0),
        HEX0);

→L7: Decod7seg port map (F(7 downto 4),
        HEX1);

 LEDR(7 downto 0) <= F;

end top_stru;   -- END architecture
```

Figure 5.16. VHDL implementation for the top circuit of Figure 5.14.

The VHDL implementations for the remaining components, but for the 7-segments decoder, are provided next.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_unsigned.all; -- to use the + operator

entity C1 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C1;

architecture circuit of C1 is
begin
  F <= A + B;
end circuit;
```

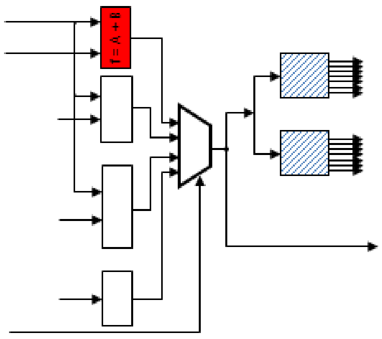


Figure 5.17. VHDL implementation for the C1 component.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C2 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C2;

architecture circuit of C2 is
begin
  F <= A or B;
end circuit;
```

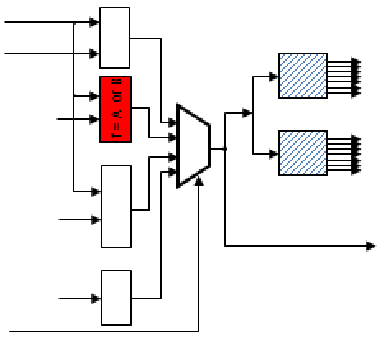


Figure 5.18. VHDL implementation for the C2 component.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C3 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C3;

architecture circuit of C3 is
begin
  F <= A xor B;
end circuit;
```

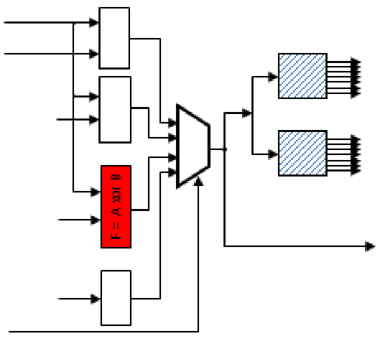


Figure 5.19. VHDL implementation for the C3 component.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C4 is
port (A: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C4;

architecture circuit of C4 is
begin
  F <= not A;
end circuit;
```

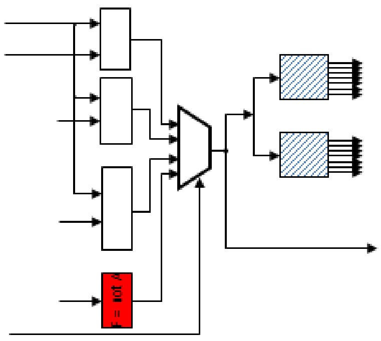


Figure 5.20. VHDL implementation for the C4 component.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity mux4x1 is
port (w, x, y, z: in std_logic_vector(7 downto 0);
       s: in std_logic_vector(1 downto 0);
       m: out std_logic_vector(7 downto 0)
      );
end mux4x1;

architecture circuit of mux4x1 is
begin
 m <= w when s = "00" else
      x when s = "01" else
      y when s = "10" else
      z;
end circuit;
```

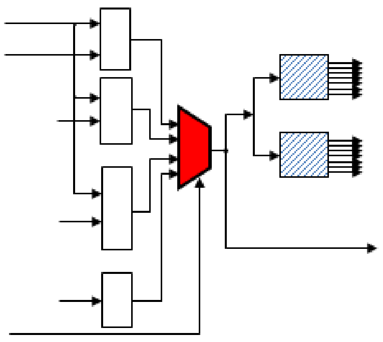


Figure 5.21. VHDL implementation for the mux4x1 component.

5.6 Laboratory Assignment

The laboratory objectives are:

- to implement a 7-segments decoder in VHDL;
- to add the implemented decoder to the calculator.

Using the components provided in Section 5.5, write the VHDL implementation for the 7-segments decoder, and make the necessary modifications in the top_calc component as indicated in Figure 5.16.

The whole design has in total 7 files (8 components), as follows:

- *c1.vhd*, provided in Figure 5.17;
- *c2.vhd*, provided in Figure 5.18;
- *c3.vhd*, provided in Figure 5.19;
- *c4.vhd*, provided in Figure 5.20;
- *mux4x1.vhd*, provided in Figure 5.21;
- *decod7seg.vhd*, to be designed (see Section 5.4 and Figure 5.12);
- *top_calc.vhd*, partially provided in Figure 5.16.

Before starting the modifications in the top-level file (*top_calc.vhd*), make sure to have a good understanding of component declaration and instantiation procedures in VHDL. As pointed out in Figure 5.16, just one *decod7seg* should be included in the components declaration section of the architecture, and two copies of this component should be instantiated using *port map* statements.

The VHDL implementation listed in Figure 5.12 decodes only four codes, “000” (‘U’), “001” (‘F’), “010” (‘S’), and “011” (‘C’). The calculator needs to displays all possible hexadecimal values, which means that the decoder receives 16 different input data, as shown in Figure 5.22. The expected decoded outputs are shown in Figure 5.22, but the values in the range between 9H and DH. Thus, to design the decoder, the missing codes should be defined, and the VHDL implementation in Figure 5.12 should be adapted according. The entity for the 7-segments decoder, as shown in Figure 5.15, has a 4 bits input signal, and a 7 bits output signal.

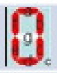
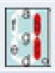
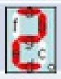
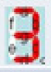
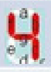
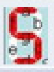
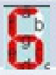
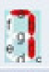
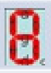
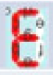
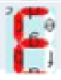
Input	Output *6543210*	Display
0000	1000000	
0001	1111001	
0010	0100100	
0011	0110000	
0100	0011001	
0101	0010010	
0110	0000010	
0111	1111000	
1000	0000000	
...	...	9, A, b, C, d
1110	0000110	
1111	0001110	

Figure 5.22. Binary to 7-segments decoding table.

The tasks to be completed in this laboratory session, using Altera’s Quartus II EDA tool are as follows:

- Create a project with all the components shown in Figure 5.15;

- Create a 7-segments decoder component using the VHDL design entry editor;
- Modify the top-level component provided in Figure 5.16, in order to add the new component declaration. The two instances are already created and identified by labels L6 and L7;
- Perform the synthesis;
- Perform the simulation and fix errors, if any;
- Prototype and test the circuit in the FPGA board.

The **project creation** process has been well discussed and explained in previous chapters. For the **synthesis** activity, it is essential to import the *DE2_pin_assignments.qsf* file, as discussed in Chapter 2, Step 5, and shown in Figure 2.7.

The **simulation** activity is performed using ModelSim. To start the simulator in Quartus II, select *Tools –> Run Simulation Tool –> RTL Simulation*. Wait for ModelSim to open and perform the following steps:

- Simulate –> Start Simulation;
- In the "Start Simulation" window, look for the "Design" tab;
- In the "work" Library, click on the "+" sign, and select the "top_calc" Entity;
- Click OK to start the simulation;
- In the Objects window drag and drop the SW vector (A, B, and the two selection bits) in the Wave window. Do the same for $LEDR_{7..0}$, $HEX0_{6..0}$, $HEX1_{6..0}$, F, F1, F2, F3 and F4.
- Using the "Force" option, as explained in Chapter 1 "Step 4", set $SW_{7..0}$ to 03H (A = 00000011), $SW_{15..8}$ to 02H (B = 00000010), and $SW_{17..16}$ to 00. Set the simulation run length to 100 ps, and press the *Run* button. Next, set $SW_{17..0}$ to the remaining values listed in Table 5.2, always running for 100 ps after each input combination.

In Figure 5.23, the selection bits SW_{17} and SW_{16} are set to "00", "01", "10", and "11", in order to exercise all four operations. The A ($SW_{7..0}$) and B ($SW_{15..8}$) inputs are set to the sequence of values as listed in Table 3.1.

When the expected results are obtained in the simulation process, the next step is the **FPGA prototyping**. To test the calculator functionality in an actual piece of hardware, go back to Quartus II, in choose menu *Tools –> Programmer*. Follow the instructions provided in Chapter 1, Step 5, observing the "hardware setup" instructions in Figure 1.32 and in Figure 1.33. Test the circuit in the FPGA board, using switches $SW_{15..0}$ to provide the A and B inputs, and switches $SW_{17..16}$ to choose the operation to be performed. The operation result is shown in binary in $LEDR_{7..0}$, and in hexadecimal on the 7-segment displays HEX0 and HEX1.

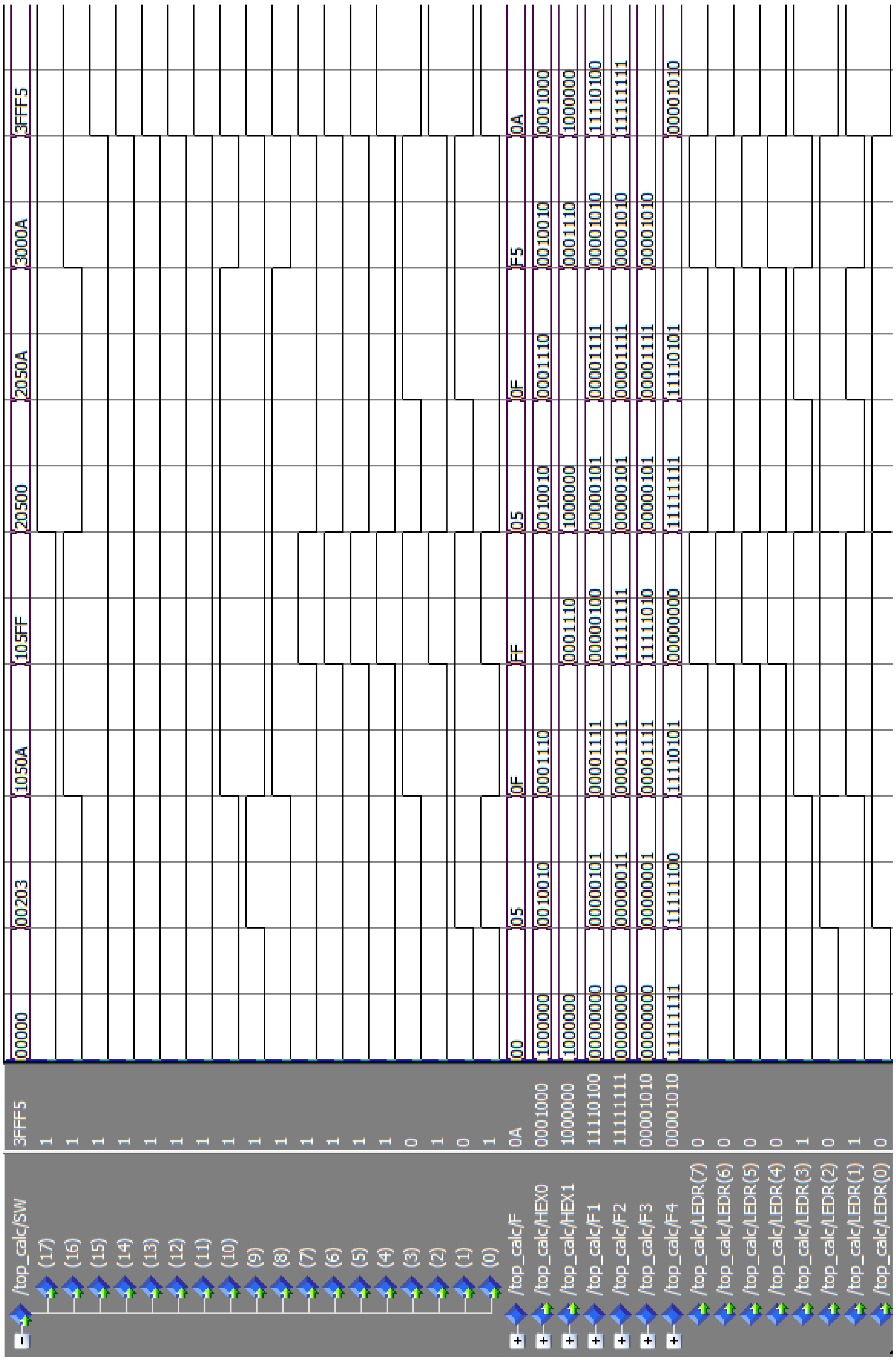


Figure 5.23. Expected simulation results.

In Table 5.2, the first column has some input value examples. The second column is filled in with the selection bits ("00", "01", "10", and "11"). The third column should be filled in with the results obtained from the manual evaluation of equations F1, F2, F3 and F4. The fourth and fifth columns can be obtained straight from the simulation results, according to the waveforms in Figure 5.23. The sixth and seventh columns should be filled in with the FPGA execution results.

Table 5.2. Truth table for the calculator.

Inputs		**Outputs**				
$SW_{15..0}$ B A	**$SW_{17..16}$ Selection**	***F1 = A + B*** ***F2 = A or B*** ***F3 = A xor B*** ***F4 = not A***	**Simulation HEX1 HEX0**	**Simulation $LEDR_{6..0}$**	**FPGA HEX1 HEX0**	**FPGA $LEDR_{6..0}$**
00 00	**00**	F1 =				
02 03	**00**	F1 =				
05 0A	**01**	F2 =				
05 FF	**01**	F2 =				
05 00	**10**	F3 =				
05 0A	**10**	F3 =				
00 0A	**11**	F4 =				
FF F5	**11**	F4 =				