Digital Design and Computer Architecture

J. Spjut

Lab 3: Adventure Game

Introduction

In this lab may you will design a **Finite State Machine** (FSM) that implements an adventure game! You will then enter the FSM into the Schematic Editor in Xilinx ISE Project Navigator, then simulate the game using ModelSim, and finally you can play your game using ModelSim.

Please read and follow the steps of this lab closely. Start early and ask questions if parts are confusing. It is much easier to get your design right the first time around than to make a mistake and spend large amounts of time hunting down the bug. As always, don't forget to read the entire lab and refer to the "What to Turn In" section at the end of this lab before you begin.

You will design your FSM using the systematic design approach described in Section 3.4.5 of the textbook.

1. Design

The adventure game that you will be designing has seven rooms and one object (a sword). The game begins in the Cave of Cacophony. To win the game, you must first proceed through the Twisty Tunnel and the Rapid River. From there, you will need to find a Vorpal Sword in the Secret Sword Stash. The sword will allow you to pass through the Dragon Den safely into Victory Vault (at which point you have won the game). If you enter the Dragon Den without the Vorpal Sword, you will be devoured by a dangerous dragon and pass into the Grievous Graveyard (where the game ends with you dead).

This game can be factored into two communicating state machines as described in Section 3.4.4. One state machine keeps track of which room you are in, while the other keeps track of whether you currently have the sword.

The Room FSM is shown in Figure 1. In this state machine, each state corresponds to a different room. Upon `reset`, the machine's state goes to the Cave of Cacophony. The player can move among the different rooms using the inputs `n`, `s`, `e`, or `w`. When in the Secret Sword Stash, the `sw` output from the Room FSM indicates to the Sword FSM that the player is finding the sword. When in the Dragon Den, signal `v`, asserted by the Sword FSM when the player has the Vorpal Sword, determines whether the next state will be Victory Vault or Grievous Graveyard; the player must not provide any directional inputs. When in Grievous Graveyard, the machine generates the `d` (dead) output, and on Victory Vault the machine asserts the `win` output.

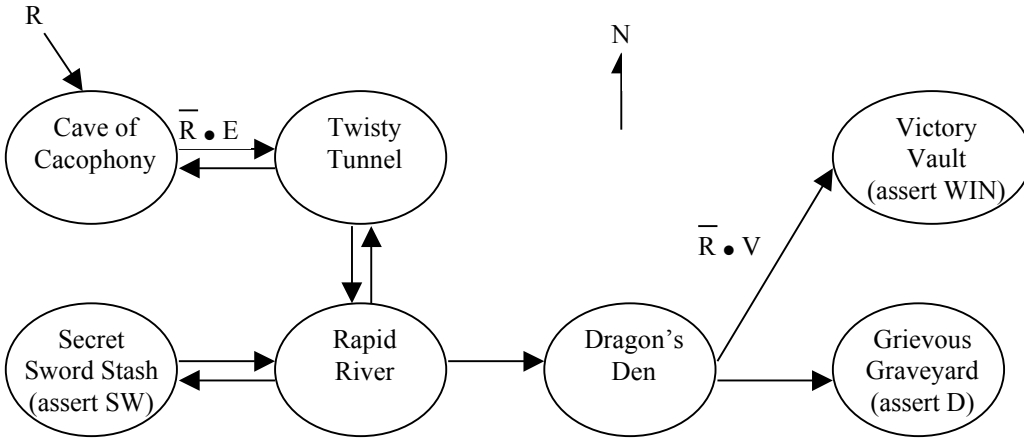

Figure 1. Partially Completed State Transition Diagram for Room FSM

In the Sword FSM (Figure 2), the states are "No Sword" and "Has Sword." Upon `reset`, the machine enters the "No Sword" state. Entering the Secret Sword Room causes the player to pick up a sword, so the transition to the "Has Sword" state is made when the `sw` input (an output of the Room FSM that indicates the player is in the Secret Sword Stash) is asserted. Once the "Has Sword" state is reached, the `v` (vorpal sword) output is asserted and the machine stays in that state until `reset`.

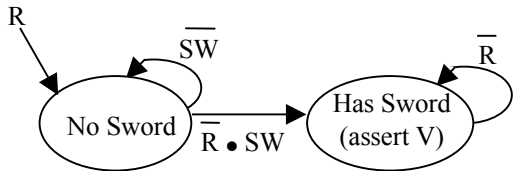

Figure 2. State Transition Diagram for Sword FSM

The state of each of these FSM's is stored using D flip-flops. Since flip-flops have a clock input, this means that there also must be a `clk` input to each FSM, which determines when the state transitions will occur.

So far, we have given an English description and a State Transition Diagram for each of the two FSM's. You may have noticed, however, that the diagram in Figure 1 is incomplete. Some of the transition arcs are labeled, while others are left blank.

Complete the State Transition Diagram for the Room FSM now by labeling all arcs so that the FSM operates as described.

The next step in the design process is to enumerate the inputs and outputs for each FSM. Figure 3 shows the inputs (on the left) and outputs (on the right) of the Room FSM and Figure 4 does this for the Sword FSM. Note that for navigational purposes the Room FSM should output `s0-s6`, indicating which of the seven rooms our hero is in. This is the last step of the design that will be given to you. Be sure to use input and output names (and capitalization) that exactly match these figures so that your design will play nicely with the testbench when you simulate later.

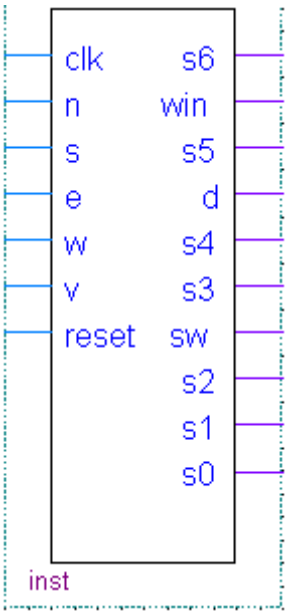

Figure 3. Symbol for Room FSM, showing its inputs and outputs

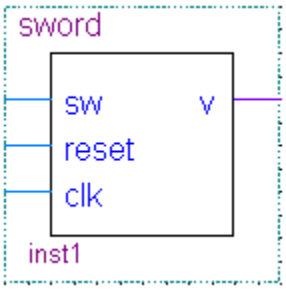

Figure 4. Symbol for Sword FSM, showing its inputs and outputs

Next, draw a state transition table for each FSM showing how the current state and inputs determine next state. The left side of the tables should have a column for the current state, and separate columns for each of the inputs. The right side should have a column for the next state. Also draw outputs tables, with the current state on the left, and the output(s) on the right. These tables are a way of representing the FSM's that is an alternative to the diagrams in Figure 3 and Figure 4.

On the left side of the table for the Room FSM, you do not need to fill in every possible combination of values for all inputs (that would make for a rather large number of rows

in your table!). Instead, for each state you only need to show the combinations of inputs for which there is an arc leaving that state in the state transition diagram. For example, when the input N is asserted and the current state is Twisty Tunnel, the behavior of the FSM is unspecified and thus does not need to be included in the table.[1] Also, you do not need to show rows in the table for what happens when more than one of the directional inputs is specified at once. You can assume that it is illegal for more than one of the *N*, *S*, *E*, and *W* inputs to be asserted simultaneously. Therefore, you can simplify your logic by making all the other directional inputs of a row "don't care" when one legal direction is asserted. By making careful use of "don't cares," your table need not contain more than a dozen rows.

The next step in FSM design is to determine how to encode the states. By this, we mean that each state needs to be assigned a unique combination of zeros and ones. Common choices include binary numeric encoding, one-hot encoding, or Gray encoding. A one-hot encoding is recommended for the Room FSM (i.e. Cave of Cacophony=0000001) and makes it trivial to output your current state `s0...s6`, but you are free to choose whichever encoding you think is best. Make a separate list of your state encodings for each FSM.

Now rewrite the table using the encoding that you chose. The only difference will be that the states will be listed as binary numbers instead of by name.

You are now approaching the heart of FSM design. Using your tables, you should be able to write down a separate Boolean logic equation for each output and for each bit of the next state (do this separately for each FSM). In your equations, you can represent the different bits of the state encoding using subscripts: S_1, S_2, etc. Depending on which state encoding you chose, a different number of bits will be required to represent the state of the FSM, and thus you will have a different number of equations. Simplify your equations where possible.

As you know, you can translate these equations into logic gates to implement your FSMs directly in hardware. That is what you will do in the next section.

2. Schematics

Start Quartus and create a new project named "lab03_xx" (where xx are your initials).

By now, you are familiar with the Schematic Editor. In this lab, however, you will learn how to create **hierarchical** schematic designs. In the same way that you can add symbols such as AND and OR gates to your schematic, you can add sub-components that are themselves specified by schematics. This creates a hierarchy of schematics.

Note that the flip-flop in the schematic editor is called DFF. It has asynchronous active-low set and reset inputs named PRN and CLRN. Both of these should be connected to VCC so that the element behaves as an ordinary flip-flop. (Yes, this gets pretty annoying…)

[1] Since the behavior of the FSM is unspecified in cases like this, the actual behavior of the FSM that you build in these cases is up to you. In a real system, it would be wise to do something reasonable when the user gives illegal inputs. In this game, we don't care what your game does when given bad inputs.

You will use a hierarchical design for your adventure game by doing the following:

1. Create a schematic for the Sword FSM. Name it sword.bdf.
2. Use File -> Create/Update -> Create Symbol Files for Current File to make a symbol for your schematic. Name it sword.bsf.
3. Create a schematic and symbol for the Room FSM.
4. Create a top-level schematic named lab3_xx.bdf. Place symbols for the Room and Sword FSMs. They are accessed using the Symbol Tool just like logic gates, but appear under your project tab at the top of the list of symbols. Wire these together. The inputs and outputs of your top level schematic will determine which signals will be available in the simulator when you play the game, so you should make sure to include at least `clk`, `reset`, `n`, `s`, `e`, and `w`, as inputs and the current room `s0-s6` as an output.

If you modify the inputs or outputs for a block that you have already created, regenerate the symbol file for the block. If the symbol has already been used in a higher level of the hierarchy, right click on the symbol and choose Update Symbol or Block… to update it with the modified symbol.

When you are done, generate Verilog files for each of your three schematics. (You can switch between schematics using the Files tab in Project Navigator). Inspect the files in a text editor to make sure they look reasonable.

3. Simulation

Now it is time to fire up your game in ModelSim and play it.

It would be possible to apply the inputs and clock signal using force statements as you did in the previous labs, but this becomes rather tedious. A better approach is to use a SystemVerilog testbench. Copy the testbench from

`\\charlie.hmc.edu\Courses\Engineering\E85\Labs\lab3die_tb.sv`

to your lab3_xx directory.

Create a new project in ModelSim and add all three Verilog modules generated from your schematics along with the testbench module.

Edit the `lab3die_tb.sv` testbench file. Study the file until you understand it. Lines beginning with `//` are comments. The file defines inputs and outputs of type "`logic`." It then instantiates (creates) the adventure game module. It assumes certain signal names; if they don't match your design, correct your design to match. Modify the name of the module to match your initials. It uses a `forever` statement to generate a clock with a period of 100 units (nanoseconds). For example #50 indicates a delay of 50 nanoseconds. It then uses another `initial` block to apply the inputs every cycle. The inputs take the poor player straight to the dragon's den to meet a hideous fate.

Compile all of the modules and start the simulation on the testbench. Be sure you have unchecked the "Enable optimization" box in the Start Simulation dialog or the signals may not appear in the Objects window. Add all of the inputs and outputs to the wave window. Also, go into your lab3_xx module (in the upper left pane) and add the `sw` and `v` signals. Type `run 800`. The testbench will automatically create the clock and apply the inputs so you don't need to type force statements. Check that the player goes through the expected states and then dies. Print the results.

If this doesn't work, debug your design. If you make changes to the inputs or outputs of a block, be sure to regenerate the symbol and update the blocks where the symbol is used.

Save a copy of the testbench in case you need to refer to it later. Modify the testbench so that the player first fetches the vorpal sword before confronting the dragon. Recompile all the files. Type `restart -f` in the transcript window to restart the simulation without having to add all the waves. Run for 800 ns again. If all goes well, you can celebrate your victory.

What to Turn In

You must submit an electronic copy of the following items (in the correct order, and each part clearly labeled) via Sakai. Submit to the Lab 3 Assignment. These should all be included in a single file (.pdf).

1. Please indicate how many hours you spent on this lab**.** This will be helpful for calibrating the workload for next time the course is taught
2. A completed State Transition Diagram for the "Room" FSM. Scanned, hand drawn diagrams are acceptable so long as they are neat and clearly readable.
3. Your tables listing (1) next state in terms of current state and inputs and (2) output in terms of current state. You need tables for each FSM.
4. A list (one for each FSM) of your binary encoding for each state.
5. The revised copy of your tables, using your binary encoding.
6. Your Boolean logic equations for the outputs and each bit of the next state in terms of the previous state and inputs.
7. An image of your schematics for both the "Room" and "Sword" FSM's.
8. An image of your schematic for the game (built by connecting both FSM).
9. Two images of your simulation waveforms: one that shows you playing the game and winning (entering "Victory Vault"), and another that shows an example of losing the game (entering the "Grievous Graveyard"). **Your signals must be printed in the following order: `clk`, `n`, `s`, `e`, `w`, `r`, `win`, `d`, $s_6 \ldots s_0$, `sw`, `v`.** Please place the images in a landscape orientation so that they fit better on the page.
10. EXTRA CREDIT: It is a little known fact that the Twisty Tunnel is located beneath the dining commons and that by heading north one can reach the dormitories. Extend your adventure game with more interesting rooms or objects. There will be a prize for the most interesting working enhancement!