ECSE 425 Project Report 1

Edem Nuviadenu, 260779440
Nathanael Lemma, 260779759
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec

I. Introduction

In this deliverable, we built and tested a finite-state machine to identify the commented characters in a C code. The finite-state machine was implemented in VHDL and has the following ports: clk, reset, input and output. The clk signal is used for synchronization purpose where as reset is used for initialization and setting to an initial value upon reset. The input holds an 8bit ASCII code corresponding to a character. Finally, the output determined is a 0/1 depending on the input.

II. FSM

A. State Diagram

In identifying the necessary states required for this FSM, one key element was ensuring that each state was as mutually exclusive as possible. As such, there should exist a path to different states given the possible commenting scenarios where SLASH_CHARACTER, STAR_CHARACTER, NEWLINE_CHARACTER are involved. In order to detect a comment we decided to place the output signal value on the transition between states as shown in Fig 1, hence implementing this mainly from a Next State Logic approach. Overall we found 5 possible states to complete our comment detection logic shown below. Note that state changes back to S0 upon active high reset input irrespective of current state. This can be seen in the code implementation.

B. Code Implementation

The state machine was implemented in VHDL by first identifying the port map of the structure, which involved clk, reset, output (as std logic) and input (as std logic vector). This input port holds the 8 bit ASCII values that will be read and compared in the state machine. In order to identify comments in C code, we enumerate 5 states (S0 - S4). This implementation allows us to detect comments in not only compilable C code but also in edge cases where the C code isn't compilable.

1) Process Block: The key logic of our FSM is written in one process block, where sequential statements can be seen. Our sensitivity list consists of the clk and reset, to execute the logic in the process block based on changes in either value. Upon the rising edge of a clock we perform our FSM logic which includes a synchronous reset on active high, otherwise we check the current state and proceed to output the necessary '0'(is not a comment) or '1' (is comment) values on the transition of the NSL (Next State Logic). A sample of state 0 is shown in Fig 2 on page 2. This ends our process, taking into account all the possible commenting scenarios.

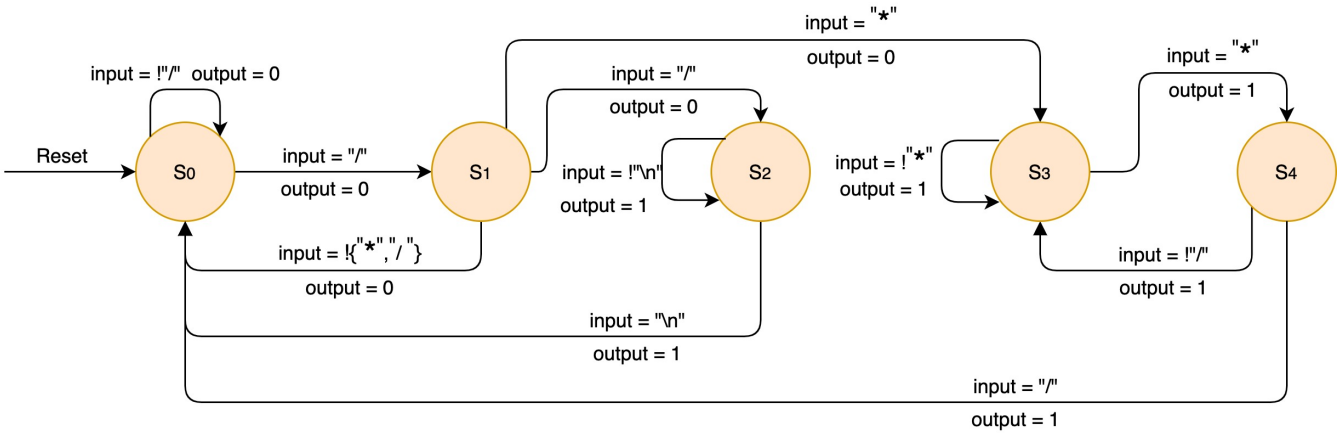

Fig. 1. State Diagram

```
21  type t_state is (S0,S1,S2,S3,S4); --enumerate state
22  signal state : t_state;
23
24  begin
25
26  -- Insert your processes here
27  process (clk, reset)
28
29  begin
30    if (rising_edge(clk)) then
31      if(reset = '1') then
32         output <= '0';
33         state <= S0;
34      else
35      case state is
36         when S0 =>
37            if(input = SLASH_CHARACTER) then
38               output <='0';
39               state <= S1;
40            else
41               output <= '0';
42               state <= S0;
43            end if;
44
```

Fig. 2. Snippet of State Logic

```
# ** Note: Example case, reading a meaningless character
#    Time: 0 ps  Iteration: 0  Instance: /fsm_tb
# ** Note: ____________________
#    Time: 1 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Test 1: //basic\n
#    Time: 3 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: ____________________
#    Time: 11 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Test 2: /*basic\n
#    Time: 13 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: ____________________
#    Time: 21 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Test 3: /*code\ncode*/
#    Time: 23 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: ____________________
#    Time: 36 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Test 4: *//*code
#    Time: 36 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: ____________________
#    Time: 44 ns  Iteration: 0  Instance: /fsm_tb
# ** Note: Test 5: /a//code
#    Time: 46 ns  Iteration: 0  Instance: /fsm_tb
```

Fig. 4. Command line Results

2) RTL View: The RTL image shown in Fig 3 was taken upon compilation of our results. This shows how our FSM logic translated to the necessary logic circuits ie the internal structure of our comment detection system. Our hardware design utilizes primarily 2 registers, a multiplexer and an or gate. The first register (colored in yellow) controls our state logic and the other stores the necessary output based on our state logic.

III. Results

A. Test Bench

In our test we made sure to cover as much cases as possible. We grouped our test cases in to two: *basic cases* and *special cases* shown in the table on the right. In total this resulted in five main scenarios inside our TestBench code implementation. Basic cases are where the sequence of characters ”//” and ”/*” appear in the beginning of the line and are covered in Test 1 - 3. Where as, special cases are were the character sequence ’//” and ”/*” appear in the middle of a line. We assumed the C code provided needing not to be compilable thus we considered any characters after character sequence ’//” and ”/*” to be considered comments in this special case. These scenarios are covered in Test 4 and 5. Upon writing our test bench code to cover these basic and and special cases, the code (fsm.vhd) and testbench (fsm_tb.vhd) files were compiled on ModelSim by running the command *souce fsmt_tb.tcl*. The command line results of this compilation are shown in Fig 4. above and the resulting wave form simulated shown in Fig 5 on page 3.

Basic	”//code\n”	0011111
	”/*code\n”	0011111
	”/*code\ncode*/”	0011111111111
Special	”*//*code”	00011111
	”/a//code”	00001111

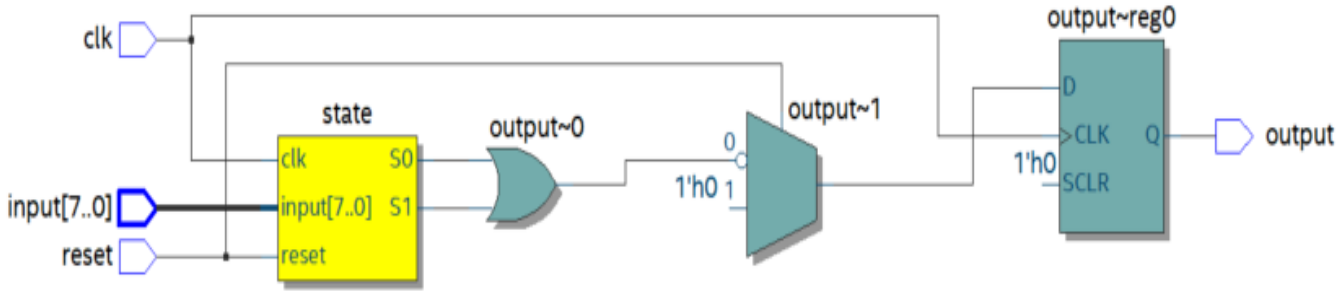

Fig. 3. RTL View

B. Wave Form Results

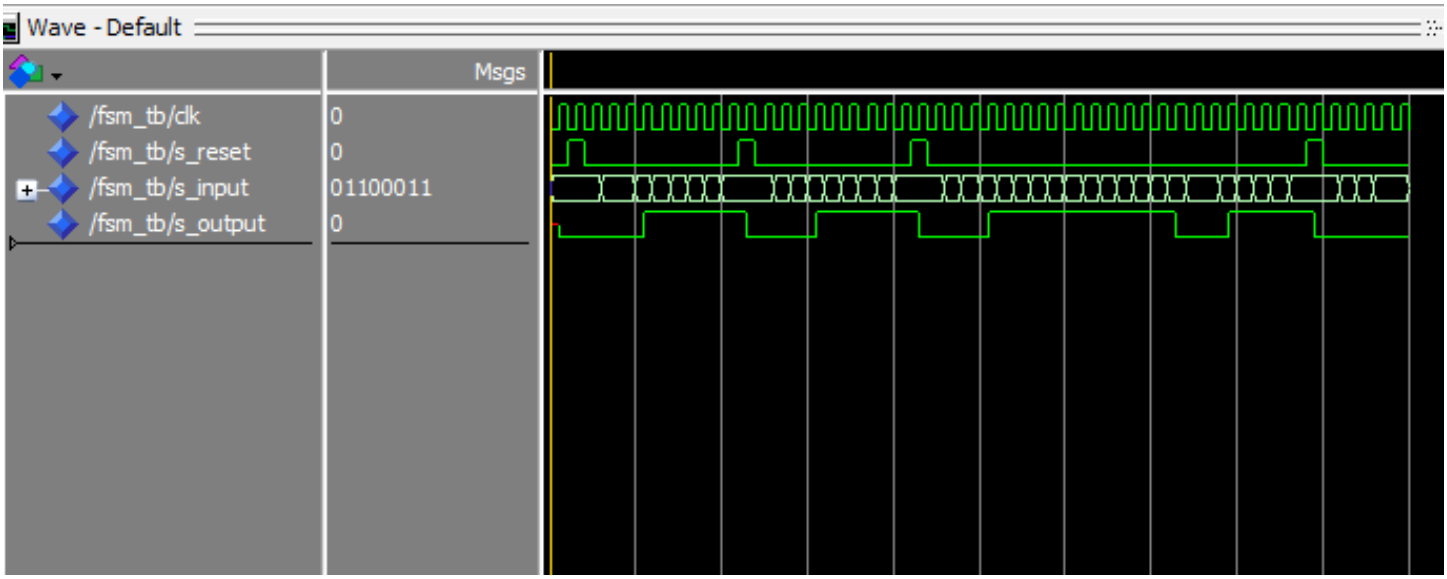

Fig. 5. Wave simulation