Lab Five: State Machines

Ben Smith

***Abstract*—The usage of state machines to control logic in Verilog and System Verilog are introduced in this document. Proper style and the recommendations of Altera for their Quartus development environment.**

I. Introduction

IN this lab we will look at the applications and different types of state machines. We will walk through a number of examples from Altera's technical documentation to investigate the different types of state machines available to the digitial designer and why you might choose a particular one.This lab is intended to introduce the student to the following concepts:

- Logic primitives on a FPGA
- Quartus development environment
- Synthesis of a block based design
- Assigning pins for a design
- Programming an Altera FPGA

A. Included Screencasts

A number of screencasts are included with this set of labs. They are available on Youtube and as zipped MP4s on the course website. They are intended to be short and to the point so they cover individual topics.

1) TIME - Stuff
2) TIME - Stuff
3) TIME - Stuff
4) TIME - Stuff

II. The Case Statement

VERILOG makes use of the case statement like most other programming 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 state names show up in Signaltap for easy debugging.
2) The organized syntax creates more readable code.
3) easily expandable to include more states.

It is important to note that logic implemented with the case statement could be modeled with multiple If-Else statements. Most designers choose to use the case statement because of it's concise readable format.

```
case(<StateRegister>)
    <Case 1>:<Statement>
    <Case 2>:<Statement>
    ....
    default:<Statement>
```

Listing 1: Instantiation example form the testbench

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 [1]

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. A mealy machine typically requires less states to perform the same operation as a Moore machine. Output defined in transitions. typically more flexable [2]

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. Output defined in states. [3]

C. 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. Signaltap can also be configured to show the named state values for ease of debugging. ScreencastX shows the use of encoded states in Signaltap.

D. Example State Machine

We could code a state machine directly from a state table. If you don't know how to do it well it will come back haunt you it has for me.

E. Coding the example state machine

A Mealy machine has outputs that depend on both the state and the inputs. When the inputs change, the outputs are updated immediately, without waiting for a clock edge. The outputs can be written more than once per state or per clock cycle.

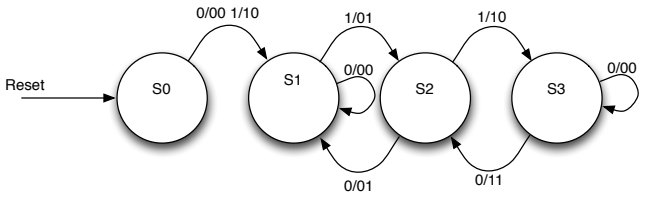

Fig. 1: Example Mealy machine

```
// 4-State Mealy state machine
module mealy_mac
(
  input clk, data_in, reset,
  output reg [1:0] data_out
);

  // Declare state register
  reg   [1:0]state;

  // Declare states
  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

  // Determine the next state synchronously, based on the
  // current state and the input
  always @ (posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else
      case (state)
        S0:
          if (data_in) state <= S1;
          else         state <= S1;
        S1:
          if (data_in) state <= S2;
          else         state <= S1;
        S2:
          if (data_in) state <= S3;
          else         state <= S1;
        S3:
          if (data_in) state <= S2;
          else         state <= S3;
      endcase
  end

  // Determine the output based only on the current state
  // and the input (do not wait for a clock edge).
  always @ (state or data_in)
  begin
    case (state)
      S0:
        if (data_in) data_out = 2'b00;
        else         data_out = 2'b10;
      S1:
        if (data_in) data_out = 2'b01;
        else         data_out = 2'b00;
      S2:
        if (data_in) data_out = 2'b10;
        else         data_out = 2'b01;
      S3:
        if (data_in) data_out = 2'b11;
        else         data_out = 2'b00;
    endcase
  end
endmodule
```

Listing 2: Mealy Machine

A Moore machine's outputs are dependent only on the current state. The output is written only when the state changes. State transitions are synchronous. which makes the Moore machine a little more resilient against bugs related to asynchronous feedback and race conditions.

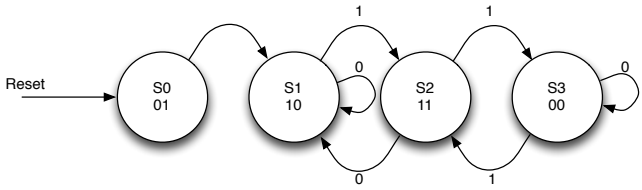

Fig. 2: Example Moore machine

```
// 4-State Moore state machine
module moore_mac
(
  input clk, data_in, reset,
  output reg [1:0] data_out
);

  // Declare state register
  reg   [1:0]state;

  // Declare states
  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

  // Output depends only on the state
  always @ (state) begin
    case (state)
      S0:
        data_out = 2'b01;
      S1:
        data_out = 2'b10;
      S2:
        data_out = 2'b11;
      S3:
        data_out = 2'b00;
      default: data_out = 2'b00;
    endcase
  end

  // Determine the next state
  always @ (posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else
      case (state)
        S0:
          state <= S1;
        S1:
          if (data_in)  state <= S2;
          else          state <= S1;
        S2:
          if (data_in)  state <= S3;
          else          state <= S1;
        S3:
          if (data_in)  state <= S2;
          else          state <= S3;
      endcase
  end
endmodule
```

Listing 3: Moore Machine

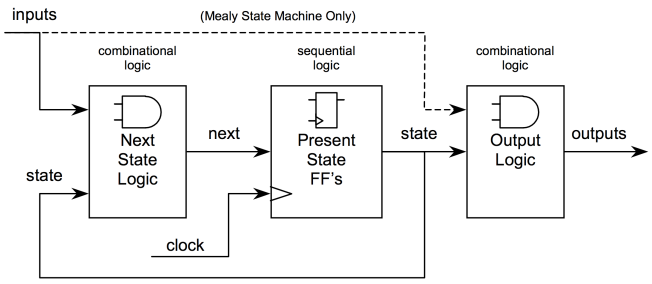

Fig. 3: Illustration of FSM logic [4]

```
        begin
38        rd = 1'b1;
          if (!ws) next = DONE;
40        else next = READ;
        end
42
      DONE :
44      begin
          ds = 1'b1;
46        next = IDLE;
        end
48    endcase
    end
50endmodule
```

Listing 4: Example of enumerated state machine [4]

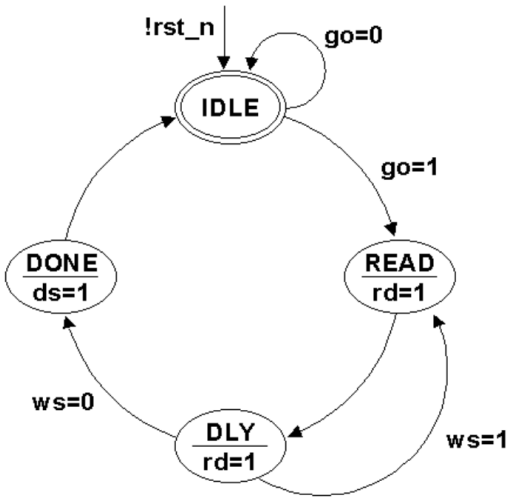

Fig. 4: State diagram for example code [4]

```
//double always block Moore type state machine
2 //Example from Clifford Cummings 2003 SNUG paper
      "Synthesizable Finite State Machine Design Techniques
      Using the New SystemVerilog 3.0 Enhancements"

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

8 parameter IDLE = 2'b00,
            READ = 2'b01,
10          DLY = 2'b11,
            DONE = 2'b10;
12
reg [1:0] state, next;
14
//Note all non blocking statements in this section
16 always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
18  else state <= next;

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

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

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

36      DLY :
```

Let's take a look at why the author of this code recomends this particular style of coding.

- 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 in line next assignments that follow the contour of the RTL code.

F. Important recommendations from altera

Quartus will recognize 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 recommendations for writing state machines that we will follow.

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- 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.
- 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.

IV. Lab Procedure

Laboratory Demo: Control of LED though state machine

Specification: Control the location of a single illuminated LED on the on board LEDs with the onboard switches though a mealy state machine.

Deliverable: DE0-Nano configured with control design

Process : Output LEDs will be inspected for proper operation as the student controls the LED

A. Binary sequence detector

A common use of a state machine is to recognize a sequence in an incoming signal. This can be useful when you want to take action on a specific trigger. This style of trigger is very common in digital bus systems. You will be given a digital sequence to write a detector for. The Analog Discovery's digital output can be used to generate your test signal. Illuminate the LED's on the Nano for one second once the signal has been detected.

Laboratory Demo: Binary sequence detector

Specification: Detect an incoming sequence of 4'h55

Deliverable: DE0-Nano configured with Comparator, Clean timing report from TimeQuest

Process : Output LEDs will be inspected for proper operation as the student cycles through the inputs

V. Lab Report

VI. Conclusion

References

[1] Altera. (2013) Quartus ii handbook. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii51007.pdf

[2] W. Foundation. (2013) Mealy machines. [Online]. Available: https://en.wikipedia.org/wiki/Mealy_machine

[3] ——. (2013) Moore machines. [Online]. Available: https://en.wikipedia.org/wiki/Moore_machine

[4] C. Cummings. (2002) he fundamentals of efficient synthesizable finite state machine design using nc-verilog and buildgates. [Online]. Available: http://www.sunburst-design.com/papers/CummingsICU2002_FSMFundamentals.pdf