Lab 1

ece 260c. **Lab 1** Design Space Exploration

**Your name / Your PID**

Please enter your name and PID above – you agree to the academic integrity and course policies outlined in the [Syllabus](https://abkcourses.github.io/ece260c/docs/broken).

This lab will use the same Docker container as before. **Please ensure GUI is enabled** - see the [Software Setup Guide](https://abkcourses.github.io/ece260c/docs/tools) if you are unsure.

GitHub Classroom for accessing all necessary data files and submitting your work.

[Click here to unlock the assignment in GitHub Classroom.](https://abkcourses.github.io/ece260c/docs/broken)

Then run,

| gh repo clone ABKCourses/ece260c-lab1-YourUsername  cd ece260c-lab1-YourUsername |
| --- |

# Section I Synthesis & Technology Mapping

Synthesis is the process of transforming your RTL design (in our case, written in Verilog) into a gate-level netlist that is suitable for physical implementation. In ECE 260B, you may have had limited exposure to synthesis tools like Synopsys Design Compiler. You likely encountered typical inputs — including the Verilog RTL, a PDK-specific Liberty file, and an SDC (Synopsys Design Constraints) file — and outputs such as the synthesized Verilog netlist and estimated PPA (Performance, Power, Area) metrics.

In this section, we’ll explore the synthesis tool Yosys, focusing on how circuit representations evolve from your original RTL input to the final gate-level netlist. Along the way, we’ll examine some of the complexities involved in synthesizing real-world designs for real-world process technologies — and how various options within the tool can help you tune and improve your Quality of Results (QoR).

## Starting with Yosys

[Yosys](https://yosyshq.readthedocs.io/en/latest/) is the open-source synthesis tool that is commonly paired with OpenROAD. While it does not support the full feature set and QoR expected of commercial tools, it is rapidly evolving and improving. Yosys is also widely adopted in the FPGA domain, with some FPGA vendors – such as [Quicklogic](https://quicklogic-quicklogic-fpga-toolchain.readthedocs-hosted.com/en/latest/index.html) and [Cologne Chip](https://www.colognechip.com/docs/ug1002-toolchain-install-latest.pdf) – even using it as their primary synthesis tool.

Yosys is already a part of the ece260c-essential *Image I* Docker image you have used in Lab 0. OpenROAD-flow-scripts invokes Yosys as the primary tool in its Synthesis step.

**Yosys can be started without OpenROAD or ORFS, however, using the yosys command.**

Before running this command, cd into the section1/ directory. Then, run yosys – you should see a license/version notice and then the yosys> interactive prompt.

Yosys stores your design in-memory as a series of modules. When you open it, the session starts with an empty design. Here, we will be using the included section1/gcd.v, adapted from ORFS. **Run the following command within yosys to load the design:**

| read\_verilog section1/gcd.v |
| --- |

You can verify whether the design is loaded by running ls to list the loaded modules.

**Q1.1** Paste a screenshot below of the ls command run while gcd is loaded.

Before diving into specific commands, it is important to understand the role of Design Space Exploration (DSE) in the synthesis flow. A typical DSE involves evaluating multiple architectural and implementation options to optimize for various metrics such as area, timing, power, or performance. It helps designers better understand trade-offs, identify bottlenecks, and make informed decisions to improve the overall Quality of Results (QoR). Now that the modules have been loaded into memory, we can begin exploring the design using several key Yosys commands – including, **but not limited to**, the following:

| **Command** | **Description** |
| --- | --- |
| cd .. cd <module> | Allows you to step inside a module in the design  (A convenient version of select) |
| ls ls <module> | When outside any module, lists modules  When inside a module (using cd), lists instances and ports. |
| select \* select <query> | Makes it possible to select a subset of logic by object name(s) or by advanced functions like logic cones |
| show show <module> | Renders the design to a graph and displays it in a new graphical window |
| stat  stat <module> | Without any flags, shows counts of specific cells |
| dump  dump <module> | Dumps the selected module out as RTLIL, the intermediate textual representation Yosys uses. |

We can now start to understand how the design flows through the synthesis tool. First, Run the following: show Mux0

A new GUI window should open with the schematic for the Mux0 module – you may need to click the zoom-to-fit button.

**Note**: you may see errors in the terminal while show is running, even after it has closed. You can hit the enter key to return to the yosys prompt if this happens.

**Q1.2** Paste a screenshot of the schematic. What elements are visible?

### Processes

In Verilog, a [**process**](https://verilogams.com/refman/modules/discrete-processes.html) refers to a procedural block that executes in response to certain events, which may be explicitly or implicitly defined. You should be familiar with the following:

* always @\* / always\_comb / always\_latch blocks – which execute whenever a signal the block depends on changes
  + always\_comb / always\_latch are SystemVerilog features that combat always @\*’s ambiguity by helping ensure correct intent, such as whether the block should infer combinational logic or latches.
* always @(posedge clk) or always @(negedge clk) blocks – which execute on an edge, typically used for clocks or IOs. They're typically used to describe sequential logic.
* initial blocks – which run once at the start of a simulation. While primarily used for testbenches and initialization in simulation, some FPGA platforms may reinterpret them as actions triggered on reset.

The process model is advantageous because it aligns naturally with event-driven simulator architectures, while also being synthesizable into either pure combinational logic or **stateful** logic implemented using flip-flops or latches.

What you will notice in the schematic of Mux0 is that it is challenging to discern the behavior because processes are opaque i.e., they are not broken down into logic gates or control flow.

When Yosys transforms the design, it adds constructs that are not easily expressible in Verilog, like explicit latches. That’s why, when you want to export or view Yosys’s full database, the resulting code is provided in [RTLIL](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/yosys_internals/formats/rtlil_rep.html) (Register Transfer Level Intermediate Language) – an intermediate representation language used by Yosys during synthesis. It provides a low-level, implementation-friendly view of a Verilog design by translating behavioral and structural constructs into a flattened set of logic primitives, wires and cells. This format is useful for understanding exactly how high-level Verilog maps to synthesizable logic.

**Q1.3** Run the following code and paste a screenshot of the resulting RTLIL. Then, in 2-3 sentences, compare the RTLIL to the Verilog source for Mux0 in gcd.v

| dump Mux0 |
| --- |

Let us now map processes to real logic: Run the proc command to convert all processes in the design to a circuit representation. Then, run show Mux0 again.

**Q1.4** In 1-2 sentences, what can you see now in Mux0? Does it match the behavior described in Verilog in section1/gcd.v?

Now, let’s try sequential (stateful) logic. Run show RegEn.

**Q1.5** Paste a screenshot. In 2-3 sentences, what instances are present? What do you notice about how the always block in the original Verilog was converted?

## The Synthesis Flow

With an understanding of processes, we can now continue the synthesis process.

The **first step** in synthesis is to tell Yosys the **hierarchy** of the design – the relationship between modules based on how they instantiate one another. The top module serves as the **root** of this hierarchy; it is not instantiated by any other module and typically represents the top-level block or chip being synthesized.

Run the following to set the top module to gcd

| hierarchy -top gcd |
| --- |

Typically, before synthesis, we would execute flatten to flatten the netlist before performing optimizations, as optimizations are conservative and do not cross module barriers.

To keep the schematic from exploding in size, however, we will do this later. Let us perform a multi-module synthesis with optimization as follows:

| opt synth stat |
| --- |

For your reference, you can look at the yosys references for [opt](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/using_yosys/synthesis/opt.html) and [prep](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/cmd/prep.html) (which combines several optimization passes).

**Q1.6** Paste a screenshot of the stat command run above. In 1-2 sentences, what do you notice is different in terms of cell composition compared to what you saw in the schematic you talked about in Q1.4?

### Primitives & The Internal Cell Library

**Q1.7** Run show Mux0 again and paste a screenshot here (you may need to zoom in). In 2-3 sentences, what do you notice is different from the previous versions of Mux0 (like in Q1.4) in terms of cells utilized and the structure of the circuit network?

**Hint:** you can get a general idea of logic flow by looking from left to right (inputs on left, first stage next, …, outputs on right). $\_ANDNOT\_ implements the function Y = A & ~B and $\_ORNOT\_ = Y = A | ~B. A suffix

What you saw above is the evolution of the logic database through two different stages of what Yosys calls its “[Internal Cell Library](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/cell_index.html)”. Internal Cells, also called *Primitives*, are powerful technology-agnostic building blocks used to represent a design in a modular and abstract form. In Yosys, your input Verilog is first instantiated into a modular netlist containing a mix of processes and what are known as “word-level” internal cells that correspond directly to Verilog’s (multi-bit) mathematical-logical operations (i.e. a 32-bit AND, a 16-bit subtract, or a 4-bit DFF) and even other high-level constructs such as FSMs or memory blocks (for which Yosys contains several [extraction tools](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/cmd/memory_dff.html) to lift out). When the synth command is called, Yosys lowers these word-level internal cells to gate-level cells – logic primitives that more closely resemble the elements found in real PDKs, such as AND gates, muxes, and DFFs.

In the section1/sub folder, you will ask Yosys to write out the post-synthesis netlist that is currently loaded in both Verilog and RTLIL. Assuming you’re in the section1/ folder, do this by running :

| write\_verilog sub/postsynth.v write\_rtlil sub/postsynth.rtlil |
| --- |

Confirm that both files were created successfully. Verify that the files are present and non-empty. **You may wish to commit at this stage as a way to backup your progress.**

## 

## Optimization and Techmapping

So far, our design has remained in the technology-agnostic domain. We need to transform it to use our PDK’s standard cells – techmapping. A naive techmapping is possible – the common cells in the Internal Library can usually be mapped 1:1 to PDK cells. This would, however, leave performance on the table as optimizations using the delay data of PDK cells is possible, and these optimizations involve analyzing several possible representations for a given circuit and its subcircuits.

Yosys uses [UC Berkeley’s ABC](https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm), a synthesis tool capable of performing combinational and sequential logic optimization for both ASICs and FPGAs. When we call Yosys’ abc command, it translates your design into an ABC-compatible format – optionally splitting it into combinational sections – then calls ABC to optimize the logic, and finally maps the results back into Yosys’s internal representation. While ABC can be scripted and run independently, Yosys provides sensible defaults that make this process efficient and easy to use, saving considerable time during synthesis.

Internally, ABC operates by converting your circuit into an AND-Inverter Graph (AIG) — a simplified combinational representation composed of two-input AND gates and inverters. AIGs are particularly well-suited for a variety of optimization tasks, including redundancy elimination, logic equivalence checking, and efficient search for optimized circuit structures. ABC can also perform retiming.

While ABC can be run in a technology-agnostic mode — optimizing logic purely at the primitive level (which Yosys's synth command already performs) — we will be using it with a Liberty file. A Liberty file provides detailed timing, power, and functional information for each standard cell in a given PDK (typically grouped by threshold voltage or cell style) – for a quick tutorial check [here](https://courses.cs.umbc.edu/graduate/CMPE641/Fall08/cpatel2/slides/lect05_LIB.pdf). ABC uses this information to map portions of the logic (called "cuts") to optimal standard cells, continuing this process until the entire design has been transformed. This technology mapping can be guided by constraints such as timing or area, enabling more efficient and physically-aware synthesis. OpenROAD can also call into ABC at runtime to re-optimize the design based on updated constraints – this is part of the [rmp](https://openroad.readthedocs.io/en/latest/main/src/rmp/README.html) module.

Now, run the following script to clear out the design, synthesize it, and **finally techmap it with the IHP 130 PDK** (a copy of the liberty file is included in your repo):

| design -reset # Clear out the loaded design read\_verilog gcd.v hierarchy -top gcd  flatten  # Technology-agnostic Verilog-level optimizations prep  # Synthesize to technology-agnostic gate-level primitives  synth  # Replace any agnostic DFF primitives that cannot be mapped directly to IHP 130's DFF cells  dfflibmap -liberty sg13g2\_stdcell\_typ\_1p20V\_25C.lib  # Call into ABC to synthesize with a timing goal of 4000 ps = 4 ns on the worst path. abc -D 4000 -liberty sg13g2\_stdcell\_typ\_1p20V\_25C.lib # Remove disconnected wire that ABC optimized away  opt\_clean |
| --- |

**Q1.8** Run the following and paste a screenshot here. You should be able to see the Area statistics.

stat -liberty sg13g2\_stdcell\_typ\_1p20V\_25C.lib

For your reference, you can take a look at more advanced techmapping for [memories](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/using_yosys/synthesis/memory.html), [FSMs](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/using_yosys/synthesis/fsm.html), or arithmetic units. Take a look at Yosys’ [extract](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/using_yosys/synthesis/extract.html) functionality or how [extract\_fa](https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/extract_fa.html) is used to add full-adder cells available in [some PDKs](https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/blob/master/flow/platforms/sky130hd/cells_adders_hd.v).

## Timing and Power with OpenSTA

While stat gives you a cell-based area estimate, you likely also want timing and power information to fully evaluate your design. Unfortunately, this infrastructure is not yet part of Yosys so we will instead rely on [OpenSTA](https://github.com/parallaxsw/OpenSTA/tree/master).

Before we can use OpenSTA, we need to export a techmapped netlist. You will also need to submit this, so: (i) run the following command (assuming you’re in ./section1) and then, (ii) after verifying that the file is not empty, exit yosys using the exit command:

write\_verilog sub/postmap.v

**Exiting Yosys**, you can run OpenSTA by running sta in your terminal – it will also present an interactive CLI.

**Q1.9** We will start with checking whether timing is met. Run the following and paste a screenshot below of the final timing report. In 1-2 sentences, is timing met?

| read\_verilog sub/postmap.v read\_liberty sg13g2\_stdcell\_typ\_1p20V\_25C.lib link\_design gcd # OpenSTA defaults to using the PDK's units loaded from the liberty file. Try running report\_units. create\_clock -name clk -period 4 {clk} # Write a Standard Delay Format file as part of your submission write\_sdf sub/postmap.sdf report\_checks |
| --- |

While the report\_checks output is useful when we have a specific clock frequency target in mind, we’ll also want to determine the **maximum achievable performance** of our synthesized design — that is, the highest frequency it can reliably support based on current timing constraints. OpenSTA provides a convenient function for this: report\_clock\_min\_period.

**Q1.10** Run report\_clock\_min\_period and paste a screenshot below.

In 3-4 sentences, explain how the minimum period or compares to the 4000 ps target we asked ABC for above? (For those unfamiliar with, please check [here](https://www.intel.com/content/www/us/en/docs/programmable/683152/23-3/maximum-frequency-fmax.html))

Does this performance, relative to the 4000 ps target, indicate anything about synthesis’ effort i.e.

* Did ABC fail to meet the timing goal?
* Did ABC meet the timing goal and then stop?
* Did ABC meet the timing goal and then some?

Based on this, do you think it is possible to extract even more performance by resynthesizing with a lower clock period target? If so, how much lower would you try?

We can also get a sense of power usage. OpenSTA provides the ability to run static power analysis, with support for loading VCD files from simulation (check the OpenSTA [repo](https://github.com/The-OpenROAD-Project/OpenSTA) for implementation details).

**Q1.11** Run report\_power and paste a screenshot below:

If you are interested, you can take a look at the [OpenSTA manual](https://github.com/The-OpenROAD-Project/OpenSTA/blob/master/doc/OpenSTA.pdf).

**Ensure you answered all questions in this section and ensure all the files you need for submission are in the section1/sub folder and that they are non-empty. Check the submission checklist in section1/sub/README.md.**

# Section 2 Design Space Exploration

Design Space Exploration (DSE) is the process of identifying key tunable parameters within a design and searching for optimal configurations based on performance, power, area, and other design constraints. In modern chip development, DSE is a continuous process that occurs at every level in the flow — from high-level architecture to low-level physical implementation. For example, microarchitects often write small simulation models to explore pipeline-level behavior and evaluate the performance of novel design ideas, particularly in comparison to previous generations within the same product family. These models are progressively refined as they undergo reviews and validation through various stages of the design cycle. In some cases, architects may even develop early RTL prototypes or simplified physical implementations to support their proposals with concrete data.

Physical Designers also have these tunable knobs. From what we covered in Week 1 Lecture 2 on floorplanning, you may remember that macros can be defined at different utilization levels, aspect ratios, or even standard cell types (i.e. low-Vt/high-density).

As chip design has grown increasingly complex and resource-intensive, the need to automate DSE has only become more critical. Automation approaches range from simple brute-force parameter sweeps to more advanced optimization techniques, such as machine learning–based tools like ORFS’s [AutoTuner](https://openroad-flow-scripts.readthedocs.io/en/latest/user/InstructionsForAutoTuner.html), and even emerging deep learning methods.

However, tools like AutoTuner often require running the entire physical design flow, which can be time-consuming. In early-stage exploration, it’s usually more practical to rely on simplified models or to run only a portion of the flow to generate quick performance, area, or power estimates. Once a promising design point is validated at an early stage, it can progress to later stages of refinement and eventually be evaluated through the full implementation flow.

## Our Design

We will try out DSE by running a synthesis-only sweep on a design called ASQRT. This unit implements a configurable, parallel square root accelerator using an iterative algorithm to compute the square root of a 32-bit input integer — with each iteration of the algorithm generating or converging one bit of the output. The design includes N\_PIPES, which controls the number of **parallel square root pipelines**. Each pipeline can be configured with a N\_DEPTH parameter, which determines how many iterations are performed per cycle — effectively controlling how many cycles are required to compute the result (i.e., deeper pipelines reduce latency per bit). The N\_CYCLES parameter sets the total number of cycles a pipeline is allowed to run. It must be greater than or equal to the number of output bits divided by the pipeline depth. **We will be running DSE over N\_PIPES, N\_DEPTH, and N\_CYCLES.**

For this section, cd into the repo’s ./section2 directory. You can take a look at the RTL in section2/rtl.

Also, we will need the matplotlib dependency to make graphs in Python. Install it by running:

pip install matplotlib

Our first step is to define certain design requirements – these will allow us to bound the range of our variables and reduce the number of variables.

We will start by establishing a performance requirement: we want a throughput of 1 square root per cycle. This gives us a 1:1 relationship between N\_PIPES and N\_CYCLES, i.e., if a single pipeline takes 2 cycles to compute a result, we will need 2 parallel pipelines to maintain our target throughput. At the end of this sweep, we will generate what can be referred to as a “frequency-independent iso-performance” comparison graph. The frequency-independent qualifier is key: while we are normalizing for throughput across configurations, the actual performance in terms of real-world throughput will still depend heavily on the maximum achievable frequency (), which is why we will also analyze and plot .

**Q2.1** Knowing that each iteration of our square root operation produces one bit of the result, what is the minimum number of iterations required to compute the full correct output of the square root operation?

Hint: Consider the rounded-down square root of the largest possible 32-bit unsigned integer, and how many bits are needed to represent it.

**Q2.2** Knowing what we know about synthesis hierarchy and optimization, in what cases do you think the synthesis tool can or cannot correct for “overdesign” i.e. if we were to set N\_DEPTH too high, such that there are redundant iterations that “do nothing” because the square-root result has already converged? Answer below in 4-5 sentences.

Your answer to Q2.1 will help guide the definition of a **relationship between** N\_DEPTH **and** N\_CYCLES, and in turn, with N\_PIPES. This enables us to **reduce our design search space from three variables to one**, simplifying our DSE process.

**Q2.3** Express the mathematical relationship between N\_PIPES with N\_CYCLES and N\_DEPTH with N\_CYCLES.

We will continue to express this dependency explicitly in our Python code, which will drive DSE.

## Scripting for DSE

**Q2.4** Open section2/explore.py – this is our primary DSE script. Find and fill in the definitions for N\_DEPTH and N\_PIPES within the loop, including the resolution of the environment variables. Paste a screenshot of your updated code here:

For your convenience, the rest of the script has already been completed. Please review it to ensure you understand the flow: it calls Yosys and OpenSTA with different parameter combinations, generates synthesis and timing reports, and uses those reports to build a 2D plot for analysis.

Next, we will provide Yosys and OpenSTA with the necessary scripts — explore.py will call synth.tcl for Yosys and analysis.tcl for OpenSTA, respectively.

**Q2.5** Open synth.tcl and follow the instructions to complete it. When it’s done, screenshot it and paste it here. You will commit this file later.

**Q2.6** Open analysis.tcl and follow the instructions to complete it. When it’s done, screenshot it and paste it here. You will commit this file later.

**Q2.7** Run the completed explore.py. A GUI graph should pop up. If it doesn’t, you can also open sub/dse.png. Paste a screenshot here.

**Q2.8** In 5-6 sentences, write your own analysis based on what the graph shows you.

* Can you pick a best design in terms of or area?
* Can you pick a best design in general – perhaps by justifying some sort of balance of factors?

**Q2.9** Pick any “good design” (based on what you looked at above in Q2.8) and, below, write out its N\_CYCLES and provide its statistics ( and area) – you may look at dse.table.json for this.

**Q2.10** Imagine that you had done this design space sweep using the full P&R flow – running not just synthesis/STA but also place and route. Answer in 5-6 sentences:

* How much longer would it take? Write down your intuition — you don’t need precise numbers, but think about how the complexity and runtime might scale compared to synthesis and STA alone. Consider what extra steps P&R introduces, and how those could impact total exploration time.
* Does this run-time have implications for how many design configurations you can test?
* Do you think that performing P&R will improve the accuracy of analysis results? Would it be worthwhile given the longer run-time?

It is now time to prepare your submission. Unlike other sections, there is no submission subdirectory. **Please ensure the following files are present, completed, and non-empty in ./section2:**

* explore.py
* synth.tcl
* analysis.tcl
* dse.png
* dse.table.json
* dse.postmap.v
* dse.stat.json
* dse.analysis.json

If there are more files present, it’s okay to leave them be. You may wish to commit to save your progress here.

**Ensure you answered all questions in this section and ensure all the files you need for submission are in the directory before continuing to the next section.**

# Section III Closing the Loop

Now that you have completed a design space exploration, it is time to validate your selected design through a full implementation. The goal of the synthesis-only DSE in Section II was to enable exploration across a wider range of configurations using the same amount of compute resources — a common and critical consideration in industry workflows.

In a more realistic setting, a team might select a top-N set of candidate designs for full-flow implementation, rather than committing to just one. However, for this lab, you will take the single best design you selected in Q2.9 and run it through the complete OpenROAD-flow-scripts flow, including placement, clock tree synthesis, and routing — thereby validating its true performance, area, and physical feasibility.

From the root of the repo, cd into ./section3. Then, run orfs\_copy (similar to Lab 0) to bring in an instance of OpenROAD-flow-scripts (this should not be committed later).

In ./section3, you will find config.mk, constraints.sdc, and the RTL again. You should be familiar with these files from Lab 0 – they are what drive implementation in ORFS. What we are doing here is running a custom design in ORFS without adding it to ORFS’ folder structure.

These files are mostly complete, except you need to complete the VERILOG\_TOP\_PARAMS section in config.mk.

**Q3.1** Complete config.mk and paste a screenshot below.

Now, run ORFS using the command that follows:

| make --file=OpenROAD-flow-scripts/flow/Makefile DESIGN\_CONFIG=config.mk |
| --- |

**Q3.2** When the flow has completed, you will see the familiar directories reports / logs / results / objects.

Open logs/ihp-sg13g2/asqrt\_top/base/6\_report.log and search for “report\_design\_area”

What is the area and utilization reported?

In 1-2 sentences, describe whether what you observe is similar to what the DSE process estimated, keeping in mind that the area reported by DSE is at 100% utilization?

Now, open section3/reports/ihp-sg13g2/asqrt\_top/base/6\_finish.rpt and search for “data arrival time” under the “finish report\_checks -path\_delay max reg to reg” section. The positive of this number represents your minimum clock period.

You can compare this to the you estimated in DSE using the following formula:

Given a clock period , Keep in mind this will output frequency in Hz and it requires that T is defined in seconds (i.e. ).

**Q3.3** What is the clock period and ?

In 1-2 sentences, how does this compare to the you found in Q2.9?

**Q3.4** Answer the following in 5-6 sentences:

* Based on both timing and area, do you think the DSE process gave you a realistic estimate of this design’s performance?
* Was the synthesis and STA output (from Section II) generally pessimistic or optimistic compared to the full P&R results?
* Recalling your answer in Q2.10, has your response changed after seeing the P&R run-time and/or Quality-of-Results?

**For this section, you will submit your completed config.mk file alongside the ORFS output directories (reports / logs / results / objects). You do not need to move them to any submission directory.**

If you placed ORFS in the directory with a different directory name than the default OpenROAD-flow-scripts, please delete it before committing.

You can now commit all the files in ./section3 and its subdirectories.

**Ensure you answered all questions in this section and ensure all the files you need for submission are in the directory before continuing to the next section.**

# Finalization

Please review each section and ensure you have answered all the questions.

Ensure you have committed and pushed all files required for submission for each section.

Go to your GitHub repo and verify that the push was successful. Your push is all that is required for submission to count in GitHub and you may push until the Lab deadline.

### Additional Notes

In this lab, we did not cover retiming – performance optimization performed by moving logic around register boundaries. [ABC supports this](https://yosyshq.readthedocs.io/projects/yosys/en/0.47/cmd/abc.html) however.

We also did not cover repipelining – a more general case of optimization whereby the entire pipeline is configured with differing numbers of stages. Repipelining in industry contexts is currently done manually and has been one of the [major drivers for CPU performance increases](https://www.cs.cmu.edu/afs/cs/academic/class/15740-f03/public/doc/discussions/uniprocessors/technology/deep-pipelines-isca02.pdf) in recent years.

Automated repipelining is a topic associated with High-Level Synthesis as HDLs like Verilog require explicit FSM controllers that are not amenable to repipelining changes that would, say, change pipeline depth or wait cycles.

**<end>**