ECSE 222 Lab 1

7-Segment Decoder & 5-Bit Adder

Group 56
Glen Xu 260767363 & Cheng Chen 260775674
March 2019

Introduction

In this lab, we are aiming to design and implement a 7-segment decoder and a 5-bit adder written in VHDL language by Altera Quartus II FPGA design software. The ModelSim simulation program will be used for digital simulation, and the code will run on the Altera DE1-SoC board.

7-Segment Decoder

Each decoder has 7 segments that can be turned on or off individually, as shown in the picture below. When all segments are controlled together, numbers can be displayed. In this lab, the digits in hexadecimal system will be displayed, i.e. 0~9 and a~f.

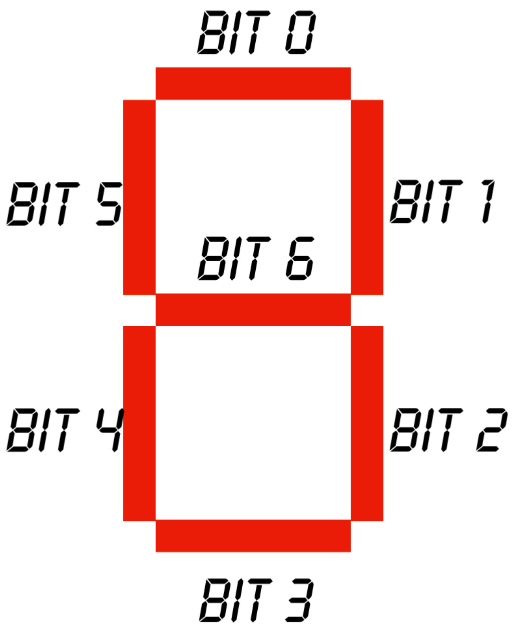


Figure 1 7-segment LED display

On the Altera DE1-SoC board, the 7-segment LED displays are active low, which means that when the input bit is 0, the segment is turned on, and when the input bit is 1, the segment is turned off. The display of all the hexadecimal digits are shown in the following diagram.

In VHDL language, each input of the decoder has to be a 4-bit binary number. For example, f in hexadecimal system has to be inputed as 1111. We implemented the decoder in VHDL language by creating the decoder entity, with 4-bit vectors as input, and 7-bit vectors as output. We then assigned each 7-bit vector to their corresponding 4-bit vector in the architecture.

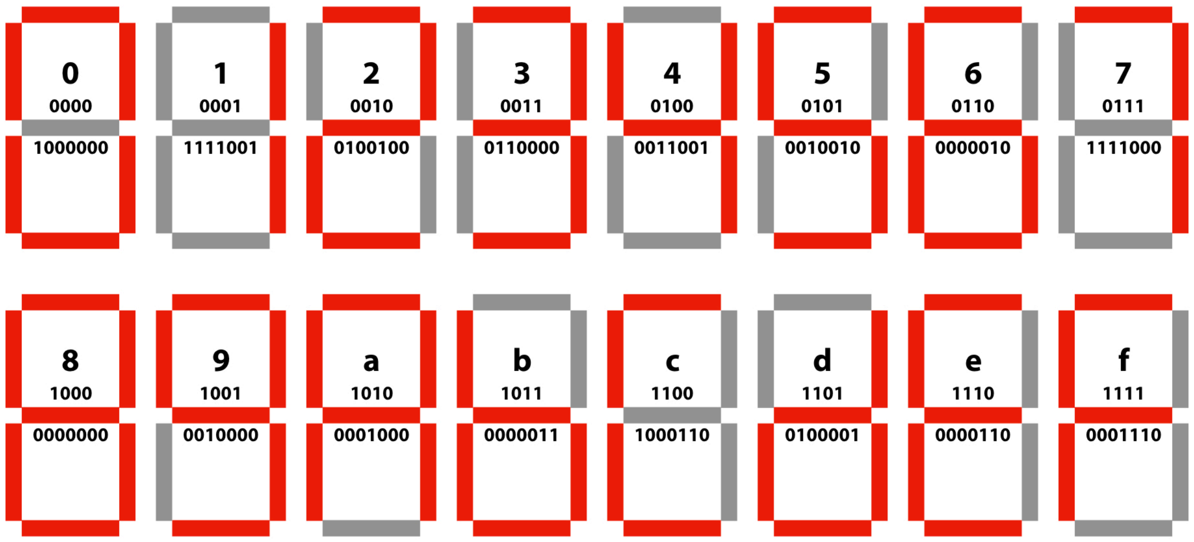


Figure 2 7-segment encoded output for hexadecimal digits

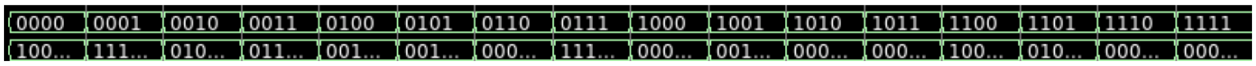
```
--import libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Declare entity
entity g56_7_seg_decoder is
	Port (
		code		: in std_logic_vector (3 downto 0);
		segments		: out std_logic_vector (6 downto 0)
		);
end g56_7_seg_decoder;

architecture behaviour of g56_7_seg_decoder is

begin

	process(code)
	begin
		case code is
				when "0000" => segments <= "1000000"; -- 0
				when "0001" => segments <= "1111001"; -- 1
				when "0010" => segments <= "0100100"; -- 2
				when "0011" => segments <= "0110000"; -- 3
				when "0100" => segments <= "0011001"; -- 4
```

```
                    when "0101" => segments <= "0010010"; -- 5
                    when "0110" => segments <= "0000010"; -- 6
                    when "0111" => segments <= "1111000"; -- 7
                    when "1000" => segments <= "0000000"; -- 8
                    when "1001" => segments <= "0010000"; -- 9
                    when "1010" => segments <= "0001000"; -- a 10
                    when "1011" => segments <= "0000011"; -- b 11
                    when "1100" => segments <= "1000110"; -- c 12
                    when "1101" => segments <= "0100001"; -- d 13
                    when "1110" => segments <= "0000110"; -- e 14
                    when "1111" => segments <= "0001110"; -- f 15
                    when others => segments <= "1010101";
               end case;

     end process;

end behaviour;
```

We successfully compiled the VHDL code of the decoder in Altera Quartus II FPGA. We then generated a test bench framework with Altera Quartus II FPGA, and then added a loop to input all the 4-bit binary numbers.

```
test : PROCESS
-- optional sensitivity list
-- (       )
-- variable declarations
BEGIN
     for i in 0 to 15 loop
          code <= std_logic_vector(to_unsigned(i,4));
          wait for 10 ns;
     end loop;
WAIT;
END PROCESS test;
```

We added both the decoder VHDL code and the test bench into the ModelSim, in order to simulate the circuit digitally. The inputs are shown on the first row, and the outputs on the next



```
0000   0001   0010   0011   0100   0101   0110   0111   1000   1001   1010   1011   1100   1101   1110   1111
100... 111... 010... 011... 001... 001... 000... 111... 000... 001... 000... 000... 100... 010... 000... 000...
```

Figure 3 digital simulation

As shown in the picture above, all the inputs and outputs match accordingly. That is how we make sure that our implementation is correct.

5-Bit Adder

The 5-bit adder takes two 5-bit binary numbers as inputs and return both the inputs and the summation result in the format of 7-segment decoder as outputs. Since each number requires 2 display areas on the board, so 6 values will be returned in total.

We first created the adder entity, with 2 inputs and 3 outputs. The 2 inputs are 5-bit binary vectors, and the 3 outputs are 13-bit vectors. The 13 bits will be break down into two 7 bits so that each number can be displayed on the 7-segment display.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity g56_lab1_adder is
     Port ( A, B      : in  std_logic_vector(4  downto 0);
       decoded_A      : out std_logic_vector(13 downto 0);
       decoded_B      : out std_logic_vector(13 downto 0);
  decoded_AplusB      : out std_logic_vector(13 downto 0));
end g56_lab1_adder;
```

Then we implemented the architecture. The 7-segment decoder entity was imported as a component, and thus instantiate intermediate signals.

```
architecture behaviour of g56_lab1_adder is
     -- import the 7-seg decoder(g56_lab1_1503) we implemented before
     component g56_7_seg_decoder is
          port(
               code            : in std_logic_vector(3 downto 0);
               segments        : out std_logic_vector(6 downto 0)
          );
     end component g56_7_seg_decoder;
     -- temporary variable to store 8-bit sum
     signal AplusB         : std_logic_vector(7 downto 0);
```

For the vectors A, B and AplusB, their least significant 4 bits can be easily be represented in a hexadecimal number. The most significant leftover bit, however, is not a full 4-bit number, so that it cannot be converted into a hexadecimal number directly. Instead, zeros have to be added with the most significant bit to make a full 4-bit binary number. We implemented this into the behaviour of the adder.

```
-- output
     -- A least sig digit
     decoderA0: g56_7_seg_decoder PORT MAP(code => A(3 downto 0),
segments => decoded_A(6 downto 0));
     -- A most sig digit
     decoderA1: g56_7_seg_decoder PORT MAP(code => ("000" & A(4)),
segments => decoded_A(13 downto 7));
     -- B least sig digit
     decoderB0: g56_7_seg_decoder PORT MAP(code => B(3 downto 0),
segments => decoded_B(6 downto 0));
     -- B most sig digit
     decoderB1: g56_7_seg_decoder PORT MAP(code => ("000" & B(4)),
segments => decoded_B(13 downto 7));

     AplusB <= std_logic_vector(unsigned("000" & A) + unsigned("000" &
B));
     -- A + B least sig digit
     decoderAplusB0: g56_7_seg_decoder PORT MAP(code => AplusB(3
downto 0), segments => decoded_AplusB(6 downto 0));
     -- A + B most sig digit
     decoderAplusB1: g56_7_seg_decoder PORT MAP(code => AplusB(7
downto 4), segments => decoded_AplusB(13 downto 7));
end behaviour;
```

As shown in the code, a total of six 7-segment decoder components are instantiated. Each of these decoders is corresponded to each 7-segment display. After successfully compiling the all the codes, we connected the inputs and outputs to the switch and LED on the Altera board, by the pin planner of Quartus Prime. After downloading the circuit, we can set the input by the switches and see the output by the 7-segment LED displays on the board.

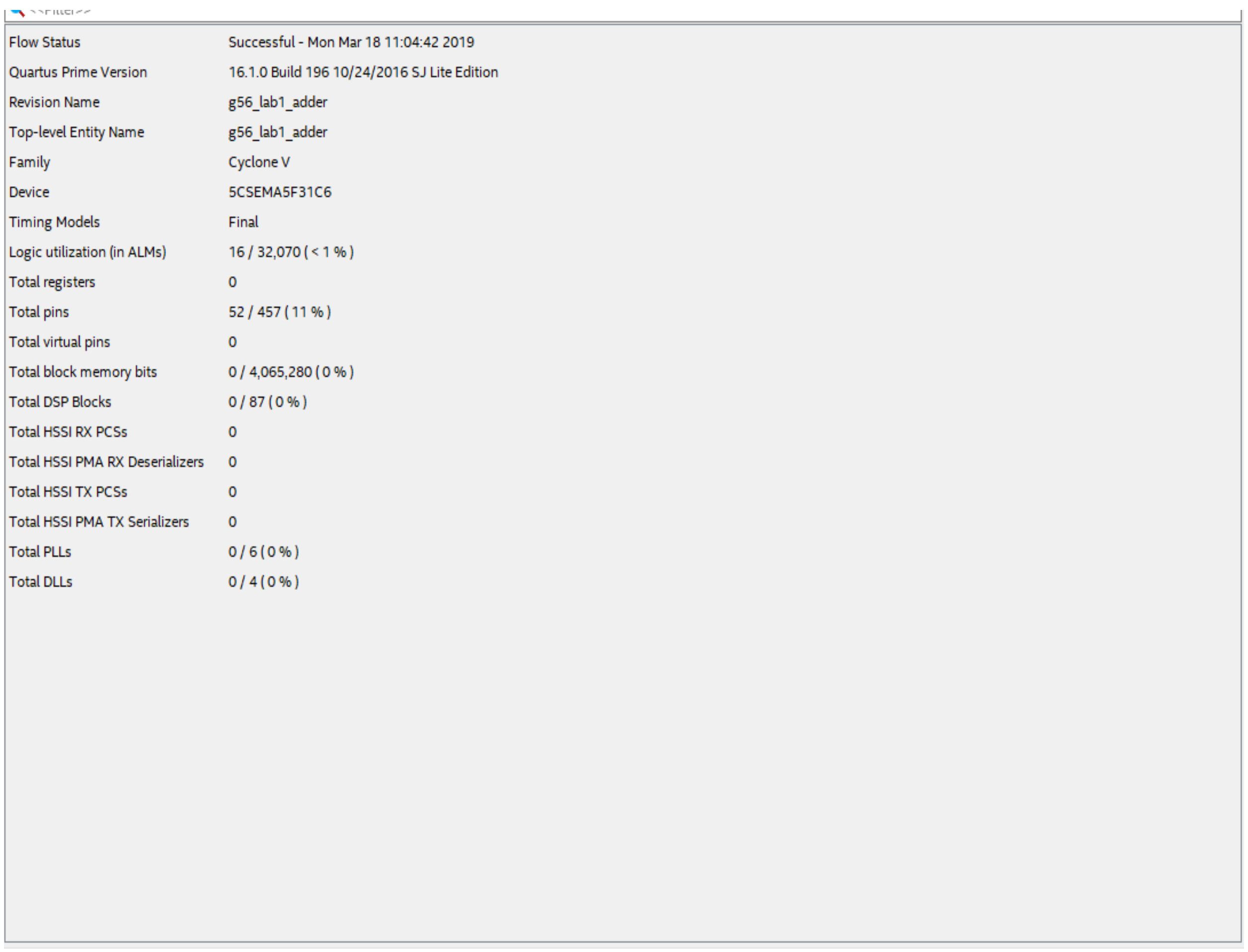
Tester

Discussion: We wrote the code on the Quartus software and tested it with the Altera DE1-SoC board. At the beginning of the test, we did not input any number(i.e. Operands are zero). Theoretically, we should get zero for the sum which matched the number shown on the board during the test. Also, we let both operands be 11111(1F in hex). Theoretically, the sum should be 3F which also matched the number shown on the board. Then, we used several other numbers to add and they all matched the sum we should get in theory. Thus, our 5-bit adder was implemented correctly.

FPGA resource utilization

We can see from the Compilation Report that our top level entity is g56_lab1_adder and in this lab we used device 5CSEMA5F31C6. In the RTL schematic diagram for 7-segment decoder, taking B1 as an example, “code” is the input and “segments is the output”. In the VHDL code we wrote (under “begin” section), different code (4 bit) maps to its corresponding segments (7 bit) which is shown by the Mux drawn in the RTL diagram. For example, code: 0000 maps to segments: 1000000. In the RTL schematic diagram for adder, we used A and B as input and decoded_A, decoded_B and decoded_AplusB as output. As shown in the diagram, we first added

A and B together and its VHDL code is “AplusB <= std_logic_vector(unsigned("000" & A) + unsigned("000" & B));” . Then, we used six 7-segment decoder to decode the sum as shown by the six green boxes in the RTL diagram which corresponds to the “begin” section in the VHDL code.



Flow Status	Successful - Mon Mar 18 11:04:42 2019
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g56_lab1_adder
Top-level Entity Name	g56_lab1_adder
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	16 / 32,070 (< 1 %)
Total registers	0
Total pins	52 / 457 (11 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 4 compilation report

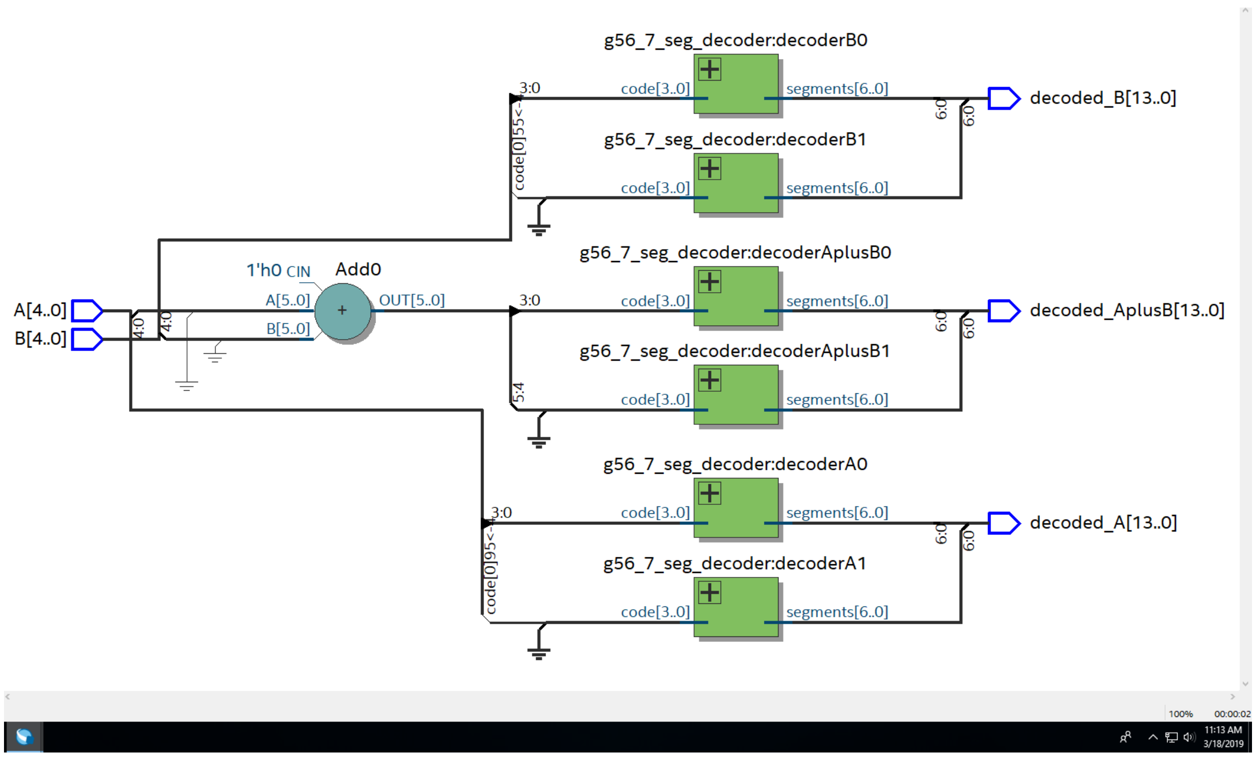

g56_7_seg_decoder:decoderB0
g56_7_seg_decoder:decoderB1
g56_7_seg_decoder:decoderAplusB0
g56_7_seg_decoder:decoderAplusB1
g56_7_seg_decoder:decoderA0
g56_7_seg_decoder:decoderA1
code[3..0]
segments[6..0]
decoded_B[13..0]
decoded_AplusB[13..0]
decoded_A[13..0]
A[4..0]
B[4..0]
1'h0 CIN
Add0
A[5..0]
B[5..0]
OUT[5..0]
3:0
5:4
6:0
4:0
code[0]55<-
code[0]95<-

Figure 5 RTL schematic diagram for adder

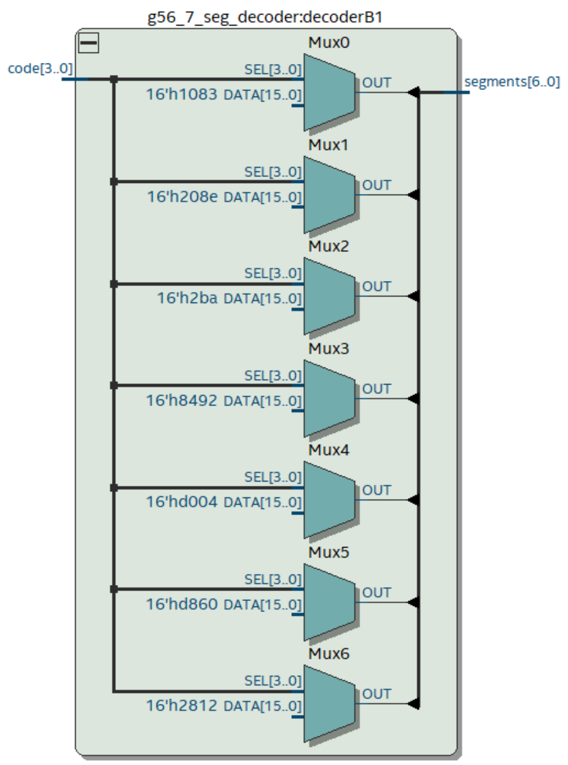


Figure 6 RTL schematic diagram for 7-segment decoder