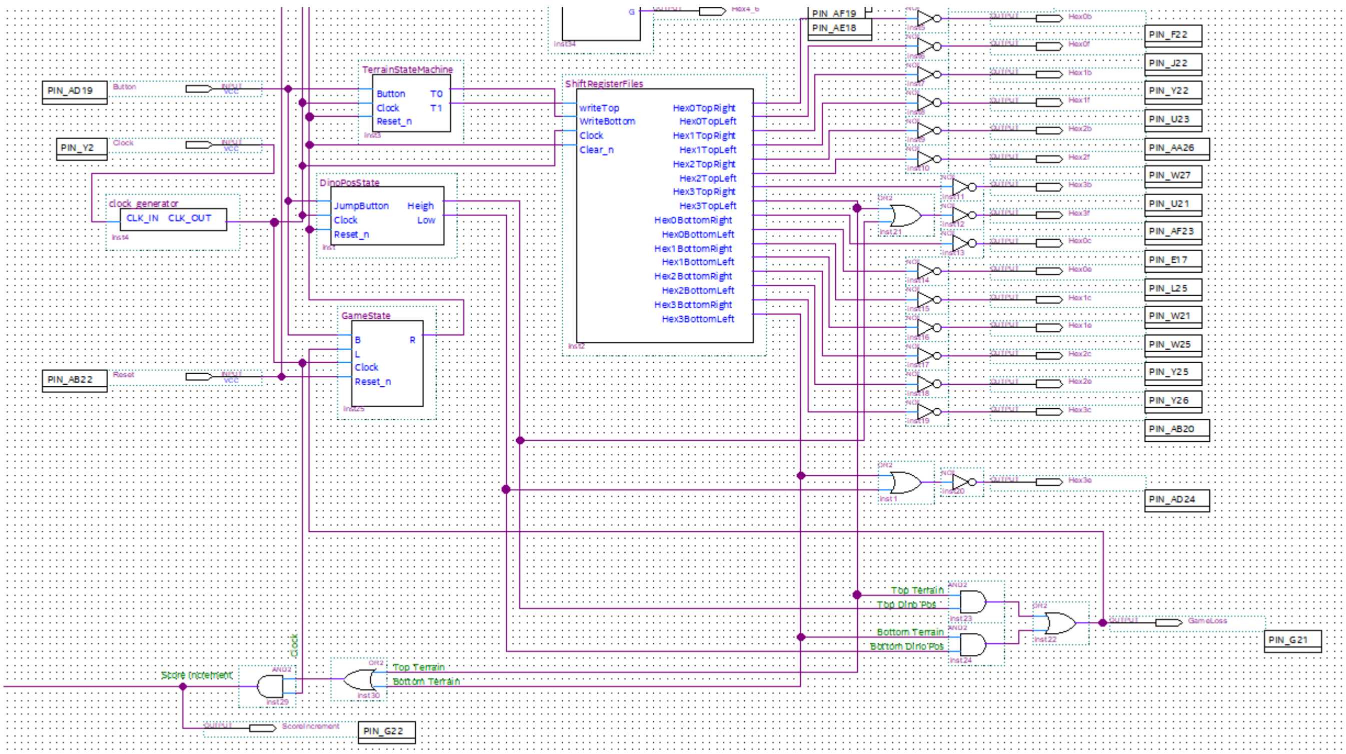
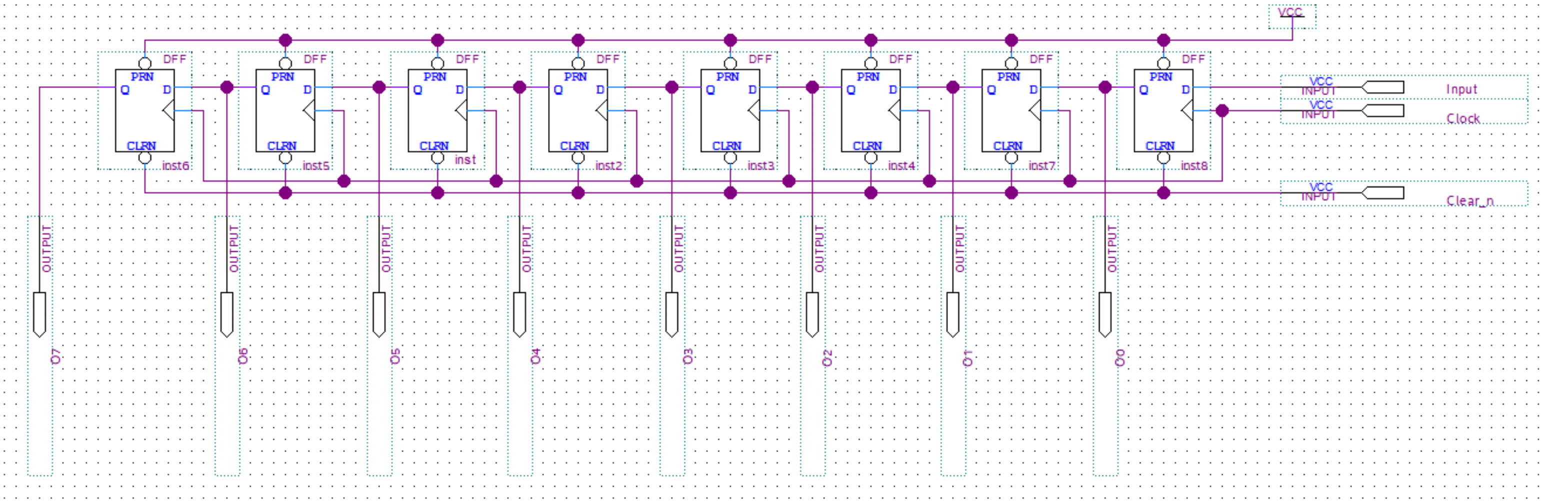
The goal of my CPRE 281 final project is to implement the Google Dinosaur game on the FPGA boards running on the seven-segment display. This game, at its base, implements State machines for game state, the jumping motion of the dinosaur, and the terrain generation. Shift register files do the terrain propagation across the screen. The game is designed to run off the seven-segment displays. Hex 3 through 0 are used to display the obstacles moving toward the dinosaur in Hex 3 on the left. As an extended goal for the project, I wanted to implement the score and high score in the top right-hand corner of the game screen for the actual dinosaur game. I successfully implemented the base game of my project shown below and then moved on to the implement score, with an 8-bit counter, eight-bit adder, and an eight-bit register.

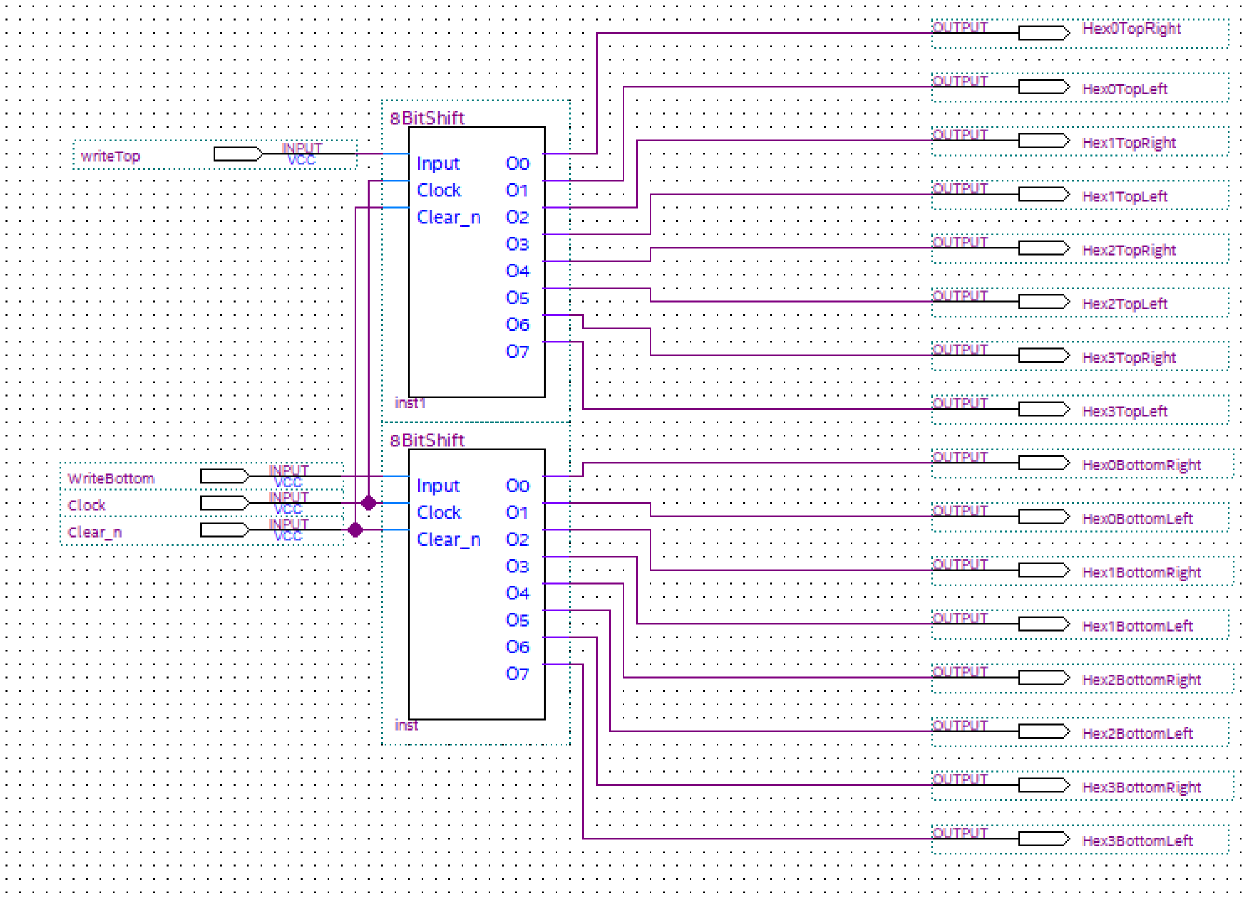


Shift Register File

The terrain propagation across the seven-segment displays is done with a shift register file, as mentioned in the introduction. There are two shift registers, one for the lower obstacles (cactus in the game) and one for the upper obstacle (bird). Each shifter will have 8-bits for each vertical segment on the seven-segment display. Each of these outputs goes to the corresponding display location. The first location on the 8-bit shifter registers is the right-most column on the seven-segment display, and the rest run left to the right, respectively. The Reset switch is necessary for new games to make the terrain blank, and the preset is unused, so tied to VCC.

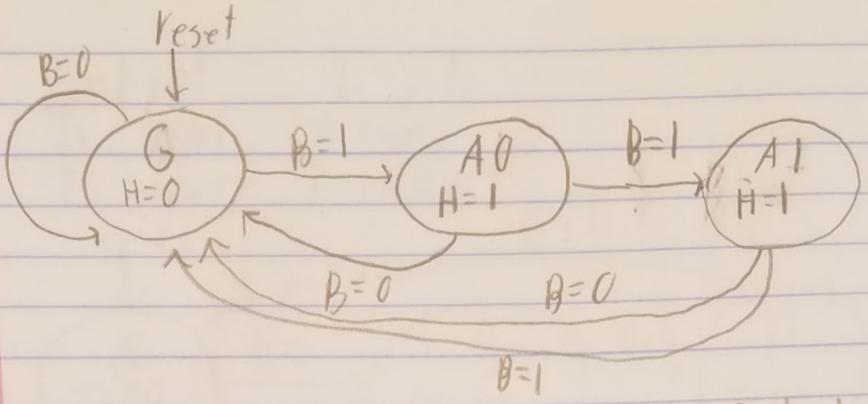


Two of these shift registers are used to create the shift register file. One is used to hold the upper terrain, and the other holds the low terrain. The two registers must be able to have no obstacles, only upper obstacles, or only upper obstacles. Having upper and lower obstacles is a does not exist state and avoided by the state machine handling terrain generation and inputting into the shift register file. The last bits are compared to the dinosaur location to determine to end game or continue. The two inputs for each bit, upper and lower, are done with a terrain generation state machine setting each bit's upper and lower. The upper and lower last bit are also compared, to do score incrementing for the extended version of the game that has scoring.



Dinosaur Jumping State Machine

The Dinosaur location can be jumping in the upper position, running on the ground, or in the lower positiion. The dinosaur cannot be jumping continuously without touching the ground for more than two terrain obstacles or clock cycles. The dinosaur is portrayed by a vertical bar in the far left 7-segment display of the four seven-segment displays making up the game screen. In the game, there are big obstacles and small obstacles. Two vertical bars in a row create the biggest obstacle. The dinosaur needs to jump over this obstacle but not any farther. This State machine will have three states. One running over the ground can go to the next stage of the first jump slot or stay on the ground without user input. The second state has the dinosaur in the air and can either go back to the ground state or can continue to the second in the air state. This state has the dinosaur in the air and will head back to the ground state regardless of user input. It outputs two bits one for the higher position and one for the lower position. These two cannot exist simultaneously due to the state machine. There are two bits to make comparing easier in the future. There is one input outside of the clock. A button is used to input whether the user wants to jump or maintain his jump longer. If the dinosaur has already maintained his jump for two clock cycles or two terrain obstacles, the state machine can overwrite this, and he goes back to the ground. In no case will the dinosaur need to jump for more than those two clock cycles. The reset position of the state machine is the lower position, with the dinosaur is on the ground. The ground is the state the dinosaur sits at between games.



Preset State	Next State B (Button)=0	Next State B=1	Output H (UP)	L (Down)
G (ground)	G	A1	0	1
A0 (Air 0)	G	A2	1	0
A1 (Air1)	G	G	1	0

$G = 00$ ($S_1 S_0$) $A0 = 01$ ($S_1 S_0$) $A1 = 10$ ($S_1 S_0$)

Present State		Next State B=0	Next State B=1	Output H	Output L
G	00	00	01	0	1
A0	01	00	10	1	0
A1	10	00	00	1	0
	11	dd	dd	d	d

w	S_1	S_0	S_{1new}	S_{0new}	H
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	d	d	d
1	0	0	0	1	
1	0	1	1	0	
1	1	0	0	0	
1	1	1	d	d	

S_{1new}

w \ S_1S_0	00	01	11	10
0	0	0	d	0
1	0	1	d	0

$S_{1new} = w S_0$

H

S_0 \ S_1	0	1
0	0	1
1	1	d

$H = S_1 + S_0$

S_{0new}

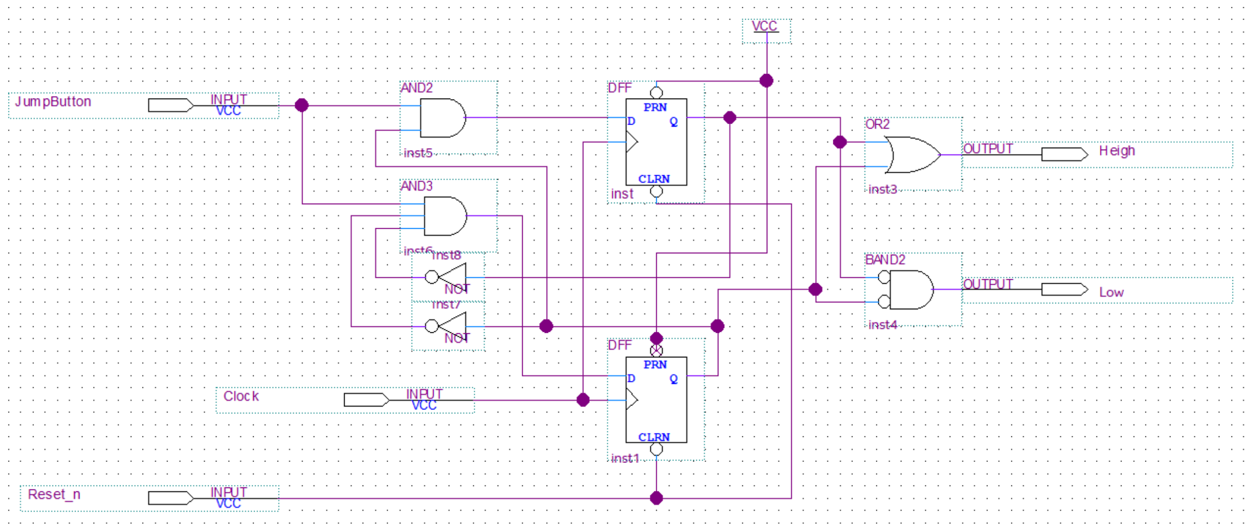
w \ S_1S_0	00	01	11	10
0	0	0	d	0
1	1	0	d	0

$S_{0new} = \overline{S}_1 \overline{S}_0 w$

L

S_0 \ S_1	0	1
0	1	0
1	0	d

$L = \overline{S}_0 \overline{S}_1$



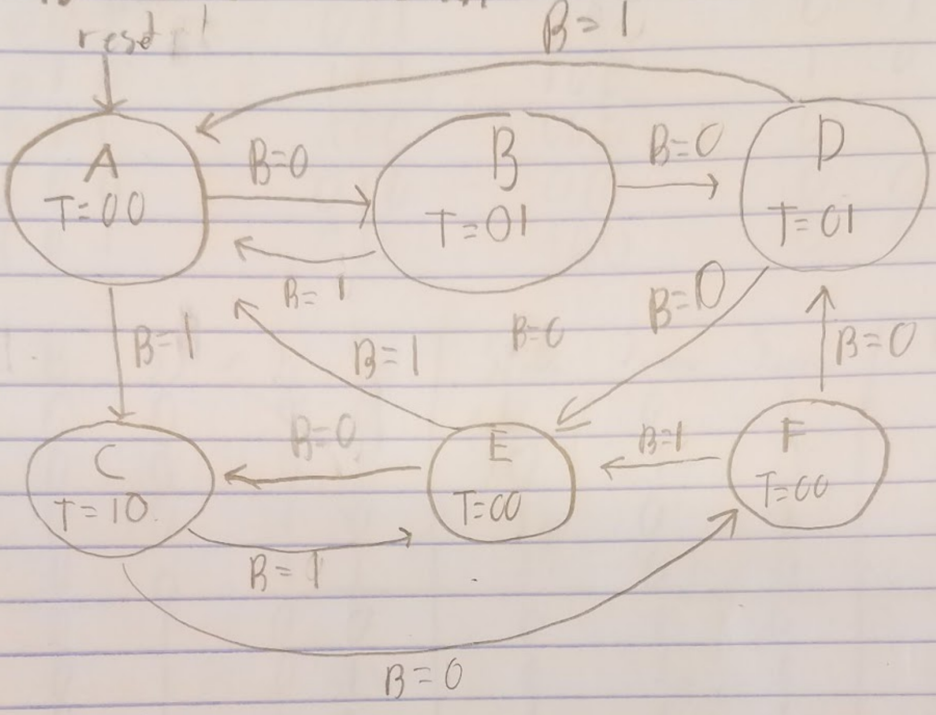
Terrain Generation

A state machine handles the terrain generation; it will have two outputs for each shift register. It is designed to only have two lower barriers in a row, which is the longest jump length of the dinosaur. In addition to going from an upper obstacle to lower or vice versa, there needs to be at least one state with no terrain to ensure the dinosaur can always successfully navigate through the obstacles. The reset state of the state machine has no obstacles generated. This state is maintained between games. The overall design of the state machine seems relatively disorganized. The chaos in design is intentional to make the terrain generation look relatively random throughout gameplay. After testing the game, it does not fall into any recognizable pattern, although there cannot be too much diversity in the game as a whole, with only two types of terrain. The input of the state machine is the user input button for dinosaur jumping. Surprisingly, this alone worked adequatly to make a relatively randomized terrain generator.

Output T_1 , T_0

T_1 upper terrain

T_0 lower terrain



Present State	Next State B=0	Next State B=1	Output T_1	Output T_0
A	B	C	0	0
B	D	A	0	1
C	F	E	1	0
D	E	A	0	1
E	C	A	0	0
F	D	E	0	0

A	000	D	011
B	001	E	100
C	010	F	101

Preset State	Next State		Out put	
	B=0	B=1	T_1	T_0
000	001	010	0	0
001	011	000	0	1
010	101	100	1	0
011	100	000	0	1
100	010	000	0	0
101	011	100	0	0

B	S_2	S_1	S_0	$S_{2\,new}$	$S_{1\,new}$	$S_{0\,new}$	T_1	T_0
0	0	0	0	0	0	1	0	0
0	0	0	1	0	1	1	0	1
0	0	1	0	1	0	1	1	0
0	0	1	1	1	0	0	0	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	1	0	0
0	1	1	0	d	d	d	d	d
0	1	1	1	d	d	d	d	d
1	0	0	0	0	1	0		
1	0	0	1	0	0	0		
1	0	1	0	1	0	0		
1	0	1	1	0	0	0		
1	1	0	0	0	0	0		
1	1	0	1	1	0	0		
1	1	1	0	d	d	d		
1	1	1	1	d	d	d		

S_2new

$S_1S_0 \backslash BS_2$	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	d	d	0
10	1	d	d	1

$S_2 new = S_1\bar{S_0} + S_1\bar{B} + S_0 B S_2$

$S_1 new$

$S_1S_0 \backslash BS_2$	00	01	11	10
00	0	1	0	1
01	1	1	0	0
11	0	d	d	0
10	0	d	d	0

$S_{1new} = \bar{B}S_2 + \bar{S_1}S_0\bar{B} + \bar{S_1}\bar{S_0}B\bar{S_2}$

$S_0 new$

$S_1S_0 \backslash BS_2$	00	01	11	10
00	1	0	0	0
01	1	1	0	0
11	0	d	d	0
10	1	d	d	0

$S_{0new} = \bar{B}\bar{S_2}\bar{S_1} + \bar{S_1}S_0\bar{B} + \bar{B}S_1\bar{S_0}$

T_1

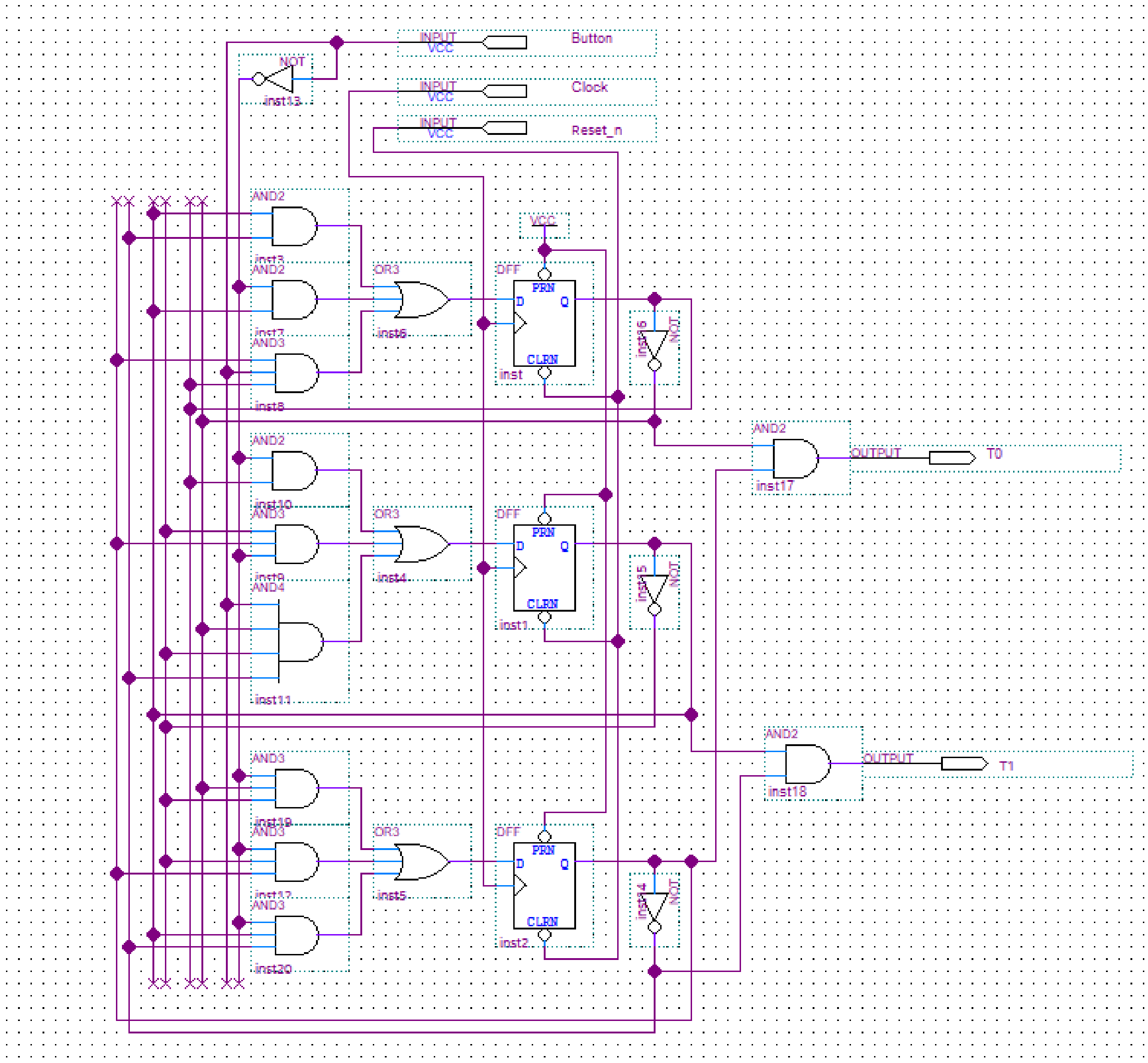
$S_0 \backslash S_2S_1$	00	01	11	10
0	0	1	d	0
1	0	0	d	0

$T_1 = \bar{S_0}S_1$

T_0

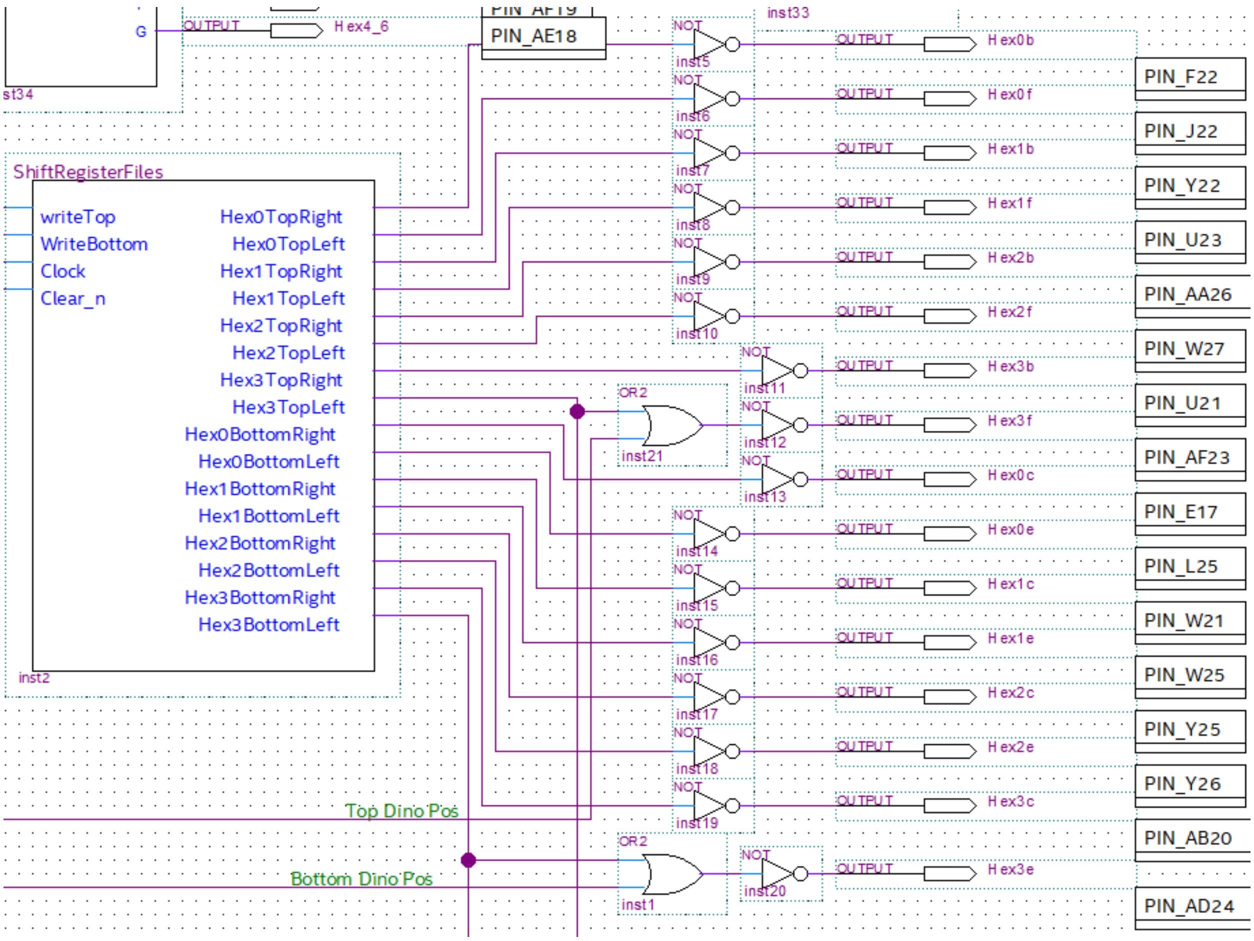
$S_0 \backslash S_2S_1$	00	01	11	10
0	0	0	d	0
1	1	1	d	0

$T_0 = S_0\bar{S_2}$



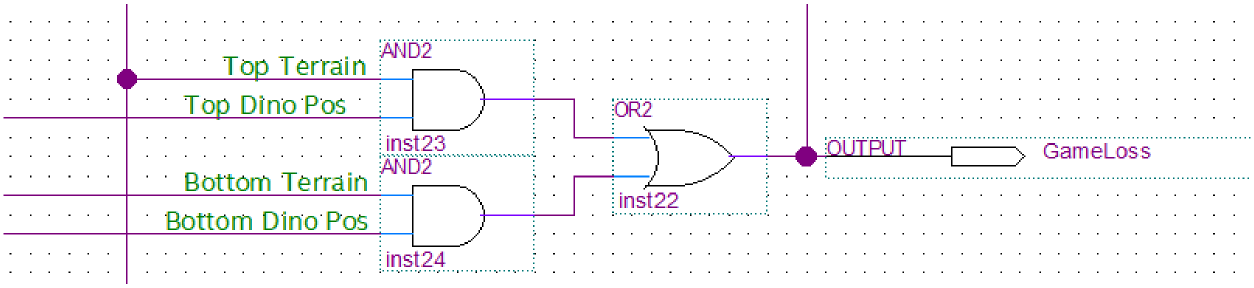
Display of Game

The base game display is on Hex 0 through Hex 3; the final terrain location and dinosaur position outputs need to be or’ed because both can affect the display. Then, all outputs need to be negated because the seven-segment displays require a 1 to be blank and a 0 to be lit up.

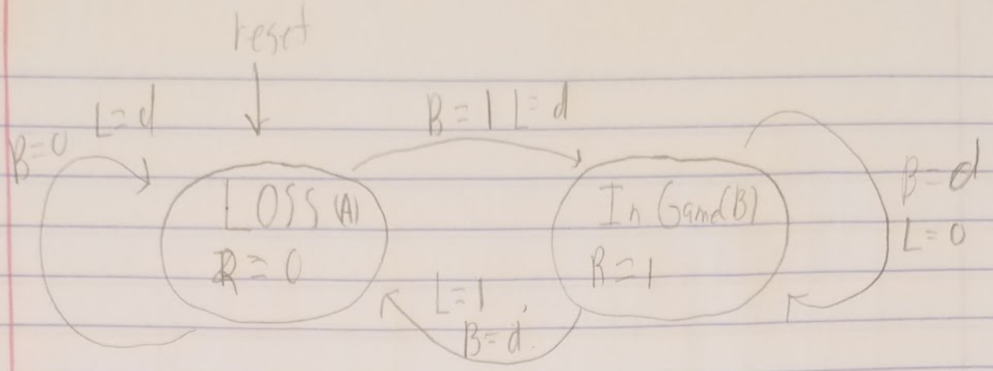


Game State

The game state handles if the player has reached a game loss or if the player wants to start a new game. There are two states; in-game and game lose. The state machine has two inputs, the button input for starting a new game and the game loss input. The game loss input was created to minimize the inputs to the state machine and simplify it as a whole. The logic of the game state determines if the game is in a position where the player loses. It checks if the dinosaur is in the same location as a terrain obstacle, so if it moves into a bird or runs into a cactus. The logic is TopTerrain*TopDinoPos+BottomTerrain*BottomDinoPos = GameLoss. This output is then tied to a LED for debugging purposes and runs into the State Machine.



There is only one output to the State machine named ‘reset.’ The output is used to reset the dinosaur position state machine, the terrain generation state machine, and clears the shifter to make an empty game whenever a new game is started. In the full version of the game, the output also resets the current score to zero. A new game is started when in the loss state, and the user inputs a button depression which also jumps the dinosaur. This input is the same way the user starts a new game on Google’s dinosaur game.



L= Low Pos • Hex3 Bottom Left + HeighPosHex3TopLeft

	Current State	Next State B=0 L=0	B=0 L=1	B=1 L=0	B=1 L=1	Output R
0	A	A	A	B	B	0
1	B	B	A	B	A	1
	0	0	0	1	1	0
	1	1	0	1	0	1

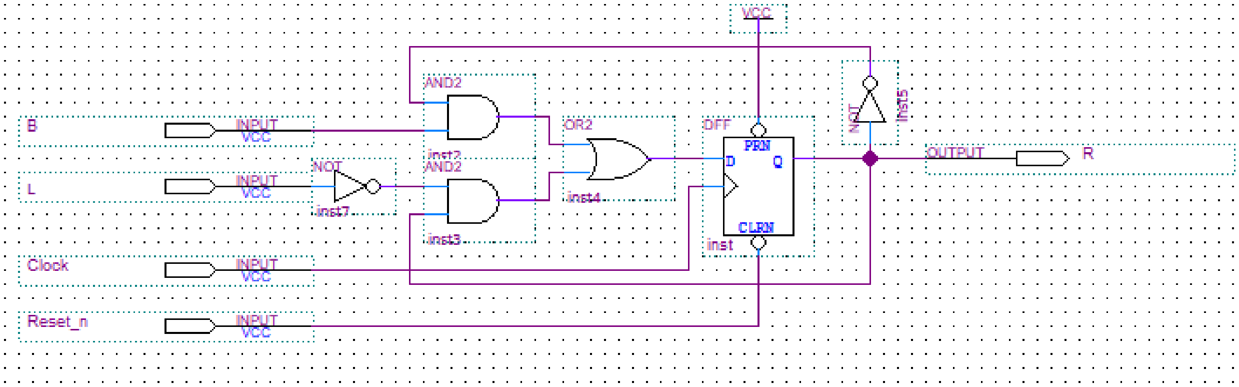
B	L	S	S_{new}	R
0	0	0	0	0
0	0	1	1	1
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	1	
1	1	1	0	

S_{new}

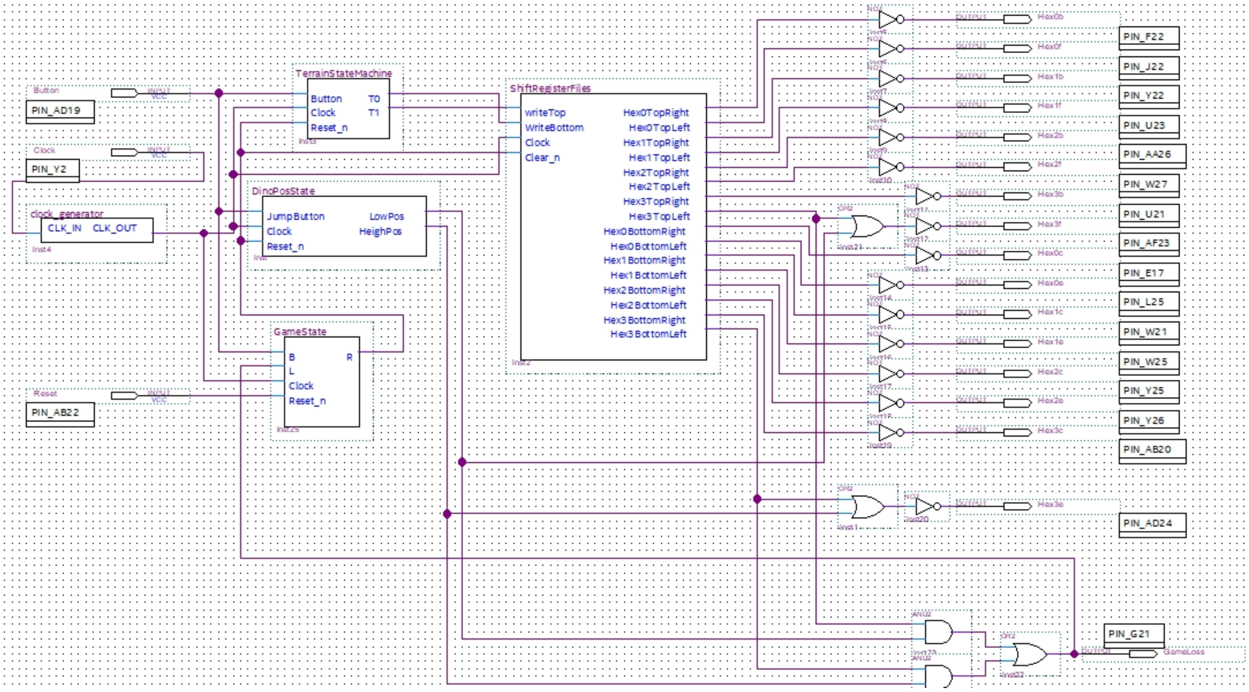
S \ BL	00	01	11	10
0	0	0	1	1
1	1	0	0	1

$S_{new} = \bar{S} B + S \bar{L}$

$R = S$



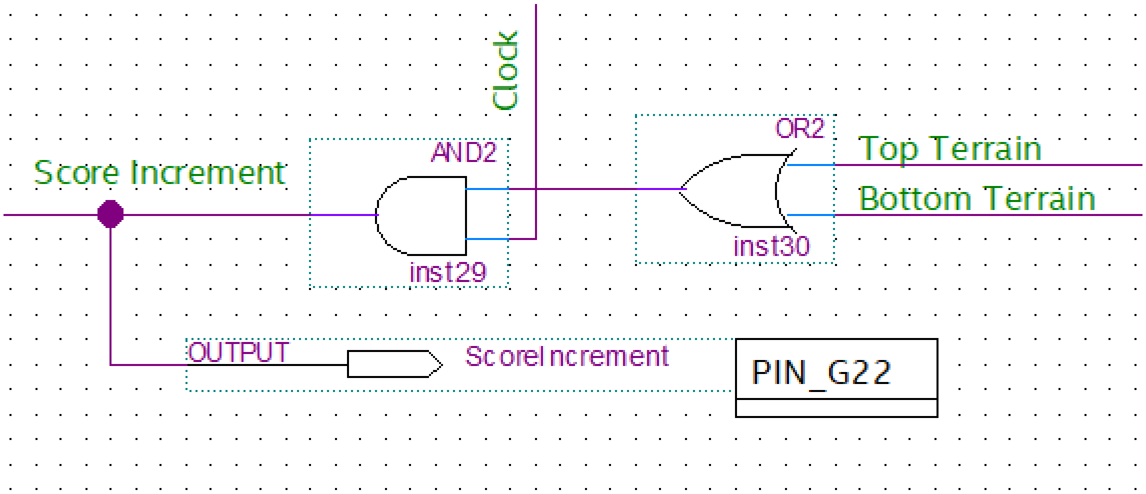
The game state machine is the final addition finishing the base game I wanted to implement for the minimum goal. The game played well but missed a big aspect of many video games, and the Dinosaur game, more specifically, a score and high score.



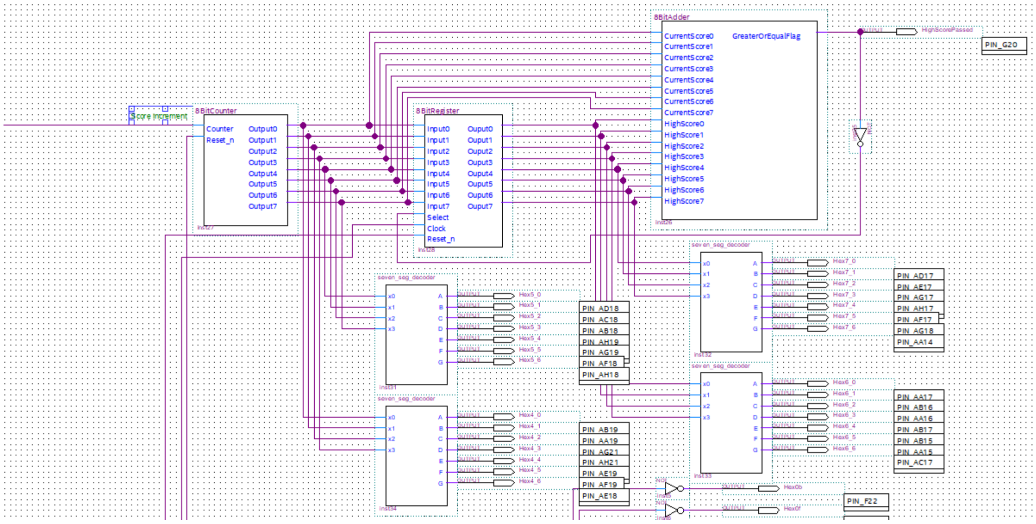
Score Counter

The Score is the number of obstacles passed by the user. To count the number of terrain pieces passed by the dinosaur, the output of the two final terrain locations, the same one used for determining game states, is or’ed and then and’ed with the clock.

This design made it so that if two of the same terrain piece were in a row, the score would still increment, and if there were no terrain, the score would not increment. This data ended up being the most critical part of the entire score system. The score increment line is also connected to a LED for debugging purposes.

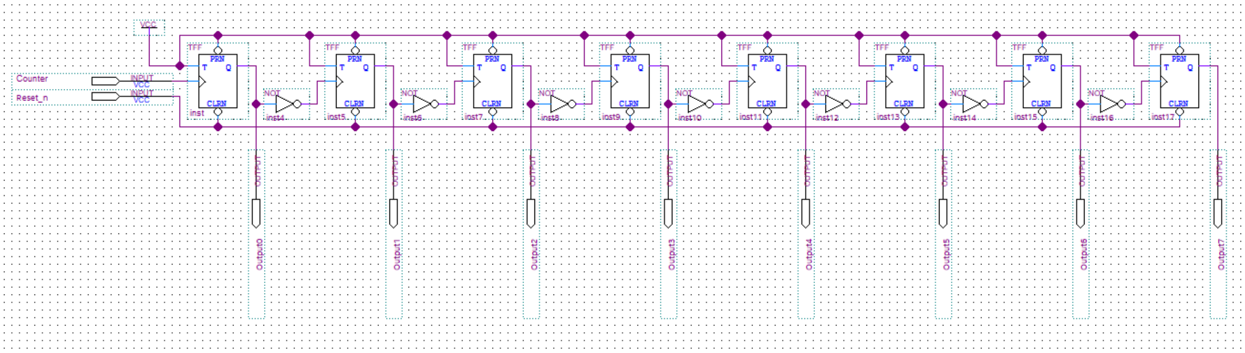


This portion of the project was made up of four parts. An 8-bit counter to keep track of the current score, an 8-bit register to hold the high score, and an 8-bit adder to compare the current score and high score and save the current score accordingly, and then finally, the seven-segment decoder for displaying the score during and in between games.



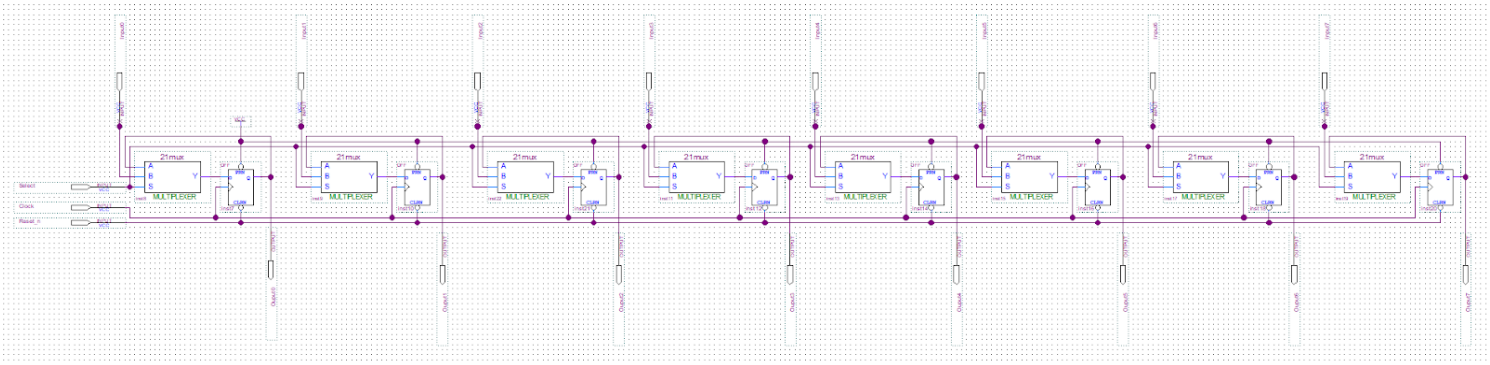
8-Bit Counter

The score counter is an 8-bit asynchronous up counter with reset. I made it asynchronous to make it as simple as possible. I was not concerned by the internal delay caused by this design because of how slow the clock speed and therefore score increment speed is for the game. The reset pin is connected to the output of the Game State state machine to allow the score to reset between games. The counter line is connected to the above-described score increment line. The score is then output to the two seven-segment displays, the register for storing if it is the high score, and the adder for comparing to the current high score. I chose to count up in binary and not alter my counting sequence to be more conducive to decimal because I was limited on the number of seven-segment displays. I could only have two for the current score if I wanted it to be as similar to the actual game and display both simultaneously. I displayed the numbers in hexadecimal because that allowed me to have the maximum score of 255 as FF instead of 99 if I used decimal.



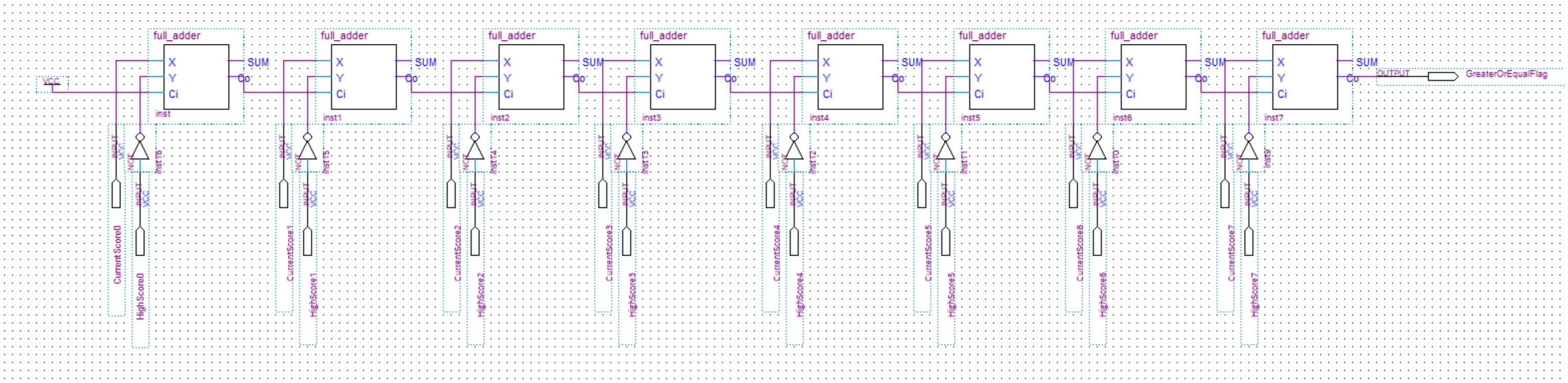
8-Bit Register

Since the last reset, the 8-bit register has stored a high score. The reset is done by a switch on the board or by a new flash of the game onto the FPGA board. It outputs the score to the adder for comparing and the seven-segment display decoders. The inputs are a clock for the D flip flops, a select line, and 8-bits of input from the counter. On the way to the register, the select line is negated because the built-in multiplexers in Quartus have the A-line refer to a select of 1, and the B line refers to 0. The A-line inputs itself back into the store, and the B line writes the counter number into the register. The 8-Bit Register is the only aspect of this project that makes use of the 50MHz clock and not the significantly slowed output of the clock generator. This is so it will update almost instantaneously from the counter if the high score needs to be updated.



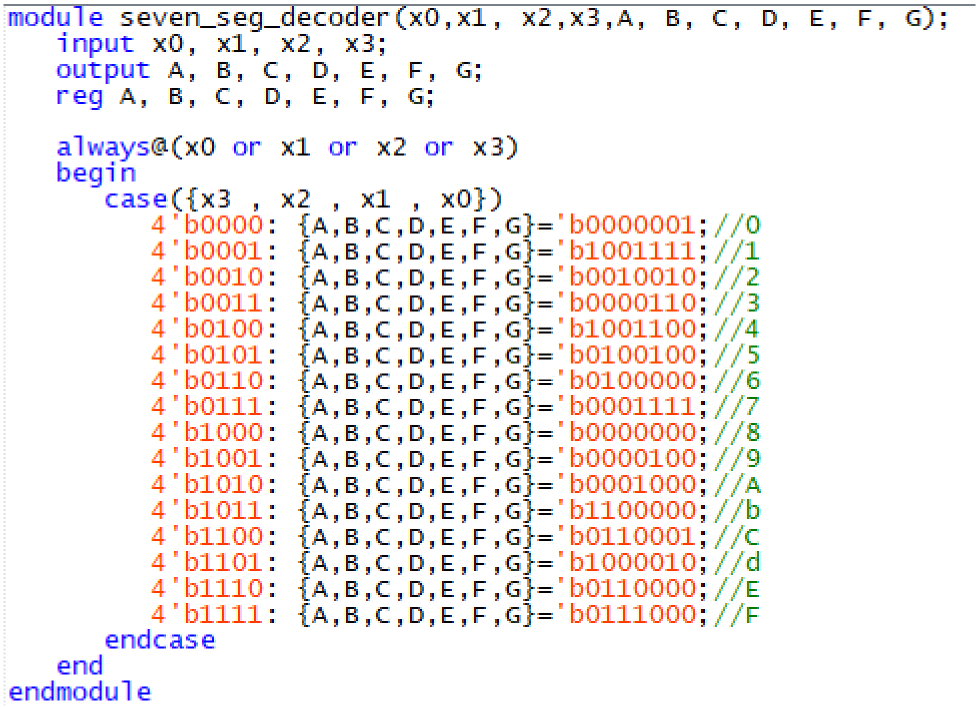
8-Bit Adder

The 8-bit adder compares the current score and high score to see if the current score is larger than the high score and needs to be updated to be the current score. It does this with the carry flag. It takes the current high score and subtracts the current score from it. The sum byte is then ignored and the carry bit is taken to the select line of the register negated. The 8-bit adder is made of 8 full adders with the carries going to the next making it a ripple carry adder. Although this can cause issues due to the delays in an adder similar to the 8-bit counter, I was not concerned about this because the delay of 8 is not particularly high. With such a slow clock speed and lack of need of precision I went with the simpler design. This makes it so if the current score is greater than the current high score it will write the current score to the high score.



Display of Score

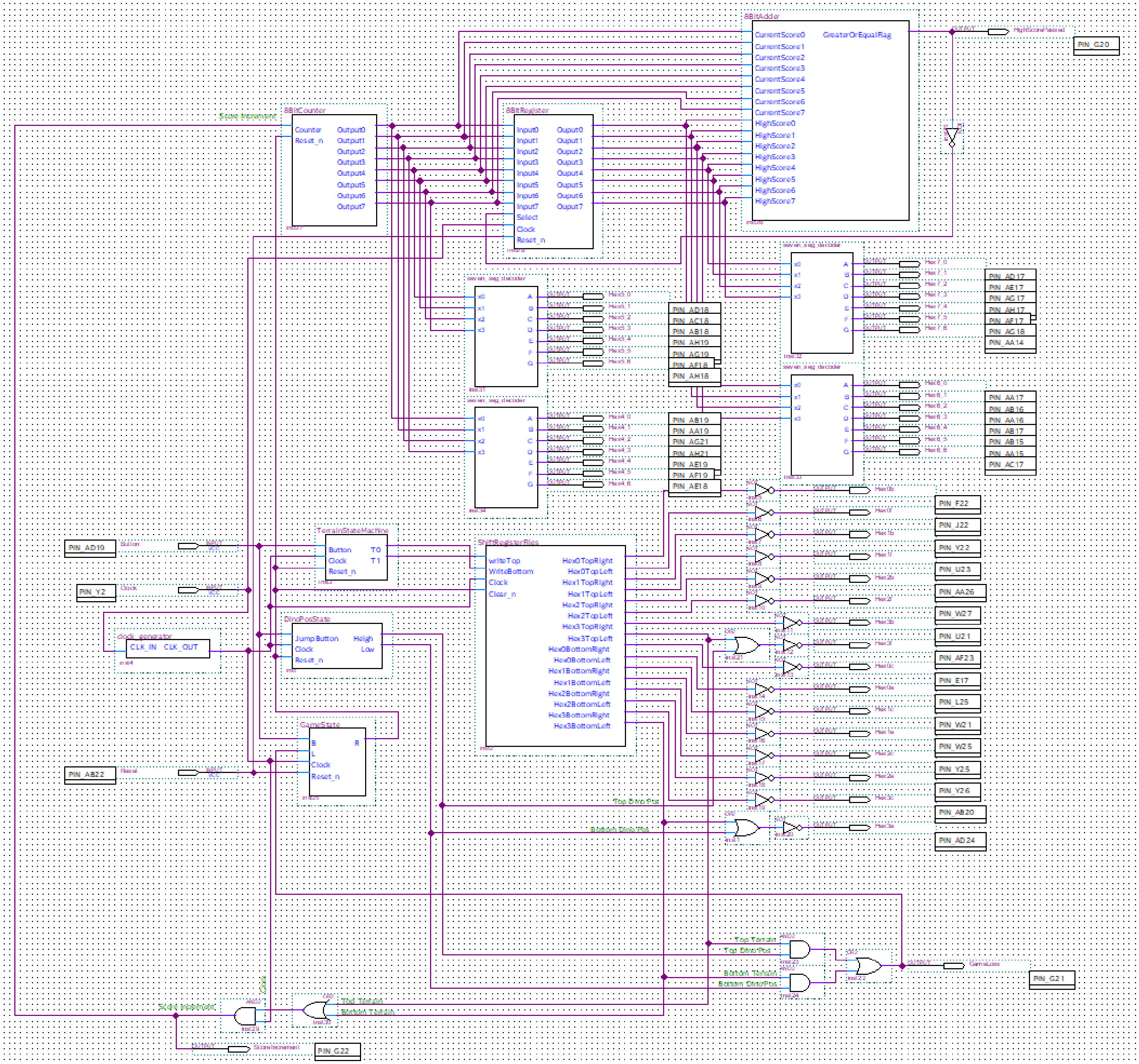
The scores, as mentioned earlier, are displayed on the seven segment displays two for each score. The current score is on Hex 5 and Hex 4 and the high score is displayed on Hex 7 and Hex 6. These are both in the hexadecimal and go from binary to the displays with the 7 segment decoder we made in lab 5 in Verilog. The left set displays the high score and the right one displays the current score, identical to the order they are displayed in Google's version. I would like to build an alternative version of the game that switches between the score but is in decimal on all 4 of the displays, which would allow a max score of 9999. As it stands, the max score is 255 displayed as FF which is still high enough that in my testing it never went above and wrapped around but a lot more conceivable to occur that the 9999 max score option.



```
module seven_seg_decoder(x0,x1, x2,x3,A, B, C, D, E, F, G);
   input x0, x1, x2, x3;
   output A, B, C, D, E, F, G;
   reg A, B, C, D, E, F, G;

   always@(x0 or x1 or x2 or x3)
   begin
      case({x3 , x2 , x1 , x0})
         4'b0000: {A,B,C,D,E,F,G}='b0000001;//0
         4'b0001: {A,B,C,D,E,F,G}='b1001111;//1
         4'b0010: {A,B,C,D,E,F,G}='b0010010;//2
         4'b0011: {A,B,C,D,E,F,G}='b0000110;//3
         4'b0100: {A,B,C,D,E,F,G}='b1001100;//4
         4'b0101: {A,B,C,D,E,F,G}='b0100100;//5
         4'b0110: {A,B,C,D,E,F,G}='b0100000;//6
         4'b0111: {A,B,C,D,E,F,G}='b0001111;//7
         4'b1000: {A,B,C,D,E,F,G}='b0000000;//8
         4'b1001: {A,B,C,D,E,F,G}='b0000100;//9
         4'b1010: {A,B,C,D,E,F,G}='b0001000;//A
         4'b1011: {A,B,C,D,E,F,G}='b1100000;//b
         4'b1100: {A,B,C,D,E,F,G}='b0110001;//C
         4'b1101: {A,B,C,D,E,F,G}='b1000010;//d
         4'b1110: {A,B,C,D,E,F,G}='b0110000;//E
         4'b1111: {A,B,C,D,E,F,G}='b0111000;//F
      endcase
   end
endmodule
```

With the score, display added, it completes all the portions to implement the Dinosaur game of 7 segment displays on an FPGA board.



How To Run and Play

The full game is stored in DinosauGame(Full) file, open the project, compile and program the FPGA board. To play, have SW3_DB set to 1 (the reset switch for the Game State and the high score) to play the game. Press and hold down KEY0_DB; this is the button to make the dinosaur jump. To make the dinosaur jump over two obstacles in a row, the button must be depressed for both clock cycles to do so. The Game itself is displayed on Hex 0 through Hex 3. The dinosaur is the single vertical bar located in the bottom left corner during the reset state. The dinosaur will stay in the far left for the game's entirety and be low or high. The score is displayed on Hex 5 and 4, and the high score is displayed on Hex 7 and 6. Both scores are displayed in Hexadecimal. There are three green LEDs that can be lit for debugging purposes. LEDG7 is Game Loss; it lights up whenever the game is in a state that the player loses. LEGG6 blinks light up every time the score should increment up to one. Finally, LEDG5 Lights up whenever the high score is passed, and the high score needs to be updated. Once the game is lost a new game can be started by holding the jump button down again similar to the initial game start. To reset the high score back to zero and restart the game if in a current game SW3_DB to zero then one.