18-244/644: Intro to ASIC and FPGA Chip Design

EX0: Shake the Rust Off

Objective and Overview

The purpose of EX0 is to reload your SystemVerilog and Digital Design skills into your mental cache, to remind yourself of how it all works. And, should you discover you have some trouble with this exercise, to give you the opportunity to get some help, to re-work your skills, or to learn something new.

In this exercise, you will design a simple hardware thread and experiment with SystemVerilog arrays. Then, you will use **EDA Playground** as a SystemVerilog development tool. Im not sure Id actually do any serious development here (though, you probably could). But, it is a fantastic way to drop a dozen lines of code to check out if a feature works the way you think it should. Very minimal setup or wasted effort!

Schedule and Scoring

This exercise is due on **18 Jan, 5:00 PM**. You are expected to begin working on it during the class period.

This exercise is worth a total of 100 points

Active Participation	20 points
Task #1: Hardware Thread	20 points
Task #2: EDA Playground	20 points
Task #3: Array-zing Registers	20 points
Task #4: Jumping Off	20 points

Active Participation means that you are physically and mentally present for the entire class period, working on the exercise. You don't need to have everything completed by the end of class: mind the due date at the top of this section.

In this exercise, you will need to write SystemVerilog code and submit a lab report document. Your lab report should be a PDF file named `{your_andrewID}_EX0.pdf` and should be committed in your GitHub repository for EX0.

Deliverables for this exercise include SystemVerilog code and a lab report document, all pushed into the repo. **PLEASE** check all **"For Credit:"** checkpoints before submitting!!!

A Note about Collaboration

EX0 is to be accomplished individually. All work must be your own. You may ask other students for general assistance, but you may not copy their work.

You are encouraged to talk with other students and discuss the things you've discovered. Reach out to each other if you have a bug or problem. But, don't end up copying their work or having them do work for you. Remember, everything you turn in must be your own work.

Task #1: Hardware Thread

We will use a simple hardware thread as an example for future exercises and discussions this semester. This is where it starts.

Design a hardware thread that will be used to determine the range between the maximum and minimum of a series of numbers. On the clock edge where `go` is asserted and every clock edge after, up until (and including) the edge where `finish` is asserted, your hardware thread will take a look at the `data_in` value. Of all of such values, determine which is the largest and which is the smallest (these are unsigned values). Output on `range` the difference between the largest and smallest.

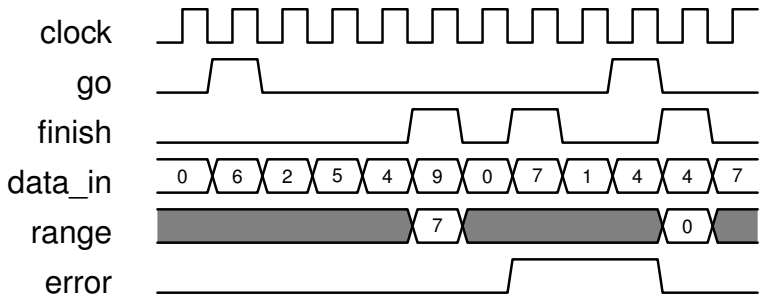
Here is the module header:

```
module RangeFinder
  #(parameter WIDTH=16)
   (input  logic [WIDTH-1:0] data_in,
    input  logic             clock, reset,
    input  logic             go, finish,
    output logic [WIDTH-1:0] range,
    output logic             debug_error);
```

A few details:

- All signals (`go`, `finish`, `data_in`) are sampled synchronously by the design. The `reset` signal is asynchronous.
- It would be an error for `go` and `finish` to be asserted at the same time. If that happens, ignore any inputs until the next `go`
- It would also be an error for `finish` to occur before `go`. Handle it the same way.
- It would also be an error for `go` to occur a second time before a `finish`. Handle it the same way.
- When `go` is first asserted, the value will be on `data_in` at the same clock edge.
- The `range` value must be correct on the clock edge where `finish` is asserted. At all other times, we don't care what `range` is.
- Feel free to use your library file from 18-240.

Looks pretty easy, eh? There is a testbench in the GitHub repo to ensure youre thinking of this the same way we are. Heres a waveform to illustrate:



For credit: Write your implementation of module `RangeFinder` inside the file `task1/RangeFinder.sv`. For full credit, your module must pass the testbench found inside `task1/RangeFinder_test.sv`. You'll test your module in the next task.

Task #2: EDA Playground

EDA Playground is a great web-based tool that lets you simulate and synthesize your code using a variety of different software, some of it proprietary. It is run by Doulos, a company that specializes in training for EDA engineers. Im sure EDA Playground is useful for their classes, and Im grateful that they make it public for the rest of us to mess around with.

Open EDA Playground in a web browser. Make an account with your school email before proceeding further.

The code for any one "system" is known as a *playground*. This website will save playgrounds for you, as long as you have an account. Please make the playgrounds you use for academic work to be private, just so we dont have an AIV incident.

Each playground starts off with two blank editor windows, one for a design and one for a testbench. You can see the default filenames at the top of each pane. On the left is a list of options. There you can choose your tools, libraries and some other configuration choices. Even though **Yosys** is on the list, we wont be using it here today. Instead, choose **Synopsis VCS** as your simulation tool.

Paste **RangeFinder** from Task #1 and punch that blue "Run" button on the top menu bar and check if it simulates properly. If so, you should not see any messages printed. There is a `$monitor` statement, but it is commented out in the code I distributed.

On the left is a checkbox next to **Open EPWave after run**. What does that do? Try it. It wont work.

Youll get an error message that will look a bit cryptic. In order to understand, read the spec yes, open up the PDF on Canvas that has the SV Specification, IEEE 1800-2023. Searching on some of the words in the error message will bring you to an explanation of what a VCD file is and give you some hint of how to use it. But, it turns out that EDA Playgrounds help file is quicker and to the point. Im not going to hold your hand on this, figure out how to get the wave viewer working, as it is pretty useful. Oh, and note what page you found the definition of a VCD file in the spec, as Im gonna ask for it from you.

For credit: Inside your lab report document, include the following:

1. A short paragraph documenting your experience with EDA Playground.
2. The page and section in the SV spec where the definition of a VCD file exists.
3. A screenshot of the text output of the simulator with the `$monitor` statement turned on. If you finish this task during the class period, you can get checked off by a TA. In that case, write **"Task Checked Off"** in your report.
4. A screenshot of the EPWave viewer. If you finish this task during the class period, you can get checked off by a TA. In that case, write **"Task Checked Off"** in your report.

Task #3: Array-zing Registers

This task will introduce you (or re-introduce you) to SystemVerilog arrays. For lots more details, including information about dynamic arrays, take a look at the 18-240/341 textbook, chapter 13.

In this task, you will design a simple Register File by using 2D arrays in SystemVerilog.

Part A: SystemVerilog Arrays, a Quick Overview

You've been using arrays already. Consider the following array, which the SystemVerilog specification calls a *vector*:

```
logic [15:0] some_signal;
```

This is a regular 16-bit signal. But, it is also an array of single-bit signals. By writing `some_signal[5]`, for example, you can access the value of the 6th bit of the signal. The least-significant bit (that is, the rightmost) is `some_signal[0]`.

But, what if we want a 2-dimensional array? Simple:

```
logic [7:0][15:0] some_array;
```

This is also an array. It has 8 entries and each entry is a 16-bit signal. By writing `some_array[0]` you can access the first entry of the array, a 16-bit value. By writing `some_array[0][3]`, you can access the 4th bit of the first entry of the array, a single-bit value.

This array is known as a *packed* array. The actual implementation of this array is a $8 \times 16 = 128$ bit vector. All the bits are arranged in one physically contiguous vector. The difference between `some_array` and `logic [127:0] some_vector` is that you can more easily access the subfields of `some_array`. After all `some_array[4]` is an 8-bit value and is a bit easier to think of than `some_vector[39:32]`.

SystemVerilog also has an additional way to define a 2-dimensional array. This is an *unpacked* array:

```
logic [15:0] unpacked_array[8];
```

In this example, `unpacked_array` represents eight values, each of which is 16-bits in size. Thus, it is very similar to `some_array`. From a synthesis perspective, an unpacked array is intended to be accessed in an element-by-element basis. But, practically, the hardware for both packed and unpacked arrays is very similar.

SystemVerilog allows for an arbitrary number of dimensions, either packed or unpacked or both. So, this is legal (though probably overkill):

```
logic [15:0][8:0][4:0][12:0] overkill_array[8][1:10][20];
```

Today, let's explore 2D arrays. Remember that the left square brackets indicate the number of entries in the array, and the right brackets indicate the bit width of each entry.

Part B: Register File

With that in mind, you will design a 1-read/write register file (RF) using a 2D array. The module header for the RF is shown below:

```
module RegisterFile
   (input  logic [7:0] din,
    input  logic [3:0] addr,
    input  logic clock, reset_L,
    input  logic read, write,
    output logic [7:0] dout,
    output logic error);
```

Your `RegisterFile` will have 16 entries, each entry being 8-bits wide. You can select an entry to read from or write to using input `addr`.

If `write` is asserted at a given clock edge, the selected entry will be written with the value of input `din` at that clock edge. If `read` is asserted, the stored value of the selected entry should appear in output `dout` on the same clock edge.

The RF can support only a read or a write at a given time. If both `read` and `write` are asserted at the same time, `error` is asserted for that clock cycle. Moreover, if an entry is read **without having been written to first** (since the last reset), `error` is also asserted for that clock cycle. On any clock cycle when `error` is asserted, `dout` is set to zero and the internal state of the RF does not change.

Lastly, `reset_L` is asynchronous, active-low, and wipes the contents of the RF to all zeros.

For credit: Write your implementation of module `RegisterFile` inside file `task3/RegisterFile.sv` and commit it to your repository. For full credit, your module must pass the testbench found inside `task3/RegisterFile_test.sv`.

Task #4: Jumping Off

A couple of exercises feature a **Jumping Off** section. This is the first one. Their purpose is to invoke a sense of curiosity about the tools introduced during the exercise. As a result, Jumping Off sections are fairly open-ended.

There aren't specific insturctions for this task. Instead, we want you to explore and play around with the tools youve seen today. Here's a couple of things you could try out:

- Try out other simulators on EDA Playground
- Explore more around the EPWave viewer.
- Explore the SV spec and experiment with some interesting feature you discovered.

If you would like a little more SystemVerilog design practice, you could also delve into online resources such as HDLBits or ChipDev which offer neat collections of byte-sized digital circuit design questions. Who knows, they might come in handy for interview prep ;P

Regardless, spend at least 30 minutes in concentrated effort with some focus on exploring. Have fun!

> **For credit:** In your lab report, submit a paragraph describing what you explored, what you expected, and what you found. Prof. Nace would be especially proud of you if you found out something he didnt know and described it well enough here to teach him how it works :) Please include any relevant links / images / graphs necessary.

Turning Stuff In

To submit this exercise, follow these steps:

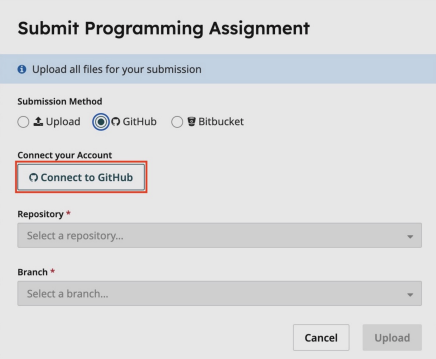
1) Create a branch of the commit you want to submit, naming it as your final commit and push it to GitHub. Tag it as well.

```
$ git branch gradescope_final
$ git checkout gradescope_final
$ git add --all
$ git commit -m "My final submission.  Yay, I'm done!"
$ git push -u origin gradescope_final
$ git tag -a final -m "Final submission for 18244 EX0"
$ git push --tags
```

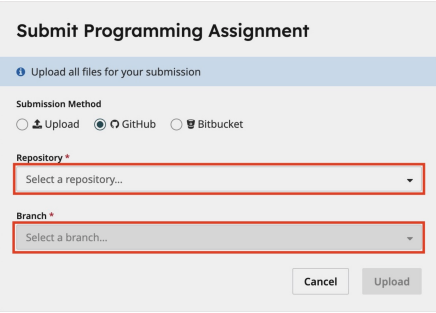
2) Log onto Gradescope and select the EX0 assignment.

3) Submit via GitHub by linking your GitHub account. Im assuming you have a GitHub account; if not, you should get one. You will only need to do this step once. Future projects will already have this linkage in place.

Please ensure the 18-244/644 class is selected, otherwise you wont be able to select your repository. You may have to press "Request," and then you should see the checkmark next to the 18-244/644 organization.



4) Select the repository you want to submit.



5) Select the "gradescope_final" branch that you want to submit.

6) Press submit!