Lab 3: State Machines

THE FLOW OF STATES IN SEQUENCE DETECTION

March 7, 2018

Richard Carter
Clemson University
Department of Electrical and Computer Engineering
rcarte4@clemson.edu

1 Abstract

State machines are incorporated into many of the digital electronics around us. For this project, we attempted to implement a Moore state machine in order to create a digital ”lock” system. The design of our state machine followed a traditional design, with the exception of output logic due to our output merely being the encoded representation of our machine’s current state. Our testbenches showed that the functionality of our state machine was working as intended, and the DE1 ported hardware worked as planned.

2 Introduction

The objective of this project was to construct a Moore state machine that acts as a sequence detector for a set of four different 5-bit sequences. One of these sequences would put the machine into an "UNLOCKED" state, while the other three would put it into a "FALSE UNLOCKED" state. All other sequences would keep the device in a "LOCKED" state. The output of the state machine would simply be the encoded binary representation of the current state of the state machine, and the input to this machine would be the sequence bit, which would be read in on each clock cycle.

Additionally, an objective of this project was to port the functionality of our Moore state machine to the DE1 Altera board. The clock used for advancing the state machine would be emulated by a push-button input. The input would be determined by a single switch on the board, and the machine's output would be shown using 4 of the board's red LEDs.

3 DESIGN

3.1 OVERALL DESIGN

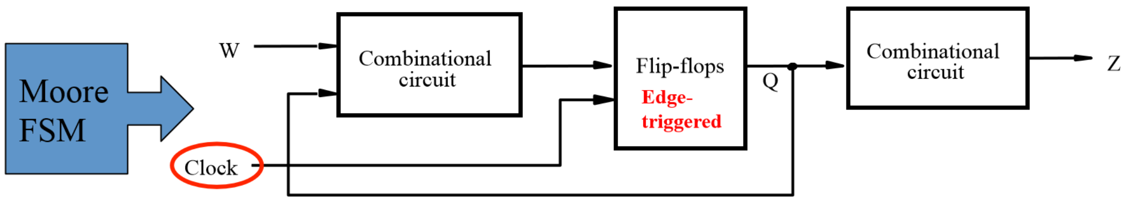

Figure 1: Moore State Machine Diagram

Our design started out based on the model of a Moore state machine presented in lecture, which is shown in Figure 1. However, we soon figured out that the combinational circuit used to determine the output from the data stored in the flip-flop was not needed, as the required output would just be the data stored in the flip-flop (which is the current state of the state machine). Therefore, only the first two blocks are used.

3.2 STATE LOGIC

The combinational logic used to determine the next state of the state machine is based off of the truth table that we determined for the system, shown in Table 1. State `R` and states `L1` through `L5` are all "LOCKED" states, with the `ERR` state used for error catching during testing. All locked states will revert back to the `R` (reset) state if a sequence that cannot lead to a "FALSE UNLOCK" or "UNLOCK" state is detected. The three pre-defined false unlock sequences will all lead to the `FU` state, and the single unlock sequence will lead to the `U` state. Both the `U` and `FU` states will change to the `R` state after one clock cycle.

Table 1: Truth Table for Lock/Unlock State Machine

CURRENT	NEXT		OUTPUT
0000 (R)	1(L1)	0(R)	0000
0001 (L1)	1(R)	0(L2)	0001
0011 (L2)	1(L3)	0(R)	0011
0010 (L3)	1(L5)	0(L4)	0010
0110 (L4)	1(FU)	0(FU)	0110
0111 (L5)	1(FU)	0(U)	0111
0101 (ERR)	1(ERR)	0(ERR)	0101
0100 (FU)	1(R)	0(R)	0100
1100 (U)	1(R)	0(R)	1100

In VHDL, we accomplished this in an architecture through a when/else assignment [1]. Each conditional is representitive of a potential next state given both the incoming input and the current state. A portion of the code for this is shown in Listing 1.

```
next_state <= L1 when (seq_bit = '1' and prev_state = R)
   else
         R when (seq_bit = '0' and prev_state = R) else
         R when (seq_bit = '1' and prev_state = L1) else
         L2 when (seq_bit = '0' and prev_state =L1) else
         L3 when (seq_bit = '1' and prev_state = L2) else
         ...
```

Listing 1: Next State Code

3.3 STORAGE

For storing the current state, we created the following process shown in Listing 2 that will act as our flip-flop. We created a signal with a default value in order to give the state machine a starting state (which is `R`).

```
architecture behav of state_reg is
         signal current_state : std_logic_vector(3 downto 0)
            := "0000" ;
begin
         state_out <= current_state;

         data_sync : process( clock )
```

```
                begin
                        if( rising_edge(clock) ) then
                                current_state <= state_in;
                        end if ;
                end process ; -- data_sync
end architecture ; -- arch
```

Listing 2: "Flip Flop" Code

4 TESTING

In order to confirm full functionality of our state machine, a test bench was created that would simulate a runthrough of all inputs and state changes. These tests can be grouped into three sections, which will be explained in detail in the following sections.

4.1 LOCKED STATES

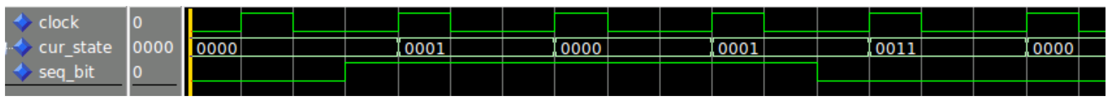

Figure 2: Timing Diagram of Locked States Testing

A pre-defined functionality of our state machine is having incorrect sequences (not FALSE UNLOCKED or UNLOCKED sequences) loop back into the first locked state as soon as it can be determined that the sequence will not lead to `FU` or `U` states. Using the truth table shown in Figure 1 on page 2, we determined that the inputs `0, 11,` and `100` were the only three input sequences that would cause a state change back to the first locked state (`R`). Figure 2 shows our confirmation that our state machine is functioning properly regarding "locked" states.

4.2 FALSE UNLOCKED STATES

We also tested to make sure all `FU` states could be reached by appropriate sequences. Figure 3 shows our confirmation that the `10111` sequence results in the `FU` state and then changes to the `R` state after a clock cycle. Figures 4 and 5 show the same, but for sequences `10100` and `10101` respectively.

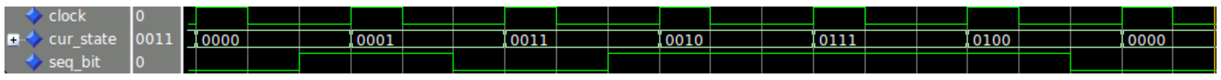

Figure 3: FU State Testing for `10111`

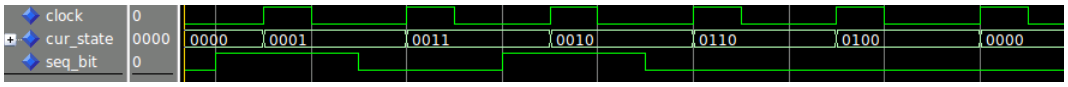

Figure 4: FU State Testing for `10100`

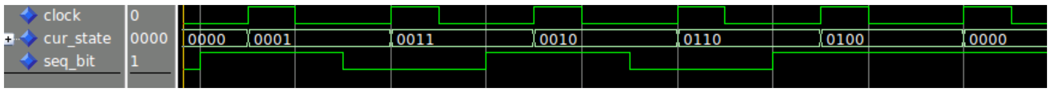

Figure 5: FU State Testing for `10101`

4.3 UNLOCKED STATE

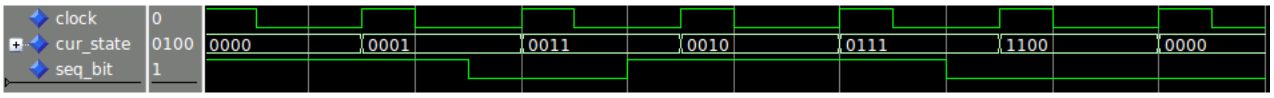

Figure 6: Unlock State Test

Lastly, the unlock sequence was tested to make sure the `U` state is achieved properly, as well as being set to `R` after a clock cycle. Figure 6 shows our confirmation that this functions as planned.

5 Conclusion

The creation of our state machine lock system was a complete success. After creating a wrapper VHDL file for porting purposes, we tested our hardware physically on the DE1 board and it worked as planned. If there were any lessons to be learned from this project, it would be the value of running test benches before porting to physical hardware. Due to not having an initial state stored in our flip-flop, there may have been undefined behavior in our state machine had we just ported it from the get-go.

References

[1] Sigasi. Signal Assignments in VHDL: with/select, when/else and case, 2011.