ECE 2300 Lab 3 Report

Max McCarthy (msm296)
Raghav Kumar (rk524)
Robert Zhang (rdz26)

Introduction

Purpose:

Finite state machines serve to organize the combinational and sequential logic of complex digital systems. The machine receives one or more inputs and uses those values to establish a state. States values are stored in flip-flops and are used to calculate one or more outputs of the machine.

Moore vs. Mealy:

In a Moore machine, the output is determined only by the current state. In a Mealy machine, the output depends on both the current state and the input. Mealy states can often be used to implement the same system as a Moore machine, but in fewer states.

Problem Summary:

Design a game that allows two players to demonstrate their reaction time. Players will initiate a round, then wait for a signal light. Upon seeing the light, each player must try to press their button before their opponent is able to do the same; the winner gains a point. If a player presses their button before the light turns on, or before it's possible to react from the light turning on, then their opponent gains the point, and a light turns on, indicating that a false start occurred. The first player to reach a certain number of points is the winner.

Tasks to Complete:

1. Implement core game
2. Modify game to include false start functionality
3. Modify game to include multiple round functionality
4. Upload to test board

Tools Used:

1. Quartus II
2. DE0-CV Board

End Result:

The game functions as described in Problem Summary. After either player has reached 2 points, that player's letter is displayed.

Inputs/Outputs

Overall (lab3.v) Inputs:

1. CLK: Clock.
2. RESET: Resets game (active high).
3. PLAYER_A: Signal from Player A's button (active low).
4. PLAYER_B: Signal from Player B's button (active low).
5. NEXT: Signal to start new round (active low).

Overall (lab3.v) Outputs:

1. SIGNAL: Active high bit that tells when the signal light is on.
2. SCORE_A [3:0]: The value of Player A's score, in binary.
3. SCORE_B [3:0]: The value of Player B's score, in binary.
4. WINNER [3:0]: The binary value of the winner of the game. Stored as the binary equivalent of the hex value of either A(1010) or B(1011).
5. STATE [3:0]: Current FSM state; the state number of each state is a 4-bit binary number.
6. FALSE_START: Active high when either player has false started. False start is defined anytime before ten clock cycles after SIGNAL is activated.
7. ADDRESS [2:0]: Output of "address_generator" module. Used to generate random DATA in "prandom" module.
8. DATA [9:0]: Output of "prandom" module. Determines length of countdown to activate SIGNAL.

Address Generator for the Countdown Time Generator (address_generator.v) Inputs:

1. CLK: Clock
2. RESET: Active-high bit that resets the incrementing counter within this module to 0.

Address Generator for the Countdown Time Generator (address_generator.v) Outputs:

1. Q [2:0]: Three-bit output representing the address to be looked up in the countdown time generator module.

8-Address ROM Table (prandom.v) Inputs:

1. ADDRESS [2:0]: Provides the address (binary number) of the number to be read.

8-Address ROM Table (prandom.v) Outputs:

1. DATA [9:0]: Provides the binary representation of the time to be loaded into the countdown timer.

Incrementing Counter (counter.v) Inputs:

1. CLK: Clock.
2. increment: Active-high bit that tells the counter when to increment.
3. RESET: Active-high bit that forces the counter to 0 when high.

Incrementing Counter (counter.v) Outputs:

1. value_out [3:0]: A four-bit output that provides the current value of the incrementing counter

Countdown Timer (countdown.v) Inputs:

1. LOAD: Active-high bit that causes the countdown timer to load the value in DATA when activated.
2. RESET: Active-high bit that causes the countdown timer to reset to 0.
3. CLK: Clock.
4. DATA [9:0]: A binary number that is to be loaded into the countdown timer when LOAD is high.

Countdown Timer (countdown.v) Outputs:

1. DONE: Is high when the countdown timer reaches 0, otherwise is low.

Verilog Constructs & Modules Used

The address generator, 8-address ROM table, countdown timer, and incrementing counters are separate verilog modules, given that they are relatively self-contained elements. These modules are all interfaces with the overall lab3.v module.

Originally, we factored the FSM portion of the circuit out into a separate fsm.v module, that would be used within lab3.v. However, over the course of testing and debugging, we found that this setup both complicated the code, and led to a great deal of bugs and errors. As such, we eventually incorporated the FSM into the overall lab3.v module.

Modifications

For testing purposes, we were instructed to modify our code to accommodate a game that ends after either player gains two points instead of five.

FSM

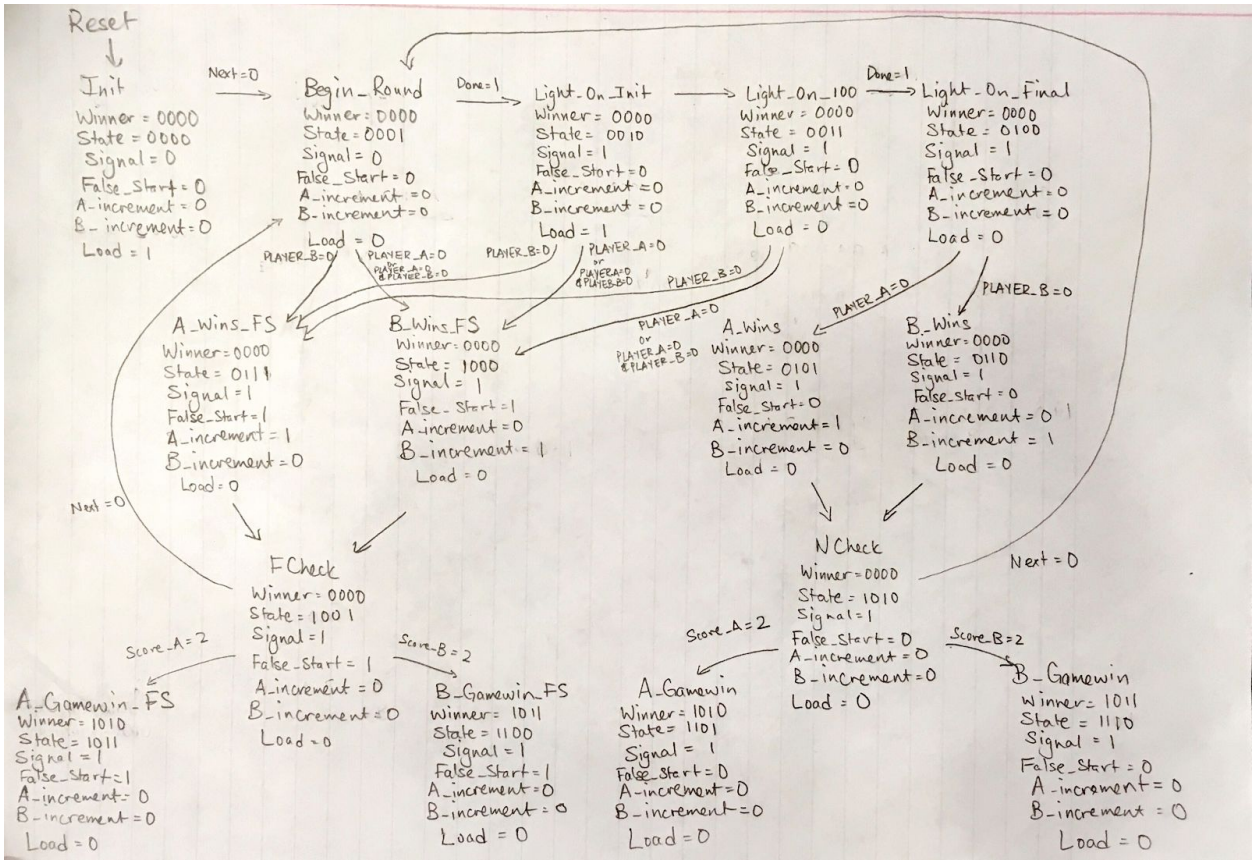
Part A:

Before false start functionality is implemented, the FSM is mostly linear. The game goes through the states and ends with either Player A or Player B wins.

Part B:

After false start and multiple round functionality is implemented, the FSM grows several branches. There is one main linear portion at the beginning, which then branches at different points due to the false start portion. The FSM also cycles through these branches until either player reaches the point cap and is declared the winner.

Part B Diagram:



Circuit Functions

The primary function of the overall circuit is to allow to players to play a game that tests their reaction times. This is done through several subsections of the circuit, each of which serves a discrete purpose.

Incrementing Counter:

Serves as a score counter for the each player. When toggled, the value stored within the counter increases by one. When the reset input is triggered, the counter resets its value to 0. There are two of these in the overall circuit, one for each player. The SCORE_A and SCORE_B outputs each come from the value_out output of an incrementing counter.

Address Generator:

Is a free-running, incrementing counter that increments on each clock cycle, and can also be reset. As such, the address that it outputs through Q depends on the time at which it's called - so it generates what appears to be a random address in the 8-address ROM table.

8-Address ROM Table:

A table containing 8 addresses, each of which point to a different time value (a binary number). The address that is called depends on the value being outputted by the address generator at any moment. These time values can then be loaded into the DATA input of the overall lab module (and in turn into the LOAD input of the countdown timer).

Countdown Timer:

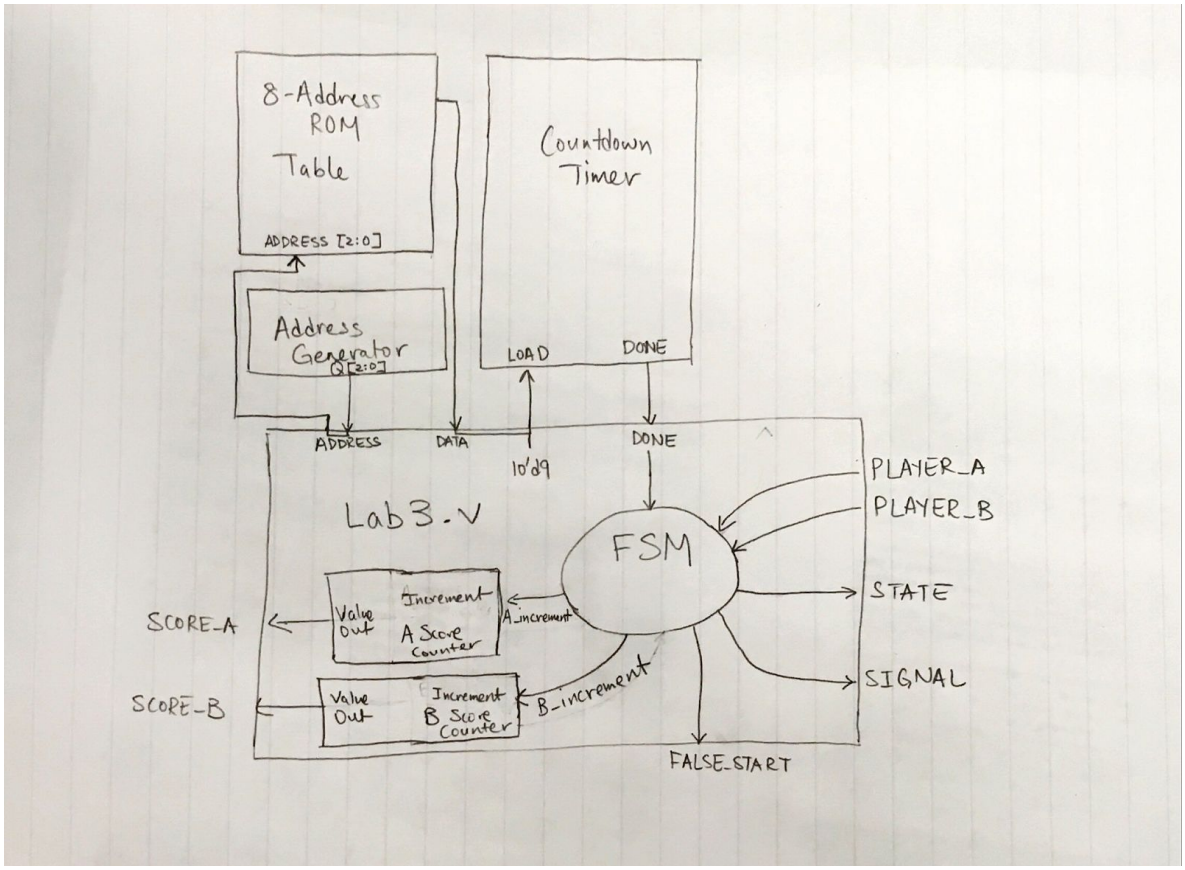
A countdown timer that can be loaded with custom time values. From this value, the timer begins to count down, until it reaches 0, at which point the output DONE becomes high. Used for counting down before the signal light turns on, as well as for counting the 100ms cooldown period.

Overall Module:

Contains the FSM, which ties everything together in this circuit. From the initial state, once NEXT is activated, a new game begins. At any point from now on until the cooldown period has passed from the signal light turning on, if a player presses their button, their opponent will gain a point, and the false-start light will turn on until the next round. If both players false start, player B gains the point. After the cooldown period has passed, the first player to press their button gains a point. If a player gains a point normally, the FSM checks if the player's score is greater than a certain amount designated as the "win" score (for the purposes of this lab, we set it to 2), the player then wins the game. The same occurs if the player wins via a false start, but the false-start light remains on. Each player's score, as well as the FSM state are outputs from this module. In addition, there are 1-bit outputs, SIGNAL and FALSE_START that indicate whether players should press the button, and whether the previous point was due to a false start, respectively.

Path of Data Flow

Diagram of the various modules in the circuit:



The data from the players' end primarily flows from four inputs: PLAYER_A and PLAYER_B, which represent the input buttons of Player A and Player B, respectively, RESET, which resets the entire game, and NEXT, which tells the game to proceed to the next round.

When the game is initialized through RESET being activated, it waits for the players to press the NEXT button, which causes the FSM to move to the next state. Here, it loads a random value into the countdown timer (through the LOAD input), which was generated in the 8-address ROM from the address given by the address generator (based on the time elapsed since it was reset); the address passes through the overall module, through the ADDRESS input, over to the ADDRESS input of the 8-address ROM module.

This value is loaded into the countdown timer (as the LOAD output of the overall module is high), and then the counter is instructed to begin counting down. Right after it's loaded, the FSM transitions into a new state (where LOAD is low), as it waits for the DONE input from the countdown timer to go high, which happens when the countdown timer reaches 0. At that point, LOAD becomes high again, and a new value, equivalent to 9 cycles, is loaded into the Countdown Timer. This is the cooldown period after the signal turns on. At the same time, the signal light turns on.

At any point, when the player presses their respective button, the signal flows into the FSM, which then causes it to change state, depending on what the current state of the FSM is. If the state is between “Init” and “Light_On_Final” (exclusive), then the state transitions to the one where the false-starting player loses. Otherwise, the signal causes the player to win, transitioning the FSM to the appropriate state.

Whenever a player false-starts, the FALSE_START output becomes high until the next round, causing the false start LED to turn on. When a player gains a point through any means, the signal from the FSM to the appropriate incrementing counter becomes high, incrementing the score for that player by 1. The signal then immediately becomes low, to prevent a player’s points from being incremented multiple time. Each incrementing counter has an output that connects to either the SCORE_A output or the SCORE_B output of the overall module, depending on the player that the counter is keeping track of.

Finally, when an overall winner is decided, the FSM will instruct the WINNER output to display the appropriate hexadecimal value corresponding to the player that won (i.e. for Player A, display 1010; for Player B, display 1011).

Implementation & Testing

Combinational Logic:

Several portions of this game made significant use of combinational logic. One of the most important uses was in determining the conditions for state changes and outputs of the FSM. We also used combinational logic to implement our submodules.

Sequential Logic:

The sequential logic played a major role in this game because it’s a game of reaction time. In expanding Part A’s FSM, we added two features of the game that both rely heavily on time. The false start must be sequentially appropriate to keep the game accurate. Also, having multiple rounds made the storage of each player’s score essencial; this could not be done without the use of sequential logic.

Debouncing:

Debouncing is a technique that accounts for output transitions that fluctuate (for instance, when an input goes from 0 to 1, it may bounce around between 0 and 1 for a little bit). It results in the circuit having a cleaner output - and is necessary because having signals that briefly fluctuate between 0 and 1 instead of cleanly changing between values can cause unintentional behaviors in the circuit.

Debugging:

During the course of testing the FSM, we found that always blocks made the outputs particularly problematic. In addition, as stated before, including the FSM in a separate module also created problems that were difficult to trace. When we replaced certain parts of the FSM with assignment statement to wires (particularly the outputs), and put the FSM inside of the overall module, the bugs disappeared.

Additional Comments:

The use of an FSM as an organization tool made implementing the game much easier than it would have otherwise been. It also allowed us to more clearly identify where we ran into issues throughout the process.

Conclusion

To conclude our game worked according to the specifications in the lab handout. We learnt a great deal about verilog programming and FSM logic from this lab. We also got a lot of experience with debugging as the final lab did not complete in time and we had to go for multiple Office Hours to get the lab working.

Report Distribution

Max: Report Template and Editing, Introduction, Modifications, Inputs/Outputs, FSM, Implementation & Testing

Raghav: Conclusion, Debugging of actual lab

Robert: Inputs/Outputs, Circuit Functions, Modules Used, Path of Data Flow, Verilog Constructs & Modules Used