EX1: Introduction to Yosys

Objective and Overview

The purpose of Exercise 1 is to get introduced to the star of the course, the Yosys suite of tools.

In this exercise, you will use Yosys to synthesize the RangeFinder hardware thread from EX0. You will generate circuit diagrams, and experiment to see how simple changes to your code or configuration of your tools can make a difference in the resulting circuit.

Schedule and Scoring

This exercise is due on **1 Feb, 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: Yosys Synthesis	20 points
Task #2: Fun with For-Loops	20 points
Task #3: Hierarchy and Optimizations	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}_EX1.pdf` and should be committed in your GitHub repository for EX1.

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

EX1 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 #0: Installation

Make sure you have completed HW0 before starting this exercise!

Task #1: Yosys Synthesis

In thistask, you will use Yosys to synthesize the `RangeFinder` hardware thread you built in EX0.

For starters, read your FSM-D into Yosys and visualize it using the `show` command. Can you identify components of your design in the diagram?

It is possible that you'll run into an error like:

```
For formats different than 'ps' or 'dot' only one module must be selected.
```

To resolve that error, take a look at the documentation of the `show` command with `help show`. Or, your might just call `flatten` on your design to replace module instantiations with the module. You'll learn more about this in Task #3.

> **For credit:** In your lab report append a diagram of your design after reading it into Yosys and calling `show`. Briefly identify components of the diagram that relate to your hardware thread.

Now that your hardware thread has been loaded into Yosys, its time to elaborate the design through various passes. Use `hierarchy`, `proc`, `alumacc`, `techmap`, and `abc` to synthesize your design.

Dont forget to optimize after each pass and `clean` your design after the last pass, `abc`. Inspect the state of your hardware thread using the `show` command in between each pass: in what ways did each pass alter your design?

If you want more information about a command, you can ask for help at the yosys command line:

```
yosys> help proc
```

Additionally you can take a look at this official Yosys tutorial.

> **For credit:** For each of the 5 passes submit in your lab report:
>
> 1. A diagram of your hardware thread after the pass (generated with `show`)
> 2. A brief description of how the pass changed/elaborated your design

Task #2: Fun with For-Loops

In this task you will experiment with for-loops in SystemVerilog. So far, you have likely only used loops when creating testbenches for 18-240 endeavors. However, loops can be used inside synthesizable `always_comb` blocks, as long as the loop can be unrolled at compile time.

This allows for incredibly scalable designs, often at a high hardware cost, as you'll come to see in this task.

Part A: Priority Encoder

Consider a combinational priority encoder (PE) module with the following signature:

```
module PriorityEncoder #(parameter W = 4)
(
    input  logic [W-1:0]             request_vec,
    output logic                     error,
    output logic [$clog2(WIDTH)-1:0] granted_idx);
```

It takes in a `request_vec` of a specified length (assumed to be a power of 2). It outputs the `granted_idx`, the index of the most significant bit that is active in the vector.

Assuming `W=4`, the encoder would output an index of 1 if `request_vec` = 0011 and 3 if `request_vec` = 1011, for example.

If the request vector is all zeros, `error` is asserted and `granted_idx` becomes 0.

For this part, write a **`PriorityEncoder`** inside `task2/PriorityEncoder.sv`, assuming a width of 4. You can use a single `always_comb` block with `if-else` statements. We have provided a **non-exhaustive** testbench inside `task2/PE_test.sv` to guide you. You can use EDA Playground (remember that?) to simulate your design.

> **For credit:** Inside your lab report include a screenshot of your SystemVerilog code for the `PriorityEncoder` module (always assuming `W=4`).

Part B: For-Loop Scaling

Now consider having a PE with much larger `request_vec` (8, 32, 256). Copy-pasting if/else statements might not be the best way to go about it...

Thankfully, a for-loop can be used to render your PE much more scalable. Inside the `always_comb` block of your design, replace the `if-else` clause with a for-loop. The loop should use an `int i` inside the loop guard and have `W`-many iterations.

In each iteration, you should check wether `request_vec[i]` is asserted and handle `granted_idx`. Also don't forget to deal with `error`, and make sure both your outputs are given default values outside the loop, at the begining of the `always_comb` statement.

If all went well, the provided testbench should still work for `W=4`. However, you can now easily scale your encoder up by simply toggling `W`

For credit: Inside your lab report include a screenshot of your modified SystemVerilog code for the `PriorityEncoder` module. Then briefly answer the following questions:

1. We assure you your new encoder remains synthesizable. Why do you think that is the case?
2. What do you think would make a for-loop unsynthesizable?
3. What do you think your encoder looks like when synthesized?

Part C: For-Loops Synthesized

SystemVerilog for-loops are powerful but can prove costly. Unlike software, loops in SystemVerilog directly lead to more hardware being created. You will explore this effect now.

Open Yosys and read your `PriorityEncoder` module. Then synthesize it using the default `synth`. Record the `Number of cells` printed at the end of the run. You can also get this info using the `stat` command.

Repeat this for `W` values of **4**, **8**, **16**, **32**, and **128**.

For credit: In your lab report record the number of cells for each of the **5** encoder sizes. Then, answer the following questions:

1. What trend do you observe between the number of cells and the `PriorityEncoder` bit-width? Does the number of cells scale linearly?
2. Why do you think the number of cells scales the way it does?

Feel free to include graphs to justify your answer. Also make sure to commit your code inside `task2/PriorityEncoder.sv` in your GitHub repository.

Task #3: Hierarchy and Optimizations

Part A: Flattened vs Non-flattened

In this section, you will experiment with hierarchical synthesis on a design with a lot of redundancy.

Inside `task3/muls_starter.sv` write a **new** module `my_multiply` that takes 2 16-bit numbers as input and produces a single 32-bit output. The module should multiply the 2 input numbers: use the "*" operator here (Yosys is smart enough to generate logic for a hardware multiplier). You are highly recommended to use the provided `BITS` and `NUM_MULS` macros to make it easier to experiment later.

Then, instantiate 10 copies of `my_multiply`, passing each of them a distinct set of inputs `a[i]` and `b[i]` and putting the output into `prod[i]`. It's recommended to use a "generate for loop" (Google it if you don't know what that is) here, along with the `NUM_MULS` macro.

Synthesize it in hierarchical mode. Note how we are not synthesizing to a specific architecture, just generic "LUTs"

```
$ yosys -p 'read_verilog -sv muls_starter.sv; synth -lut 4 -top muls; stat'
```

Write down the total number of LUT cells in the module, as well as the total CPU run time (both should be near the end of the Yosys log).

Then, try synthesizing it in flattened mode:

```
$ yosys -p 'read_verilog -sv muls_starter.sv; synth -lut 4 -top muls -flatten;
    stat'
```

Note any differences in runtime and utilization.

Now, repeat this for **5** different quantities of multipliers (both the flattened and non-flattened variants). Make sure you go down to very small quantities (i.e. 1 multiplier), as well as substantially larger than the default (i.e. 40), and some values in between.

Create a table showing the runtime, LUT utilization, and normalized LUT utilization (which is the utilization divided by the quantity of multipliers), in both flattened and non-flattened mode, for each of the quantities.

For credit: In your lab report, create a table showing the **runtime**, **LUT utilization**, and **normalized LUT utilization** (LUT utilization divided by the quantity of multipliers), for **both** flattened and non-flattened designs for each of the **5** multiplier quantities.

Then answer the following questions:

1. What does the trend look like in terms of runtime for the flattened vs non-flattened runs as quantity increases?
2. What does the trend look like in terms of normalized LUT utilization for flattened vs non-flattened runs as quantity increases?
3. Write a clear explanation of why you think those two trends go the way they do.
4. Speculate as to why the normalized LUT utilization is not always necessarily an integer value.

Part B: Tricking the Optimizer

Finally, let's try to set up a scenario that tricks the optimizer.

Without changing your `my_multiply` module at all, modify just the inputs to it (in the place in `muls` where you instantiate the instances of `my_multiply`) such that a few bits are lopped off one of the inputs.

For example, for a 32-bit value, changing `.Ain(a[i])` to `.Ain(a[i][19:0])` would lop off the top 12 bits to make it a 20-bit value. Think about what you would expect this to do to the utilization.

Repeat the same synthesis and table-making process as before, until you notice the new trend in the utilization and runtime values.

For credit: Include your table of results in your lab report. Record the same metrics you recorded in **Part A**. Then, answer the following questions:

1. What do the new trends look like in terms of utilization for the flattened vs non-flattened modes?
2. How do the values compare to the previous section? Why do you think this is the case?
3. Using what you've learned, give one advantage of each of the synthesis modes, as well as what scenarios you might want to use one vs the other.
4. Let's say you were building an entire CPU for a laptop, like the Apple M1. Would you want to run your synthesis (and placement/routing) algorithms in flattened, or hierarchical (non-flattened) mode? Explain the reasoning for your answer.

Make sure your code for this Part is committed in the GitHub repo!

Task #4: Jumping Off

Experiment on your own with Yosys. Below are some options you *could* explore:

Option 1:

One tool you might find interesting to play with is `netlistsvg`. It's similar to the `show` command in Yosys, but a bit nicer to look at, especially for smaller designs. To use it, try the following series of Yosys commands (with some Verilog modules from past classes/projects):

```
yosys> read_verilog -sv file1.sv file2.sv file3.sv
yosys> hierarchy -top module_you_want_to_see
yosys> proc
yosys> write_json out.json
```

Then, open up the `out.json` file and copy-paste the entire contents into the textbox at the netlistsvg demo page `https://neilturley.dev/netlistsvg/` and hit "Render" to see the result! (note that if your design is way too big, it might cause netlistsvg to freeze).

Try it with a few modules! If you took 18-240, some modules from your assignments might be interesting to look at (you might need to delete any testbench code that's left over in the file, as Yosys doesn't always parse it correctly). You might also find it interesting to run the Yosys `flatten` command after the `hierarchy` command, and see how this improves (or worsens) the visualization!

Option 2:

Look into the Linity Graph Visualization tool, a Visual Studio Code extension. You can install it directly through the IDE or through their marketplace page.

Try getting visualizations of a couple of modules (either from this course or from previous classes like 18-240). Explore the different functionalities this extension provides

> **For credit:** Include in your submission a description of what you learned and what you experimented with. Include some screenshots and describe what you think is interesting about them.
>
> Remember, we will be grading you based on your creativity and extent of exploration.

Turning Stuff In

To submit this exercise, follow these steps:

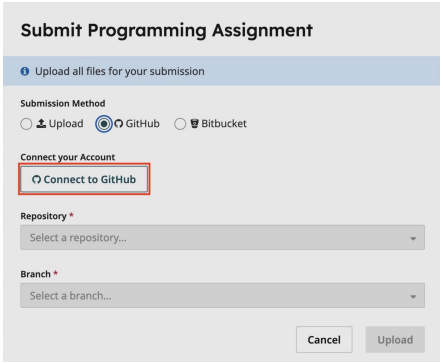
1) Create a branch of the commit you want to submit, naming it as your final commit and push it to GitHub. Tag it as well.

```
$ 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 EX1"
$ git push --tags
```

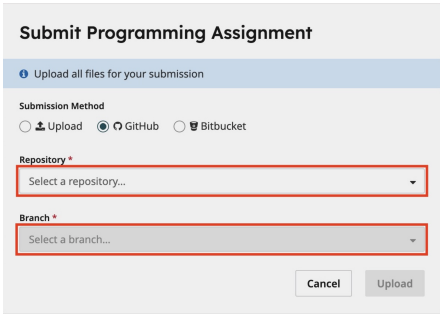
2) Log onto Gradescope and select the EX1 assignment.

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!