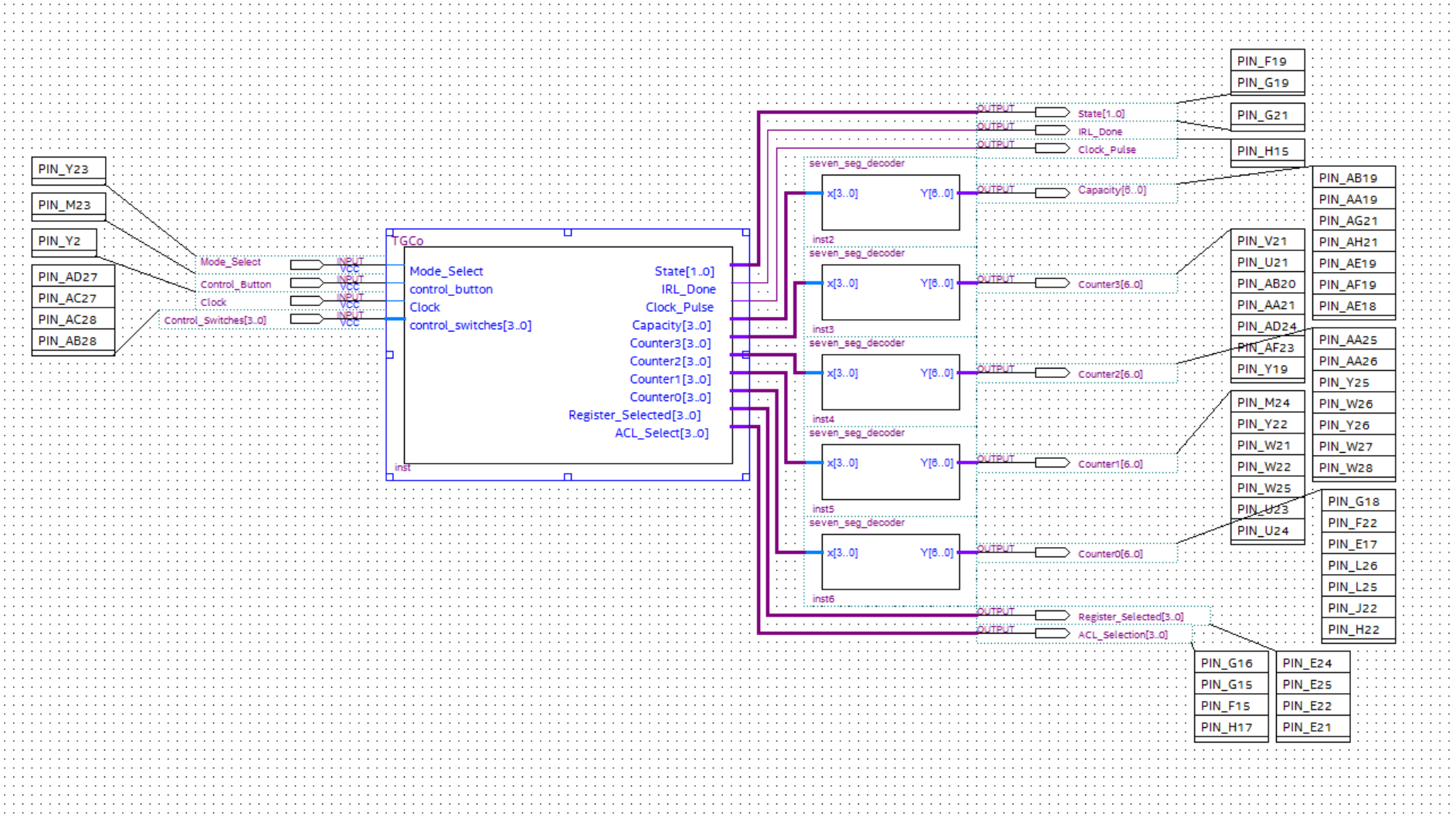
Final Project Report
Sean Griffen
CprE 281 Fall 2018
Section V

For my final project, I followed the specifications for the provided project, Project #3: Traffic Light Fairness System. The following report contains all block diagram files and Verilog code used to complete this project, starting with the top-level entity of every module and ending with the lowest-level entity.

Module FPM:


PIN_Y23
PIN_M23
PIN_Y2
PIN_AD27
PIN_AC27
PIN_AC28
PIN_AB28
Mode_Select
Control_Button
Clock
Control_Switches[3..0]
INPUT
VCC
TGCo
Mode_Select
control_button
Clock
control_switches[3..0]
State[1..0]
IRL_Done
Clock_Pulse
Capacity[3..0]
Counter3[3..0]
Counter2[3..0]
Counter1[3..0]
Counter0[3..0]
Register_Selected[3..0]
ACL_Select[3..0]
inst
seven_seg_decoder
x[3..0]
Y[6..0]
inst2
inst3
inst4
inst5
inst6
OUTPUT
State[1..0]
IRL_Done
Clock_Pulse
Capacity[6..0]
Counter3[6..0]
Counter2[6..0]
Counter1[6..0]
Counter0[6..0]
Register_Selected[3..0]
ACL_Selection[3..0]
PIN_F19
PIN_G19
PIN_G21
PIN_H15
PIN_AB19
PIN_AA19
PIN_AG21
PIN_AH21
PIN_AE19
PIN_AF19
PIN_AE18
PIN_V21
PIN_U21
PIN_AB20
PIN_AA21
PIN_AD24
PIN_AF23
PIN_Y19
PIN_AA25
PIN_AA26
PIN_Y25
PIN_W26
PIN_Y26
PIN_W27
PIN_W28
PIN_M24
PIN_Y22
PIN_W21
PIN_W22
PIN_W25
PIN_U23
PIN_U24
PIN_G18
PIN_F22
PIN_E17
PIN_L26
PIN_L25
PIN_J22
PIN_H22
PIN_G16
PIN_G15
PIN_F15
PIN_H17
PIN_E24
PIN_E25
PIN_E22
PIN_E21

This is the diagram of the module Final_Project_Machine or FPM, as it is named in the project files and .bsf's. It is the module that connects the desired switch and button inputs and outputs of the TGCo module, which will be described in detail later in this report. Here is a description of the inputs and outputs.

Inputs:

Mode_Select:

This is controlled by switch 17 on the board and controls the mode of the system; switch = 0 being mode A (mode specified in the project description) and switch = 1 being mode B (mode specified by the project description).

Control_Button:

This is controlled by key 0 on the board and is only used to select the capacity for the lane, starting from lane 0 to lane 3, during the initial register file loading mode (mode specified in the project description). When the button is pushed, the program assumes that you are submitting the desired capacity for that lane and updates the register file on the next clock cycle.

Clock:

This is the provided 50MHz clock on the board. It is used to provide clocks to the finite state machine, 4_4bit_Register_File module, TLCF module, IRL_Machine module, and ACL_Machine module, all of which will be described later in this report.

Control_Switches:

These switches are mapped to switches 0, 2, 3, and 3 on the board and, while in the initial register file load mode, control the capacity for the lanes, and, while in Mode A or Mode B, control which lane gets cars added to it, all switches off being no lane selected, switch 0 on being lane 0 selected, switch 1 on being lane 1 selected, switch 2 on being lane 2 selected, and switch 3 being lane 3 selected.

Outputs:

State:

This is mapped to LED's RLED0, and RLED1 on the board and correspond to the state TGCo is in both LEDs off being the capacity load state, LED RLED0 on signifying that the machine is in Mode A, and LED RLED1 on signifying that the machine is in Mode B.

IRL_Done

This is mapped to LED GLED7 on the board and, when on, signifies that the initial capacity loading is done, and TGCo is free to switch between Mode A and Mode B using the Mode_Select switches.

Capacity:

This is provided by a seven segment decoder and outputs to seven segment display HEX4. This corresponds to the capacity that the currently selected lane is bound to.

Counter 3:

This is provided by a seven segment decoder, the same design used by output Capacity, and outputs to seven segment display HEX3 on the board. This displays the current number of cars in lane 3.

Counter 2:

This is provided by a seven segment decoder, the same design used by output Capacity, and outputs to seven segment display number HEX2 on the board. This displays the current number of cars in lane 2.

Counter 1:

This is provided by a seven segment decoder, the same design used by output Capacity, and outputs to seven segment display number HEX1 on the board. This displays the current number of cars in lane 1.

Counter 0:

This is provided by a seven segment decoder, the same design used by output Capacity, and outputs to seven segment display number HEX0 on the board. This displays the current number of cars in lane 0.

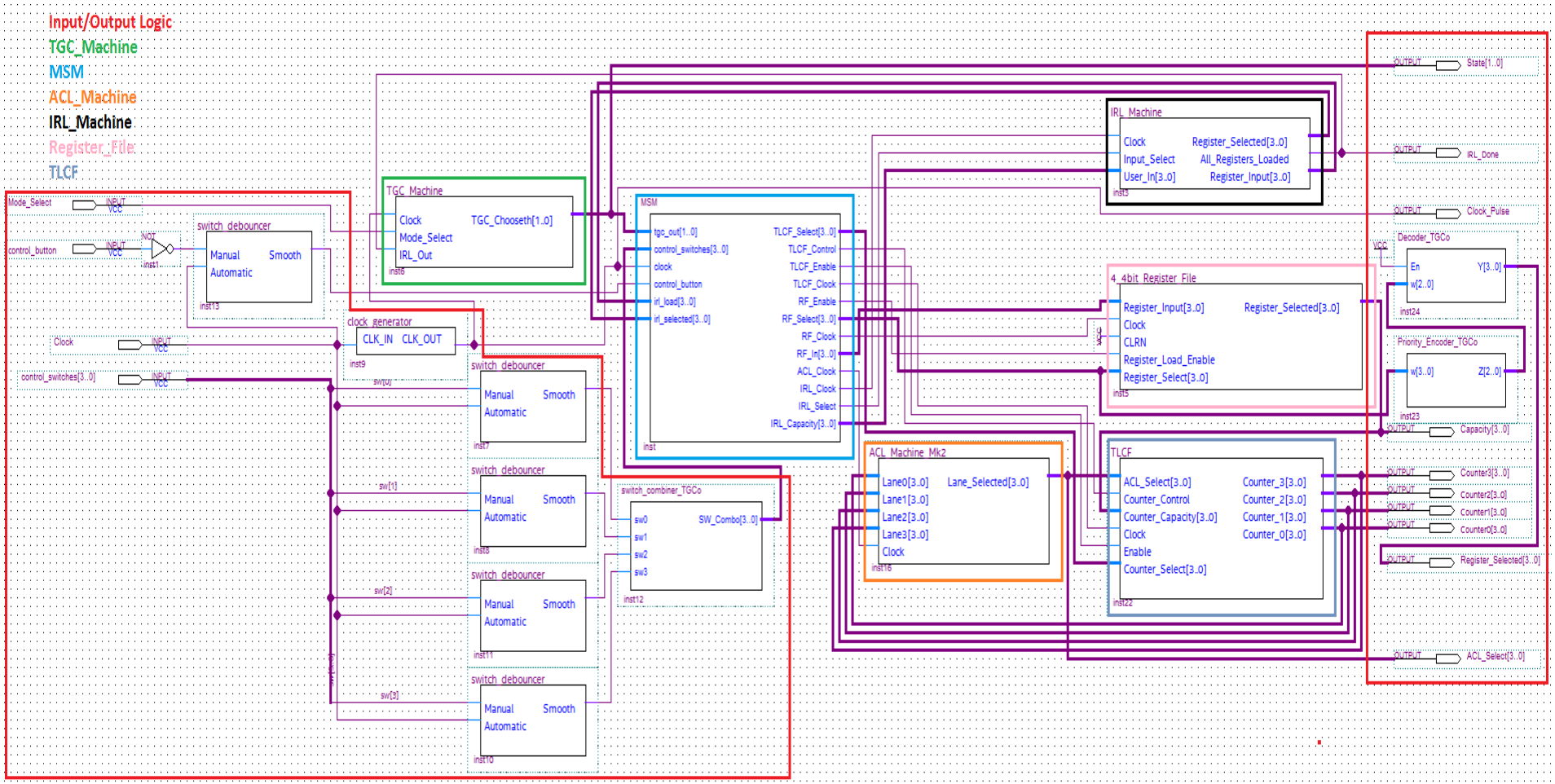
Register_Selected:

This is mapped to LED's GLED 0, GLED1, GLED2, and GLED 3 on the board and correspond to the current register and lane currently selected by the input Control_Switches. All LED's off being no register or lane selected, LED GLED 0 on being register 0 and lane 0 is selected, LED GLED 1 on being register 1 and lane 1 is selected, LED GLED 2 on being register 2 and lane 2 is selected, and LED GLED 3 on being register 3 and lane 3 is selected.

ACL_Selection:

This is mapped to LED's RLED13, RLED14, RLED15, and RLED16 on the board and corresponds to which lane currently has a green light during Mode B, LED RLED13, on being lane 0 has a green light, LED RLED14, on being lane 1 has a green light, LED RLED15, on being lane 2 has a green light, and LED RLED16, on being lane 3 has a green light. The lights are only accurate in Mode B and do not reflect which lane has a green light in Mode A, as no lane has a green light in Mode A.

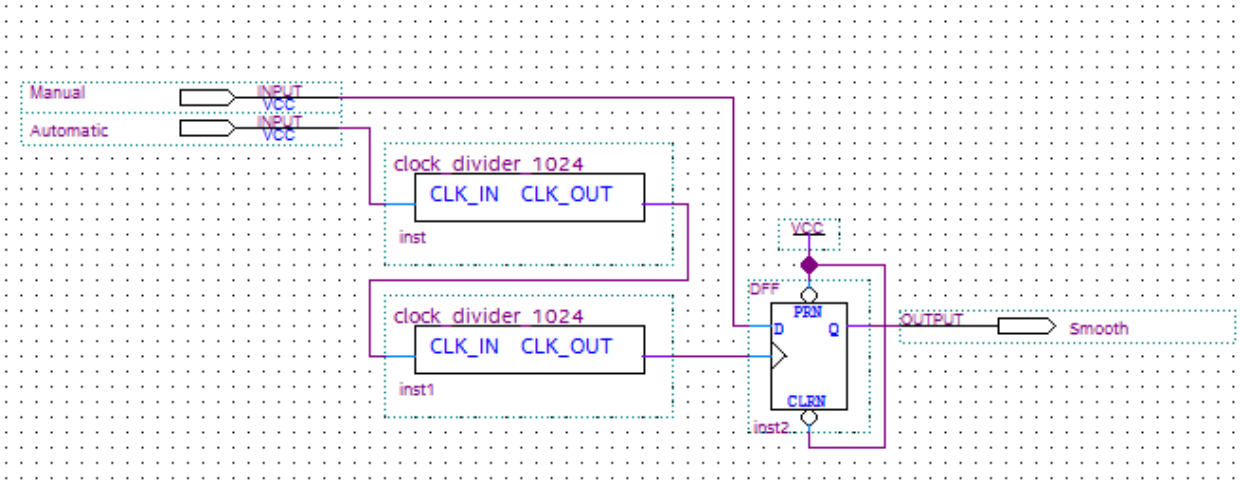
Module TGCo:


Input/Output Logic
TGC_Machine
MSM
ACL_Machine
IRL_Machine
Register_File
TLCF
Mode_Select
control_button
Clock
control_switches[3..0]
switch_debouncer
Manual
Smooth
Automatic
clock_generator
CLK_IN
CLK_OUT
TGC_Machine
Clock
Mode_Select
IRL_Out
TGC_Chooseth[1..0]
MSM
tgc_out[1..0]
control_switches[3..0]
clock
control_button
irl_load[3..0]
irl_selected[3..0]
TLCF_Select[3..0]
TLCF_Control
TLCF_Enable
TLCF_Clock
RF_Enable
RF_Select[3..0]
RF_Clock
RF_In[3..0]
ACL_Clock
IRL_Clock
IRL_Select
IRL_Capacity[3..0]
switch_combiner_TGCo
SW_Combo[3..0]
ACL_Machine_Mk2
Lane0[3..0]
Lane1[3..0]
Lane2[3..0]
Lane3[3..0]
Lane_Selected[3..0]
IRL_Machine
Input_Select
User_In[3..0]
All_Registers_Loaded
Register_Input[3..0]
Register_Selected[3..0]
4_4bit_Register_File
Register_Load_Enable
Register_Select[3..0]
TLCF
ACL_Select[3..0]
Counter_Control
Counter_Capacity[3..0]
Enable
Counter_Select[3..0]
Counter_3[3..0]
Counter_2[3..0]
Counter_1[3..0]
Counter_0[3..0]
Decoder_TGCo
Priority_Encoder_TGCo
OUTPUT
State[1..0]
IRL_Done
Clock_Pulse
Capacity[3..0]
Counter3[3..0]
Counter2[3..0]
Counter1[3..0]
Counter0[3..0]
Register_Selected[3..0]
ACL_Select[3..0]

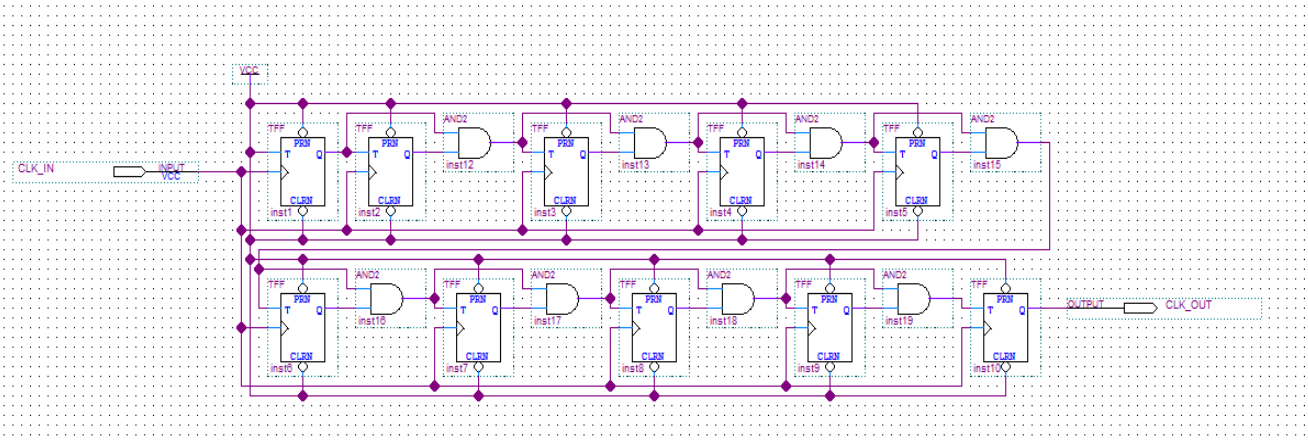
The_Grand_Combo or TGCo, as it is named in the project files and .bsf's, is where all of the segments of the project get wired together. The inputs from the FPM module correspond directly to these inputs:

FPM Input		TGCo Input
Mode_Select	→	Mode_Select
Control_Button	→	control_button
Clock	→	Clock
Control_Switches	→	control_switches

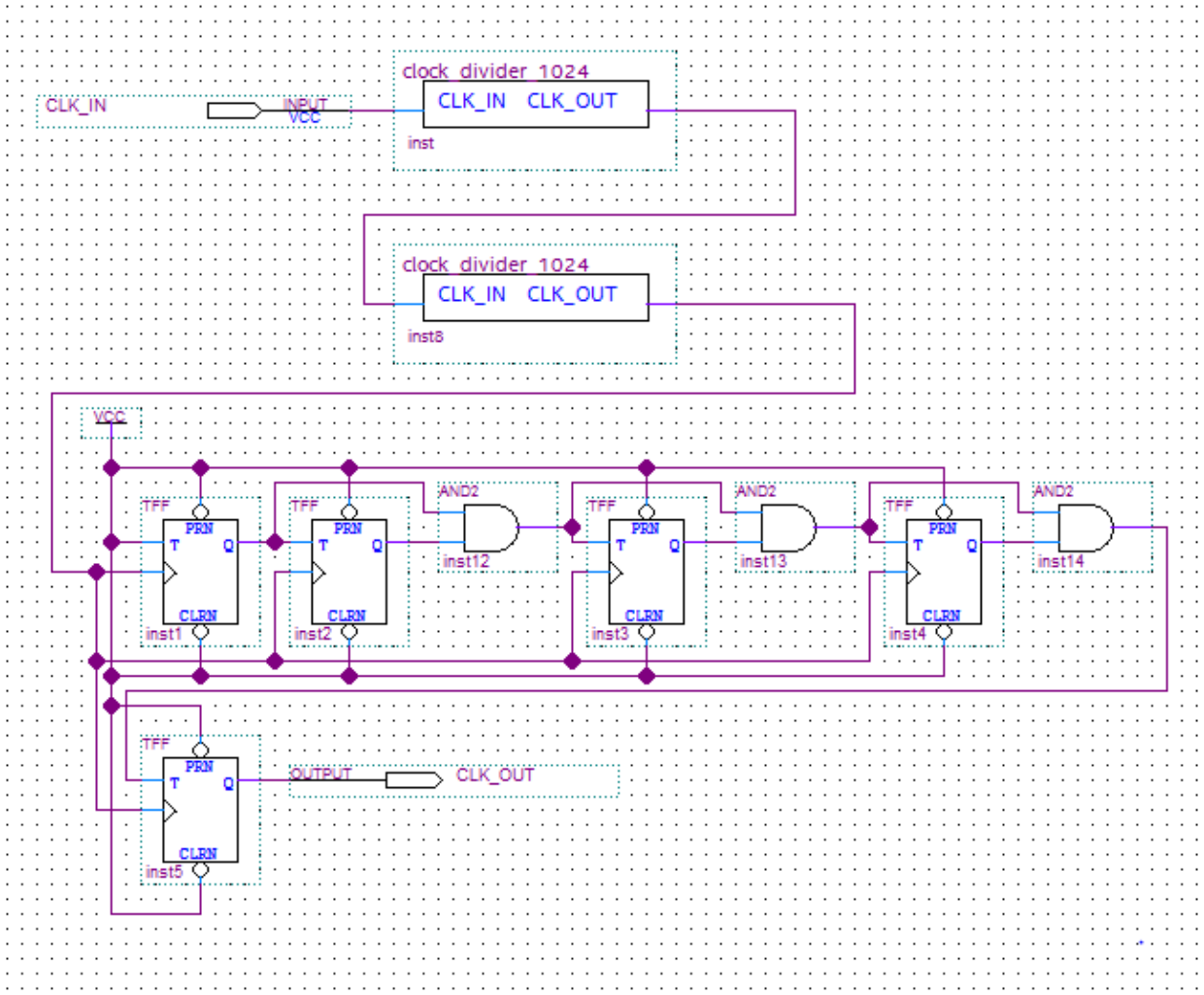
These inputs also share the same description and functions as those in the FPM module. The inputs control_switches and control_button are passed through the switch_debouncer module made in Lab 11. Here is the diagram for that:



The switch debouncer takes in a switch or button input, and a 50MHz clock that is divided by 2048 by the modules clock_divider_1024 shown here:



This smooths out the input by getting rid of any “bounce” between 1 and 0 that a switch or button may have when set to 1 or 0, ensuring a clear “1” or “0” output.
The Clock input is passed through the clock_generator module, provided to us in Lab 11. Here is the diagram for that:



This uses the same clock_divider_1024 module as in the switch_debouncer and some extra T Flip-Flops to slow the clock input from 50MHz to something more manageable like a little over 1 Hz. The T Flip-Flops act like an input divider, dividing the input speed by 2. Stacking these together slows down the input wanted to be slowed down by $2^{how\ many\ T\ flip\ flips\ there\ are}$.
This makes all of the switch flips and button presses in the machine work and make everything readable.

After the input switches and button are debounced and the clock is slowed down to something manageable, the switch inputs are put back into a bus by the module switch_combiner_TGCo shown here:

```
module switch_combiner_TGCo(sw0, sw1, sw2, sw3, SW_Combo);

   input sw0, sw1, sw2, sw3;
   output [3:0] SW_Combo;

   assign SW_Combo[3] = sw3;
   assign SW_Combo[2] = sw2;
   assign SW_Combo[1] = sw1;
   assign SW_Combo[0] = sw0;
endmodule
```

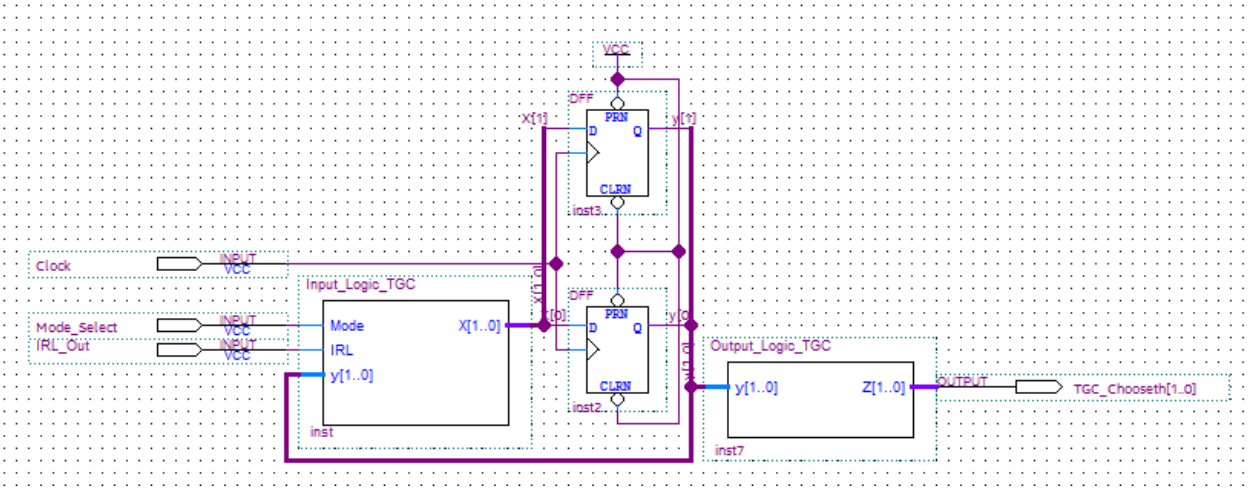
This 4-bit bus along with the button and clock, are then fed into the control_switches input to the module MSM. This module is described later. The Mode_Select input is fed into the module TGC_Machine, described later in this report.

The outputs of the TGCo module also correspond directly to the outputs of the FPM module:

FPM Input		TGCo Input
State	→	State
IRL_Done	→	IRL_Done
Clock_Pulse	→	Clock_Pulse
Capacity	→	Capacity
Counter3	→	Counter3
Counter2	→	Counter2
Counter1	→	Counter1
Counter0	→	Counter0
Register_Selected	→	Register_Selected
ACL_Selection	→	ACL_Select

These, like the inputs, share the same function as the outputs in module FPM. The only differences being in the Counter outputs. In this module, the counter outputs have not been through a seven_seg_decoder module, so they are passed as a 4-bit bus instead of a 7-bit bus.

Module TCG_Machine:



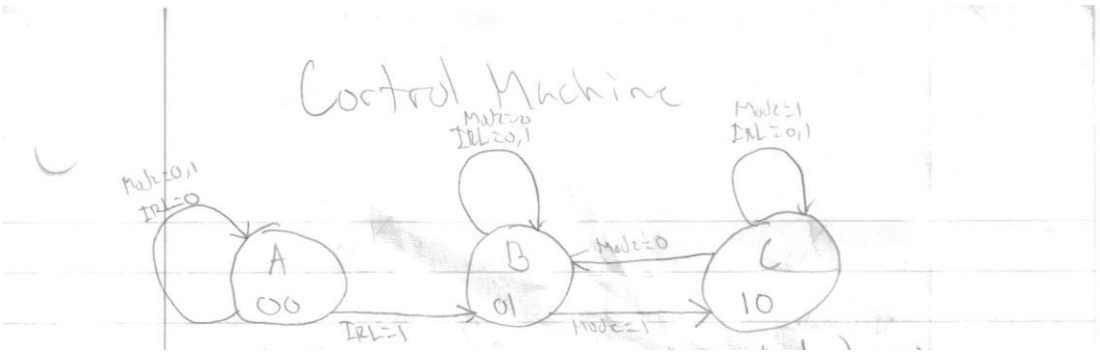
Module The_Grand_Control or TGC_Machine, as it is named in the project files and .bsf's, is the finite state machine that switches between loading capacities into the registers, and, once that is completed, switches between Mode A and Mode B based on a switch input, named Mode_Select. The switch input along with an input from module IRL, described later, are put into the block Input_Logic_TGC. The Verilog code for this module is shown here:

```
module Input_Logic_TGC (Mode, IRL, y, X);

   input Mode, IRL;
   input [1:0]y;
   output [1:0]X;

   assign X[1] = (Mode & y[0]) | (Mode & y[1]);
   assign X[0] = (~Mode & y[0]) | (~Mode & y[1]) | (IRL & ~Mode) | (IRL & ~y[1] & ~y[0]);

endmodule
```

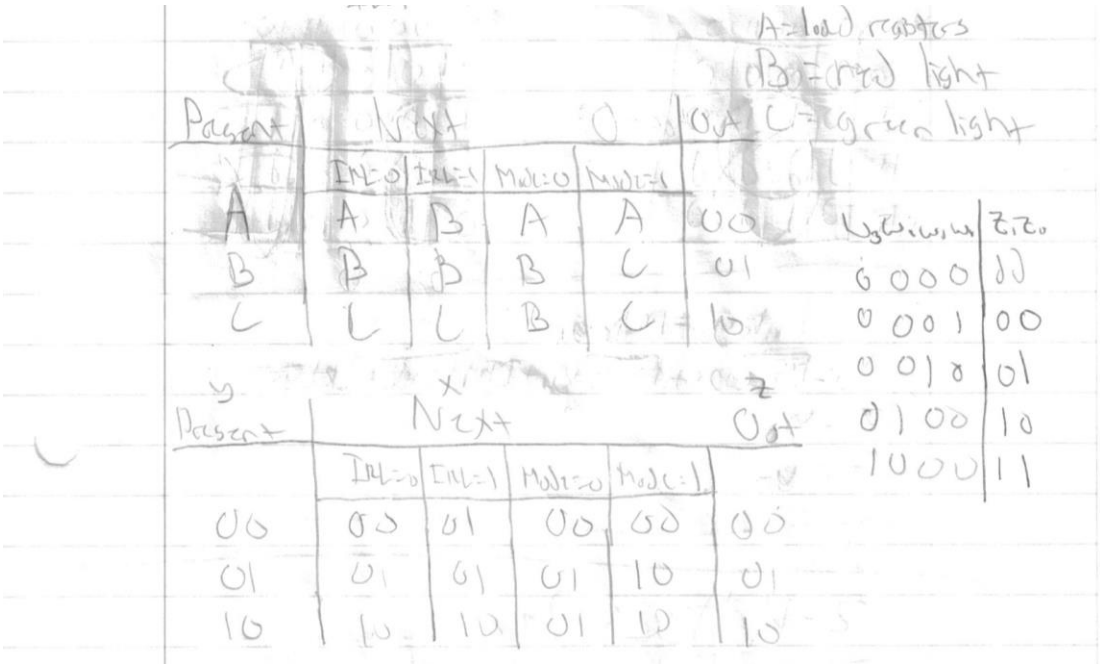
This block outputs a 2-bit bus that corresponds to the next state the machine going to be in, 00, 01, or 10. The block never outputs 11. These states map to which mode the project is in, 00 is the initial capacity loading, 01 is Mode A, and10 is Mode B. This 2-bit number is then distributed into two different D Flip-Flops which update every clock cycle, and then put into the block Output_Logic_TGC:

```
module Output_Logic_TGC (y, Z);

   input [1:0]y;
   output [1:0]Z;

   assign Z[1] = y[1];
   assign Z[0] = y[0];

endmodule
```

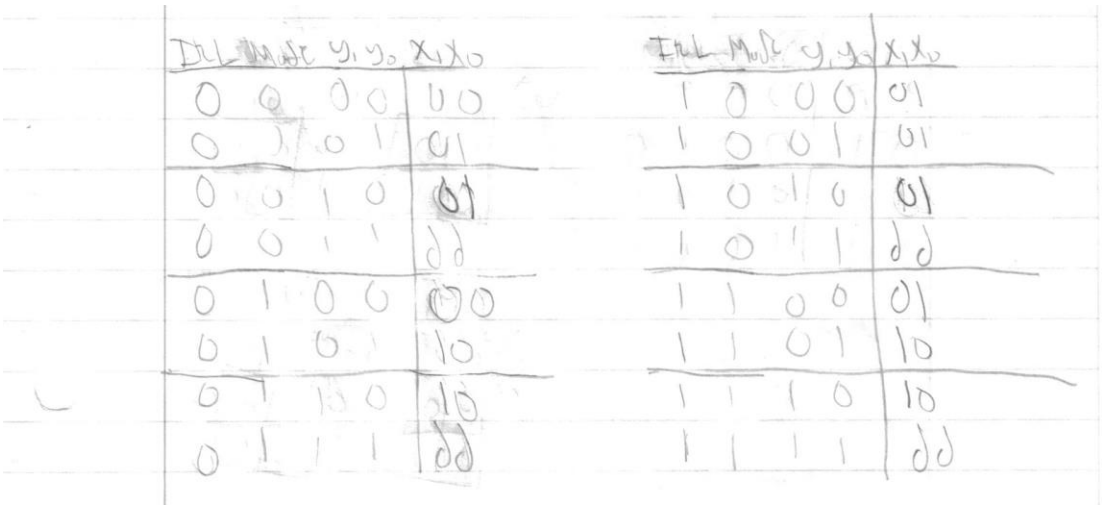
This outputs a 2-bit number which is interpreted as the state TGCo is to be in. This output corresponds directly to what is outputted by the Input_Logic_TGC.
The input and output logic of this module were derived from the following state diagram:



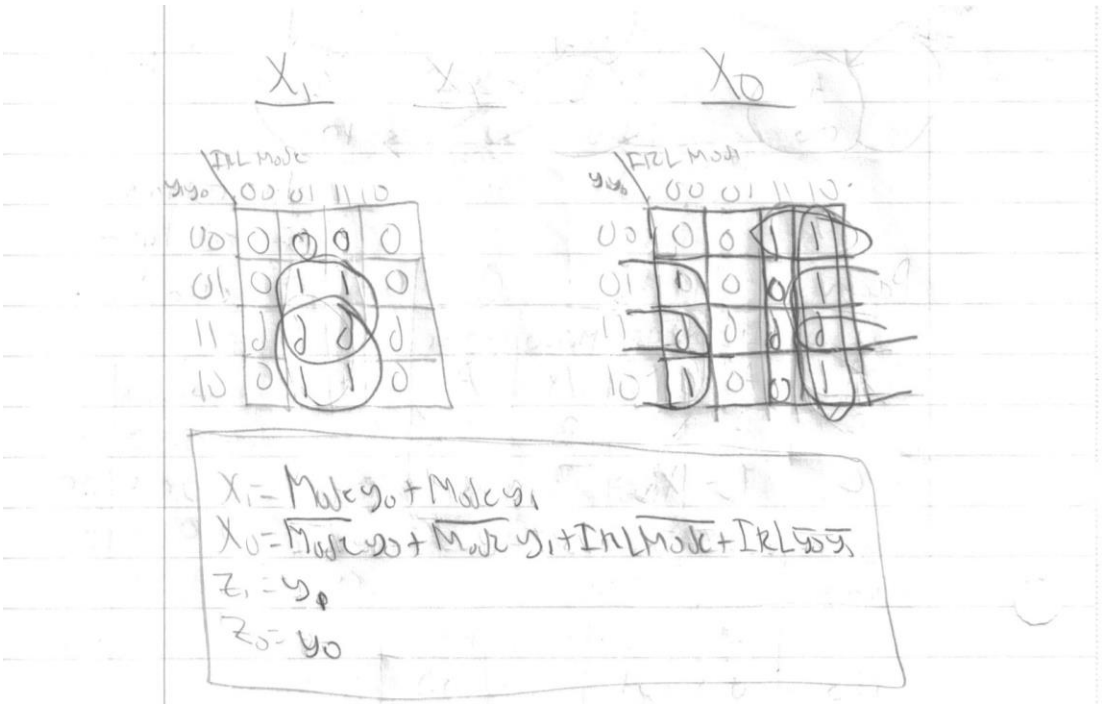
This diagram was then put into a state table



which was then put into a truth table



To derive the next state logic, K-maps were used for each bit



I then had full equations for each next state bit, which was written in Verilog and compiled as the block Input_Logic_TGC. The outputs were the same as the current state, so K-maps were not needed to derive equations for each bit. This is outputted as the 2-bit bus TGC_Chooseth.

Module MSM:

```
module MSM(tgc_out, control_switches, clock, control_button, irl_load, irl_selected, TLCF_Select, TLCF_Control, TLCF_Enable, TLCF_Clock,
           RF_Enable, RF_Select, RF_Clock, RF_In, ACL_Clock, IRL_Clock, IRL_Select, IRL_Capacity);

   input clock, control_button;
   input [1:0]tgc_out;
   input [3:0] control_switches, irl_load, irl_selected;
   output TLCF_Clock, TLCF_Enable, TLCF_Control, ACL_Clock, RF_Enable, RF_Clock, IRL_Clock, IRL_Select;
   output [3:0]TLCF_Select, RF_Select, RF_In, IRL_Capacity;

   reg TLCF_Clock, TLCF_Enable, TLCF_Control, ACL_Clock, RF_Enable, RF_Clock, IRL_Clock, IRL_Select;
   reg [3:0]TLCF_Select, RF_Select, RF_In, IRL_Capacity;

   always@ (tgc_out or control_switches or clock or control_button or irl_load or irl_selected)
   begin

      case(tgc_out)

         //Initial load capacity to register file
         2'b 00:
            begin

               IRL_Capacity = control_switches;
               IRL_Select = control_button;
               IRL_Clock = clock;
               TLCF_Clock = 0;
               TLCF_Control = 0;
               TLCF_Select = 4'b 0000;
               TLCF_Enable = 0;
               RF_Enable = 1;
               RF_Clock = clock;
               RF_Select = irl_selected;
               RF_In = irl_load;
               ACL_Clock = 0;
            end
         //Mode A (All lanes have red light)
         2'b 01:
            begin

               IRL_Capacity = 4'b 0000;
               IRL_Select = 0;
               IRL_Clock = 0;
               TLCF_Clock = clock;
               TLCF_Control = 0; //Add cars
               TLCF_Select = control_switches;
               TLCF_Enable = 1;
               RF_Enable = 0;
               RF_Clock = 0;
               RF_Select = control_switches;
               RF_In = 4'b 0000;
               ACL_Clock = 0;
            end
         //Mode B (1 lane has green light)
         2'b 10:
            begin

               IRL_Capacity = 4'b 0000;
               IRL_Select = 0;
               IRL_Clock = 0;
               TLCF_Clock = clock;
               TLCF_Control = 1; //Green light
               TLCF_Select = control_switches;
               TLCF_Enable = 1;
               RF_Enable = 0;
               RF_Clock = 0;
               RF_Select = control_switches;
               RF_In = 4'b 0000;
               ACL_Clock = clock;
            end
      endcase
   end
endmodule
```

Module Mode_Select_Mux or MSM, as it is named in the project files and .bsf's, takes in inputs and decides, based on the state chosen by the module TGC_Machine, where those inputs go to the rest of the module TGCo.
Here is a list of those inputs and outputs:

Inputs:

clock

The board clock, as described in module TGCo.

control_button

The button on the board, as described in module TGCo.

tgc_out

The output from module TGC_Machine. This is the “select line” for this mux.

control_switches

The switches on the board, as described in module TGCo.

irl_load

The data from module IRL_Machine to be loaded in to module 4_4bit_Register_File.

irl_selected

The register to load data into, provided by IRL_Machine.

Outputs:

TLCF_Clock

The clock input that goes to module TLCF.

TLCF_Enable

Enables the selection of a lane. Goes to module TLCF.

TLCF_Control

If the project machine is in Mode B, this is set to 1 so a lane can have a green light.

TLCF_Select

Inputted to TLCF to select a lane to add cars to.

ACL_Clock

The clock input that goes to module ACL_Machine_Mk2

RF_Enable

Enables the module 4_4bit_Register_File to load data.

RF_Clock

The clock input that goes to the register file.

RF_Select

Goes to the register file to select which register to read or write to.

RF_In

The data to write to a register in the register file.

IRL_Clock

The clock input that goes to the module IRL_Machine

IRL_Select

The input into IRL_Machine that, when 1, selects the data from the input control_switches as the capacity for a lane.

IRL_Capacity

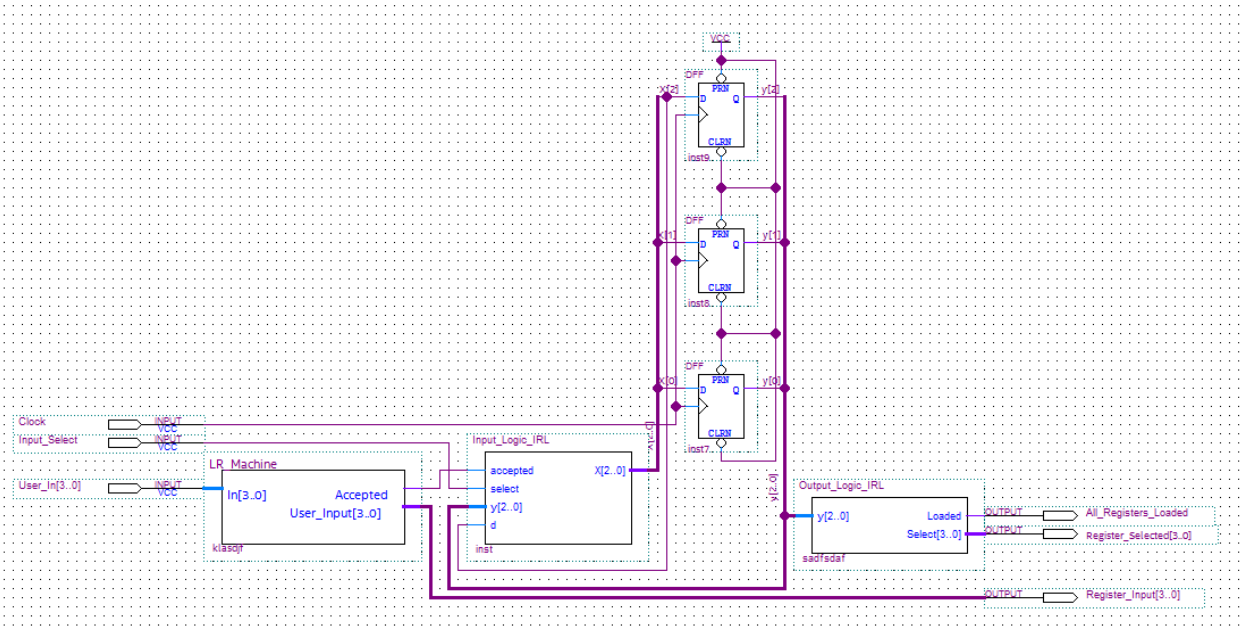
The desired capacity for a lane, provided by the input_control switches.

If the machine is in the initial capacity loading state, TGC_Machine is outputting a 00, then the machine is in the initial capacity loading stage. Because of this, IRL_Capacity is set to what is on control_switches, IRL_Select is set to what control_button is, TLCF_Clock is set to 0 to ensure that no lane counter can be selected, TLCF_Control is set to 0 to ensure that no lane has a green light, TLCF_Select is set to the 4 bit number 0000 to ensure that no lane counter is selected, RF_Enable is set to 1 because in this stage loading data into a register is desired, RF_Clock is set to the clock input, RF_Select is set to what is selected by the IRL_Machine to load data into a specific register, RF_In is ultimately set to what the control_switches are, and ACL_Clock is set to 0 to ensure that no lane is selected to have a green light.

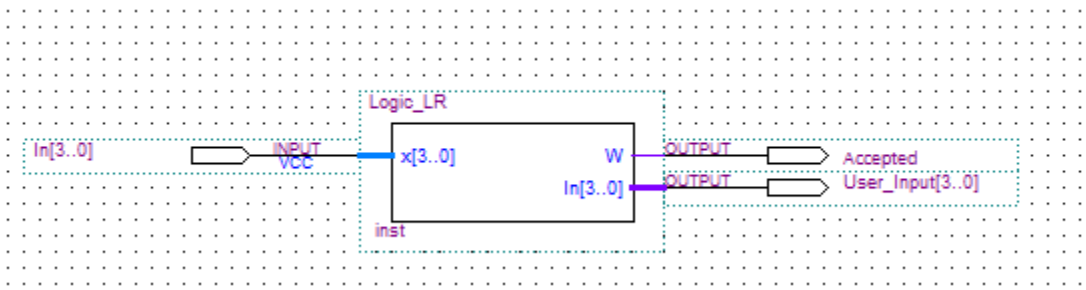
If the machine is in Mode A, TGC_Machine is outputting a 01, then every lane has a red light no register loading is being done. Because of this, IRL_Capacity is set to all 0s, IRL_Select is set to 0 to ensure TGC_Machine never goes back to the initial capacity loading state, IRL_Clock is set to 0, TLCF_Clock is set to the input clock because a lane is to be selected, TLCF_Select is set to what the input control_switches is to select a lane, TLCF_Enable is set to 1, RF_Enable is set to 0 because no loading is being done to the register file, RF_Clock is set to 0 for that same reason, RF_Select is set to what the input control_switches is to ensure that correct capacity for the selected lane is being read from the register file, RF_In is set to all 0s, and ACL_Clock is set to 0 for the same reason as in the last state.

If the machine is n Mode B, TGC_Machine is outputting a 10, then 1 lane has a green light. Because of this all of the IRL and RF outputs stay the same, TLCF_Clock is the same, TLCF_Control is set to 1 because one lane is going to have a green light, TLCF_Select says on the control_switches, TLCF_Enable stays 1, and ACL_Clock is set to 1 to start counting clock cycles and choose which lane has a green light.

Module IRL_Machine:



Module Initial_Register_Load or IRL_Machine as it is named in the project files and .bsf's is used for, as the name suggests, loading the register file with capacities for the lanes during the initial capacity loading stage of the project machine, and then outputs a 1 once it is completed with its task. Loading starts with register 0 and ends with register 3. Per the project requirements, the capacity cannot be 0 and this is checked using the module Load_Register or LR_Machine shown here:



This module takes in a 4 bit input and then either outputs a 1 if that input is anything other than 0, or a 0 if that input is 0. It also outputs a 4-bit number that is the inputted 4-bit number, just modified with either a 1 or a 0. The Verilog code for this module is shown here:

```
module Logic_LR(x, W, In);

    input [3:0]x;
    output W;
    output [3:0]In;

    assign W = (x[3]) | (x[2]) | (x[1]) | (x[0]);

    assign In[3] = W & x[3];
    assign In[2] = W & x[2];
    assign In[1] = W & x[1];
    assign In[0] = W & x[0];

endmodule
```

This modified 4 bit input and “validness” is then inputted into the block Input_Logic_IRL:

```
module Input_Logic_IRL (accepted, select, y, d, X);

    input accepted, select, d;
    input [2:0]y;
    output [2:0]X;

    reg [2:0]X;

    always @(d or accepted or select or y)
    begin

        if (d == 0)
            begin

                if (select == 1)
                    begin

                        if (accepted == 1)
                            begin

                                case(y)

                                    3'b 000: X = 3'b 001;
                                    3'b 001: X = 3'b 010;
                                    3'b 010: X = 3'b 011;
                                    3'b 011: X = 3'b 100;
                                    3'b 100: X = 3'b 100;
                                    3'b 101: X = 3'b 100;
                                    3'b 110: X = 3'b 100;
                                    3'b 111: X = 3'b 100;

                                endcase
                            end
                        else
                            begin

                                X = y;
                            end
                    end
                else
                    begin

                        X = y;
                    end
            end
        else
            begin

                X = y;
            end
    end
endmodule
```

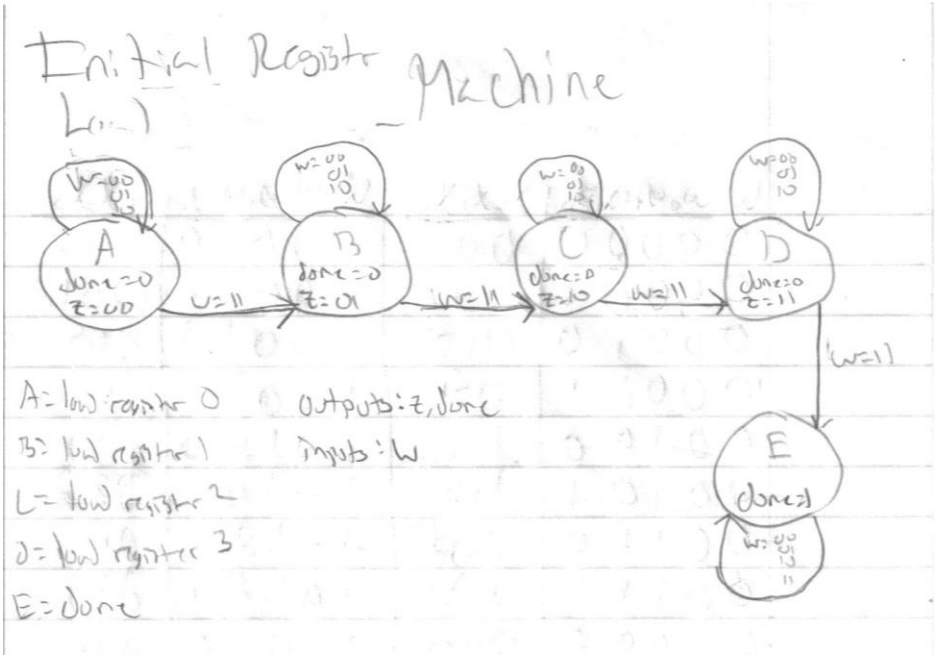
This block outputs a 2-bit number that corresponds to the next state that the module will be in, similar to what was done in TGC_Machine. The state corresponds to a register to load the validated capacity 000 meaning load into register 0, 001 meaning load into register 1, 010 meaning load into register 2, and finally 011 meaning load into register 3. There is 1 more bit though, one may notice. This last bit, or most significant I should say, represents whether or not the module is done loading or not, 0 if not done, 1 if done. Once this 1 is outputted it will forever stay 1 to ensure that the user cannot go back into loading capacities per the project requirements.

The module then, after updating the D Flip-Flops with the new state outputs a 3-bit number Select which corresponds to the register to load data into, 0001 meaning register 0, 0010 meaning register 1, 0100 meaning register 2, and 1000 meaning register 3. This logic is shown here:

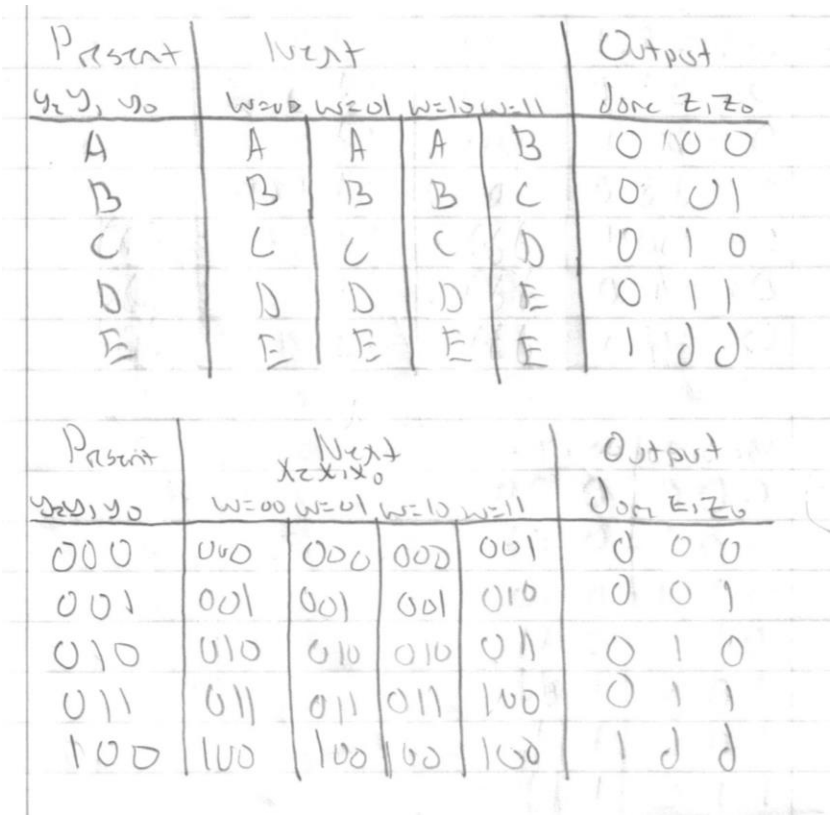
```
module Output_Logic_IRL (y, Loaded, Select);

   input [2:0]y;
   output Loaded;
   output [3:0]Select;

   reg Loaded;
   reg [3:0]Select;

   always @(y)
   begin
      case(y)

         3'b 000:
            begin

               Loaded = 0;
               Select = 4'b 0001;
            end
         3'b 001:
            begin

               Loaded = 0;
               Select = 4'b 0010;
            end
         3'b 010:
            begin

               Loaded = 0;
               Select = 4'b 0100;
            end
         3'b 011:
            begin

               Loaded = 0;
               Select = 4'b 1000;
            end
         3'b 100:
            begin

               Loaded = 1;
               Select = 4'b 0000;
            end
         3'b 101:
            begin

               Loaded = 1;
               Select = 4'b 0000;
            end
         3'b 110:
            begin

               Loaded = 1;
               Select = 4'b 0000;
            end
         3'b 111:
            begin

               Loaded = 1;
               Select = 4'b 0000;
            end
      endcase
   end
endmodule
```

Once the most significant bit of the input is a 1, the variable Loaded is set to 1, signifying that the IRL_Machine module is done.

This input and output logic was derived from the following state diagram:



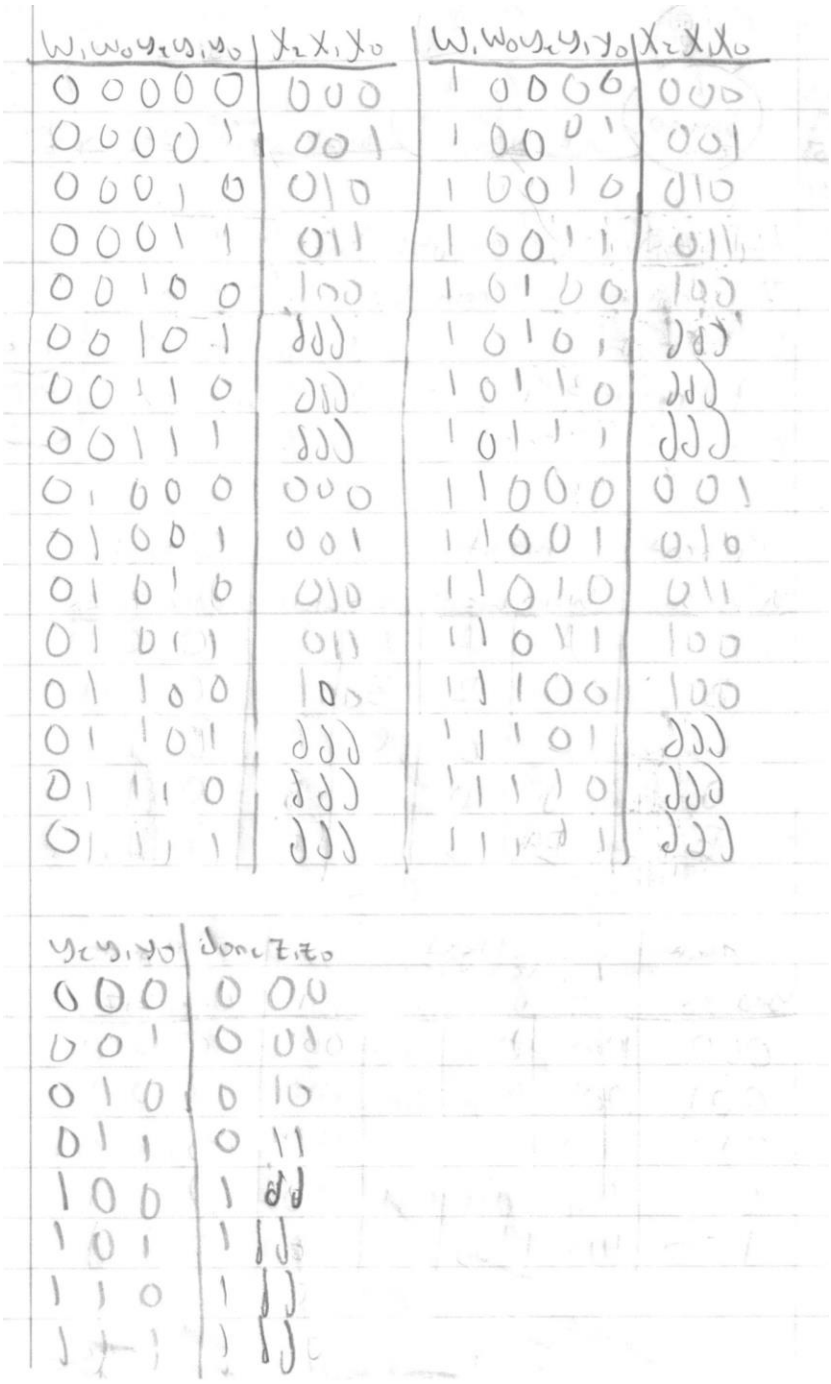
This was then put into a state table:



Present	Next				Output
$y_2 y_1 y_0$	w=00	w=01	w=10	w=11	done $z_1 z_0$
A	A	A	A	B	0 0 0
B	B	B	B	C	0 0 1
C	C	C	C	D	0 1 0
D	D	D	D	E	0 1 1
E	E	E	E	E	1 d d

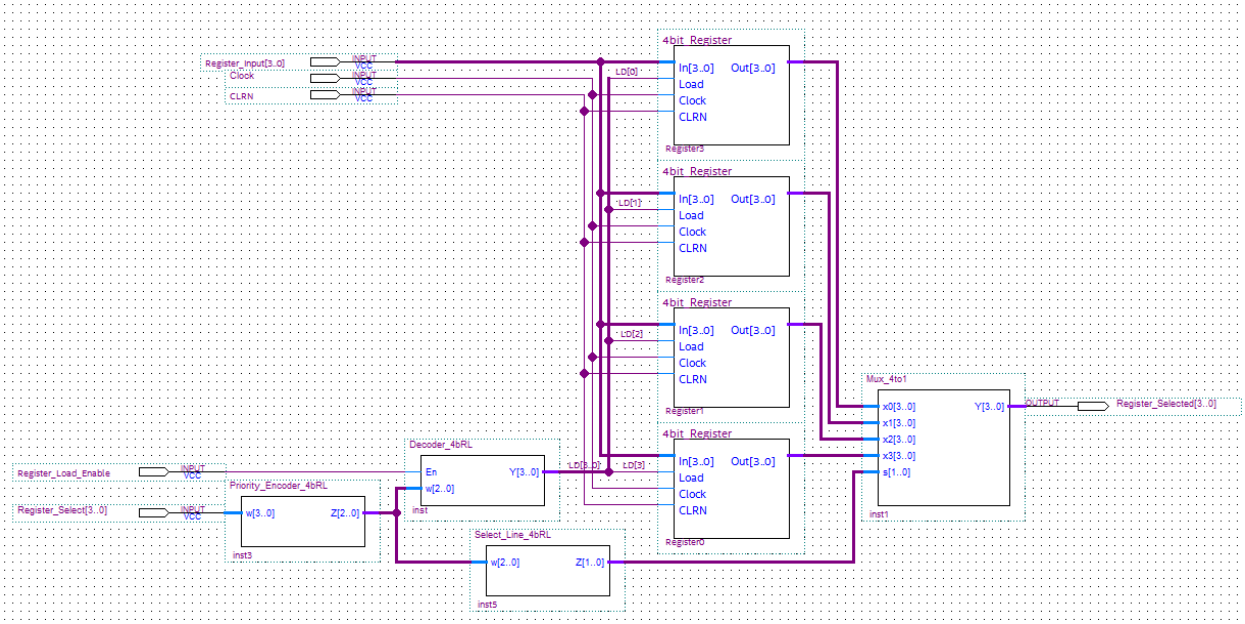
Present	Next $x_2 x_1 x_0$				Output
$y_2 y_1 y_0$	w=00	w=01	w=10	w=11	done $z_1 z_0$
000	000	000	000	001	0 0 0
001	001	001	001	010	0 0 1
010	010	010	010	011	0 1 0
011	011	011	011	100	0 1 1
100	100	100	00	100	1 d d

Which was translated into a truth table:



Because both the input and output logic were written in Verilog, and the module outputs correspond directly to the current state, no K-maps were used.

Module 4_4bit_Register_File



This is the register file for this project, as the name suggests. It holds the specific capacities for a lane. As stated in the IRL_Machine description, it is loaded during the initial capacity loading stage of the project machine, per the requirements and are read from when a lane is chosen. The lane selection and a register selection is one-to-one. For example, if cars are being added to lane 3, then register 3 is read from. Inputs are given by module MSM. The input Register_Select are put into the priority encoder Priority_Encoder_4bRL to endure no two registers are loaded to at one shown here:

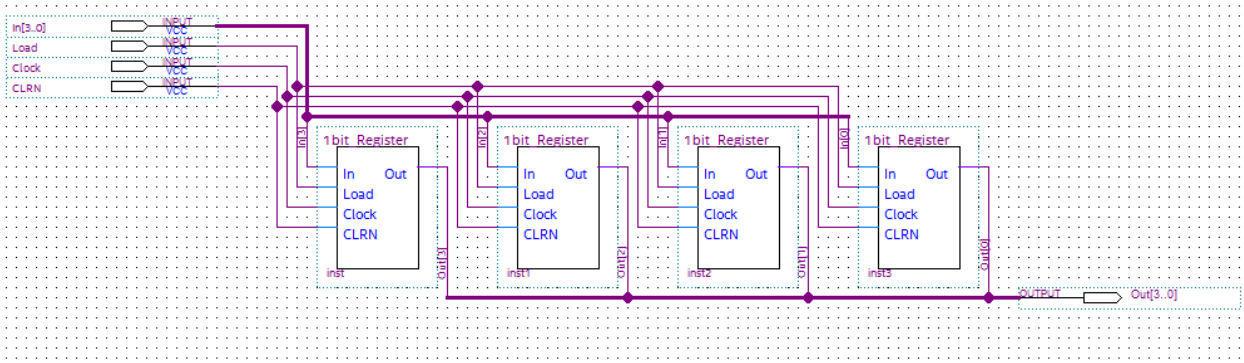
```
module Priority_Encoder_4bRL (w, Z);

   input [3:0]w;
   output [2:0]Z;

   reg [2:0]Z;

   always @(w)
   begin

      if (w[3] == 1)
         begin

            Z = 3'b 100;
         end
      else if (w[2] == 1)
         begin

            Z = 3'b 011;
         end
      else if (w[1] == 1)
         begin

            Z = 3'b 010;
         end
      else if (w[0] == 1)
         begin

            Z = 3'b 001;
         end
      else
         begin

         Z = 3'b 000;
         end
   end
endmodule
```

Priority is given in increasing register name: register 3 has priority over all other registers, register 2 has priority over registers 1 and 0, register 1 has priority over register 0 and register 0 has priority over no other registers. There is also the case where if the input line contains all 0s, all the switches are 0, then no register is to be chosen. This is taken advantage of in the module TCLF for user convenience.

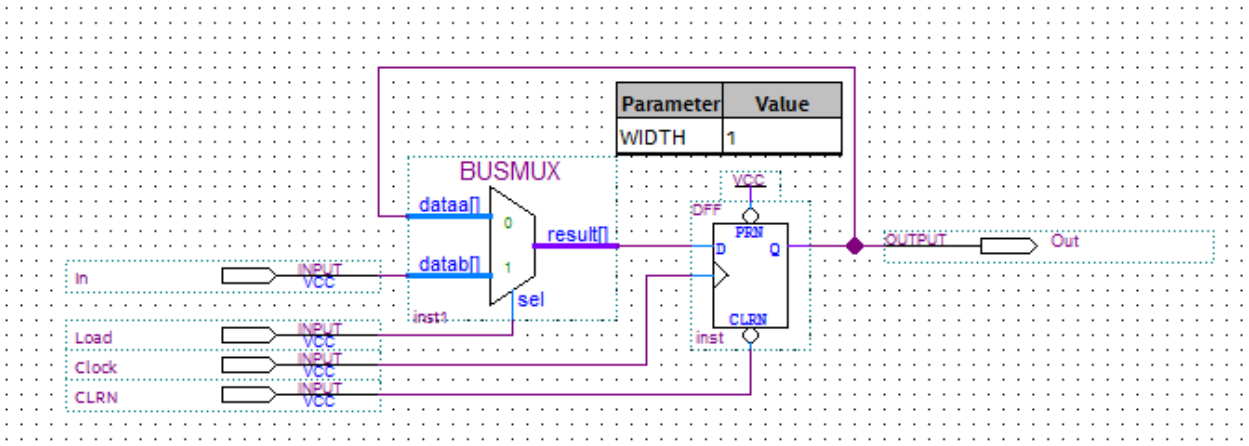
The output of the priority encoder is put into two different blocks. The first being a decoder shown here:

```
module Decoder_4bRL(En, w, Y);

   input En;
   input [2:0]w;
   output [3:0]Y;

   assign Y[0] = En & (~w[2] & ~w[1] & w[0]);
   assign Y[1] = En & (~w[2] & w[1] & ~w[0]);
   assign Y[2] = En & (~w[2] & w[1] & w[0]);
   assign Y[3] = En & (w[2] & ~w[1] & ~w[0]);

endmodule
```

This takes advantage of the one hot encoded nature of a decoder and selects which register to enable, loading whatever data is supplied through the input Register_Input into that register on the next clock cycle. The register being loaded is shown here:



This is a parallel load register similar to ones found in the class presentation slides. Each of these 4 bit registers are comprised of 4 1 bit registers shown here:



Back to the 4_4bit_Register_File, the second block the priority encoded line goes to is Select_Line_4bRL:

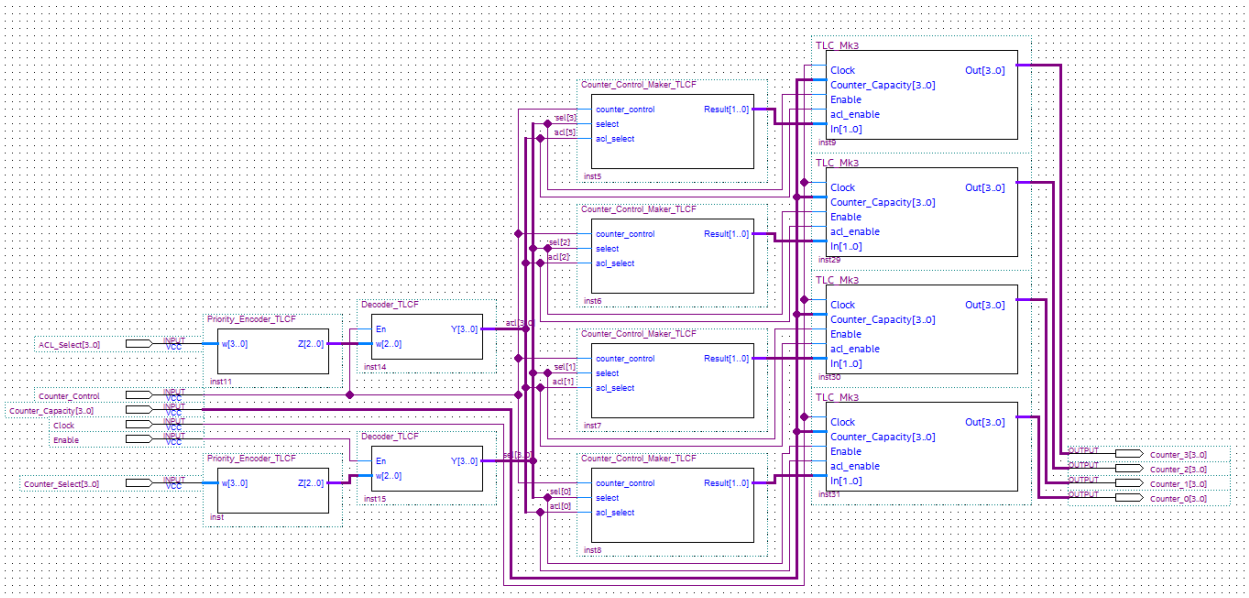
```
module Select_Line_4bRL(w, Z);

   input [2:0]w;
   output [1:0]Z;

   assign Z[1] = w[2] | (w[1] & w[0]);
   assign Z[0] = w[2] | (w[1] & ~w[0]);

endmodule
```

This block takes in the priority encoded output and turns that into a 2 bit select line for the 4_4bit_Register_File 4-1 mux shown here:

```
module Mux_4to1(x0, x1, x2, x3, s, Y);

    input [1:0]s;
    input [3:0]x0, x1, x2, x3;
    output [3:0]Y;

    reg [3:0]Y;

    always @(s or x0 or x1 or x2 or x3)
    begin
        case (s)

            2'b 00: Y = x0;
            2'b 01: Y = x1;
            2'b 10: Y = x2;
            2'b 11: Y = x3;

        endcase
    end
endmodule
```

This takes in all of the registers contents, and then outputs 1 of them based on a select line. This ensures that whatever register is being written to, if in the loading stage, the contents of that register are being read.

Module TLCF and TLC_Mk3



This module acts similar to module 4_4bit_Register_File, except with counters instead of registers. This takes in numerous inputs:

ACL_Select:

This is a 4 bit input which is determined by the module ACL_Mk2_Machine, described later. This goes through a priority encoder, identical in spec to the one used in the module 4_4bit_Register_File and then through a decoder, again identical to the one used in the register file module, to select a counter to decrement, or have a green light.

Counter_Control:

This input is 1 if the project machine is in Mode B and 0 if not. This is the enable line for the decoder that selects a line to have a green light.

Counter_Capacity:

This is a 4 bit input that comes from the module 4_4bit_Register_File. This is the capacity that a counter, when selected cannot increment over.

Clock:

This is the clock for the counters.

Enable:

This input enables this module to work and select a counter to me modified.

Counter_Select:

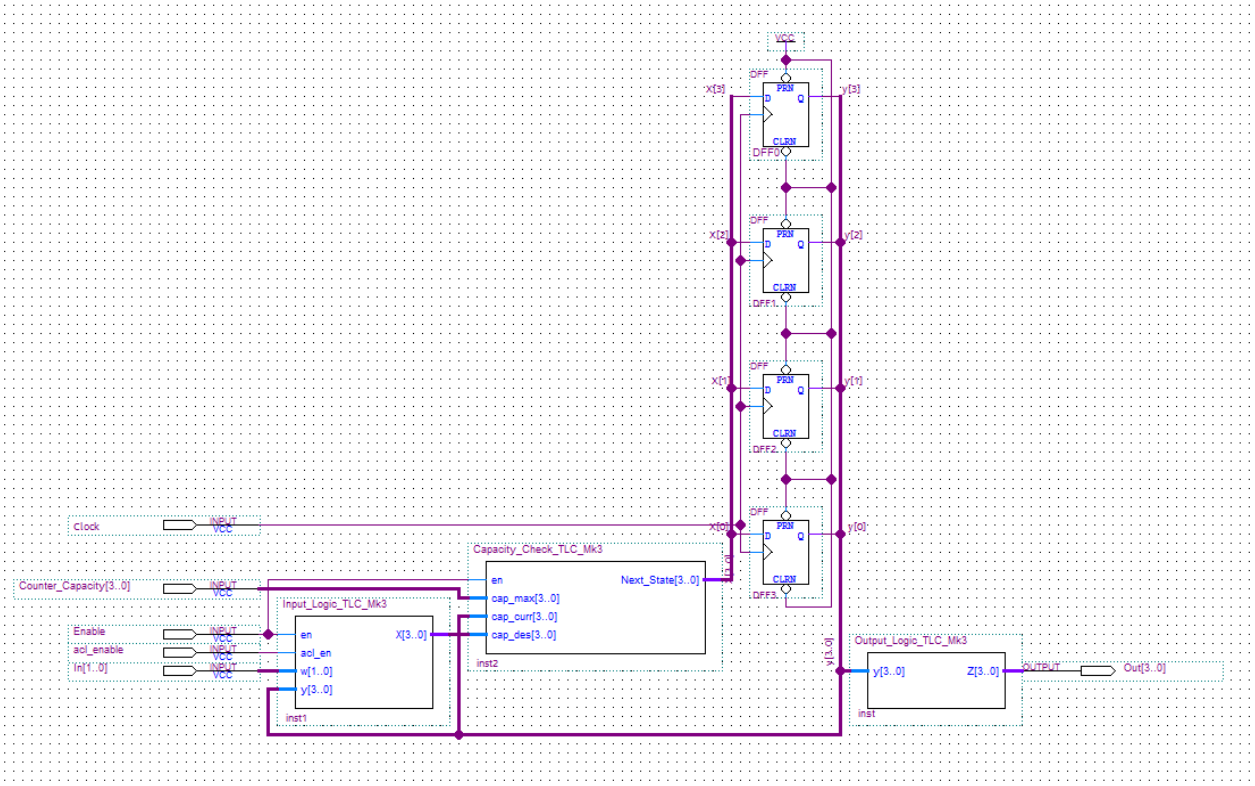
This is a 4 bit input that goes through an identical priority encoder and decoder used for the input ACL_Select. This selects a counter to be modified.

While in Mode A or Mode B, a counter is selected by the input Counter_Select to either increment in Mode A's case, or decrement or increment in Mode B's case. This selection, after choosing priority and decoding it to be one-hot encoded, is split into 4 different data lines and enters the block Counter_Control_Maker_TLCF shown here:

```
module Counter_Control_Maker_TLCF(counter_control, select, acl_select, Result);

   input counter_control, select, acl_select;
   output [1:0] Result;

   reg [1:0] Result;

   always @ (counter_control or select)
   begin

      if (counter_control == 1)
         begin

            if (acl_select == 1)
               begin

                  Result[1] = 1;
                  Result[0] = select;
               end
            else
               begin

                  Result[1] = 0;
                  Result[0] = select;
               end
         end
      else
         begin

            Result[1] = 0;
            Result[0] = select;
         end
   end
endmodule
```

This block, written in Verilog, outputs a 2-bit bus where the most significant bit is 1 if the counter is to decrement, the lane has a green light, and the least significant bit is 1 if the counter is to increment, the lane gets another car. If the counter is supposed to increment and decrement at the same time, the counter stays at the same value it is at per the specifications, this block outputs a 0 and whatever the select line is. This select line comes from the higher level input Counter_Select, which determines if the counter should increment or not.

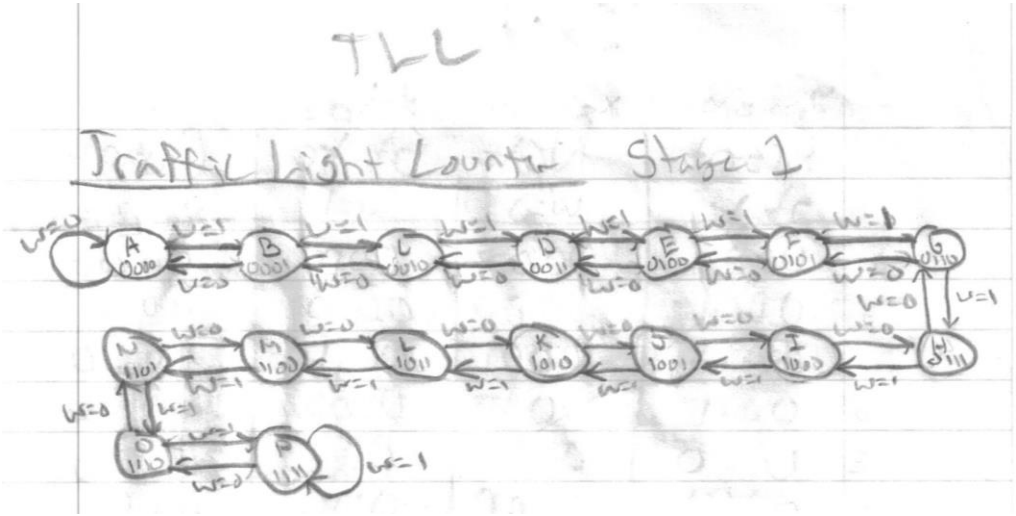
The output of the block Counter_Control_Maker goes into the actual counter itself along with a multitude of other variables. The specialized counter module TCL_Mk3 is shown here:



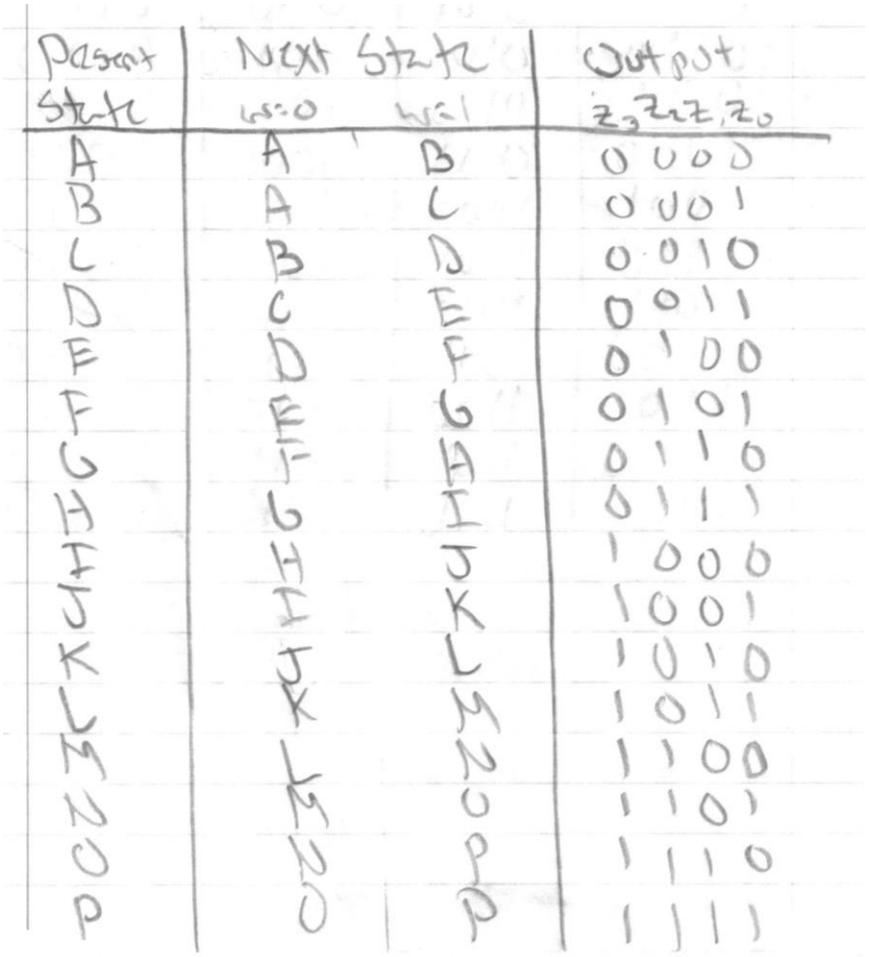
Traffic_Light_Counter_Mk3 or TLC_Mk3, as it is named in the project files and .bsf's, is the specialized counter for this project. It takes in a clock input, a 4 bit input Counter_Capacity which is the limit to what the counter can increment to, an enable line Enable which enables this counter to increment or decrement or not, an input named acl_enable which determines if this counter is selected to decrement or not and a 2 bit input In which controls if the counter is allowed to increment or decrement or not. The inputs Enable, acl_enable, and In are passed through the block Input_Logic_TLC_Mk3 shown here:

```
module Input_Logic_TLC_Mk3 (en, acl_en, w, y, X);

   input en, acl_en;
   input [1:0]w;
   input [3:0]y;
   output [3:0]X;

   reg [3:0]X;

   always @(en or w or y)
   begin

      if (en == 1)
         begin

            case (w)

               //Count up (Add car)
               2'b 01:
                  begin

                     case (y)

                        4'b 0000: X = 4'b 0001; //Go to State B
                        4'b 0001: X = 4'b 0010; //Go to State C
                        4'b 0010: X = 4'b 0011; //Go to State D
                        4'b 0011: X = 4'b 0100; //Go to State E
                        4'b 0100: X = 4'b 0101; //Go to State F
                        4'b 0101: X = 4'b 0110; //Go to State G
                        4'b 0110: X = 4'b 0111; //Go to State H
                        4'b 0111: X = 4'b 1000; //Go to State I
                        4'b 1000: X = 4'b 1001; //Go to State J
                        4'b 1001: X = 4'b 1010; //Go to State K
                        4'b 1010: X = 4'b 1011; //Go to State L
                        4'b 1011: X = 4'b 1100; //Go to State M
                        4'b 1100: X = 4'b 1101; //Go to State N
                        4'b 1101: X = 4'b 1110; //Go to State O
                        4'b 1110: X = 4'b 1111; //Go to State P
                        4'b 1111: X = 4'b 1111; //Stay in State P
                     endcase
                  end
               //Count down (Green light)
               2'b 10:
                  begin

                     case(y)

                        4'b 0000: X = 4'b 0000; //Go to State A
                        4'b 0001: X = 4'b 0000; //Go to State A
                        4'b 0010: X = 4'b 0001; //Go to State B
                        4'b 0011: X = 4'b 0010; //Go to State C
                        4'b 0100: X = 4'b 0011; //Go to State D
                        4'b 0101: X = 4'b 0100; //Go to State E
                        4'b 0110: X = 4'b 0101; //Go to State F
                        4'b 0111: X = 4'b 0110; //Go to State G
                        4'b 1000: X = 4'b 0111; //Go to State H
                        4'b 1001: X = 4'b 1000; //Go to State I
                        4'b 1010: X = 4'b 1001; //Go to State J
                        4'b 1011: X = 4'b 1010; //Go to State K
                        4'b 1100: X = 4'b 1011; //Go to State L
                        4'b 1101: X = 4'b 1100; //Go to State M
                        4'b 1110: X = 4'b 1101; //Go to State N
                        4'b 1111: X = 4'b 1110; //Go to State O
                     endcase
                  end

                     default: X = y;
            endcase
         end
      else if (acl_en == 1)
         begin

            case (w)
```

This determines if the counter is supposed to change states or not. Using Verilog, if the counter is enabled, input en is 1, the counter does whatever is on the control input w. If w is a 01, then the counter increments states, if w is a 10, the counter decrements states, and if w is neither, then the counter stays in the same state. If the counter is not enabled, but is supposed to decrement states, based on the input acl_en, then the counter decrements. In other words, input en enables the counter to increment, and acl_en enables the counter to decrement. If neither of these inputs are 1, the counter stays in the same state. All of this was derived from the following state diagram, and then modified to as needed:



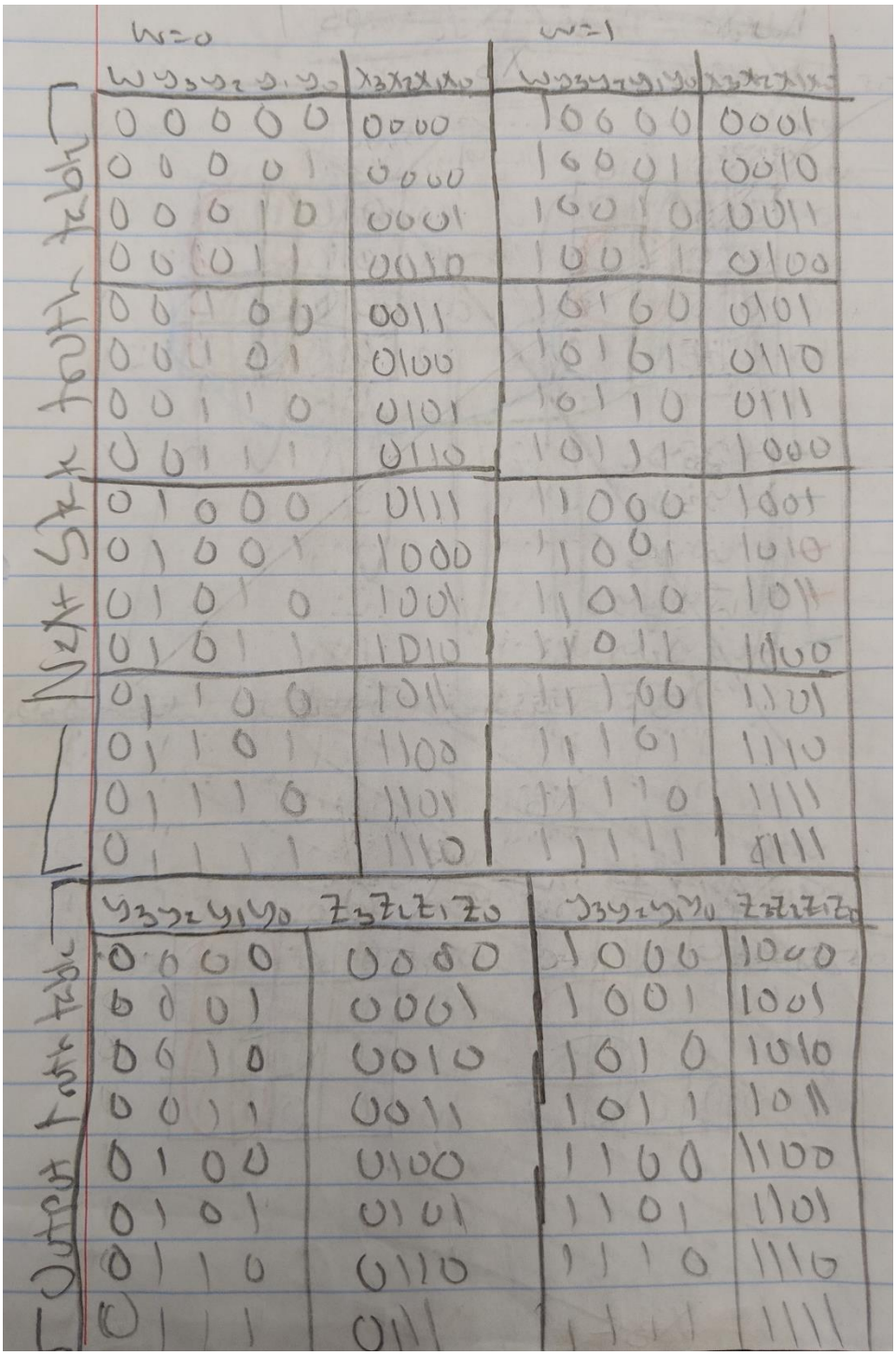
This was then put into a state table:



Present State	Next State w=0	Next State w=1	Output $z_3 z_2 z_1 z_0$
A	A	B	0 0 0 0
B	A	C	0 0 0 1
C	B	D	0 0 1 0
D	C	E	0 0 1 1
E	D	F	0 1 0 0
F	E	G	0 1 0 1
G	F	H	0 1 1 0
H	G	I	0 1 1 1
I	H	J	1 0 0 0
J	I	K	1 0 0 1
K	J	L	1 0 1 0
L	K	M	1 0 1 1
M	L	N	1 1 0 0
N	M	O	1 1 0 1
O	N	P	1 1 1 0
P	O	P	1 1 1 1

	Past State	Next State w=0	Next State w=1	Output
A	0000	0000	0001	0000
B	0001	0000	0010	0001
C	0010	0001	0011	0010
D	0011	0010	0100	0011
E	0100	0011	0101	0100
F	0101	0100	0110	0101
G	0110	0101	0111	0110
H	0111	0110	1000	0111
I	1000	0111	1001	1000
J	1001	1000	1010	1001
K	1010	1001	1011	1010
L	1011	1010	1100	1011
M	1100	1011	1101	1100
N	1101	1100	1110	1101
O	1110	1101	1111	1110
P	1111	1110	1111	1111

and then put into a truth table:



Because I used truth table coding in Verilog, I used no K-maps to derive equations for variables.

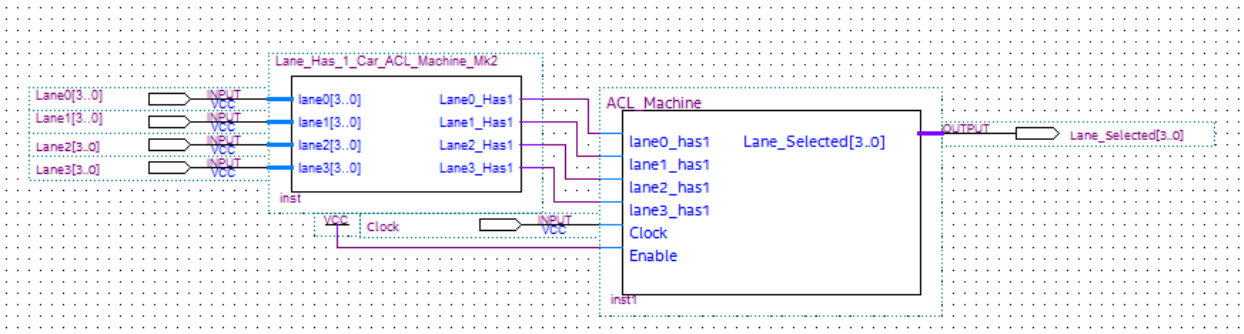
The output of the module Input_Logic_TLC_Mk3 is then passed through another block Capacicity_Check_TLC_Mk3 shown here

```
module Capacity_Check_TLC_Mk3(en, cap_max, cap_curr, cap_des, Next_State);

    input en;
    input [3:0]cap_max, cap_curr, cap_des;
    output [3:0]Next_State;

    reg [3:0]Next_State;

    always @(en or cap_max or cap_curr or cap_des)
    begin

        if (en == 1)
            begin

                if (cap_curr > cap_max)
                    begin

                        Next_State = cap_max;
                    end
            end

        if (cap_des <= cap_max)
            begin

                Next_State = cap_des;
            end
        else
            begin

                Next_State = cap_curr;
            end
    end
endmodule
```

This, as the name suggests, checks whether or not the state outputted by Input_Logic_TLC_Mk3 is over the capacity the counter is bound to. If the next state is less than or equal to the capacity, then the counter increments to that state. If it is greater than the capacity, then the counter stays in the current state.

This is then fed to D Flip-Flops which ultimately go to the block Output_Logic_TLC_Mk3 which outputs a 4-bit bus that is one to one to the counter's state.

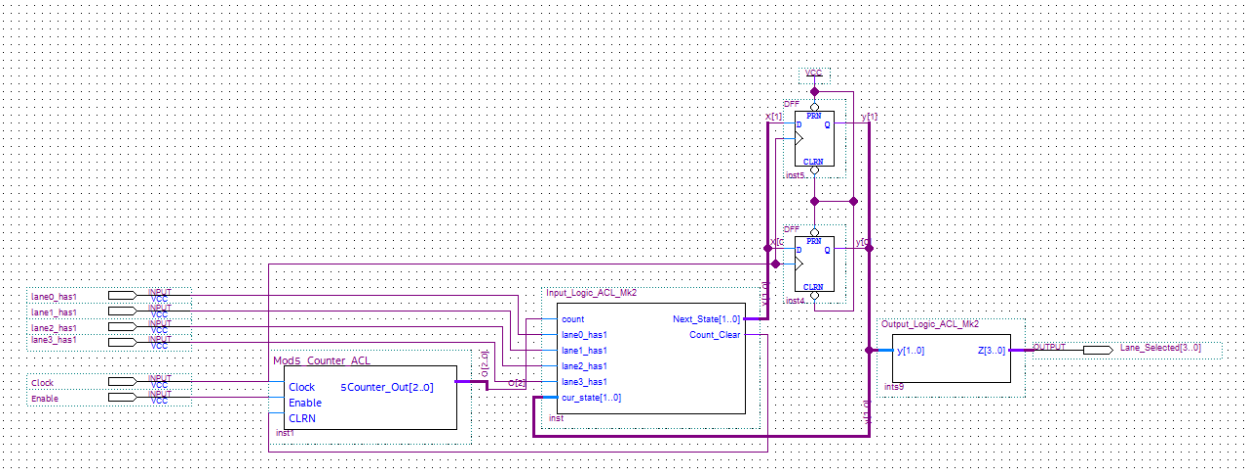
Module ACL_Machine_Mk2



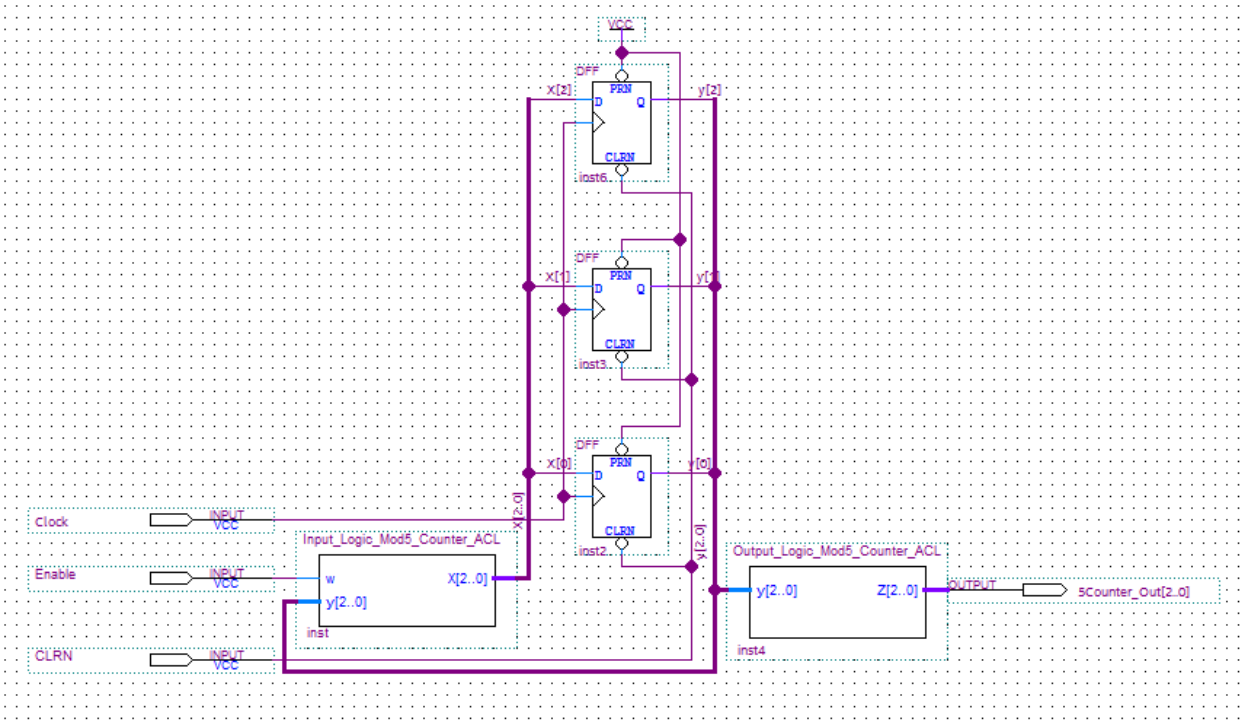
Module Auto_Change_Lanes_Mk2 or ACL_Mk2, as it is named in the project files and .bsf's is the state machine that selects a counter to decrement or not. It first determines if a lane has a car in it using the block Lane_Has_1_Car_ACL_Machine_Mk2 shown here:

```
module Lane_Has_1_Car_ACL_Machine_Mk2(lane0, lane1, lane2, lane3, Lane0_Has1, Lane1_Has1, Lane2_Has1, Lane3_Has1);

   input[3:0] lane0, lane1, lane2, lane3;
   output Lane0_Has1, Lane1_Has1, Lane2_Has1, Lane3_Has1;

   assign Lane0_Has1 = (lane0[3] || lane0[2] || lane0[1] || lane0[0]);
   assign Lane1_Has1 = (lane1[3] || lane1[2] || lane1[1] || lane1[0]);
   assign Lane2_Has1 = (lane2[3] || lane2[2] || lane2[1] || lane2[0]);
   assign Lane3_Has1 = (lane3[3] || lane3[2] || lane3[1] || lane3[0]);

endmodule
```

It takes inputs that are the current counter capacities from module TLCF and then outputs a 1 corresponding to each lane if that lane has at least 1 car in it. This is then outputted to the machine ACL_Machine:



This finite state machine takes in the outputs from the block Lane_Has_1_Car_ACL_Machine_Mk2, and then determines which lane is supposed to have a green light. Per the project specifications, the green light moves to a different lane that has cars in it after 5 clock cycles. The clock cycles are counted by the module Mod5_Counter_ACL shown here:



This counts to 5, incrementing every clock cycle, and then resetting only if told to, or if it reaches 5. This is similar to counters derived in class presentation slides. The next state is determined by the block Input_Logic_Mod5_Counter_ACL shown here:

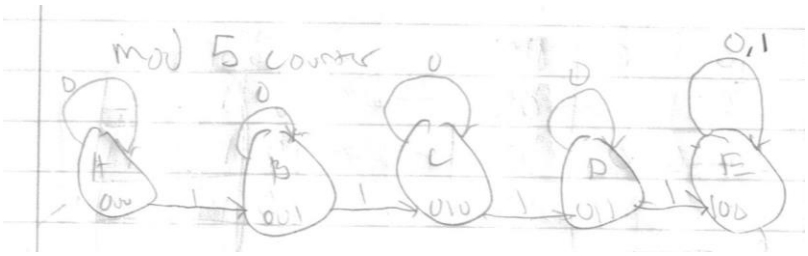
```
module Input_Logic_Mod5_Counter_ACL (w, y, X);

    input w;
    input[2:0]y;
    output [2:0]X;

    assign X[2] = (~w & y[2]) | (w & y[1] & y[0]);
    assign X[1] = (~w & y[1]) | (y[1] & ~y[0]) | (w & ~y[2] & ~y[1] & y[0]);
    assign X[0] = (~w & y[0]) | (w & ~y[2] & ~y[0]);

endmodule
```

This new state is then fed into d Flip-Flops and ultimately outputted by the block Output_Logic_Mod5_Counter_ACL:

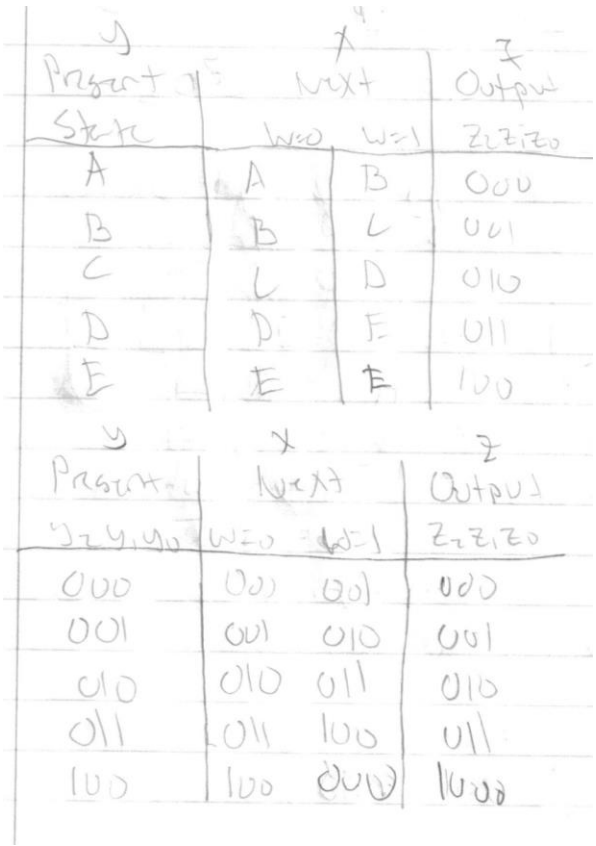
```
module Output_Logic_Mod5_Counter_ACL (y, Z);

    input [2:0]y;
    output [2:0]Z;

    assign Z[2] = y[2];
    assign Z[1] = y[1];
    assign Z[0] = y[0];

endmodule
```

The outputs are one to one with the current state.

The inputs and outputs were derived from the following state diagram



Which was then translated into this state table:



and then put into this truth table:

The output of the module Mod5_Counter_ACL is then inputted to the block Input_Logic_ACL_Mk3 in the module ACL_Machine shown here:

```
module Input_Logic_ACL_Mk2 (count, lane0_has1, lane1_has1, lane2_has1, lane3_has1, cur_state, Next_State, Count_Clear);

   input count, lane0_has1, lane1_has1, lane2_has1, lane3_has1;
   input [1:0]cur_state;
   output Count_Clear;
   output [1:0]Next_State;

   reg Count_Clear;
   reg [1:0]Next_State;

   always @(count or lane0_has1 or lane1_has1 or lane2_has1 or lane3_has1 or cur_state)
   begin

      //If the counter reaches 5
      if (count == 1)
         begin

            case(cur_state)

               //If in state 0
               2'b 00:
                  begin

                     if (lane1_has1 == 1)
                        begin
                           //Move to state 1
                           Next_State = 2'b 01;
                        end
                     else if (lane2_has1 == 1)
                        begin
                           //Move to state 2
                           Next_State = 2'b 10;
                        end
                     else if (lane3_has1 == 1)
                        begin
                           //Move to state 3
                           Next_State = 2'b 11;
                        end
                     else
                        begin
                           //Stay in state 0
                           Next_State = cur_state;
                           Count_Clear = 1;
                        end
                  end

               //If in state 1
               2'b 01:
                  begin

                     if (lane2_has1 == 1)
                        begin
                           //Move to state 2
                           Next_State = 2'b 10;
                        end
                     else if (lane3_has1 == 1)
                        begin
                           //Move to state 3
                           Next_State = 2'b 11;
                        end
                     else if (lane0_has1 == 1)
                        begin
                           //Move to state 0
                           Next_State = 2'b 00;
                        end
                     else
                        begin
                           //Stay in state 1
                           Next_State = cur_state;
                           Count_Clear = 1;
                        end
                  end

               //If in state 2
               2'b 10:
                  begin

                     if (lane3_has1 == 1)
                        begin
                           //Move to state 3
                           Next_State = 2'b 11;
                        end
                     else if (lane0_has1 == 1)
                        begin
                           //Move to state 0
                           Next_State = 2'b 00;
                        end
                     else if (lane1_has1 == 1)
                        begin
                           //Move to state 1
                           Next_State = 2'b 01;
                        end
                     else
                        begin
                           //Stay in state 2
                           Next_State = cur_state;
                           Count_Clear = 1;
                        end
                  end

               //If in state 3
               2'b 11:
                  begin

                     if (lane0_has1 == 1)
                        begin
                           //Move to state 0
                           Next_State = 2'b 00;
                        end
                     else if (lane1_has1 == 1)
                        begin
                           //Move to state 1
                           Next_State = 2'b 01;
                        end
                     else if (lane2_has1 == 1)
                        begin
                           //Move to state 2
                           Next_State = 2'b 10;
                        end
                     else
                        begin
                           //Stay in state 3
                           Next_State = cur_state;
                           Count_Clear = 1;
                        end
                  end
            endcase
         end
```

```
//If counter reaches 0 before 5 clock cycles
else
    begin

        case(cur_state)

            //If in state 0
            2'b 00:
                begin

                    if (lane0_has1 == 0)
                        begin

                            if (lane1_has1 == 1)
                                begin
                                    //Move to state 1
                                    Next_State = 2'b 01;
                                    Count_Clear = 0;
                                end
                            else if (lane2_has1 == 1)
                                begin
                                    //Move to state 2
                                    Next_State = 2'b 10;
                                    Count_Clear = 0;
                                end
                            else if (lane3_has1 == 1)
                                begin
                                    //Move to state 3
                                    Next_State = 2'b 11;
                                    Count_Clear = 0;
                                end
                            else
                                begin
                                    //Stay in state 0
                                    Next_State = cur_state;
                                    Count_Clear = 1;
                                end
                        end
                    else
                        begin
                            //Stay in state 0
                            Next_State = cur_state;
                            Count_Clear = 1;
                        end
                end

            //If in state 1
            2'b 01:
                begin

                    if (lane1_has1 == 0)
                        begin

                            if (lane2_has1 == 1)
                                begin
                                    //Move to state 2
                                    Next_State = 2'b 10;
                                    Count_Clear = 0;
                                end
                            else if (lane3_has1 == 1)
                                begin
                                    //Move to state 3
                                    Next_State = 2'b 11;
                                    Count_Clear = 0;
                                end
                            else if (lane0_has1 == 1)
                                begin
                                    //Move to state 0
                                    Next_State = 2'b 00;
                                    Count_Clear = 0;
                                end
                            else
                                begin
                                    //Stay in state 1
                                    Next_State = cur_state;
                                    Count_Clear = 1;
                                end
                        end
                    else
                        begin
                            //Stay in state 1
                            Next_State = cur_state;
                            Count_Clear = 1;
                        end
                end

            //If in state 2
            2'b 10:
                begin

                    if (lane2_has1 == 0)
                        begin

                            if (lane3_has1 == 1)
                                begin
                                    //Move to state 3
                                    Next_State = 2'b 11;
                                    Count_Clear = 0;
                                end
                            else if (lane0_has1 == 1)
                                begin
                                    //Move to state 0
                                    Next_State = 2'b 00;
                                    Count_Clear = 0;
                                end
                            else if (lane1_has1 == 1)
                                begin
                                    //Move to state 1
                                    Next_State = 2'b 01;
                                    Count_Clear = 0;
                                end
                            else
                                begin
                                    //Stay in state 2
                                    Next_State = cur_state;
                                    Count_Clear = 0;
                                end
                        end
                    else
                        begin
                            //Stay in state 2
                            Next_State = cur_state;
                            Count_Clear = 1;
                        end
                end
```

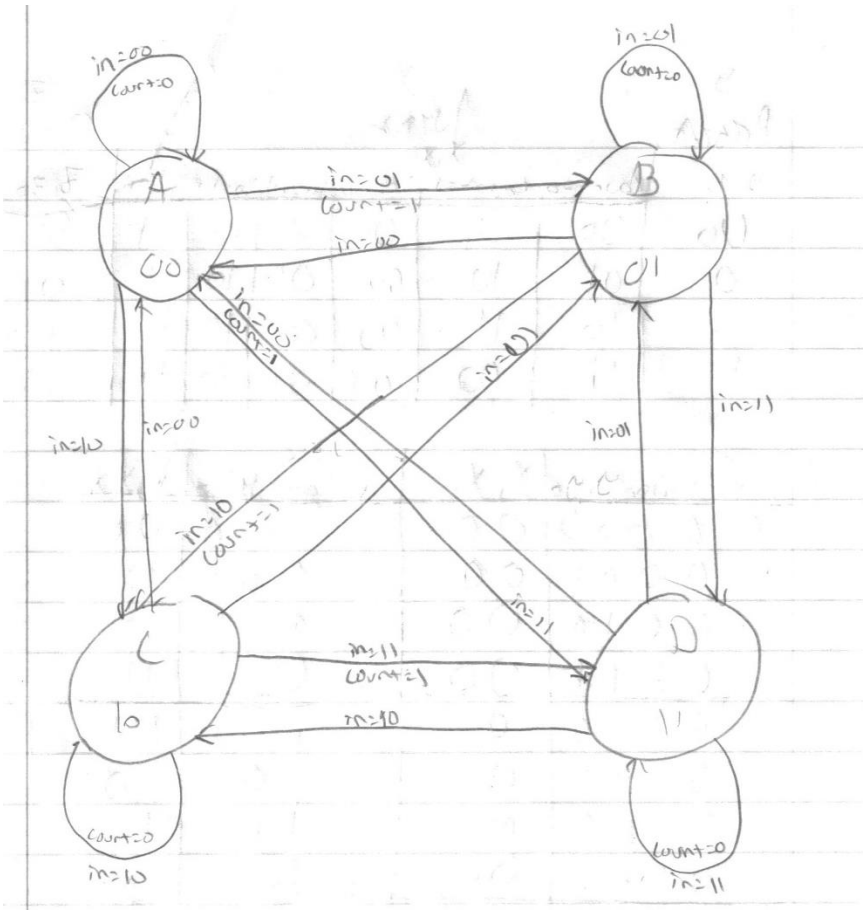
```
                //If in state 3
                2'b 11:
                    begin

                        if (lane3_has1 == 0)
                            begin

                                if (lane0_has1 == 1)
                                    begin
                                        //Move to state 0
                                        Next_State = 2'b 00;
                                        Count_Clear = 0;
                                    end
                                else if (lane1_has1 == 1)
                                    begin
                                        //Move to state 1
                                        Next_State = 2'b 01;
                                        Count_Clear = 0;
                                    end
                                else if (lane2_has1 == 1)
                                    begin
                                        //Move to state 2
                                        Next_State = 2'b 10;
                                        Count_Clear = 0;
                                    end
                                else
                                    begin
                                        //Stay in state 3
                                        Next_State = cur_state;
                                        Count_Clear = 0;
                                    end
                            end
                        else
                            begin
                                //Stay in state 3
                                Next_State = cur_state;
                                Count_Clear = 1;
                            end
                    end
            endcase
        end
    end
endmodule
```

This module selects a next state for the ACL_Machine based on if the counter has reached 5, the current state's counter has reached 0, and or there is another counter that has at least one car in it. Contrary to the priority found in modules 4_4bit_Register_File and TLCF, this always gives the green light to lane 0 first while starting out, and then checks in increasing order moving from 0 to 1 to 2 and so forth circling back to 0 after 3. If the current lane has reached 0 before the counter has reached 5, this then sends a signal, named Count_Clear, that resets the mod5 counter back to 0.

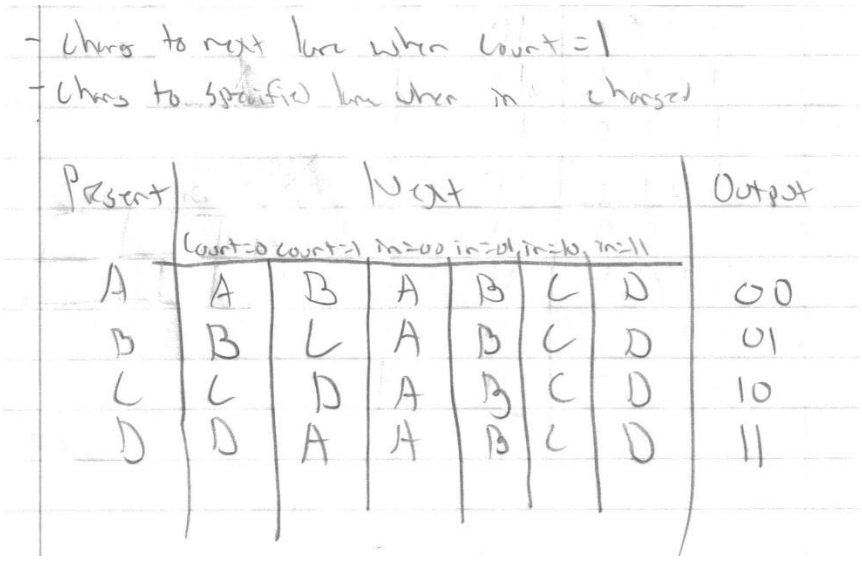
This next state is fed into D Flip-Flops and then into the block Output_Logic_ACL_Mk2 shown here:

```
module Output_Logic_ACL_Mk2 (y, Z);

    input[1:0]y;
    output[3:0]Z;

    reg [3:0]Z;

    always @(y)
    begin

        case(y)

            2'b 00: Z = 4'b 0001;
            2'b 01: Z = 4'b 0010;
            2'b 10: Z = 4'b 0100;
            2'b 11: Z = 4'b 1000;

        endcase
    end
endmodule
```

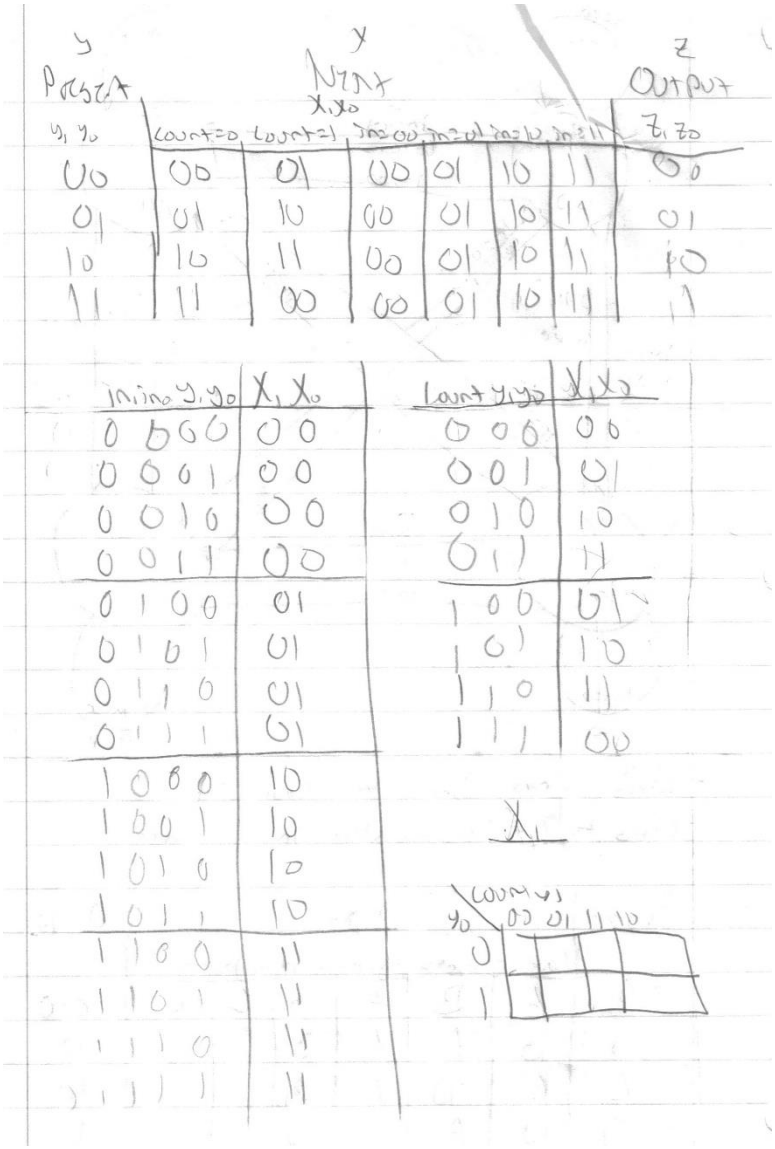
This, similar to how a decoder is one hot encoded, selects only 1 lane to decrement based on the current state.
The input and output logic were derived from the following state diagram:



Which was then translated into the following state table:



Which was then put into a truth table:



This concludes this project report. As a final note, here is a list of the different inputs and outputs used in this project and some test cases to follow if needed:

Inputs:

control_switches - switches SW0 through SW3 on the board, while initially loading lane capacities, used to select capacity for lanes. While in Mode A or B, setting switch 0 to 1 adds car lane 0, switch 1 to 1 adds cars to lane 1, and so on. Lane 3 has priority over lane 2, lane 2 has priority over lane 1 and so forth.

mode_switch - switch 17 on the board, switches between the two modes specified by the project description.

control_button - key 0 on the board. Used only for initial capacity loading.

Outputs:

HEX0 - corresponds to current amount of cars in lane 0.

HEX1 - corresponds to current amount of cars in lane 1.

HEX2 - corresponds to current amount of cars in lane 2.

HEX3 - corresponds to current amount of cars in lane 3.

HEX4 - corresponds to maximum capacity of selected lane.

LEDR0 - when lit, machine is in Mode A

LEDR1 - when lit, machine is in Mode B

LEDR17 - when lit, clock is 1

LEDR13 - when lit and in Mode B, lane 0 has a green light

LEDR14 - when lit and in Mode B, lane 1 has a green light

LEDR13 - when lit and in Mode B, lane 2 has a green light

LEDR16 - when lit and in Mode B, lane 3 has a green light

LEDG0 - when lit, loading capacity to register 0

LEDG1 - when lit, loading capacity to register 1

LEDG2 - when lit, loading capacity to register 2

LEDG3 - when lit, loading capacity to register 3

LEDG7 - when lit, loading is done, and machine can proceed to Mode A or B

Test 1:

1. Set register 0 capacity to 4 using control_switches and confirming capacity by pressing control_button
2. Set register 1 capacity to 3 using control_switches confirming capacity by pressing control_button
3. Set register 2 capacity to 2 using control_switches confirming capacity by pressing control_button
4. Set register 3 capacity to 1 using control_switches confirming capacity by pressing control_button
5. While in Mode A, fill lanes to capacity using control_switches
6. Switch to Mode B using mode_switch
7. Let lanes empty, noticing how if a lane reaches 0, the green light moves to the next lane that has cars

Test 2:

1. Set register 0 capacity to 10 (or A in hex), using control_switches
2. Set register 1 capacity to 8, using control_switches
3. Set register 2 capacity to 12 (or C in hex), using control_switches
4. Set register 3 capacity to 5, using control_switches
5. While in Mode A, fill lanes to any capacity using, control_switches
6. Switch to Mode B, using mode_switch
7. Add cars to any lane if possible, noticing how if the amount of clock cycles on a lane reaches 5, the green light moves to the next full lane, and how if a lane has a green light and user is trying to add cars to it, the lane does not increment, and the green light moves on to the next full lane after 5 clock cycles.

Test 3:

1. Set register 0 capacity to any 4-bitnumber, using control_switches
2. Set register 1 capacity to any 4-bit number, using control_switches
3. Set register 2 capacity to any 4-bit number, using control_switches
4. Set register 3 capacity to any 4-bit number, using control_switches
5. While in Mode A, fill lanes to any capacity using, control_switches
6. Switch to Mode B, using mode_switch
7. Add cars to any lane if possible, noticing how if the amount of clock cycles on a lane reaches 5, the green light moves to the next full lane, and how if a lane has a green light and user is trying to add cars to it, the lane does not increment, and the green light moves on to the next full lane after 5 clock cycles.
8. Switch between Mode A and Mode B until satisfied with the machine.