

VHDL LAB #3

Rachel Ruddy - 261162987

Natasha Lawford - 261178596

Rachel Bunsick - 261159677

Executive Summary:

In this assignment, we created structural VHDL descriptions of both half and full adders, as well as 2 corresponding testbench files to test their functionality. Then, we implemented our half and full adders to create a 4-bit ripple carry adder as well as a one digit BCD adder. Structural and behavioral VHDL descriptions were written for both the RCA and BCD adders as well as exhaustive testbenches for each description.

VHDL Code and Explanations:

Structural Description of Half-Adder:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity half_adder is
5  port (a: in std_logic;
6       b: in std_logic;
7       s: out std_logic;
8       c: out std_logic);
9  end half_adder;
10
11 architecture structural of half_adder is
12 begin
13     --sum is equal to the xor of inputs a and b:
14     s <= a xor b;
15     --carry is equal to the and of a and b:
16     c <= a and b;
17 end structural;
```

The structural description of the half adder maps the inputs and outputs of the circuit to their corresponding gates which produce the expected output. The half adder has two 1 bit inputs: a and b, as well as two outputs: s (sum), and c (carry). A and b are connected to 1 xor gate which produces the output s. Additionally, a and b are connected to one and gate which produces the output c.

Half-Adder Testbench:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity half_adder_tst is
6  end half_adder_tst;
7
8  architecture Testbench of half_adder_tst is
9  component half_adder is
10     Port ( a : in std_logic;
11           b : in std_logic;
12           s : out std_logic;
13           c : out std_logic);
14  end component;
15
16     signal a, b : std_logic := '0';
17     signal s_structural, c_structural : std_logic;
18
19  begin
20     i1 : entity work.half_adder
21         port map (a => a, b => b, s => s_structural, c => c_structural);
22
23  process
24  begin
25     --test case 1: expect sum=0, carry=0
26     a <= '0'; b <= '0';
27     wait for 10 ns;
28
29     --test case 2: expect sum=1, carry=0
30     a <= '1'; b <= '0';
31     wait for 10 ns;
32
33     --test case 3: expect sum=1, carry=0
34     a <= '0'; b <= '1';
35     wait for 10 ns;
36
37     --test case 4: expect sum=0, carry=1
38     a <= '1'; b <= '1';
39     wait for 10 ns;
40
41     wait;
42  end process;
43
44  end Testbench;
```

Since we have 2 input bits, a and b, we exhaustively test all the combinations of cases since there are only 4. We list the expected values of s and c in the comments above each block to check the functionality.

Structural Description of Full-Adder:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity full_adder is
5  Port (a: in std_logic;
6       b: in std_logic;
7       c_in: in std_logic;
8       s: out std_logic;
9       c_out: out std_logic);
10 end full_adder;
11
12 architecture structural of full_adder is
13 begin
14     --sum is equal to the xor of inputs a and b and c_in:
15     s <= a xor b xor c_in;
16
17     --carry is equal to (a and b) or (c_in and a_xor_b):
18     c_out <= (a and b) or (c_in and (a xor b));
19
20 end structural;
```

The structural description of the full adder maps the inputs a, b, and c_in (carry in) to the gates which execute the desired functionality. The output s (sum) is the output of a 3 input xor gate which takes in all three inputs (a, b, c_in). The output c_out (carry out) comes from taking the or of a and b together as well as c_in and (a xor b).

Full-Adder Testbench:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity full_adder_tst is
6  end full_adder_tst ;
7
8  architecture Testbench of full_adder_tst is
9      component full_adder is
10         port (
11             a : in std_logic;
12             b : in std_logic;
13             c_in : in std_logic;
14             s : out std_logic;
15             c_out : out std_logic);
16         end component;
17
18         signal a, b, c_in : std_logic := '0';
19         signal s_structural, c_out_structural : std_logic;
20
21     begin
22         i1 : full_adder
23             port map (a => a, b => b, c_in => c_in, s => s_structural, c_out => c_out_structural);
24
25     process
26     begin
27         --test case 1: expect s=0, c_out=0
28         a <= '0'; b <= '0'; c_in <= '0';
29         wait for 10 ns;
30
31         --test case 2: expect s=1, c_out=0
32         a <= '1'; b <= '0'; c_in <= '0';
33         wait for 10 ns;
34
35         --test case 3: expect s=1, c_out=0
36         a <= '0'; b <= '1'; c_in <= '0';
37         wait for 10 ns;
38
39         --test case 4: expect s=0, c_out=1
40         a <= '1'; b <= '1'; c_in <= '0';
41         wait for 10 ns;
42
43         --test case 5: expect s=1, c_out=0
44         a <= '0'; b <= '0'; c_in <= '1';
45         wait for 10 ns;
46
47         --test case 6: s=0, c_out=1
48         a <= '1'; b <= '0'; c_in <= '1';
49         wait for 10 ns;
50
51         --test case 7: expect s=0, c_out=1
52         a <= '0'; b <= '1'; c_in <= '1';
53         wait for 10 ns;
54
55         --test case 8: expect s=1, c_out=1
56         a <= '1'; b <= '1'; c_in <= '1';
57         wait for 10 ns;
58
59         wait;
60     end process;
61 end Testbench;

```

The full adder also explicitly tests the 8 different cases for the variation of the 3 inputs (a,b,c_in). The expected outputs for s and c_out are put in the comments to confirm that the full adder is working as expected for each test case.

Structural Description of 4-bit Ripple-Carry Adder:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity rca_structural is
5  port ( A: in std_logic_vector (3 downto 0);
6        B: in std_logic_vector (3 downto 0);
7        S: out std_logic_vector (4 downto 0));
8  end rca_structural;
9
10 architecture structural of rca_structural is
11     signal carry : std_logic_vector(2 downto 0);
12     begin
13         --start with half adder
14         HA_LSB: entity work.half_adder
15         port map(a => A(0),
16                 b => B(0),
17                 s => S(0),
18                 c => carry(0));
19
20         FA_1: entity work.full_adder
21         port map(a => A(1),
22                 b => B(1),
23                 c_in => carry(0),
24                 s => S(1),
25                 c_out => carry(1));
26
27         FA_2: entity work.full_adder
28         port map(a => A(2),
29                 b => B(2),
30                 c_in => carry(1),
31                 s => S(2),
32                 c_out => carry(2));
33
34         FA_MSB: entity work.full_adder
35         port map(a => A(3),
36                 b => B(3),
37                 c_in => carry(2),
38                 s => S(3),
39                 c_out => S(4)); --Last carry is the MSB of the sum
40
41     end architecture structural;
```

The structural description of the 4-bit rca instantiates 1 half adder for adding the least significant bits since there is no carry in, as well as 3 full adders which “ripple” by using the previous c_out as the c_in for the following full adder. For the last full adder, the carry out is assigned to be the MSB of the final sum. The rest of the bits of the sum are simply represented by the sum from each respective full or half adder.

Behavioral Description of 4-bit Ripple-Carry Adder:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity rca_behavioral is
6  port (A: in std_logic_vector (3 downto 0);
7        B: in std_logic_vector (3 downto 0);
8        S: out std_logic_vector (4 downto 0));
9  end rca_behavioral;
10
11 architecture behavioral of rca_behavioral is
12     begin
13         process(A,B)
14         begin
15             S <= std_logic_vector(unsigned('0' & A) + unsigned('0' & B));
16         end process;
17     end behavioral;
```

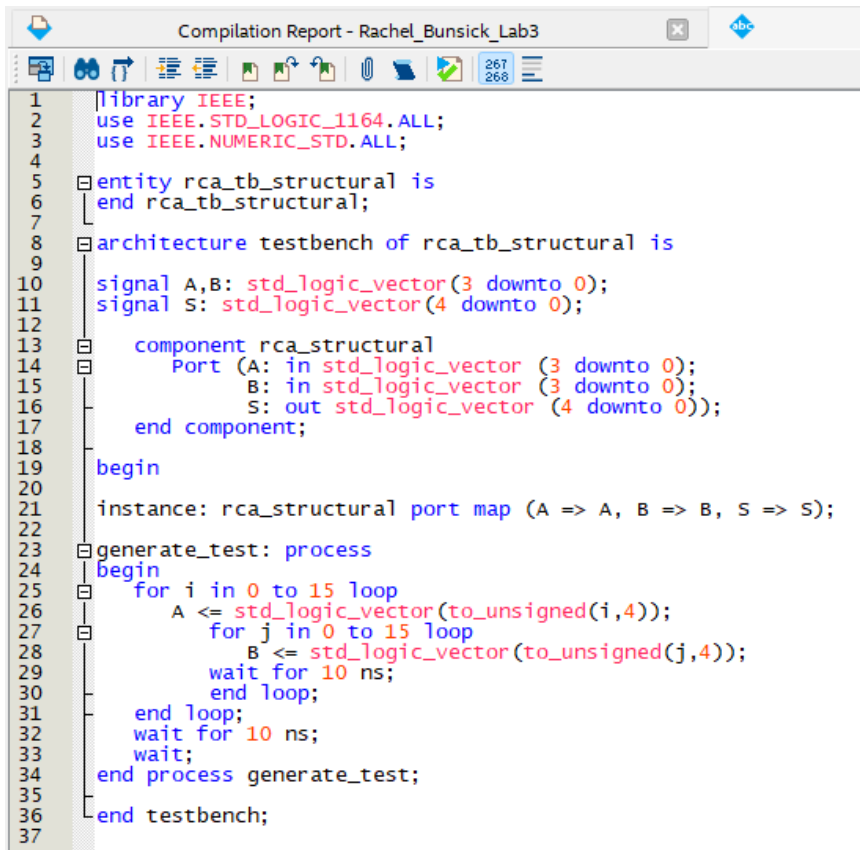
The behavioral description of the 4-bit rca appends a 0 to the front of both a and b to change their size from 4 bit to 5 bit. Then, we make use of the addition operator in VHDL to add the two 5-bit numbers together. Then, we cast the result of this addition to `std_logic_vector` as this is what we have declared the type of the output to be in the entity declaration.

RCA Behavioral Testbench:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity rca_tb is
6  end rca_tb;
7
8  architecture testbench of rca_tb is
9
10 signal A,B: std_logic_vector(3 downto 0);
11 signal S: std_logic_vector(4 downto 0);
12
13 component rca_behavioral
14   port (A: in std_logic_vector (3 downto 0);
15         B: in std_logic_vector (3 downto 0);
16         S: out std_logic_vector (4 downto 0));
17 end component;
18
19 begin
20
21 instance: rca_behavioral port map (A => A, B => B, S => S);
22
23 generate_test: process
24 begin
25   for i in 0 to 15 loop
26     A <= std_logic_vector(to_unsigned(i,4));
27     for j in 0 to 15 loop
28       B <= std_logic_vector(to_unsigned(j,4));
29       wait for 10 ns;
30     end loop;
31   end loop;
32   wait for 10 ns;
33   wait;
34 end process generate_test;
35
36 end testbench;
37
38
```

The rca behavioral testbench instantiates a component using the behavioral description written for the rca. Then, the testbench exhaustively iterates through all the possible cases by changing bit of b for some specific a. Then, a is changed by one bit and the process repeats.

RCA Structural Testbench:



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity rca_tb_structural is
6  |end rca_tb_structural;
7
8  architecture testbench of rca_tb_structural is
9
10     signal A,B: std_logic_vector(3 downto 0);
11     signal S: std_logic_vector(4 downto 0);
12
13     component rca_structural
14     | Port (A: in std_logic_vector (3 downto 0);
15           | B: in std_logic_vector (3 downto 0);
16           | S: out std_logic_vector (4 downto 0));
17     |end component;
18
19     begin
20
21     instance: rca_structural port map (A => A, B => B, S => S);
22
23     generate_test: process
24     |begin
25     |   for i in 0 to 15 loop
26     |   |   A <= std_logic_vector(to_unsigned(i,4));
27     |   |   for j in 0 to 15 loop
28     |   |   |   B <= std_logic_vector(to_unsigned(j,4));
29     |   |   |   wait for 10 ns;
30     |   |   |   end loop;
31     |   |   end loop;
32     |   |   wait for 10 ns;
33     |   |   wait;
34     |   end process generate_test;
35
36     end testbench;
37
```

The rca structural testbench instantiates a component using the structural description written for the rca. Then in the same way as the behavioral rca testbench, all the possible cases are tested by changing bit of b for some specific a. Then, a is changed by one bit and the process repeats.

One-Digit Binary Coded Decimal (BCD) Adder

Structural:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity bcd_adder_structural is
6  port (
7      A : in std_logic_vector (3 downto 0);
8      B : in std_logic_vector (3 downto 0);
9      S : out std_logic_vector (3 downto 0); -- Sum output in BCD
10     C : out std_logic; --carry if result > 9
11 end bcd_adder_structural;
12
13 architecture structural of bcd_adder_structural is
14     -- component declaration
15     component rca_structural
16     port(
17         A : in std_logic_vector (3 downto 0);
18         B : in std_logic_vector (3 downto 0);
19         S : out std_logic_vector (4 downto 0));
20     end component;
21
22     --signal declaration
23     signal temp_sum : std_logic_vector(4 downto 0);
24     signal temp_carry : std_logic;
25     signal temp_BCD : std_logic_vector(3 downto 0);
26     signal final_BCD : std_logic_vector(4 downto 0);
27 begin
28     u1 : rca_structural
29     port map(A, B, temp_sum);
30     temp_carry <= (temp_sum(4)) OR (temp_sum(3) AND temp_sum(2)) OR (temp_sum(3) AND temp_sum(1));
31     temp_BCD(3) <= '0';
32     temp_BCD(0) <= '0';
33     temp_BCD(2) <= temp_carry;
34     temp_BCD(1) <= temp_carry;
35     u2 : rca_structural
36     port map(temp_BCD, temp_sum(3 downto 0), final_BCD);
37     C <= temp_carry;
38     S <= final_BCD(3 downto 0);
39 end structural;
```

The BCD adder takes two BCD digits, A and B, which are 4 bits of type `std_logic_vector`. The adder outputs S, the sum of the two digits, where S is a BCD representation of a number 0-9. If $A+B > 9$, the decimal representation of this sum is 1X where X is a digit between 0 and 8 (since the largest one digit BCD addition we can do is 9+9). In this case, the C output, of type `std_logic`, is '1', and S is the LSB 'X' in binary representation. If the sum is less than 10, C is '0'.

The structural representation uses the RCA component we previously created. First, A and B are added with the RCA. This addition (`temp_sum`) is not necessarily in BCD format though, since $A+B$ might be greater than 9. To account for this, we can use a truth table/K-map to find the cases in which $\text{temp_sum} > 9$, which indicates that the output C would be '1'. If $\text{temp_sum} > 9$, we add "0110" to `temp_sum`, and if $\text{temp_sum} < 10$, we add "0000". So, we add "0C_{out}C_{out}0" to `temp_sum`, where C_{out} is, according to our K-map, $(\text{temp_sum}(4) \text{ OR } (\text{temp_sum}(3) \text{ AND } \text{temp_sum}(2)) \text{ OR } (\text{temp_sum}(3) \text{ AND } \text{temp_sum}(1)))$. Finally, this addition will set the sum back into BCD format, and we assign C to the value of C_{out} to account for if $A+B > 9$.

Structural BCD Adder Testbench:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity tb_bcd_adder_structural is
6  |end tb_bcd_adder_structural;
7
8  architecture arch of tb_bcd_adder_structural is
9
10     -- component declaration
11     component bcd_adder_structural
12     |
13     port(
14         A : in std_logic_vector (3 downto 0);
15         B : in std_logic_vector (3 downto 0);
16         S : out std_logic_vector (3 downto 0); -- sum output
17         C : out std_logic); -- carry output if result > 9
18     end component;
19
20     -- signal declarations
21     signal A : std_logic_vector (3 downto 0);
22     signal B : std_logic_vector (3 downto 0);
23     signal S : std_logic_vector (3 downto 0);
24     signal C : std_logic;
25
26     begin
27         -- instantiate the test unit
28         u1 : bcd_adder_structural
29             port map (A, B, S, C);
30
31         -- test cases
32         process
33         |begin
34         |    for i in 0 to 9 loop
35         |    |    for j in 0 to 9 loop
36         |    |    |    A <= std_logic_vector(to_unsigned(i,4));
37         |    |    |    B <= std_logic_vector(to_unsigned(j,4));
38         |    |    |    wait for 10 ns;
39         |    |    end loop;
40         |    end loop;
41
42         |    wait;
43         |end process;
44     end arch;
45
```

The structural BCD adder testbench instantiates a component using the structural description of the BCD adder. Then, it runs through every case of the addition of two one-digit BCD numbers. That is all cases of addition where A and B can be decimal numbers 0-9 inclusive. We use a nested for loop to achieve an exhaustive test of all possible additions.

Behavioral BCD Adder:

```
Compilation Report - Rachel_Bunsick_Lab3      bcd_adder_behavioral.vhd
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity bcd_adder_behavioral is
6  port ( A : in std_logic_vector (3 downto 0);
7        B : in std_logic_vector (3 downto 0);
8        S : out std_logic_vector (3 downto 0);
9        C : out std_logic);
10 end bcd_adder_behavioral;
11
12 architecture behavioral of bcd_adder_behavioral is
13     signal temp_sum : unsigned (4 downto 0);
14     signal adjusted_sum : unsigned (4 downto 0);
15
16 begin
17     process(A, B)
18     variable sum : unsigned(4 downto 0);
19     variable A_unsigned : unsigned(3 downto 0);
20     variable B_unsigned : unsigned(3 downto 0);
21     variable S_unsigned : unsigned(3 downto 0);
22     begin
23         A_unsigned := unsigned(A);
24         B_unsigned := unsigned(B);
25         S_unsigned := "0000";
26         sum := ('0' & A_unsigned) + ('0' & B_unsigned);
27         if (sum > 9) then
28             S_unsigned := resize((sum + "00110"),4);
29             C <= '1';
30         else
31             S_unsigned := sum(3 downto 0);
32             C <= '0';
33         end if;
34         S <= std_logic_vector(S_unsigned);
35         -- add the two bcd numbers
36         temp_sum <= ('0' & unsigned(A)) + ('0' & unsigned(B));
37         report "A: " & std_logic_vector(A) & " B: " & std_logic_vector(B) & " Temp Sum: " & std_logic_vector(temp_sum);
38         -- check if sum is greater than 9
39         if (temp_sum > "1001") then
40             -- add 0110 to adjust
41             adjusted_sum <= temp_sum + "0110";
42             C <= '1';
43         else
44             adjusted_sum <= temp_sum;
45             C <= '0';
46         end if;
47         S <= std_logic_vector(adjusted_sum(3 downto 0));
48     end process;
49 end behavioral;
```

The behavioral BCD adder sums the 5-bit versions of A and B (concatenates '0' as the MSB of A and B respectively). It also initializes the output sum to be "0000". If the sum of A and B is greater than 9, then we resize the output sum to be 5-bits and we take the sum of A and B and add 6 to it to properly account for the carry and we set the carry out to be 1. If the sum is less than 9, then we set the output sum to be A + B and set the carry out to be 0.

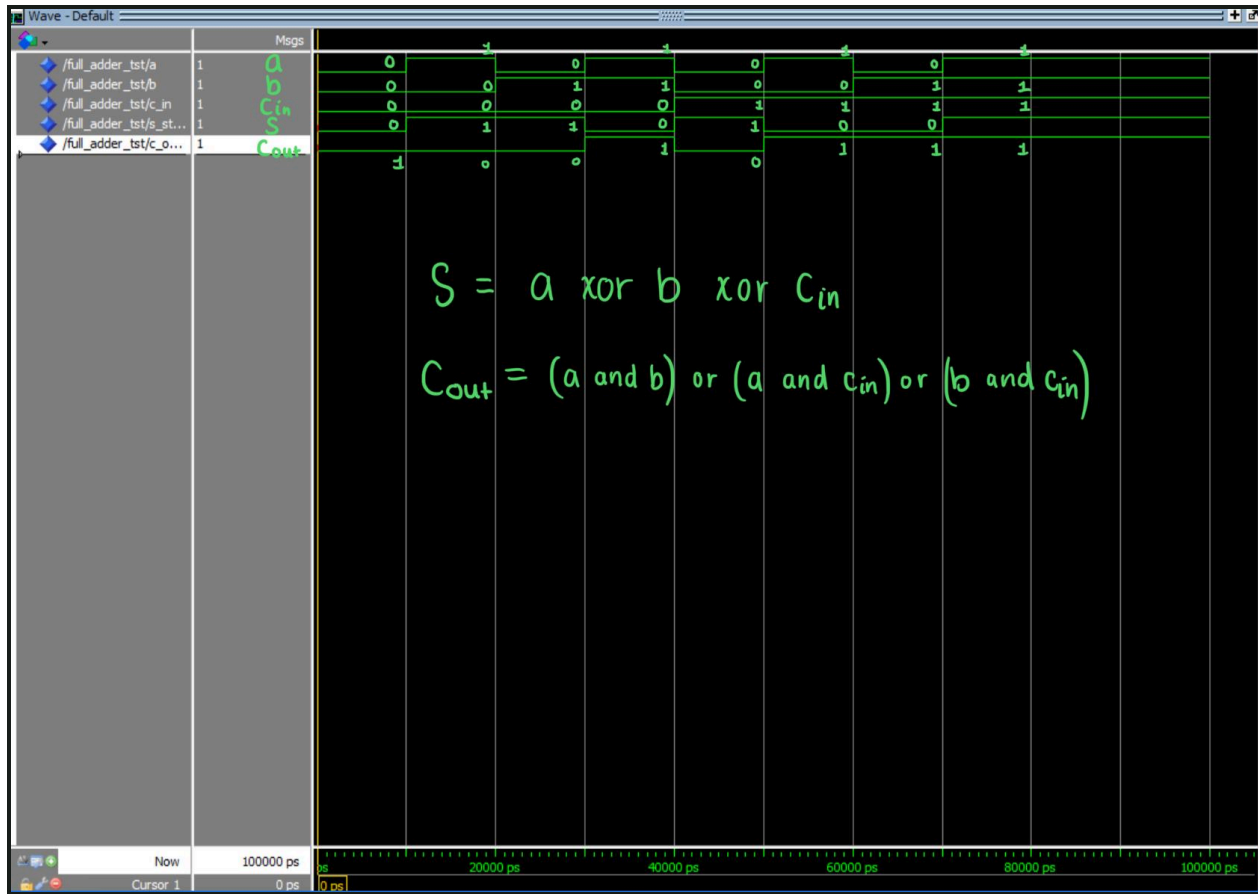
Behavioral BCD Adder Testbench:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity tb_bcd_adder_behavioral is
6  | end tb_bcd_adder_behavioral;
7
8  architecture arch of tb_bcd_adder_behavioral is
9  | -- component declaration
10 | component bcd_adder_behavioral
11 | | port (
12 | |     A : in std_logic_vector (3 downto 0);
13 | |     B : in std_logic_vector (3 downto 0);
14 | |     S : out std_logic_vector (3 downto 0);
15 | |     C : out std_logic;
16 | | end component;
17 |
18 | signal A : std_logic_vector (3 downto 0);
19 | signal B : std_logic_vector (3 downto 0);
20 | signal S : std_logic_vector (3 downto 0);
21 | signal C : std_logic;
22 | begin
23 |     u1 : bcd_adder_behavioral
24 |     | port map (A, B, S, C);
25 |     process
26 |     | begin
27 |         A <= (others => '0');
28 |         B <= (others => '0');
29 |
30 |         for i in 0 to 9 loop
31 |             for j in 0 to 9 loop
32 |                 A <= std_logic_vector(to_unsigned(i,4));
33 |                 B <= std_logic_vector(to_unsigned(j,4));
34 |                 wait for 10 ns;
35 |             end loop;
36 |         end loop;
37 |
38 |         wait;
39 |     end process;
40 | end arch;
```

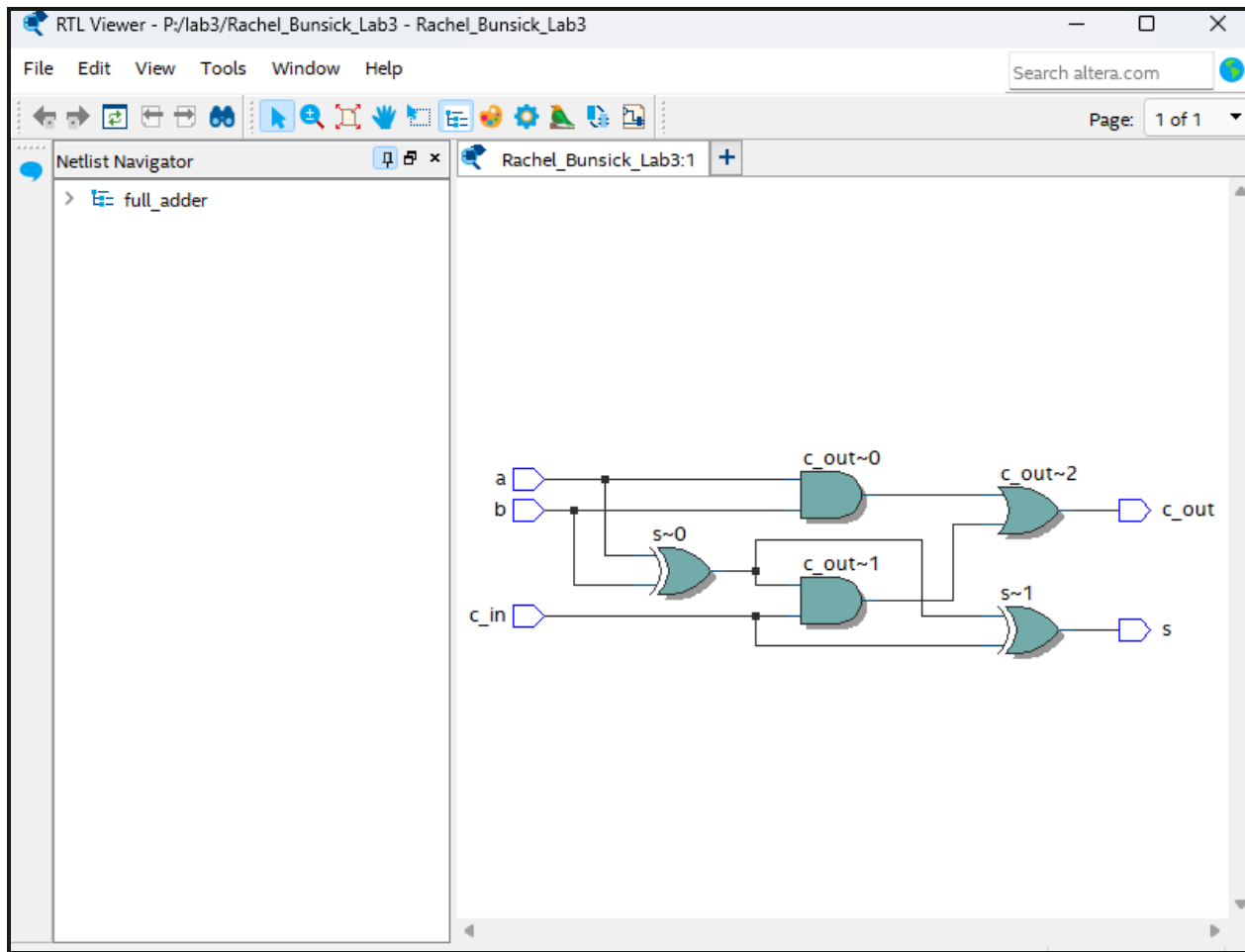
The behavioral BCD adder testbench instantiates a component using the behavioral description of the BCD adder. Then, it runs through every case of the addition of two one-digit BCD numbers. That is all cases of addition where A and B can be decimal numbers 0-9 inclusive. We use a nested for loop to achieve an exhaustive test of all possible additions.

Schematics and Waveforms:

Full Adder:

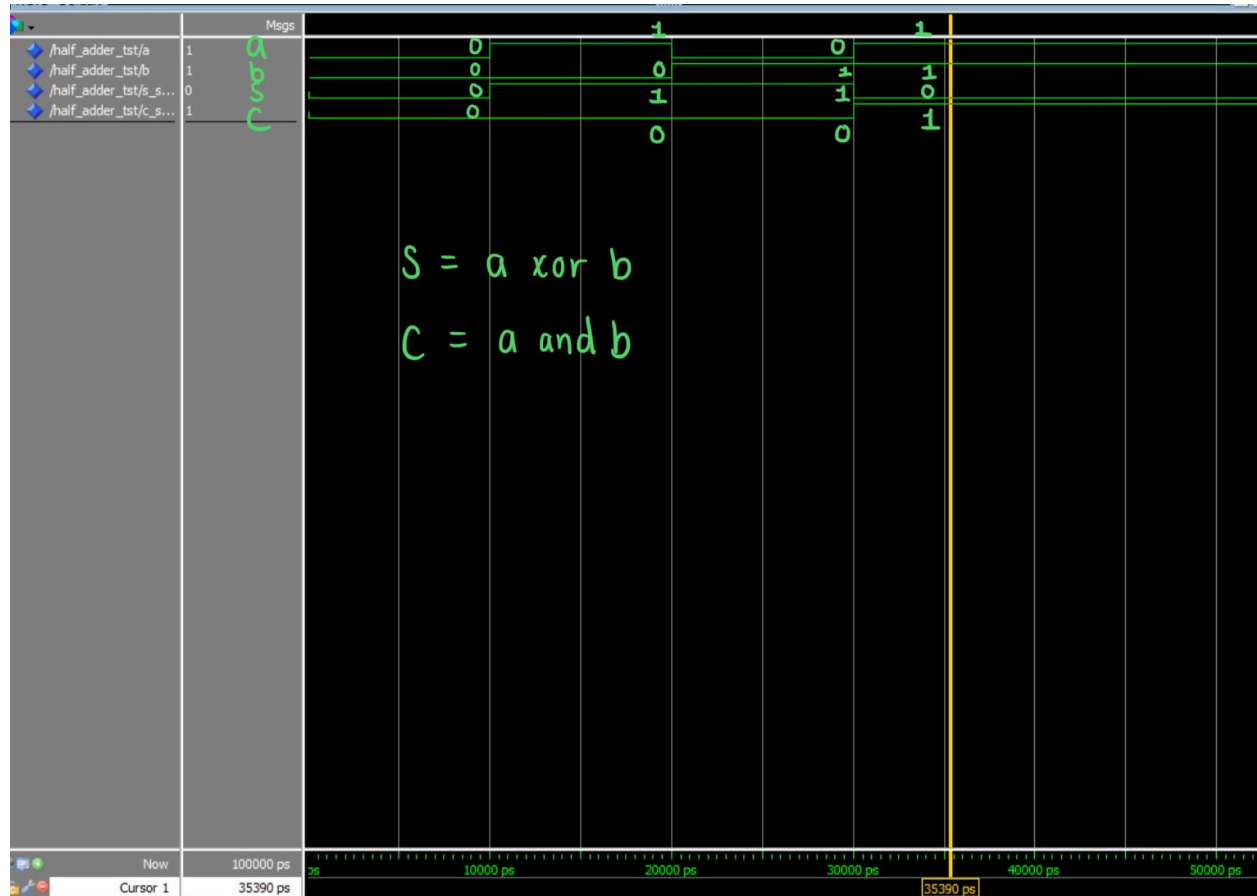


From the above waveform of the full adder, we can validate that the full adder performs as expected though our exhaustive test. Each of the eight possible test cases, with the different inputs of a, b, and carry-in creates the expected output of the sum and the carry-out (see Full-adder Testbench for reference).

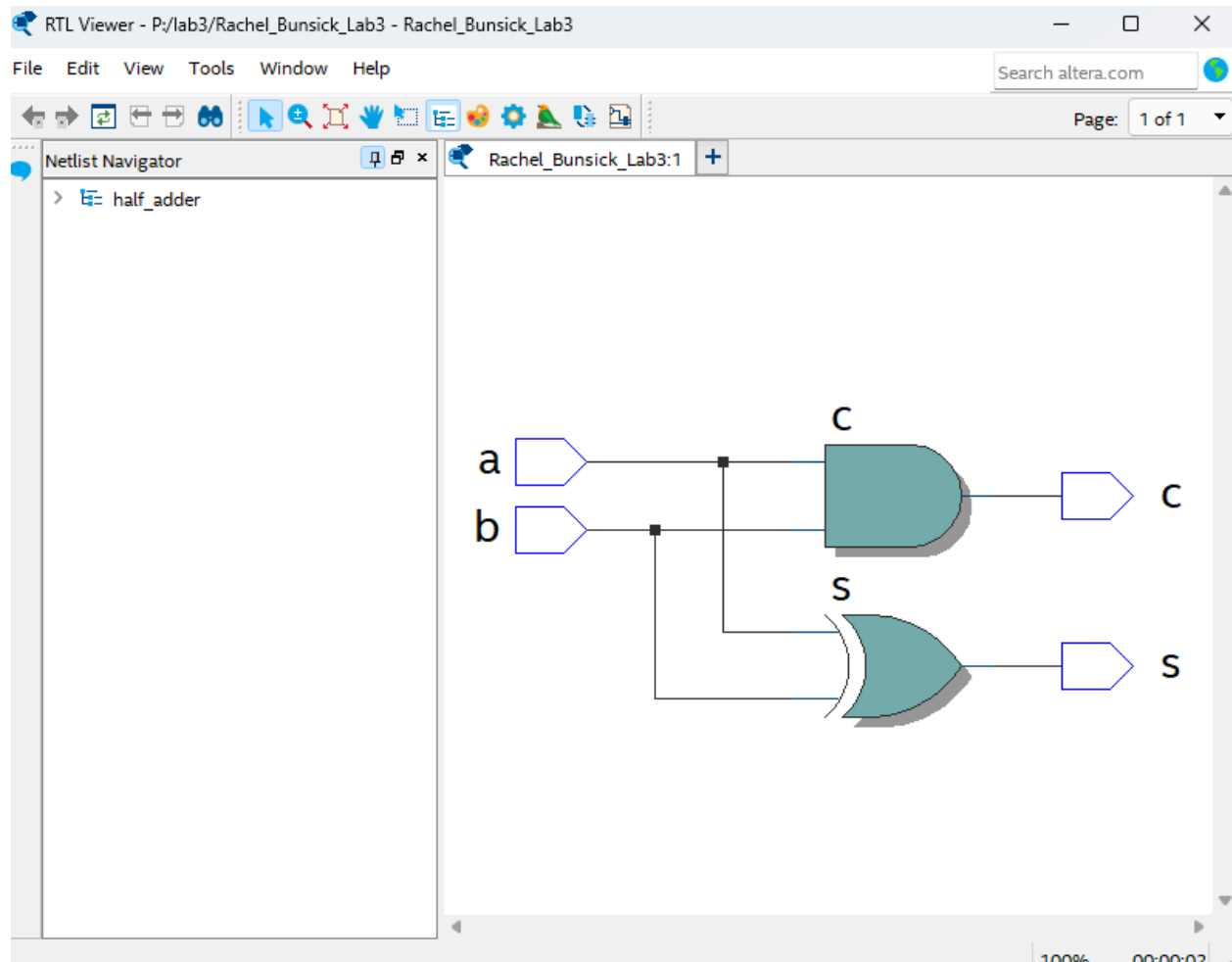


Above is the circuit implementation of the full adder. The same steps are taken as described in the Structural Description of the Full Adder (see above for reference) to obtain `c_out` and sum from inputs `a`, `b` and `c_in`.

Half Adder:

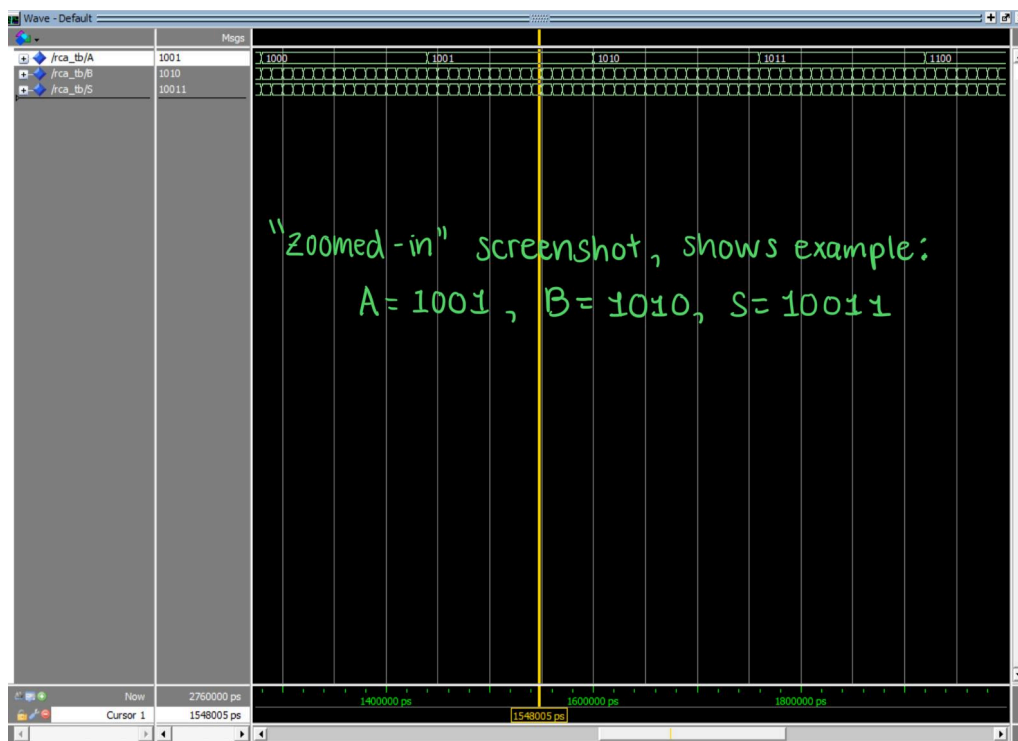
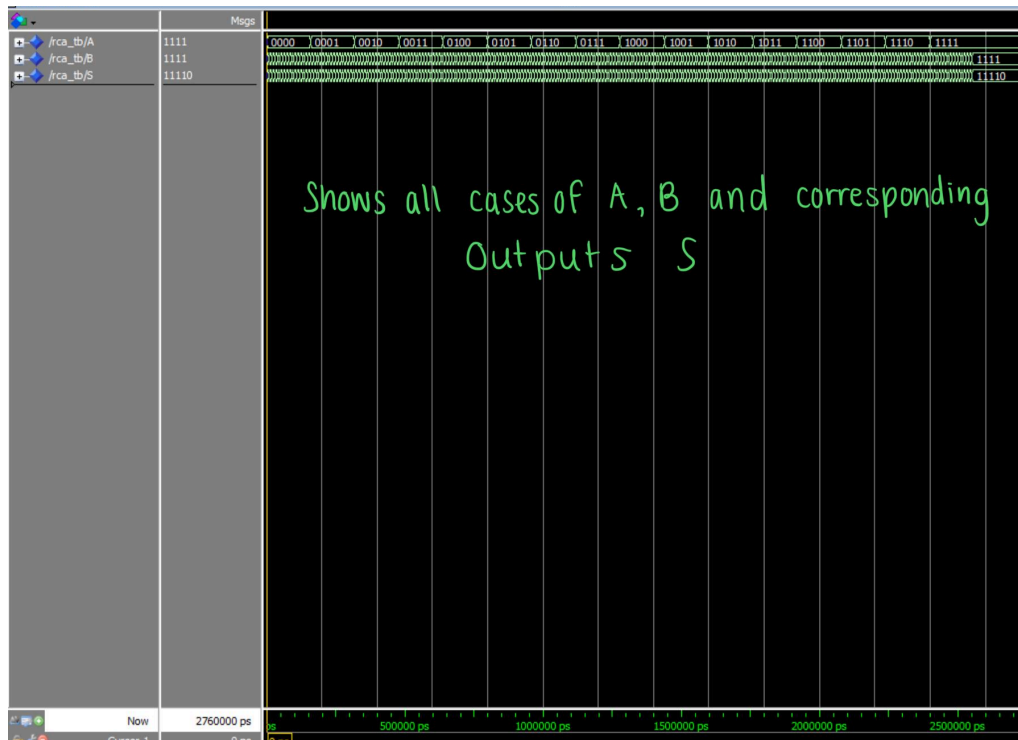


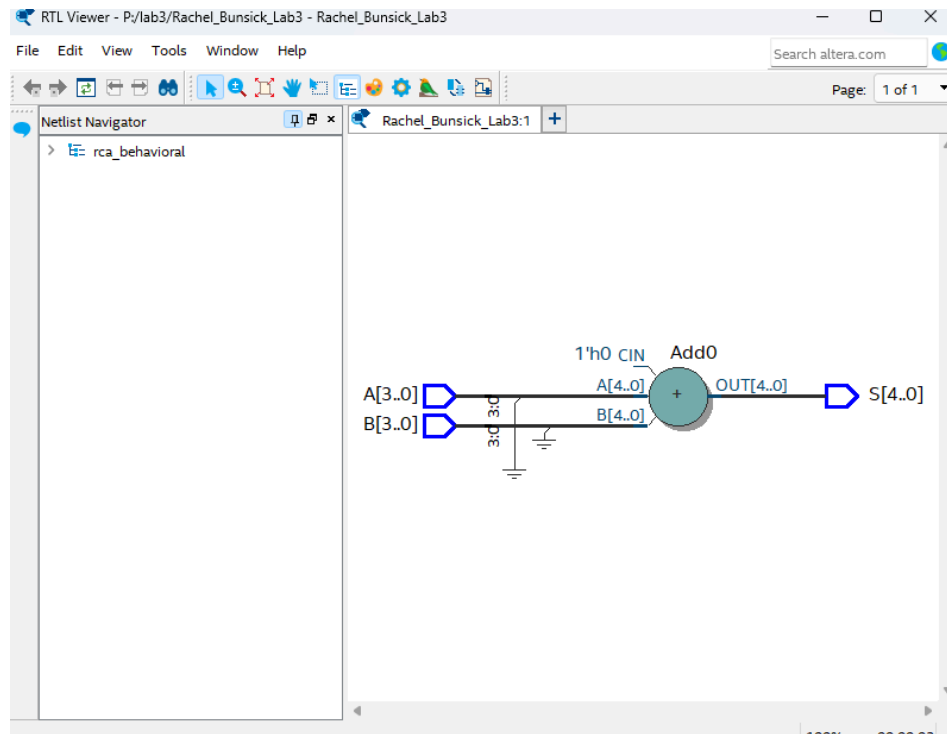
From the above waveform of the half adder, we can validate that the half adder performs as expected through our exhaustive test. Each of the four possible test cases, with the different inputs of a and b, creates the expected output of the sum and the carry (see Half-adder Testbench for reference).



Above is the circuit implementation of the half adder. The inputs a and b are xored to get the sum, and anded to obtain the carry. See above Structural Description of the Half Adder if needed.

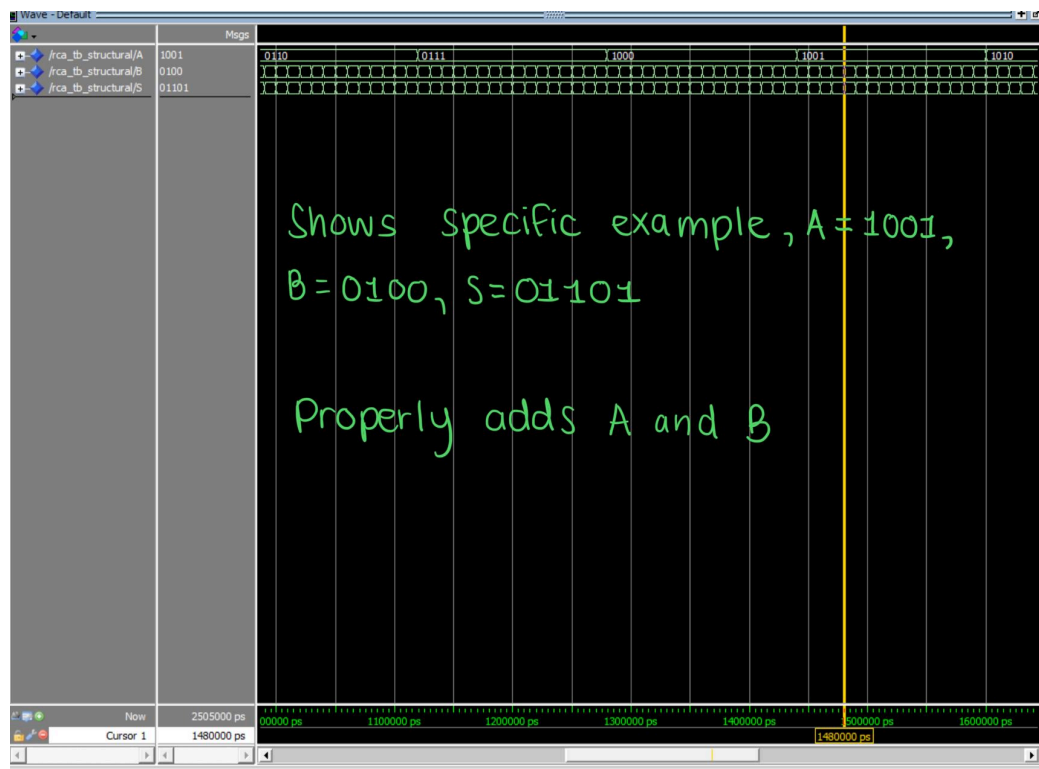
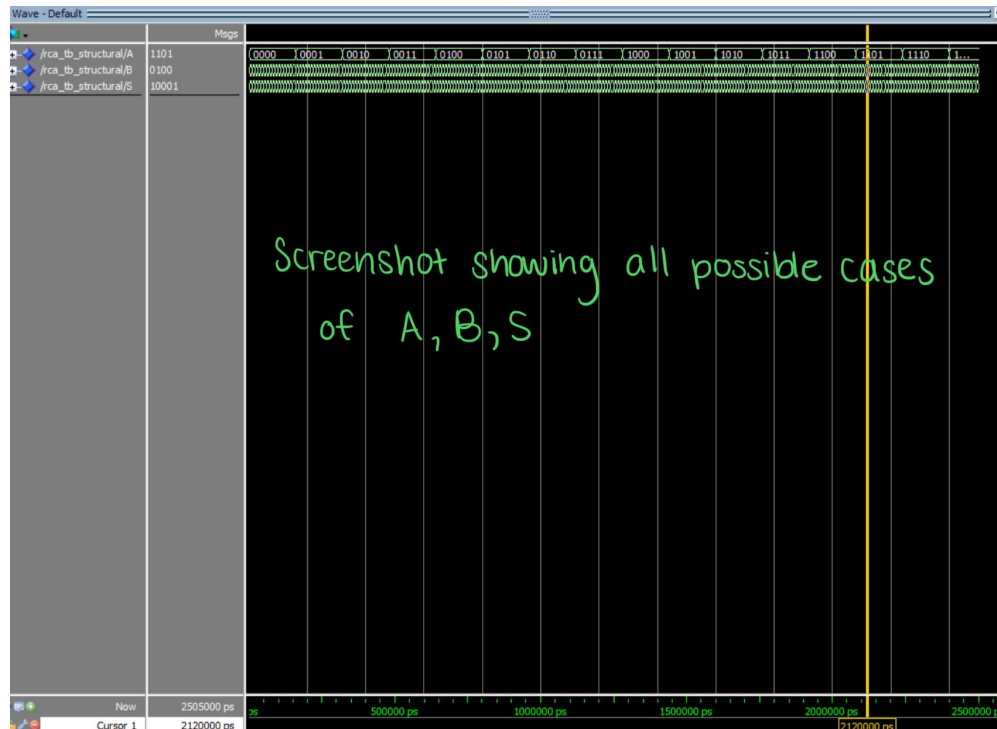
RCA Behavioral Timing Diagrams:

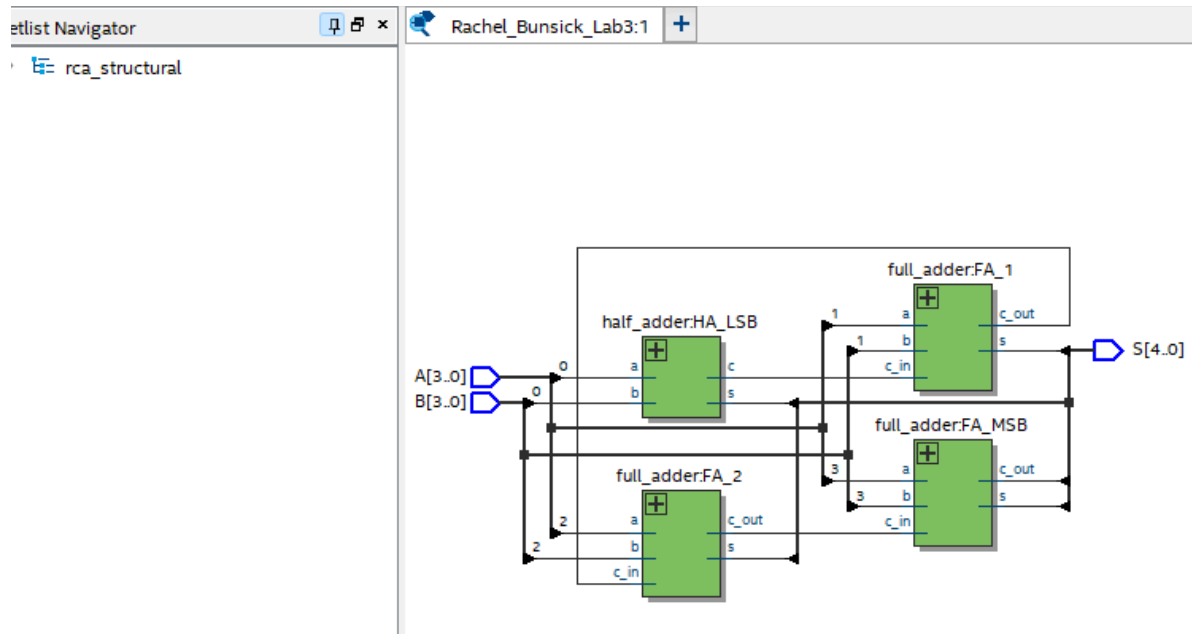




Above is the schematic representation of the RCA behavioral adder. The 4-bit inputs A and B are both connected to the “Add0” block. The “Add0” block adds A and B and represents its sum appropriately as a 5-bit output S accordingly.

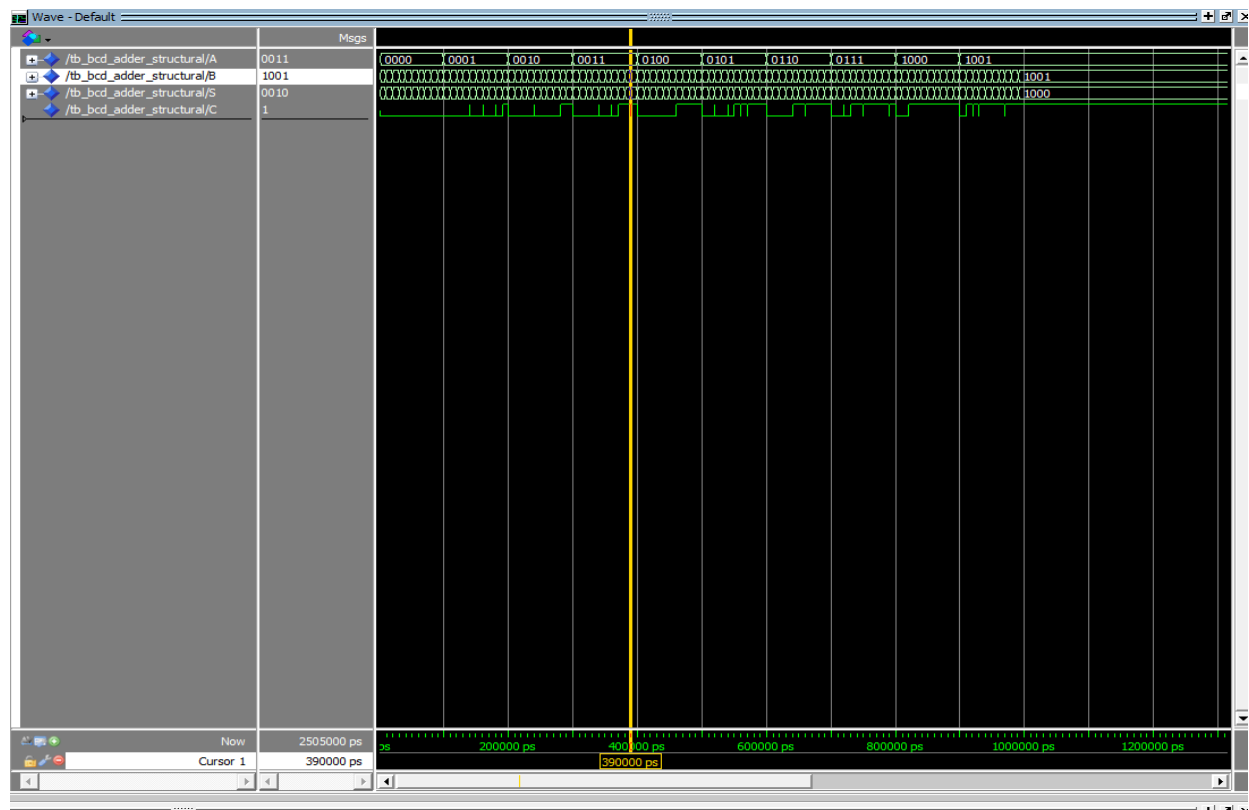
RCA structural:



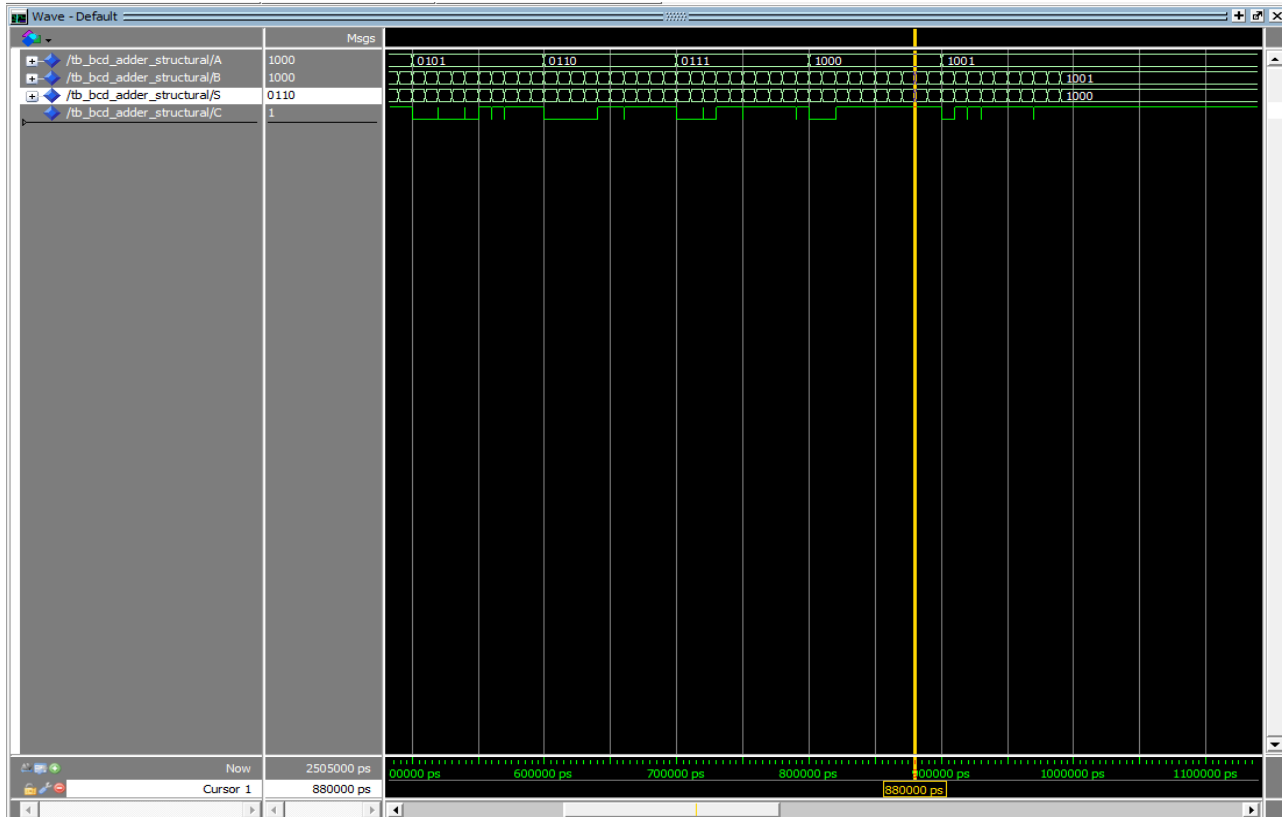


Above is the circuit implementation of the RCA. The same steps are taken as described in the Structural Description of the RCA to obtain the sum, S (5 bits), from inputs A , and B (both 4 bits). A half adder is used to add the least significant bits of A and B to get the LSB of S , then three full adders are used to add the other bits with the previous adder's carry-out. The last carry out from `full_adder:FA_MSB` becomes the MSB of the sum S . See Structural Description of 4-bit Ripple-Carry Adder on page 4 for reference if needed.

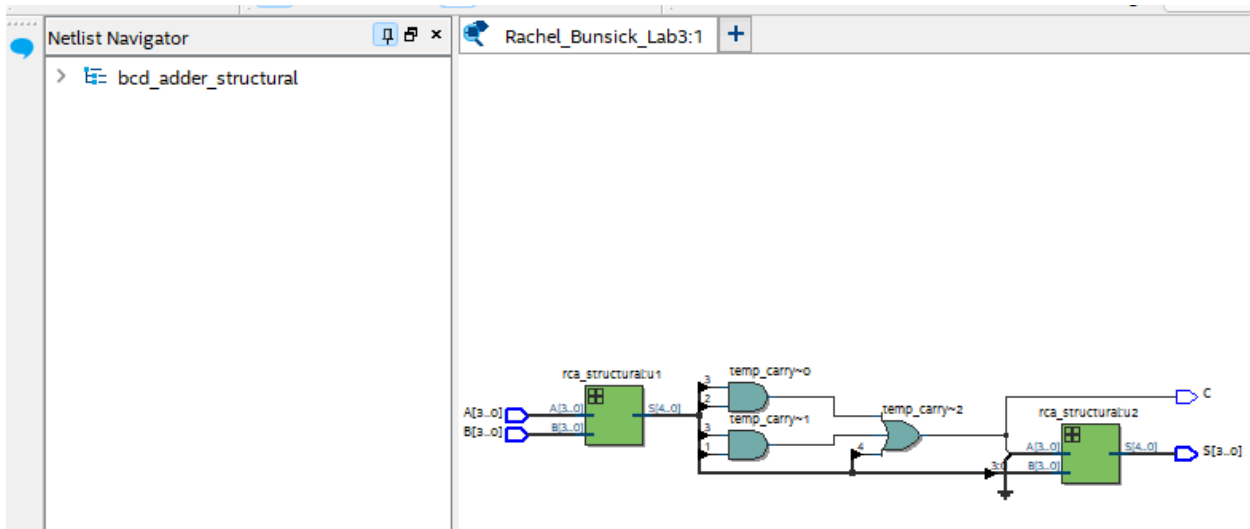
BCD structural:



Edge case: test case example for BCD adder. $A=0011$ and $B=1001$, or in decimal, $3+9=12$. We expect the carry output to be 1 and the sum output to be 2, which is what we see as $C='1'$ and $S = "0010"$.

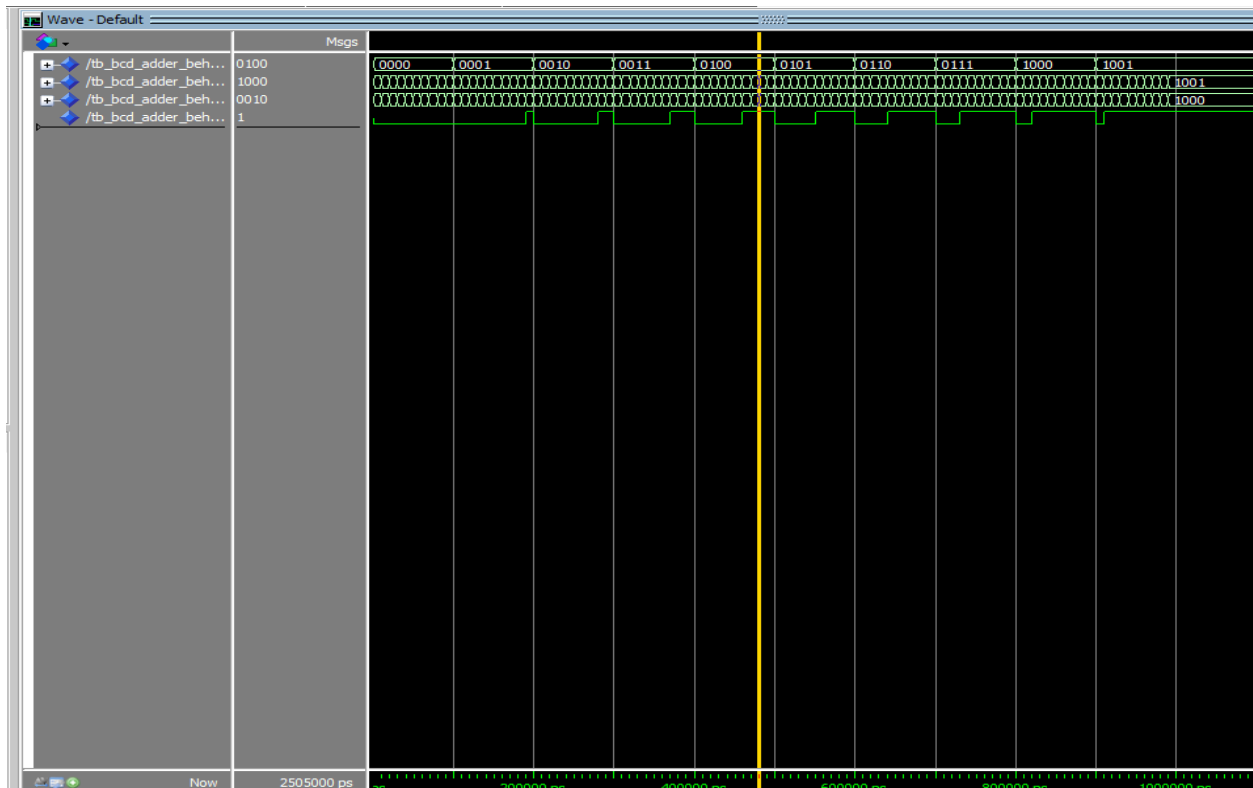


Test case example: $A = 8 = B$. $8+8=16$, so we expect the carry output to be 1 and the sum to be 6, which is what we see as $S = "0110"$ and $C = '1'$.

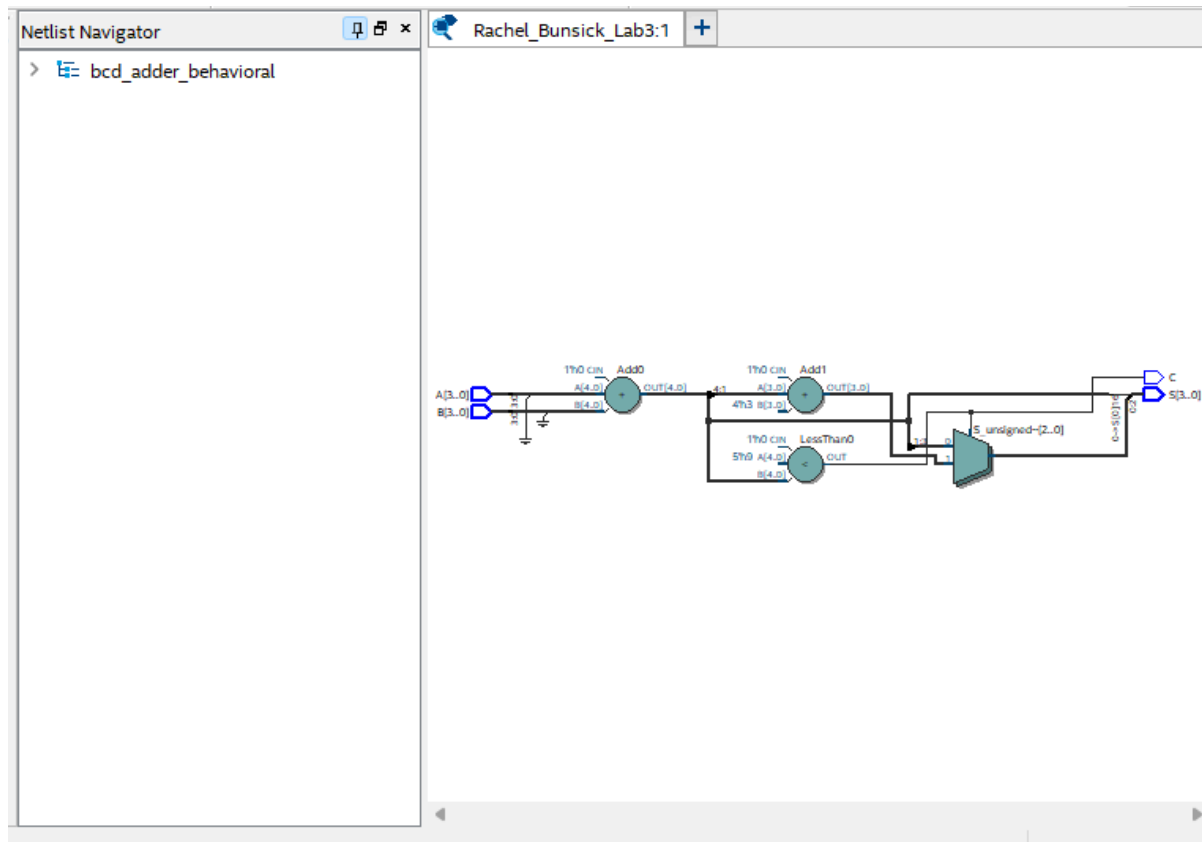


Above is the circuit implementation of the BCD adder. The same steps are taken as described in the Structural Description of the BCD Adder to obtain the four bit sum, S, and the carry, C (one bit), from inputs A, and B (both 4 bits). See One-Digit Binary Coded Decimal (BCD) Adder Structural on page 6 for reference if needed.

BCD behavioral:

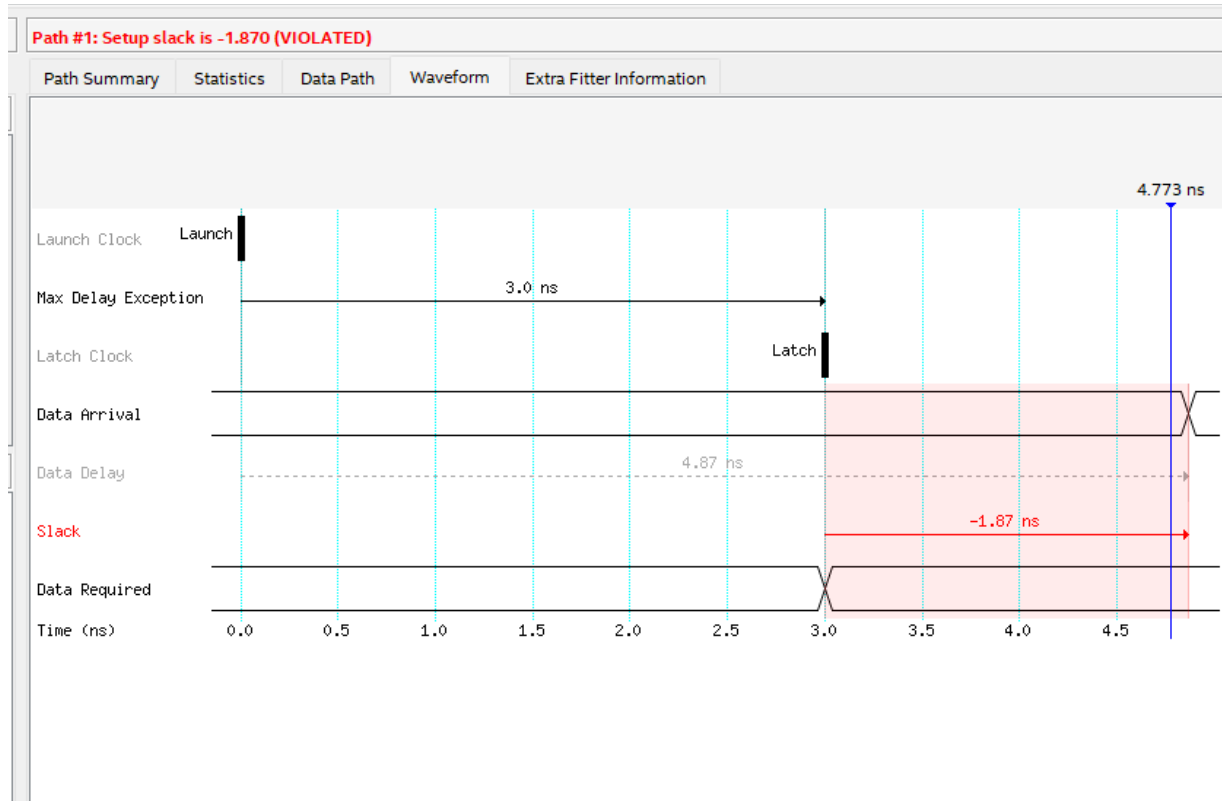


Test case example for BCD behavioral. In this edge case, we are adding $A = "0100"$ with $B = "1000"$, so $4+8 = 12$. We expect the carry out to be 1 and the sum to be 2, which is what we see as $C = '1'$ and $S = "0010"$.



This schematic diagram demonstrates how the BCD adder in behavioral form works. We use the “Add0” mechanism to add A and B in binary form. We check if this sum is greater than 9 with the “LessThan0” mechanism. Then, the output of the “LessThan0” mechanism becomes the select for a 2-1 MUX, where M[0] is the sum A+B and M[1] is the sum A+B+ “0110”. Then, the output of the MUX becomes the output sum and the carry is the output of our “LessThan0” module.

Critical Path Waveform:



The critical path of the design using the Fast 1,100 mV85C Model is from B[2] to S[3]. The data delay is 4.87 ns. So, the slack of the critical path is -1.870 ns.

Number of Pins and Logic Modules used in Design:

	RCA		One-Digit BCD Adder	
	Structural	Behavioral	Structural	Behavioral
Logic Utilization (in LUTs)	4/32,070	3/32,070	7/56,480	5/56,480
Total pins	13/457	13/457	14/268	14/268