10. Testbenches

10.1. Introduction

In previous chapters, we generated the simulation waveforms using modelsim, by providing the input signal values manually; if the number of input signals are very large and/or we have to perform simulation several times, then this process can be quite complex, time consuming and irritating. Suppose input is of 10 bit, and we want to test all the possible values of input i.e. $2^{10} - 1$, then it is impossible to do it manually. In such cases, testbenches are very useful; also, tested design more reliable and prefer by the other clients as well. Further, with the help of testbenches, we can generate results in the form of csv (comma separated file), which can be used by other softwares for further analysis e.g. Python, Excel and Matlab etc.

Since testbenches are used for simulation purpose only (not for synthesis), therefore full range of VHDL constructs can be used e.g. keywords 'assert', 'report' and 'for loops' etc. can be used for writing testbenches.

Modelsim-project is created in this chapter for simulations, which allows the relative path to the files with respect to project directory as shown in Section 10.2.5. Simulation can be run without creating the project, but we need to provide the full path of the files as shown in Lines 30-34 of Listing 10.5.

Lastly, mixed modeling is not supported by Altera-Modelsim-starter version, i.e. Verilog designs with VHDL and vice-versa can not be compiled in this version of Modelsim.

10.2. Testbench for combinational circuits

In this section, various testbenches for combinational circuits are shown, whereas testbenches for sequential circuits are discussed in next section. For simplicity of the codes and better understanding, a simple half adder circuit is tested using various simulation methods.

10.2.1. Half adder

Listing 10.1 shows the VHDL code for the half adder which is tested using different ways,

Listing 10.1 Half adder¶

```vhdl
-- half_adder.vhd

library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
  port (a, b : in std_logic;
        sum, carry : out std_logic
    );
end half_adder;

architecture arch of half_adder is
begin
  sum <= a xor b;
  carry <= a and b;
end arch;
```

10.2.2. Simple testbench

Note that, testbenches are written in separate VHDL files as shown in Listing 10.2. Simplest way to write a testbench, is to invoke the 'design for testing' in the testbench and provide all the input values in the file, as explained below,

Explanation Listing 10.2

In this listing, a testbench with name 'half_adder_simple_tb' is defined at Lines 7-8. Note that, entity of testbench is always empty i.e. no ports are defined in the entity (see Lines 7-8). Then 4 signals are defined i.e. a, b, sum and carry (Lines 11-12) inside the architecture body; these signals are then connected to actual half adder design using structural modeling (see Line 15). Lastly, different values are assigned to input signals e.g. 'a' and 'b' at lines 16 and 17 respectively.

In Line 22, value of 'a' is 0 initially (at 0 ns), then it changes to '1' at 20 ns and again changes to '0' **at** 40 ns (**do not confuse with after 40 ns, as after 40 ns is with respect to 0 ns, not with respect to 20 ns**). Similarly, the values of 'a' becomes '0' and '1' **at** 40 and 60 ns respectively. In the same way value of 'b' is initially '0' and change to '1' at 40 ns at Line 23. In this way 4 possible combination are generated for two bits ('ab') i.e. 00, 01, 10 and 11 as shown in Fig. 10.1; also corresponding outputs, i.e. sum and carry, are shown in the figure.

To generate the waveform, first compile the 'half_adder.vhd and then 'half_adder_simple_tb.vhd' (or compile both the file simultaneously.). Then simulate the half_adder_simple_tb.vhd file. Finally, click on 'run all' button (which will run the simulation to maximum time i.e. 60 ns here at Line 22)and then click then 'zoom full' button (to fit the waveform on the screen), as shown in Fig. 10.1.

Problem: Although, the testbench is very simple, but input patterns are not readable. By using the **process statement** in the testbench, we can make input patterns more readable along with inclusion of various other features e.g. report generation etc., as shown in next section.

Listing 10.2 Simple testbench for half adder¶

```
1   -- half_adder_simple_tb.vhd
2
3   library ieee;
4   use ieee.std_logic_1164.all;
5
6
7   entity half_adder_simple_tb is
8   end half_adder_simple_tb;
9
10  architecture tb of half_adder_simple_tb is
11      signal a, b : std_logic;  -- inputs
12      signal sum, carry : std_logic;  -- outputs
13  begin
14      -- connecting testbench signals with half_adder.vhd
15      UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry =>
16  carry);
17
18      -- inputs
19      -- 00 at 0 ns
20      -- 01 at 20 ns, as b is 0 at 20 ns and a is changed to 1 at 20 ns
21      -- 10 at 40 ns
22      -- 11 at 60 ns
23      a <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
24      b <= '0', '1' after 40 ns;
    end tb ;
```

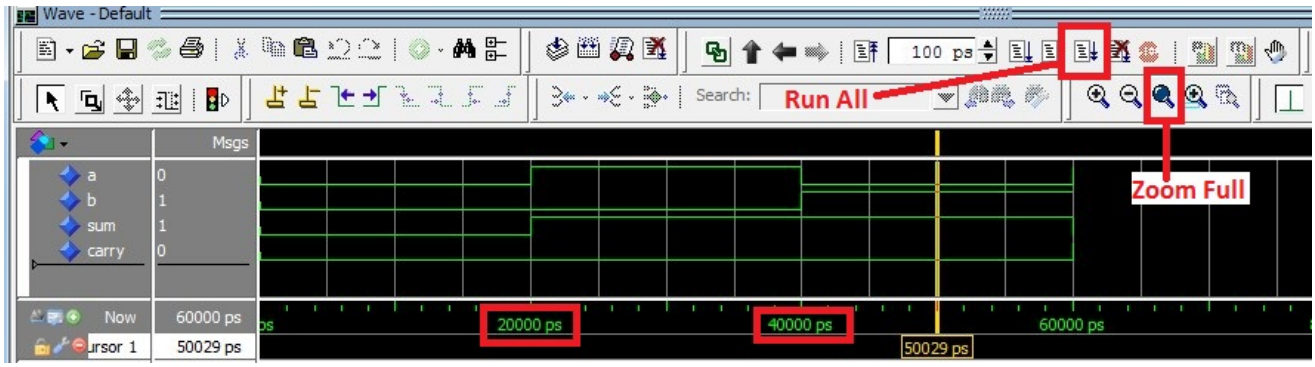

Fig. 10.1 Simulation results for Listing 10.2

10.2.3. Testbench with process statement

In Listing 10.3, process statement is used in the testbench; which includes the input values along with the corresponding output values. If the specified outputs are not matched with the output generated by half-adder, then errors will be generated. **Note that, process statement is written without the sensitivity list**.

Explanation Listing 10.3

The listing is same as previous listing till Line 15, and then process statement is used to define the input patterns, which can be seen at lines 20-21 (00), 27-28 (01), 33-34 (10) and 39-40 (11). Further, expected outputs are shown below these lines e.g. line 23 shows that the sum is 0 and carry is 0 for input 00; and if the generated output is different from these values, e.g. error is generated by line 50 for input pattern 01 as shown in Fig. 10.3; as sum generated by half_adder for line 46-47 is 1, whereas expected sum is defined as 0 for this combination at line 49. Note that, the adder output is correct, whereas the expected value is entered incorrectly; and error is displayed on 'transcript window' of modelsim.

Also, 'period' is defined as 20 ns at Line 18; and then used after each input values e.g line 22, which indicates that input will be displayed for 20 ns before going to next input values (see in Fig. 10.3).

Listing 10.3 Testbench with process statement¶

```
1    half_adder_process_tb.vhd
```

```
-- half_adder_process_tb.vhd

library ieee;
use ieee.std_logic_1164.all;


entity half_adder_process_tb is
end half_adder_process_tb;

architecture tb of half_adder_process_tb is
    signal a, b : std_logic;
    signal sum, carry : std_logic;
begin
    -- connecting testbench signals with half_adder.vhd
    UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry =>
carry);

    tb1 : process
        constant period: time := 20 ns;
        begin
            a <= '0';
            b <= '0';
            wait for period;
            assert ((sum = '0') and (carry = '0'))  -- expected output
            -- error will be reported if sum or carry is not 0
            report "test failed for input combination 00" severity error;

            a <= '0';
            b <= '1';
            wait for period;
            assert ((sum = '1') and (carry = '0'))
            report "test failed for input combination 01" severity error;

            a <= '1';
            b <= '0';
            wait for period;
            assert ((sum = '1') and (carry = '0'))
            report "test failed for input combination 10" severity error;

            a <= '1';
            b <= '1';
            wait for period;
            assert ((sum = '0') and (carry = '1'))
            report "test failed for input combination 11" severity error;

            -- Fail test
            a <= '0';
            b <= '1';
            wait for period;
            assert ((sum = '0') and (carry = '1'))
            report "test failed for input combination 01 (fail test)" severity
error;


            wait; -- indefinitely suspend process
        end process;
end tb;
```

Fig. 10.2 Simulation results for Listing 10.3

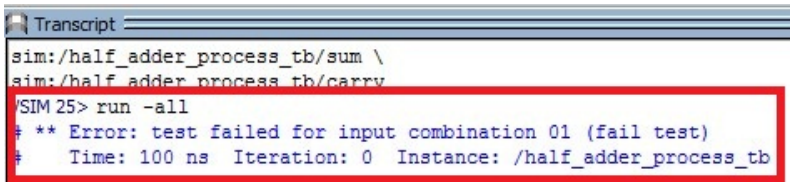

Fig. 10.3 Error generated by Listing 10.3

10.2.4. Testbench with look-up table

The inputs patterns can be defined in the form of look-table as well as shown in Listing 10.4, instead of define separately at different location as done in Listing 10.3 e.g. at lines 20 and 27 etc.

Explanation Listing 10.4

Basic concept of this Listing is similar to Listing 10.3 but written in different style. Testbench with lookup table can be written using three steps as shown below,

- **Define record** : First we need to define a record which contains the all the possible columns in the look table. Here, there are four possible columns i.e. a, b, sum and carry, which are defined in record at Lines 15-18.
- **Create lookup table** : Next, we need to define the lookup table values, which is done at Lines 20-28. Here positional method is used for assigning the values to columns (see line 22-27); further, name-association method can also be used as shown in the comment at Line 23.
- **Assign values to signals** : Then the values of the lookup table need to be assigned to half_adder entity (one by one). For this 'for loop' is used at line 35, which assigns the values of ''test-vector's 'a' and 'b' '' to signal 'a' and 'b' (see comment at Line 36 for better understanding). Similarly, expected values of sum and carry are generated at Lines 41-44. Lastly, report is generated for wrong outputs at Lines 46-50.

The simulations results and reported-error are shown in Fig. 10.4 and Fig. 10.5 respectively.

Listing 10.4 Testbench with look-up table¶

```
1    half_adder_lookup_tb.vhd
```

```
-- half_adder_lookup_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

entity half_adder_lookup_tb is
end half_adder_lookup_tb;

architecture tb of half_adder_lookup_tb is

    signal a, b : std_logic; -- input
    signal sum, carry : std_logic; -- output

    -- declare record type
    type test_vector is record
        a, b : std_logic;
        sum, carry : std_logic;
    end record;

    type test_vector_array is array (natural range <>) of test_vector;
    constant test_vectors : test_vector_array := (
        -- a, b, sum , carry   -- positional method is used below
        ('0', '0', '0', '0'), -- or (a => '0', b => '0', sum => '0', carry =>
'0')
        ('0', '1', '1', '0'),
        ('1', '0', '1', '0'),
        ('1', '1', '0', '1'),
        ('0', '1', '0', '1')  -- fail test
        );

begin
    UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry =>
carry);

    tb1 : process
    begin
        for i in test_vectors'range loop
            a <= test_vectors(i).a;  -- signal a = i^th-row-value of
test_vector's a
            b <= test_vectors(i).b;

            wait for 20 ns;

            assert (
                        (sum = test_vectors(i).sum) and
                        (carry = test_vectors(i).carry)
                    )

            -- image is used for string-representation of integer etc.
            report  "test_vector " & integer'image(i) & " failed " &
                    " for input a = " & std_logic'image(a) &
                    " and b = " & std_logic'image(b)
                    severity error;
        end loop;
        wait;
    end process;

end tb;
```

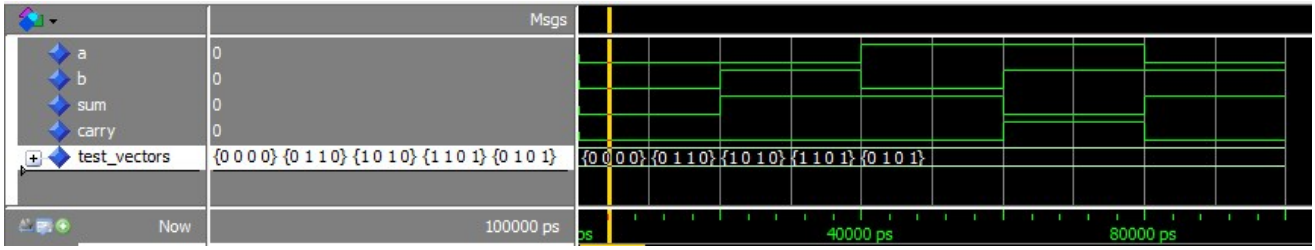

Fig. 10.4 Simulation results for Listing 10.4

```
VSIM 40> run -all
# ** Error: test_vector 4 failed  for input a = '0' and b = '1'
#    Time: 100 ns  Iteration: 0  Instance: /half_adder_lookup_tb
```

Fig. 10.5 Error generated by Listing 10.4

10.2.5. Read data from file

In this section, data from file 'read_file_ex.txt' is read and displayed in simulation results. Date stored in the file is shown in Fig. 10.6.

```
0 0 11
0 1 01
1 0 11
```

Fig. 10.6 Data in file 'read_file_ex.txt'

Explanation Listing 10.5

To read the file, first we need to define a buffer of type 'text', which can store the values of the file in it, as shown in Line 17; file is open in read-mode and values are stored in this buffer at Line 32.

Next, we need to define the variable to read the value from the buffer. Since there are 4 types of values (i.e. a, b, c and spaces) in file 'read_file_ex.txt', therefore we need to define 4 variables to store them, as shown in Line 24-26. Since, variable c is of 2 bit, therefore Line 25 is 2-bit vector; further, for spaces, variable of character type is defined at Line 26.

Then, values are read and store in the variables at Lines 36-42. Lastly, these values are assigned to appropriate signals at Lines 45-47. Finally, file is closed at Line 52. The simulation results of the listing are show in Fig. 10.7.

Listing 10.5 Read data from file¶

```
1    read_file_ex.vhd
```

```vhdl
-- read_file_ex.vhd


library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all; -- require for writing/reading std_logic etc.

entity read_file_ex is
end read_file_ex;

architecture tb of read_file_ex is
    signal a, b : std_logic;
    signal c : std_logic_vector(1 downto 0);

    -- buffer for storing the text from input read-file
    file input_buf : text;  -- text is keyword

begin

tb1 : process
    variable read_col_from_input_buf : line; -- read lines one by one from
input_buf

    variable val_col1, val_col2 : std_logic; -- to save col1 and col2 values of
1 bit
    variable val_col3 : std_logic_vector(1 downto 0); -- to save col3 value of 2
bit
    variable val_SPACE : character;  -- for spaces between data in file

    begin

        -- if modelsim-project is created, then provide the relative path of
        -- input-file (i.e. read_file_ex.txt) with respect to main project
folder
        file_open(input_buf, "VHDLCodes/input_output_files/read_file_ex.txt",
read_mode);
        -- else provide the complete path for the input file as show below
        -- file_open(input_buf,
"E:/VHDLCodes/input_output_files/read_file_ex.txt", read_mode);

        while not endfile(input_buf) loop
          readline(input_buf, read_col_from_input_buf);
          read(read_col_from_input_buf, val_col1);
          read(read_col_from_input_buf, val_SPACE);           -- read in the
space character
          read(read_col_from_input_buf, val_col2);
          read(read_col_from_input_buf, val_SPACE);           -- read in the
space character
          read(read_col_from_input_buf, val_col3);

          -- Pass the read values to signals
          a <= val_col1;
          b <= val_col2;
          c <= val_col3;

          wait for 20 ns;  --  to display results for 20 ns
        end loop;

        file_close(input_buf);
        wait;
    end process;
end tb ; -- tb
```

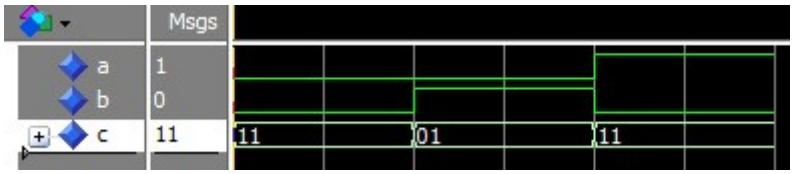

Fig. 10.7 Simulation results of Listing 10.5

10.2.6. Write data to file

In this part, different types of values are defined in Listing 10.6 and then stored in the file. Here, only 'write_mode' is used for writing the data to file (not the 'append_mode').

Explanation Listing 10.6

To write the data to the file, first we need to define a buffer, which will load the file on the simulation environment for writing the data during simulation, as shown in Line 15 (buffer-defined) and Line 27 (load the file to buffer).

Next, we need to define a variable, which will store the values to write into the buffer, as shown in Line 19. In the listing, this variable stores three types of value i.e. strings (Lines 31 and 34 etc.), signal 'a' (Line 35) and variable 'b' (Line 37).

Note that, two keyword are used for writing the data into the file i.e. 'write' and 'writeline'. 'write' keyword store the values in the 'write_col_to_output_buf' and 'writeline' writes the values in the file. Remember that, all the 'write' statements before the 'writeline' will be written in same line e.g. Lines 34-37 will be written in same line as shown in Fig. 10.8. Lastly, the simulation result for the listing is shown in Fig. 10.9.

Listing 10.6 Write data to fil¶

```
1    write_file_ex.vhd
```

```
-- write_file_ex.vhd


library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all; -- require for writing std_logic etc.

entity write_file_ex is
end write_file_ex;

architecture tb of write_file_ex is
    signal a : std_logic;

    file output_buf : text;  -- text is keyword
begin

    tb1 : process
        variable write_col_to_output_buf : line; -- line is keyword
        variable b : integer := 40;
        begin
            a <= '1';  -- assign value to a
            wait for 20 ns;

            -- if modelsim-project is created, then provide the relative path of
            -- input-file (i.e. read_file_ex.txt) with respect to main project
folder
            file_open(output_buf,
"VHDLCodes/input_output_files/write_file_ex.txt",  write_mode);
            -- else provide the complete path for the input file as show below
            --file_open(output_buf,
"E:/VHDLCodes/input_output_files/write_file_ex.txt",  write_mode);

            write(write_col_to_output_buf, string'("Printing values"));
            writeline(output_buf, write_col_to_output_buf);  -- write in line 1

            write(write_col_to_output_buf, string'("a = "));
            write(write_col_to_output_buf, a);
            write(write_col_to_output_buf, string'(", b = "));
            write(write_col_to_output_buf, b);
            writeline(output_buf, write_col_to_output_buf);    -- write in new
line 2

            write(write_col_to_output_buf, string'("Thank you"));
            writeline(output_buf, write_col_to_output_buf);   -- write in new
line 3

            file_close(output_buf);
            wait; -- indefinitely suspend process
        end process;
end tb ; -- tb
```

```
Printing values
a = 1, b = 40
Thank you
```

Fig. 10.8 Data in file 'write_file_ex.txt'

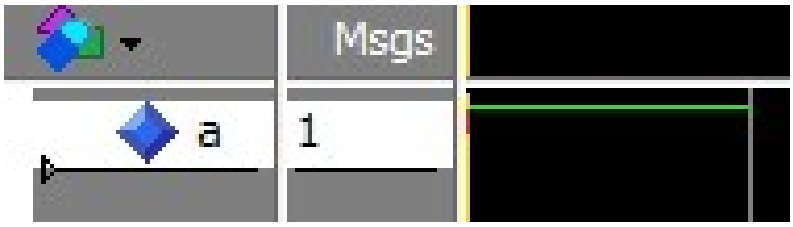

Fig. 10.9 Simulation results of Listing 10.6

10.2.7. Half adder testing using CSV file

In this section, both read and write operations are performed in Listing 10.7. Further, csv file is used for read and write operations. Content of input and output files are shown in Fig. 10.11 and Fig. 10.12 respectively.

Please read Listing 10.5 and Listing 10.6 to understand this part, as only these two listings are merged together here.

Addition features added to listing are shown below,

- Lines 63-64 are added to skip the header row, i.e. any row which does not start with boolean-number(see line 42).
- Also, error will be reported for value 'b' if it is not the boolean. Similarly, this functionality can be added to other values as well.
- Lastly, errors are reported in CSV file at Lines 96-109. This can be seen in Fig. 10.12. It's always easier to find the location of error using csv file as compare to simulation waveforms (try to find the errors using Fig. 10.12 and compare it with Fig. 10.12).

Listing 10.7 Half adder testing using CSV file¶

```
1  -- read_write_file_ex.vhd
2
3  -- testbench for half adder,
4
5  -- Features included in this code are
6      -- inputs are read from csv file, which stores the desired outputs as well
7      -- outputs are written to csv file
8      -- actual output and calculated outputs are compared
9      -- Error message is displayed in the file
10     -- header line is skipped while reading the csv file
11
12
13 library ieee;
14 use ieee.std_logic_1164.all;
15 use std.textio.all;
16 use ieee.std_logic_textio.all; -- require for writing/reading std_logic etc.
17
18 entity read_write_file_ex is
19 end read_write_file_ex;
20
21 architecture tb of read_write_file_ex is
22     signal a, b : std_logic;
23     signal sum_actual, carry_actual : std_logic;
24     signal sum, carry : std_logic;  -- calculated sum and carry by half_adder
25
26     -- buffer for storing the text from input and for output files
27     file input_buf : text;  -- text is keyword
```

```
    file output_buf : text;  -- text is keyword

begin
  UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry => carr

  tb1 : process
  variable read_col_from_input_buf : line; -- read lines one by one from input_buf
  variable write_col_to_output_buf : line; -- write lines one by one to output_buf

  variable buf_data_from_file : line;  -- buffer for storind the data from input r
file
  variable val_a, val_b : std_logic;
  variable val_sum, val_carry: std_logic;
  variable val_comma : character;  -- for commas between data in file

  variable good_num : boolean;
  begin

  -- ##########################################################################
   -- Reading data

      -- if modelsim-project is created, then provide the relative path of
      -- input-file (i.e. read_file_ex.txt) with respect to main project folder
      file_open(input_buf, "VHDLCodes/input_output_files/half_adder_input.csv",
read_mode);
      -- else provide the complete path for the input file as show below
      -- file_open(input_buf, "E:/VHDLCodes/input_output_files/read_file_ex.txt",
read_mode);

      -- writing data
      file_open(output_buf, "VHDLCodes/input_output_files/half_adder_output.csv",
write_mode);

      write(write_col_to_output_buf,

string'("#a,b,sum_actual,sum,carry_actual,carry,sum_test_results,carry_test_result
      writeline(output_buf, write_col_to_output_buf);

      while not endfile(input_buf) loop
        readline(input_buf, read_col_from_input_buf);
        read(read_col_from_input_buf, val_a, good_num);
        next when not good_num;  -- i.e. skip the header lines

        read(read_col_from_input_buf, val_comma);           -- read in the space
character
        read(read_col_from_input_buf, val_b, good_num);
        assert good_num report "bad value assigned to val_b";

        read(read_col_from_input_buf, val_comma);           -- read in the space
character
        read(read_col_from_input_buf, val_sum);
        read(read_col_from_input_buf, val_comma);           -- read in the space
character
        read(read_col_from_input_buf, val_carry);

        -- Pass the variable to a signal to allow the ripple-carry to use it
        a <= val_a;
        b <= val_b;
        sum_actual <= val_sum;
        carry_actual <= val_carry;

        wait for 20 ns;  --  to display results for 20 ns

        write(write_col_to_output_buf, a);
        write(write_col_to_output_buf, string'(","));
        write(write_col_to_output_buf, b);
        write(write_col_to_output_buf, string'(","));
```

```
95            write(write_col_to_output_buf, sum_actual);
96            write(write_col_to_output_buf, string'(","));
97            write(write_col_to_output_buf, sum);
98            write(write_col_to_output_buf, string'(","));
99            write(write_col_to_output_buf, carry_actual);
100           write(write_col_to_output_buf, string'(","));
101           write(write_col_to_output_buf, carry);
102           write(write_col_to_output_buf, string'(","));
103
104           -- display Error or OK if results are wrong
105           if (sum_actual /= sum) then
106             write(write_col_to_output_buf, string'("Error,"));
107           else
108             write(write_col_to_output_buf, string'(","));  -- write nothing
109
110           end if;
111
112           -- display Error or OK based on comparison
113           if (carry_actual /= carry) then
114             write(write_col_to_output_buf, string'("Error,"));
115           else
116             write(write_col_to_output_buf, string'("OK,"));
117           end if;
118
119
120           --write(write_col_to_output_buf, a, b, sum_actual, sum, carry_actual, carr
              writeline(output_buf, write_col_to_output_buf);
          end loop;

          file_close(input_buf);
          file_close(output_buf);
          wait;
        end process;
      end tb ; -- tb
```

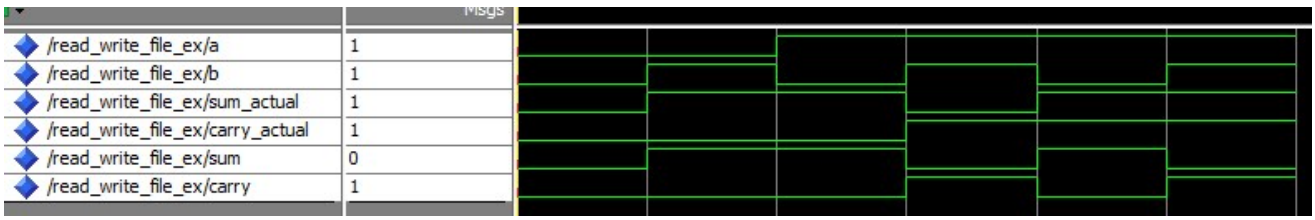

Fig. 10.10 Simulation results of Listing 10.7

#a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
1	0	1	1
1	1	1	1

Fig. 10.11 Content of input file 'half_adder_input.csv'

#a	b	sum_actual	sum	carry_actual	carry	sum_test_results	carry_test_results
0	0	0	0	0	0		OK
0	1	1	1	0	0		OK
1	0	1	1	0	0		OK
1	1	0	0	1	1		OK
1	0	1	1	1	0		Error
1	1	1	0	1	1	Error	OK

Fig. 10.12 Content of input file 'half_adder_output.csv'

10.3. Testbench for sequential circuits

In Section 10.2.3, we saw the use of process statement for writing the testbench for combination circuits. But, in the case of sequential circuits, we need clock and reset signals; hence two additional blocks are required. Since, clock is generated for complete simulation process, therefore it is defined inside the separate process statement. Whereas, reset signal is required only at the beginning of the operations, hence it is not defined inside the process statement. Rest of the procedures/methods for writing the testbenches for sequential circuits are same as the testbenches of the combinational circuits.

10.3.1. Simulation with infinite duration

In this section, we have created a testbench which will not stop automatically i.e. if we press the 'run all' button then the simulation will run forever, therefore we need to press the 'run' button as shown in Fig. 10.13.

Explanation Listing 10.8

> Listing 10.8 is the testbench for mod-M counter, which is discussed in Section 8.3.2. Here 'clk' signal is generated in the separate process block i.e. Lines 27-33; in this way, clock signal will be available throughout the simulation process. Further, reset signal is set to '1' in the beginning and then set to '0' in next clock cycle (Line 37). If there are further, inputs signals, then those signals can be defined in separate process statement, as discussed in combination circuits' testbenches.
>
> The simulation results are shown in Fig. 10.13, where counter values goes from 0 to 9 as M is set to 10 (i.e. A in hexadecimal). Further, use 'run' button for simulating the sequential circuits (instead of run-all), as shown in the figure.

Listing 10.8 Testbench with infinite duration for modMCounter.vhd¶

```
1    modMCounter_tb.vhd
```

```
1  -- modMCounter_tb.vhd
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity modMCounter_tb is
8  end modMCounter_tb;
9
10
11 architecture arch of modMCounter_tb is
12     constant M : integer := 10;
13     constant N : integer := 4;
14     constant T : time := 20 ns;
15
16     signal clk, reset : std_logic;  -- input
17     signal complete_tick : std_logic; -- output
18     signal count : std_logic_vector(N-1 downto 0);  -- output
19 begin
20
21     modMCounter_unit : entity work.modMCounter
22         generic map (M => M, N => N)
23         port map (clk=>clk, reset=>reset, complete_tick=>complete_tick,
24                     count=>count);
25
26     -- continuous clock
27     process
28     begin
29         clk <= '0';
30         wait for T/2;
31         clk <= '1';
32         wait for T/2;
33     end process;
34
35
36     -- reset = 1 for first clock cycle and then 0
37     reset <= '1', '0' after T/2;
38
39 end arch;
```

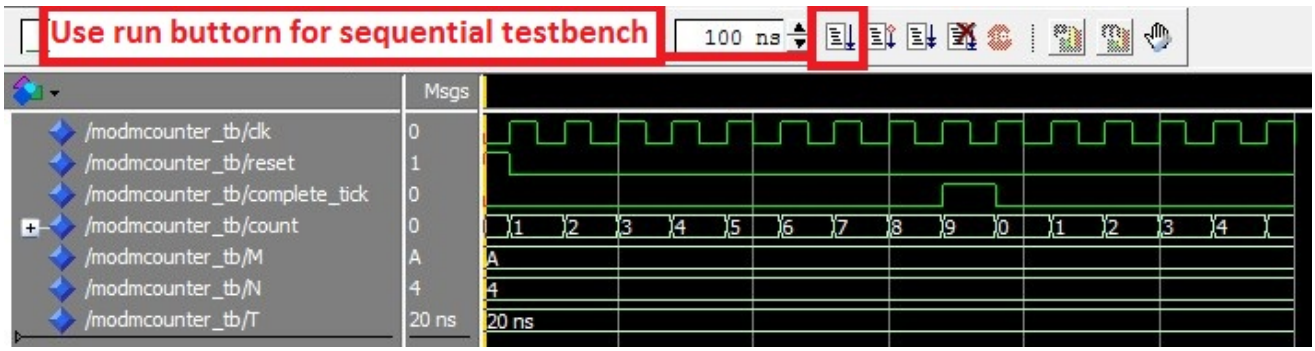

Fig. 10.13 Simulation results of Listing 10.8

10.3.2. Simulation for finite duration and save data

To run the simulation for the finite duration, we need to provide the 'number of clocks' for which we want to run the simulation, as shown in Line 23 of Listing 10.9. Then at Lines 47-52 are added to close the file after desired number of clocks i.e. 'num_of_clocks'. Also, the data is saved into the file, which is discussed in Section 10.2.6. Now, if we press the **run all** button, then the simulator will stop after 'num_of_clocks' cycles. **Note that, if the data is in 'signed or unsigned' format, then it can not be saved into the file. We need to change the**

data into other format e.g. 'integer', 'natural' or 'std_logic_vector' etc. before saving it into the file, as shown in Line 73. The simulation waveforms and saved results are shown in Fig. 10.14 and Fig. 10.15 respectively.

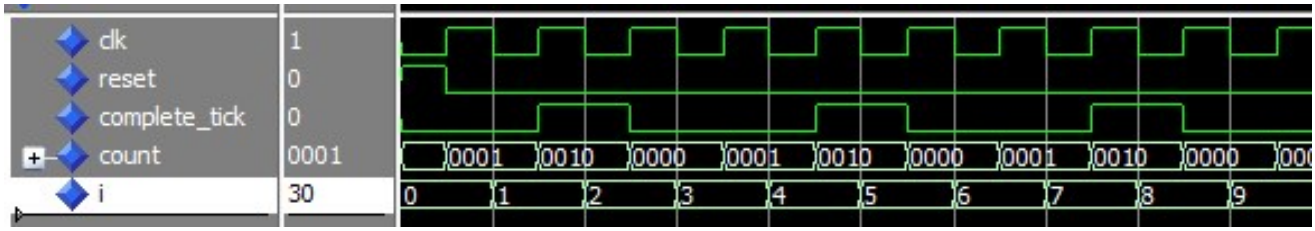

Fig. 10.14 Simulation results of Listing 10.9

clock_tick	count
0	0
0	1
1	2
0	0
0	1
1	2
0	0
0	1
1	2
0	0
0	1
1	2

Fig. 10.15 Partial view of saved data by Listing 10.9

Listing 10.9 Testbench with finite duration for modMCounter.vhd¶

```
1   -- modMCounter_tb2.vhd
2
3   library ieee;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6   use std.textio.all;
7   use ieee.std_logic_textio.all; -- require for std_logic etc.
8
9   entity modMCounter_tb2 is
10  end modMCounter_tb2;
11
12
13  architecture arch of modMCounter_tb2 is
14      constant M : integer := 3;  -- count upto 2 (i.e. 0 to 2)
15      constant N : integer := 4;
16      constant T : time := 20 ns;
17
18      signal clk, reset : std_logic;  -- input
19      signal complete_tick : std_logic; -- output
20      signal count : std_logic_vector(N-1 downto 0);  -- output
21
22      -- total samples to store in file
23      constant num_of_clocks : integer := 30;
24      signal i : integer := 0; -- loop variable
25      file output_buf : text; -- text is keyword
26
27  begin
28
```

```
    modMCounter_unit : entity work.modMCounter
        generic map (M => M, N => N)
        port map (clk=>clk, reset=>reset, complete_tick=>complete_tick,
                  count=>count);


    -- reset = 1 for first clock cycle and then 0
    reset <= '1', '0' after T/2;

    -- continuous clock
    process
    begin
        clk <= '0';
        wait for T/2;
        clk <= '1';
        wait for T/2;

        -- store 30 samples in file
        if (i = num_of_clocks) then
            file_close(output_buf);
            wait;
        else
            i <= i + 1;
        end if;
    end process;


    -- save data in file : path is relative to Modelsim-project directory
    file_open(output_buf, "input_output_files/counter_data.csv", write_mode);
    process(clk)
        variable write_col_to_output_buf : line; -- line is keyword
    begin
        if(clk'event and clk='1' and reset /= '1') then  -- avoid reset data
            -- comment below 'if statement' to avoid header in saved file
            if (i = 0) then
              write(write_col_to_output_buf, string'("clock_tick,count"));
              writeline(output_buf, write_col_to_output_buf);
            end if;

            write(write_col_to_output_buf, complete_tick);
            write(write_col_to_output_buf, string'(","));
            -- Note that unsigned/signed values can not be saved in file,
            -- therefore change into integer or std_logic_vector etc.
             -- following line saves the count in integer format
            write(write_col_to_output_buf, to_integer(unsigned(count)));
            writeline(output_buf, write_col_to_output_buf);
        end if;
    end process;
end arch;
```

10.4. Conclusion

In this chapter, we learn to write testbenches with different styles for combinational circuits. We saw the methods by which inputs can be read from the file and the outputs can be written in the file. Simulation results and expected results are compared and saved in the csv file and displayed as simulation waveforms; which demonstrated that locating the errors in csv files is easier than the simulation waveforms. Further, we saw the simulation of sequential circuits as well, which is slightly different from combination circuits; but all the methods of combinational circuit simulations can be applied to sequential circuits as well.