ECE3270 Digital Computer Design - Lab 1

BCD CONVERSION AND ADDER

February 8, 2019

Aayahna Herbert
Clemson University
Department of Electrical and Computer Engineering
`aayahnh@g.clemson.ed`

Abstract

The goal of this lab was to design combinatorial circuits that can perform binary-to-decimal number conversion, specifically when adding. This design lab was split into four parts: (1) designing a circuit that converts a four-bit binary number into its five-bit hardware representation, (2) designing a decoder that accepts the five-bit hardware representation from Part 1 as an input and outputs the appropriate value to two 7-segment displays, (3) building a full adder entity to create a four-bit ripple carry adder circuit that takes in and adds two four-bit number and produces those two inputs as its output, and (4) design the main combinatorial circuit that adds two input BCD digits and a carry-in, outputs the two-digit BCD sum along with an error bit when the sum exceeds 19, and displays the results on two 7-segment displays. In this lab, all parts of the lab were designed and executed successfully on the board and through simulation done on ModelSim.

1 Introduction

While working in VHDL, the connection between coding for simulation and coding for the DE1-series board is an important one in terms of how to design the entities and the components. When designing each circuit from the four parts of the lab, the focus was on the entity and how it would be modified to be used on the DE1-series board instead of for ModelSim simulation. After assuring the translation from simulation to board will be possible through the entity setup, the architecture of the entity at hand is done in a way that is correct, efficient, and the most logical. The next sections will go into details about how each part of the lab was constructed and the reason it was done the way it was.

2 BCD

2.1 Design Method

The design for the BCD consisted of the entity and the behavioral architecture for the entity. The entity consisted of a four-bit number V = v(3)v(2)v(1)v(0) as the input and a five-bit number M = m(4)m(3)m(2)m(1)m(0) as the output. The purpose of this circuit is to take the binary number V and convert it to M, its two-digit equivalent in binary form. The number range the converter takes is 0 to 12; anything outside of this range should be unreadable.

Using the corresponding values of V and M from Figure 8.1, a behavioral architecture was made using case-when sequential statements learned in Chapter 6.6.7 from Brown [1] with V in the sensitivity list for the process. Because the lab deals with a five-bit output, a Karnaugh Map would be very tedious and a pain for the reader of the VHDL code to try to comprehend. Some type of sequential statement, I felt, would be the best way to describe the hardware since the value of M depended on V and every value is one-to-one so doing either an if-then-else or case statement would be the most logical and efficient way of coding the architecture.

2.2 Testing and Simulation

When it came to simulating the BCD design with a test bench, the test cases chosen, as shown in Figure 8.2, were meant to test the lower, middle, and higher numbers. The low numbers tested were 0, 1, and 2; the middle number tested was 9 - right before the fifth bit changes from a 0 over to a 1; and the higher numbers tested were 10, to see test the fifth bit changing to a 1, and 15, to make sure it reaches the max and an error doesn't occur. When simulated in ModelSim, the wave forms, as displayed in Figure 8.3, reflected the expected results from the test bench, making it successful.

3 Seven-Segment Decoder

3.1 Design Method

The design for the decoder consisted of the entity and the behavioral architecture for the entity. The entity consisted of a four-bit number c = c(3)c(2)c(1)c(0) as the input and two seven-bit numbers o0 = o0(6)o0(5)o0(4)o0(3)o0(2)o0(1)o0(0) and o1 = o1(6)o1(5)o1(4)o1(3)o1(2)o1(1)o1(0) as the outputs. The values of c for this lab will be the same values of M from Part 1 of the lab. Also, an important note when assigning the binary values to o0 and o1 is that seven-segment displays are active low, meaning 0's turn them on while 1's shut them off. The purpose of this circuit is to take the binary number c and display the number as a two-digit number on two 7-segment displays, o0 and o1. The number range the decoder takes is 0 to 15; anything outside of this range shouldn't be readable by the decoder and its display.

Using the corresponding values of M from Figure 8.1 for c, a behavioral architecture was made using case-when sequential statements with c in the sensitivity list for the process. Because Part 2 deals with displaying a certain number on the segment displays based off the value of C. A case-statement for each digit of the number to be displayed, saying when c is a certain binary value, then each display will show the digital representation of the digit by assigning the equivalent value of it to o0 and o1; Figure 8.4 shows how the binary numbers correspond to the display. Using two separate case-when sequential statements for each seven-segment display, I felt, would be the best way to describe the hardware since the binary value of the displays depend on c's binary value and every value is one-to-one so doing case statements would be the most logical and efficient way of coding the architecture.

3.2 Timing and Simulation

When it came to simulating the seven-segment decoder with a test bench, the test cases chosen, as shown in Figure 8.5, were meant to test the lower, middle, and higher numbers, as well as testing numbers outside of the circuit's range to make sure this and future errors would be caught correctly. The low numbers tested were 00 and 01; the middle numbers tested were 9 - right before the fifth bit changes from a 0 over to a 1 - and 10 to see test the fifth bit changing to a 1, and 15; the higher numbers tested were 11 and 15; and the numbers 011 and 16 were tested to make sure an error occurs and the circuit does not display the number since it's outside of the range. When simulated in ModelSim, the wave forms, as displayed in Figure 8.6, reflected the expected results from the test bench, making it successful.

4 Four-Bit Ripple Carry Adder

4.1 Design Method

Before the ripple carry adder could be designed, a full carry adder was designed due to a ripple carry adder being comprised of four full adders - as seen in Figure 8.7. The full adder's entity had ci, a, and b as the input and s and co as its output. Using the truth table provided in Figure 8.7, the data-flow architecture for the full adder provided the SOP equations for s and co.

The design for the four-bit ripple carry adder consisted of the entity and the structure architecture for the entity. The entity is made up of a four-bit number a = a(3)a(2)a(1)a(0), another four-bit number b = b(3)b(2)b(1)b(0), and a carry in value cin as the inputs and a four-bit number s = s(4)s(3)s(2)s(1)s(0) and a carry out value cout as the outputs. The purpose of this circuit is to add inputs a, b, and cin together and show their sum through cout and s, mimicking a single, five-bit sum. The number range the converter takes is 0 to 31 because the highest number that can be reached from four-bits is 15 plus a high carry in; anything outside of this range should be unreadable.

A structural architecture had the full adder component, three signals c1, c2, and c3, and four instances of the full adder with their port maps corresponding to their positions in the full adder. Because the ripple carry has four full adders, it would be logical and efficient to structure the architecture as a component and adding instances to it.

4.2 Testing and Simulation

When it came to simulating the ripple carry adder with a test bench, the test cases chosen, as shown in Figure 8.8, were meant to test the lower, middle, and higher numbers, as well as testing numbers closer to the adder's range to make sure the carry out functioned correctly. The low number tested was 3+2+0=5; the middle numbers tested were 5+6+0=11 and 8+7+0=15; and the higher number tested was 15+15+0=30. When simulated in ModelSim, the wave forms, as displayed in Figure 8.9, reflected the expected results from the test bench, making it successful.

5 BCD Converter and Adder

5.1 Design Method

The design for the BCD converter and adder consisted of the entity and the structural and behavioral mixed architecture for the entity. The entity consisted of two four-bit BCD numbers, A4 = A4(3)A4(2)A4(1)A4(0) and B4 = B4(3)B4(2)B4(1)B4(0), and a carry-in value, CIN1 as the inputs, a five-bit number S-Cout = SC(4)SC(3)SC(2)SC(1)SC(0) as a buffer, and an error value, ERROR, and six, seven-bit values S0, S1, A0, A1, B0, and B0, as the outputs. The purpose of this circuit is to add two BCD digits, A4 and B4, plus a carry-in, CIN1, take their sum, S-Cout, and display it, along with the values of A4 and B4, on 7-segment displays, done using S1 and S0, A1 and A0, and B1 and B0.

The number range of the converter and adder is 0 to 19; anything outside of this range should be unreadable and indicated by setting the error bit, ERROR, to '1'.

A behavioral and structural mixed architecture was made using the components from the four-bit ripple carry adder, BCD converter, and 7-segment decoder; one instance of the ripple carry adder and the BCD converter, and three instances of the decoder; and a behavioral process for the error-bit. The ripple carry adder instance maps A4, B4, CIN1, and S-Cout in order to add the three inputs and output the sum plus the carry-out value. The BCD instance maps the output S-Cout as the input (hence why it was defined as a buffer) to convert the sum into a two-digit decimal number stored in the signal m-out. The first instance of the decoder maps the signal m-out as its input in order to show the digital number on two 7-segment displays; the other two instances map A4 and B4 as its inputs in order to display those on the 7-segment displays, as well. The last part of the architecture is the behavioral side of it, using a process to set up an if-then-else statement to set the error-bit to high when the sum of the converter and adder is beyond its range.

5.2 Testing and Simulation

When it came to simulating the entire BCD converter and adder with a test bench, the test cases chosen, as shown in Figure 8.10, were meant to test the lower, middle, and higher numbers, as well as testing numbers close to and outside of the converter and adder's range to make sure the error-bit functioned correctly. The low number tested was 1+3+1=5; the middle number tested was 6+2+1=9; the higher number tested was 11+7+1=19 - these numbers for A4 and B4 were chosen to make sure that values greater than 9 were not displayed on their corresponding 7-segment displays. When simulated in ModelSim, the wave forms, as displayed in Figure 8.11, reflected the expected results from the test bench, making it successful.

6 Results

When all four parts of the lab were modified to be tested on the Altera DE1-SoC Board using guidance from Smith [2], they compiled and functioned correctly, getting the same results found in all of their separate ModelSim test waves. The common issue I ran into - outside of simple syntax issues - when working on the lab parts was the ModelSim not being able to compile as a result of me not initializing each input variable used for each entity; by simply setting values and going back in to compile correctly, the simulations compiled and tested the parts correctly. When the final part was tested on the board with all of the other parts combined with it, it displayed the values A4 and B4 and their sum S-Cout on two 7-segment displays each, the sum, carry-out, and error-bit through the LED lights, and the switches are used to control A4, B4, and the carry-in value. The 7-segment displays zero out or turn off when either A4 or B4 exceed 9 or when the sum exceeds 19.

7 Conclusion

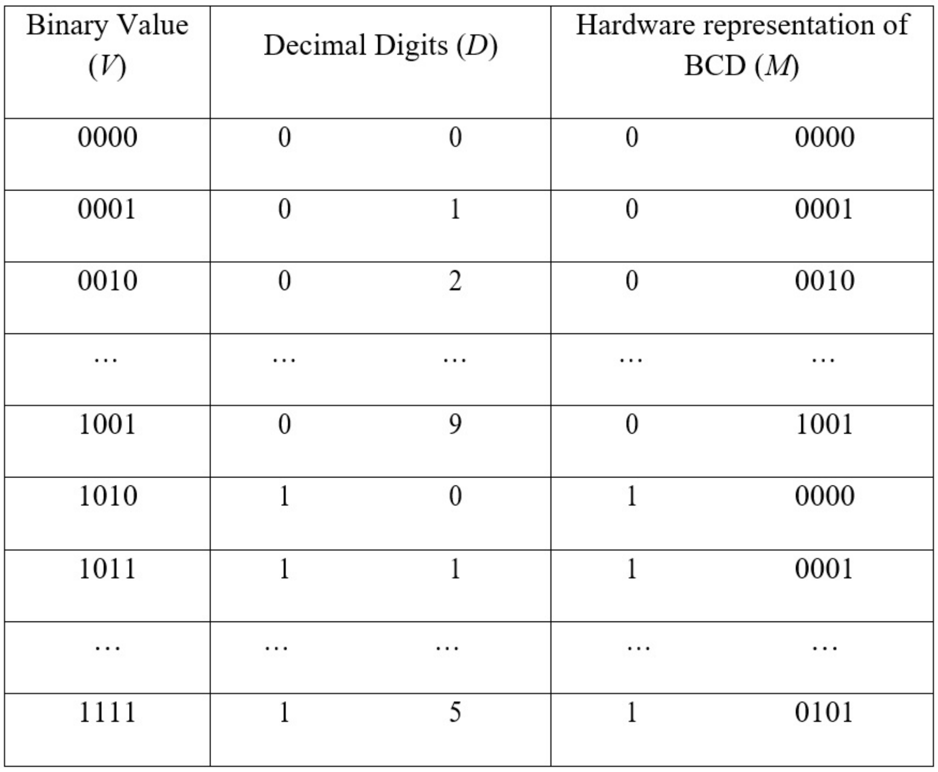
This lab has given me the chance to test and apply my knowledge about VHDL coding, behavioral and entity elements and techniques, learned from class lectures. By doing this lab, I have a clearer understanding on how each of the components made in this lab work and what they all need in order to do their jobs correctly. If I could go back and change how things were done throughout the lab, I would add more test bench cases, especially for both possible values for the carry-in value for the parts dealing the ripple carry adder. I would probably also go back and modify the case statements used for the 7-segment display's architecture by using an if-then-else statement for the first digit since it can only be a '1' or a '0'. Overall, this was a great lab to get used to with VHDL coding and how to go about analyzing the specifications to make the most efficient code for the hardware so it's easily comprehensible.

References

[1] S. Brown and Z. Vranesic. Fundamental of digital logic with vhdl design. Textbook, New York, USA, 2009.

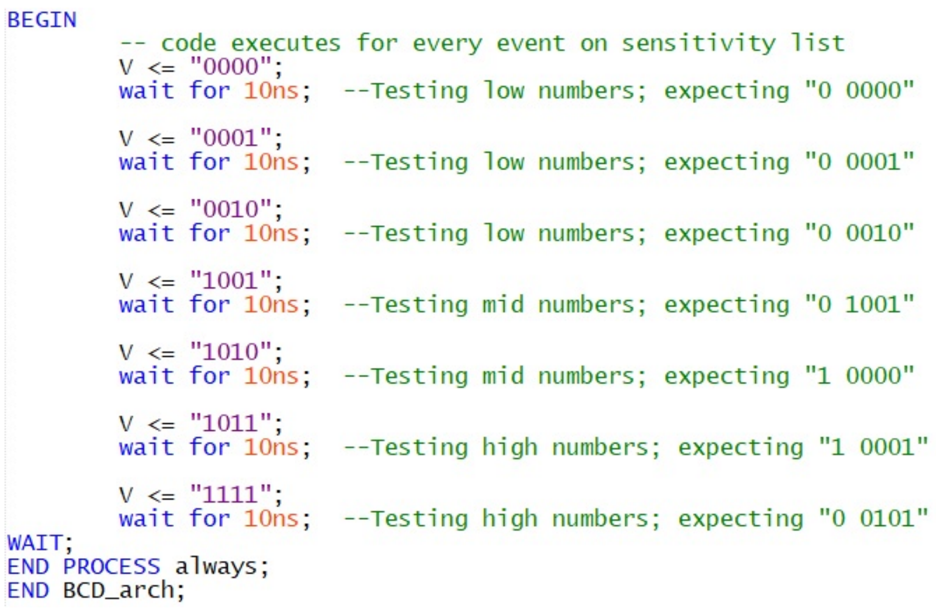
[2] M. Smith. Digital computer design lab manual. Manual, South Carolina, USA, 2019.

8 Appendix



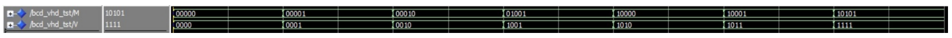
Binary Value (V)	Decimal Digits (D)		Hardware representation of BCD (M)	
0000	0	0	0	0000
0001	0	1	0	0001
0010	0	2	0	0010
…	…	…	…	…
1001	0	9	0	1001
1010	1	0	1	0000
1011	1	1	1	0001
…	…	…	…	…
1111	1	5	1	0101

Figure 8.1: BCD Input and Output Values



```
BEGIN
        -- code executes for every event on sensitivity list
        V <= "0000";
        wait for 10ns;  --Testing low numbers; expecting "0 0000"

        V <= "0001";
        wait for 10ns;  --Testing low numbers; expecting "0 0001"

        V <= "0010";
        wait for 10ns;  --Testing low numbers; expecting "0 0010"

        V <= "1001";
        wait for 10ns;  --Testing mid numbers; expecting "0 1001"

        V <= "1010";
        wait for 10ns;  --Testing mid numbers; expecting "1 0000"

        V <= "1011";
        wait for 10ns;  --Testing high numbers; expecting "1 0001"

        V <= "1111";
        wait for 10ns;  --Testing high numbers; expecting "0 0101"
WAIT;
END PROCESS always;
END BCD_arch;
```

Figure 8.2: BCD Test Bench Cases



```
/bcd_vhd_tst/M  10101  00000  00001  00010  01001  10000  10001  10101
/bcd_vhd_tst/V  1111   0000   0001   0010   1001   1010   1011   1111
```

Figure 8.3: BCD Simulated Test Waves

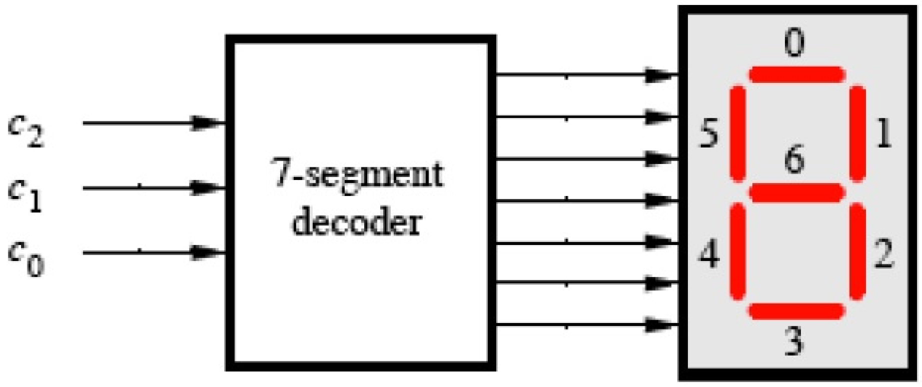
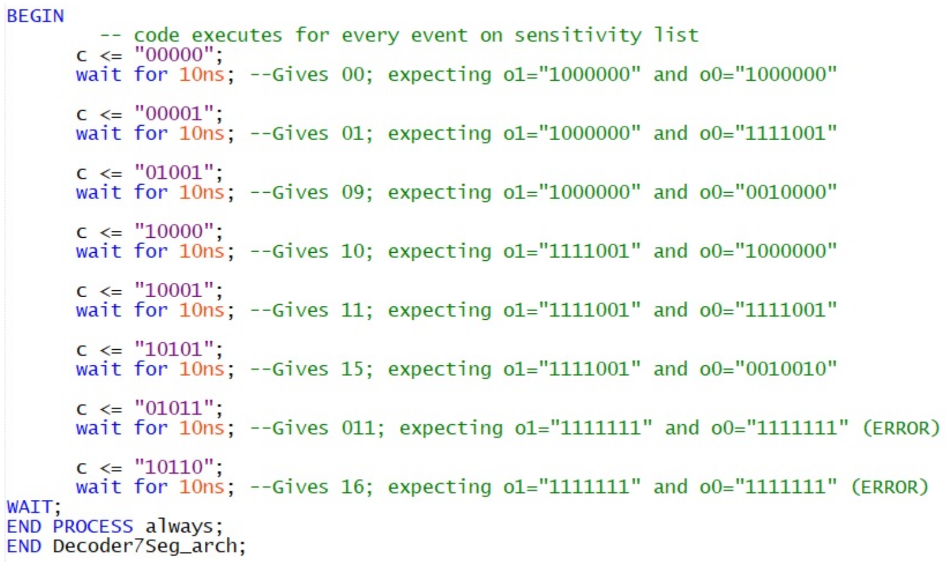
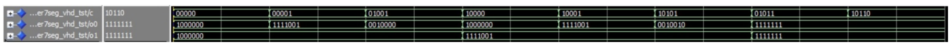


Figure 8.4: Seven-Segment Decoder and Display Figure



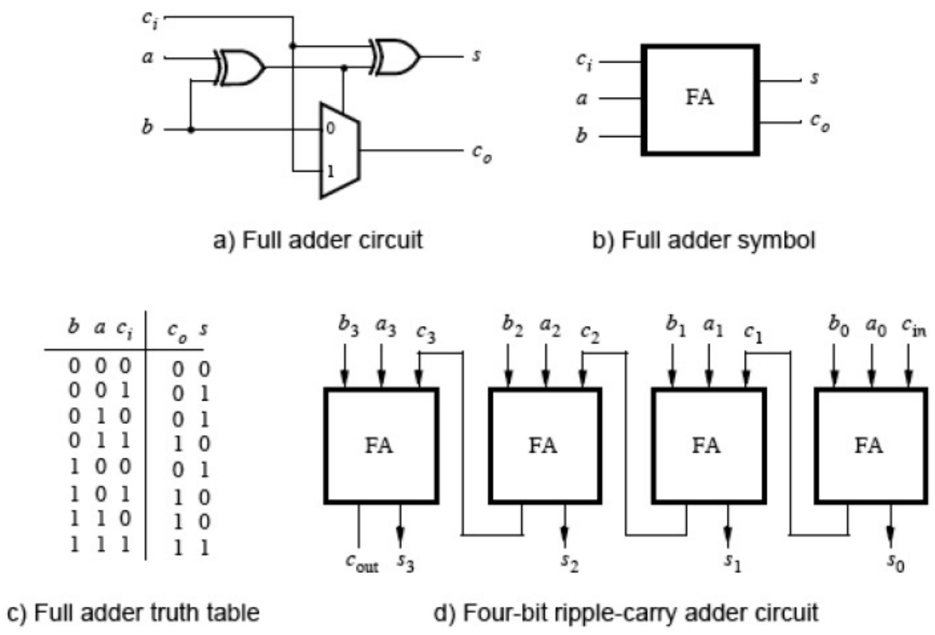
```
BEGIN
        -- code executes for every event on sensitivity list
      c <= "00000";
      wait for 10ns; --Gives 00; expecting o1="1000000" and o0="1000000"

      c <= "00001";
      wait for 10ns; --Gives 01; expecting o1="1000000" and o0="1111001"

      c <= "01001";
      wait for 10ns; --Gives 09; expecting o1="1000000" and o0="0010000"

      c <= "10000";
      wait for 10ns; --Gives 10; expecting o1="1111001" and o0="1000000"

      c <= "10001";
      wait for 10ns; --Gives 11; expecting o1="1111001" and o0="1111001"

      c <= "10101";
      wait for 10ns; --Gives 15; expecting o1="1111001" and o0="0010010"

      c <= "01011";
      wait for 10ns; --Gives 011; expecting o1="1111111" and o0="1111111" (ERROR)

      c <= "10110";
      wait for 10ns; --Gives 16; expecting o1="1111111" and o0="1111111" (ERROR)
WAIT;
END PROCESS always;
END Decoder7Seg_arch;
```

Figure 8.5: Seven-Segment Decoder Test Bench Cases



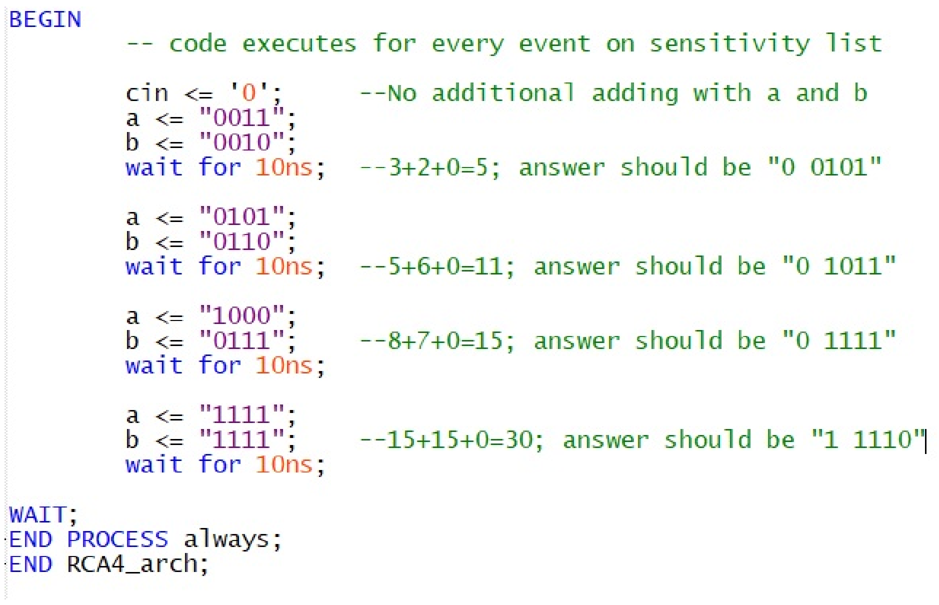
```
...er7seg_vhd_tst/c   10110    00000    00001    01001    10000    10001    10101    01011    10110
...er7seg_vhd_tst/o0  1111111  1000000  1111001  0010000  1000000  1111001  0010010  1111111
...er7seg_vhd_tst/o1  1111111  1000000                    1111001                    1111111
```

Figure 8.6: Seven-Segment Decoder Simulated Test Waves



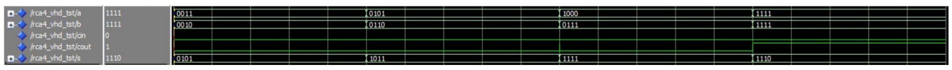
b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 8.7: Full Adder and Ripple Carry Adder Architectural Diagrams



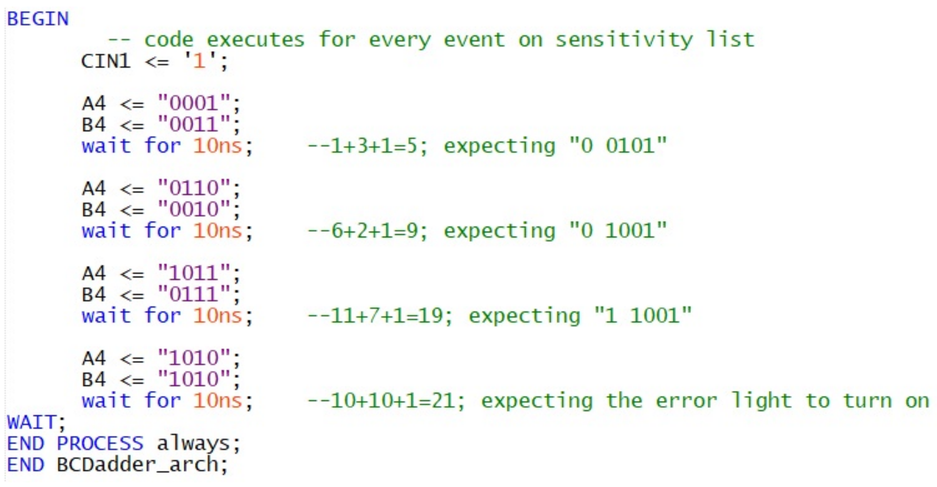
```
BEGIN
        -- code executes for every event on sensitivity list

        cin <= '0';     --No additional adding with a and b
        a <= "0011";
        b <= "0010";
        wait for 10ns;  --3+2+0=5; answer should be "0 0101"

        a <= "0101";
        b <= "0110";
        wait for 10ns;  --5+6+0=11; answer should be "0 1011"

        a <= "1000";
        b <= "0111";    --8+7+0=15; answer should be "0 1111"
        wait for 10ns;

        a <= "1111";
        b <= "1111";    --15+15+0=30; answer should be "1 1110"
        wait for 10ns;

WAIT;
END PROCESS always;
END RCA4_arch;
```

Figure 8.8: 4-Bit Ripple Carry Adder Test Bench Cases



/rca4_vhd_tst/a 1111 0011 0101 1000 1111
/rca4_vhd_tst/b 1111 0010 0110 0111 1111
/rca4_vhd_tst/cin 0
/rca4_vhd_tst/cout 1
/rca4_vhd_tst/s 1110 0101 1011 1111 1110

Figure 8.9: 4-Bit Ripple Carry Adder Simulated Test Waves



```
BEGIN
        -- code executes for every event on sensitivity list
      CIN1 <= '1';

      A4 <= "0001";
      B4 <= "0011";
      wait for 10ns;    --1+3+1=5; expecting "0 0101"

      A4 <= "0110";
      B4 <= "0010";
      wait for 10ns;    --6+2+1=9; expecting "0 1001"

      A4 <= "1011";
      B4 <= "0111";
      wait for 10ns;    --11+7+1=19; expecting "1 1001"

      A4 <= "1010";
      B4 <= "1010";
      wait for 10ns;    --10+10+1=21; expecting the error light to turn on
WAIT;
END PROCESS always;
END BCDadder_arch;
```

Figure 8.10: BCD Conversion and Adder Test Bench Cases

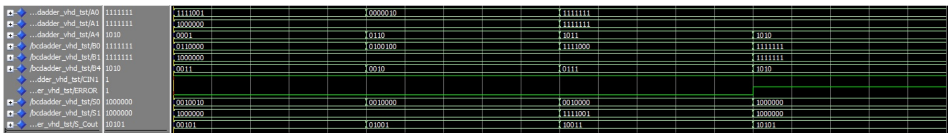

...dadder_vhd_tst/A0 1111111
...dadder_vhd_tst/A1 1111111
...dadder_vhd_tst/A4 1010
/bcdadder_vhd_tst/B0 1111111
/bcdadder_vhd_tst/B1 1111111
/bcdadder_vhd_tst/B4 1010
...dder_vhd_tst/CIN1 1
...er_vhd_tst/ERROR 1
/bcdadder_vhd_tst/S0 1000000
/bcdadder_vhd_tst/S1 1000000
...er_vhd_tst/S_Cout 10101
1111001
0000010
1111111
1000000
1111111
0001
0110
1011
1010
0110000
0100100
1111000
1111111
1000000
1111111
0011
0010
0111
1010
0010010
0010000
0010000
1000000
1000000
1111001
1000000
00101
01001
10011
10101

Figure 8.11: BCD Conversion and Adder Simulated Test Waves