Michael Chan, Siddhanta Shrestha

ECE 332: Lab Report 3

**Introduction**

The goal of this lab was to familiarize ourselves with verilog inorder to implement a Run Length Encoding (RLE) state machine using the ModelSim software. There were two main components in this lab, the rle code and the rle testbench. The code for the RLE state machine is used to count the number of consecutive bits in an 8 bit stream. The bits are shifted when a bit breaks the consecutive chain. Testing this was done with the testbench where we fed the RLE test bitstreams for simulation.

**Detailed Procedure**

We were given two files for this lab that contained starter code necessary for this lab. These files were stored in libraries that we created within folders. The starter code for rle.enc.v contained information on the general function of the state machine as well as the transitions between states. Our job was simply to write the functionality and logic of each state using the state diagram that was provided for us. For the testbench we simply set up the code and provided sample bitstreams for the simulation which was fed into the state machine.

We then compiled the code and simulated it in order to see the functioning state machine. To simulate the test bench code, we had to ensure that the library for where the rle.enc.v file was selected to ensure that it would be applied.

**Hardware Changes**

There were no hardware setups necessary for this lab since we did not use the board. This lab was primarily done on ModelSim.

**Software Changes**

The majority of the software was either provided to us in the starter code or was easily derived from the state machine diagram that was provided. The diagram (and help from the comments in the code) showed us exactly what we needed to initialize/set the registers/variables to. We had to ensure that the registers were set using the correct verilog format. For example, setting a vector of 8 binary values we had to use 8’b00000000 (or just 8’b0). Only one that required code were count and shift bits.

For count bits we first needed to know if we were looking at a new bitstream. If yes then we want to know exactly what bit (0 or 1) is going to be counted. This is found in shift\_buf[0]. Then we notify the system of a new bitstream coming in and finally increase the bit counter by one. In the event that it is not a new bitstream we need to check to see if the current bit is the value we are counting and increase the bit counter accordingly, else change the new bitstream to signify that the bitstream has finished counting.

| if(new\_bitstream) begin  value\_type <= shift\_buf[0];  new\_bitstream <= 0; //1'b0;  bit\_count <= bit\_count + 1; //23'b00000000000000000000001; end else begin  if(shift\_buf[0] == value\_type) begin  bit\_count <= bit\_count + 1; //23'b00000000000000000000001;  end  else begin  new\_bitstream <= 1; //1'b1;  end end |
| --- |

Count bits code

For shift bits the only change added was an if statement that would run if it was not a new bitstream. In this case we essentially shift the bits and check how many consecutive bits there are in the stream during this process

| if(!new\_bitstream) begin  shift\_buf <= shift\_buf >> 1;  shift\_count <= shift\_count + 1; //4'b0001; end |
| --- |

Shift bits code

For the testbench, we established a clock that continues forever and a reset signal that pulsed once. Initialized three 8-bit test vectors. Set both send\_readyt and recv\_readyt to 1. This allows the state machine to move to the correct next state. The end\_of\_streamt was also set to 0 to ensure that all the registers do not reset back to the initialized values..

| // clock running forever  always  begin  // Clock signal assignment  clkt = 0;  forever #1 clkt = ~clkt;  end  // reset for a few cycles  initial  begin  // Reset signal assignment  rstt = 0;  #1 rstt = 1;  #4 rstt = 0;  end |
| --- |

Clock and Reset code

The test vectors were fed to the in\_datat at set time intervals to allow for proper counting of the bitstreams. We chose a time delay of #50. After the third and last test vector was fed to in\_datat and the #50 time delay has occurred, recv\_readyt was set to 0. This ensured that in\_datat would not be copied to the shift\_buf, which would continue the counting process incorrectly. Finally another after another delay of #50 the end\_of\_steamt would be set to 1 to signal that the counting has completed.

**Problems Encountered and Solutions**

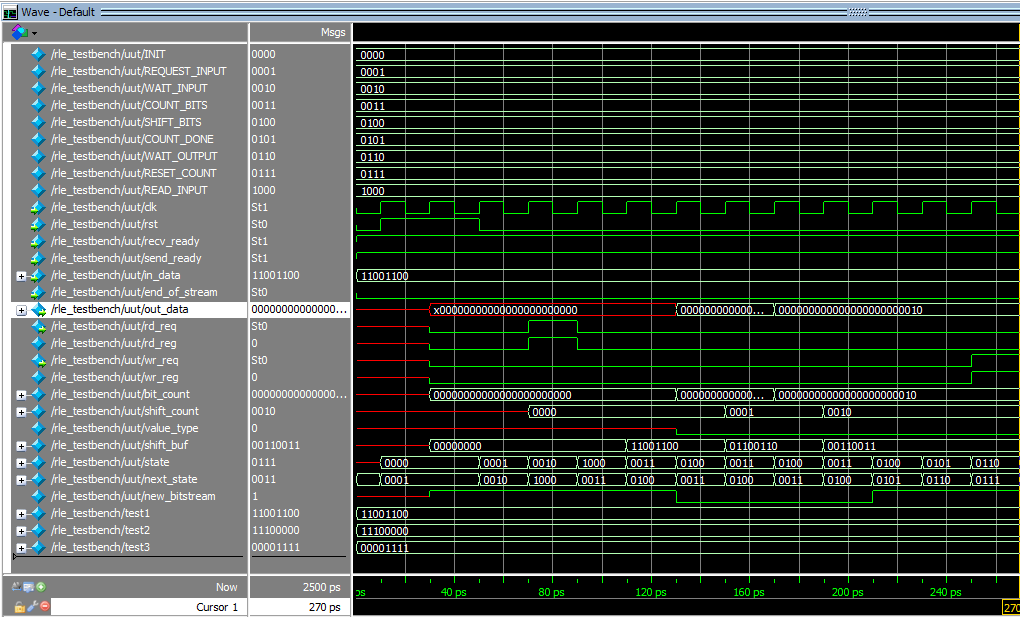
We encountered a few problems during the setup of the ModelSim program. This issue was mainly where we were saving the verilog files. The issue came up when attempting to run our test bench file and it showed that the RLE file didn’t exist. The issue was solved by simulating the test bench file with the RLE library selected.

A minor issue we came across was the incorrect coding of the RLE verilog program. We did not assign the values of our variables correctly using binary format. This was noticed when we attempted to run our test program. We ensured that the values were chosen in accordance with the state diagram.

Another issue encountered in our testbench was when to tell the program that the end\_of\_stream has been reached. Incorrect timing would lead to incorrect counting. If end\_of\_stream was set to 1 to early, a portion of the third 8-bit test vector would not be counted. Setting too late wouldn’t matter as long as recv\_ready was set to 0. Trial and Error was required to determine how much delay was needed.

**Results and Conclusions**

Compiling and running our testbench.v with the library of rle.enc.v shows that our RLE program works. We input three 8-bit test vectors and the out\_data showed the correct number of 0’s or 1’s in a bitstream.

****

When the wr\_req goes from 0 to 1, the out\_data is written. In this case our first test vector had a series of two 0’s before changing to 1. The resulting out\_data written is: out\_data = 000000000000000000000010

We verified that the state machine works as intended. Setting send\_ready and recv\_ready to 1 allowed the RLE to start and continue counting. Setting end\_of\_stream to 1 was verified to reset everything back to the initialized values.