LAB 1: BIT ADDER

Brandon Chin

CSC343 - Instructor: Prof. Izidor Gertner

2/20/2015

Lab 1: Bit Adder

Objective:

To extend our knowledge in ModelSim by designing, simulating, and testing 1-bit, 2-bit, and 16-bit adder circuits.

Functionality and Specifications:

A bit adder is a digital circuit that is designed to take two binary string inputs, and return the sum of those inputs.

1-Bit Full Adder

The 1-bit full adder has three inputs (A,B, and Cin) and two outputs (S and Cout). Each of these inputs and outputs are 1-bit binary numbers. A and B are the numbers being added, and Cin is the carry in input. S is the sum of A, B, and Cin, and Cout is the carry out output. The carry out is required for the cases where the adder will need an extra bit to store the sum (1 + 1 = 10, and 1 + 1 + 1 = 11).

Here is the truth table for a 1-bit full adder:

Input			Output	
A	B	Cin	Cout	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Using this truth table, we are able to derive logical expressions describing this table:

Sum = [(not A)(not B)(Cin)]+[(not A)(B)(not Cin)]+[(A)(not B)(not Cin)]+[(A)(B)(Cin)],
Cout = [(not A)(B)(Cin)]+[(A)(not B)(Cin)]+[(A)(B)(not Cin)]+[(A)(B)(Cin)].

Here is the 1-bit full adder VHDL code:

```
-- This is just to make a reference to some common things needed.
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- We declare the 1-bit adder with the inputs and outputs
-- shown inside the port().
-- This will add two bits together(x,y), with a carry in(cin) and
-- output the sum(sum) and a carry out(cout).
entity BIT_ADDER is
        port( a, b, cin         : in  STD_LOGIC;
              sum, cout         : out STD_LOGIC );
end BIT_ADDER;

-- This describes the functionality of the 1-BIT adder.
architecture BHV of BIT_ADDER is
begin

        -- Calculate the sum of the 1-BIT adder.
        sum <=  (not a and not b and cin) or
                        (not a and b and not cin) or
                        (a and not b and not cin) or
                        (a and b and cin);

        -- Calculates the carry out of the 1-BIT adder.
        cout <= (not a and b and cin) or
                        (a and not b and cin) or
                        (a and b and not cin) or
                        (a and b and cin);
end BHV;
```

2-Bit Adder

The Cin bit is usually grounded to 0, since we just want the sum of A and B only. However, often times 1-bit full adders are used as components in a cascade of adders in order to create larger bit adders. The Cout bit of one adder becomes the Cin bit of the successive adder. This is how we create our 2-bit adder.

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- This describes the black-box view of the component we are
-- designing. The inputs and outputs are again described
-- inside port(). It takes two 2-bit values as input (x and y)
-- and produces a 2-bit output (ANS) and a carry out bit (Cout).

entity add2 is
        port( a, b              : in    STD_LOGIC_VECTOR(1 downto 0);
              ans               : out   STD_LOGIC_VECTOR(1 downto 0);
              cout              : out   STD_LOGIC                );
end add2;
```

```
-- Although we have already described the inputs and outputs,
-- we must now describe the functionality of the adder (ie:
-- how we produced the desired outputs from the given inputs).

architecture STRUCTURE of add2 is

-- We are going to need two 1-bit adders, so include the
-- design that we have already done above.

component BIT_ADDER
        port( a, b, cin         : in  STD_LOGIC;
              sum, cout         : out STD_LOGIC );
end component;

-- Now create the signals which are going to be necessary
-- to pass the outputs of one adder to the inputs of the next
-- in the sequence.
signal c0, c1 : STD_LOGIC;
begin

c0 <= '0';
b_adder0: BIT_ADDER port map (a(0), b(0), c0, ans(0), c1);
b_adder1: BIT_ADDER port map (a(1), b(1), c1, ans(1), cout);

END STRUCTURE;
```

16-Bit Adder-Subtractor

To design a 16-bit adder would be very similar - port mapping individual 1-bit adders together in a cascade. However, we want our 16-bit adder to also be able to subtract numbers. We also want it to detect overflow exceptions (for example, adding more than the largest positive number stored in 16 bits, or subtracting more than the most negative number stored in 16 bits).

Components

Before getting to the design of the 16-bit adder-subtractor, we must define the entities of our components. Here, I've created an XOR logic gate, called "xorGate", as an entity to be used later with the port maps. I have also created the 1-bit full adder from before, however this time I have simplified the logical expression a bit for convenience.

```
-- XOR gate
library ieee;
use ieee.std_logic_1164.all;

entity xorGate is
   port( a, b : in std_logic;
         f : out std_logic);
end xorGate;
--
architecture ARCH of xorGate is
begin
   f <= a xor b;
end ARCH;

-- FULL BIT ADDER
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
   port( x, y, cin : in std_logic;
         sum, cout : out std_logic);
end full_adder;

architecture ARCH of full_adder is
begin
   sum <= (x xor y) xor cin;
   cout <= (x and (y or cin)) or (cin and y);
end ARCH;
```

Putting It Together

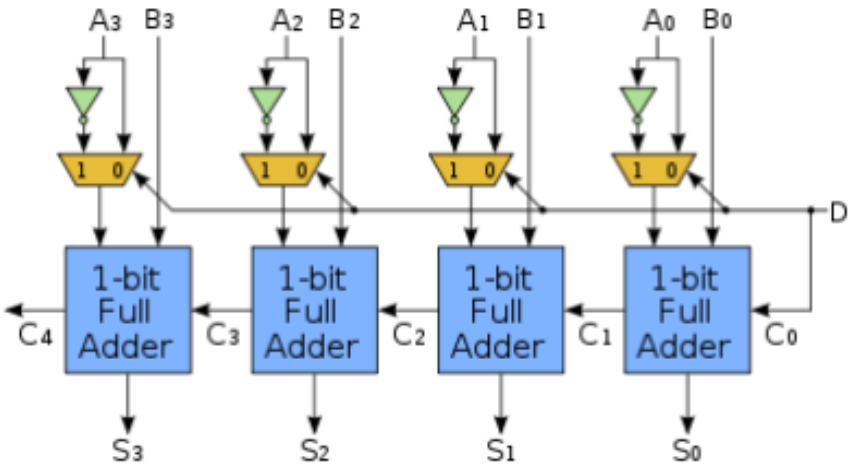
Next, we will define our 16-bit adder-subtractor and include these entities as components into our design. There are three inputs to our adder-subtractor, A, B, and mode, where mode is the operation between addition or subtraction (0 = addition, 1 = subtraction). There are also three outputs, S, cout, and overflow, where overflow represents whether or not overflow has been detected after the operation (0 = no overflow, 1 = overflow detected).

```
-- 16-bit Adder/Subtractor
library ieee;
use ieee.std_logic_1164.all;

entity addsub16 is
   port( mode            : in std_logic;
         A               : in std_logic_vector(15 downto 0);
         B               : in std_logic_vector(15 downto 0);
         S               : out std_logic_vector(15 downto 0);
         cout, overflow  : out std_logic);
end addsub16;

architecture struct of addsub16 is

   component xorGate is            --XOR component
       port( a, b : in std_logic;
             f : out std_logic);
   end component;

   component full_adder is              --FULL ADDER component
      port( x, y, cin : in std_logic;
            sum, cout : out std_logic);
   end component;
```

Port Maps

Finally, we include the port maps, connecting all of these components together. We also created two signals C and X, where C are the intermediate carry bits, and X are the XOR outputs between the circuit.

```
   signal C     :std_logic_vector(15 downto 0);         -- intermediate carries
   signal X     : std_logic_vector(15 downto 0);        -- xor outputs

begin
   GX0: xorGate port map(mode, B(0), X(0));
   GX1: xorGate port map(mode, B(1), X(1));
   GX2: xorGate port map(mode, B(2), X(2));
   GX3: xorGate port map(mode, B(3), X(3));
   GX4: xorGate port map(mode, B(4), X(4));
   GX5: xorGate port map(mode, B(5), X(5));
   GX6: xorGate port map(mode, B(6), X(6));
   GX7: xorGate port map(mode, B(7), X(7));
   GX8: xorGate port map(mode, B(8), X(8));
   GX9: xorGate port map(mode, B(9), X(9));
   GX10: xorGate port map(mode, B(10), X(10));
   GX11: xorGate port map(mode, B(11), X(11));
   GX12: xorGate port map(mode, B(12), X(12));
   GX13: xorGate port map(mode, B(13), X(13));
   GX14: xorGate port map(mode, B(14), X(14));
   GX15: xorGate port map(mode, B(15), X(15));


   FA0: full_adder port map(A(0), X(0), mode,  S(0), C(0));
   FA1: full_adder port map(A(1), X(1), C(0),  S(1), C(1));
   FA2: full_adder port map(A(2), X(2), C(1),  S(2), C(2));
   FA3: full_adder port map(A(3), X(3), C(2),  S(3), C(3));
   FA4: full_adder port map(A(4), X(4), C(3),  S(4), C(4));
   FA5: full_adder port map(A(5), X(5), C(4),  S(5), C(5));
   FA6: full_adder port map(A(6), X(6), C(5),  S(6), C(6));
   FA7: full_adder port map(A(7), X(7), C(6),  S(7), C(7));
   FA8: full_adder port map(A(8), X(8), C(7),  S(8), C(8));
   FA9: full_adder port map(A(9), X(9), C(8),  S(9), C(9));
   FA10: full_adder port map(A(10), X(10), C(9),  S(10), C(10));
   FA11: full_adder port map(A(11), X(11), C(10),  S(11), C(11));
   FA12: full_adder port map(A(12), X(12), C(11),  S(12), C(12));
   FA13: full_adder port map(A(13), X(13), C(12),  S(13), C(13));
   FA14: full_adder port map(A(14), X(14), C(13),  S(14), C(14));
   FA15: full_adder port map(A(15), X(15), C(14),  S(15), C(15));

   overflow_out: xorGate port map(C(14), C(15), overflow);
   cout <= C(15);
end struct;
```



(4-bit Adder-Subtractor Digital Circuit Design)

Subtraction (w/ Two's Complement)

If we look at the above circuit design, we will see an example of a 4-bit adder-subtractor that we can use to help understand what we are doing in our 16-bit adder-subtractor. In order to perform addition as well as subtraction, we have to use a method called two's complement, which allows us to represent negative integers in binary. Positive numbers are represented as themselves, however negative numbers are represented by the two's complement of their absolute value. In other words, take the absolute value of the negative number, invert each bit (0's become 1's and 1's become 0's), then add one to the string. This new form is your negative number. Something to notice -- strings whose most significant bit are a 0 are positive, and those that are a 1 are negative.

This can make it easier to perform subtraction -- In the diagram, there is an input called D (in our addsub16, it is called mode), and this is used to determine the operation. D is fed into two ports, one is the first 1-bit full adder as the cin, and the other is into a series of multiplexors (in our addsub16, we use XOR gates). If D is 0, the multiplexor allows the two inputs to enter the adders as normal, and sets the first cin to 0 as well. However, if D is 1, this inverts one of the input bits before it is passed into the adder, and the cin is set to 1, which adds 1 to our result. This takes care of subtraction by basically **adding A to the negative of B**.

Overflow Detection

Overflow is defined as a condition that occurs when the result of an operation is greater in value than the given register that it must be stored in. In our example, there are two cases when overflow can occur. One, when we add more than the largest positive 16-bit signed integer, or two, when we subtract more than the most negative 16-bit signed integer. Now how can we detect this in our hardware?

We must look at the two most significant carry out bits of our circuit, *i.e.* the carry in and carry out of the last full adder of our circuit. Let the carry out of the full adder adding the least significant bit be called C(0), then the carry out of the full adder adding the two most significant bits are C(14) and C(15). On our circuit, we define overflow to be the result of the following equation:

Overflow = C(14) XOR C(15)

The XOR of the C(14) and C(15) differ if there's either a 1 being carried in and a 0 being carried out, or if there's a 0 being carried in and a 1 being carried out.

Case 1: If a 1 is carried in and a 0 is carried out, then A(15) = 0 and B(15) = 0, and the sum is 1. This is the case when two non-negative numbers are added, and yet a negative number becomes the result.

Case 2: If a 0 is carried in and a 1 is carried out, then A(15) = 1 and B(15) = 1, and the sum is 0. This is the case when two negative numbers are added, and yet a non-negative number becomes the result.

In either case, the result is incorrect, and overflow returns a 1, indicating overflow has been detected.

Simulation:

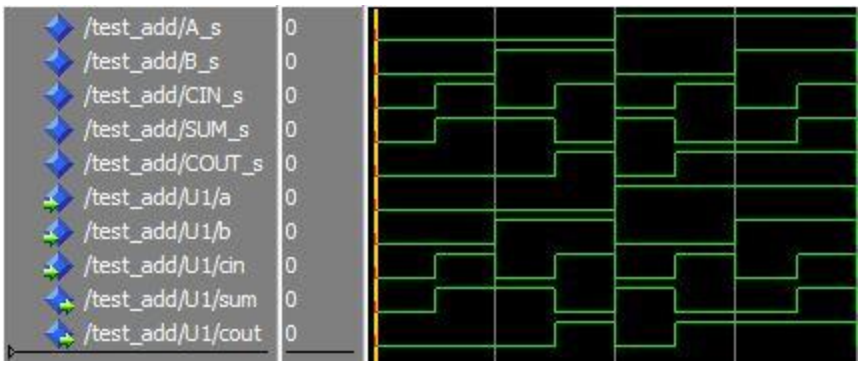
We simulated and tested our circuit designs by writing a test VHDL file and running it through a waveform simulation on ModelSim.

1-Bit Adder

```
-- A testbench is used to rigorously tests a design that you have made.
-- The output of the testbench should allow the designer to see if
-- the design worked.  The testbench should also report where the testbench
-- failed.

-- This is just to make a reference to some common things needed.
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Decare a testbench.  Notice that the testbench does not have any
-- input or output ports.
entity TEST_ADD is
end TEST_ADD;
```

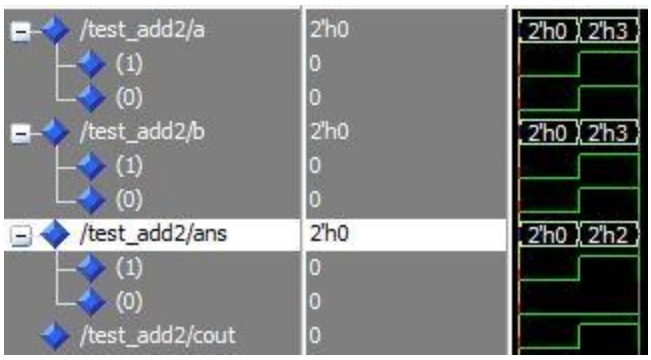
```
-- Describes the functionality of the tesbench.
architecture TEST of TEST_ADD is

	-- The object that we wish to test is declared as a component of
	-- the test bench. Its functionality has already been described elsewhere.
	-- This simply describes what the object's inputs and outputs are, it
	-- does not actually create the object.
	component BIT_ADDER
		port( a, b, cin		: in  STD_LOGIC;
		sum, cout		: out STD_LOGIC );
	end component;

	-- Specifies which description of the adder you will use.
	for U1: BIT_ADDER use entity WORK.BIT_ADDER(BHV);

	-- Create a set of signals which will be associated with both the inputs
	-- and outputs of the component that we wish to test.
	signal A_s, B_s : STD_LOGIC;
	signal CIN_s    : STD_LOGIC;
	signal SUM_s    : STD_LOGIC;
	signal COUT_s   : STD_LOGIC;


	-- This is where the testbench for the BIT_ADDER actually begins.
	begin

	-- Create a 1-bit adder in the testbench.
	-- The signals specified above are mapped to their appropriate
	-- roles in the 1-bit adder which we have created.
	U1: BIT_ADDER port map (A_s, B_s, CIN_s, SUM_s, COUT_s);

	-- The process is where the actual testing is done.
	process
	begin

		-- We are now going to set the inputs of the adder and test
		-- the outputs to verify the functionality of our 1-bit adder.

		-- Case 0 : 0+0 with carry in of 0.

		-- Set the signals for the inputs.
		A_s <= '0';
		B_s <= '0';
		CIN_s <= '0';

		-- Wait a short amount of time and then check to see if the
		-- outputs are what they should be. If not, then report an error
		-- so that we will know there is a problem.
		wait for 10 ns;
		assert ( SUM_s = '0'  ) report "Failed Case 0 - SUM" severity error;
		assert ( COUT_s = '0' ) report "Failed Case 0 - COUT" severity error;
		wait for 40 ns;
```

```
        -- Case 1 : 0+0 with carry in of 1.
        A_s <= '0';
        B_s <= '0';
        CIN_s <= '1';
        wait for 10 ns;
        assert ( SUM_s = '1'  ) report "Failed Case 1 - SUM" severity error;
        assert ( COUT_s = '0' ) report "Failed Case 1 - COUT" severity error;
        wait for 40 ns;

        -- Case 2 : 0+1 with carry in of 0.
        A_s <= '0';
        B_s <= '1';
        CIN_s <= '0';
        wait for 10 ns;
        assert ( SUM_s = '1'  ) report "Failed Case 2 - SUM" severity error;
        assert ( COUT_s = '0' ) report "Failed Case 2 - COUT" severity error;
        wait for 40 ns;

        -- Case 3 : 0+1 with carry in of 1.
        A_s <= '0';
        B_s <= '1';
        CIN_s <= '1';
        wait for 10 ns;
        assert ( SUM_s = '0'  ) report "Failed Case 3 - SUM" severity error;
        assert ( COUT_s = '1' ) report "Failed Case 3 - COUT" severity error;
        wait for 40 ns;

      -- Case 4 : 1+0 with carry in of 0.
      A_s <= '1';
      B_s <= '0';
      CIN_s <= '0';
      wait for 10 ns;
      assert ( SUM_s = '1'  ) report "Failed Case 4 - SUM" severity error;
      assert ( COUT_s = '0' ) report "Failed Case 4 - COUT" severity error;
      wait for 40 ns;

      -- Case 5 : 1+0 with carry in of 1.
      A_s <= '1';
      B_s <= '0';
      CIN_s <= '1';
      wait for 10 ns;
      assert ( SUM_s = '0'  ) report "Failed Case 5 - SUM" severity error;
      assert ( COUT_s = '1' ) report "Failed Case 5 - COUT" severity error;
      wait for 40 ns;

      -- Case 6 : 1+1 with carry in of 0.
      A_s <= '1';
      B_s <= '1';
      CIN_s <= '0';
      wait for 10 ns;
      assert ( SUM_s = '0'  ) report "Failed Case 6 - SUM" severity error;
      assert ( COUT_s = '1' ) report "Failed Case 6 - COUT" severity error;
      wait for 40 ns;

        -- Case 7 : 1+1 with carry in of 1.
        A_s <= '1';
        B_s <= '1';
        CIN_s <= '1';
        wait for 10 ns;
        assert ( SUM_s = '1'  ) report "Failed Case 7 - SUM" severity error;
        assert ( COUT_s = '1' ) report "Failed Case 7 - COUT" severity error;
        wait for 40 ns;

    end process;
END TEST;
```



2-Bit Adder

```
architecture TEST of TEST_ADD2 is

        component add2
                port( a, b      : in    STD_LOGIC_VECTOR(1 downto 0);
                      ans       : out   STD_LOGIC_VECTOR(1 downto 0);
                      cout      : out   STD_LOGIC       );
        end component;

        for U1: add2 use entity WORK.ADD2(STRUCTURE);
        signal a, b             : STD_LOGIC_VECTOR(1 downto 0);
        signal ans              : STD_LOGIC_VECTOR(1 downto 0);
        signal cout             : STD_LOGIC;

        begin
        U1: add2 port map (a,b,ans,cout);


                process
                begin

                -- Case 1 that we are testing.

                        a <= "00";
                        b <= "00";
                        wait for 10 ns;
                        assert ( ANS = "00" )   report "Failed Case 1 - ANS" severity error;
                        assert ( Cout = '0' )   report "Failed Case 1 - Cout" severity error;
                        wait for 40 ns;

                -- Case 2 that we are testing.

                        a <= "11";
                        b <= "11";
                        wait for 10 ns;
                        assert ( ANS = "10" )   report "Failed Case 2 - ANS" severity error;
                        assert ( Cout = '1' )   report "Failed Case 2 - Cout" severity error;
                        wait for 40 ns;
                end process;
END TEST;
```

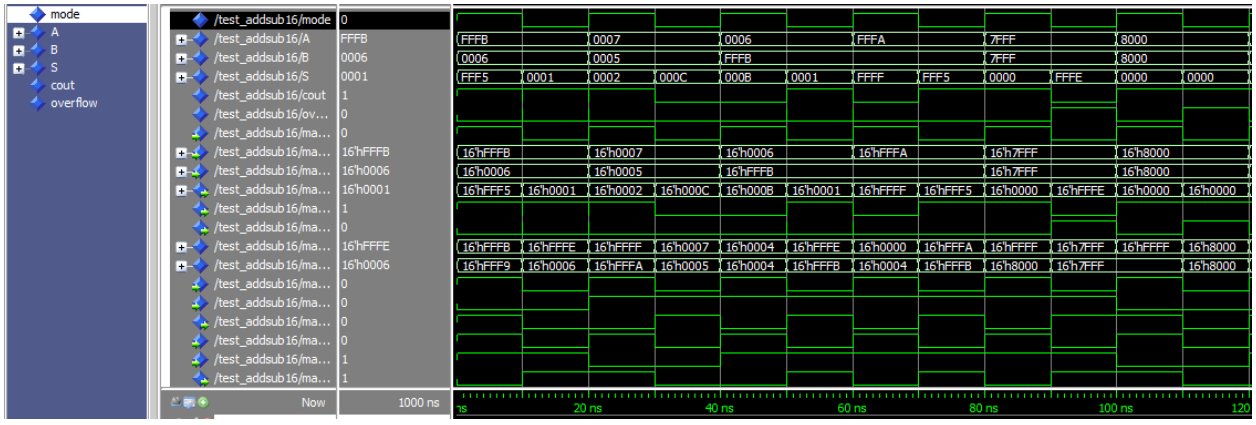


16-Bit Adder-Subtractor

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

entity test_addsub16 is
end test_addsub16;

architecture arch of test_addsub16 is
   component addsub16 is
      port( mode         : in std_logic;
            A            : in std_logic_vector(15 downto 0);
            B            : in std_logic_vector(15 downto 0);
            S            : out std_logic_vector(15 downto 0);
            cout, overflow  : out std_logic);
   end component;

   signal mode          : std_logic;
   signal A, B          : std_logic_vector(15 downto 0);
   signal S             : std_logic_vector(15 downto 0);
   signal cout, overflow : std_logic;


begin
   mapping: addsub16 port map(
                              mode,
                     A(15 downto 0),
                     B(15 downto 0),
                     S(15 downto 0),
                           cout, overflow);

   process
   begin
      mode <= '1'; -- subtraction
      wait for 10 ns;
      mode <= '0'; -- addition
      wait for 10 ns;
   end process;
```

```
process
   variable errCnt : integer := 0;
begin
   -- TEST 1 (negative +- positive)
   A <= "1111111111111011";
   B <= "0000000000000110";

   wait for 20 ns;
   assert (cout = '1') report "Error" severity error;
   assert (S = "0000000000000001") report "Error" severity error;
   assert (overflow = '0') report "Error" severity error;
   if(cout /= '1' or overflow /= '0') then
      errCnt:= errCnt + 1;
   end if;

   --TEST 2 (positive +- positive)
   A <= "0000000000000111";
   B <= "0000000000000101";

   wait for 20 ns;
   assert (cout = '0') report "Error" severity error;
   assert (S = "0000000000001100") report "Error" severity error;
   assert (overflow = '0') report "Error" severity error;
   if(cout /= '0' or overflow /= '0') then
      errCnt:= errCnt + 1;
   end if;

  -- TEST 3 (positive +- negative)
  A <= "0000000000000110";
  B <= "1111111111111011";

  wait for 20 ns;
  assert (cout = '1') report "Error" severity error;
  assert (S = "0000000000000001") report "Error" severity error;
  assert (overflow = '0') report "Error" severity error;
  if (cout /= '1' or overflow /= '0') then
      errcnt := errcnt + 1;
  end if;

  -- TEST 4 (negaitve +- negative)
  A <= "1111111111111010";
  B <= "1111111111111011";

  wait for 20 ns;
  assert (cout = '1') report "Error" severity error;
  assert (S = "1111111111110101") report "Error" severity error;
  assert (overflow = '0') report "Error" severity error;
  if (cout /= '1' or overflow /= '0') then
      errcnt := errcnt + 1;
  end if;
```

```
    -- TEST 5 (positive overflow)
    A <= "0111111111111111";
    B <= "0111111111111111";

    wait for 20 ns;
    assert (cout = '0') report "Error" severity error;
    assert (S = "1111111111111110") report "Error" severity error;
    assert (overflow = '1') report "Error" severity error;
    if (cout /= '0' or overflow /= '1') then
        errcnt := errcnt + 1;
    end if;

    -- TEST 6 (negative overflow)
    A <= "1000000000000000";
    B <= "1000000000000000";

    wait for 20 ns;
    assert (cout = '1') report "Error" severity error;
    assert (S = "0000000000000000") report "Error" severity error;
    assert (overflow = '1') report "Error" severity error;
    if (cout /= '1' or overflow /= '1') then
        errcnt := errcnt + 1;
    end if;

     ------------ SUMMARY ----------------
     if(errCnt = 0) then
        assert false report "Good!"  severity note;
     else
        assert false report "Error!"  severity error;
     end if;

   end process;
end arch;
```



Conclusion:

Beginning with a basic component, we continued to extend the functionality of our circuit. The full adder is very important in digital circuits because of how common it is used. Our lab connected multiple full adders to create adders of larger magnitudes, able to add strings of multiple bits. Then, we took this concept, and allowed our circuit to not only add, but subtract as well. This was essentially achieved by adding A to the negative of B. Last, we enabled our circuit to detect overflow occurrences by observing the carry ins and carry outs of the most significant full adder.