Casino-type Game

PURPOSE - In this lab, you will design a casino-type game using the previous lab's random number generator. Also, you will learn about how to write a synthesizable Verilog code for finite state machines.

1. Writing the Verilog description

You will have to design a game using the random number generator from the previous lab. This game will function as follows: **On pressing the Start button (e.g., reset button from Lab 4),** the random number generator will generate numbers. When the Roll button is pressed, you will use these two hex digits to play the game.

To reuse the most code possible, you will have the two random hex numbers displayed in real-time on the seven segment display (on either the left two or the right two digits, just like in lab 4), and then display the win "UI" or lose "LO" notification on the remaining two digits. **Read the following example to understand the proper operation of the game:**

A sample play of the game is as follows:

1) The player starts to generate random numbers. **The player presses the Start/Reset button, so the random numbers are generated** on two digits, just as in lab 4.

2) The player then **taps** the Roll button. Sum up the current two hex numbers, if the sum is greater than 25, then "UI" is displayed, and the player wins. If the sum is less than 5, then "LO" is displayed (on the other remaining two digits), and the player loses. In both cases, the game is over. If neither of the above cases is hit, store the sum and proceed to the next step. (here hex digits are represented by their 4-bit number, i.e., A=10, F=15)

3) If the sum was between 5 and 25 (5 and 25 included), store the sum as "target" the player has to roll again, and 'Ao' is displayed. On pressing the roll button, take the sum if the new sum is less than the target, roll again. If the new sum is greater than or equal to the target but lesser than or equal to 25, the player wins, and if it is greater than 25, the player loses.

4) The "UI" or "LO" is still displayed until the start button's next tap.

For simplicity, "LO" can be displayed before the first tap of the roll button.

Of course, the player can always see the value of the hex digits, and thus tap the Roll button at the right time to win or lose the game, but this is done for debugging purposes.

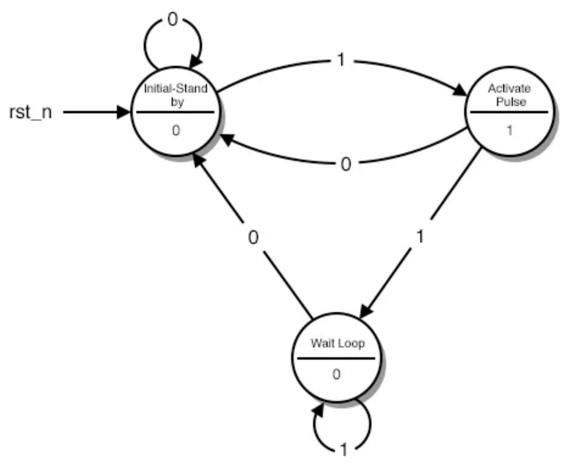
2. Verification

Design and write a testbench for your FSM controller and simulate all the above scenarios. Synthesize and implement the game to verify the correctness. You will have to write a **.xdc** file to correctly map the pins (you only need to add two more lines for the Roll button.) Download the bitstream file to the board. Show the correct functioning to your TA for full credit.

Notes

The control logic for this game is best implemented as **a finite state machine.** The Verilog files (from the textbook) show a similar design. They include a top-level module that instantiates the counter(our case uses an LFSR). A clock_divider (same as used in the previous lab, but you will want to set a different time constant), and a partial state machine module will implement—a dice game control (ours implements a similar game). The top-level module also implements logic to control the 7-segment LEDs, which you will need to correct (it has don't cares to begin with). Complete the Verilog modules, and implement the design. Then, you will download the bitstream file to the lab boards and verify the game.

The class textbook contains a diagram of a sample state machine for a similar game implemented in the control module. You can similarly implement this state machine or simplify it if you understand the game requirements sufficiently. To test and verify your design, you should set a reasonably large time constant (50 or 60) for the clock divider so that the dice roll very slowly so that you can see what roll is active and stop at the interesting numbers. To play the game, you should set a shorter time constant (say 15).



You will have to draw your SM chart of the dice game and its implementation design with F/Fs and basic gates, including all the internal interconnection between components. **You will have to attach this diagram to the lab note**book, which you will turn in! A sample state machine diagram/ state transition diagram is shown on the left. The diagram should contain only states and transitions. A state transition diagram is not a flowchart. A state table is a concise summary of the FSM states. The columns include the state, input, next state, and output.

The dice game's output will have to drive at least one of the 7-segment LEDs of the BASYS3 board. The LED will indicate "UI" in case that the player wins, "LO" that the player loses, or "Ao" in case that the player has to play/roll again. When the game is reset, the 7-segment LED displays "0". Figure out (using the manual for the BASYS-3 board manual, available from the class website) what pin assignments you have to use to activate the 7 segments of each 7-segment LED properly. Also, assign the proper pins to use **one push button** as **the Reset** of the game entity, **another push button** as **Rb**. For the **CLK_IN** input in your game entity, assign PIN **W5** in the Constraints File (.xdc) of your project, which will have to be used for implementation.

The first step to verifying your design is to write a test bench that tests your FSM module. This will allow you to exhaustively test all the scenarios and conditions to ensure that your code behaves appropriately. Testbenches also have the advantage of enabling you to plot intermediate wires and signals. This allows for a far greater level of debugging for your design than downloading it straight to the board. It is recommended that you verify your state machine's functionality before proceeding to download it to the board. It is good to plot the FSM state and next state values in the waveform for ease of understanding. **For this lab, you only need to do the behavioral simulation**.

To verify and demonstrate proper operation, **you must display the two final numbers of the two dice after a roll**. To do that, you will have to modify the above Verilog files accordingly to drive two more

7segment LEDs properly. Alternatively, you could display the binary value of the two dice using 6 of the green LEDs. Still, you will need to drive all four 7-segment LEDs for future labs, so if you do so now, you can reuse that code later. The best way to do this is to use another instance of clock divider and set its time constant parameter to a fast value such as 10 or 15. Use this fast clock to run a 2-bit counter which cycles 0-3, then write a combinatorial 2-to-4 decoder that takes this 2-bit value and creates the 4 LED select output signals, and chooses which value to put on the 7-segment outputs (which are common to all 4 LEDs) based on which LED is current. You should have an internal 7-bit value for each of the 4 LEDs and choose which of these to assign to the output pins based on the decoder value. By cycling the LED select fast enough, it appears to the human eye that all four LEDs are always on. So you end up with one clock divider running the game logic, and another running the LED selection logic. You can run the game logic slower but keep the LED selection fast this way.

Suppose you are having trouble getting your game to function correctly. In that case, you may want to bring the state machine current-state value out to the green LEDs as a binary value, so you can watch your game progress as you push the Rb button. If you show the sum of the two dice instead of the individual values on green LEDs, you should have room for both the roll total and the current state value. If you have the dice values on 7segment LEDs, then you have all 16 green LEDs to show internal values, another advantage of doing the 7-segment logic now.

SUMMARY -- During this lab, you learned general rules about how to write synthesizable code in Verilog, and you implemented and verified an electronic dice game

Lab Notebook deliverables:

- Brief description of what you did in the lab, what steps were followed
- Discussion of results:
  a. What were the test cases you used and discuss why you chose the set of test cases? **Explain how your test cases verify each of the FSM scenarios.**
  b. Attach a snippet of the reports of the tool.
     i. State transition diagrams and state table. (Refer to the description in the notes section)
     ii. Table and Simulation waveforms (**behavioral only**) showing all the test cases used above. All test cases must be visible in the waveforms; make sure the FSM state is visible in the waveform.
     iii. Schematic screenshot of synthesized design.
     iv. Utilization report: synthesis and implementation.
     v. Timing report.
     vi. Power report

  NOTE: Only attach relevant sections, not the entire report.
- Summary or Conclusion:
  a. Were all design goals met?
  b. What were the difficulties encountered?
  c. Any improvement you would like to make time permitting? Etc.…
- Verilog Source code:
  (Font: Courier, size 8pt, single line spacing, no paragraph spacing)
- All files must be labeled (ex. Figure 1: Simulation of eight-bit adder…). You should also highlight more interesting parts of your attachments.