I. Introduction

FSMs, otherwise known as Finite State Machines, have been used for decades to describe the behavior of machine behavior. These powerful machines capture the behavior of systems from traffic lights to robots to elevators, using a network of transitions, states, and outputs. Often these machines combine other circuit elements such as registers or barrel shifts, to achieve more complex actions. In the case of the Robot FSM constructed in DP4, the FSM employed the use of, registers, shifting techniques, and a working knowledge of the interaction between states to create a robot that could maneuver through a pipe path, perform maintenance within sections of the pipe, as well as report the current location (relative to map).

II. Method

Prior to the robot, a diagram—otherwise known as a state diagram—was constructed to illustrate the transitions, outputs, and the corresponding action(s) of each state. To create the state diagram, the initial step was to set the states. For example, to move the robot, the robot would require a “move” state where it would prompt the robot to move in the direction where there would be no obstacles. This umbrella concept of moving the robot involved several states: a state to check if the robot should turn left, a state to compute the left turn, respectively, two states for turning right, and a state for no turns. The amalgamation of states provided the necessary judgments to move through the pipes correctly and safely. Much of the planning for the state diagram was initialized by broad concepts of the desired behavior then narrowed into (often several) states where this behavior was better defined.

An important feature of state diagrams was the demonstration of the relationship between states. A transition, or the inputs, into each state determined the next state. Using the previously mentioned move state example, to determine whether it was a right turn, left turn, or no turn, several input combinations had to be considered.

To determine where there was a wall, an input named “wll” was described to be a two-bit vector, with the orientation of [left, right, front]. When a bit was zero, it indicated that there was no wall in that direction. For instance, the combination [110] denoted that there were walls to the left and right, but no wall in front-- in respect to the robot. Wll was the transition condition placed between the idle state and the state that corresponded with a specific wll combination. Figure 1, seen below, illustrates the relationship between states and transitions. Using the state diagram as a guide, implementing the FSM in Verilog became a straightforward task.

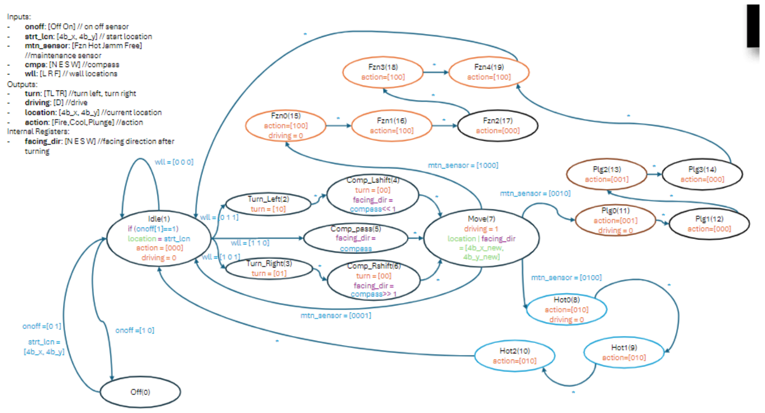

Figure 1. The state machine based on the characteristics of the robot. Transitions are denoted by the blue arrows and the value attached to the arrow. States are the ovals. Outputs are represented by the value inside each state.

As mentioned earlier the state diagram outlined the inputs and transitions of the FSM, therefore instantiating the inputs and outputs became a process finding and listing these variables. Similarly, creating parameters, which can be thought of as states, also was simply listing the states from the FSM. However, there were a few key features not noted in the state diagram like the clock input and registers that need to be included in the Verilog code. With this, the prerequisites to establishing a functional FSM were complete.

FSMs follow a set structure: an always block reliant on a clock input along with other asynchronous signals (signals independent to the clock) and another block triggered by changes in the current state. The first always block would set the next state while the second set the outputs. Together they allow an FSM to sequentially set states and push outputs. From the state diagram, following the path of arrows, the first always block can be implemented, on the other hand, the second always block focuses on the values listed inside of each state (the outputs). Thus, listing the eternal values would produce the second always block.

It was in the second block where implementation can become difficult depending on the required techniques. In the case of the Robot FSM, the compass input, “cmps”, required shifting bits to create a continuous boundary effect.

Concluding the main module, to verify that the behavior of the coded FSM matched the desired behavior, a testbench was created. Here, the main module was instantiated, and various combinations of inputs are tested to guide the debugging process. The Robot FSM testbench followed the outline illustrated by the pipe maze—toggling maintenance signals, indicating wall locations, and providing compass directions in relation to the given map.

III. Result

Following the steps listed prior, the Robot FSM was able to mimic the desired behavior. The Robot was intended to consistently update the location register to the grid coordinates of the map. Since the location register was 8 bits wide (4 bits to represent the x coordinates and 4 bits for the y) by adding or subtracting 16 or one, the x and y coordinates could increment/ decrement independent of each other as seen in Figure 2 below.

In addition to location updates, the Robot had to execute certain actions based on the required maintenance. For example, brown maintenance required the robot to perform toggle between plunging and waiting for four clock cycles, then heat for one more cycle before returning to idle. The other sequences are depicted in Figure 1 and are the states following the "move" state. The correct sequence of actions can be seen in the waveform based on the testbench created above.

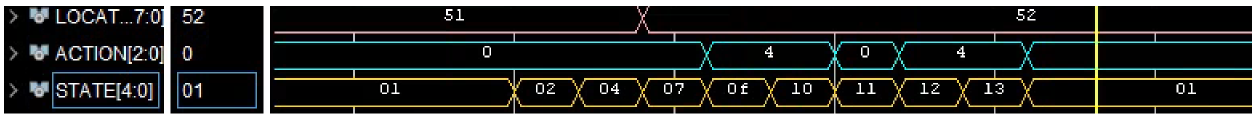

Figure 2. The location, action, and state waveforms produced by the instructor provided testbench. Location outputs structured in the form x-coordinate y-coordinate (e.g. x-coordinate = 5, y-coordinate = 1). Actions are dependent on state, when state was 0f, it triggers the blue maintenance sequence.

IV. Discussion

The steps used to create the Robot FSM can be implemented for many others. Using the same overall structure FSMs can become simple straightforward models to achieve simple behaviors. Nonetheless, it is important to note that while it is an effective model, for larger scale systems-- ones that include many states-- implementing an FSM can be tedious. While coding the Robot FSM, it was found that much of the code was dedicated to assigning the next state. With a system that needed hundreds of states, this task of assigning the next state for every state would become arduous. This is not to say that FSM are not important, in fact FSM are the backbone to many common use appliances. This, however, is a question of are there more efficient systems for larger projects with numerous states.

V. Conclusion

FSM are powerful tools used to create systems that can model a specified behavior using states, transitions, and outputs. Progression from state to state is dependent on transitions, otherwise known as inputs, and each state can have a specified output. When coding these models, they are created using two always blocks, one is triggered on the clock signal as well as any asynchronous signals and the other is triggered by changes in the current state. Together these blocks create the desired behavior.