

EXPERIMENT 8

DESIGN AND SIMULATION OF A 4-BIT RIPPLE-CARRY ADDER USING FOUR FULL ADDERS IN VHDL

Purpose

Familiarization with VHSIC Hardware Description Language (VHDL) and with VHDL design tools. VHDL is an increasingly important tool in digital design used for automated specification and testing of digital systems. In this exercise, you will write a VHDL specification for a full adder and use this full adder component to create a 4-bit ripple-carry (RCA) adder. More specifically, you will be instantiating four full adder components to structurally model the 4-bit RCA. You will also simulate and test your VHDL “code” using ModelSim.

References

VHDL tutorial
ModelSim user manual
<http://www.model.com/>

Materials Required

ModelSim

Background

In lab8, the VHDL code is generated from the schematic in Quartus II automatically. However, if the circuit is too large or too complicated to draw, writing the VHDL code is a good option. An example VHDL specification is given below, along with a function block diagram. Here you will be creating a VHDL description of a full adder as illustrated in Figure 8-1. A portion of the VHDL code for this adder is provided below. To create the 4-bit RCA, the full adder component will be instantiated four times and structurally connected. A short illustrative example follows for clarification.

Figure 8-1 shows the block diagram of a full adder with input ports a , b , and carry input c_{in} , and output ports sum and carry output, c_{out} . File FA.vhd (shown below) implements the full adder component.

Figure 8-2 shows the structural model of a 4-bit RCA, using four full adders. It has two 4-bit input ports “A[3:0]” & “B[3:0]”, and one carry in input, c_{in} set to ‘0’. The 4-bit RCA outputs its result on the “S[3:0]” and carry out, c_{out} , ports.

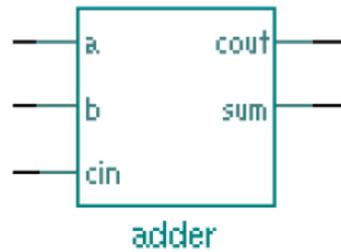


Figure 8-1

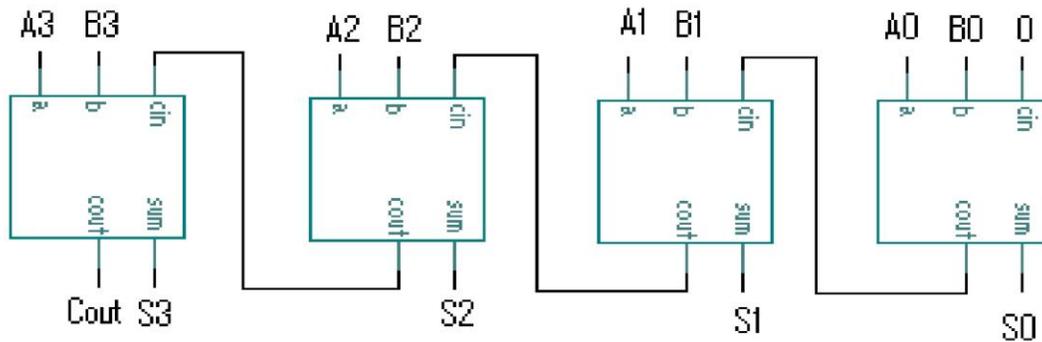


Figure 8-2. Structural description of 4-bit RCA using four full adders.

VHDL CODE

The following file shows a portion of the VHDL description of the full adder component. This is the simplest model of the full adder.

```
-- FA.vhd : full adder component
library ieee;
use ieee.std_logic_1164.all;

entity FA is
    port(a, b : in std_logic;
         cin : in std_logic;
         cout : out std_logic;
         sum : out std_logic);
end entity FA;

architecture df_FA of FA is
begin
    cout <= ???;
    sum <= a xor b xor cin;
end architecture df_FA;
```

The VHDL statement for calculating “sum” is pretty straight forward; it’s just addition of the three input operands. The Boolean equation for “sum” can be written as: **sum = a ⊕ b ⊕ cin**

A carry output is asserted only when two or more inputs are asserted. You'll need to generate the minimized equation for "cout" and insert the VHDL form of this equation in the FA component.

The following file shows a portion of the VHDL description of the structural model of the 4-bit RCA, using four of the full adder components, shown in the previous VHDL file.

```
-- file RCA.vhd

library ieee;
use ieee.std_logic_1164.all;

entity RCA is
    port(a, b : in std_logic_vector(3 downto 0);
          cout : out std_logic;
          sum : out std_logic_vector(3 downto 0));
end entity RCA;

architecture struct_RCA of RCA is

    signal cin: std_logic_vector(3 downto 0);
    component FA is
        port(a, b : in std_logic;
              cin : in std_logic;
              cout : out std_logic;
              sum : out std_logic);
    end component;

begin

    cin(0) <= '0';

    FA0 : FA
        port map(a(0), b(0), cin(0), cin(1), sum(0));

    FA1 : FA
        port map(???, ??, ??, ??, ??);

    FA2 : FA
        port map(???, ??, ??, ??, ??);

    FA3 : FA
        port map(???, ??, ??, ??, ??);

end architecture struct_RCA;
```

This structural model of a 4-bit RCA instantiates four FA components by "port mapping" them. The first adder's carry in is set to '0' as shown above. For the rest of the full adders, the carry input is the carry output of the previous full adder. Hence, the carries ripple up in this circuit, which gives it the name, ripple-carry adder. This 4-bit RCA has two input ports 'a' & 'b' each of 4-bit widths. You'll need to "port map" the remaining three full adder components to complete the design.

In short, the sum for the 4-bit RCA is contributed by four full adders and the carry is propagated from the first through fourth full adder to the output carry port. It is to be noted that, the addition is described as " $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{cin}$ " which means the carry generation in between the additions is taken into consideration at every step of addition operation.

Simulation with *ModelSim*:

This section will explain the design of test bench required for simulating the 4-bit RCA developed in the previous section.

The testbench will cycle through all 256 possible combinations of the 4-bit 'a' and 'b' inputs, allowing the user to check the accuracy of the outputs. Instead of writing all the input combinations in the test bench code, a simple for loop is used to generate all input combinations. Furthermore, an incorrect signal is used to automatically check the output correctness, such that the designer only needs to look at the final value of incorrect to tell if the circuit is functioning properly (i.e. at the end of the simulation, if incorrect is '0' then the RCA is working correctly; otherwise, if the RCA is not working correctly, then incorrect will be asserted when the output is not correct, and will remain asserted throughout the rest of the simulation).

You need to write your own testbench to exhaustively test the 8 input combinations of the FA component. Since there are only 8 combinations, it would probably be easiest to write the eight combinations sequentially, without using a loop, and check the results manually (i.e.

```
a <= '0';
b <= '0';
cin <= '0';

wait for 50 ns;
a <= '0';

b <= '0';
cin <='0';
wait for 50 ns;

.
.
.
)
```

To allow the operator to see the output resulting from each set of inputs, a delay of 50 ns is given between each test using a wait statement. The value of 50 ns is selected somewhat randomly, since the simulation does not include gate propagation delay.

Using a testbench is much easier than manually entering each combination by hand. A testbench also automatically documents your testing procedure for others and allows you to easily repeat your test as needed in the future.

Testbench for RCA is as follows:

```
-- tb_RCA.vhd :
testbench library ieee;
use ieee.std_logic_1164.all; use
ieee.std_logic_arith.all; use
ieee.std_logic_unsigned.all;

entity tb_RCA is
end entity tb_RCA;

architecture testbench of tb_RCA is

component RCA is
    port(a, b : in std_logic_vector(3 downto 0);
        cout : out std_logic;
        sum : out std_logic_vector(3 downto
0)); end component;

signal a, b, sum : std_logic_vector(3 downto 0);
signal cout : std_logic;
signal incorrect : std_logic := '0';
signal input_vec: std_logic_vector(7 downto 0);
signal test_a, test_b, test_sum: std_logic_vector(4 downto 0);

begin

UUT : RCA
    port map(a, b, cout, sum);

a <= input_vec(3 downto 0);
b <= input_vec(7 downto 4);

test_a <= '0' & input_vec(3 downto 0);
test_b <= '0' & input_vec(7 downto 4);
test_sum <= test_a + test_b;

inputs : process
begin

input_vec <= "00000000";
for i in 0 to 255 loop

    wait for 50 ns;

    if (cout /= test_sum(4)) or (sum /= test_sum(3 downto 0)) then
incorrect <= '1';

    end if;

    Input_vec <= input_vec + 1;
end loop;

wait;

end process inputs;

end architecture testbench;

configuration cfg_tb_RCA of tb_RCA is
    for testbench

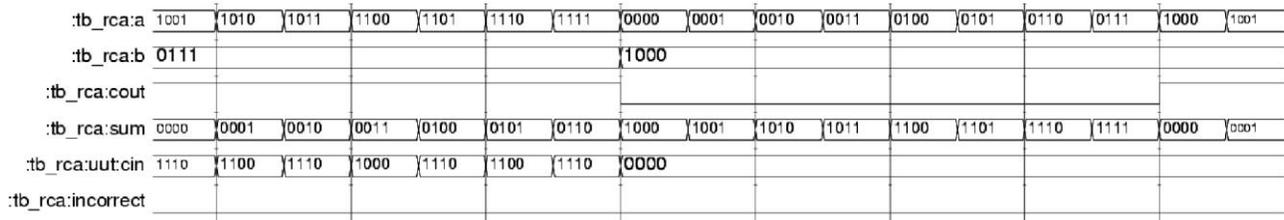
        for UUT: RCA
        end for;

    end for;
end cfg_tb_RCA;
```

To this end, it is easiest to also write a simulation macro to control the simulation (i.e. specify what external and internal signals you wish to view and how long you want to run the simulation). The simulation macro below views the external I/Os, a, b, c_{out} and sum, the internal c_{in} signal, and the testbench's *incorrect* signal. The run `-all` command runs the simulation indefinitely, until a `wait;` statement is encountered, like at the end of the testbench. You'll need to write your own simulation macro for testing the FA component. The simulation macro for testing the RCA component is given below.

```
-- RCA.do : simulation macro add
wave a
add wave b add
wave cout add wave
sum
add wave :tb_RCA:UUT:cin add
wave incorrect
run -all
```

Here's what a portion of the simulation results should look like for the 4-bit RCA. You can zoom in and out using the *zoom* pulldown menu.



Following gives the step by step process:

Procedure

1. Create a project named lab8
2. Add existing file including FA.vhd, RCA.vhd, tb_RCA.vhd and tb_FA.vhd in this project.
3. Complete the FA.vhd, RCA.vhd and tb_RCA.vhd.
4. Compile all vhd files.
5. Simulate tb_FA.vhd. From the *Tools* pulldown menu select *TCL -> Execute Macro* and choose the simulation macro you wrote for the FA component (FA.do). This will run the simulation, and bring up the waveform window where you can view the results.
6. Simulate tb_RCA.vhd. This time choose RCA.do.
7. Once you checked all input combinations and are certain your FA design works correctly, make a hardcopy of the wave window, and turn this in along with a hardcopy of your VHDL code, testbench, and simulation macro.