Students: Kevin (Kaiwen) Xue - 20814292, Anni Wang - 20849760

BME 393L: Lab 3

4.1 Four-bit adder by employing Seven-segment

The code for the four-bit adder is shown in **Figure 1** below.

```
-- ^^^ seven_segment entity and architecture declaration above ^^^
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity full_adder is port(
         SW:      in std_logic_vector(9 downto 0); -- The 4 bit data to be displayed
         HEX3:    out std_logic_vector(6 downto 0);
         HEX2:    out std_logic_vector(6 downto 0);
         HEX1:    out std_logic_vector(6 downto 0);
         HEX0:    out std_logic_vector(6 downto 0)
         );
end entity full_adder;

architecture full_adder_architecture of full_adder is

component seven_segment is port (
      data_in: in std_logic_vector(3 downto 0);
      blank: in std_logic;
      data_out: out std_logic_vector(6 downto 0)
);
end component;
      signal c: unsigned(4 downto 0);
      signal a: std_logic_vector(3 downto 0);
      signal b: std_logic_vector(3 downto 0);
begin
      hex2_inst: entity work.seven_segment(behavioral) port map (sw(3 downto 0), '0', hex2);
      hex3_inst: entity work.seven_segment(behavioral) port map (sw(9 downto 6), '0', hex3);

      hex0_inst: entity work.seven_segment(behavioral) port map (a, '0', hex0);
      hex1_inst: entity work.seven_segment(behavioral) port map (b, '0', hex1);


      c <= ('0' & unsigned(sw(3 downto 0))) + ('0' & unsigned(sw(9 downto 6)));
      a <= std_logic_vector(c(3 downto 0));
      b <= ("000" & std_logic(c(4)));


end architecture full_adder_architecture;
```

Figure 1: VHDL code with the four-bit adder entity declaration and sending the input and output values to seven-segment HEX displays

Next, the VHDL code was deployed to the FPGA board and tested. The code being tested on the FPGA board is shown in **Figure 2** below.

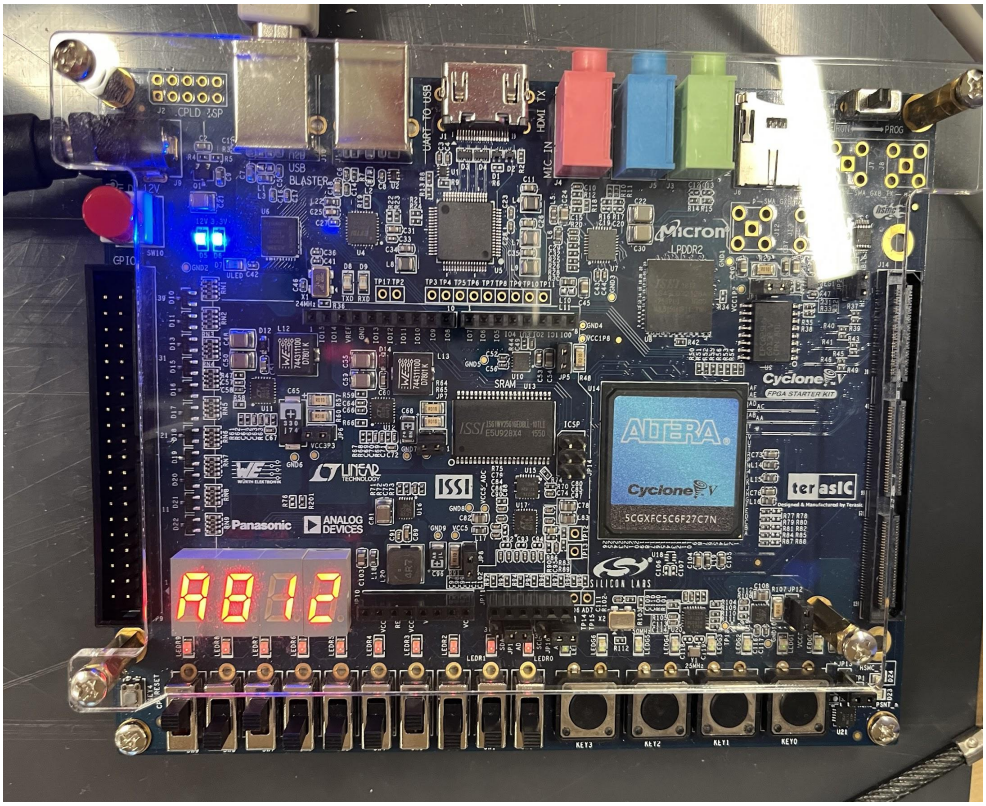

Figure 2: Testing VHDL code from **Figure 1** with the four bit adder and sending the input values to HEX3 and HEX2 and output value to HEX1 and HEX0. Showing the case of the A + 8 = 12.

4.2 Heart bit counter

The code for the heart bit counter is shown in **Figure 3** below.

```
-- ^^^ seven_segment entity and architecture above |^^^
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity heart_bit_counter is port(
         KEY: in std_logic_vector(3 downto 0);
         HEX1:    out std_logic_vector(6 downto 0);
         HEX0:    out std_logic_vector(6 downto 0);
         LEDG:    out std_logic_vector(4 downto 0)
         );
end entity heart_bit_counter;

architecture heart_bit_counter_architecture of heart_bit_counter is

component seven_segment is port (
      data_in: in std_logic_vector(3 downto 0);
      blank: in std_logic;
      data_out: out std_logic_vector(6 downto 0)
);
end component;
signal counter: unsigned(4 downto 0);
signal counter_intermediate: std_logic_vector(4 downto 0);
signal blank_hex1: std_logic;
begin
      hex0_inst: entity work.seven_segment(seven_segment_architecture) port map (counter_intermediate(3 downto 0), '0', hex0);
      hex1_inst: entity work.seven_segment(seven_segment_architecture) port map (("000" & counter_intermediate(4)), blank_hex1, hex1);

      counter <= counter + 1 when rising_edge(KEY(0));
      counter_intermediate <= std_logic_vector(counter);
      LEDG <= counter_intermediate;

      blank_hex1 <= '0' when (counter_intermediate(4) = '1') else '1';

end architecture heart_bit_counter_architecture;
```

Figure 3: VHDL code with the heart bit counter entity and architecture declaration and sending the outputs to the seven-segment HEX displays and the green leds (LEDG)

Next, the VHDL code was deployed to the FPGA board and tested. The code being tested on the FPGA board is shown in **Figure 4** and **Figure 5** below.

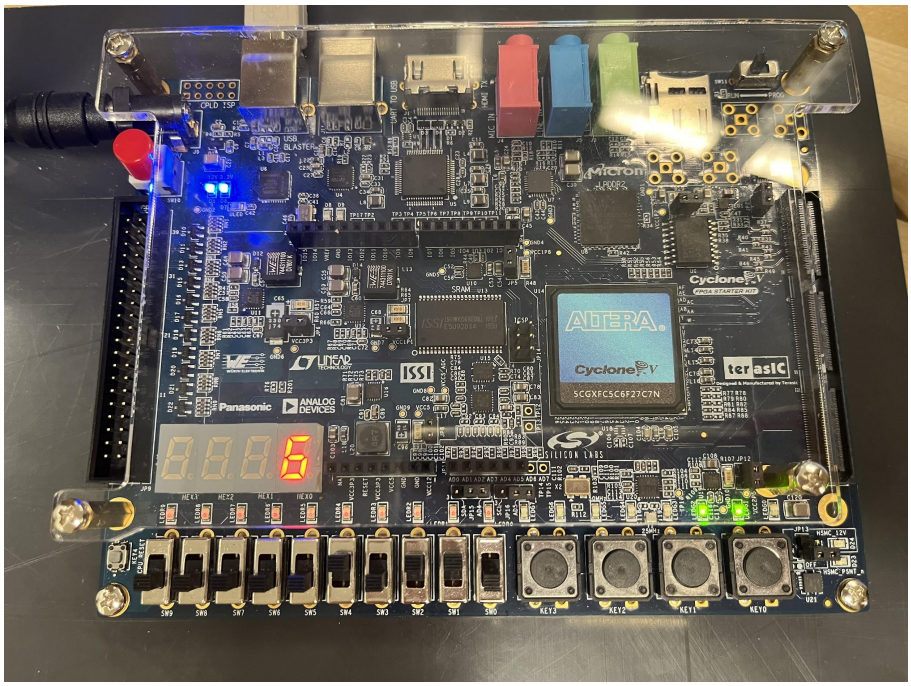

Figure 4: Testing VHDL code from **Figure 3** with the heart bit counter, and sending the output value to HEX1, HEX0, and green leds (LEDG). Showing the output of 6, with HEX1 blanked.

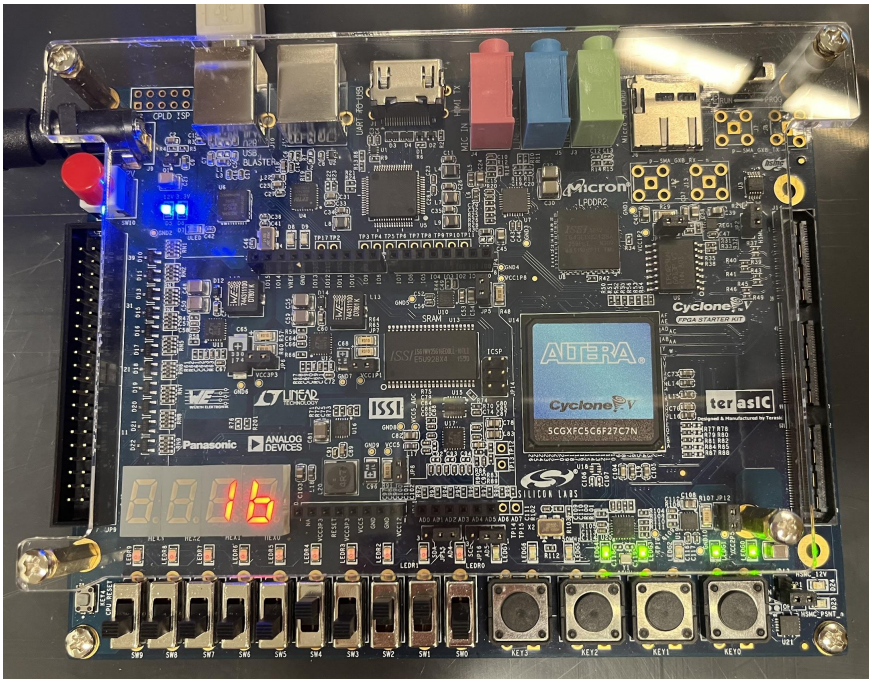

Figure 5: Testing VHDL code from **Figure 3** with the heart bit counter and sending the output value to HEX1, HEX0, and green leds (LEDG). Showing the output of 1b, which requires both HEX1 and HEX0.

4.3 Elevator controller

The code for the elevator controller is shown in **Figure 3** below. Inputs A and B (SW[1] and SW[0] respectively) represent the current floor of the elevator, and inputs Y and Z (SW[9] and SW[8] respectively) represent the target floor of the elevator. LEDR[0] represents the output for ENABLE, and LEDG[0] represents the output for DIRECTION.

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

entity elevator_controller is port(
         SW: in std_logic_vector(9 downto 0);
         LEDG:    out std_logic_vector(4 downto 0);
         LEDR:    out std_logic_vector(4 downto 0)
         );
end entity elevator_controller;

architecture elevator_controller_architecture of elevator_controller is
signal a,b,y,z: std_logic;
begin
      a <= SW(1);
      b <= SW(0);
      y <= SW(9);
      z <= SW(8);
      LEDR(0) <= (a and z and (not b or not y)) or (b and y and (not a or not z));
      LEDG(0) <= (not a or (y and z));

end architecture elevator_controller_architecture;
```

Figure 6: VHDL code with the elevator controller entity and architecture declaration, sending the outputs to red (LEDR[0]) and green (LEDG[0])

Next, the VHDL code was deployed to the FPGA board and tested. The code being tested on the FPGA board is shown in **Figure 7** and **Figure 8** below.

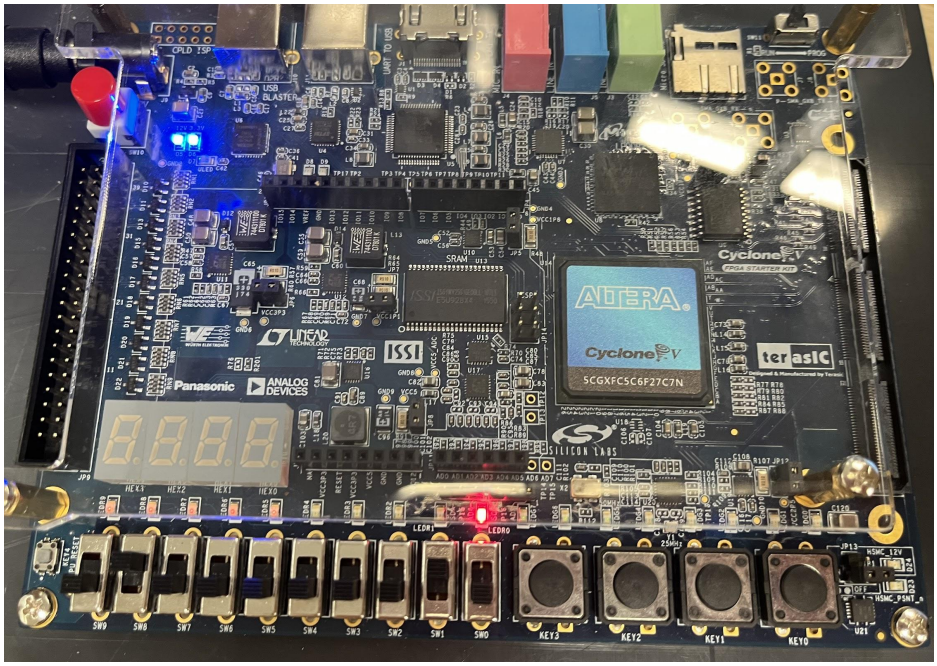

Figure 7: Testing VHDL code from **Figure 6** with the elevator controller and sending the outputs to LEDR[0] for ENABLE and LEDG[0] for DIRECTION. Showing the case of the current floor of 2 with the target floor of 1.

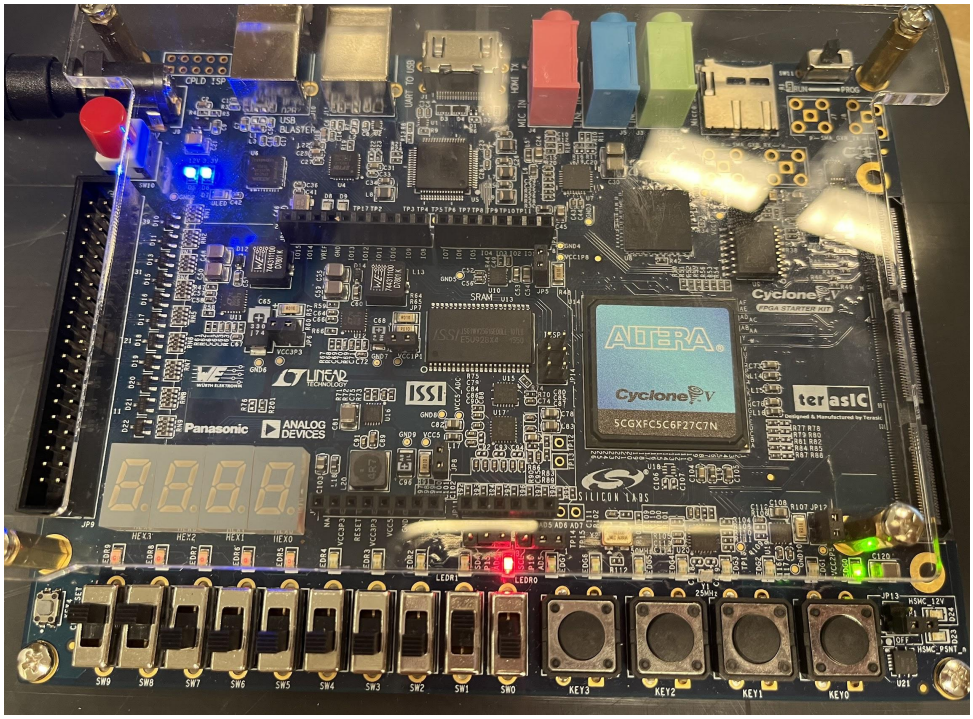

Figure 8: Testing VHDL code from **Figure 6** with the elevator controller and sending the outputs to LEDR[0] for ENABLE and LEDG[0] for DIRECTION. Showing the case of the current floor of 2 with the target floor of 3.

4.4 Component instantiation

The code for the four-bit multiplier is shown in **Figure 9** and **Figure 10** below. The seven-segment entity and architecture declaration weren't included for the sake of space, but they were used for displaying operands and the product on the HEX displays.

```
-- ^^^ seven_segment entity and architecture declaration above ^^^
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity four_bit_adder is port(
         input_a:    in std_logic_vector(3 downto 0);
         input_b:    in std_logic_vector(3 downto 0);
         output_sum:    out std_logic_vector(4 downto 0)
         );
end entity four_bit_adder;

architecture four_bit_adder_architecture of four_bit_adder is
begin
      output_sum <= std_logic_vector(('0' & unsigned(input_a(3 downto 0))) + ('0' & unsigned(input_b(3 downto 0))));
end architecture four_bit_adder_architecture;

-- ^^^ four_bit_adder entity and architecture declaration above ^^^
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity four_bit_multiplier is port(
         SW:       in std_logic_vector(7 downto 0);
         HEX3:     out std_logic_vector(6 downto 0);
         HEX2:     out std_logic_vector(6 downto 0);
         HEX1:     out std_logic_vector(6 downto 0);
         HEX0:     out std_logic_vector(6 downto 0)
         );
end entity four_bit_multiplier;
```

Figure 9: VHDL code with the four-bit adder entity and architecture declaration, and the four-bit multiplier entity declaration

```
architecture four_bit_multiplier_architecture of four_bit_multiplier is

component seven_segment is port (
      data_in: in std_logic_vector(3 downto 0);
      blank: in std_logic;
      data_out: out std_logic_vector(6 downto 0)
);
end component;
component four_bit_adder is port(
         input_a:    in std_logic_vector(3 downto 0);
         input_b:    in std_logic_vector(3 downto 0);
         output_sum:    out std_logic_vector(4 downto 0)
         );
end component four_bit_adder;

signal operand_1, operand_2: std_logic_vector(3 downto 0);
signal intermediate_1, intermediate_2, intermediate_3: std_logic_vector(4 downto 0);
signal product_0, product_1, product_2: std_logic;
signal four_bit_adder_inst_1_input_1, four_bit_adder_inst_1_input_2: std_logic_vector(3 downto 0);
signal four_bit_adder_inst_2_input_1, four_bit_adder_inst_2_input_2: std_logic_vector(3 downto 0);
signal four_bit_adder_inst_3_input_1, four_bit_adder_inst_3_input_2: std_logic_vector(3 downto 0);
signal hex2_input, hex3_input: std_logic_vector(3 downto 0);

begin

hex0_inst: entity work.seven_segment(behavioral) port map (SW(3 downto 0), '0', hex0);
hex1_inst: entity work.seven_segment(behavioral) port map (SW(7 downto 4), '0', hex1);
hex2_inst: entity work.seven_segment(behavioral) port map (hex2_input, '0', hex2);
hex3_inst: entity work.seven_segment(behavioral) port map (hex3_input, '0', hex3);

four_bit_adder_inst_1: entity work.four_bit_adder(four_bit_adder_architecture) port map (four_bit_adder_inst_1_input_1, four_bit_adder_inst_1_input_2, intermediate_1);
four_bit_adder_inst_2: entity work.four_bit_adder(four_bit_adder_architecture) port map (four_bit_adder_inst_2_input_1, four_bit_adder_inst_2_input_2, intermediate_2);
four_bit_adder_inst_3: entity work.four_bit_adder(four_bit_adder_architecture) port map (four_bit_adder_inst_3_input_1, four_bit_adder_inst_3_input_2, intermediate_3);

four_bit_multiplier_process: process (SW)
begin

operand_1 <= SW(3 downto 0);
operand_2 <= SW(7 downto 4);

product_0 <= operand_1(0) and operand_2(0);

four_bit_adder_inst_1_input_1 <= '0' & (operand_1(3) and operand_2(0)) & (operand_1(2) and operand_2(0)) & (operand_1(1) and operand_2(0));
four_bit_adder_inst_1_input_2 <= ((operand_1(3) and operand_2(1)) & (operand_1(2) and operand_2(1)) & (operand_1(1) and operand_2(1)) & (operand_1(0) and operand_2(1)));

product_1 <= intermediate_1(0);

four_bit_adder_inst_2_input_1 <= intermediate_1(4 downto 1);
four_bit_adder_inst_2_input_2 <= ((operand_1(3) and operand_2(2)) & (operand_1(2) and operand_2(2)) & (operand_1(1) and operand_2(2)) & (operand_1(0) and operand_2(2)));

product_2 <= intermediate_2(0);

four_bit_adder_inst_3_input_1 <= intermediate_2(4 downto 1);
four_bit_adder_inst_3_input_2 <= ((operand_1(3) and operand_2(3)) & (operand_1(2) and operand_2(3)) & (operand_1(1) and operand_2(3)) & (operand_1(0) and operand_2(3)));

hex2_input <= (intermediate_3(0) & product_2 & product_1 & product_0);
hex3_input <= intermediate_3(4 downto 1);

end process;

end architecture four_bit_multiplier_architecture;
```

Figure 10: VHDL code with the four-bit multiplier architecture declaration

Next, the VHDL code was deployed to the FPGA board and tested. The code being tested on the FPGA board is shown in **Figure 11** and **Figure 12** below.

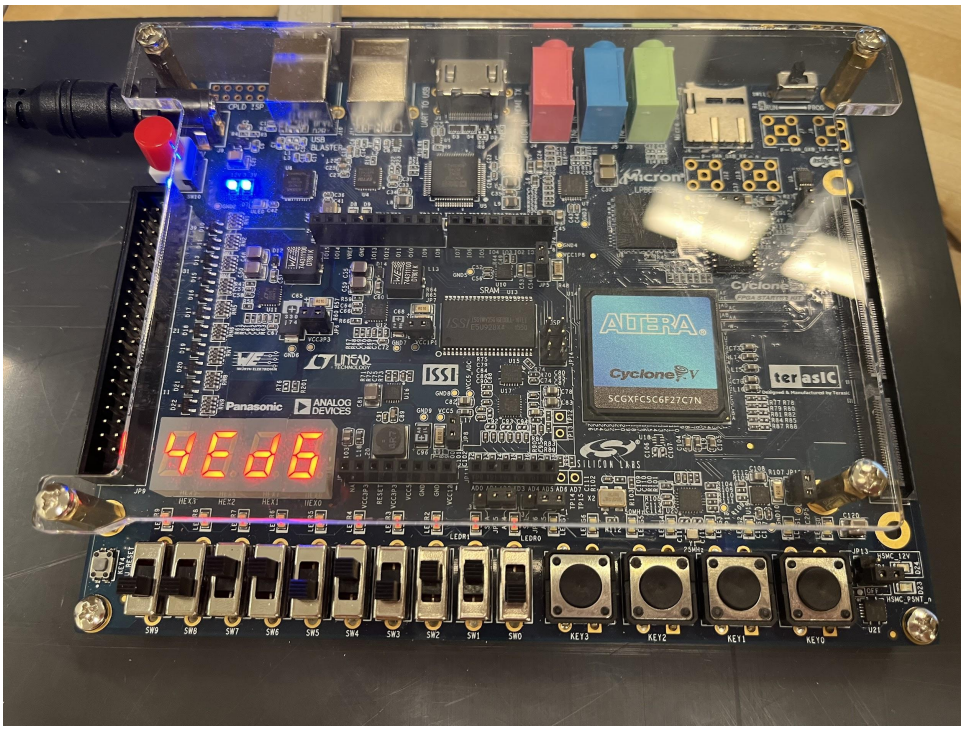

Figure 11: Testing the four-bit multiplier VHDL code from **Figure 9** and **Figure 10** with the input operands sent to HEX0 and HEX1, and the product of the operands sent to HEX2 and HEX3. Showing the case of the D x 6 = 4E.

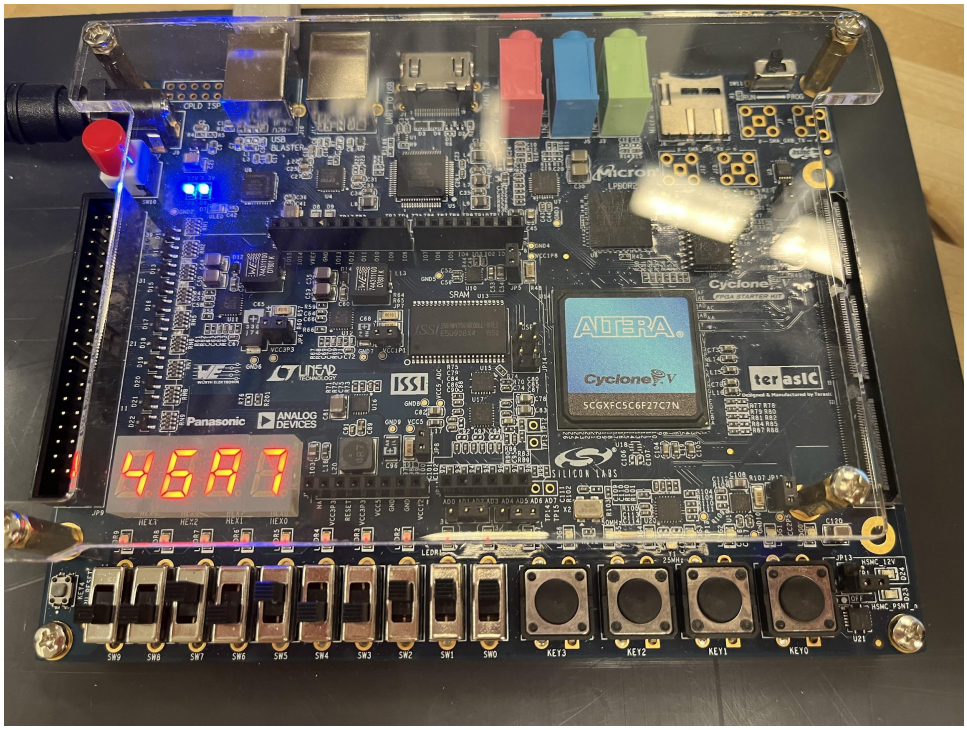

Figure 12: Testing the four-bit multiplier VHDL code from **Figure 9** and **Figure 10** with the input operands sent to HEX0 and HEX1, and the product of the operands sent to HEX2 and HEX3. Showing the case of the A x 7 = 46.

5. Post-lab

1. This design is not expandable at all. This is because it uses only 2 bits to store the current and destination floors. This results in a limit to the number of floors that the controller can support without a redesign to $2^2 = 4$. If four floors were to be encoded with only 2-bits, then the fail safe criteria would no longer be met. For example, with the following encoding, first - 00, second - 01, third - 10, forth - 11, then there wouldn't be a combination reserved for the fail safe case.
   Adding support for a 4th floor would require the expansion of the 2-bit current and destination floor inputs to 3-bit inputs. The floors could be encoded as follows: first - 001, second - 010, third - 011, and fourth - 100. The encoding 000 would be used as fail safe, as if any port becomes broken, all three bits would be pulled to 000. The other possible combinations 110 and 111, will not be used. If the inputs were ever 110 or 111, ENABLE would be set to 0. The logic for this circuit should be one that requires the least number of gates so that it can be more reliable. To do so, a truth table based on the required inputs and outputs can be generated, with X's (don't care values) placed in their respective positions. A K-map can then be used to simplify the boolean equations to ensure that the circuit uses the least amount of wires and gates so that the system is more reliable.

2. $log_2(50\,000\,000) = 25.57542476$
   50 MHz / 2^{25} = 1.49011611938, 50 MHz / 2^{26} = 0.74505805969. Depending on the use case, if the desired frequency is ≤ 1 Hz, then the 50 MHz clock should be divided 26 times. If the desired frequency is ≥ 1 Hz, then the 50 MHz clock should be divided 25 times.