**CRN: 22840**

# <u>CET3136 – Logic Devices Programming</u>

Spring 2025

Experiment 6

***Test Benches***

*Performed By:*

**Anthony Paul Sevarino**

*Submitted to:*

**Prof. Ashley Evans**

**Department of Electrical & Computer Engineering Technology (ECET)**
**School of Engineering, Technology, and Advanced Manufacturing (ETAM)**
**Valencia College**

Date Submitted
02/20/2025

## Introduction

The intent of this lab is to implement a test bench for a combinatorial logic circuit, and test it against both a functioning and malfunctioning circuit. No form of hardware is required for this experiment, as it focuses on the software idea of *testbenching*. This is the processing of testing a circuit or system, without the need for installing it onto any hardware, both saving resources, and time. One of the major benefits of testbenching is that it allows for the direct failure analysis of each component of a system. The system used to demonstrate this subject is a 9-bit parity checker, utilizing a combinatorial behavioral coding style. Recall that *parity* is the determination of whether there are an even or odd number of ones in a binary value.

## LIST OF EQUIPMENT/PARTS/COMPONENTS/SOFTWARE

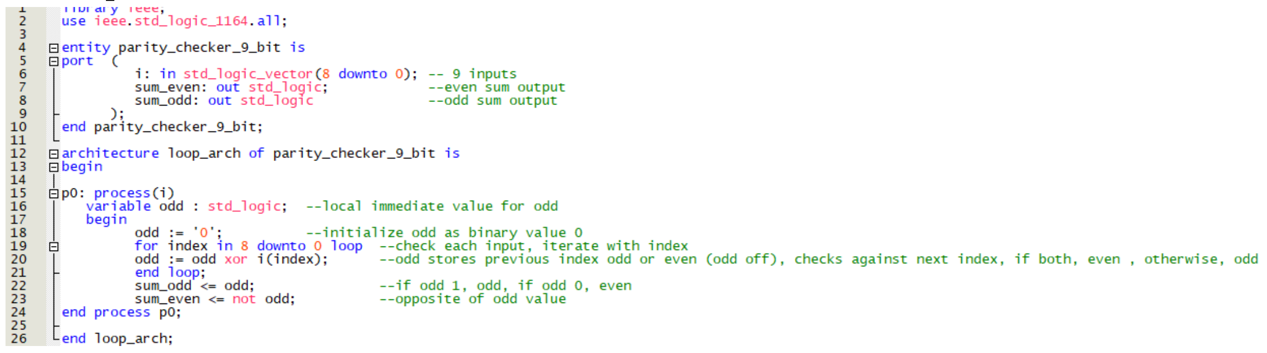
- Windows Desktop
- Quartus FPGA Design Software

## PROCEDURE / DISCUSSION

### *Part 1 – The Provided System*

As there is no system design or hardware implementation in this part, the study will branch directly into code explanation.

### *Code Explanation Part 1*



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

entity parity_checker_9_bit is
port   (
            i: in std_logic_vector(8 downto 0); -- 9 inputs
            sum_even: out std_logic;            --even sum output
            sum_odd: out std_logic              --odd sum output
         );
end parity_checker_9_bit;

architecture loop_arch of parity_checker_9_bit is
begin

p0: process(i)
   variable odd : std_logic;  --local immediate value for odd
   begin
            odd := '0';           --initialize odd as binary value 0
            for index in 8 downto 0 loop  --check each input, iterate with index
            odd := odd xor i(index);      --odd stores previous index odd or even (odd off), checks against next index, if both, even , otherwise, odd
            end loop;
            sum_odd <= odd;               --if odd 1, odd, if odd 0, even
            sum_even <= not odd;          --opposite of odd value
end process p0;

end loop_arch;
```

*Figure 1 – Provided code for Parity Checker*

---

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

entity parity_checker_9_bit  is
port       (
                              i: in std_logic_vector(8  downto  0); -- 9 inputs
                              sum_even: out std_logic;                          --even sum output
                              sum_odd: out std_logic                                   --odd sum output
                  );
end parity_checker_9_bit;
```

```
architecture  loop_arch of parity_checker_9_bit  is
begin

p0: process(i)
          variable odd : std_logic;   --local immediate value for odd
          begin
                              odd := '0';                                        --initialize odd as binary value 0
                              for index in 8 downto  0 loop          --check each input, iterate with index
                              odd := odd xor i(index);                --odd stores previous index odd or even (odd off),
checks against next index, if both, even , otherwise, odd
                              end loop;
                              sum_odd <= odd;                                                      --if odd 1, odd, if odd 0,
even
                              sum_even <= not odd;                                   --opposite of odd value
end process p0;

end loop_arch;
```

The entity portion of this code takes 9 inputs, in the form of a vector labelled *i*. There are two outputs, one for the even result, and one for the odd.

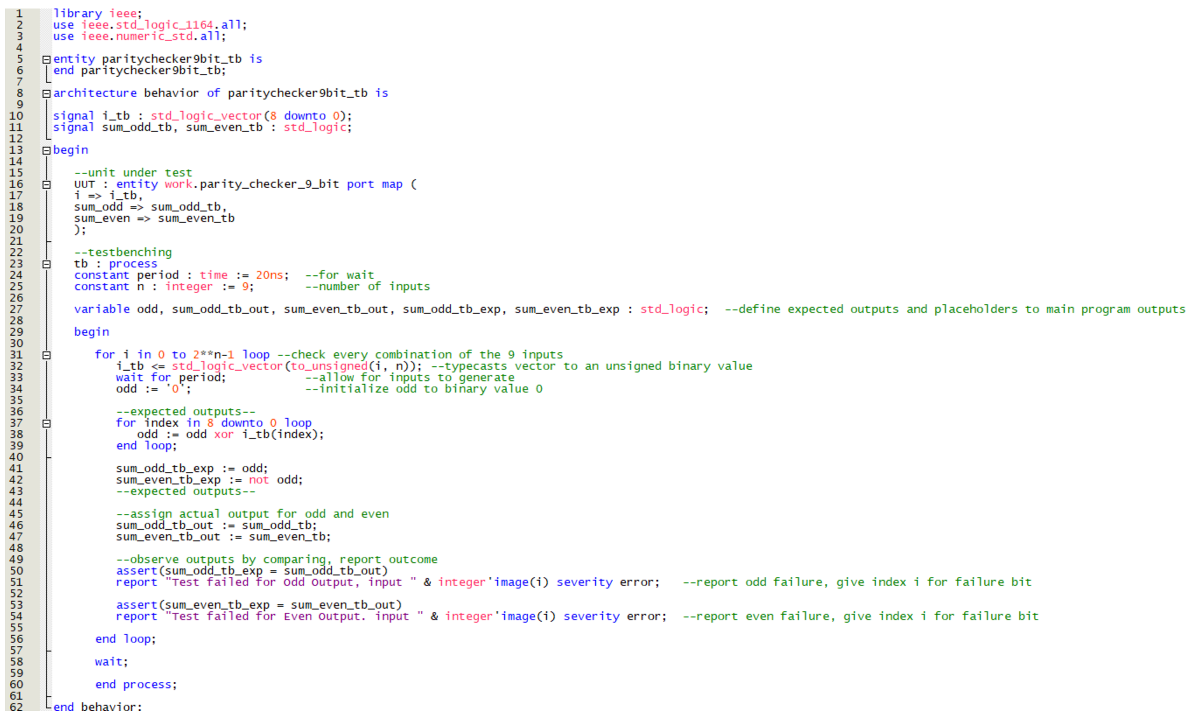
In the architecture section, a combinatorial behavioral style is used. A process is started, with a single standard logic variable defined, called *odd*. Within the process, odd is instantiated to be the binary value 0, and a *for loop* drives the logic test for the system. Within this loop, which iterates 9 times (once for each bit), the current odd state is checked against the next bit. In other words, at the start, odd (which is 0), is compared against the first index in the vector. This value is XOR'd and, if the index is 1, odd will become 1. Checking the next bit, since 1 is now stored in *odd*, if the next bit is 1, XOR states that it will become a zero, indicating that there is currently an even number of ones in the value. This will continue for each bit input. Finally, the odd result will obtain the value stored in *odd*, and the even result will store *not odd*. Thus, only one output can be logic high.

## *Part 2 – The Test Bench*

As there is no system design or hardware implementation in this part, the study will branch directly into code explanation.

### *Code Explanation Part 2*

Included below are both an image, and text inclusion of the complete code segment for part 2 of the experiment:



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

entity paritychecker9bit_tb is
end paritychecker9bit_tb;

architecture behavior of paritychecker9bit_tb is

signal i_tb : std_logic_vector(8 downto 0);
signal sum_odd_tb, sum_even_tb : std_logic;

begin

   --unit under test
   UUT : entity work.parity_checker_9_bit port map (
   i => i_tb,
   sum_odd => sum_odd_tb,
   sum_even => sum_even_tb
   );

   --testbenching
   tb : process
   constant period : time := 20ns;  --for wait
   constant n : integer := 9;       --number of inputs

   variable odd, sum_odd_tb_out, sum_even_tb_out, sum_odd_tb_exp, sum_even_tb_exp : std_logic;  --define expected outputs and placeholders to main program outputs

   begin

      for i in 0 to 2**n-1 loop --check every combination of the 9 inputs
         i_tb <= std_logic_vector(to_unsigned(i, n)); --typecasts vector to an unsigned binary value
         wait for period;            --allow for inputs to generate
         odd := '0';                 --initialize odd to binary value 0

         --expected outputs--
         for index in 8 downto 0 loop
            odd := odd xor i_tb(index);
         end loop;

         sum_odd_tb_exp := odd;
         sum_even_tb_exp := not odd;
         --expected outputs--

         --assign actual output for odd and even
         sum_odd_tb_out := sum_odd_tb;
         sum_even_tb_out := sum_even_tb;

         --observe outputs by comparing, report outcome
         assert(sum_odd_tb_exp = sum_odd_tb_out)
         report "Test failed for Odd Output, input " & integer'image(i) severity error;   --report odd failure, give index i for failure bit

         assert(sum_even_tb_exp = sum_even_tb_out)
         report "Test failed for Even Output. input " & integer'image(i) severity error;  --report even failure, give index i for failure bit

      end loop;

      wait;

      end process;

end behavior;
```

*Figure 2 -Completed Code for Testbench*

Below is the text included finalized code for the second part of this report.

---

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

entity paritychecker9bit_tb  is
end paritychecker9bit_tb;

architecture  behavior of paritychecker9bit_tb  is

signal i_tb : std_logic_vector(8 downto 0);
signal sum_odd_tb, sum_even_tb : std_logic;

begin

            --unit under test
            UUT : entity work.parity_checker_9_bit  port map (
            i => i_tb,
            sum_odd => sum_odd_tb,
            sum_even => sum_even_tb
            );

            --testbenching
            tb : process
            constant period : time := 20ns;      --for wait
            constant n : integer := 9;           --number of inputs

            variable odd, sum_odd_tb_out, sum_even_tb_out, sum_odd_tb_exp, sum_even_tb_exp : std_logic;           --
define expected outputs and placeholders to main program outputs
```

```
        begin

                for i in 0 to 2**n-1 loop --check every combination of the 9 inputs
                        i_tb <= std_logic_vector(to_unsigned(i, n));            --typecasts vector to an unsigned
binary value
                        wait for period;                                --allow for inputs to generate
                        odd := '0';                                             --initialize odd to binary
value 0

                        --expected outputs--
                        for index in 8 downto 0 loop
                                odd := odd xor i_tb(index);
                        end loop;

                        sum_odd_tb_exp := odd;
                        sum_even_tb_exp := not odd;
                        --expected outputs--

                        --assign actual output for odd and even
                        sum_odd_tb_out := sum_odd_tb;
                        sum_even_tb_out := sum_even_tb;

                        --observe outputs by comparing, report outcome
                        assert(sum_odd_tb_exp = sum_odd_tb_out)
                        report "Test failed for Odd Output, input " & integer'image(i) severity error; --report
odd failure, give index i for failure bit

                        assert(sum_even_tb_exp = sum_even_tb_out)
                        report "Test failed for Even Output. input " & integer'image(i) severity error;         --
report even failure, give index i for failure bit

                end loop;

                wait;

                end process;

end behavior;
```

Notice that the entity statement here is empty, This is because, since this is a testbench, there is no hardware inclusion necessary, and any inputs and outputs will be obtained from the main system.

The architectural section starts with two signal instantiations, one for the input vector, and two for the even and odd results. A *Unit Under Test* block is created based on the original system file, and a port map is used to assign the input and outputs.

Next, a process is created, called *tb*. This process starts with the definition of two constants, period and n (the first used for wait statements, and the second later used for a loop representing the number of inputs.) Following this, a set of variables are defined, those being *odd* (for the logic of the test) and even and odd results both for the original file output, and the expected outputs generated by the testbench. In this process, the testbench file will utilize its own logic to conduct testing, and compare its results to the original systems results to determine whether it will perform properly without needing to test it on the development board.

The process begins with a for loop which iterates $2^{n-1}$ times, representing every possible combination for the 9 bit input vector. The input vector is typecasted to an unsigned bit value to be analyzed later in the program, and a wait statement is used to allow for the inputs to generate (as there is no behavioral data between this and the rest of the loop to prevent the logic from running concurrently). *Odd* is set to be the binary value '0' here so that it resets for each input combination provided.

A nested for loop then iterates the current input combination 9 times, once for each bit, and uses the same logic as present in the main system to determine the parity of the value. This result is stored in the *Expected* odd and even results. Next, the output from the original system is assigned to the local variables within the testbench file.

Finally, an *assert/report* statement throws an error message if there are any failures present, noting whether it is an even or odd misalignment, as well as the particular bit combination it failed on. After the loop concludes, another wait statement is used to prevent the program from running infinitely.

## VALIDATION OF DATA

### *Lab Demonstration*

In the demonstration video provided below, an explanation as to the contents, as well as steps of the experiment are described and displayed.

https://youtu.be/FCbz49yBIXo

### *Functional Simulation Analysis*

In order to use this testbench to analyze the success of the original file, *Modelsim*, or another capable functional simulation analyzer, must be used to report any failures, and produce a system waveform. Below is the obtained result (presented as a zoomed version of the waveform in order to fit the dimensions of this report) as well as the transcript output:

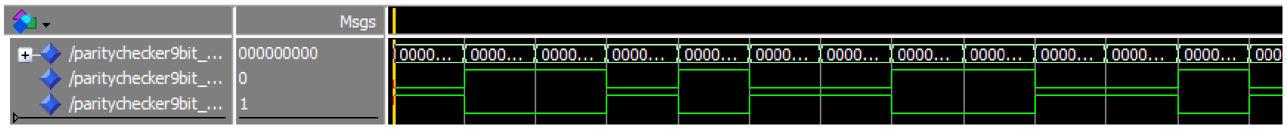
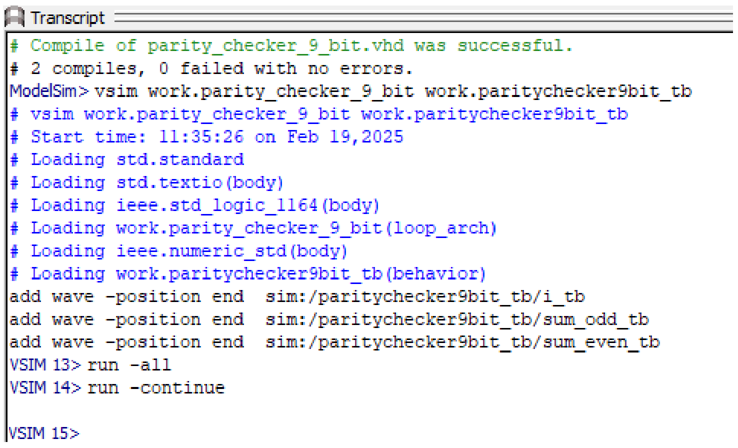


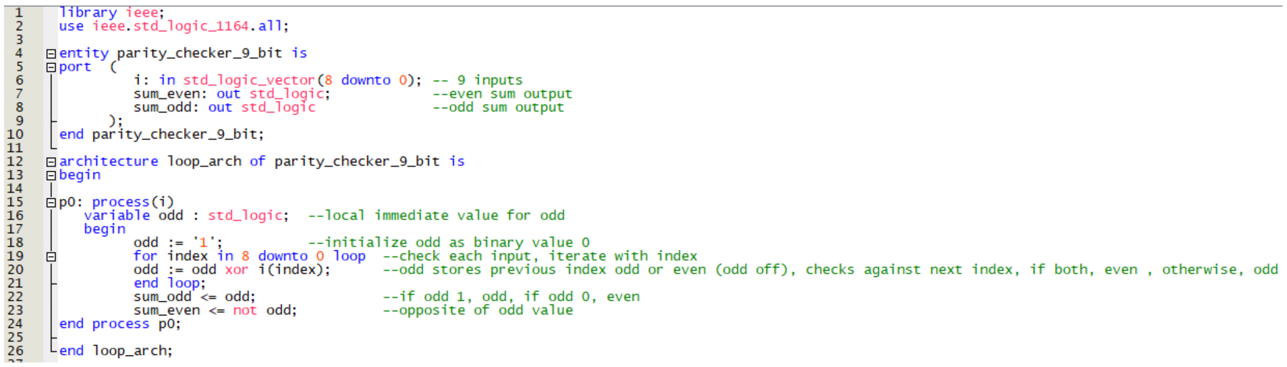
Figure 3 – Successful Testbench Waveform Result



```
Transcript
# Compile of parity_checker_9_bit.vhd was successful.
# 2 compiles, 0 failed with no errors.
ModelSim> vsim work.parity_checker_9_bit work.paritychecker9bit_tb
# vsim work.parity_checker_9_bit work.paritychecker9bit_tb
# Start time: 11:35:26 on Feb 19,2025
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.parity_checker_9_bit(loop_arch)
# Loading ieee.numeric_std(body)
# Loading work.paritychecker9bit_tb(behavior)
add wave -position end  sim:/paritychecker9bit_tb/i_tb
add wave -position end  sim:/paritychecker9bit_tb/sum_odd_tb
add wave -position end  sim:/paritychecker9bit_tb/sum_even_tb
VSIM 13> run -all
VSIM 14> run -continue

VSIM 15>
```

*Figure 4 - Transcript Result for Working Testbench*

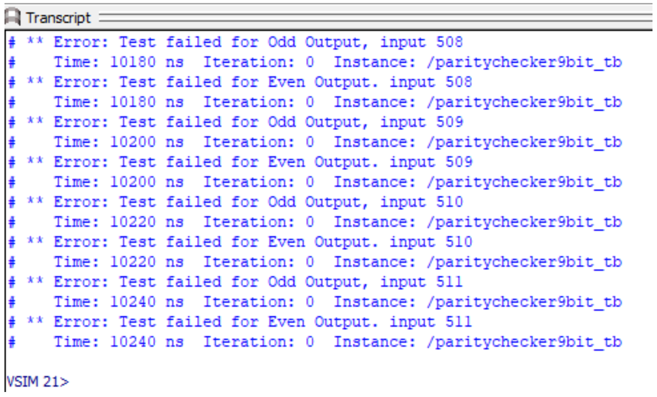
Note that after the RUN command, the VSIM simply continues to the next line, indicating no failure messages were thrown. While this is the intended result, and indicates the original program functions the same as the testbench does, it would be beneficial to intentionally break the original system to ensure the testbench would catch any failures. Thus, the *Odd* variable in the original system is changed from ‘0’ to ‘1’. This should completely flip every result the program could possibly have as it logically inverts every value. This code is provided below:



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

entity parity_checker_9_bit is
port    (
            i: in std_logic_vector(8 downto 0); -- 9 inputs
            sum_even: out std_logic;            --even sum output
            sum_odd: out std_logic              --odd sum output
        );
end parity_checker_9_bit;

architecture loop_arch of parity_checker_9_bit is
begin

p0: process(i)
    variable odd : std_logic;   --local immediate value for odd
    begin
            odd := '1';         --initialize odd as binary value 0
            for index in 8 downto 0 loop  --check each input, iterate with index
            odd := odd xor i(index);      --odd stores previous index odd or even (odd off), checks against next index, if both, even , otherwise, odd
            end loop;
            sum_odd <= odd;               --if odd 1, odd, if odd 0, even
            sum_even <= not odd;          --opposite of odd value
end process p0;

end loop_arch;
```

*Figure 5 - Altered Original System Code*

The files are saved, and recompiled within *Modelsim* in order to retest the systems. Upon conducting this process, the transcript throws errors explained within the testbench code, pictured below:

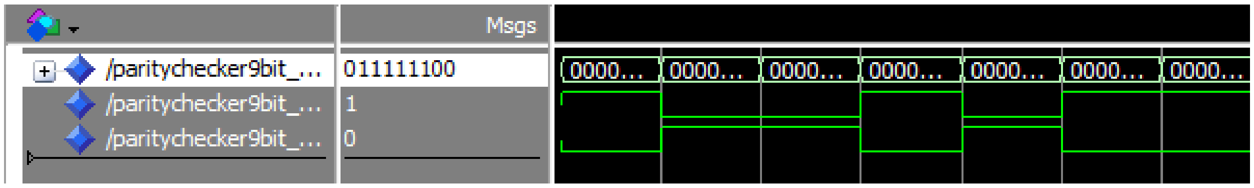


```
Transcript
# ** Error: Test failed for Odd Output, input 508
#    Time: 10180 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Even Output. input 508
#    Time: 10180 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Odd Output, input 509
#    Time: 10200 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Even Output. input 509
#    Time: 10200 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Odd Output, input 510
#    Time: 10220 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Even Output. input 510
#    Time: 10220 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Odd Output, input 511
#    Time: 10240 ns  Iteration: 0  Instance: /paritychecker9bit_tb
# ** Error: Test failed for Even Output. input 511
#    Time: 10240 ns  Iteration: 0  Instance: /paritychecker9bit_tb

VSIM 21>
```

*Figure 6 - Failures in Testbench Transcript*

Each line indicates a failure on either the odd or even result, as well as the combination which was failed (there are 511 even failures, and 511 odd failures, indicating each combination was tested as $2^{n-1} * 2$ (one for even and one for odd) is 512).

Also, analyzing the "failure" waveform and comparing it to the "success" waveform shows that they are exactly inverted:



## CONCLUSION

The experiment was conducted successfully and efficiently demonstrated the use of a testbench to check the functionality of a combinatorial behavioral circuit acting as a 9-bit parity checker. Including the precise failure messages throughout the code offers a unique insight into program functionality, and would allow for an engineer to efficiently repair a system. Testbenching itself proves to be a valuable tool that teams can use to produce systems which have a form of redundant verification, wherein two engineers attack a problem in their own way, ensuring that it is conducted properly. In the future, more advanced circuits should be used to enhance this fundamental understanding of the strengths behind testbenching, such as with structural based programs and systems.

## REFERENCES

[1] Professor Ashley Evans, *Logic Devices Programming Lab Manual*, 1st ed. Orlando, FL: Valencia College, 2025.