Garrett Trickey and Matthew Smith
CPRE 281
Spring '24

Final Project

Names: Garrett Trickey, Matthew Smith

Professor Selim

CprE 281

28 April 2024

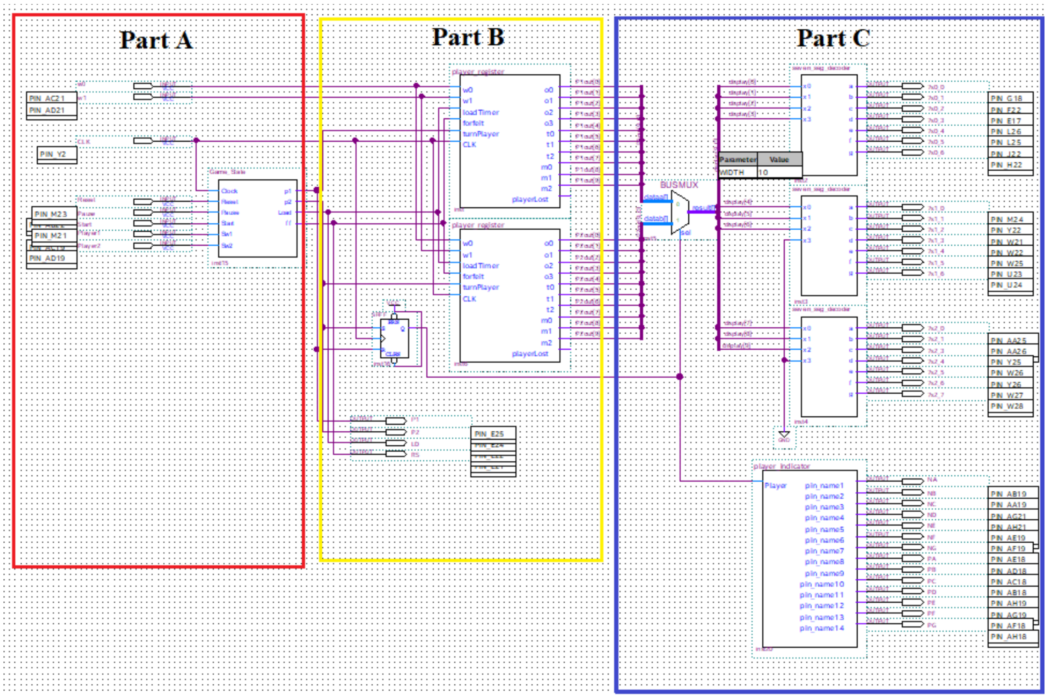
Final Project Report

Within this project, we decided to work on the Chess Clock Timer. This report will detail all the work done by Garrett Trickey and Matthew Smith, which includes the working and derivations of the built circuits and Verilog code that is utilized in our project.

Purpose:

Our project aimed to simulate a chess clock, a tool that is used in chess tournaments for setting player's time during matches. We developed a digital emulation of this device, utilizing a combination of a finite state machine and a register file. Essentially, our design operates as a down counter, managing and alternating between players' time allocation. As one player's time decreases, our system saves the remaining time for the opposing player, making a seamless transition as the game progresses until on e player's time runs out.

Top Level Diagram

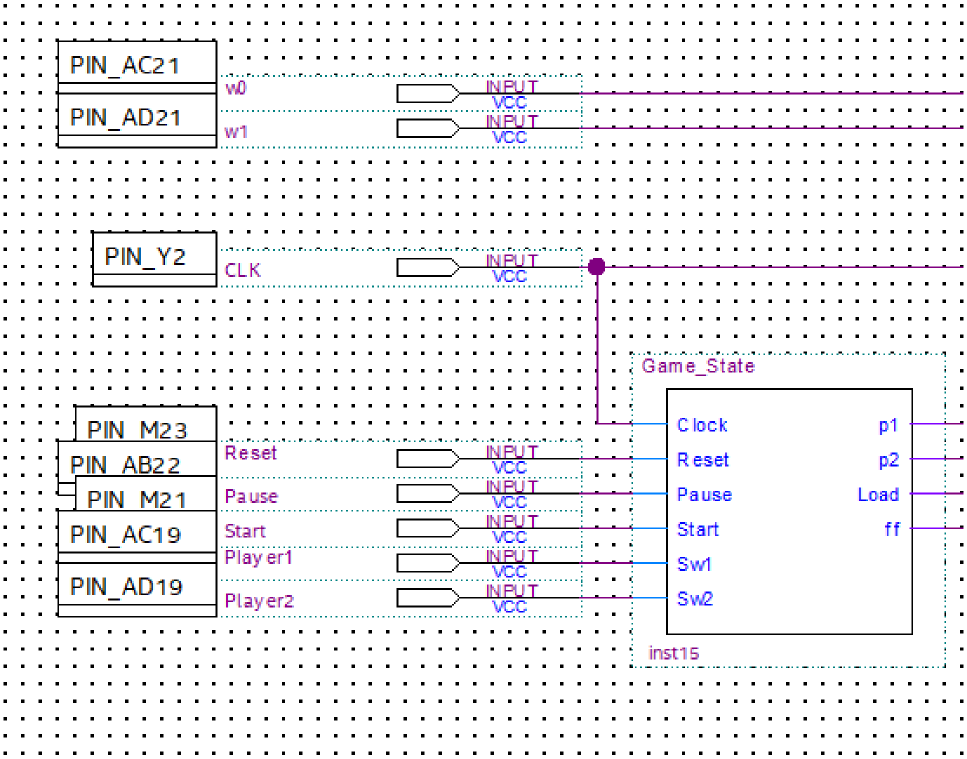


Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project

This is the top-level diagram, where Part A contains our Finite State Machine, Part B contains our Register File and Down Counter with additional logic, Part C contains our outs onto the seven segment displays.

Part A: Finite State Machine

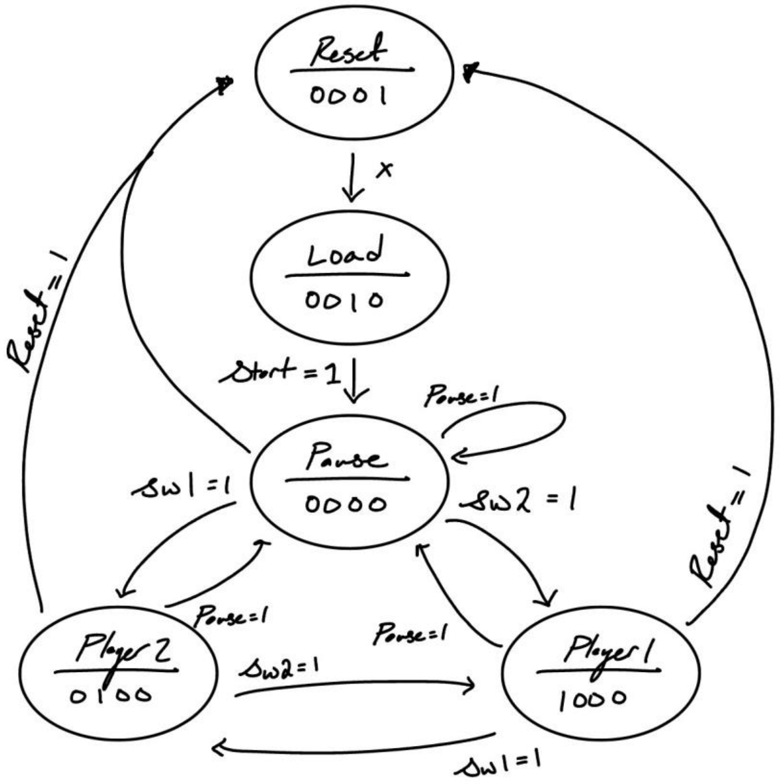


Within our chess timer, we needed to store different game states. The break down consists of our Reset State, Load State, Pause State, Player1 State, Player2 State. Reset State purpose is to clear the timer and start back to allow a new game to start again. The Load state purpose is primarily to prevent the case of loading new times into the register file and hitting clear at the same time. This is a buffer state before going onto the Pause State. The pause state main functionality is to pause the timer and not allow any of the players time to decrement while in this state. Player 1 and Player 2 States act as an enable for the timer to decrement their state.

Garrett Trickey and Matthew Smith
CPRE 281
Spring '24

Final Project

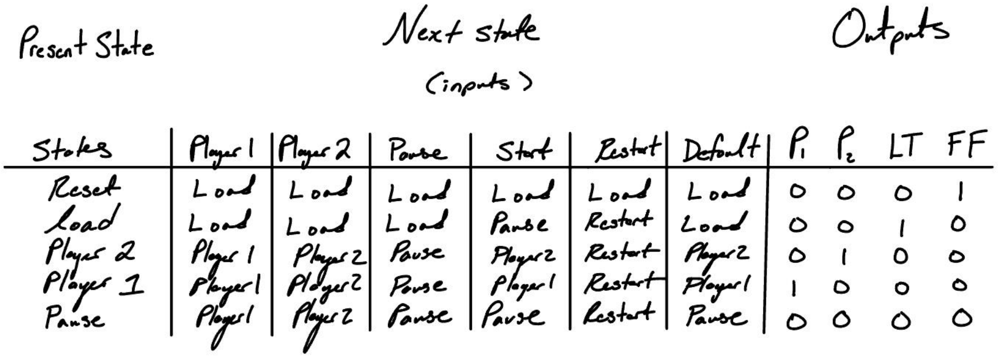
State Diagram:



State Table: One Hot Encoded:

Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project



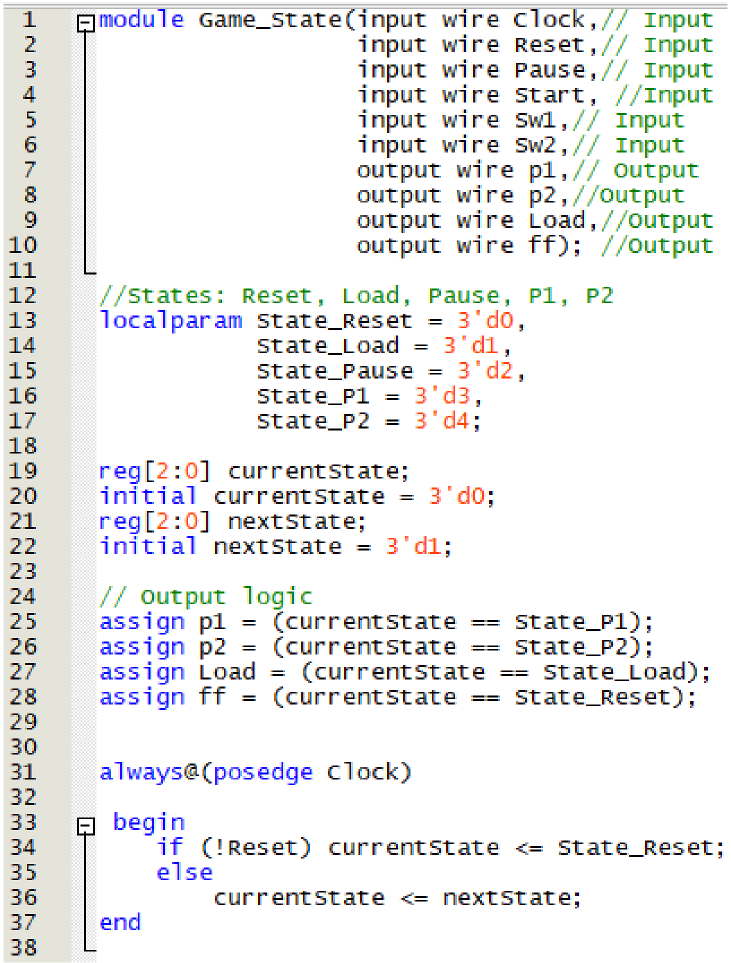
Present State	Next state (inputs)						Outputs			
States	Player 1	Player 2	Pause	Start	Restart	Default	P_1	P_2	LT	FF
Reset	Load	Load	Load	Load	Load	Load	0	0	0	1
load	Load	Load	Load	Pause	Restart	Load	0	0	1	0
Player 2	Player 1	Player 2	Pause	Player 2	Restart	Player 2	0	1	0	0
Player 1	Player 1	Player 2	Pause	Player 1	Restart	Player 1	1	0	0	0
Pause	Player 1	Player 2	Pause	Pause	Restart	Pause	0	0	0	0

Within this state table configuration, we are using a one hot encoding method. This includes using as many state variables as there are states. This method each state all but one output are equal to zero, as shown in our puts. Noticing at the Pause state there is no

Implementation of the FSM using Verilog:

Garrett Trickey and Matthew
Smith
CPRE 281
Spring '24

Final Project

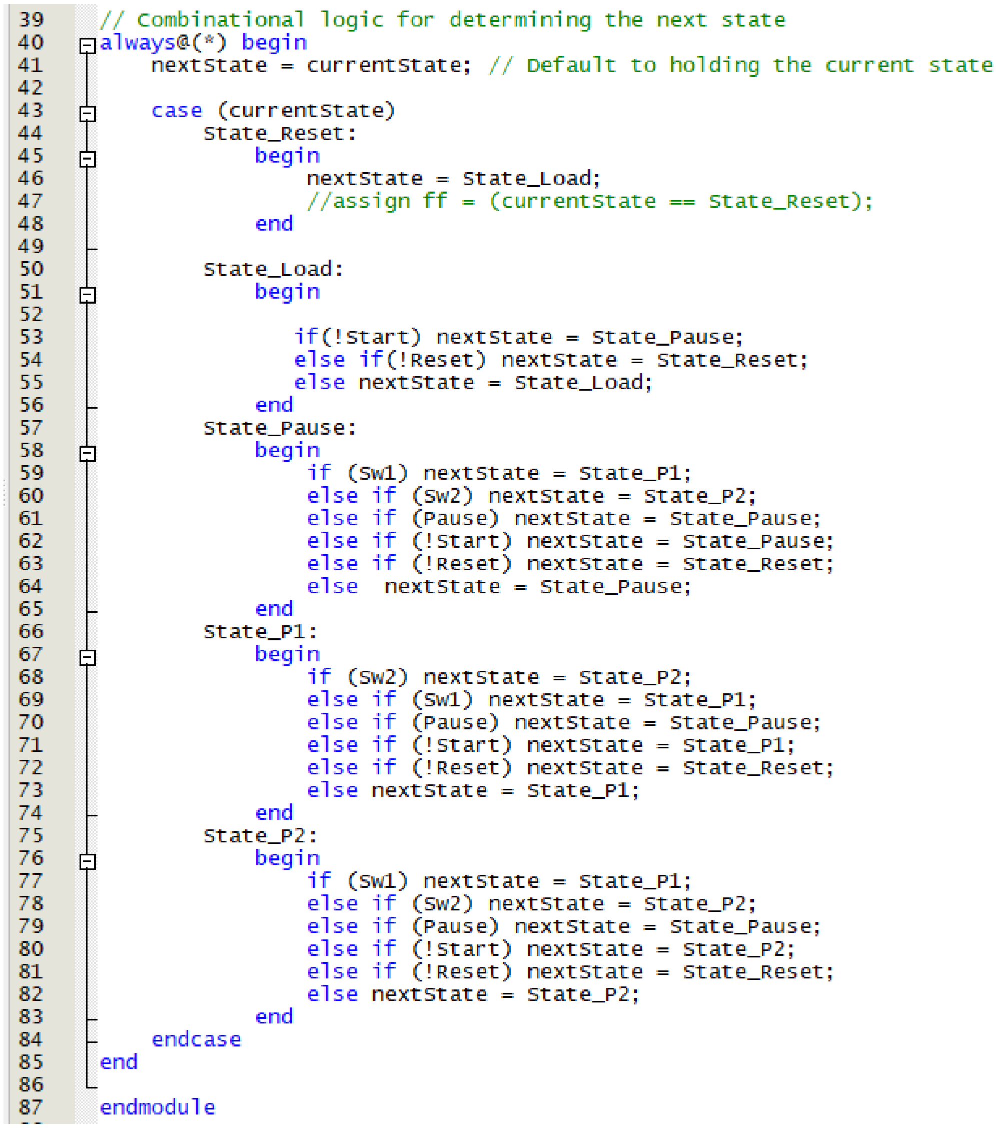


```
module Game_State(input wire Clock,// Input
                  input wire Reset,// Input
                  input wire Pause,// Input
                  input wire Start, //Input
                  input wire Sw1,// Input
                  input wire Sw2,// Input
                  output wire p1,// Output
                  output wire p2,//Output
                  output wire Load,//Output
                  output wire ff); //Output

//States: Reset, Load, Pause, P1, P2
localparam State_Reset = 3'd0,
           State_Load = 3'd1,
           State_Pause = 3'd2,
           State_P1 = 3'd3,
           State_P2 = 3'd4;

reg[2:0] currentState;
initial currentState = 3'd0;
reg[2:0] nextState;
initial nextState = 3'd1;

// Output logic
assign p1 = (currentState == State_P1);
assign p2 = (currentState == State_P2);
assign Load = (currentState == State_Load);
assign ff = (currentState == State_Reset);


always@(posedge Clock)

 begin
    if (!Reset) currentState <= State_Reset;
    else
        currentState <= nextState;
end

```

We decided the best route to implement the FSM is through Verilog, to maintain this logic we need to define our inputs and outputs as wires, wires in Verilog are used to connect input and output of ports of a module instantiation together. Our states are defined under localparam, this was the easiest method to encode our states in Verilog, since we have 5 states, we need 3 bits to store these into our FSM.

Then our Output logic is based off the outputs of our FSM into our register file with the down counter, that will be explained later in **PartB,** each of these outputs are determined based off which state its currently located, since FSM operate by being only in one state at a time or clock cycle. This works out nicely in our outputs. The output ff is utilized as a reset output for the timers for part B.

Every time this program runs, we need to synchronous our state transition at the beginning by stating our current state to be located in load. This allows us to load in which time we want, whether its 1,2,5, or 7 mins into the register file and when the input start is hit, will start our game



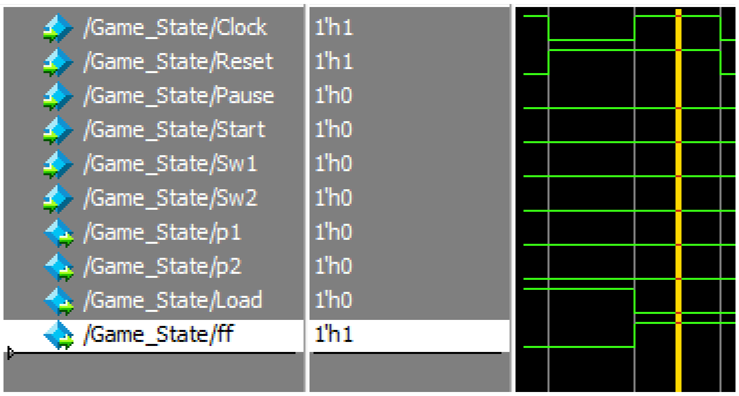
```
// Combinational logic for determining the next state
always@(*) begin
    nextState = currentState; // Default to holding the current state

    case (currentState)
        State_Reset:
            begin
                nextState = State_Load;
                //assign ff = (currentState == State_Reset);
            end

        State_Load:
            begin

               if(!Start) nextState = State_Pause;
               else if(!Reset) nextState = State_Reset;
               else nextState = State_Load;
            end
        State_Pause:
            begin
                if (Sw1) nextState = State_P1;
                else if (Sw2) nextState = State_P2;
                else if (Pause) nextState = State_Pause;
                else if (!Start) nextState = State_Pause;
                else if (!Reset) nextState = State_Reset;
                else  nextState = State_Pause;
            end
        State_P1:
            begin
                if (Sw2) nextState = State_P2;
                else if (Sw1) nextState = State_P1;
                else if (Pause) nextState = State_Pause;
                else if (!Start) nextState = State_P1;
                else if (!Reset) nextState = State_Reset;
                else nextState = State_P1;
            end
        State_P2:
            begin
                if (Sw1) nextState = State_P1;
                else if (Sw2) nextState = State_P2;
                else if (Pause) nextState = State_Pause;
                else if (!Start) nextState = State_P2;
                else if (!Reset) nextState = State_Reset;
                else nextState = State_P2;
            end
    endcase
end

endmodule
```

This following code above represents the logic in which our states are able to shift to other states based of user input. This code is based off of our state diagram from earlier, this is implemented using switch case statements. The always@(*) operation is used to solve any incomplete specified lists, this will always update its elements regardless of your sensitivity list.

Each switch case is representing each state, based off what input that was selected will move said current state to the next state. For example in the Pause state, if Sw2 is pressed on, then it will then declare the next state as State_P2 and then have that set as the current state.

Questa Testing of FSM:

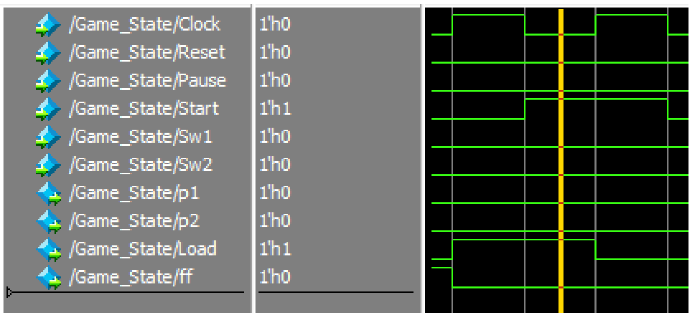
Game State Reset



```
/Game_State/Clock   1'h1
/Game_State/Reset   1'h1
/Game_State/Pause   1'h0
/Game_State/Start   1'h0
/Game_State/Sw1     1'h0
/Game_State/Sw2     1'h0
/Game_State/p1      1'h0
/Game_State/p2      1'h0
/Game_State/Load    1'h0
/Game_State/ff      1'h1
```

As State Reset goes high, and the positive edge of the clock goes high the FF wire will output a high that will then input into the register file in PartB to clear the registers and time. for a new time and game to start.

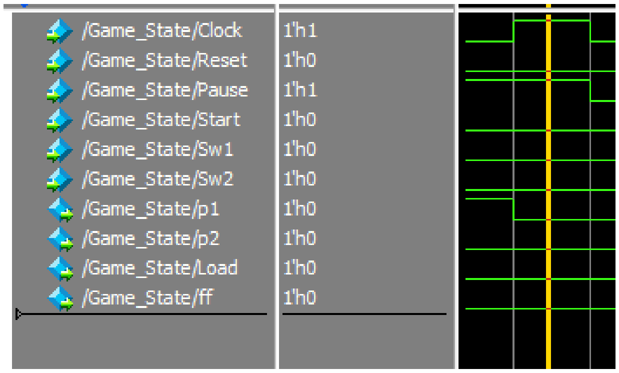
Gamestate Load and Start



```
/Game_State/Clock   1'h0
/Game_State/Reset   1'h0
/Game_State/Pause   1'h0
/Game_State/Start   1'h1
/Game_State/Sw1     1'h0
/Game_State/Sw2     1'h0
/Game_State/p1      1'h0
/Game_State/p2      1'h0
/Game_State/Load    1'h1
/Game_State/ff      1'h0
```

After FF has outputted high and cleared the register files, it is now time to load in new data, i.e. our time inputs, within this state we can allow w0 and w1 to preset those register files from one to seven minutes. Once those times have been selected, you can then select start which will allow the FSM to switch between Pause, Player1, Player 2, or back to Reset. Until you switch on start the timers will be unable to start decrementing.

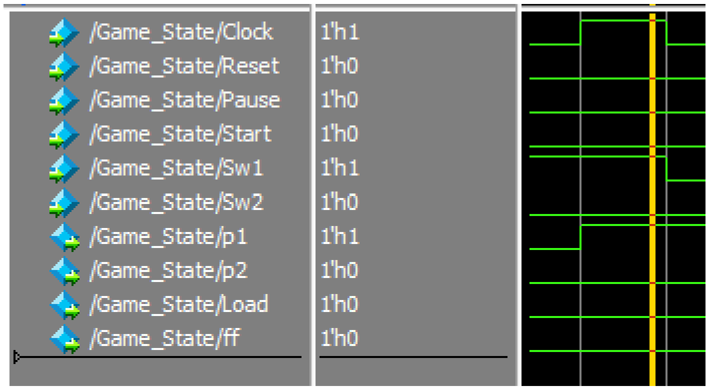
Gamestate Pause



/Game_State/Clock	1'h1
/Game_State/Reset	1'h0
/Game_State/Pause	1'h1
/Game_State/Start	1'h0
/Game_State/Sw1	1'h0
/Game_State/Sw2	1'h0
/Game_State/p1	1'h0
/Game_State/p2	1'h0
/Game_State/Load	1'h0
/Game_State/ff	1'h0

After the load time had been selected and the game has started, the pause state is now available to be selected through our SW3_DB. Essentially removes the enable from either player register file to continue decrementing from the outputs of p1 or p2. This pause state still retains the values for both player 1 and player 2. The only method to exit the pause state is to either enable the reset, or go to player 1 or two.

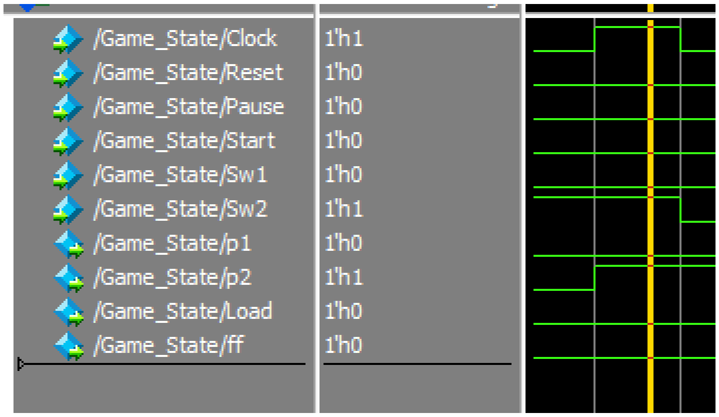
Gamestate Player 1



/Game_State/Clock	1'h1
/Game_State/Reset	1'h0
/Game_State/Pause	1'h0
/Game_State/Start	1'h0
/Game_State/Sw1	1'h1
/Game_State/Sw2	1'h0
/Game_State/p1	1'h1
/Game_State/p2	1'h0
/Game_State/Load	1'h0
/Game_State/ff	1'h0

Once you have selected Sw1 through KEY1_DB, this will output a high signal to the player register 1 into the turn player input. This input acts as an enable where the down counter will activate and the register file.

Gamestate Player 2



/Game_State/Clock	1'h1
/Game_State/Reset	1'h0
/Game_State/Pause	1'h0
/Game_State/Start	1'h0
/Game_State/Sw1	1'h0
/Game_State/Sw2	1'h1
/Game_State/p1	1'h0
/Game_State/p2	1'h1
/Game_State/Load	1'h0
/Game_State/ff	1'h0

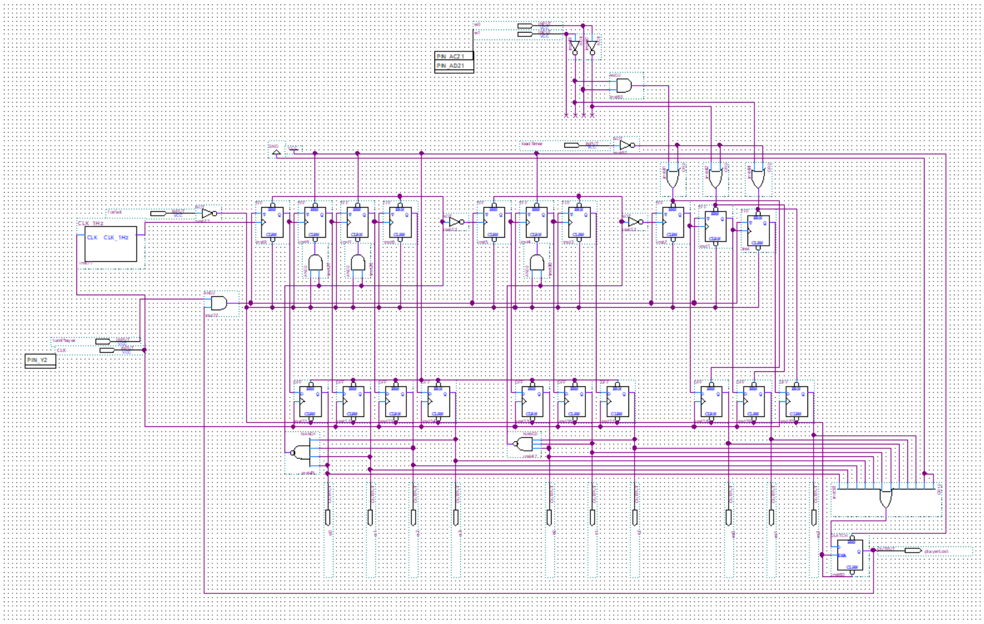
Once you have selected Sw2 through KEY2_DB, this will output a high signal to the player register 2 into the turn player input. This input acts as an enable where the down counter will activate and the register file.

Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

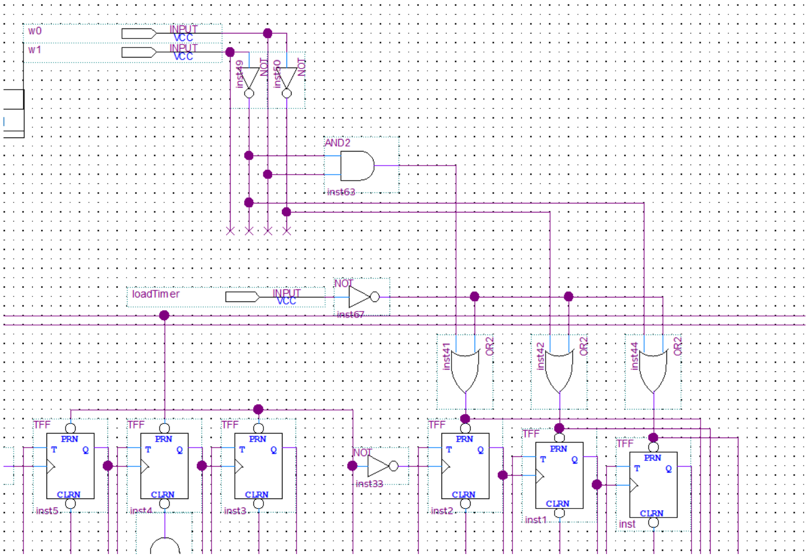
Final Project

Part B: Register and Additional Logic

Player Register



Here is a wide overview of the player_register.bdf circuit. To the left are forfeit, turnPlayer, and clock inputs, as well as a 50MHz to 1Hz converter. Moving inwards, there is a 4-3-3 series of TFF down counters and their respective DFF registers. These are stored in an output register that updates each cycle of the DE2-115's internal clock. The extra circuitry towards the bottom right determines if the player has run out of time and stops the player from counting past 0:00. Finally, the user supplies their desired timer preset time using the circuit shown below.



While in the Load state, determined by the game state FSM, whatever values the user selects are stored in these registers after a clear cycle. This circuit can be seen in the previous image, using an inverted signal from the loadTimer input in an OR configuration with the outputs of the previously described circuit. This acts as a single-bit gate for our input circuit.

Since the PRN is active on a low signal, we have to invert the outputs of our circuit. The table for this circuit is

w1	w0	Dec.	m0	m1	m2	/m0	/m1	/m2
0	0	1	1	0	0	0	1	1
0	1	2	0	1	0	1	0	1
1	0	5	1	0	1	0	1	0
1	1	7	1	1	1	0	0	0

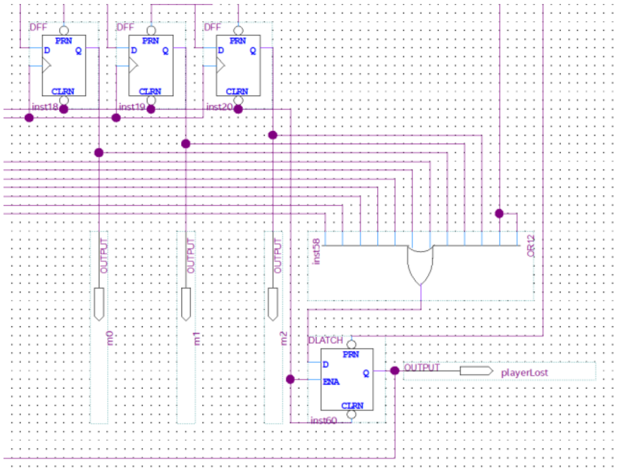
From this table, we can see that /m1 is just the inverse of our input w0, and /m2 is also an inverse, but of w1. Finally, the logic necessary for /m0 is high only when w0 is high and w1 is low. This gives us the following boolean expressions.

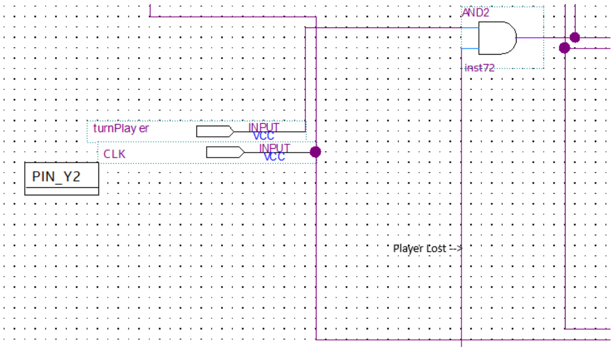
/m0 = w0&/w1

/m1 = /w0

/m2 = /w1

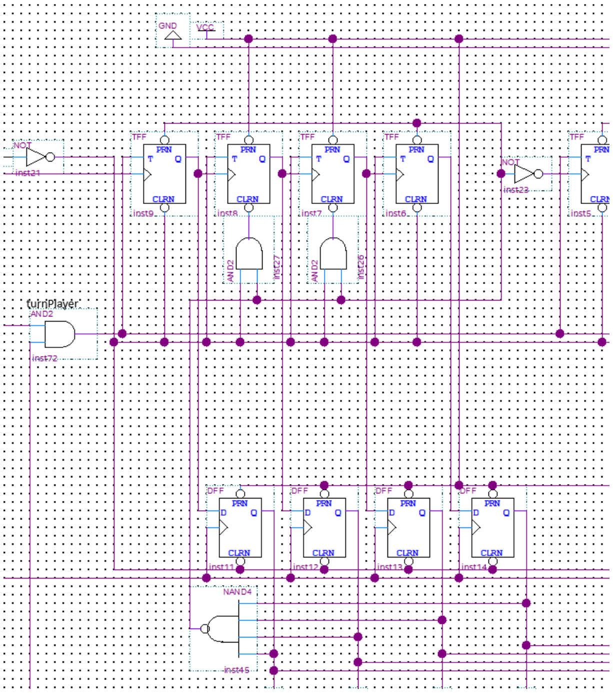
From here, the latch tracking if the player has lost will no longer prevent the circuit from receiving input.





For the state when player time is 0 to be saved in memory, we first run all the register outputs to an OR gate. The 2 inputs on the end are tied to GND so they draw logic low. Finally, when every output of the register is 0, the timer circuit will not be able to count down any further. Also, this latch cannot be cleared until the state of the circuit is returned to reset, where a new time can be loaded.

These enables lead to the inputs of our asynchronous down counter circuits. These down counters also use the current state of the time stored in the output register to reset the counters as soon as they move past 0. In the case of the 4-bit register, this would be when the TFF’s output is 1111, instead the down counter circuit is preset to 1001(9). The same is done for the 3-bit counter, except instead, 101 is preset when the register is 111. Additionally, the condition for which each subsequent counter is triggered by the output of these “rounding” circuits, which all cascade from the ones’ place in the circuit.

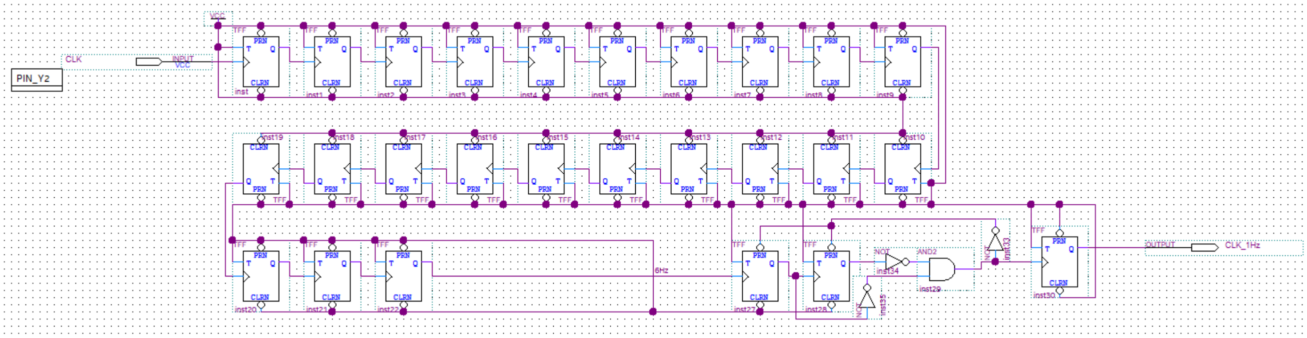


Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project

Timing Circuit

Since we only want to count in 1-second increments and not in nanoseconds for our timer circuit, there is a 1Hz clock that acts as the clock signal for the player_register. This is done by reducing the FPGA’s internal clock of 50MHz down to roughly 6Hz using 23 TFF’s. From here, a 3-1 down counter counts each clock cycle and toggles a TFF at half that frequency. 3 counts of a 6Hz clock becomes a 2Hz clock, which is stored in our final TFF as a 1Hz clock.



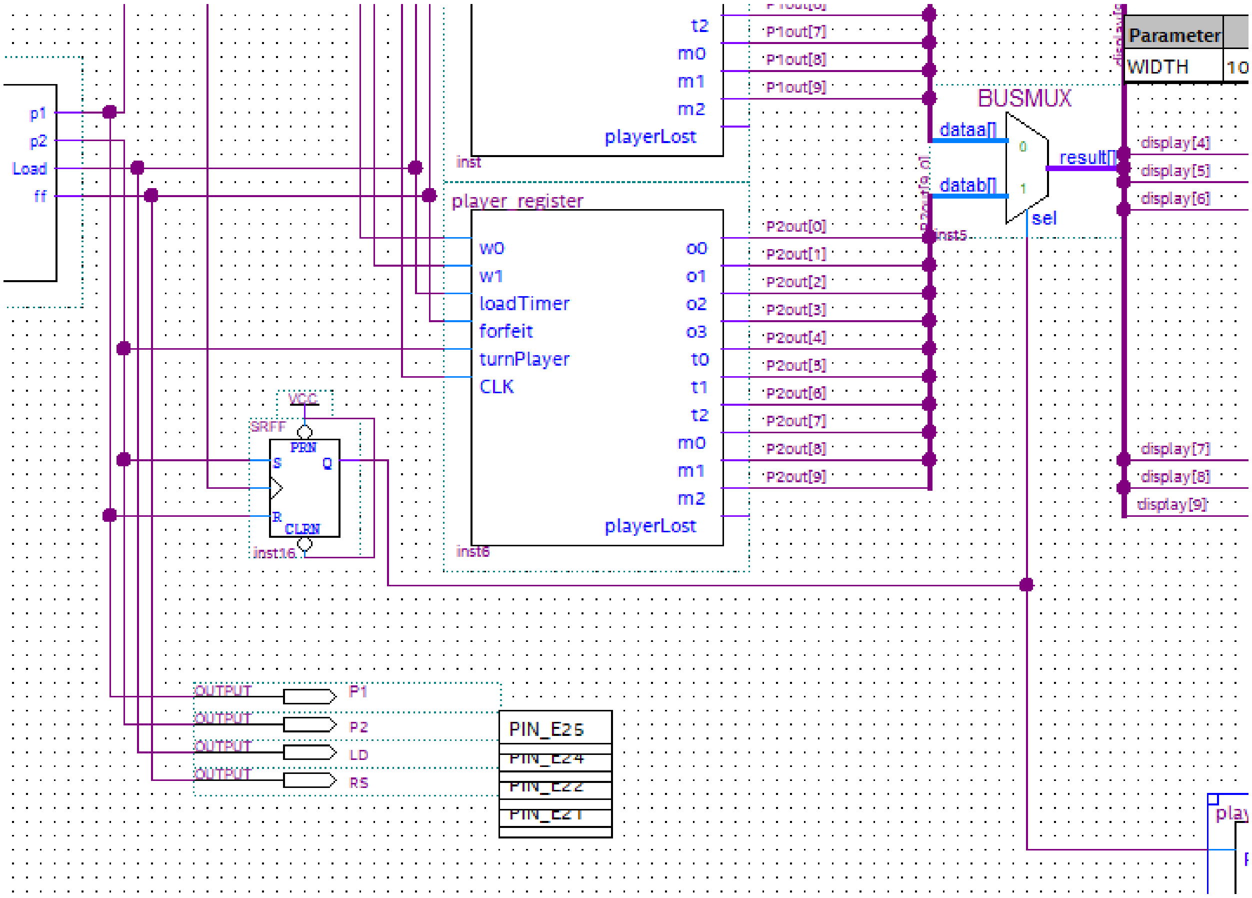
This circuit is only used in the player_register circuits, since all of our current time registers and game_state FSM updates with the FPGA’s internal clock. Since we have nearly no bouncing issues using these two clocks together to control the circuit, we’ve continued to use this arrangement. Testing of other system clock speeds rather made our registers update too slowly, showing the rounding state of our counters for a split second before moving on to the intended value.

Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project

Output Select

The selector for our output is an SR latch saving the state of the most recent player. The output signal from this circuit the selects the output for the BUSMUX controlling which player’s time is displayed and which of the two indicators “P1” or “P2” are showing, specifically during the pause state. Since as we move from either player’s turn to the pause state outputs p1 and p2 both drop to low, we can’t use a typical D latch. In that scenario, the selector for the BUSMUX’s would always be 0. There are othe r more elegant ways solve this particular problem, but this was a quick and simple solution. This picture also shows our state outputs, as indicated by P1, P2, LD, and RS.



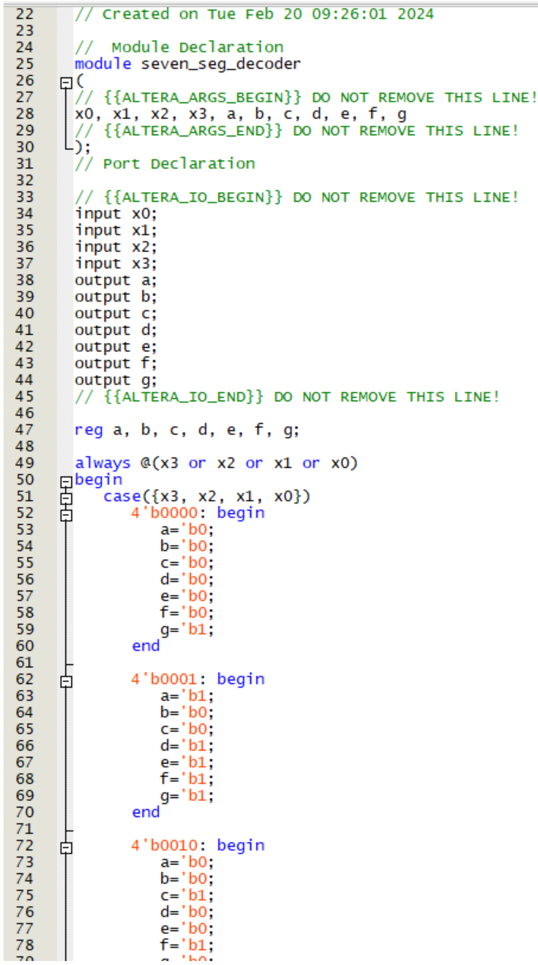
Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project

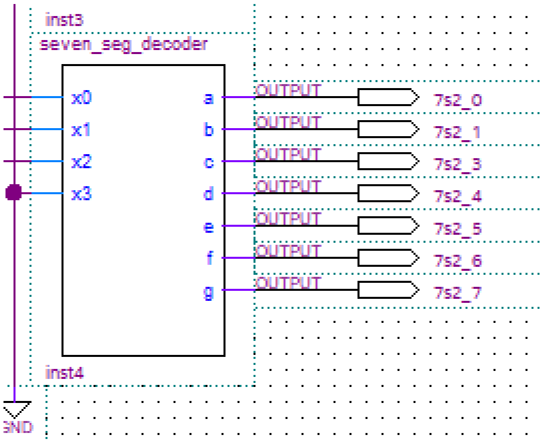
Part C: Output

Output Circuits

Our 4-bit input 7-segment display circuits were used from our previous labs and were coded in Verilog, as shown here.



```
// Created on Tue Feb 20 09:26:01 2024

//  Module Declaration
module seven_seg_decoder
(
// {{ALTERA_ARGS_BEGIN}} DO NOT REMOVE THIS LINE!
x0, x1, x2, x3, a, b, c, d, e, f, g
// {{ALTERA_ARGS_END}} DO NOT REMOVE THIS LINE!
);
// Port Declaration

// {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
input x0;
input x1;
input x2;
input x3;
output a;
output b;
output c;
output d;
output e;
output f;
output g;
// {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

reg a, b, c, d, e, f, g;

always @(x3 or x2 or x1 or x0)
begin
   case({x3, x2, x1, x0})
      4'b0000: begin
         a='b0;
         b='b0;
         c='b0;
         d='b0;
         e='b0;
         f='b0;
         g='b1;
      end

      4'b0001: begin
         a='b1;
         b='b0;
         c='b0;
         d='b1;
         e='b1;
         f='b1;
         g='b1;
      end

      4'b0010: begin
         a='b0;
         b='b0;
         c='b1;
         d='b0;
         e='b0;
         f='b1;
```

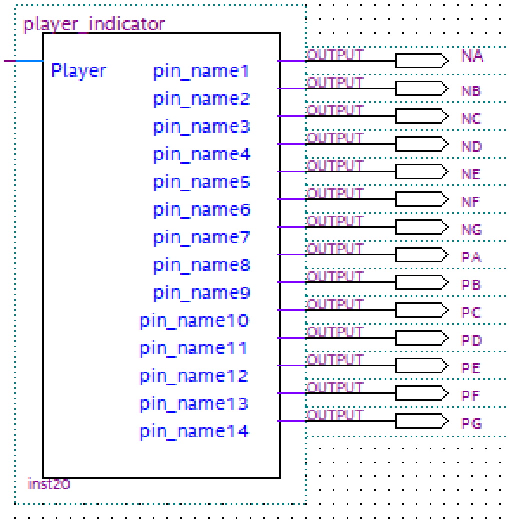


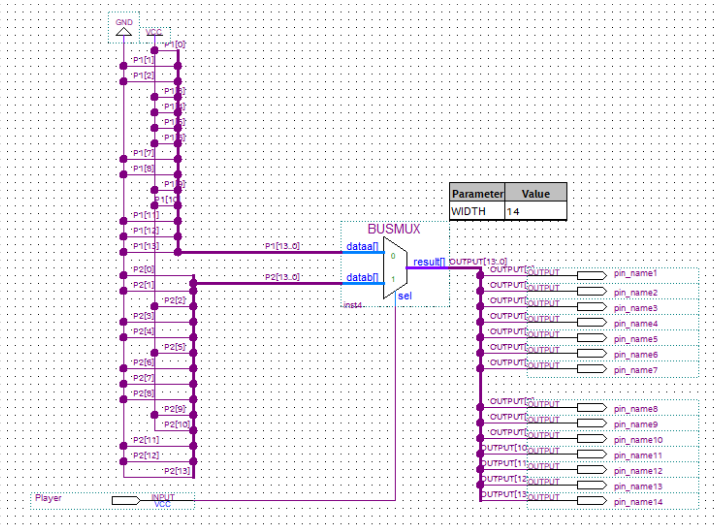
In our code, a case statement is used to check when our input values correspond to some encoded output value. This encoding is determined by the layout of leds on the 7-segment display.

As far as the outputs for each players turn, “P1” and “P2”, the following circuit is used. This circuit uses the VCC and GND components to output the 2 states.

Garrett Trickey and Matthew Smith
CPRE 281
Spring ‘24

Final Project





Parameter	Value
WIDTH	14

Since our outputs for this circuit are so simple, establishing certain values for our two outputs is sufficient. However, had we wanted to distinguish between a PA(pause) state and an LD(load) state, this would have been a more complicated circuit.