4C1: Integrated System Design

Practical 2: Self Checking Testbench

Rahul Seth

17302557

Introduction

The Practical aimed to develop a comprehensive testbench that included the following components:

1. **Stimulus Generator:** Generates the inputs for testing the parking lot counter.
2. **Monitor Module:** Compares the outputs from the parking lot counter with the expected outputs and gives a pass/fail criterion.
3. **Top Level Testbench Module:** Connects the stimulus generator, monitor module and the parking lot counter.

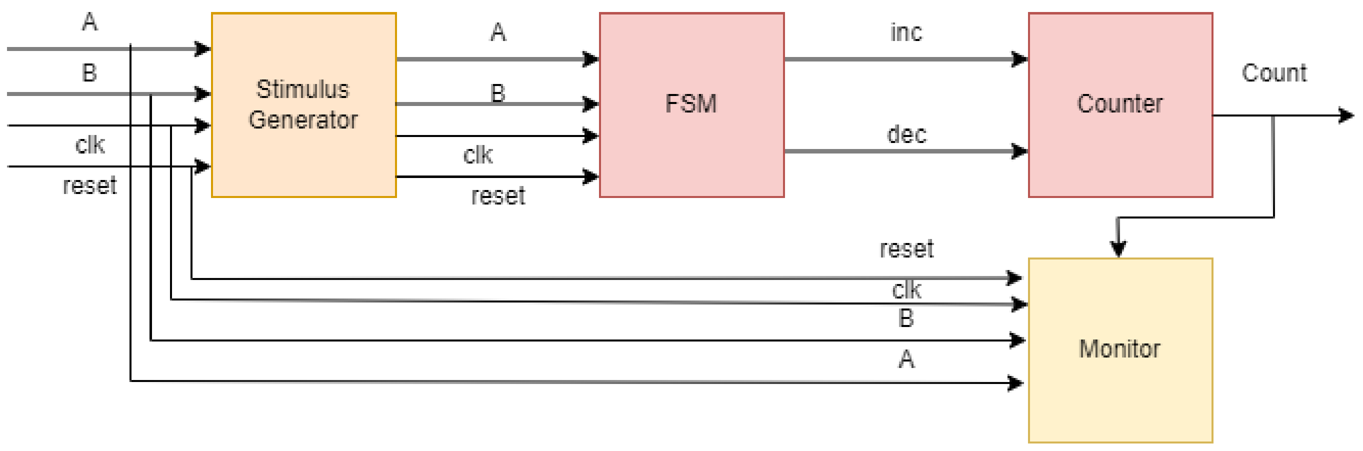

Fig 1: Proposed Design

1. Top Level Testbench

The top-level testbench module is designed to instantiate the parking lot counter, the stimulus generator, and the monitor unit. First, the stimulus generator is called, which gives us the test vectors needed for testing. The stimulus for the generator is stored and then passed onto the parking lot counter which gives us the output as per the inputs. The stimulus generator also computes the expected increase or decrease value. All outputs from the generator, including increase or decrease expected, is passed onto to the monitor module along with the output from the parking lot counter which gives us the actual car count.

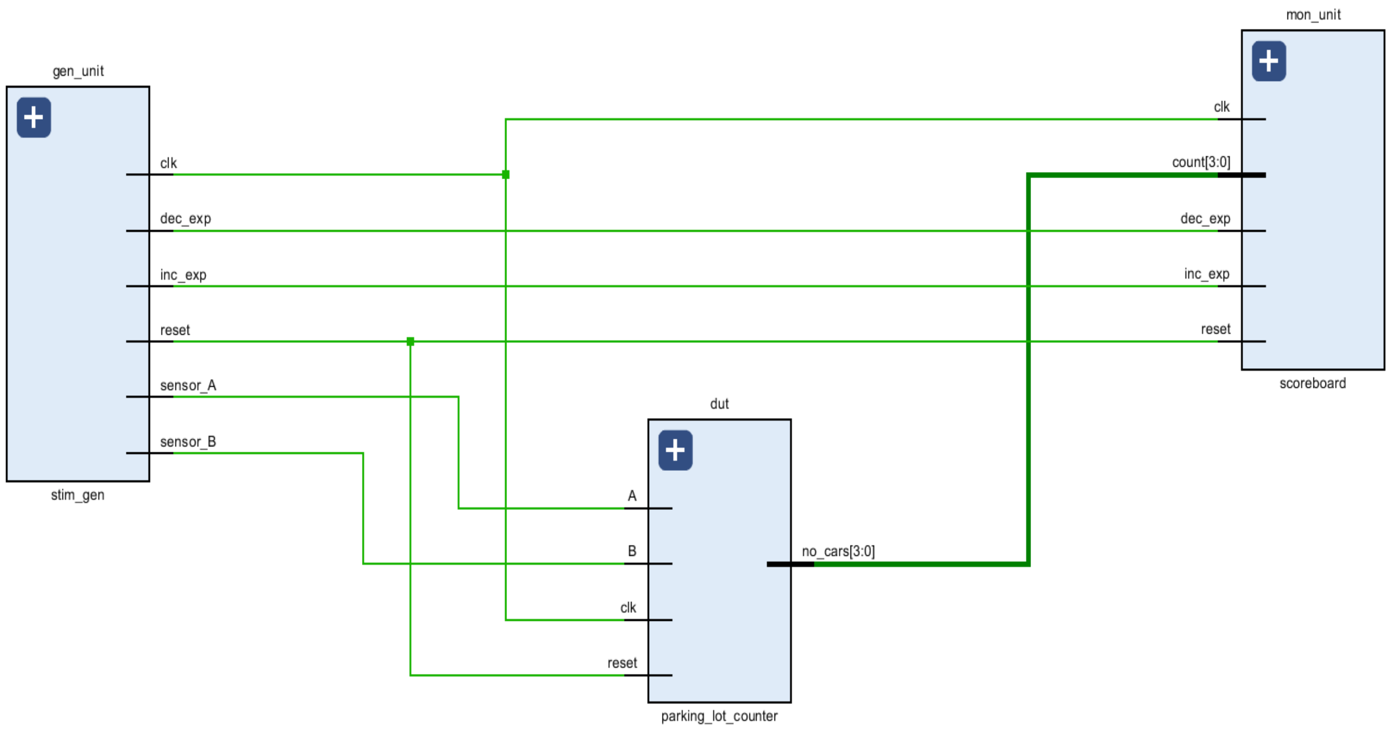

Fig 2: Top level testbench elaborated design

2. Stimulus Generator

The stimulus generator is responsible for producing the test vectors with which we test our parking lot counter. Tasks and loops are used to develop a comprehensive and robust set of test vectors.

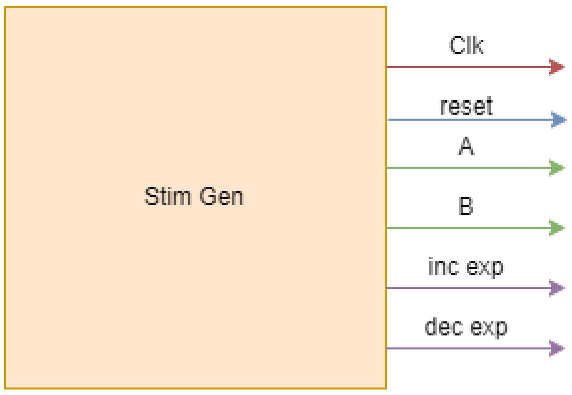

Fig 3: Proposed Design for stimulus generator

It consists of the following tasks (refer stim_gen.sv):

1. **Generate stim:** Calls allow the below functions.
2. **Initialize:** starts up the entire system. test vectors for reset.
3. **Car enters** generates test vectors for car entering. Stim for ports A and B.
4. **Car exits** generate test vectors for car exiting.
5. **Car halfway in** generate test vectors for car half entering.
6. **Car exit half out:** generate test vectors for car half exiting.

This module also computes the expected increase or decrease value. These two variables are 2-bit numbers which can take the value of 0 or 1. If any of these are asserted to 1 that means one should expect an increase or decrease. The values are also driven to the monitor module to compute the desired count value and in turn, compare that with the actual count value.

The *generate stim* task brings tasks (2-6) together using for loops. This was done to clean up the code for simplicity. This way I was able to test different test cases like:

1. Reset functionality
2. Counter increase or decrease
3. Counter Limit
4. Situations the FSM does not recognize

3. Monitor Block

The monitor block takes in inputs from the stimulus generator and the parking lot counter (dut) itself. The stimulus generator drives in the clock, reset, increase, or decrease expected and the dut drives in the car count which is computed from the A and B inputs from the stimulus generator.

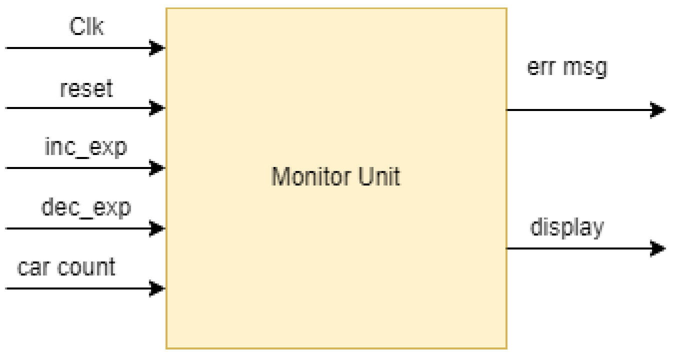

Fig 4: Proposed Design for monitor block

There is an internal counter that is running in the scoreboard module that calculates the desired count based upon the expected values. This expected value is compared to the actual car count from the parking lot counter module and then depending upon the result an error message is displayed if there is something wrong.

4. Parking Lot counter

The parking lot counter is a top-level file that instantiates the finite state machine module and the 4-bit counter module.

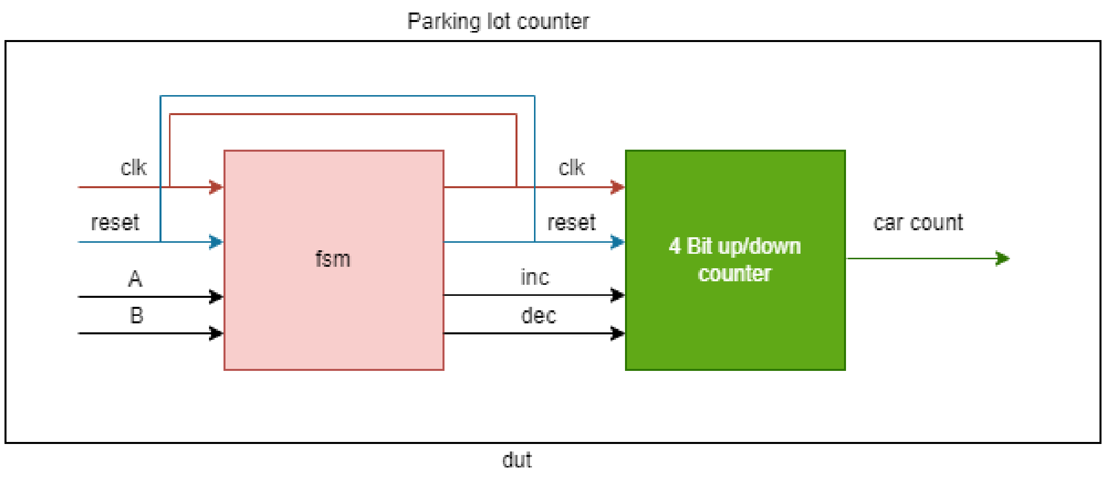

Fig 5: Proposed top level module design

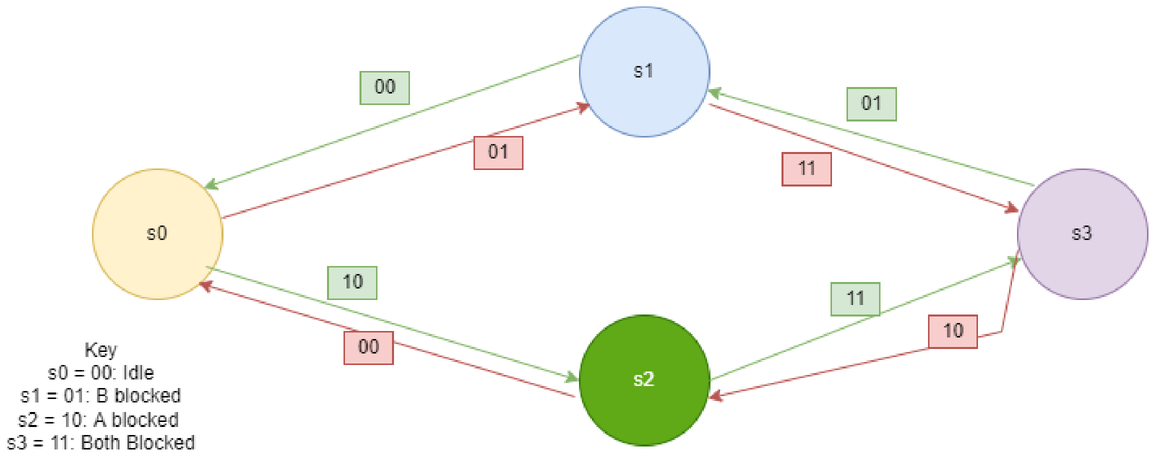

Fig 6: Proposed State Machine Design

The parking lot counter top-module, the fsm and the 4-bit counter were tested as part of practical 1 so they are validated. Although my design does not consider edge cases where; a car enters half in and changes its mind or vice versa. Due to this lack of logic in the design, test vectors that were induced, show these bugs in the design.

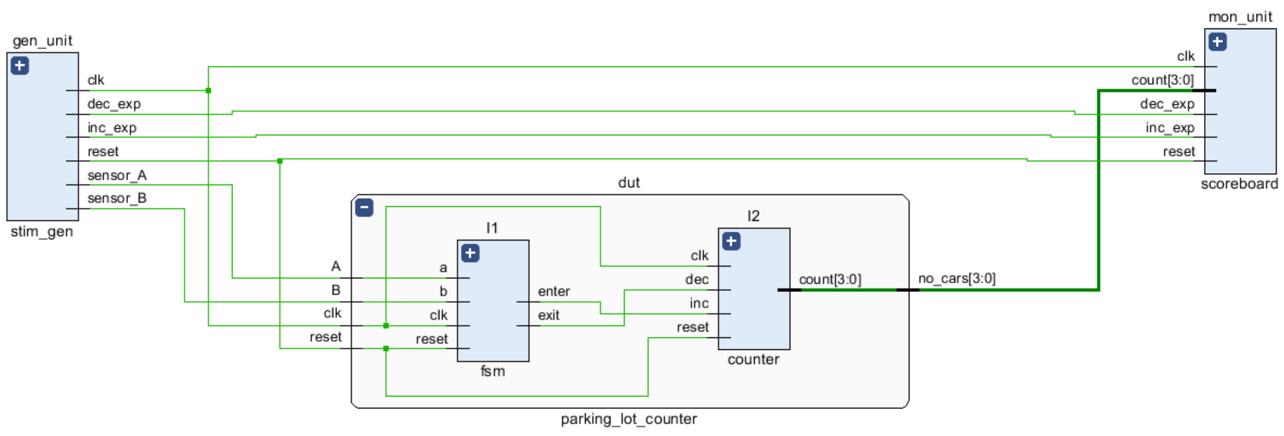

Fig 8: Elaborated Design for Parking lot counter

Results

The table represented below are the various test cases that the parking lot counter was validated with:

Test Cases (Refer stim_gen.sv)						
Test Case for Car Enters						
Test #	***Sensor A***	***Sensor B***	***Inc exp***	***Dec exp***	***Pass***	***Fail***
1	0	0			🟩	
	1	0				
	1	1				
	0	1				
	0	0	1	0		
Test Case for Car Exits						
2	0	0			🟩	
	0	1				
	1	1				
	1	0				
	0	0	0	1		
Test Case for Car enters changes mind mid-way						
3	0	0				🟥
	1	0				
	1	1				
	1	0				
	0	0	0	0		
Test Case for Car Exits and changes mind mid-way						
4	0	0				🟥
	0	1				
	1	1				
	0	1				

	0	0	0	0		🟥
Test Case for Parking lot full (Enter Sequence x 18)						
5	0	0			🟩	
	1	0				
	1	1				
	0	1				
	0	0	1	0		

Table 1: Results for test cases

Referring to the table above, you can see how test case #3 and #4 failed for my proposed design. This was because my FSM does not account for edge cases i.e., it does not take into consideration that the car enters and changes mind mid-way or vice versa. This was caught by my error handling logic and displays the error in the tcl console and the log file.

Similarly, Test Case #5 was introduced to check how does the counter module behave if it exceeds the max number which is 15 spaces in our case. This passed because as expected the counter restarted the counting, but it was caught by the error handling logic.

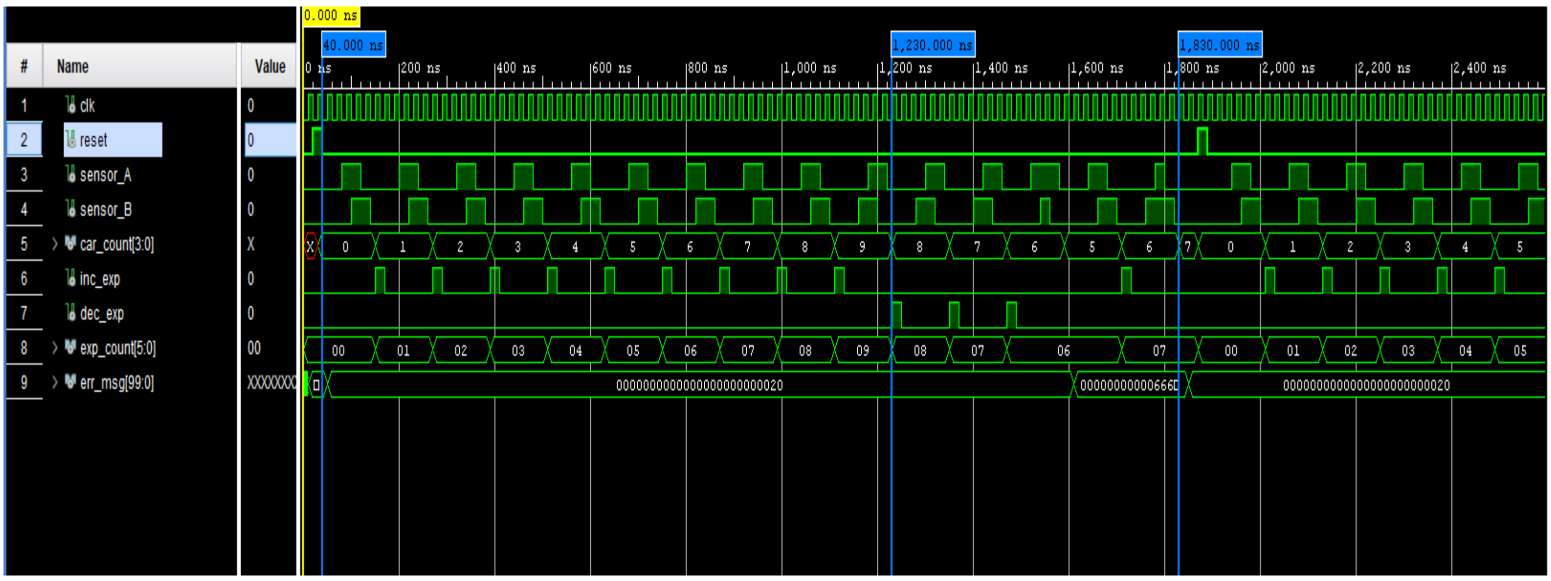

Fig 9: Timing Diagram for the whole system

The timing diagram above represents the whole system in action. The marker visible at 40ns represents the initialize function which asserts reset and if we refer to marker 2 at 1230ns we can see that the car enters task works successfully. Similarly, marker 3 at 1830ns is a representation of the car exits task.

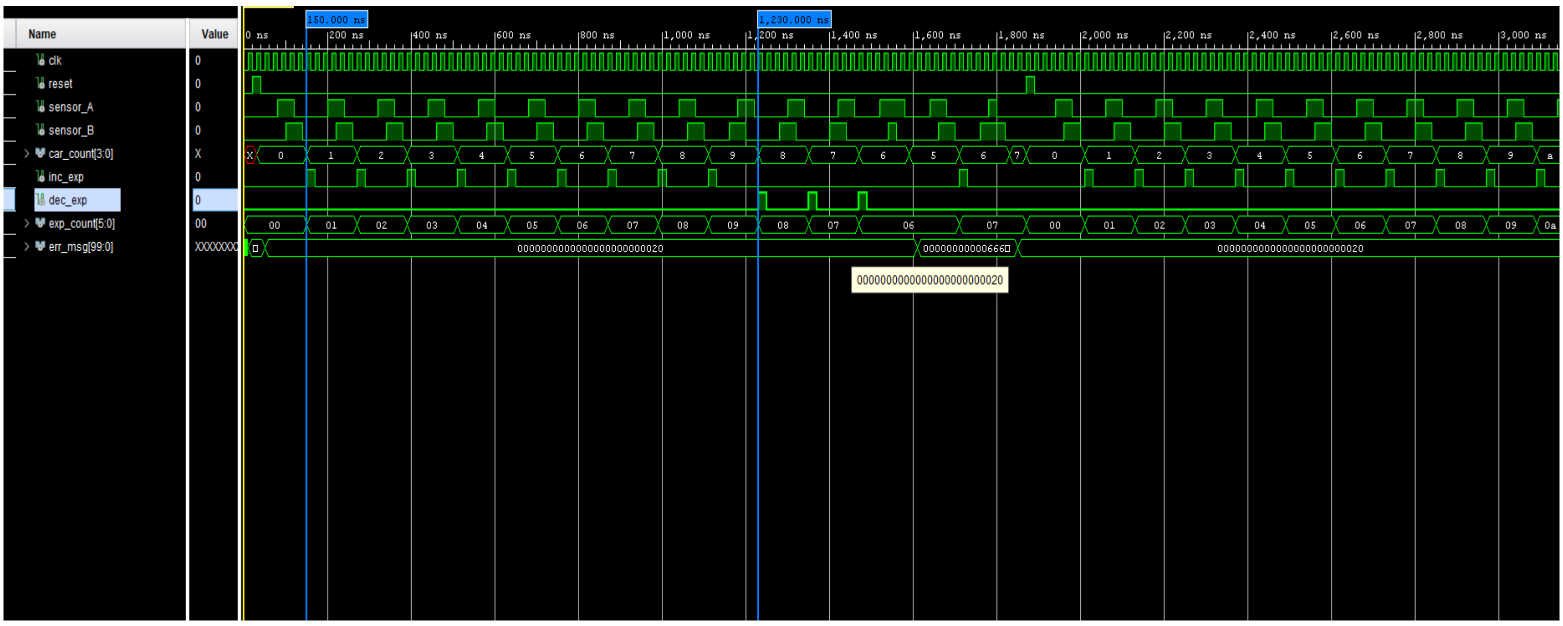

Fig 10: Timing Diagram showing inc_exp and dec_exp

Referring to the diagram above, marker # 1 at 150ns represents the logic for increase expected and marker # 2 at 1230ns represents the logic for decrease expected. The idea here is that these values should assert as soon as the entry or exit sequence is completed. These expected values are sent to the monitor module and are used to compute the desired count value for validation purposes.

Conclusions

After checking and analyzing my results and timing diagrams the following conclusions were drawn:

1. The parking lot counter does not recognize cases like a car changes their mind mid-way while entering or exit.
2. There is provision in the counter for relaying the information that the parking lot is full. The counter would restart after reaching the upper limit; 15 in our case.
3. During the first practical when a less comprehensive testbench was made to these the parking lot counter, these errors were not caught but due to robust nature of a comprehensive testbench and the ease of adding additional test vectors make it easier to test and validate a design better.
4. Potentially, we could add more testcases to this simulation like:
    a. Differentiating between a pedestrian and a car.
    b. Faulty sensors readings, etc.

****

References

1. FPGA Prototyping by Verilog Examples – Pong P. Chu
2. 4C1 Lecture notes
3. Stack Overflow