ECSE 425 Report

ECSE 425: Computer Architecture
Group Number: 23
Group Members: Karl Doumar, Edward Latulipe-Kang
Student ID: 260733160, 260746475
Professor: Amin Emad
February 1, 2021

ECSE 425 Project#1: Finite State Machines

Introduction

This project illustrates the usefulness of Finite State Machines by showing that one can utilize simple ports and std_logic in order to achieve certain tasks such as identifying commented characters, through the use of clock cycles.

FSM Design and Architecture

To properly explain our design, we must first begin by understanding the idea behind the process and how the separate parts of the FSM are interpreted within each clock cycle. Please note that the state diagram is presented at the end of this report. For the design choice implemented, it is important to also distinguish between comment blocks and comment lines, as well as the difference between confirming a commented character and having no commented characters present. To that end, the design of our FSM requires 5 different states: “NoComment”, “Confirming”, “CommentBlock”, “Commentline”, and “Terminating”. These are shown in Figure 1. ModelSim begins setting up the process by verifying if there is a rising edge, since our design only checks the ACSII character at each uptick of the clock cycle. Once entered inside the process, we check if reset was toggled or not. If reset is not toggled, the finite state machine then operates as follows. Initially, the design will start in the “NoComment'' state. should the design detect the beginning of a comment via the “/” symbol it will transition into the “Confirming” state. This state is used to determine whether the incoming string of signals is a comment block or a simple line comment based on the next input signal. Indeed, should the next signal be another “*” then we transition into the “CommentBlock” state. If the next signal is “/” then we transition into the “CommentLine” state. Finally, if the next signal corresponds to none of these symbols, then we return to the “NoComment” state to indicate that the string of signals is not actually a comment. Additionally, The “CommentBlock” and “CommentLine” states are abandoned only if the appropriate signal is received. Both states employ a self-contained loop so as to dismiss any other signal as a comment. In particular, the “CommentBlock” is complemented by the “Terminating” state. This is useful, because it allows us to check if the secondary signal is also a “/”. If true, this will terminate the comment. Otherwise, the state reverts back to “CommentBlock”. “CommentLine” only searches for the new line character “\n”. Once this is obtained, the comment is also terminated. Finally, “CommentBlock” and “CommentLine” operate independently and can never interact. The intermediate states act as a buffer for what portion of a comment can be considered a comment.

```
if rising_edge(clk) then
        if reset = '1' then
                state <= NoComment;
        else
           case state is
                when NoComment =>
                        if input = SLASH_CHARACTER then
                                state <= Confirming;
                        else
                                state <= NoComment;
                        end if;
                when Confirming =>
                        if input = STAR_CHARACTER then
                                state <= CommentBlock;
                        elsif input = SLASH_CHARACTER then
                                state <= CommentLine;
                        else
                                state <= NoComment;
                        end if;
                when CommentBlock =>
                        if input = STAR_CHARACTER then
                                state <= Terminating;
                        else
                                state <= CommentBlock;
                        end if;
                when CommentLine =>
                        if input = NEW_LINE_CHARACTER then
                                state <= NoComment;
                        else
                                state <= CommentLine;
                        end if;
                when Terminating =>
                        if input = SLASH_CHARACTER then
                                state <= NoComment;
                        else
                                state <= CommentBlock;
                        end if;
          end case;
         end if;
end if;
```

Figure 1. FSM VHDL

Testing and Discussion

During the testing process of our design we found a few flaws in its implementation. Despite the results being fairly close to what we wanted, it wasn't exactly what we needed. For instance, when trying to determine what was a comment or not, our implementation considered the secondary toggling comment character as part of the comment, when at the beginning of the comment, or as part of the non-comment, when at the end of the comment. This implementation was not clean and was more of an approximate solution. To fix this, we implemented the output signal assignment as a process such that it needed to be dependent upon the change in input signal. This can be seen in Figure 2. That is to say, the output can only change the instant the input does. This required a new process block and prevented the overlapping issue we've had.

```
-- This process is used to update what's a co
process(input)
    begin
        if    state = NoComment then
            output <= '0';
        elsif state = Confirming then
            output <= '0';
        elsif state = CommentBlock then
            output <= '1';
        elsif state = CommentLine then
            output <= '1';
        elsif state = Terminating then
            output <= '1';
        else
            output <= '0';
        end if;
    end process;
```

Figure 2. Output Assignment

Furthermore, to test the actual functionality of our implementation we have created a few test scenarios. These can be seen in Figure 3. We've noticed early on that we did not need to test all cases, as a lot of the cases either overlap or can be absorbed into one another. Therefore, we have written tests that vary the usage of "/", "*", and "\n" to see if our design can accurately detect comments from non comments.

Additionally, we have written assertions at various points of each test to ensure that the desired output is obtained at each step or point of interest.

```
# ** Note: Example 1: /*ASCII\n*/
#    Time: 0 ps  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 6 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 2: /*ASCII*ASCII/*/
#    Time: 6 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 14 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 3: /*ASCII*//
#    Time: 14 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 20 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 4: ///*ASCII*/\n
#    Time: 20 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 28 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 5: //ASCII*/\n
#    Time: 28 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 34 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 1: /*ASCII\n*/
#    Time: 34 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 40 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 2: /*ASCII*ASCII/*/
#    Time: 40 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: _____
#    Time: 48 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Example 3: /*ASCII*//
#    Time: 48 ns  Iteration: 0  Instance: /fsm_tb
```

Figure 3. Test Cases

As per Figure 3, we have no assertion errors. We can further compare the test examples to the corresponding timing diagram in Figure 4 to establish their correctness.

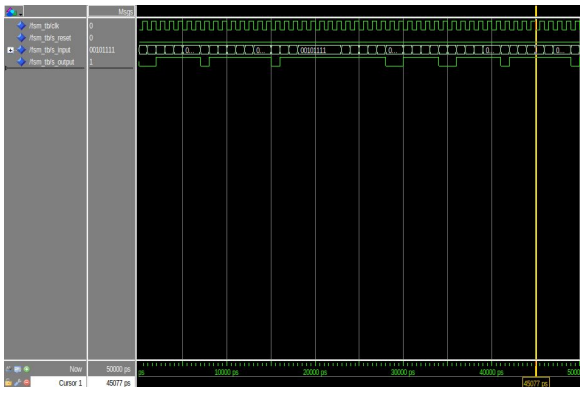

Figure 4. Timing Diagram

Note that ASCII here simply refers to some placeholder that is not a comment toggle. This

could be a string of letters and numbers. We opted to use one constant ASCII symbol of “0000000” as the effects would be identical to having an actual word. More importantly however, using the constant allows us more time for additional tests during the 50 ns execution. Finally, Through the various tests, we have shown that our FSM is, in fact, robust enough to handle edge cases consisting of combinations of “/”, “\n”, and “*” characters. The signals generated from running the fsm also attest to the proper functioning of our implementation.

Please continue to the next page for the state diagram.

State Diagram

the FSM discussed in the previous section is shown. The image is too dense to be seen clearly while maintaining the IEEE format. We have decided to increase its size for a more convenient analysis.

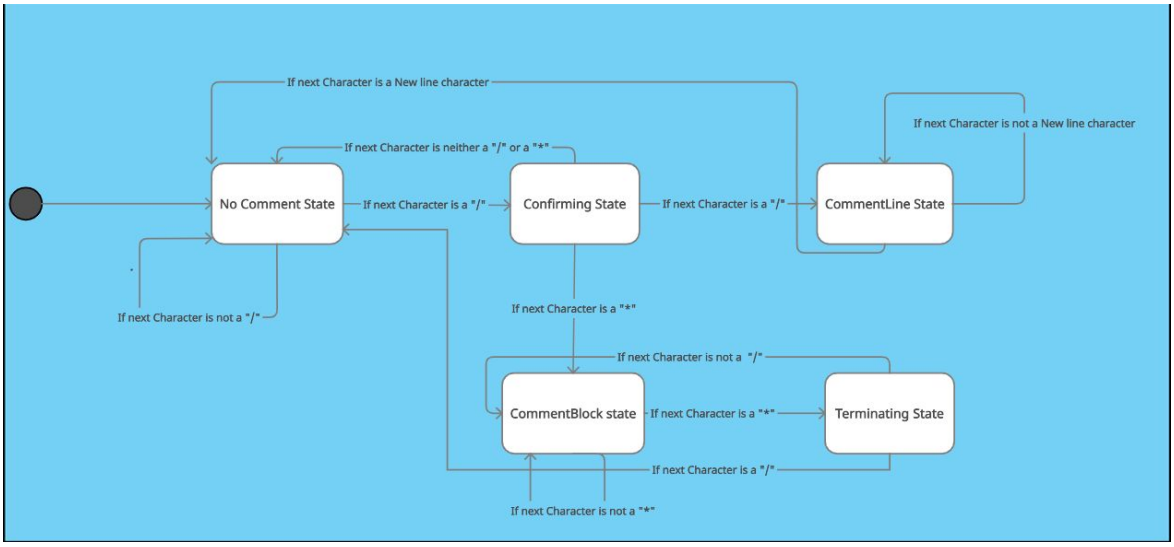

Figure 5. State Diagram