Group 14C: J. Lagasse, A. Sjogren, O. Onyenokwe, L. Wu
ECE 354: Lab 3 Report
3/30/2017

1. Introduction:

The main objective of this lab was to develop a Run Length Encoding (RLE) compression system using Verilog. We then verified the output using a functional simulation tool called ModelSim. RLE compression is done by taking a data stream, recording number of bits in a row that are the same, and outputting the value of the bit and number of its iterations. This compression algorithm works best with data that has hundreds or thousands of the same bit in a row, such as in black and white photographs where each pixel's value is either all ones or all zeroes. The key parts of this lab were to learn how to understand a general finite state machine (FSM) of implementing a RLE algorithm, write a custom design in Verilog that implements the FSM, write a Verilog testbench to simulate inputs to the design, and verify that the design works properly by performing a functional simulation in ModelSim.

2. Detailed Procedure:

1. We started this lab by reading the provided lecture slides. From here, we analyzed the finite state machine and other resources about the RLE algorithm to understand what changes we needed to make to the provided Verilog code.
2. We then implement these changes to the best of our ability and tested them manually in ModelSim.
3. Once we saw that the first 8 bit segment of input data we used was working as expected via manual simulation, we started working on coding the testbench to be able to easily simulate a stream of several 8 bit segments in a row to test in ModelSim. We set internal variables in the RLE verilog code to outputs so that we could program the test bench to read these internal variables as inputs. This allowed us to view the internal variables when simulating the testbench in ModelSim. This streamlined our testing significantly and allowed us to verify that all parts of our RLE code were working as expected, not just the inputs and outputs. These internal variables included the current state, the next state, shift buffer, shift count, end of bitstream, and more.
4. Once the testbench was filled with all the data we wanted to test, including a clock signal, reset signal, nine consecutive 8 bit input segments, and more, we then ran our RLE code and examined that the output.
5. We made changes to the RLE code and the testbench code until everything worked as expected. The output that we were expecting was 24 bit segments that were produced whenever the input data changed from a 1 to a 0 or vice versa. The most significant output bit was the input data type (1 or 0) while the remaining 23 bits consisted of the number of 1's or 0's that were input in a row. We also made sure that the output would continue to count up correctly even when new 8 bit segments were introduced.

3. Hardware Changes:

Since this project was about programming a Verilog file and an associated testbench, we are breaking up the report so that the Verilog code for the RLE encoder is the hardware component and that the testbench is the software component. This is because Verilog is a hardware description language, so even though it is a type of code it explicitly represents only hardware components. The testbench however is considered to be software programming in a sense even though it is in Verilog because a test bench only sends input values to a program, it does not represent any physical hardware.

```
74  always @(posedge clk) begin
75      if(rst) state <= INIT;
76      else state <= next_state;
77
78      case(state)
79      // STATE 0
80      INIT: begin
81          //Initialize registers
82          bit_count <= 0;
83          shift_buf <= 0;
84          rd_reg <= 0;
85          wr_reg <= 0;
86          new_bitstream <= 1;
87      end
88
89      // STATE 1
90      REQUEST_INPUT: begin
91          //Assert rd_req signal to FIFO by setting rd_reg
92          //FIFO takes rd_req signal at next clock
93          rd_reg <= 1;
94          shift_count <= 0;
95      end
96
97      // STATE 2
98      WAIT_INPUT: begin
99          //De-assert rd_req by setting rd_reg
100         rd_reg <= 0;
101     end
102
```

Figure 1: Beginning of Sequential Logic; States 0-2

```
103     // STATE 3
104     READ_INPUT : begin
105         //FIFO provides valid data after taking rd_req
106         //shift_buf stores 8 bit input data
107         shift_buf <= in_data;
108     end
109
110     // STATE 4
111     COUNT_BITS: begin
112         //Count number of consecutive bits in shift_buf
113         //If new type of bit starts, store bit ID in value_type register
114         //If current value_type and shift_buf[0] is not matched, notify current encoding
115         if(new_bitstream) begin
116             new_bitstream = 0;
117             value_type <= shift_buf[0];
118             bit_count <= bit_count + 1;
119         end
120         else begin
121             if(shift_buf[0] == value_type) begin
122                 bit_count <= bit_count + 1;
123             end
124             else begin
125                 new_bitstream = 1;
126             end
127         end
128     end
129
130     // STATE 5
131     SHIFT_BITS: begin
132         //Right shift the shift_buf
133         //Increase shift_count
134         if(!new_bitstream) begin
135         shift_buf = shift_buf >> 1;
136         shift_count <= shift_count + 1;
137         end
138     end
139
```

Figure 2: Sequential Logic; States 3-4

```
130         // STATE 5
131         SHIFT_BITS: begin
132             //Right shift the shift_buf
133             //Increase shift_count
134             if(!new_bitstream) begin
135             shift_buf = shift_buf >> 1;
136             shift_count <= shift_count + 1;
137             end
138         end
139
140         // STATE 6
141         COUNT_DONE: begin
142             //Assert wr_req by setting wr_reg
143             //FIFO will take wr_req signal in next clock cycle
144             wr_reg <= 1;
145         end
146
147         // STATE 7
148         WAIT_OUTPUT : begin
149             //De-assert wr_req by setting wr_reg
150             wr_reg <= 0;
151         end
152
153         // STATE 8
154         RESET_COUNT : begin
155             //Reset bit counting register after passing encoded data to output side FIFO
156             bit_count <= 0;
157         end
158
159         endcase
160     end
161
```

Figure 3: Sequential Logic; States 5-8

Almost all of the Verilog RLE code was provided to us by the TAs. Our responsibility was to fill in the sequential logic which represented the finite state machine. Therefore, the first changes that were made to the code were to implement the most basic components of the finite state machine. We did this by setting all of the registers that were explicitly given in the finite state machine diagram to the values shown in the slides. Once this was done, the states init, request_input, wait_input, read_input, count_done, wait_output, and reset_count were complete.

The majority of the code to be written and modified on our own occurred within the count_bits and shift_bits states. These two states typically were in a sort of loop for the duration of the counting, as these two methods were responsible for reading each individual input values and incrementing the bit_count value. If it is a new bitstream, the state count_bits sets the first value of the shift_buffer to value type, and adds 1 to the bit_count to account for this bit. The state shift_bits then moves the shift buffer right 1 place, and adds 1 to the shift count. As long as the first bit in the newly shifted shift buffer is equal to the value type, the bit count will increase by one. These two states will alternate back and forth until it needs to get a new 8 bit segment or the bit stream ends. If we get a new 8 bit segment, we go back to the read_input state and read in another

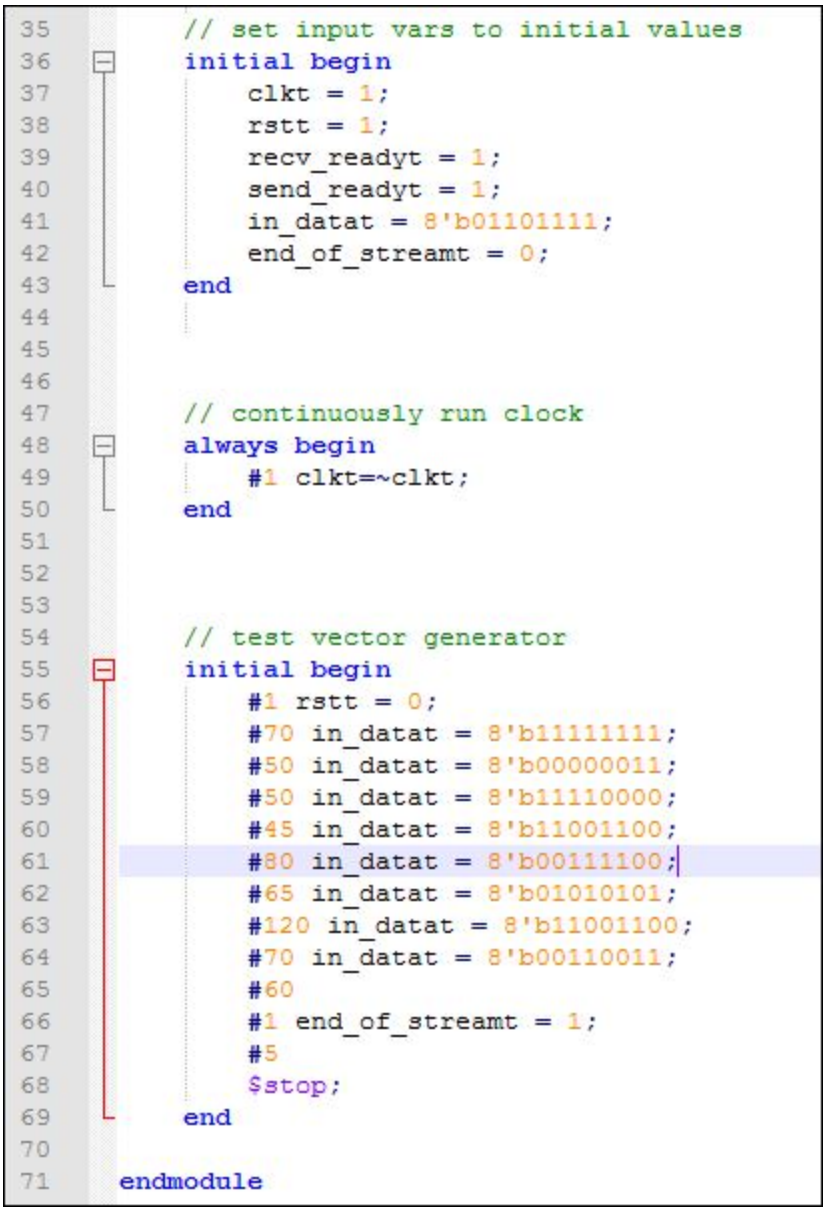
segment before returning to counting and shifting. If the bit stream ends, the register end_bit_stream is set to 1 and the next state becomes count_done. All of this is shown in Figures 1-3.

4. Software Changes:

The software aspect of this project consisted of implementing the testbench. We were given a basic outline of the testbench and then filled it in with our own data. We also coded wrote so that we could see the internal variables from the RLE code in the testbench, as shown in Figure 4. Figure 5 shows the values we set upon initialization and our sequence 8 bit segments, stored in the in_data register. Note that lines 36-43 detail what values we wanted to initialize at the beginning of the simulation. Lines 48-50 create an alternating clock pulse whose period is 1*100ps or 100ps. Lastly, lines 55-69 force the reset to turn off for the rest of the simulation, feed the program 8 consecutive 8 bit segments of data inputs, set end_of_stream to 1, and then give the program a delay so that the reinitialization can be observed.

```
1   `timescale 100 ps/10 ps
2
3   // The `timescale directive specifies that
4   // the simulation time unit is 100 ps and
5   // the simulator timestep is 10 ps
6
7   module rle_testbench;
8       // signal declaration
9       reg clkt, rstt;
10      reg recv_readyt, send_readyt;
11      reg [7:0] in_datat;
12      reg end_of_streamt;
13      wire [23:0] out_datat;
14      wire rd_reqt, wr_reqt;
15
16      // internal vars
17      wire rd_regt,wr_regt;           //Write request for output side FIFO
18      wire [22:0] bit_countt;         //Store number of consecutive bits in bit stream
19      wire [3:0] shift_countt;            //Current shift amount of shift_buf
20      wire value_typet;               //Bit ID
21      wire [7:0] shift_buft;          //Store 8 bit segment of bit stream comes from input side FIFO
22      wire [3:0] statet;
23      wire [3:0] next_statet;
24      wire new_bitstreamt;
25
26
27
28      // instantiate the circuit under test
29      rle_enc uut
30          (.clk(clkt), .rst(rstt), .recv_ready(recv_readyt), .send_ready(send_readyt), .in_data(in_da
31          .rd_regp(rd_regt), .wr_regp(wr_regt), .bit_countp(bit_countt), .shift_countp(shift_countt),
32
33
34
```

Figure 4: Testbench Part 1



```
35      // set input vars to initial values
36      initial begin
37          clkt = 1;
38          rstt = 1;
39          recv_readyt = 1;
40          send_readyt = 1;
41          in_datat = 8'b01101111;
42          end_of_streamt = 0;
43      end
44
45
46
47      // continuously run clock
48      always begin
49          #1 clkt=~clkt;
50      end
51
52
53
54      // test vector generator
55      initial begin
56          #1 rstt = 0;
57          #70 in_datat = 8'b11111111;
58          #50 in_datat = 8'b00000011;
59          #50 in_datat = 8'b11110000;
60          #45 in_datat = 8'b11001100;
61          #80 in_datat = 8'b00111100;
62          #65 in_datat = 8'b01010101;
63          #120 in_datat = 8'b11001100;
64          #70 in_datat = 8'b00110011;
65          #60
66          #1 end_of_streamt = 1;
67          #5
68          $stop;
69      end
70
71  endmodule
```

Figure 5: Testbench Part 2

5. Problems Encountered and Solutions:

We did not run into many problems while doing this lab. One issue we had involved setting the delays for the input data in the testbench. Without proper delay, the correct input of 8-bit sequences was not read correctly, as the inputs did not change at the right times for the program to notice before it began shifting and counting bits. We could not use the same amount of delay for each segment because each segment had a different combination of 1's and 0's. The more frequently that a sequence switched between 1 and 0, the longer it took for the segment to finish encoding because each switch between 1 and 0 required the state diagram to break out of its loop and produce an output, which wasted several clock cycles each time it was called.

In cases where the delay was too long, the program read the old 8 bit segment value instead of the new one. This is because the new value was not available until after the request_input state. When the delay was too short, the program would skip over it because the new value did not last for a long enough amount of time

to be read in the request_input state, so the program read the next segment instead. It took us a few tries to get the hang of how the timing worked so that all of our 8-bit sequences were read in the correct sequence. We also did not know how to run the program so that the end of stream bit was always visible so that we could see the program return to the initialization state after the last sequence. We fixed this by adding additional delays at the end of the testbench so that we could see the changes after setting end_of_stream value high.

6. Results and Conclusion:

Our lab was able to properly decode a stream of 8 bit segments into 24 bit outputs, which was the main goal of the lab. The RLE simulated in the Verilog code identified the bit type and counted how many consecutive bits occurred in a row. We tested 9 different 8 bit segments in a row and examined that different combinations of 1's and 0's were handled correctly. The correct outputs counted the correct number of 1's or 0's in a row, and continued to keep track properly as new 8 bit segments were introduced. We were also able to implement a fully functional test bench to simulate these 8 bit segments without having to type them in manually. Overall, all objectives were met and the project was a success.

However, during the demo we were made aware that we had accidentally left some of our sequential code as blocking statements, meaning that they updated one at a time instead of all at the same time (at the positive edge of the clock). The main reason we did not catch the mistake is because it did not cause any errors in our outputs, and our state machine continued to run as expected. One would think that if something as crucial as using the wrong type of blocking/nonblocking statement was done in the code that the code would automatically fail. However, none of the statements that used the wrong type of statement relied on anything that was immediately changed at the instant of the clock edge, so there was no dependency and thus no ill effect. We will be more vigilant about using the right type of statements in the next project.