Lab 8: Fooling a 2D Parity Check

Ben Heard

Due: 15 April 2022

Abstract

I was surprised at the complexity introduced in implementing the 2D parity checking system that we've done over the last lecture. In particular, the need to introduce an error in a data stream poses an interesting challenge in testing the implementation.

In this lab you'll be introducing errors in the correct spot so as to foil the 2D parity error checking. You'll demonstrate the simulation to the TA.

1 Description

In lecture we discussed a system that implements a 2D parity error checking scheme. A more detailed discussion of a complete system is below including a transmitter, receiver, and the channel between them.

The transmitter accepts 4 ASCII character parallel data on a valid/ready handshake from an upstream provider and delivers those characters along with the 2D parity check bits to the receiver. The data are delivered as 40 serial bits through a serial channel. The receiver then receives those 40 bits in a serial manner, checks the parity bits, and provides the data to a downstream consumer as 4 ASCII character parallel data along with error indication on a valid pulse. Timing diagrams are show in the lectures.

The complete implementation is provided in a single file in your repository as parity2d_system.v.

The implementation uses a simple counter to introduce errors in the channel at fixed intervals. NOTE: these intervals are not related to the sending of data; just regular over time with respect to a free-running counter.

The current implementation allows one transfer of data between transmitter and receiver to be correct and the second transfer of the same data to contain errors. You are tasked with changing the error introduction in such a way as to fool the 2D parity checking scheme.

2 Current Error Injection

In the current implementation, the serial channel between the transmitter and receiver is governed by a dataflow assign statement: assign rx_ser_data = tx_ser_data XOR err_inject. The XOR is used so that a 0 on err_inject will pass the tx_ser_data to the receive side without changing. When a 1 is on err_inject then the bit passed to the receive side is flipped, introducing an error.

The err_inject signal itself is gated by en_err_inject. When disabled, err_inject will always have the value 0. When enabled by asserting en_err_inject high err_inject is allowed to assert at specific count values.

A separate counter free-runs from reset. When that counter hits either value of 37 or 45, err_inject is asserted. These numbers were selected only because they worked out to be within the transmission of data during the second transfer; the one I wanted errors in.

3 What To Do

You are to change the timing of the introduced errors so that you fool the 2D parity checking scheme in the example. Recall that flipping a single bit in one word will be caught by both row and column parity checking. Flipping two bits in a word will only be caught by the column parity checking (row parity fails because 2 bits have been flipped).

To get column parity checking to fail, you need to change the same two bit positions in two different words. The two positions in each word will cause the row parity checking to fail as above. And the two bits being in the same column between words will cause the column parity checking to fail.

There are a number of approaches but the easiest will be to download the file, evaluate how the 37/45 affect data in what is transmit and received, to figure out what the values should be (4 of them) to hit the same two bits in two different words, and to simulate at the command line.

4 Checking Your Testbench Implementation

You'll need a clean workspace and run through the following commands.

```
% vlib work
% vmap work work
% vlog parity2d_system.v
% vsim -c parity2d_system_tb

VSIM> run -all
```

This will create a work library, map it in the modelsim.ini , compile the Verilog, and then simulate the testbench that you've modified as per the requirements above.

5 What To Turn In

Once you've edited the testbench with your bit positions to fool the system, add, commit, and push the testbench back to your repository.