E85 Lab 4: Finite State Machines

E85 Spring 2016
Due: 2/24/16

Overview:
This lab focuses on the creation of finite state machines circuits and their synthesis using Verilog as a hardware description language. You will learn how to build a finite state machine, and check its function with a self-checking testbench. Finally you will create an implementation of the door lock on the E85 lab as a real world example of a finite state machine.

What to turn in:
1. Your answers to the questions in part A
2. Did you complete part B?
3. Your answers to the questions in part C
4. The module and testbench codes for part C
5. The images of testbench waveforms for part C

Feedback:
Let us know:
What went well in this assignment?
Were there points of confusion?
How long did you need to work on this assignment?
Part A – Finite State Machines

To understand the D-Latch and FF in timing as in class please do questions:

3.3 and 3.5 from the ARM edition of the text

To understand how to create a FSM. Please complete question 3.26 in the text.

Follow the normal steps. In general you can choose your encoding for the states at any point after (2):

(1) Identify the inputs and outputs
(2) Determine how many states you will need
(3) Create a state transition diagram
(4) Create state and output tables from the state transition diagram
(5) Create Boolean expression for the next state and outputs
(6) Draw the circuit for the resulting system

Due to the complexity of this FSM, you do not need to complete step (6) for 3.26. Please use a 1-hot encoding for the states, this will allow you to not write out the encoding in tabular form as the states each have their own bit.

To better understand FSMs as circuits please complete question 3.31. This is going from the circuit backwards but you should be able to go backwards and forwards
Part B – Tutorial on Finite State Machines

In this tutorial you will learn how to create a finite state machine and write appropriate Verilog for the machine.

For this example we will create a 6 function calculator for 4-bit binary numbers. The functions will be ADD, SUB, AND, OR, XOR, NOR.

We will use two buttons to select the function being displayed as the output. The input numbers will be provided via the toggle switches on the DE2 board. We will supply a reset button for the system as well.

Because we do not know the clock speed of the target system in general it would be wise to capture if a buttons has been pressed rather than only if it is currently pressed.

To do this we can create a simple finite state machine.

We will call the input button B, the output if the button is pressed Bis and the output if the button was pressed Bwas.

Our state transition diagram for this Moore machine is:

**CLK: Rising edge of button (B); Bis = B**

The circuit this is equivalent to is:
We will now write the Verilog for this machine.

As normal we start with the module definition:

```verilog
module button( input logic Breset, B,
                output logic Bwas, Bis);
```

At this point we could take one of two approaches. Either we could implement the circuit or we could implement the state transition diagram. For this tutorial we will do both.

First we implement the circuit.

The canonical form of a D-Flip Flop with an asynchronous reset is:

```verilog
always_ff @ (posedge clk, posedge reset) begin
    if(reset) q <= 0;
    else q <= d;
end
```

Here we permute this to be:

```verilog
always_ff @ (posedge B, posedge Breset) begin
    if(reset) Bwas <= 0;
    else Bwas <= 1;
end
```

To finish the circuit implementation we need to connect Bis to B.

```verilog
assign Bis = B;
```

**Putting it all together we have:**

```verilog
module button( input logic Breset, B,
               output logic Bwas, Bis);

    always_ff @(posedge B, posedge Breset) begin
        if(Breset) Bwas <= 0;
        else Bwas <= 1;
    end

    assign Bis = B;

endmodule
Our alternative to using the circuit would be to write Verilog directly from the state transition diagram. We note that the arrows with the exception of reset are unconditional.

Using the standard form for a Moore machine with the extra assign statement for Bis results in:

```verilog
module button( input logic Breset, B,
               output logic Bwas, Bis);

typedef enum logic [0:0] {S0, S1} statetype;
statetype state, nextstate;
always_ff @(posedge B, posedge Breset)
begin
  if(Breset) state <= S0;
  else state <= nextstate;
end

always_comb
  case(state)
    S0: Bwas = 0;
    S1: Bwas = 1;
    default: Bwas = 0;
  endcase

  assign nextstate = Bis;
  assign Bis = B;
endmodule
```

Implementing the calculator

Having figured out how if a button was pressed we may now create a finite state machine that uses both was and is conditions to display the 6 functions on two 4 input numbers. We will call the input numbers R1 and R2.

When we first turn our calculator on we want the displayed output to be 0. Then the user must hold down the one button (called A) while pressing and releasing the second button (called B). This should change the display to output R1 + R2. Again holding down A while pressing and releasing B should create R1 – R2 on the display. The process continues until the user cycles around to addition again.

The resulting state transition diagram is on the next page. Remember that when a wire/signal is not present in a bubble of a state transition diagram it is not asserted (its value is 0).

Although the diagram looks complex it really is the same two things over and over again.

Please consider carefully how the diagram accomplishes the stated goal. “Else” is used to note that all other conditions lead to the location. Also note that Awas is never used so we may just use the signal A rather than creating a machine to remember A’s past.
CLK is on DE2 board

S0  Result = 0  Breset

S1  Result = R1 + R2

S2  Result = 0

S3  Result = R1 - R2

S4  Result = 0

S5  Result = R1 | R2

S6  Result = 0

S7  Result = R1 & R2

S8  Result = 0

S9  Result = R1 ^ R2

S10 Result = 0

S11 Result = ~(R1 | R2)

Bwas & Ais & BIs

Reset

Bwas & Ais & BIs

Bwas & Ais & BIs
At this point we can draw a circuit that creates the desired machine or we can directly write Verilog from the diagram. The latter is easier but has the hazard of making novices think they are programming.

**HDL is not programming. You are describing a circuit.**

We start with the module definition and the definition

```verilog
module lab4tutorial(input logic [7:0] SW, 
                    input logic [2:0] KEY, 
                    input logic CLOCK_50, 
                    output logic [4:0] LEDR);
endmodule
```

We then add some internal signals to make the code more readable and instance the button module:

```verilog
module lab4tutorial(input logic [7:0] SW, 
                    input logic [2:0] KEY, 
                    input logic CLOCK_50, 
                    output logic [4:0] LEDR);

logic Ais, Bwas, Bis;
logic resetB, reset, clk;
logic advance;
logic [3:0] R1, R2;
logic [4:0] result;

assign clk = CLOCK_50;
assign Ais = ~KEY[0];
assign R1 = SW[7:4];
assign R2 = SW[3:0];
assign LEDR = result;
assign reset = ~KEY[2];

assign advance = Bwas & Ais & (~Bis);

button B(resetB, ~KEY[1], Bwas, Bis);
endmodule
```

CLOCK_50 is a 50 MHz clock on the DE2 board. KEY are the buttons, SW and LEDR you met before in lab 3.

Notice the advance logic signal replaces our condition to change to the next displayed function on all of the arrow in the state transition diagram. Also notice the KEYs are normally high so we invert them.

Next we write the code that defines the states and the next state logic:

```verilog
typedef enum logic [3:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11} statetype;

statetype state, nextstate;
always_ff @(posedge clk, posedge reset)
begin
  if(reset) state <= S0;
```
else state <= nextstate;
end

Next we implement the next state logic, we need to use if/else here:

always_comb
  case (state)
    S0: nextstate = S1;
    S1: if(advance) nextstate = S2; else nextstate = S1;
    S2: nextstate = S3;
    S3: if(advance) nextstate = S4; else nextstate = S3;
    S4: nextstate = S5;
    S5: if(advance) nextstate = S6; else nextstate = S5;
    S6: nextstate = S7;
    S7: if(advance) nextstate = S8; else nextstate = S7;
    S8: nextstate = S9;
    S9: if(advance) nextstate = S10; else nextstate = S9;
    S10: nextstate = S11;
    S11: if(advance) nextstate = S0; else nextstate = S11;
    default: nextstate = S0;
  endcase

Lastly we implement the outputs:

always_comb
  case (state)
    S0: result = 0;
    S1: result = R1 + R2;
    S2: result = 0;
    S3: result = R1 - R2;
    S4: result = 0;
    S5: result = R1 | R2;
    S6: result = 0;
    S7: result = R1 & R2;
    S8: result = 0;
    S9: result = R1 ^ R2;
    S10: result = 0;
    S11: result = ~(R1 | R2);
    default: result = 0;
  endcase

assign Breset = (state == S0) | (state == S2) | (state == S4) | (state == S6) | (state == S8) | (state == S10);
Notice in the last block we simply used + and – to do the arithmetic. The Verilog synthesizer is smart enough to do this operation for us without an explicit implementation for the circuits to create the adders!

The downside to how the code is currently written is that the synthesizer may create two adders one to add and one to subtract. The subtract operation is performed using 2’s complement encoding so the reuse of the adder would be possible. Because adders are large piece of hardware this is non-optimal in many cases. For now though it is okay. We will learn a more optimal way to do this in the next lab.

The full System Verilog file with comments is available at:

http://pages.hmc.edu/bbryce/E85S16/files/lab4tutorial.sv

Download the code and make sure it compiles nicely.

**Extend the example code to add NAND as a function.**

**Now check that the hardware works by programming the DE2 board with the bitfile for your 7 function calculator.**

You will want to press down the on KEY buttons firmly to keep them registered as pressed.

**Self-Checking Testbench**

Having been introduced to if/else in the context of Verilog they can be used as *programming* constructs for a testbench. We can use Verilog to write a procedure that *is programming* to test our module.

This trips many novices up: writing procedures in testbenches is essentially programming, though it is a program to *simulate hardware stimulus* and check the results.

The Verilog you write for *synthesis* is a *circuit*. A wire must have a value at all times in a circuit. A variable in a program does not have this requirement. In a circuit many logical calculations occur in parallel (each NAND or NOR is independently looking at its inputs and places its output on a wire). In a program, statements occur sequentially within a procedure. Although it is possible to write sequential Verilog code in an always block with blocking statements, it rarely is the correct way to make a circuit!

Part of a self-checking testbench is given on the next page.

To fit on a single page the version given shows only the first 2 tests. The entire file is downloadable here:

http://pages.hmc.edu/bbryce/E85S16/files/lab4tutorialTest.sv
module lab4tutorialTest();

//internal signals for simulation
logic [3:0] R1, R2;
logic reset, clk, A, B;
logic [4:0] result;

//for testing code only
logic [31:0] count;
logic flag;

//Create our hardware to test, keys flipped to simulate the buttons
calculator dut({R1, R2}, {~reset, ~B, ~A}, clk, result);

// generate clock with 10 ns period
always begin
    clk <= 1; #5; clk <= 0; 
end

initial begin
    count = 0; // set the count to 0
    flag = 1;
    // reset is initially true, goes low at 18 time units
    reset <= 1;
    #15;
    reset <= 0;
    #10; // align data changes to falling edge of clock

    // Add, test 0
    R1 = 4'b0010;
    R2 = 4'b0001;
    A = 1'b1;
    B = 1'b0;
    #16; // wait until after next clock edge to test
    // Test
    if(result === 5'b0011) $display("Passed Test: %d", count);
    else begin
        $display("Failed Test: %d", count);
        flag = 0;
    end
    count = count + 1;
end

// Subtract, test 1
// R1 is still 0010, R2 is still 0001, etc...
B = 1'b1;
#10; // wait 1 cycle
B = 1'b0;
#16; // wait until after next clock edge to test
if(result === 5'b0001) $display("Passed Test: %d", count);
else begin
    $display("Failed Test: %d", count);
    flag = 0;
end
count = count + 1;
#4; // wait until clock edge

if(flag == 1) $display("All tests passed!");
else $display("Some tests failed :-(");
endmodule
Looking at the Verilog we see several things. First we see Verilog used stylistically as a *programming language* rather than for hardware description. The primary block of code is in the *initial* block. This code is run once at the start of the *simulation*.

Because of the use of the blocking = sign rather than the non-blocking <= the assignments in this block happen in order as listed. Although the blocking operator can be used to create hardware, particularly complex combinational circuits, it is important to remember that real hardware always requires that everything happen at once so the non-blocking <= is more naturally used in most cases to make hardware.

Here we are just trying to tell the simulator what we want to happen so the more normal sequential procedure of the blocking operator is my default.

In addition to the main block for testing there is an *always* block that creates the clock. This runs *always* and forever. You can have as many blocks as you want run in parallel. Verilog is again primarily a hardware description language. So each block is independent. Each always block will run whenever its sensitivity is satisfied.

The other new items here are the use of programmatic if and else to test cases rather than the “okay” signal we used in Lab 3. We combine this with the $display macro which prints text in to the transcript window of the simulator. The %d allows us to print a decimal number.

*Try out the testbench supplied with ModelSim. Look at both the transcript and the waves. Try plotting the state in the DUT along with the inputs and outputs. Make sure they all make sense.*

The repetitive nature of the code in this testbench indicates that we could create a looping structure to perform the tests. We will create more advanced testbenches in later labs.
Part C – Implement the Door Lock

The E85 lab has a push button door lock. Study the operation of the door lock. It has 5 input buttons and a knob. In this part of the lab you will make an implementation of a similar door lock as a finite state machine. The only differences will be the success code and that our DE2 boards only have 4 buttons rather than 5.

Write down a paragraph describing the lock’s operation and what happens when you get the code correct and incorrect. What buttons are pressed in the state between keying in each number of the “code”? How will you make use of this?

Any FSM needs a clock to advance the state of the machine. This clock is provided for you. It will run much faster than you pressing buttons! This means it will not be possible to press the buttons simultaneously. As a result you will need to use a trigger circuit like the one in the tutorial to remember what buttons are pressed.

Draw a state transition diagram that implements the lock with the correct answer being: 12-34-2.

Write Verilog to implement the state transition diagram.

To keep the implementations slightly more uniform please use the following top module definition, and initial code:

```verilog
module lab4lock(input logic [1:0] SW,
                 input logic [3:0] KEY,
                 input logic CLOCK_50,
                 output logic [0:0] LEDR);

  logic clk, reset, success, knob, b1, b2, b3, b4;
  assign {b4, b3, b2, b1} = ~KEY[3:0];
  assign LEDR[0] = success;
  assign knob = SW[0];
  assign reset = SW[1];
  assign clk = CLOCK_50;
```

The internal signals clarify the interface: knob is turning the knob with 1 being the knob is turned; b1 is the number 1, b2 is the number 2, b3 is the number 3, and b4 is the number 4. LEDR can be used to show success (that the door will open).

Write a testbench to check the function of your lock. Simulate both success and failure. Be sure to simulate the fact that not all buttons are pressed at the same time! In your waves, be sure to plot all input and output signals as well as the internal state of the machine.

Synthesize your code and try it out on the DE2 board. In general momentary switches (buttons) may bounce (connect multiple times). This can lead to unexpected inputs. The buttons on the DE2 board are debounced by RC circuits. They will work well if pressed nicely.