18-240: Structure and Design of Digital Systems

Lab3: Count to 15? Is that like Magic Squares?

Objective and Overview

In this lab, you will explore FSM design and implementation strategies. You will implement the same simple FSM in two different ways. You will discover that testing FSMs requires a different method of testbench construction. You will implement your design on the DE2-115, compare the synthesis results of the two versions, and then add additional capability to the FSM.

I recommend reviewing the reading from LDVUS 7.3, especially section 7.3.2, as preparation for this lab. In addition, I *highly* recommend ensuring your lab partner has read (and understands) those sections as well.

Schedule and Scoring

Lab 3 is a two-week exercise. You will want to accomplish some of the design task prior to coming to lab, but the remainder should be easily completed during your lab sessions. **You are expected to show up, be present and work on the lab for the entirety of the first lab session.**

8 October	Waveform Viewer Bootcamp
10 – 12 October	Complete tasks 1-4.
17 – 20 October	Fall Break – no lab session.
24 – 26 October	Task 5. Demo all tasks by end of lab.

Lab 3 is worth 100 points, split among five tasks: Structural Implementation, Abstract Implementation, Coverage, Synthesis and Expansion. The first four tasks are worth 15 points each, for a total of 60 points. The Expansion task is worth 25 points. 15 points are also assigned for code quality: good style, adherence to standards, and quality of your testbenches.

The first four tasks **must be completed** during your first week in lab. The Expansion task is designed for your second week in lab.

There is no lab report required for this lab. Submit all your SV files to Gradescope by the end of your 2nd lab session.

Late Penalties (the standard drill)

Late demo or code: You will lose 10% of the points for an item (demo of an individual task or for code) for each day late up to a week late. After a week, you will get zero points. (After which, you still need to complete the demo to pass the course.) When counting days, we will not include any weekend or holiday day on which there were no TA office hours.

A Note about Teams and Collaboration

By now, you should be getting used to working with a random partner, which is a great skill to have as a modern engineer. Continue to be a good teammate, communicate well, and work well.

Once again, all work you turn in for this lab is expected to be that of your team. You may ask other students for general assistance, but you may not copy their work. Likewise, you may not copy work from other sources.

Debugging With the Waveform Viewer

The absolute most important debugging tool for FSMs is the *waveform viewer*. Sequential circuits (i.e. FSMs) deal in signals that are *sequences* and an understanding of those sequences will be of great aid in verifying your design. The waveform viewer is specifically designed to let you visualize *sequences*. Time spent learning to use the waveform viewer will inevitably pay off in this lab.

The Mission: A "Sum to 15" Game

Let's play a game! This game is one in which two players take turns saying a number between 1 and 9 (inclusive). The number may not be a number previously stated. If, of all the numbers a player has said, there exist 3 numbers whose sum is 15, that player wins.

To make it simple, the "computer" player will always start the game, and will always say "5." The human player will then choose any other number desired, as long as it is "9." (This is a pretty simple version of the game). The computer player will then take "6." From this point, play should continue, but ensure the computer player makes the best available choice at each point. **To be very clear: your FSM needs to play through these initial moves.** Upon reset, it should show the computer choosing 5, then waiting for the human to take 9, etc.

If the human player ever enters an invalid move (i.e. a number that has already been taken, or one which is not 1-9), play should not advance. Simply ignore the invalid input until the human player comes to his or her senses and plays a proper move. When the computer player wins, wait until reset is pressed before starting another game.[1]

The input to your circuit is a four-bit value specifying the human player's move. Of course, the circuit will also need a clock and a reset signal.

The circuit should output a four-bit value showing the computer's move. Another single-bit output is a "win" signal, only active after the computer has won the game.

First, draw the state transition diagram for your FSM. Next, design the next state and output logic. Use a fully-encoded assignment scheme. Your state-transition table is probably somewhat large, so we aren't going to force you to K-map it or use Boolean algebra to minimize the equations. By

[1]Oh, did you think the humans stood a chance? I, for one, welcome our new FPGA-based "Count-to-15" playing overlords.

this point, you are the experts of combinational circuit design, so make good choices to design the next-state and output-generation logic. Make sure your design is correct before moving on.

Task 1: Structural Implementation

Implement your FSM in synthesizable SystemVerilog, using a structural-ish style. You need not use strictly structural style (i.e. gates) for your combinational circuitry — continuous assign statements are probably better. However, you will have explicit flip-flops.

Use continuous-assign statements for your combinational circuitry. These statements should only use the "gate" operations (&, | and ~).. Do not use **`always_comb`** blocks. Do not use the tertiary operator (i.e. **`(is_true) ? A : B`**). Your equations need not be minimized, as Quartus will do that for you.

Use the following D-flip-flop module to hold the state. Note that this D-flip-flop has a synchronous reset. What does synchronous reset mean? I hope you will understand by the time you've finished this lab.

```
module dFlipFlop(
  output logic q,
  input  logic d, clock, reset);

  always_ff @(posedge clock)
    if (reset == 1'b1)
      q <= 0;
    else
      q <= d;

endmodule: dFlipFlop
```

Your FSM should look something like this.

```
module myExplicitFSM(
  output logic [3:0] cMove,
  output logic       win,
  output logic       q0, q1, q2, // connect to FF outputs(add more if needed)
  input  logic [3:0] hMove,
  input  logic       clock, reset);

  logic d0, d1, d2;  // connect to FF inputs (add more if needed)

  // Example instantiation of D-flip-flop.
  // Add more as necessary.
  dFlipFlop ff0(.d(d0), .q(q0), .*),
            ff1(.d(d1), .q(q1), .*)...

  // Next state logic goes here: combinational logic that drives
  // next state (d0, etc) based upon input hMove and the
  // current state (q0, q1, etc).
  assign d0 = ...
  assign d1 = ...

  // Your output logic goes here: combinational logic that
  // drives cMove and win based upon
  // current state (q0, etc) and hMove.
  assign cMove = ....
  assign win = ....

endmodule: myExplicitFSM
```

That wasn't so hard, was it?[2] Don't start celebrating just yet. We still need to test it. Previously, we've seen how easy it is to test combinational logic, but sequential logic is an entirely different beast. Fortunately for us, our design is small enough that we can test each state and each state transition. That won't always be the case, but we can worry about that later.

You need to design a set of test vectors that will take the FSM through every state and through every transition. Think about how to do this; remember, you can only control the FSM by its input sequence. (Don't forget, you are allowed to reset the FSM in the middle of the test.)

Below, we provide you with a sample testbench module that demonstrates a few key aspects of sequential testbenches. It uses a style very similar to that found in the *Logic Design Using SystemVerilog* textbook (section 7.3.2). Note, for instance, that the state bits (q0, q1, etc.) are brought from the FSM and are additional inputs to the testbench. This unusual step allows the **$monitor** statement to print them and let us know what the current state is. Knowing this information makes testing and debugging the FSM much, much easier.

```
module myFSM_test(
  input  logic [3:0] cMove,
  input  logic       win,
  input  logic       q2, q1, q0,
  output logic [3:0] hMove,
  output logic       clock, reset);

  initial begin
    clock = 0;
    forever #5 clock = ~clock;
  end

  initial begin
    $monitor($time,, "state=%b, cMove=%d, hMove=%d, win=%b",
```

[2]Good thing you are a master of combinational logic!

```
            {q2, q1, q0}, cMove, hMove, win);
    // initialize values
    hMove <= 4’h4; reset <= 1'b1;

    // reset the FSM
    @(posedge clock); // wait for a positive clock edge
    @(posedge clock); // one edge is enough, but what the heck
    @(posedge clock);

    @(posedge clock); // begin cycle 0
    reset <= 1'b0;    // release the reset

    // start an example sequence -- not meaningful for the lab
    hMove <= 4’h4; // these changes are after the clock edge
                    // which means the state change happens
                    // AFTER the next clock edge

    @(posedge clock); // begin cycle 1
    hMove <= 4’h9;
    @(posedge clock); // begin cycle 2
    hMove <= 4’h4;
    @(posedge clock); // begin cycle 3
    hMove <= 4’h9;

    // could check FSM outputs like so.  Be careful about timing
    #1 if (~win)
      $display("Oops, incorrect hMove value at cycle 3");

    @(posedge clock);
    #1 $finish;
end

endmodule: myFSM_test
```

The most important new thing we see here is **`@(posedge clock);`**. When this is written as a statement in an **`initial`** process, it instructs the simulation to wait until the next positive clock edge before continuing to the next statement. In this way, we can easily synchronize the input vector relative to the clock. For example, in the above, we are changing the input slightly after the clock edge (by using a non-blocking assignment). The input will then drive the next-state generation logic to determine the next state for the *next* clock edge.

Oh, I should probably mention that you should think hard about the reset generating code shown above. In particular, why is the reset signal not being generated in the same **`initial`** block as the **`clock`** like was shown in class? Also remember that the flip-flops have a synchronous reset—an unusual feature. Most flip-flops we’ve seen (and will see) have an asynchronous reset. I’m having you use synchronous to force you to understand the difference; as this is a point of confusion for many students.

Expand upon the testbench skeleton to test all states and transitions in your FSM. It's useful to document (in comments) what the desired FSM behavior is, or to have the testbench output "Ooops" statements whenever it detects undesired behavior.

For credit: Show the TA simulation output proving that your testbench successfully tests all your FSM states (including the reset transition).

Task 2: A More Abstract FSM Design

The structural style of FSM design is fine when you have a few states, but what happens when things become complicated? We would never build a big combinational circuit at the gate level, so why would we build a big sequential circuit at the gate and flip-flop level? Instead, let's design the FSM at a higher level of abstraction, and let the synthesis tool take care of the low level logic. Let's look at a SystemVerilog FSM example that uses symbolic state names and can be transcribed nearly literally from a state transition table or diagram.

Also, you can use whatever procedural statements you like for this part (i.e. you are released from the restriction on combinational statements).

```
module myAbstractFSM (
  output logic [3:0] cMove,
  output logic       win,
  input  logic [3:0] hMove,
  input  logic       clock, reset);

enum logic [2:0] {S0, S1, ..} currState, nextState;
// Increase bitwidth if you need more than eight states
// Don't specify state encoding values
// Use sensible state names

// Next state logic is defined here. You are basically
//   transcribing the "next-state" column of the state transition
//   table into a SystemVerilog case statement.
always_comb begin
  case (currState)
    S0: begin
      // Assign a value to nextState based on input
      // This is an example, not a solution to part of the lab
      nextState = (hMove == 4'h9) ? S0 : S7;
    end

    // ...
    // one case for each state in your FSM
    // ...

    default: begin
      nextState = S0;
    end

  endcase
end

// Output logic defined here. You are basically transcribing
//   the output column of the state transition table into a
//   SystemVerilog case statement.
// Remember, if this is a Moore machine, this logic should only
//   depend on the current state.  Mealy also involves inputs.
always_comb begin
  cMove = 4'b0000; win = 1'b0;
  unique case (currState)
    S0: cMove = 4'b0100;
    S1: begin
           win = 1'b0;
           cMove = 4'b1000;
         end

    // ...
    // one case for each state in your FSM
    // ...

    // no default statement needed, due to unique case

  endcase
end

// Synchronous state update described here as an always_ff block.
// In essence, these are your flip flops that will hold the state
// This doesn't do anything interesting except to capture the new
// state value on each clock edge.  Also, synchronous reset.
always_ff @(posedge clock)
  if (reset)
    currState <= S0; // or whatever the reset state is
  else
    currState <= nextState;

endmodule: myAbstractFSM
```

Note that we are careful to ensure the case statements generate combinational circuitry — either by providing a **`default`** to our case statements or by using **`unique case`**.[3] This will let us indicate to the synthesis tool that this is combinational logic, and we have listed the cases we care about. By following this coding style closely, the synthesis tool is able to infer that this is an FSM, and can even change your state encoding to optimize for you.

Recreate your FSM from Part 2 in this style. You can use the same testbench as before to simulate this FSM, though you will need to modify your **`$monitor`** statement as you no longer have individual state bits. Use the **`.name`** magic (from Lecture 10) to print your state names as strings in the **`$monitor`** instead.

For Credit: This part will be checked off with Part 3.

Task 3: Simulation with Coverage Analysis

An interesting simulation option in VCS is *coverage analysis*. By activating this option, the simulator will keep track, on a line by line basis, of how many times each line of your SystemVerilog description has been exercised. In this way, you can verify that your simulation puts the FSM through each state and transition. For example, here are a few lines of a coverage analysis output (for some other FSM):

```
27          1/1                Saw00: if (in_val == 2'b11)
28          0/1     ==>              nextState = Saw0011;
29          1/1                    else if (in_val == 2'b01)
30          1/1                      nextState = Saw0001;
31                                 else
32          1/1                      nextState = Init;
```

In the above, the first column of numbers shows the line numbers in your SystemVerilog source. The second column indicates the number of times that line of code was exercised out of the total executions of the block. See that this FSM was in **`Saw00`**, but never transitioned to **`Saw0011`**. Note that if we had written a conditional statement as

```
nextState = (x==1) ? stateB : stateC;
```

we would not be able to see the transitions individually — the tool would just report that it evaluated this line 4 times.

To simulate with coverage analysis, follow the normal routine to simulate your SystemVerilog design. You will need to modify the compilation statement by providing a coverage option such that "lines" are counted (**`-cm line`**). In addition, you'll may want to force VCS to count continuous assignment statements (**`-cm_line contassign`**):

```
vcs -sverilog -cm line -cm_line contassign *.sv
```

Next run your simulation, but request the simulation accumulate coverage data:

[3] You may find **`default`** statement a better choice here, as you have a synchronous reset. Can you figure out why?

```
./simv -cm line
```

At this point, you have some coverage data (stored in a directory named **`simv.vdb`**), but you want a readable coverage report, not a binary database of unreadable stuff. Use the "Unified Report Generator" to turn that stuff into something useful.

```
urg -dir simv.vdb -format text
```

A report will be generated in the directory **`urgReport`**. The report is broken up into a bunch of separate text files. The first, **`dashboard.txt`**, gives a very concise overview, without much useful detail:

```
-----------------------------------------------------------------
Total Coverage Summary
SCORE  LINE
 84.51  84.51
```

The **`hierarchy.txt`** report gives a similarly concise overview, broken down by module. The really useful detail is in **`modinfo.txt`**, which shows how many times each line was executed and highlights (with **`==>`**) any lines that weren't executed. Note also that some lines aren't "coverable" in that they specify stuff that doesn't actually execute. These are lines with **`always_ff`**, **`begin`**, **`else`**, comments, **`endmodule`**, etc.

If you prefer prettier reports, you can skip the "**`-format text`**" part of the **`urg`** command and you will get a series of html files.

Take a look at the report. Most of it is self explanatory. You might spot a statement that was not executed, in which case you will need to update your testbench.

A second coverage option is particularly useful for this lab. Re-compile, re-simulate, and re-urg your code with:

```
vcs -sverilog -cm fsm *.sv
./simv -cm fsm
urg -dir simv.vdb -format text
```

The **`fsm`** option in the cover flag is for FSM. Take a look at the new **`modinfo.txt`** file to see that you get a report that knows about your FSM states. It tracks how often you get to each state and how often each transition is followed. Useful, eh? Update your testbench if any coverage statistic is lower than 100

There are two caveats with respect to getting VCS to show 100% test coverage. The first is that the tool will count a transition as fully tested if the transition is ever taken. To actually get full coverage, you need to actually take each transition for every combination of inputs. For instance, if the transition from **`State0`** to **`State1`** happens when **`A=1`** and **`B=X`**, you actually need two different tests to cover this transition (where **`B=0`** and where **`B=1`**). If you only run a single test with **`B=0`**, the tool will still show 100% coverage. So, you will need to have 100% coverage AND be able to show the TAs that you have each possible combination tested.

The second caveat is that the tool wants you to follow the `reset` transition from each and every state. I'm not at all concerned about testing those — I figure if you've tested `reset` at all, you probably got it correct. So, if VCS shows less than 100% coverage, but you can show that it is only missing reset transitions, you will still get full credit.

It may be useful to run `line` and `fsm` coverage at the same time. The coverage metric flag for that is `-cm line+fsm`.

For Credit: Show the TA your coverage report to prove your testbench has 100% coverage of each FSM statement and each FSM state/transition. Note that "each transition" means taking that transition for each input combination that causes the transition. If your STD shows an arrow with multiple numbers on it, you need to test that transition for each number.

Task 4: Synthesis

You now have two versions of the same FSM. In the first, you optimized the logic by hand. In the second, you let the SystemVerilog tools optimize the logic for you. Synthesize both designs using Quartus II on your lab computer.

Compare the number of logic elements, registers, and the maximum clock frequency (F_{max}) needed by the two designs. The first two measurements can be found in the "Flow Summary," while the clock frequency is found in the "TimeQuest Timing Analyzer, Slow 1200mv 85C Model, Fmax Summary". Comparing the values for the two circuits, what do you think about abstract models? In particular, can you say anything about the encoding mechanism used?

Next, download the FSM onto the DE2-115 FPGA board (use your second design). Here is your user interface specification:

Signal	Board Connection
`clock`	`KEY[0]`
`reset`	`SW[17]`
`hMove`	`SW[3:0]`
`cMove`	`HEX0`
`win`	`LEDG[0]` (lit when asserted)

Oops! The seven-segment LEDs need some sort of device to translate from a binary value. Hmm.. I wonder where you will get a component like that?

For Credit: Demonstrate your circuit to the TA. Show the TA your size and speed measurements for the two different circuits and explain your hypothesis about encoding mechanisms. Be able to demonstrate and describe what a synchronous reset means.

Task 5: Expansion – The Big Time

Now that you know how to design and write FSMs, especially in the "abstract" style that saves so much time, let's build on that to flesh out your design.

For the remainder of the lab, you will build upon your earlier design to make it a bit more user friendly. In the process, you will add more states. Yep, a bunch more states. Luckily, SV lets you write the FSM in an "abstract style" and thus adding more states isn't too much of a problem. The game will still conform to the limits on the first three moves.

It is silly to have the user push a clock button to move from state to state. Most devices don't work that way. Instead, there is a constant frequency clock that drives the FSM. One measure of the coolness of your laptop is how high that frequency is, for instance. The DE2-115 FPGA board has a 50 MHz clock hooked to one of the input pins — we will use that as the clock for our new circuit.

Using such a fast clock has some implications, however. Users have a hard time pressing buttons fast enough to keep up with such a clock. You will have to modify your state transition diagram (and thus, your FSM circuit) so that "intermediate" inputs made by the user don't take the system down the wrong path. You will also need to worry about synchronizing your inputs (as was discussed in Lecture 12).

While we are at it, our users are going to demand to be able to see more of the game conditions than were available in the earlier version.

For all these reasons, I've modified the specification of the game system as shown in the table on the next page.

Note: This is a lab about building big FSMs, not about building datapaths (be patient, that will come). Don't even think about using registers and sorting to display the `HEX` values. Instead, figure out what those values should be for the state you are in. They are a function of the state (and perhaps the inputs).

For credit: Demonstrate your synthesized circuit to the TA. Be sure to explain to the TA how you synchronized your inputs and what that means.

Inputs

`SW[3:0]`	**`hMove`**
`CLOCK_50`	the 50MHz clock
`SW[17]`	**`reset`** – At 50MHz, it will be hard to tell the difference between an asynchronous reset and a synchronous one, so you can choose whichever you like.
`KEY[3]`	**`enter`** – This key needs to be pressed in order to register that the switches have the appropriate value for **`hMove`**. The **`hMove`** takes effect as soon as **`enter`** is pressed, so changes to **`hMove`** while pressed won't matter.
`KEY[0]`	**`new_game`** – After a game has been played (and the **`win`** signal is displayed), press this button to play another. The new game should not start until this button (and the enter button) are released.

Outputs

`LEDR[3:0]`	**`cMove`** – The computer's latest move.
`LEDG[0]`	**`win`** – Lit when the computer has won.
`HEX3-0`	Displays the moves the computer has made so far. Values should be sorted, with the lowest value appearing on **`HEX3`**. HEX displays will be blank if there is no move yet for that display. For instance, at the beginning of a game, the computer has made one move, so **`HEX3`** will display **`5`** and **`HEX2/1/0`** will be blank. After the computer's second move, **`HEX2`** will display a **`6`**. If the computer's second move had somehow been a **`2`** instead, then **`HEX3`** would display **`2`**, **`HEX2`** would display **`5`** and **`HEX1/0`** would still be blank.
`HEX7-4`	Displays the moves the human has made so far, in a similar fashion. At the beginning of the game, the human has made no moves, so all four should be blank. Moves are sorted, with the smallest value showing up on **`HEX7`**. See the reference board if you are confused.