ELE 206/COS 306 Lab 5: Simon – RTL Design Project

Checkpoint 1 Due Date: November 4th, 2020
Checkpoint 2 Due Date: November 11th, 2020

The purpose of the checkpoints is to help you stay on track to complete the project in time. There will be some credit associated with submitting the checkpoints on time. If the TAs notice glaring issues with your design, they will point these out to you, but otherwise individual feedback or comments will not be provided. of course, if you are unsure about your design, we encourage you to ask the TAs for feedback during lab sessions.

Demonstration Required.

Tutorial Reading: Section 5.2
The lab closely follows the design process outlined in this reading.

Introduction

This lab is a medium-sized project intended to teach you the basics of register-transfer level (RTL) design, datapath construction, and modular testing. This project is significantly more involved than previous labs, so it's best to get started early. Note that there are weekly due dates for portions of the lab, although the only demonstration is for the final product.

Prior to beginning the lab, it is important to closely read section 5.2 of the Verilog tutorial. This section works through an example RTL design problem with example code, and it will serve as a good reference point for you as you work through this lab. As an example, we have also provided a Verilog implementation of the soda dispenser machine, which is explained in Example 5.3 page 257 of the textbook. You may download the corresponding files from Blackboard (*soda_dispenser.zip*). It is helpful to look at the HLSM, controller, and datapath implementation of the soda dispenser machine before working on this lab.

Your task for this lab is to implement a datapath and control circuit for a two-player version of Simon, a popular electronic toy designed to test the memory of its players. You can read more about the game here: `https://en.wikipedia.org/wiki/Simon_(game)`.

Essentially, the two-player version of Simon works as follows. Alice, who is player one, resets the game. Then, she enters a pattern (in our case, a 4-bit number). The machine plays back that pattern to Bob, who is her opponent. Bob then enters the pattern he saw. If he gets it right, he now gets to add his own pattern to the pattern Alice entered, forming a sequence. The machine will play back the entire sequence (Alice's pattern followed by Bob's pattern) to Alice, who then has to repeat the entire sequence of patterns. If she succeeds, she gets to add another pattern to the sequence. The game goes on until one of the players makes a mistake while repeating the pattern sequence, which keeps getting longer and harder to remember. Once one player makes a mistake, the other player wins.

You will test your circuit thoroughly via simulation.

To begin the lab, download and unzip *simon.zip* from Blackboard, and follow the instructions in this document. All of the source code for this lab, including testbenches, is stored in the */simon* folder.

A Multiplayer Version of Simon

Inputs and Outputs

Your version of the Simon machine will have the following inputs:

- `pattern [3:0]` – Four switches that can be used to form a pattern.
- `level` – A single switch that controls the difficulty level of the game.
- `rst` – A button for synchronously resetting the game.
- `pclk` – A button for advancing the state of the game. Essentially, this serves as the clock for your system.

Additionally, your machine will have the following outputs:

- `pattern_leds [3:0]` – Four LED's for either showing the current pattern or replaying a pattern sequence.
- `mode_leds [2:0]` – Three LED's that show the current mode/state of the game.

Gameplay

The game begins when the circuit is reset. To reset Simon, the `rst` button must be held down while the `pclk` button is tapped.

When the circuit is reset, two things happen.

First, the current state of the `level` switch is saved for use throughout the current game. If `level` is one, the game is in hard mode, and patterns can be any 4-bit number. If `level` is zero, the game is in easy mode, which means that patterns must have one – and only one – bit set to one. This leaves only four options: `4'b0001`, `4'b0010`, `4'b0100`, or `4'b1000`.

Secondly, the game transitions into input mode (the "initial state" for your machine). Gameplay proceeds by transitioning between four modes – input, playback, repeat, and done. Each of these modes is described in detail below.

- **INPUT** – This is your "initial" ("init") state. In this mode, whichever player is currently controlling the game gets to add a new pattern to the game's sequence. The player first creates a *legal* pattern using the four `pattern` switches, and then presses `pclk` to enter and save the pattern.

  As long as the pattern was legal, the game will transition to the playback mode. Otherwise, the game will wait in the input mode and not accept the new pattern. Legality is determined by the `level` switch's value at the game's start. Changing the level switch during a game should not affect anything.

  Finally, the `pattern_leds` should display whatever the switches are currently set to. If a `pattern` switch is flipped, the corresponding LED should change instantaneously.

- **PLAYBACK** – In this mode, the other player gets to view the sequence of patterns for the current game. Whenever `pclk` is pressed to transition from input to playback, the `pattern_leds` should show the first pattern in the game's sequence.

For each subsequent press of `pclk`, the `pattern_leds` should update to display the next pattern in the sequence. When the currently-stored sequence has shown its last value, the next clock press will take the game to the repeat mode.

- **REPEAT** – In this mode, the player that just viewed a full playback of the game's pattern sequence attempts to re-enter that sequence exactly. The player forms the first pattern they remember on the `pattern` switches and then taps `pclk` to register their guess.

  If the player's guess was correct and there are still patterns left in the sequence, the game remains in repeat mode and the player continues to make guesses to match the sequence. If a player guesses correctly up to (and including) the final pattern in the current sequence, the game transitions back to input mode for that player to append a new pattern to the sequence.

  However, as soon as a player enters an incorrect guess, the game will immediately transition to the done mode.

  As in the input mode, the `pattern_leds` should display whatever the `pattern` switches are currently set to.

- **DONE** – In this mode, the game has ended. The stored patterns can be cycled through by pressing `pclk`, just like in the playback mode. This allows the player that lost to view the correct sequence and see what error they made. When the done mode is first entered, the `pattern_leds` should display the *first* pattern in the sequence.

  If `pclk` is pressed enough to get to the last element in the sequence, the next push of `pclk` should cycle back around to the first element for another run through the sequence. The only way to escape the done mode is to reset the game.

The current mode is shown to players via the `mode_leds`. A table mapping the current mode to the status of the `mode_leds` is shown below.

Mode	`mode_leds`
Input	`3'b001`
Playback	`3'b010`
Repeat	`3'b100`
Done	`3'b111`

Example Game

Here's an example run-through of a game that should hopefully illustrate how Simon works in practice.

1. **Mode: Unknown** – The game is reset by holding down the `rst` button and tapping `pclk`. The `level` switch was set to zero, so this game is on easy mode.

2. **Mode: Input** – Alice enters `4'b0001` on the `pattern` switches, then taps `pclk`.

3. **Mode: Playback** – Alice hands the game to Bob. The `pattern_leds` display `4'b0001`. Bob notes this, and taps `pclk`.

4. **Mode: Repeat** – Bob enters `4'b0001` on the switches and then taps `pclk`. This entry was correct, so now it's Bob's turn.

5. **Mode: Input** – Bob flicks the `level` switch to one, intending to enter a more difficult pattern. He then enters `4'b1010` on the switches and taps `pclk`.

6. **Mode: Input** – Modifying the `level` switch during the game does not change the game's difficulty – the level was locked in at zero when the game was reset. Because Bob's pattern was illegal at level 0, the game remains in input mode. Bob enters `4'b1000` and taps `pclk`, successfully adding a legal pattern.

7. **Mode: Playback** – Bob hands Alice the game. The `pattern_leds` display `4'b0001` – Alice's original pattern. Alice clicks `pclk`.

8. **Mode: Playback** – The `pattern_leds` now display `4'b1000` – Bob's new pattern. Alice taps `pclk`.

9. **Mode: Repeat** – Alice enters `4'b0001` on the switches, then taps `pclk`. This entry was correct, but the sequence isn't complete.

10. **Mode: Repeat** – Alice makes a mistake, entering `4'b0100` on the switches and tapping `pclk`.

11. **Mode: Done** – The game is over, and Bob has won. The LED's display `4'b0001`. Alice presses `pclk`.

12. **Mode: Done** – The LED's display `4'b1000`. Alice realizes her mistake, but taps `pclk` one more time.

13. **Mode: Done** – The LED's display `4'b0001`, wrapping around to the first entry.

Technical Specifications

Here's a list of additional technical specifications that must be met by your design:

- **Number of Patterns to Store** – The game must be able to store a sequence of up to 64 four-bit patterns. Most people lose the game well before this limit is reached.

  If the players reach a sequence length of 64 patterns, the ideal behavior would be to begin wrapping around. The first pattern would be deleted and replaced with a new pattern in the next input mode, and the second pattern would become the first in the new sequence. When another pattern is added, the second pattern would be overwritten, and the third pattern would become the new starting point – and so on. The sequence would remain 64 patterns long in perpetuity until someone loses, with the starting point sliding along one pattern at a time.

  However, for the purposes of this lab, you do **not** have to handle the wrap-around case. If you do and can demonstrate that to the TAs, you will receive a small amount of extra credit. It shouldn't be much more code to handle this case, but it does require a bit more thinking.

- **Synchronous Resets and Transitions** – Resets are synchronous for Simon. Additionally, your internal FSM and the mode of the game should only transition when `pclk` is pressed.

- **Clock Edges** – Assume that all transitions will be on the rising clock edge, and write your Verilog as such.

Week 1: Datapath and Controller Design

Checkpoint 1 Due Date: November 4th, 2020

The first step in tackling the Simon problem is creating a high-level overview of your design. Following the example set forward in the Verilog tutorial, your high-level design process will follow three main steps:

1. Create a High-Level State Machine (HLSM) that describes the behavior you expect from your game.
2. Create a Datapath. Extract the complex actions, local storage, and other items in your HLSM that cannot be handled by an FSM and create logic to handle them in your datapath.
3. Create a Controller. Re-write your HLSM into a simple FSM that will utilize your datapath (via control signals) to produce the behavior you want.

By the due date of checkpoint 1, you should have an HLSM, a datapath, and a controller designed. At this phase, the design should be done via technical diagrams. If you draw your components by hand, take legible pictures of them for your checkpoint submission.

The requirements for each of your design's components are listed below.

High-Level State Machine

Your HLSM should look a lot like an FSM, with states, transitions, and transition conditions. However, the HLSM can specify manipulation of local storage, arithmetic operations, and other complex tasks to be performed in each state, and its transition conditions can be complicated. For example, an HLSM might specify that it will remain in a certain state, adding one to a local counter register, and only transition once that counter reaches a certain value. Feel free to write if else statements in your states as needed. This means your output values can depend on both the current state and the current inputs (Mealy machine). You are also allowed to have an "is_legal" high-level module which produces a 1 if the pattern is a legal easy-mode pattern.

An HLSM should fully capture the desired behavior of your entire circuit. You may draw it however you like, but please follow these guidelines:

1. Use the rising clock edge for all transitions, and do *not* AND the rising clock edge with every transition condition. All transitions in your machine will be taken synchronously with the clock edge, so just make that implicit.

2. Ensure that the values for the outputs of your circuit (`pattern_leds` and `mode_leds`) are clearly indicated for each HLSM state. You can do this in place or in a separate table – whatever is cleanest for your drawing.

3. Don't draw explicit reset transitions from all states to your initial state. Just mark your initial state clearly to save space on your diagram. As usual, leave `rst` out of your transition conditions as well.

4. However, feel free to use `rst` in the action/output logic within your states as needed – just don't use it for transition conditions.

Once you have an HLSM design that you think will work, you can move on to creating a datapath!

A Datapath

To create your datapath, first identify all of the local storage, complex actions, and complex conditions in your HLSM. Analyze what sort of datapath components (i.e. adders, comparators, registers, memories) you might need, and then figure out how to connect those components together to make the datapath work.

The inputs to the datapath should be simple control signals that your control module will drive or external inputs (such as the `pattern` switches). The outputs from your datapath will be Boolean indicator signals (like comparator outputs) that your controller will need for transitions. Additionally, your datapath may directly drive external outputs (such as `pattern_leds`).

One additional note – you will be provided a 64-entry 4-bit memory which you **must** use as a component in your datapath. The memory unit is illustrated in Figure 5.1, and it has the following ports:

- `clk` – The clock for the memory module.

- `rst` – A synchronous reset signal that writes zeros to all entries in the memory on a positive clock edge.

- `r_addr` – The entry to be read.

- `r_data` – The output containing the data stored at `r_addr`. This output is asynchronous – as soon as `r_addr` is updated, `r_data` will reflect that change.
- `w_addr` – The entry to be written.
- `w_data` – The data to be written to `w_addr`.
- `w_en` – The write-enable signal. When this signal is high, data will be written from `w_data` to `w_addr`'s entry on a positive clock edge. Writes are synchronous, and will *only* happen on clock edges.

Note that you do *not* have to use the `rst` signal for this memory module if you don't want to. If you choose not to use it, simply wire it to zero in your diagram.

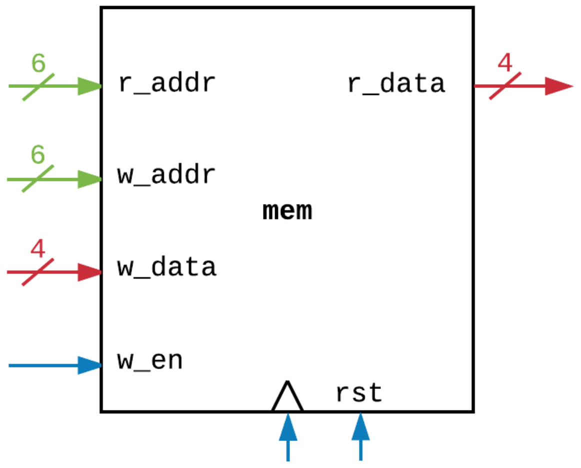

Figure 5.1: Provided 64-entry 4-bit Memory

Draw your datapath as you see fit, but please follow these guidelines:

1. Feel free to use high-level modules like adders, comparators, multi-function registers, and the like. We do **not** want a gate-level description of all of these components – just make assumptions about the control signals available and mark your modules so that it's clear what they do. When in doubt, keep it simple!

2. Clearly label the bit-width of every wire in your design.
3. Clearly indicate the inputs and the outputs of your datapath, and provide names for each. Signal names should be simple but descriptive enough to explain what they do. For example, a control signal that resets a counting register to zero could be labeled with `count_rst`.

Once you have a datapath design that you believe is complete, save a copy of your design and move on to FSM creation!

A Controller

Your final task for preliminary design work is to convert your HLSM into a simple FSM that will serve as the control unit of your circuit. You can do this with a few easy steps:

1. Replace complex actions in your states with simple output logic to manipulate datapath control signals.
2. Replace complex conditions on your transitions and in your output logic with simple Boolean indicators from the datapath's comparator outputs.

Once you perform these steps, your HLSM will be reduced to a simple FSM. Draw that FSM, following the guidelines listed below:

1. For each state, list the controller outputs that are on (active) during that state. If an output does not appear at all in a state's output list, then the output is understood to be set to zero for that state.
2. Feel free to create a Mealy machine, where the outputs in a state can also be affected by the current inputs. For example, you might want different outputs in the same state based on whether `rst` is high or not.
3. Once again, leave `rst` out of all of your transition conditions, and simply mark your initial state clearly.
4. Finally, all transition conditions should be simple Boolean logic statements (ORs, ANDs, and NOTs) based on simple signals. Complex conditions, like numerical comparisons or legality checking for input patterns, should be calculated in the datapath and compressed to simple Boolean indicator signals that can be read by the controller.

Once your FSM design is complete, save yourself a copy of it and move on to submission.

Demonstration

Demonstration for this checkpoint is not required. However, we encourage you to visit one of the lab help sessions and show your HLSM, controller, and datapath to one of the TAs. If the TA notices a glaring error with your design, they will point it out to you. But otherwise, individual feedback or comments will not be provided as it gets very hard to dive into everyone's design. Of course, you should feel free to request for more feedback.

Write-up and Submission

Submit your HLSM, datapath, and controller FSM designs as a single PDF file on Gradescope – if you handwrote them, take pictures and save them as a PDF using a document editor of your choice. Name your PDF "*ELE206_Simon_c1_netid.pdf*", with *netid* replaced with your Princeton NetID.

Congratulations! You've finished work for week one. It's a good idea to start the next part early, as it may take longer to complete!

Week 2: Datapath and Controller Implementation

Checkpoint 2 Due Date: November 11th, 2020

Now that you have a design hammered out for your datapath and controller, you'll implement them in Verilog and test them individually.

Datapath Implementation

Let's begin with the datapath – open up *SimonDatapath.v*, which has been started already for you. You'll notice that the first three inputs for the datapath have been declared for you:

```verilog
module SimonDatapath(
  // External Inputs
  input        clk,           // Clock
  input        level,         // Switch for setting level
  input  [3:0] pattern,       // Switches for creating pattern
  //...
);
```

Add control signal inputs, Boolean indicator outputs, and external outputs as needed. Then, get started on creating your internal datapath logic. You have total freedom with regard to how you implement your datapath – just remember to use the provided memory unit, which has already been included in the provided code.

Controller Implementation

Once your datapath is completed, you should implement your controller FSM. This part should be fairly familiar to you – it's essentially the same process that you performed in Lab 4! The code for the controller should be written in *SimonControl.v*. Once again, some code has been supplied for you to help you get started. Feel free to modify it as you wish.

Modular Testing

Once both your datapath and controller have been completed, it's time to test them out. Instead of immediately wiring them together, we're going to test these components individually.

Modular design is a very important way to make comprehensible and reusable circuits, and modular testing makes debugging a large system much easier. Instead of trying to debug the entire Simon game as a monolithic circuit, we're going to test its primary components individually. This will detect and remove simple bugs that would be much harder to discover and fix once the whole system comes together.

In *SimonDatapath.t.v* and *SimonControl.t.v*, write simple testbenches for each of your modules. You should create several test cases for each module – the tests do not have to cover every possible case, but they should at least simulate some common behaviors that you expect during gameplay.

For example, you'll probably want to test your datapath to ensure that you can write a pattern into memory and read it back out. You'll also want to test your controller, ensuring that it transitions states as you expect based on the Boolean indicators it will eventually receive from the datapath.

We've included a few helpful testing macros in both files for you to use. Feel free to use them or ignore them – they're just there for your convenience.

Once you've completed your testbenches, compile them with the following commands:

```
iverilog -g2005 -Wall -Wno-timescale -o SimonControlTest SimonControl.t.v
iverilog -g2005 -Wall -Wno-timescale -o SimonDatapathTest SimonDatapath.t.v
```

Fix any compiler errors or warnings, and then execute your tests with the following commands:

```
vvp SimonControlTest
vvp SimonDatapathTest
```

Ensure that there are no failures. You'll see some warnings that look like this when you compile the datapath tests:

```
VCD warning: array word SimonDatapathTest.dpath.mem.mem[0] will conflict with
    an escaped identifier.
```

Just ignore those warnings throughout the rest of the lab – they're caused by dumping out array contents (in this case, your memory) to a VCD file.

If you need waveforms for debugging, they will be dumped out as *SimonControlTest.vcd* and *SimonDatapathTest.vcd*. You can open them with these commands (use one at a time):

```
gtkwave SimonControlTest.vcd
gtkwave SimonDatapathTest.vcd
```

Write-up Question 1:
Briefly explain the test cases you wrote for your testbenches. What behaviors do your test cases verify, and why did you focus on those cases?

Write-up Question 2:
Did you have to modify your design from Week 1? If so, what did you change?

Module Combination and Testing

Now that your individual controller and datapath modules have been written and tested, you will combine them to form the full Simon circuit. Then, you will run the game against a basic testbench written by the TAs and add your own testbench.

Connecting the Datapath and Control Units

This part should be fairly easy – just open up *Simon.v* and add wires for all of the connections between the datapath and control module. Then, connect the two modules by adding those wires to the ports you created in *SimonDatapath.v* and *SimonControl.v*. The module instances are already declared for you – just add to them!

Please do not modify the Simon module's port list. It is needed for the TA testbench to work correctly.

Running the TA Testbench

There is a basic testbench in *SimonTA.t.v* that actually simulates the basic gameplay sequence described at the beginning of this lab manual. To run it against your code, just run the following command:

```
iverilog -g2005 -Wall -Wno-timescale -o SimonTATest SimonTA.t.v
```

Fix any compiler errors or warnings, and then execute the tests with the following command:

```
vvp SimonTATest
```

Ensure that there are no failures. If you need waveforms for debugging, they will be dumped out as *SimonTATest.vcd*. You can open them with this command:

```
gtkwave SimonTATest.vcd
```

Writing Your Own Testbench

The TA testbench only tests a subset of Simon's functionality. For example, it only performs tests where `level` was set to zero when the game was reset. Your task is to write functional tests that verify Simon's behavior under a wider set of conditions.

Write your code in *Simon.t.v*. Feel free to model your code on the code in *SimonTA.t.v* – we've even included the macros we used there in your testbench for your convenience!

Once your testbench is completed, run your tests with the following command:

```
iverilog -g2005 -Wall -Wno-timescale -o SimonTest Simon.t.v
```

Fix any compiler errors or warnings, and then execute the tests with the following command:

```
vvp SimonTest
```

Ensure that there are no failures. If you need waveforms for debugging, they will be dumped out as *SimonTest.vcd*. You can open them with this command:

```
gtkwave SimonTest.vcd
```

Write-up Question 3:
Explain your functional tests for the overall Simon module. What use cases do they test, and how do they work?

(OPTIONAL) Verification with VCD Viewer

Similar to Lab4, we have created a VCD viewer for the VCD files that your Simon code produces. Please note that this part is completely optional. You may use this to verify that your code is working properly. Here are the usage instructions:

- Go to `https://verilog-simon-viewer.vercel.app/`
- Make sure you have your final VCD file, e.g. *SimonTATest.vcd*. Click on "Choose file" and select your VCD file.
- Select your signals (you can now also type it in) for **level**, **rst**, **mode_leds**[**2:0**], **pattern_leds**[**3:0**], **pattern**[**3:0**] using the "Set Me" drop down menus.

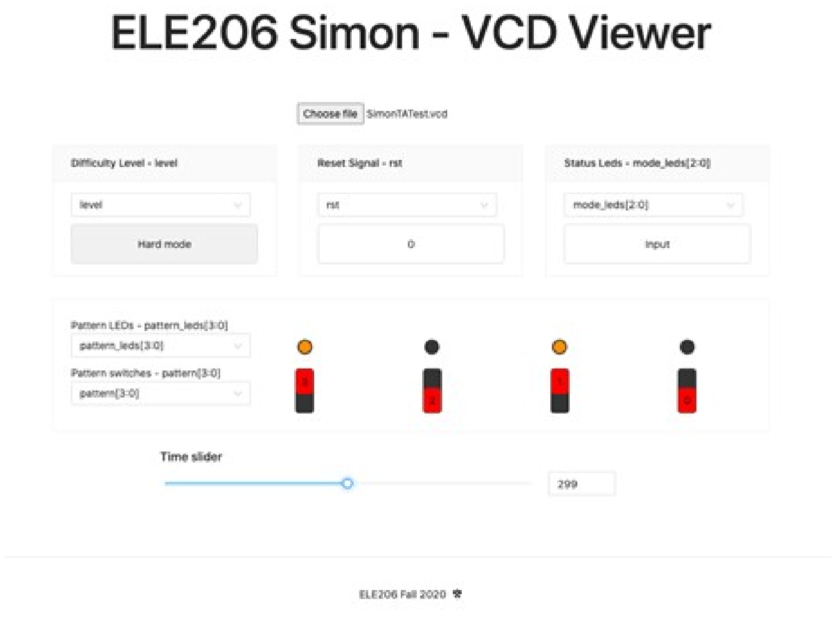

Figure 5.2: Sample output of the VCD viewer.

- Use the “Time Slider” to check your signals are changing correctly. Your viewer should look like Fig. 5.2 after the above steps (the name of your vcd file and time may be different).

Demonstration [85 pts]

For your demonstration this week, visit one of the lab help sessions on Zoom to show a graduate TA your GTKWave outputs. Show them that your level switch works correctly and that all modes perform as expected. If you work as a team, both students are expected to be present at the demo.

Write-up and Submission [15 pts]

In your write-up for this week, answer the numbered questions from the text in this section briefly but completely. Feel free to create the write-up in whatever program you want to use, but save and submit it as a PDF. Name your PDF *ELE206_Simon_c2_netid.pdf*, with *netid* replaced with your Princeton NetID. Finally, upload and submit your PDF file to Gradescope.