

Lab 7,8,10-LEGO Power Functions IR Transmitter Design

Name: Robert Bara

Section #: 003

Date: 4/3/20

Summary/Abstract (10 points)

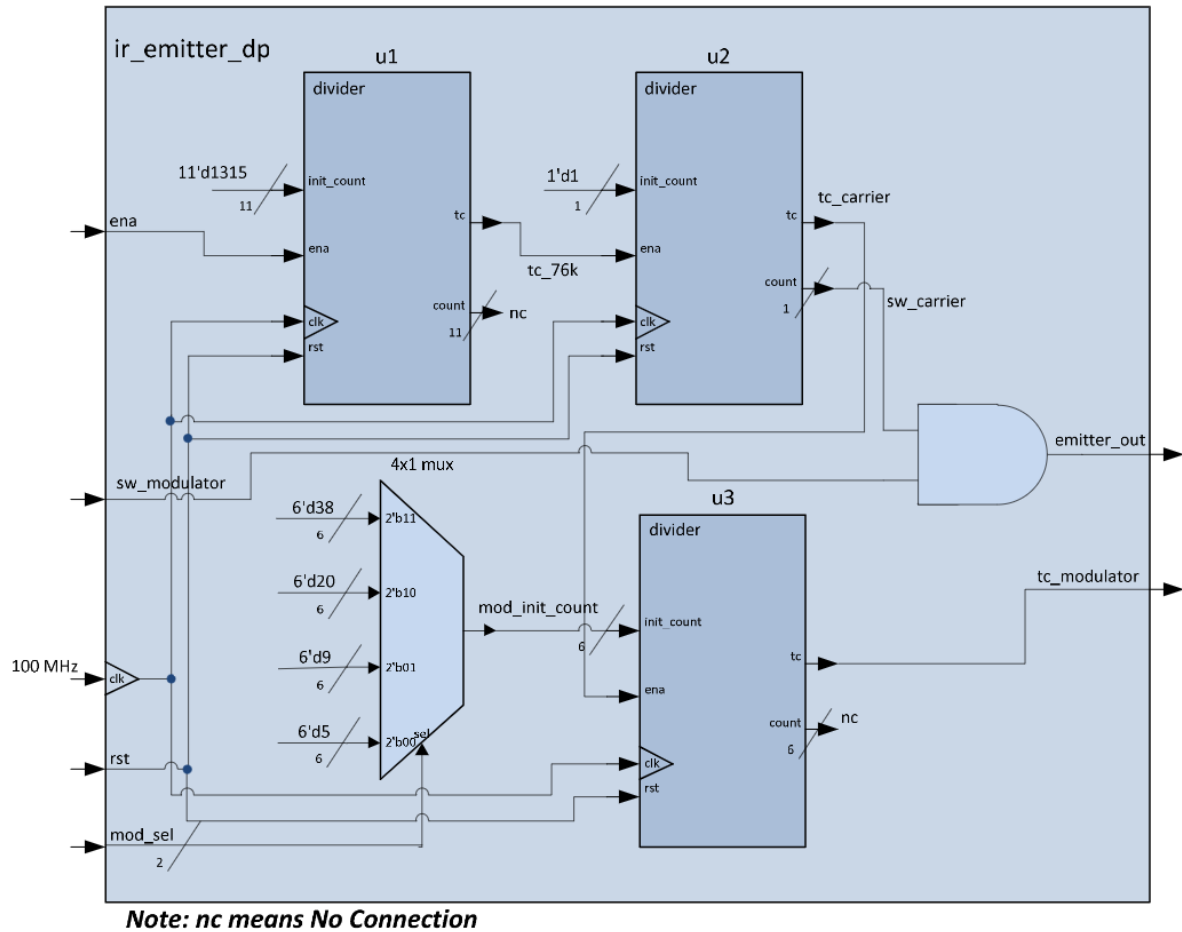
The idea of these labs is to dive into building a Lego Power Functions IR Transmitter and the first step in lab 7 is to create the IR emitter datapath by instantiating counters/dividers by using a divider module that will be created. This will then generate a 38 kHz modulated, invisible EM wave that will get fed into the IR Emitter Finite State Machine from lab 10. In lab 8, an 11 input Switch Decoder module will be created to take an input from the IR transmitter and feed it into the Controller module, then the Emitter Datapath, and finally out of the transmitter. Finally, in Lab 10, all the modules are instantiated together to feed in a switch signal, a bit length signal, clock, and reset signal to create the Transmitter.

Introduction (10 points)

Relevant background information includes understanding how synchronous logic works with combinational logic to create Counters/Dividers, Datapaths, and Finite State Machines. The Counters/Dividers will be applied to the transmitter by creating a parameterized module: divider that will be instantiated into an emitter (transmitter), which will eventually go into the receiver module consisting of a bandpass filter and demodulator. The carrier will also have a 50% duty cycle and will create a divide by two module. In total lab 7 generates 3 divider modules: 1 to generate the 76kHz pulse train, 1 acting as a 1-bit counter (single flip flop), and 1 that counter/divider that counts the number of 38kHz. Lab 8 creates a controller to input the 11 switches into the dividers from lab7. Lab 10 then takes the bit length input, puts it through a synchronizer to then feed into the pulse button debounce, which will create the modulation of the individual switches that are inputted into the switch decoder, this then goes into the IR Emitter Finite State Machine which generates the IR wave because of Lab 7's Datapath and finally the signal comes out of the module. This also of course uses a clock as an input signal and has a reset signal that will go back to the DONE stage of the finite state machine and start the start bit, then the switches, then the stop bit again until the message is complete or the reset is pressed again. A Real-world application for a design similar to this may include TV's or audio equipment since the remote or controls for hardware uses infrared (IR) communication by using a transmitter and receiver.

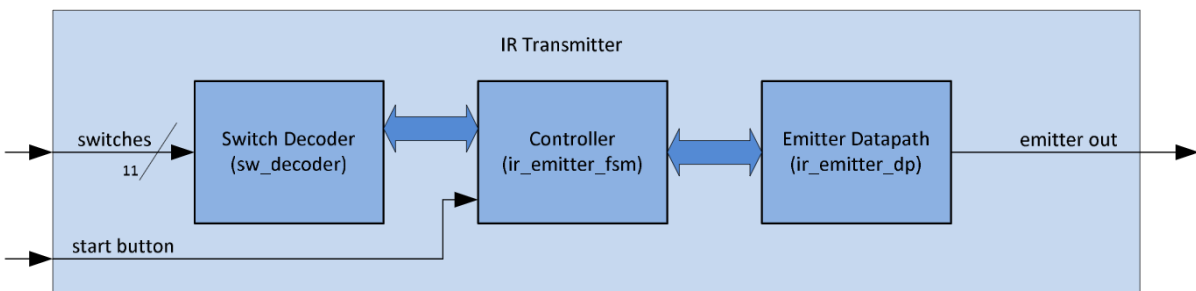
Procedure (15 points)

The transmitter was created first by creating the divider module that followed the outline listed in the lab 7 manual. This divider module is then instantiated to create a 38kHz modulated signal from the 100MHz clock input. The datapath for the infrared emitter then instantiates 3 dividers to implement a signal that counts down from 100MHz by using the logic $100\text{MHz}/.038\text{MHz}=2,632$ counts. This can be done in Verilog by instantiating the divider module created 3 times with an instantiation of the emitter_out as an "and" gate, and building a 4x1 multiplexer by using combinational logic, specifically a case statement:



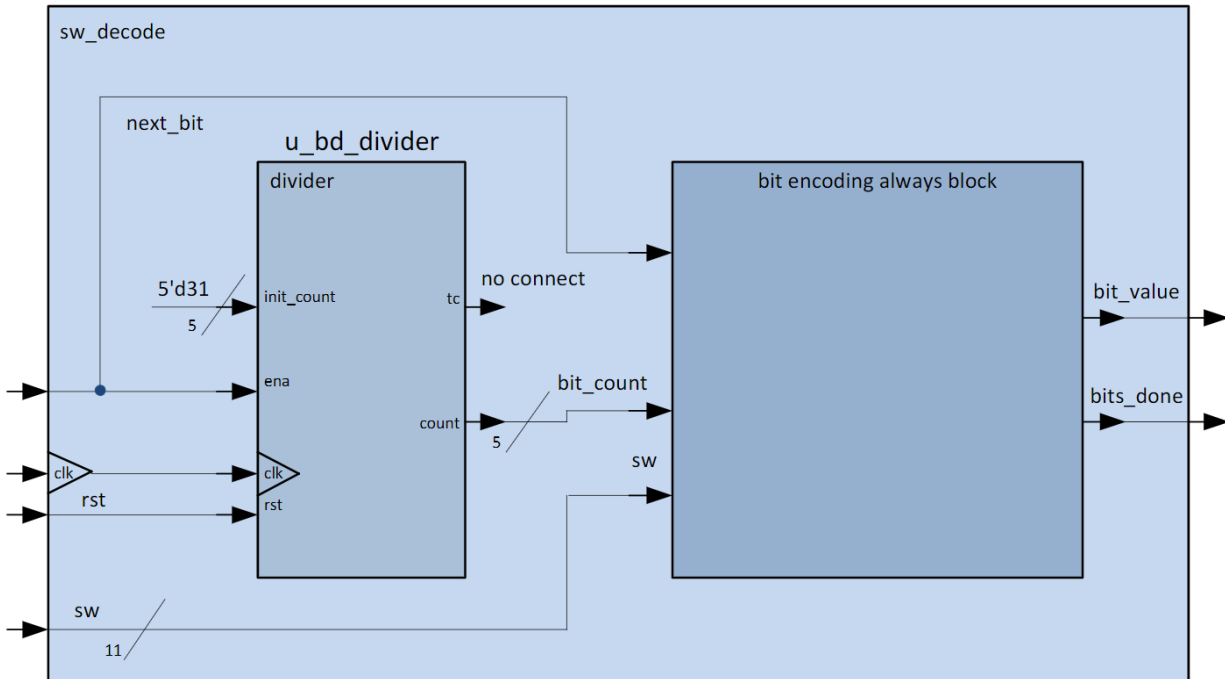
[Block Diagram for IR emitter and Divider modules (Lab 7)]

For the switch decoder, the following top-level block diagram demonstrates how the decoder fits into the rest of the circuit:



[Block Diagram for IR Transmitter and Switch Decoder]

The actual switch decoder itself has a block diagram design as follows:



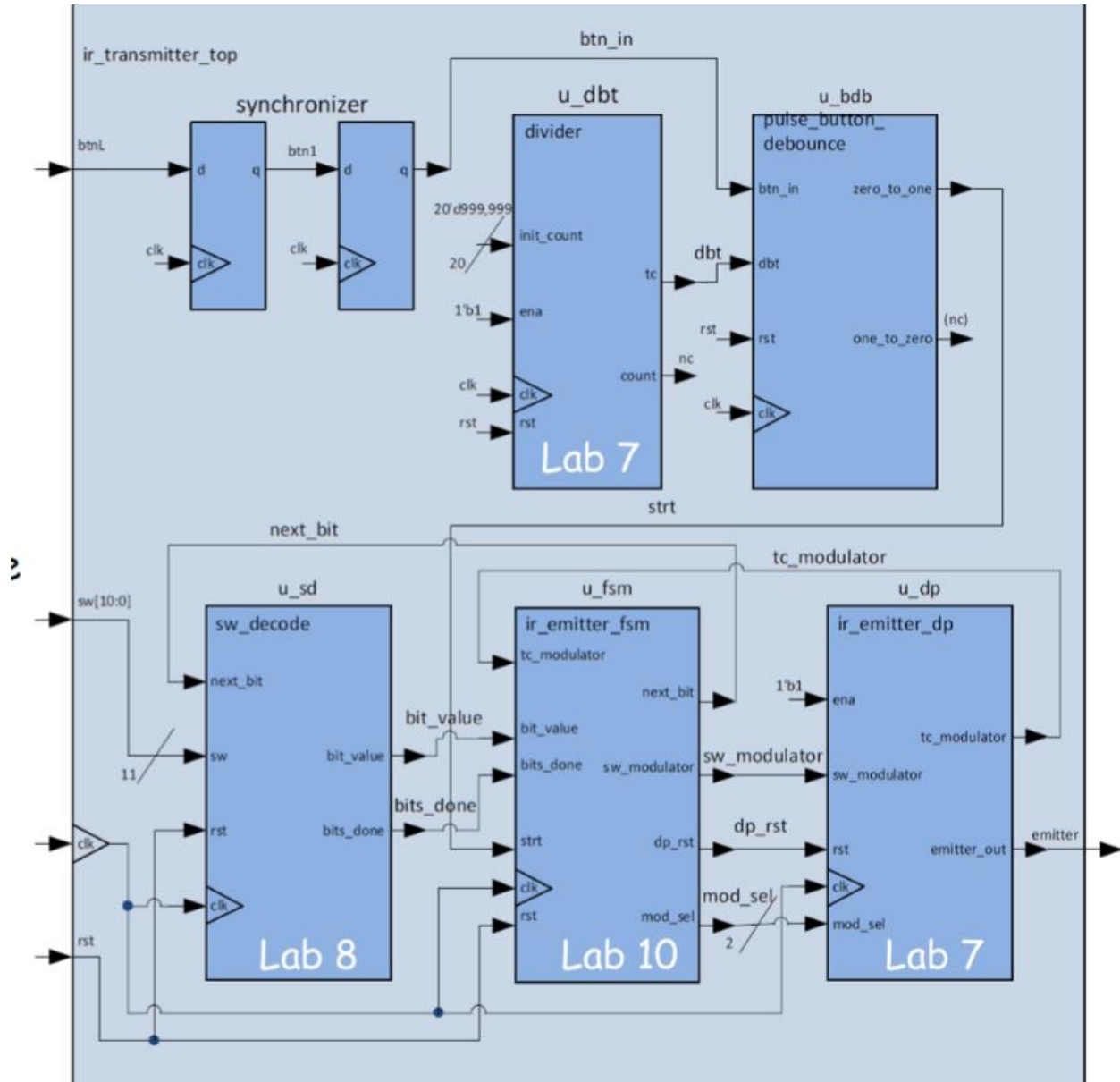
[Switch Decoder Block Diagram Specifications]

From this the decoder instantiates the divider module from lab 7 and then creates combinational logic to form a case statement and if statement corresponding to the following truth table:

bit_count	bit_value	Description
4'd15	$\sim \text{bit_count}[4]$	Toggle bit. Needs inverting as it starts with logic one: (initial count = 5'd31)
4'd14	sw[10]	Escape bit
4'd13	sw[9]	Channel MS bit
4'd12	sw[8]	Channel LS bit
4'd11	sw[7]	Address bit
4'd10	sw[6]	Mode bit 2
4'd9	sw[5]	Mode bit 1
4'd8	sw[4]	Mode bit 0
4'd7	sw[3]	Data bit 3
4'd6	sw[2]	Data bit 2
4'd5	sw[1]	Data bit 1
4'd4	sw[0]	Data bit 0
4'd3	$\sim (\sim \text{bit_count}[4] \wedge \text{sw}[7] \wedge \text{sw}[3])$	LRC bit 3
4'd2	$\sim (\text{sw}[10] \wedge \text{sw}[6] \wedge \text{sw}[2])$	LRC bit 2
4'd1	$\sim (\text{sw}[9] \wedge \text{sw}[5] \wedge \text{sw}[1])$	LRC bit 1
4'd0	$\sim (\text{sw}[8] \wedge \text{sw}[4] \wedge \text{sw}[0])$	LRC bit 0

[Truth Table for the combinational and priority logic (bit encoding always block)]

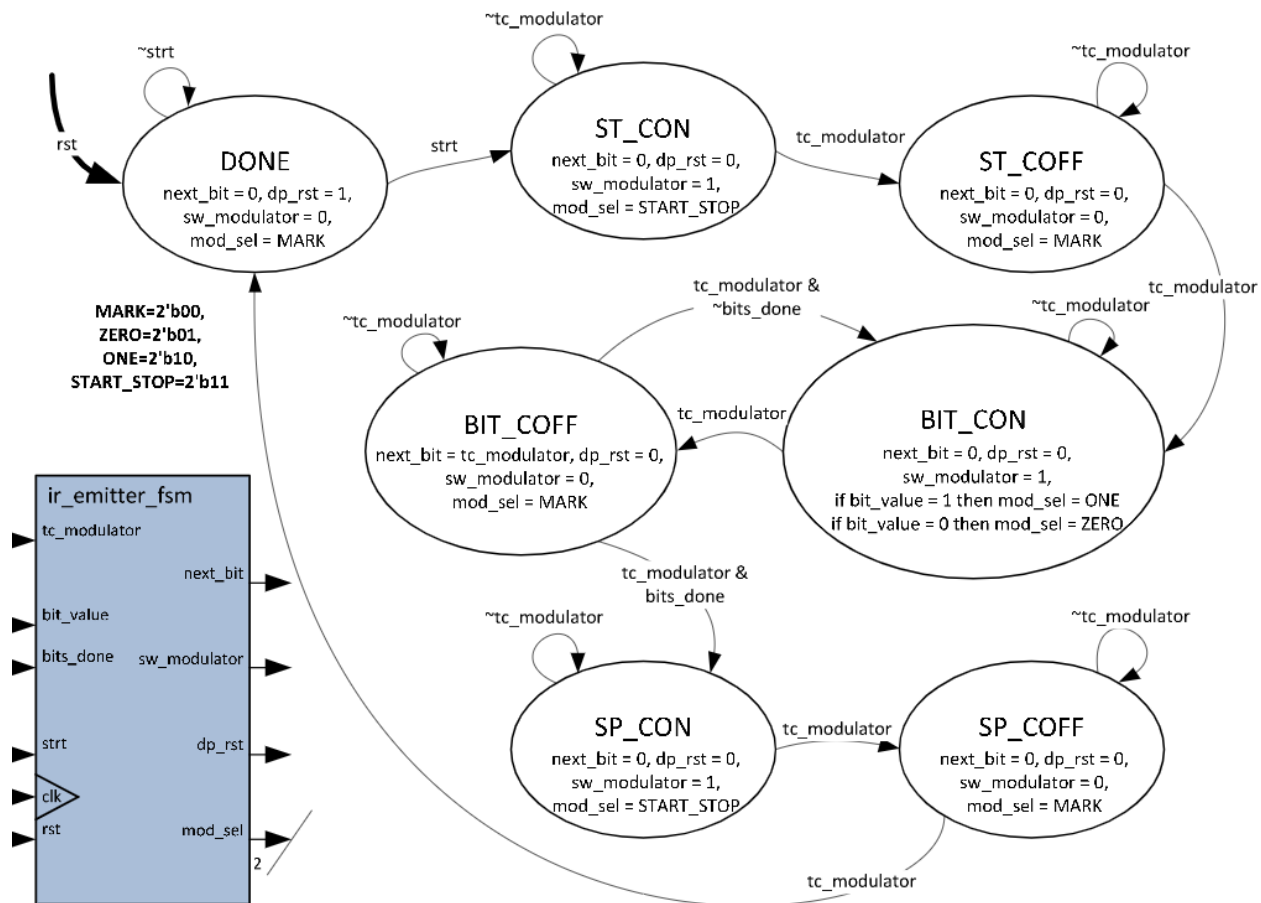
Lab 10 delivers a picture of the entire top level block diagram and discusses how these individual modules that were created in the previous labs, get instantiated:



[Block Diagram of the IR Transmitter as shown in the Lecture]

Essentially, the bit length goes through a synchronizer that feeds into the pulse button debounce module that was provided. The pulse button debounce then creates triggers the start bit in the IR Finite State Machine, which modulates based on the IR Emitter Datapath signal that generates the 38kHz from lab 7, this feeds back into the Finite State Machine starting the ST_CON and ST_COFF stages (which signify that the message from the input is starting), the signal goes low to initialize, then the state

BIT_CON and BIT_COFF feed the signal inputted by the switches that go through the decoder module from lab 8, the message then completes and bit done is high, so the Finite State Machine triggers SP_CON and SP_COFF to signify the stop bit. Finally the State Machine goes back to the DONE stage, however, if rest is activated throughout the process, SP_CON and SP_COFF are activated and the process restarts by going to DONE (initialization), then ST_CON and ST_COFF and so on until the input is complete. The individual module of the IR Emitter Finite State Machine is based around this State Diagram from the manual:



[IR Emitter Finite State Machine Block Diagram and State Diagram]

Further details about how each state functions can be explored from the chart below:

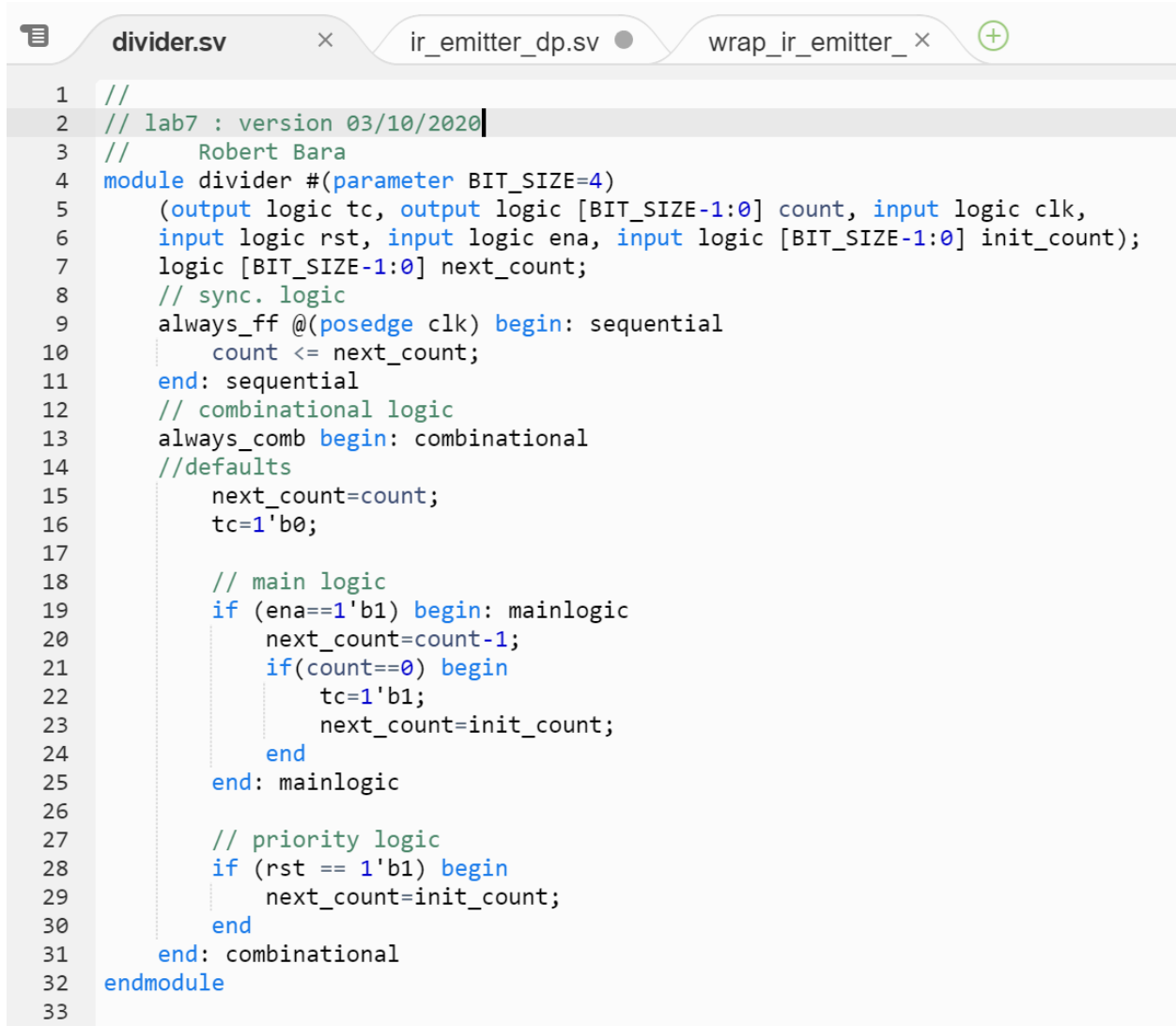
State	Description	Functional Details
DONE	Done	Arrives here after a reset or transmission complete. Activates the reset signal on the datapath module (<i>db_rst</i>). Sets the modulation select for the next state to be <i>MARK</i> . Waits here until a start (<i>strt</i>) input signal.
ST_CON	Start, Carrier On	Turns the carrier on for the <i>MARK</i> time. Sets the modulation select for the next state to be <i>START_STOP</i> . Waits until the carrier on time is complete (<i>tc_modulator</i> = 1).
ST_COFF	Start, Carrier Off	Turns the carrier off for the <i>START_STOP</i> time. Sets the modulation select for the next state to be <i>MARK</i> . Waits until the carrier off time is complete (<i>tc_modulator</i> = 1).
BIT_CON	Bit Carrier On	Turns the carrier on for the <i>MARK</i> time. Sets the modulation select for the next state to be based on the input <i>bit_value</i> : either <i>ONE</i> or <i>ZERO</i> . Waits until the carrier on time is complete (<i>tc_modulator</i> = 1).
BIT_COFF	Bit Carrier Off	Turns the carrier off for the <i>ONE</i> or <i>ZERO</i> time. Sets the modulation select for the next state to be <i>MARK</i> . Waits until the carrier off time is complete (<i>tc_modulator</i> = 1). When complete, it checks the <i>bits_done</i> input signal to determine the

State	Description	Functional Details
		next state: either <i>Bit Carrier On</i> or <i>Stop Carrier On</i> .
SP_CON	Stop, Carrier On	Turns the carrier on for the <i>MARK</i> time. Sets the modulation select for the next state to be <i>START_STOP</i> . Waits until the carrier on time is complete (<i>tc_modulator</i> = 1).
SP_COFF	Stop, Carrier Off	Turns the carrier off for the <i>START_STOP</i> time. Sets the modulation select for the next state to be <i>MARK</i> or actually: <i>Don't Care</i> . Waits until the carrier off time is complete (<i>tc_modulator</i> = 1). When leaving this state, the transmission is complete – so go to the <i>Done</i> state.

[Table Specifying what each stage of the FSM does]

Results (45 points – see sub-sections)

Throughout the course of the past few weeks I have been able to complete and get checked off for every lab, so naturally my code came together and made sense based on the simulations. The labs definitely gave me a better understanding of how timing diagrams and GTK Waves work to read them, that felt like the only confusing part usually for the lab, just understanding the simulation, the overall syntax was fairly easy in my opinion.

Design Code (15 points)


```

1  //
2  // lab7 : version 03/10/2020
3  //   Robert Bara
4  module divider #(parameter BIT_SIZE=4)
5      (output logic tc, output logic [BIT_SIZE-1:0] count, input logic clk,
6       input logic rst, input logic ena, input logic [BIT_SIZE-1:0] init_count);
7      logic [BIT_SIZE-1:0] next_count;
8      // sync. logic
9      always_ff @(posedge clk) begin: sequential
10         count <= next_count;
11     end: sequential
12     // combinational logic
13     always_comb begin: combinational
14         //defaults
15         next_count=count;
16         tc=1'b0;
17
18         // main logic
19         if (ena==1'b1) begin: mainlogic
20             next_count=count-1;
21             if(count==0) begin
22                 tc=1'b1;
23                 next_count=init_count;
24             end
25         end: mainlogic
26
27         // priority logic
28         if (rst == 1'b1) begin
29             next_count=init_count;
30         end
31     end: combinational
32 endmodule
33

```

[Figure 1a: Divider Module (lab7)]

Tackling the divider module is quite simple, it uses a sequential logic to change count to next count and then uses combinational logic to follow the block diagram specifications and description found on page 2 of the lab report. This is pretty straight forward.


```

1 //
2 // lab7 : version 03/10/2020
3 // Robert Bara
4 `timescale 1ns/1ps
5
6 module ir_emitter_dp (output logic emitter_out, output logic tc_modulator, input logic clk,
7   input logic rst, input logic ena, input [1:0] mod_sel, input logic sw_modulator);
8
9   // enter your code here
10  logic tc_76k, sw_carrier, tc_carrier;
11  logic [5:0] mod_init_count;
12
13  divider #(.BIT_SIZE(11)) u1(.tc(tc_76k),.count(),.ena(ena),.rst(rst),.clk(clk),.init_count(11'd1315));
14  divider #(.BIT_SIZE(2)) u2(.tc(tc_carrier),.count(sw_carrier),.init_count(1'd1),.ena(tc_76k),.clk(clk),.rst(rst));
15  divider #(.BIT_SIZE(6)) u3(.tc(tc_modulator),.count(),.init_count(mod_init_count),.ena(tc_carrier),.clk(clk),.rst(rst));
16  and u4(emitter_out,sw_modulator,sw_carrier);
17
18  always_comb begin
19    case(mod_sel)
20      2'b00: mod_init_count=6'd5;
21      2'b01: mod_init_count=6'd9;
22      2'b10: mod_init_count=6'd20;
23      default: mod_init_count=6'd38;
24    endcase
25  end
26 endmodule
27

```

[Figure 1b: Ir Emitter module (lab7)]

The Ir Emitter module can be accomplished by instantiating the divider module 3 times as shown in the corresponding block diagram, and I used an AND gate for emitter out, though an assign statement could have also probably worked. The 4x1 multiplexer is easily accomplished with a case statement using mod_sel as the input as shown in the block diagram.

```

1 //
2 // lab8 : version 03/17/2020
3 // Robert Bara
4 `timescale 1ns/1ps
5
6 module sw_decode (output logic bits_done, output logic bit_value,
7     input logic next_bit, input logic [10:0] sw,
8     input logic clk, input logic rst);
9
10    // enter your code here
11    logic [4:0] bit_count;
12
13
14    divider #(5) u_bd_divider(.tc(),.count(bit_count),.init_count(5'd31),.ena(next_bit),.clk,.rst);
15
16    always_comb begin
17
18        bits_done=1'b0;
19        bit_value=1'b0;
20
21        case (bit_count[3:0])
22            4'd0: bit_value=~(sw[8]^sw[4]^sw[0]); //LRC bit 0
23            4'd1: bit_value=~(sw[9]^sw[5]^sw[1]); //LRC bit 1
24            4'd2: bit_value=~(sw[10]^sw[6]^sw[2]); //LRC bit 2
25            4'd3: bit_value=~(bit_count[4]^sw[7]^sw[3]); //LRC bit 3
26            4'd4: bit_value=sw[0]; //Data bit 0
27            4'd5: bit_value=sw[1]; //Data bit 1
28            4'd6: bit_value=sw[2]; //Data bit 2
29            4'd7: bit_value=sw[3]; //Data bit 3
30            4'd8: bit_value=sw[4]; //Mode bit 0
31            4'd9: bit_value=sw[5]; //Mode bit 1
32            4'd10: bit_value=sw[6]; //Mode bit 2
33            4'd11: bit_value=sw[7]; //Address bit
34            4'd12: bit_value=sw[8]; //Channel LS bit
35            4'd13: bit_value=sw[9]; //Channel MS bit
36            4'd14: bit_value=sw[10]; //Escape bit
37            default: bit_value=~bit_count[4]; //Toggle bit. Needs inverting as it starts with logic one: init_count=5'd31
38        endcase
39
40        if (bit_count[3:0]==0 && next_bit==1) begin
41            bits_done=1'b1;
42        end
43    end
44 endmodule

```

[Figure 1d: Switch Decoder module Design Code (lab 8)]

The switch decoder is a straight forward transition from the block diagram because it instantiates a divider module and then connects it to the “bit encoding always block”, really this is just the combinational logic that follows the truth table on page 3 of the Lab 8 manual, so the easiest way to accomplish this is by creating a case statement for bit count (don’t forget to initialize bits_done and bits_value to 1'b0 before creating the cases), then for the special instance of when bit count=3'b0 and next_bit=1'b1, an if statement can be used to trigger bits_done as high 1'b1, indicating the message is decoded.

```
ir_emitter_fsm.sv
1 //
2 // lab10 : version 03/30/2020
3 // Robert Bara
4 `timescale 1ns/1ps
5
6 module ir_emitter_fsm (output logic sw_modulator, output logic [1:0] mod_sel, output logic dp_rst,
7     output logic next_bit, input logic clk, input logic rst, input logic strt,
8     input logic tc_modulator, input logic bits_done, input logic bit_value);
9
10 //Defining and assigning values to the parameters that are used in mod_sel
11 enum logic [2:0] {DONE, ST_CON, ST_COFF, BIT_COFF, BIT_CON, SP_CON, SP_COFF} state, next_state;
12 logic [1:0] MARK;
13 logic [1:0] ZERO;
14 logic [1:0] ONE;
15 logic [1:0] START_STOP;
16
17 assign MARK=2'b00;
18 assign ZERO=2'b01;
19 assign ONE=2'b10;
20 assign START_STOP=2'b11;
21
22 //Synchronous Logic
23 always_ff @(posedge clk) begin
24     state<=next_state;
25 end
26 //Combinational Logic
27 always_comb begin
28     //defaults
29     next_state=state;
30     next_bit=1'b0;
31     dp_rst=1'b0;
32     sw_modulator=1'b0;
33     mod_sel=MARK;
```

[Figure 1E: Emitter Finite State Machine Design Code (Lab 10) Part 1]

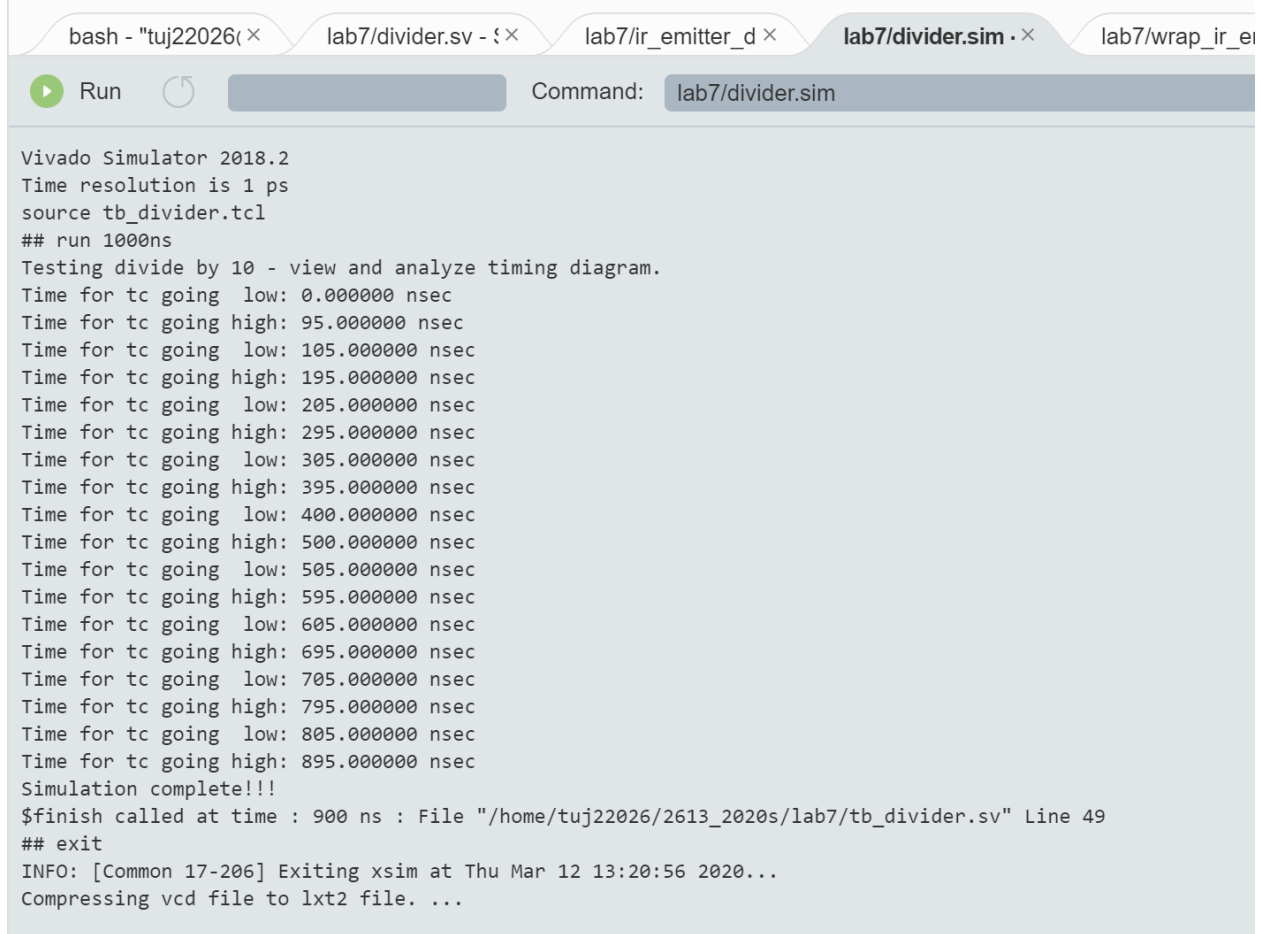
```

33     mod_sel=MARK;
34     // mainlogic
35     case(state)
36     default: begin
37         dp_rst=1'b1;
38         if(strt==1'b1) next_state=ST_CON;
39     end
40     ST_CON: begin
41         sw_modulator=1'b1;
42         mod_sel=START_STOP;
43         if(tc_modulator==1'b1) next_state=ST_COFF;
44     end
45     ST_COFF: begin
46         if(tc_modulator==1'b1) next_state=BIT_CON;
47     end
48     BIT_CON: begin
49         sw_modulator=1'b1;
50         if (bit_value==1'b1) mod_sel=ONE;
51         else mod_sel=ZERO;
52         if(tc_modulator==1'b1) next_state=BIT_COFF;
53     end
54     BIT_COFF: begin
55         next_bit=tc_modulator;
56         if (tc_modulator==1'b1 && bits_done== 1'b0) next_state=BIT_CON;
57         else if (tc_modulator==1'b1 && bits_done== 1'b1) next_state=SP_CON;
58     end
59     SP_CON: begin
60         sw_modulator=1'b1;
61         mod_sel=START_STOP;
62         if (tc_modulator==1'b1) next_state=SP_COFF;
63     end
64     SP_COFF: begin
65         if(tc_modulator==1'b1) next_state=DONE;
66     end
67     endcase
68     // priority
69     if (rst ==1'b1) begin
70         next_state=DONE;
71     end
72 end
73
74 endmodule
75

```

[Figure 1E: Emitter Finite State Machine Design Code (Lab 10) Part 2]

The emitter design uses a posedge clock signal to trigger the finite state machine to go to the next state. The states must be defined and initialized for mod_sel to use them in the combinational logic. Combinational logic can be used to initialize each state, and then create a case statement that will act as a controller. The controller is based on the state diagram and can be simplified by defaulting specific parameters such as next_bit = 1'b0 to reduce the amount of syntax. Finally, the reset is implemented with an if statement.

Simulation Results (15 points)

The screenshot shows the Vivado Simulator 2018.2 interface with a terminal window. The terminal displays the command 'lab7/divider.sim' and the simulation results. The results show a series of timing measurements for a divider module, with the simulation completing at 900 ns.

```
bash - "tuj22026( x lab7/divider.sv - f x lab7/ir_emitter_d x lab7/divider.sim x lab7/wrap_ir_ei
Run Command: lab7/divider.sim
Vivado Simulator 2018.2
Time resolution is 1 ps
source tb_divider.tcl
## run 1000ns
Testing divide by 10 - view and analyze timing diagram.
Time for tc going low: 0.000000 nsec
Time for tc going high: 95.000000 nsec
Time for tc going low: 105.000000 nsec
Time for tc going high: 195.000000 nsec
Time for tc going low: 205.000000 nsec
Time for tc going high: 295.000000 nsec
Time for tc going low: 305.000000 nsec
Time for tc going high: 395.000000 nsec
Time for tc going low: 400.000000 nsec
Time for tc going high: 500.000000 nsec
Time for tc going low: 505.000000 nsec
Time for tc going high: 595.000000 nsec
Time for tc going low: 605.000000 nsec
Time for tc going high: 695.000000 nsec
Time for tc going low: 705.000000 nsec
Time for tc going high: 795.000000 nsec
Time for tc going low: 805.000000 nsec
Time for tc going high: 895.000000 nsec
Simulation complete!!!
$finish called at time : 900 ns : File "/home/tuj22026/2613_2020s/lab7/tb_divider.sv" Line 49
## exit
INFO: [Common 17-206] Exiting xsim at Thu Mar 12 13:20:56 2020...
Compressing vcd file to lxt2 file. ...
```

[Figure 2a: Simulation results for Divider Module (lab 7)]

```

bash - "tuj22026(× lab7/divider.sv - {× lab7/ir_emitter_d × lab7/divider.sim - × lab7/wrap_ir_em ×
Run Command: lab7/wrap_ir_emitter_dp.sim

Time resolution is 1 ps
source tb_wrap_ir_emitter_dp.tcl
## run 2500us
Index 1: Frequency 37.99 kHz, Period 0.0263 msec
Index 2: Frequency 37.99 kHz, Period 0.0263 msec
Index 3: Frequency 37.99 kHz, Period 0.0263 msec
Index 4: Frequency 37.99 kHz, Period 0.0263 msec
Index 5: Frequency 37.99 kHz, Period 0.0263 msec
Index 6: Frequency 0.95 kHz, Period 1.0528 msec
Index 7: Frequency 37.99 kHz, Period 0.0263 msec
Index 8: Frequency 37.99 kHz, Period 0.0263 msec
Index 9: Frequency 37.99 kHz, Period 0.0263 msec
Index 10: Frequency 37.99 kHz, Period 0.0263 msec
Index 11: Frequency 37.99 kHz, Period 0.0263 msec
Index 12: Frequency 3.45 kHz, Period 0.2895 msec
Index 13: Frequency 37.99 kHz, Period 0.0263 msec
Index 14: Frequency 37.99 kHz, Period 0.0263 msec
Index 15: Frequency 37.99 kHz, Period 0.0263 msec
Index 16: Frequency 37.99 kHz, Period 0.0263 msec
Index 17: Frequency 37.99 kHz, Period 0.0263 msec
Index 18: Frequency 1.73 kHz, Period 0.5790 msec
Index 19: Frequency 37.99 kHz, Period 0.0263 msec
Index 20: Frequency 37.99 kHz, Period 0.0263 msec
Simulation complete!!!
$finish called at time : 2382965 ns : File "/home/tuj22026/2613_2020s/lab7/tb_wrap_ir_emitter_dp.sv" Line 59
## exit
INFO: [Common 17-206] Exiting xsim at Thu Mar 12 14:16:34 2020...
Compressing vcd file to lxt2 file. ...

```

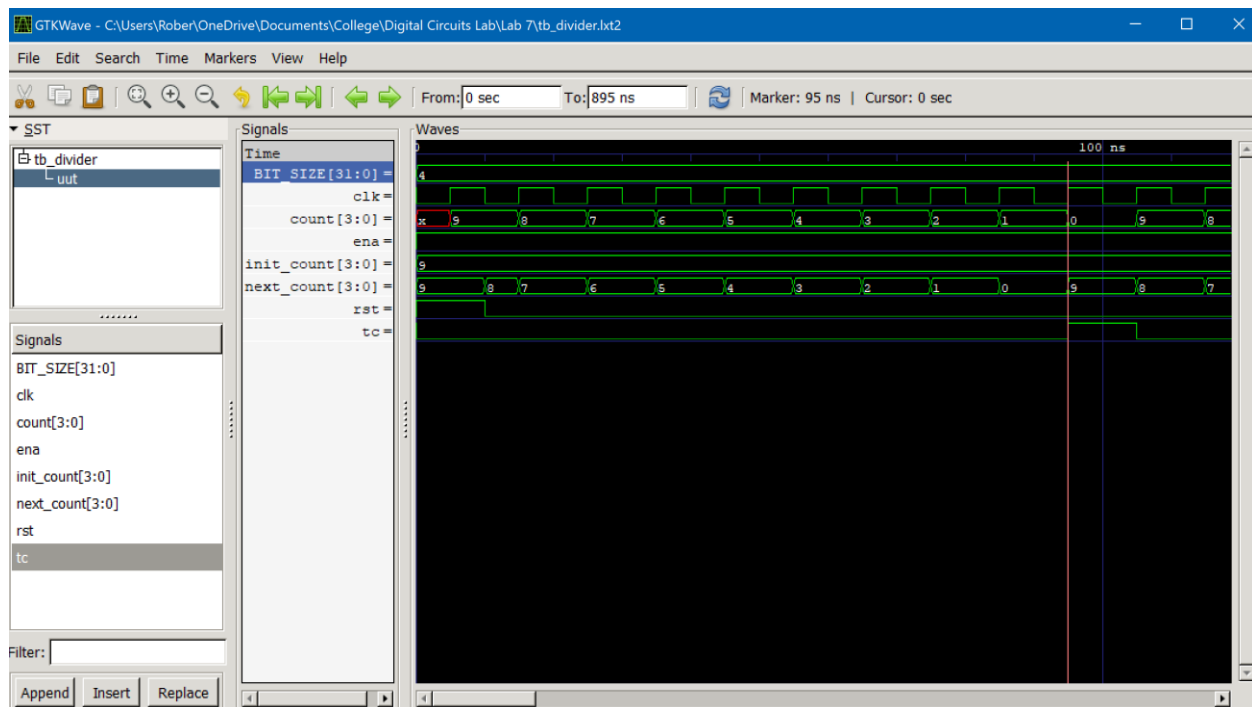
[Figure 2b: Simulation for Ir Emitter Module (lab 7)]

```

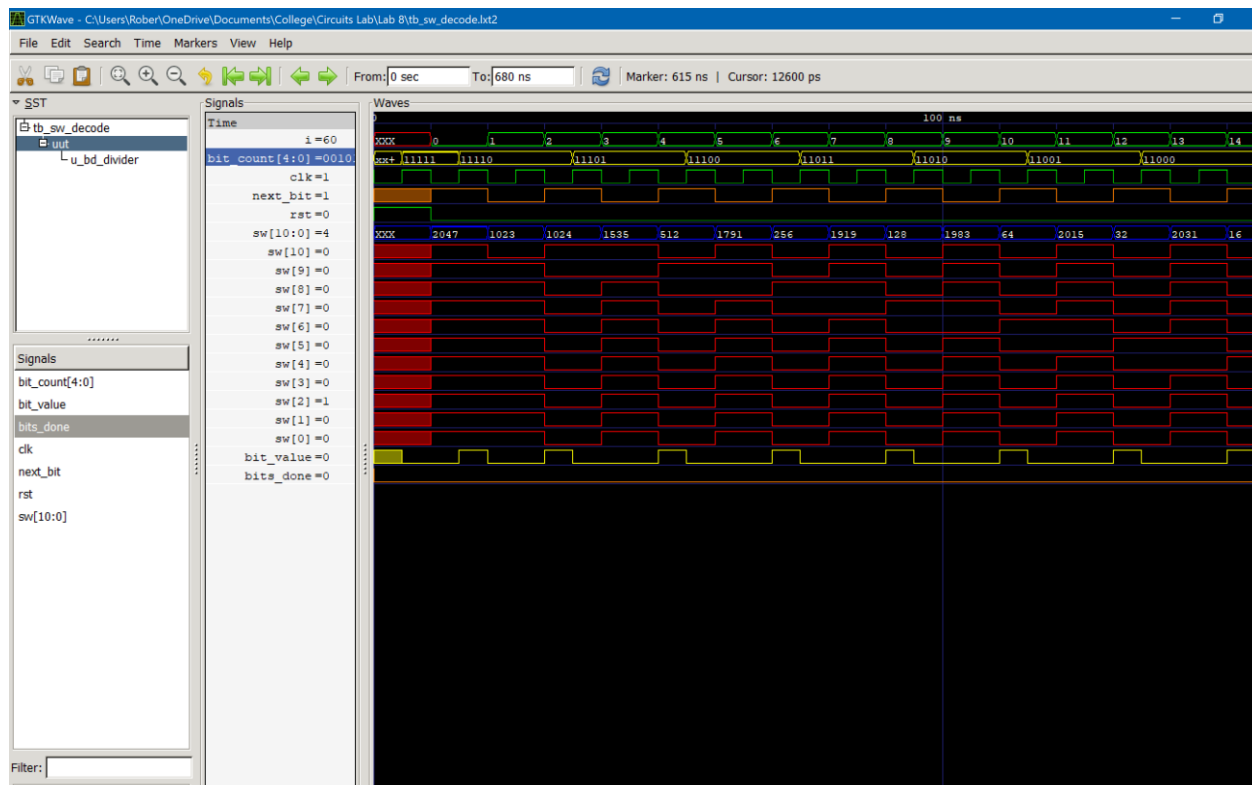
Simulation complete - no mismatches!!!
$finish called at time : 680 ns : File "/home/tuj22026/2613_2020s/lab8/tb_sw_decode.sv" Line 73
## set systemTime [clock seconds]
## puts "----Simulation date/time: [clock format $systemTime -format %D] [clock format $systemTime -format %H:%M:%S]"
----Simulation date/time: 03/17/2020 14:41:19
## exit

```

[Figure 2c: Simulation for Switch Decoder (lab8)]



[Figure 2d: Timing diagram for the Divider module (lab 7)]



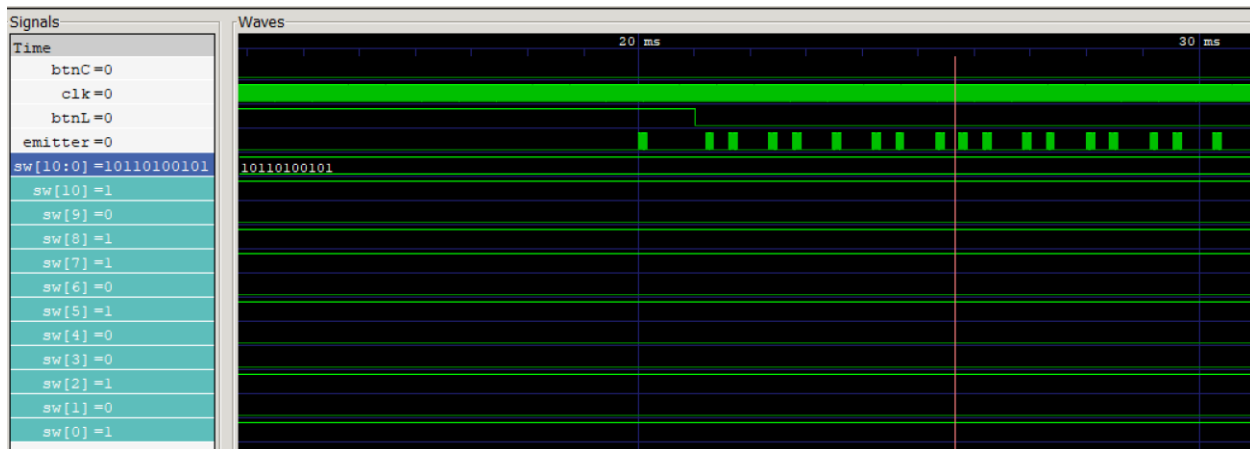
[Figure 2e: Timing diagram for the Switch Decoder module (lab 8)]

```

## run 35000us
Bit length: 1184.4000 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 421.1200 usec - Low bit
Bit length: 710.6400 usec - High bit
Bit length: 1201.4950 usec - Start/Stop bit
Simulation complete!!!
$finish called at time : 33000620 ns : File "/home/tuj22026/2613_2020s/lab10/tb_ir_transmitter_top.sv" Line 83
run: Time (s): cpu = 00:00:00.07 ; elapsed = 00:00:47 . Memory (MB): peak = 1359.281 ; gain = 0.000 ; free physical = 4567 ; free virtual = 9638
## set systemTime [clock seconds]
## puts "----Simulation date/time: [clock format $systemTime -format %D] [clock format $systemTime -format %H:%M:%S]"
---Simulation date/time: 03/31/2020 14:22:40
## exit

```

[Figure 2f: Simulation of Code for IR Emitter FSM module (lab 10)]



[Figure 2G: GTK Wave for IR Emitter FSM module (Lab 10)]

Hardware Implementation (10 points)

My lab 7 simulations were demonstrated to Yamen at 2:22pm on Thursday during the lab period of 3/12/20.

My lab 8 simulations were demonstrated to Franka at 2:06pm on Thursday during the lab period of 3/19/20.

My lab 10 simulations were demonstrated to Yamen at 1:26pm on Thursday during the lab period of 4/2/20.

Conclusion (10 points)

Overall, by creating a divider module, switch decoder module, and basically reusing previous labs beforehand, the labs created a foundation for this LEGO IR Transmitter project. These labs also

introduced how Finite State Machines operate and I was able to create new modules to get the project done, in conjunction with the modules that were built upon in previous labs. I was able to successfully generate the IR signal from the switch input by successfully creating and instantiating each module based on the overall top-block diagram, and I clarified any confusion on the theory by looking at the GTK Waves and understanding where I went wrong for the debugging process.