Lab Five: State Machines

Ben Smith

***Abstract*—Two input logic gates are synthesized for the Altera Cyclone IV FPGA using the Quartus IDE. The logic gates are verified using a System Verilog testbench and Mentor's Modelsim HDL simulator**

I. Introduction

IN this lab state machines will be explored. We will walk through a number of examples from Altera to investigate the different types of state machines available to the digitial designer and why you might choose a particular one.

1) The Case Statement: The "Enumeration" of case statements is the first unique feature of System Verilog that we will use in these labs. before you could have named your files .v or .sv and it would not have mattered. Now for the project to compile, it must be .sv.

A. Implement state machine using state table and K-Map

B. The Difference between Meely and Moore State Machines

II. The Case Statement

Verilog makes use of the case statement like most other progamming languages. The case statement provides a clear way for your code to step through a procedure. It is common to implement a state machine using the case statement for a number of reasons.

1) Enumerated types show up in Signaltap for easy debugging.
2) The organized syntax creates more readable code.
3) easily expandable to include more states.

III. The state machine

This lab will assume that you have had a basic introduction to state machines in the lecture. We will cover some topics that are particular to the FPGA and HDL implementation of the logic. Altera offers a number of templates for the creation of a state machine [**?**]:

A. 4-State Mealy Machine:

This style of logic was coined in George Mealy's 1955 paper A Method for Synthesizing Sequential Circuits. The trademark feature is that it's outputs are determined by both the current state and the current inputs. [**?**]

B. 4-State Moore State Machine:

created a year after the Mealy machine the Moore Machine was described in a 1956 paper Gedanken-experiments on Sequential Machines. The difference is the Moore machine is only dependant on it's current state. [**?**]

C. Safe State Machine:

This style of machine uses a specific altera directive that inserts extra logic to detect invalid states and returns the state machine to the initial state.

D. User-Encoded State Machine:

This can be incorporated into all of the previous types. It allows the states to be named which aids in debugging and overall code readability.

We will focus on the Mealy and Moore state machines. Now even with just these two state machines there are a number of different ways to code them. I will reference a number of papers whose author's spent alot of time measuring the advantages of each style. The top performer is a "two always block" with the state enumeration provided by System Verilog. This is going to get heavy for a minute but focus on the templates for now, understanding will come with experience.

1) Important reccomendations from altera: Quartus will recognise when you have created a state machine during synthesis. This will allow Quartus to optimize the design based on the known behavior of state machines. The Quartus II handbook offers the following reccomendations for writing state machines that we will follow.

1) Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
2) Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
3) If your design contains an operation that is used by more than one state, define the operation outside the state machine and cause the output logic of the state machine to use this value.
4) Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus II software generates regular logic rather than inferring a state machine.

```
//double always block Moore type state machine

//The combinational always block sensitivity list is sensitive to changes on the state variable and all of the
//inputs referenced in the combinational always block.

//The combinational always block has a default next state assignment at the top of the always block.

//Default output assignments are made prior to the case statement (this eliminates latches and reduces the
//amount of code required to code the rest of the outputs in the case statement and highlights in the case
//statement exactly in which states the individual output(s) change).

//In the states where the output assignment is not the default value assigned at the top of the always block, the
//output assignment is only made once for each state.

//There is an if-statement, an else-if-statement or an else statement for each transition arc in the FSM
//state diagram. The number of transition arcs between states in the FSM state diagram should equal the number
//of if-else-type statements in the combinational always block.

//For ease of scanning and debug, place all of the next assignments in a single column, as opposed to placing
//inline next assignments that follow the contour of the RTL code.

module fsm_cc1_2(
  output reg rd, ds,
  input go, ws, clk, rst_n);

parameter IDLE = 2'b00,
          READ = 2'b01,
          DLY = 2'b11,
          DONE = 2'b10;

reg [1:0] state, next;

//Note all non
always @(posedge clk or negedge rst_n)
  if (!rst_n) state <= IDLE;
  else state <= next;

//note all blocking statements in this section
 always @(state or go or ws) begin
    next = 'bx;
    rd = 1'b0;
    ds = 1'b0;

    case (state)
      IDLE : if (go) next = READ;
        else next = IDLE;

      READ :
        begin
          rd = 1'b1;
          next = DLY;
        end

      DLY :
        begin
          rd = 1'b1;
          if (!ws) next = DONE;
          else next = READ;
        end

      DONE :
        begin
          ds = 1'b1;
          next = IDLE;
        end
    endcase
   end
endmodule
```

Listing 1: Example of enumerated state machine [**?**]

IV. Lab Procedure

A. *Case State Warmup: The Multiplexer*

B. *Binary sequence dector*

We are to build a state machine that can detect a binary sequence. This structure is a common use of the state machine for identifying start and stop conditions in a datastream. This will be a simple machine based on the starting sequence for the I2C bus.

V. Lab Report

VI. Conclusion