ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

EFCL Winter School 2026

Sample Solution Exercise 03

Synthesis with Yosys

F. K. Gürkaynak
Prof. L. Benini

Last Changed: 2026.02.09

1 Introduction

In this exercise, you will learn how to go from an implementation in a hardware description language (HDL) like Verilog, SystemVerilog or VHDL to functionally equivalent gate-level netlist (a Verilog file containing only instantiating of standard cells and macros and wires connecting the cells). The cells and macros are provided by the foundry or another IP provider in a collection called standard cell library which contains all the gates that can be manufactured in a fully digital design for the target technology. The standard cell library together with all other technology-specific data needed to design a chip form the process design kit (PDK). For the winter school we use the open-source IHP130 PDK available on Github[1]. This gives us the metal stack, the IO cells, SRAMs and general technology specific setups. However, the standard-cell library are our own *EZ-cells* developed in-house during the VLSI-5 course (led by Oscar Castañeda and Prof. Christoph Studer). The library will be released publicly soon; please do not copy or distribute any of its technical details until then.

The process going from HDL to a netlist is called '`Synthesis`' and from a more abstract point of view, it is the last step of what is commonly called the '`Front-End`' of the design process[2]. However, I strongly recommend you avoid talking in terms of `Front-End` and `Back-End` since a lot of steps are internally sub-divided into a front and back, which may lead to confusion.

The leading open-source synthesis tool is called '`Yosys`'. In this exercise you will learn the needed steps to perform synthesis, both from a more general perspective and also specific to Yosys. In doing so, you will be guided through the whole process of synthesizing Croc-SoC.

Figure 1 gives a graphical overview of the subject of this exercise.

If you want to know more about how Yosys came to be and/or see the synthesis process using simple examples, you can visit Yosys' online documentation[3].

If you want a more formal explanation of the ASIC design flow, it is described in detail in the textbook[4].

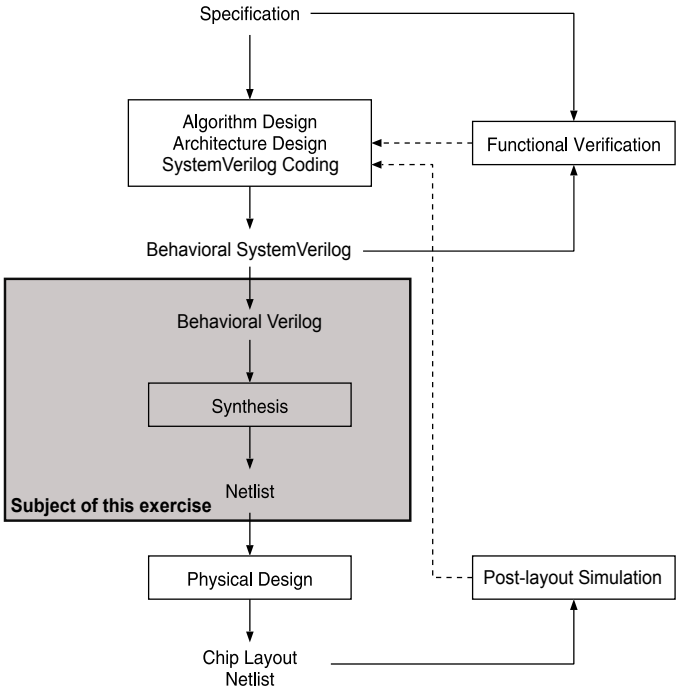

Figure 1: Excerpt of the ASIC design flow with the subject of this exercise highlighted.

[1] https://github.com/IHP-GmbH/IHP-Open-PDK

[2] The frontend describes the functionality/logic of your system think block diagram, behavioral HDL code or the netlist. The back-end is the physical implementation, like where each cell is placed or how exactly they are connected (OpenROAD flow).

[3] https://yosyshq.readthedocs.io/projects/yosys/en/latest/introduction.html

[4] Hubert Kaeslin, "Top-Down Digital VLSI Design – From Architectures to Gate-Level Circuits and FPGAs"

1.1 About the Style

We will use a number of different styles to identify different types of actions as shown below:

Student Task: Parts of the text that have a gray background, like the current paragraph, indicate steps required to complete the Exercise.

Actions that require you to select a specific menu will be shown as follows:
`menu→ sub-menu→ sub-sub-menu`

Whenever there is an option or a tab that can be found in the current view/menu we will use a BUTTON to indicate such an option.

Throughout the exercise, you will be asked to enter certain commands using the command-line[5].

```
sh>  command to be entered on the Linux command line
```

1.2 Getting Started

Student Task 1:

- Start by navigating to your home and then the exercise directory

  ```
  sh>  cd ~/ex03
  ```

- Enter the `oseda` container's shell

  ```
  sh>  oseda bash
  ```

Note 1: If you ever open a new terminal window or tab, don't forget to re-enter the oseda containers shell, otherwise you won't be able to start the open-source tools.

[5] There are many reasons for using a command-line, some functionality can not not be accessed through GUI commands, and in some cases, using the command-line will be much faster. Most importantly, things you enter on the command line can be converted into a script and executed repeatedly.

2 Synthesis: An Overview

Before delving into Yosys synthesis, lets first look at what steps an abstract synthesis tool has to perform. You can then later on try and match the different commands ran in Yosys to one of these points.

Parse Design: The first step is to load the behavioral description, in our case SystemVerilog, into the synthesis tool. The synthesis tool needs to parse the HDL language features and convert it to an internal representation. Usually this involves first parsing the design into an abstract syntax tree (AST), a representation that closely resembles how the HDL language is structured. Then in a second step the AST is converted into a representation more suited to synthesis. In this representation we are no longer interested in how a certain operation is implemented, instead we care about what it does and how it connects to other operations. At this point the design is still behavioral (it contains code-like features which can implement something in an abstract way).

Elaborate Design: The next step is to take the behavioral design and convert it to a structural design. While a behavioral model may contain abstract things like loops or conditional branching, a structural model is very close to a netlist. The only thing it contains are cells (arithmetic operations, finite-state-machines etc) and connections between cells. Any parameters in the design (ie the size of a data-bus) can be set to their final values and the corresponding values will be propagated to the subdesigns during elaboration.

Logic Optimization: With different techniques the structural representation of design can be optimized. This means the length of the longest paths is reduced, redundant operations are removed and the output shared, constant bits are propagated and so on.

Mapping: Interwoven with the logic optimization are mapping steps. In mapping we take one or more cells and convert them into another representation. In practise this usually means we start with cells representing very high-level operations (like an arbitrary arithmetic operation) and using a mapping, we transform it to a lower-level representation (like logic gates). During the last steps of synthesis we must map the internal representation to the standard cells provided by the PDK, this is called technology mapping.

In commercial tools you would also supply a set of constraints describing the timings your design needs to meet (for example different clock nets might use a different clock period). During the whole process, the tool tries to attain the goals you set. It tries to fulfill the timing constraints you set and on all non-critical paths it is then able to reclaim logic area by finding more compact representations of the same function (which usually means they are slower).

Currently, the open-source tools do not support constraints beyond a single target frequency so we will not further explore this aspect.

3 Yosys Synthesis

In the previous section you have learned what steps are necessary in order to synthesize a design described in SystemVerilog. We will now go through the synthesis process step by step using Yosys.

SystemVerilog is a complex HDL and most open-source tools currently do not support all or most of its language features, luckily Martin Povišer with support by ETH Zurich solved this problem. Martin is the developer behind yosys-slang[6], a plugin for Yosys built on top of the Slang[7] SystemVerilog language library (developed by Michael Popoloski). It offers leading coverage of the SystemVerilog language for open-source tools, see the chipsalliance test suite[8], and is able to parse and elaborated even complex PULP Platform projects (Snitch-Cluster, Mempool, Cheshire etc.). Using *yosys-slang* the entire flow to an elaborated design looks like this:

Bender: (Optional) Manages the dependencies (repos) and creates a file-list containing the paths to all files that must be inlcuded in the compile contex.

yosys-slang: Using *Slang* it parses and the source files, resolves pre-processor directives (eg defines) and builds a tree of modules starting from the specified top design and including all instantiated modules. Then *yosys-slang* will uniquify each instance of a module (resolving parameters in the process) and then elaborate the insides of the module, converting SystemVerilog syntax into circuit of Yosys' internal cells.

[6] https://github.com/povik/yosys-slang
[7] https://github.com/MikePopoloski/slang
[8] https://chipsalliance.github.io/sv-tests-results/

Yosys At the end the elaborated design is available in Yosys and represented using Yosys' internal cell representation.

3.1 Parsing

During this exercise, you will execute a lot of commands and sometimes things might fail and you would have to redo everything. Yosys does have a command history (↑) but usually you will want to execute a bunch of commands, so we recommend you create your own script and add the commands in there. Yosys can read two types of script files. The first just contains Yosys commands (file ending `.ys`) and you can run them using the command `script` in Yosys. The second one is a tcl-script (file ending `.tcl`) which is able to access yosys commands from the name space `yosys`. You can run them inside Yosys using the command `tcl`.

Note 2: In the Yosys shell you can get more information about a command using the `help` command. Eg running `help stat` gives you more information about the `stat` command.

Student Task 2:

- First, lets start the provided synthesis flow so we can later inspect the results if we run out of time, open a second Terminal and navigate to the same yosys directory you are currently in (`pwd` shows you where you are), then start the flow:

```
sh> oseda bash
sh> ./run_synthesis.sh --synth
```

- In the first Terminal go into the **`yosys`** directory and call the help of the bash script, it gives you an overview of what files the synthesis flow needs and what it outputs at the end.

```
sh> ./run_synthesis.sh --help
```

- Now start Yosys with the interactive shell

```
sh> yosys
```

  Note that the built-in `./run_synthesis.sh --open`, opens Yosys in its interactive *tcl* shell instead. You may also use that though you would need to change some of the following commands.

- First you need to collect and then load our technology data, we already prepared this for you.

```
ys> tcl scripts/init_tech.tcl
```

- Now you can load the design into Yosys, as this requires yosys-slang we first need to load the plugin

```
ys> plugin -i slang.so
ys> read_slang --top croc_chip -f src/croc.flist --keep-hierarchy
```

  The `--keep-hierarchy` changes the default behavior of yosys-slang from flattening the entire hierarchy to preserving it.

- Lets figure out what we currently have loaded in Yosys using

```
ys> stat
```

  It shows you all the modules in the current design and what they instantiate.

- Closely read the last dozens of lines of the Yosys output, what can you tell about the internal representation, what kind of granularity is it using?

Sample solution: You should see a bunch of cells starting with a $, this means it is a Yosys internal cell, and they are all lower-case. The lower-case cells represent more high-level constructs like entire arithmetic operations or N-bit wide boolean operations. There should also be the IO cells, the SRAMs and a few standard cells (they are directly instantiated in our design instead of being synthesized).

- In the following steps we will want to closely follow only one module to see what exactly Yosys does. I recommend you follow one of the `delta_counter*` modules as it is relatively simple and contain some control logic, arithmetic operations and a register (flip-flops).
  The following command will list all modules of type `delta_counter`, the `$` is added by yosys-slang to split the module name from the instance name (remember, each module is unique to its instantiation after elaboration) an the `*` lets us glob search for all of them.

```
ys> select -list t:delta_counter$*
```

- Open your selected module in one of the SystemVerilog/Verilog files mentioned in the **`src/croc.flist`** file list.
- What does your selected module do?

Sample solution:

delta_counter: It stores an integer value, the size in bits is given by `WIDTH` (remember, in **`croc_svase.sv`** all parameters have been propagated so the value you see there is not a default, it is the real value).
The value is stored in a register called `counter` with the input (value to be set) `counter_d` and the output (currently stored value) `counter_q`. It is clocked by `clk_i` and has a asynchronous falling-edge triggered reset `rst_ni`.
When `en_i` is high, it increments (`down_i==0`) or decrements (`down_i==1`) the value by an amount given by `delta_i`.
If `load_i` is high, it instead loads in a new value from `d_i`.
If `clear_i` is high, it instead sets the value to zero.
The current value is output to `q_o`.

- To quickly find your module in Yosys, you can save a selection for it using:

```
ys> select -set delta <your_module> %M
```

Where `<your_module>` is the full name of your module, something like `obi_demux$croc_chip.i_croc_soc...i_counter`.
This saves a selection containing only your module under the name 'delta'. If you want to perform a command only on your module, you then use

```
ys> \emph{command} @delta
```

Note 3: The `select` command in Yosys is extremely powerful, you can see some more use-cases in **`scripts/yosys_synthesis.tcl`** and see the full syntax in the Yosys documentation[a].

[a] https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/select.html

3.2 Print & Display

At this point you probably want to know what exactly Yosys did to your module, in fact I recommend you regularly do this in the following tasks (remember, put it in a script for easy reuse).

There are a few commands in Yosys which are useful to debug problems you have. The simplest of them is to simply `write out Verilog`:

```
ys> select @delta
ys> write_verilog -norename -noexpr -noattr -selected delta.v
ys> select -clear
```

Unless you know exactly what the above does, I recommend you use exactly this combination of commands and flags and just put it into some script (eg **`scripts/print.ys`**).

The next command just `prints statistics` including how many of which cell is in the design.

```
ys> stat @delta
```

And finally Yosys can generate `schematics` showing the selected network visually. This is something you should only use on smaller selections as it quickly becomes unwieldy.

```
ys> show -colors -width -stretch -long -prefix delta -format pdf @delta
```

`-prefix <p>` sets the path and basename of the file it is saved to. You may want to explore the flags of this command and make it your own.

You can print the `manpage` of a command using:

```
ys> help <cmd>
```

3.3 Elaboration

Equipped with a nice script to print our module[9] we can move on to elaboration.

Student Task 3:

- Explore the Verilog output of your module, which parts have already been elaborated and which haven't?

  Sample solution: The arithmetic operations and a NOT have already been elaborated but the control logic in the **`always`** process have not

- Let Yosys check and clean-up the design hierarchy

  ```
  ys> hierarchy -check -top croc_chip
  ```

  If it can't find certain modules, technology macros or other things, it would tell you here.

[9] Hint (put it in a script)

3.4 High-Level optimization

You may have noticed that all the cells we currently have in our design start with a $ and the names are all lower-case. This is characteristic of the high-level (or abstract) representation in Yosys. High-level because single cells can represent large and complex operations such as division.

the most common optimization commands in Yosys start with `opt_*`. There is also the larger `opt` pass which executes a series of `opt_*` sub-commands. The four most important way to call `opt` are:

- `opt`: The pass with the default settings
- `opt -noff`: Same as above but does not perform any flip-flop related optimizations (integrating enable signals, constant removal etc)
- `opt -fast`: Runs an alternative sequence of sub-commands that may not be quite as good but doesn't take as long
- `opt -full`: Adds some additional optimizations

As a rough guideline the closer we get to a gate-level representation the more cells we will have in our design and the costlier the commands are. So early on you can use the more powerful commands anytime you wish but later in the synthesis process you will want to use them a bit more sparingly.

Lets run some optimizations on the current representation.

Student Task 4:

- Execute the following optimization commands

```
ys> opt_expr
ys> opt_clean
ys> opt -noff
ys> fsm
ys> opt
```

  Here we also have the `fsm` command, it finds, extracts and optimizes state-machines.

- Lets continue optimizing

```
ys> wreduce
ys> peepopt
ys> opt_clean
ys> opt -full
```

  `wreduce` reduces the width (number of bits) of operations, `peepopt` performs some operator-specific optimizations and then we again perform general optimizations.

3.5 Mapping

At this point of the flow we will start mapping high-level cells to lower level implementations. Every arithmetic operation can be implemented in a number of different ways (architectures). Currently, Yosys mostly picks a good compromise and sticks with it. Choosing different architectures depending on the circumstances is one of the next big things being worked on.

Student Task 5:

- The next two commands are mapping commands.

```
ys> booth
ys> alumacc
ys> share
ys> opt
```

`booth` implements multipliers using Booth-encoding (we don't have any real multipliers) and `alumacc` collects various multiply, add and subtract operations together into $alu and then $macc (multiply-add) cells, increasing resource sharing[a].

- Next you need to map memories to flip-flops and then optimize unnecessary flip-flops away.

```
ys> memory
ys> opt -fast
ys> opt_dff -sat -nodffe -nosdff
ys> share
ys> opt -full
ys> clean -purge
```

[a] At this point you may notice that `booth` and `alumacc` do not work well together since both commands turn multipliers into other cells. This will be improved in the future. As a general rule using `booth` can result in a faster design but using only `alumacc` can reduce the area.

At this point we are about to use the most powerful mapping command in Yosys. `techmap` can take a Verilog implementation of any cell currently in Yosys and replace these cells with the implementation described in the Verilog mapping file. It has a default mapping file (which we will use) but if you are not happy with one of the architectural choices, you can use any other architecture and run it before using the default `techmap` mapper.

For us this means the abstraction level of our representation is about to change drastically. Currently it contains high-level concepts like arithmetic operators but after `techmap` they will be implemented using generic gates.

There is also another command called `extract`, it can match subcircuits and replace them with a custom cell you define. These two commands together can be very useful if you have some optimized implementation and want to use it in your design. There is a tutorial in the documentation[10].

[10] https://yosyshq.readthedocs.io/projects/yosys/en/latest/using_yosys/synthesis/extract.html

Student Task 6:

- Before running `techmap` make sure you have a `stat` and `show`/`write_verilog` output copied somewhere so we can compare it
- Run the following commands (this might take a few minutes)

```
ys> techmap
ys> opt -fast
ys> clean -purge
```

- Compare the type and number of cells used from before `techmap` to after

  Sample solution: The number of cells increased drastically since each cell implements a much smaller part of a logic function. Also they are now named something like $_AND_, this is the naming scheme of the generic gate library Yosys uses.

In the main flow the next step would be to flatten the design. This means the module instantiations are replaced with a copy of the content of the module. You are able to keep certain modules by using `setattr -set keep_hierarchy 1 <module or instance>`.

Flattening more of the design makes it harder to debug and write constraints but it increases the potential for additional optimizations.

Currently Yosys does not perform any cross-boundary optimization (propagate information like logic functions through the instantiation of a module), this makes flattening as much as possible even more important.

Student Task 7:

- You probably want to continue follow your module so lets make sure we keep it

```
ys> setattr -set keep_hierarchy 1 t:<your-module>
```

  Where `<your_module>` is the full name of your module, so for example `t:obi_demux$croc_chip.i_croc_soc...i_counter`

- Now lets flatten the rest

```
ys> flatten
ys> clean -purge
```

- You can run `hierarchy` to make sure your module is still there (it first shows the kept modules, then the ones being removed)
- At this point it can be a good idea to split buses on module ports into individual nets and make sure flip-flop and other cells have usable names. This is done for compatibility reasons (`splitnets`) and to make it easier to debug and write constraints (`rename` and `autoname`).

```
ys> splitnets -ports -format __v
ys> rename -wire -suffix _reg t:*DFF*
ys> autoname t:*DFF* %n
ys> clean -purge
```

3.6 ABC Logic Optimization & Mapping

Yosys can internally use another tool called ABC to run strong logic optimization algorithms and map to standard cells.

Similarly to Yosys, ABC also requires a script describing which commands it should execute. There are a bunch of different ABC scripts out there for example the default scripts used in Yosys, the OpenROAD-flow-script area and speed oriented scripts and also the scripts generated by OpenLANE.

Today you are going to use the ABC script used to synthesize the open-source Linux capable Basilisk[11].

The script itself is documented in a IWLS contribution[12]. The shortest possible description is that it uses a logic optimization technique called 'Lazy Man's Synthesis', which cuts your design into small functions and then replaces them using implementations from a pre-computed table. Then followed by mapping the design to the standard cells.

Student Task 8:

- Before calling ABC, you first need to map the sequential elements to standard cells

```
ys> dfflibmap -liberty ../technology/lib/ez130_8t_tt_1p20v_25c.lib
```

  You can find the above path in the **`scripts/init_tech.tcl`** script.

- Now we can call ABC

```
ys> abc -D 10000 -script scripts/abc-opt.script -liberty
       ../technology/lib/ez130_8t_tt_1p20v_25c.lib
ys> clean -purge
```

  The `-D <ps>` flag sets the target period in picoseconds, this is mostly used in the final mapping to standard cells and the buffering and resizing at the end of the ABC script.

- Congratulations! At this point you have a netlist. Before we write it out we need to deal with constant values

```
ys> setundef -zero
ys> hilomap -singleton -hicell TIEHI Y -locell TIELO Y
ys> write_verilog -noattr -noexpr -nohex -nodec netlist.v
```

3.7 OpenSTA

Now that we have a netlist of our standard cells and SRAMs, we are able to get a first timing report. The tool of choice here is OpenSTA, it is a professional static timing analysis (STA) engine that was open-sourced and supports most Synopsys design constraints (SDC) commands. The default shell it uses is tcl based. With a few commands we can setup some basic timing constraints where we tell the tool how to interpret our netlist and then we can generate a timing report. The SDC commands needed are:

`get_ports <glob pattern>`: Goes to the module boundary of the currently linked design and tries to find module ports matching the pattern (there is also `get_cells` and `get_nets` that work similar but for cells and nets inside the linked design).

`create_clock -name <name> -period <time in ns> <startpoint>` sets up a simple clock originating from the startpoint (usually a port or output of a cell) with a certain period. The name given here is the one that appears in reports and is used to reference the clock in further commands. This sets an implicit constraints for all paths starting and ending at a flip-flop driven by this clock, the signal must not change for some hold time after the clock edge and it has to go to its final value for some setup time before the next clock edge (one period later) arrives.

[11] https://arxiv.org/pdf/2405.03523
[12] https://arxiv.org/pdf/2405.04257

`set_max_delay <time in ns> -from <startpoint>` is a constraint that we give to the circuit. We tell it that the maximum time from a certain startpoint to all sequential endpoints (flip-flop and SRAMs) must be smaller than the given time. Note that without further arguments this overrides any other constraint on the selected paths eg the implicit one generated by defining the clock.

`set_input_delay -clock <clock> <delay in ns> <port>` tells the tool how late, relative to the clock edge, the chip's input signal can arrive at the port (ie how much delay is added externally eg from PCB traces). Whatever number you give is subtracted from the clock period, so the logic inside needs to go to its final value faster to account for the external delay.

`set_output_delay -clock <clock> <delay in ns> <port>` tells the tool how much time we need to leave at the output of the chip so the next chip has time to properly sample the signal. The bigger the number, the less time is left for the internal combinational path.

Note: Usually we would want to add two input and output delays with the `-min` and `-max` arguments. This is because it is also possible to be 'too fast' and make it so the signal changes too quickly while it is still supposed to be stable in the hold condition/

Student Task 9:

- Still in the **`yosys`** directory, start OpenSTA

```
sh> sta
```

- The first thing you need to do is the same as with Yosys, you need to load the technology files. This is done using the same prepared script

```
sta> source scripts/init_tech.tcl
```

- Next load the netlist we wrote out from yosys and link the top module against the technology and other modules (this is similar to yosys' `hierarchy -top croc_chip`

```
sta> read_verilog netlist.v
sta> link_design croc_chip
```

- Now the only thing left to do is set constraints. For now we ignore the timings of the IO and the paths in the JTAG module (which use the JTAG clock), we would just like to get an idea of the paths in the main clock domain.

```
sta> create_clock -name clk_sys -period 12.5 [get_ports clk_i]
sta> set_max_delay 100 -from [get_ports rst_ni]
```

- These are all the constraints needed to get a first timing report, lets print it.

```
sta> report_checks -path_delay max > timing.rpt
```

- Exit the `sta` shell and open **`timing.rpt`**, it reports only the worst path of each group. Where is the worst path in the main clock domain going through?

  Sample solution: It is starting from the cores instruction decode register, going through the register file, more logic in the core, then into the `croc_soc` module (`i_croc`) which means its in the OBI interconnect and finally it terminates at the SRAMs. So this is like the path a store instruction would take (writing back a value from the register file to the SRAMs).

- This is the end of the exercise, if you are curious feel free to experiment a bit with constraints in OpenSTA or with the synthesis commands in Yosys.

3.8 A Last Note

In the last section of the Yosys flow I simplified things a bit. You don't *need* to map the flip-flops before calling ABC. If you map the flip-flops before calling ABC, it will only receive the combinational elements, even if you use `abc -dff`. If you want to perform sequential optimizations in ABC, you need to execute ABC before running `dfflibmap` but sequential synthesis currently has some major downsides/problems.

The biggest one is the following:
Even in the combinational case, ABC works strictly per-module which decreases the optimization potential since the module boundaries cannot be changed. In the sequential case this problem gets even worse because here ABC will receive each clock-domain in each module. This might seem reasonable (and it is to some extend) but the problem comes from what is considered to be a clock-domain. Every unique combination of enable, reset and clock wire going to a flip-flop is a clock domain. the clock is obvious, reset is also mostly the same among all flip-flops but in one module you may have many different enable signals for different flip-flops.

Another big downside is that you lose the naming of the flip-flops and you often cannot recover them, this is a major problem in peripherals where you might need the names to write constraints.

So in practice, you first prepare the generic flip-flops in Yosys by mapping enable signals and synchronous resets into soft logic. There are some other things you need to deal with for compatibility reasons, they are explained in Basilisks synthesis script[13]. You probably also do not want to run it on your whole design. Instead you may want to consider splitting your design into a synchronous part where you want sequential optimization and everything else (clock domain crossings, peripherals and so on) where you do not want it. Then you run synthesis twice with either option and blackbox the stuff you do not want in this run.

[13] https://github.com/pulp-platform/cheshire-ihp130-o/blob/basilisk-dev/target/ihp13/yosys/scripts/yosys_synthesis.tcl#L172