# Open-Source Tool Installation

## Introduction

Welcome! This document serves as a introduction to using open-source tools to create a manufacturable chip from a HDL description. For this, we will be using the following open-source tools:

- Icarus Verilog, which is a Verilog compiler and simulator, helps one develop and test their own Verilog code.
- GTKWave, which is a waveform viewer, helps one debug their code by visualizing simulation data.
- Yosys, which is a synthesis tool, turns an HDL description into a list of gates and their connections.
- PDNGEN creates a custom power grid that ensures that the gates receive the proper voltage and current.
- RePlace, which is a global placement tool, creates an initial placement solution that will be legalized in detailed placement.
- OpenDP, which is a detailed placement tool, is used to legalize the global placement output.
- TritonCTS, which is a CTS (clock tree synthesis) tool, is used to create a clock tree, which a structure that distributes the clock such that every component receives it at the optimum time.
- FastRoute, which is a global routing tool, is used to allocate routing resources to plan out the way in which wires should travel such that areas are not overly congested.
- TritonRoute, which is a detailed routing tool, is used to create the actual wires for the chip based on the global routing result.
- Klayout, which is a layout viewer and editor, is used to generate the final GDSII file and to do DRC.
- OpenROAD, which is a collection of open-source tools, including many of the ones mentioned above, controlled by scripts and environment variables that turn an HDL description into a manufacturable output.
- Magic, which is a layout viewer and editor, is used to extract an equivalent topological circuit for LVS (layout vs schematic), and an equivalent circuit including parasitic capacitance and resistance for SPICE simulation.

- Netgen, which is an LVS tool, is used to compare the implemented chip against a schematic to ensure that the design is implemented properly.
- NgSpice, which is a SPICE simulator, is used to simulate our final design at the circuit level to ensure that it is functioning properly and conforming to specifications.

To help progress through this guide, a Github repository has been created. This primarily contains scripts used later in the tutorial, as well as scripts that automate certain steps of the process. This repository can be found below:

https://github.com/rlc113/Open-Source-VLSI-Tools-Tutorial

## Known Open-Source Tools

Below is a list of known open-source tools, listed by type:

**Verilog Compiler/Simulators:**

- Icarus Verilog – functions both as a compiler and a simulator. Much more details are provided later on.
- Verilator – functions both as a compiler and a simulator. Its main advantage seems to be speed, but requires certain Linux distros or Cygwin (or something equivalent). Also supports SystemVerilog
- CVC – functions both as a compiler and a simulator. It features an interpretative mode that helps the user debug.
- TKGate – Has a full GUI interface for simulation and circuit visualization, but is rather dated, so it is likely slow.
- VeriWell – functions both as a compiler and a simulator. Again, rather dated, so it is likely slow.

**Waveform Viewers:**

- GTKWave – Details are provided later on.
- Dinotrace – Another waveform viewer, but requires certain Linux distros or Cygwin (or something equivalent).
- VCDROM – Online waveform viewer – could be useful for quick debugging without installation.

**Synthesis Tools:**

- Yosys – OpenROAD's chosen synthesizer – this is the flagship open-source synthesizer of those currently available. It supports Verilog, a subset of SystemVerilog, and, with a certain plug-in, VHDL.
- Odin II – Verilog Synthesizer. It is mainly for FPGAs, but it can take a Verilog input, and create a BLIF output using simplified components such as basic AND, OR, etc.
- ABC – Logic optimizer used to simplify/speed up sequential circuits. It is used by Yosys for exactly this purpose.

**Power Network Generation/Floorplanning Tools:**

- PDNGEN – OpenROAD's chosen power grid generator.
- Parquet – Floorplanning tool based on simulated annealing – It is rather dated, so it may be slow.
- Graywolf – Mainly a placement tool, but it can do floorplanning as well – It is forked off of the last open-source version of a tool called TimberWolf, which was bought a couple of years ago. It is part of the Qflow toolflow.

**Placement Tools**

- RePlace – OpenROAD's chosen global placement tool.
- OpenDP – OpenROAD's chosen detailed placement tool.
- Graywolf – See above.
- Etesian – Full placer, and is part of the Coriolis toolkit.
- DREAMPlace – Deep learning based placement tool. Claims to achieve a 30x speedup over RePlace.
- LibrEDA – Placement and routing tool.
- Dragon – Full placer, featuring a fast mode that provides decent solutions quickly.

**CTS Tools**

- TritonCTS – OpenROAD's chosen CTS tool.
- Graywolf – This tool also has the capability to do CTS.

**Routing Tools**

- FastRoute – OpenROAD's chosen global routing tool.
- TritonRoute – OpenROAD's chosen detailed routing tool.

- Graywolf – See floorplanning section.
- Katana – Full router, and is part of the Coriolis toolkit.
- LibrEDA – See above
- Qrouter – Router based off Lee maze router algorithms, and is part of Qflow.
- Fairly Good Router – Global, self-contained routing tool.
- CUGR – Global routing tool, developed a few years ago.

**DRC Tools**

- Alliance DRC – Alliance is a set of tools that includes a DRC tool.
- Klayout – Layout editor (more on that in a later section) that is capable of running DRC. This is what OpenROAD uses.
- Magic VLSI – Layout editor (more on that in the next section) that is capable of running DRC.

**LVS Tools**

- Netgen – Purely an LVS tool that can compare between many different file formats, including Verilog, CDL, and SPICE.
- OpenRAM LVS Visualizer – Tool developed recently that is interactive and can interface with Magic, allowing for smoother debugging with circuit layout issues.
- Klayout – Layout editor (more on that in a later section) that is capable of running DRC. This is what OpenROAD nominally uses, although the script is not compatible with SKY130 at the time of writing.

**SPICE Simulators**

- NgSpice – SPICE simulator that is natively supported by SKY130. It has excellent documentation, and has, among other features, has an interactive interpreter for post-simulation analysis.
- Gnu Circuit – Simulator that is not directly based off SPICE, but uses some SPICE models., and should be compatible with certain ones
- QucsStudio – SPICE simulator with GUI to help visualize circuits. Only compatible with Windows at time of writing.

**Layout Viewers/Editors**

- Magic VLSI – Layout editor with a wide variety of functionality, including layout editing, parasitic extraction, and design rule checking.
- OpenROAD GUI – OpenROAD's built-in GUI. Since it is built to work directly with the flow, it can easily display information about timing reports, generate heat maps, and display clock trees, among other things. However, it likely will not work easily with other flows.
- Klayout Layout Viewer and Editor – OpenROAD uses this program to create the final GDS file. Has features such as the ability to view multiple layouts at once, and offers a scripting environment, enabling autonomous processes, if desired.
- Glade Layout and Schematic Editor – Layout editor with built-in DRC, LVS, and circuit extraction. Can be installed on Windows.
- Xic Graphical Input Editor – Layout editor with capability to do DRC, LVS, and parametric extraction. It can also open files remotely and process batches of data while suppressing the GUI for increased speed.

**Toolflows**

- OpenROAD – This is the toolflow we will be using in this tutorial. Much more detail about this will be provided later on.
- OpenLANE – Another toolflow that uses portions of openROAD with other tools, such as Magic.
- Coriolis – Toolkit that does placement and routing, and can be used in conjunction with other tools to create an ASIC.
- Qflow – Digital synthesis toolflow that features a GUI to facilitate understanding. It can also target FPGAs if desired.

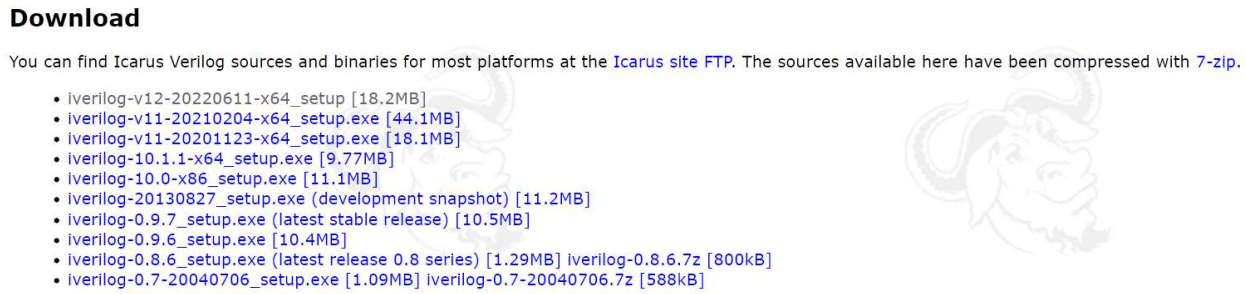
## Icarus Verilog Information, Installation, and Use

Icarus Verilog is an open-source Verilog compiler and simulator. One can use it to test and debug their Verilog code. One cannot do waveform analysis with it, but it is possible to interface with other software that has the ability to.

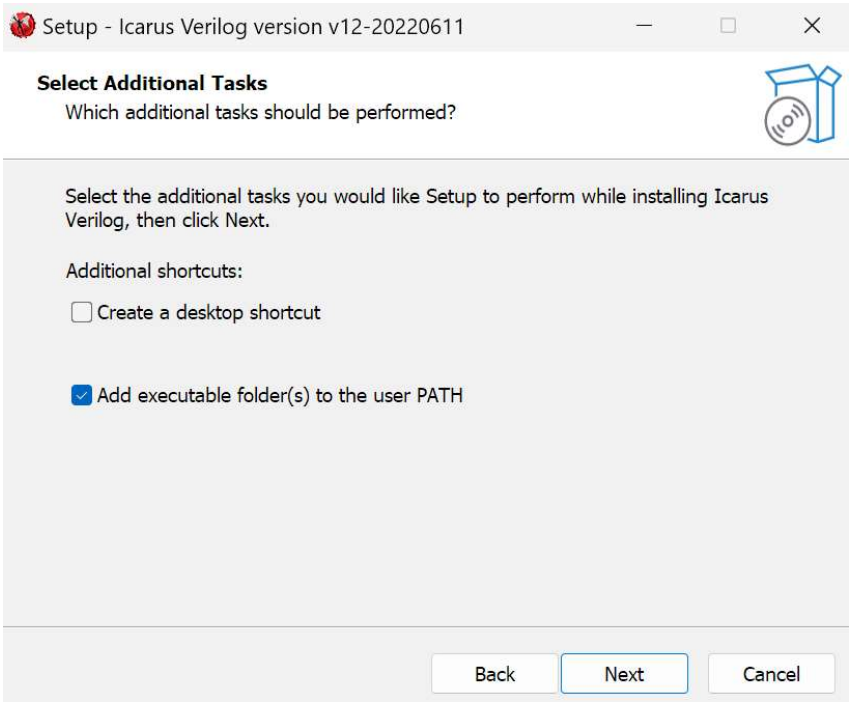
To install Icarus Verilog, please visit the following website:

https://bleyer.org/icarus/

From there, click the top link to download the installer:




Please use the default options for installation, except for the last step, where one must select the "Add executable folder(s) to the user PATH" option, which will allow one to execute the program without needing to be in the same location as the executable:




To make sure that the installation succeeded, run "iverilog -h" in a cmd window. If you receive the following, installation was successful:

```
C:\Users\megar>iverilog -h
Usage: iverilog [-EiSuvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005|-g2005-sv|-g2009|-g2012] [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-L moduledir]
                [-M [mode=]depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] [-l file] source_file(s)

See the man page for details.
```

If you receive a message saying the command is not recognized, the executable is likely not on the PATH environment variable. To fix this, add the location of the executable (likely located in "C:\iverilog\bin" or equivalent) to the PATH.

A good reference on how to get started with Icarus Verilog can be found on the Github documentation page:

https://steveicarus.github.io/iverilog/

(In particular, the "Getting Started with Icarus Verilog" and "Waveforms with GTKWave" pages are quite useful.)

In this document, we will show how to do the essential operations using Icarus Verilog, but for more advanced functionality, it is best to consult the above website.

To compile a Verilog file (i.e transform it into an executable that can be passed to the simulator), one can run "iverilog -o counter.o counter.v", where "counter.v" is an example Verilog file, and "counter.o" is our output. However, if our simulation needs to use two files, such as a module and a test bench, then we must either use an include directive, or we must include both files in our command, such as "iverilog -o counter.o counter_tb.v counter.v". Another option, mentioned in the "Getting Started with Icarus Verilog" page, is to use a command file, in which one can list all of the files necessary to their design. Also mentioned on the same page is the "-s" option, which allows the user to select what is called a root module, which is necessary for complicated designs where your root module is referenced by something else, or if you have several candidate root modules (i.e. if you have multiple testbench files.)

To run the Verilog simulation, one can simply run "vvp counter.o". All output will be displayed to the command window, or, if one uses the "-l" option, a specified logfile. There is also an interactive debug mode, which is detailed in the "VVP Interactive Mode" page.

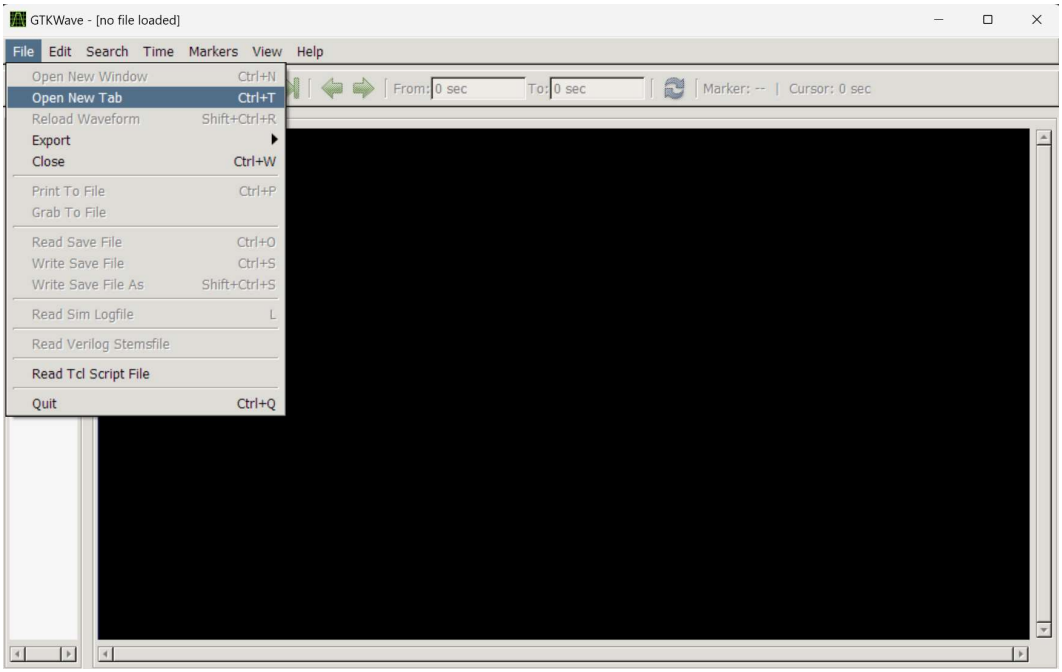
Finally, to interface with a waveform viewer, we must "dump" the signals into a VCD file. To do this, one must first configure the file to dump using the "$dumpfile" function, such as "$dumpfile("counter.vcd")". From there, the specific signals that must be dumped must also be outlined using the "$dumpvars" command. For example, if we wish to output our clock, reset, and output, we can use "$dumpvars(0, clk, rst, out)". We can also use it to output all signals

within a module – for example, if our counter module is instantiated as “DUT”, we can call “$dumpvars(0, DUT)” to dump all signals associated with the counter (it also dumps signals associated with the submodules of the design.)

## GTKWave Information, Installation, and Use

GTKWave is an open-source waveform viewer that allows users to easily debug their Verilog simulations. This software is actually installed with Icarus Verilog (assuming default settings are used) and one can check that it is installed by running “gtkwave -h”. If a command not recognized error flies, again it is likely an issue with the PATH variable, and one will need to add the executable to the PATH (the executable should be located in the “gtkwave” folder of the Icarus Verilog installation.)

When one first runs the program, they will be greeted by an empty window. To view a VCD file, go to “File” -> “Open New Tab”, from which you can select the VCD file:




From there, the module will appear in the top left corner, and clicking on it will cause signals to appear in the lower left window, which can be added to the waveform by double clicking on them. GTKWave has a similar UI to other waveform viewers, so it should be straightforward to use, but if one run into issues, one can consult the user’s guide on the website below:

https://gtkwave.sourceforge.net/gtkwave.pdf

## Ubuntu Environment Setup

To use the rest of the open-source tools that are required for the flow, we must use a Linux system. For this tutorial, we will use a virtual machine, namely VirtualBox, to run Ubuntu 22.04.

There are plenty of resources available online for how to do this, with an excellent tutorial linked below:

https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox#1-overview

If you do end up following the tutorial, make sure to download Ubuntu 22.04 – if you are unable to find it on the linked webpage, try the following website:

https://releases.ubuntu.com/

There are also a couple of things to note while installing – firstly, several of the processes that the flow uses are rather CPU and memory intensive, so make sure enough resources are allocated to the VM. Also, do not select the "Pre-allocate Full Size" option, as we may need to increase the size of the hard disk later (for now allocate around 15 GB.) Finally, within the Ubuntu installation, make sure to select "Install Ubuntu" instead of "Try Ubuntu", select "Minimal Installation" instead of "Normal Installation" and check the "Download updates while installing Ubuntu" options.

After following the above steps and proceeding with the installation, you should have a fully functional Ubuntu virtual machine. However, before proceeding, we must install a few programs. There are two options for doing this. The first option is to install them as we need them, and this is the assumed option for the rest of the tutorial. The second option involves installing everything from a provided script called "Install.sh", located in the "Install" folder of the repository. Please note that this script should be ran with "sudo", and must be executable (by running "chmod +x Install.sh"). Before running this, however, there are a couple of variables one needs to configure.

Assuming that one uses the first option, there are a couple of packages that we need to install immediately after completing the installation. We can install these by running the following command:

"sudo apt install -y git python3 ngspice tk-dev libcairo2-dev mesa-common-dev"

## openROAD Information, Installation, and Use

openROAD is the main open-source toolflow that we will be using. It is a collection of open-source tools that does synthesis, floorplanning, placement, clock tree synthesis, routing, DRC, and LVS. It will allow us to transform an HDL description of a circuit, an example of which being a Verilog file, to a manufacturable layout.

To use openROAD, we will use the openROAD-flow-scripts repository, which contains a collection of scripts and open-source PDKs that are used to drive the openROAD process. The scripts are controlled by various environment variables, some of which are set by the PDKs, while others are set by design-specific config files. This setup is very convenient from a user

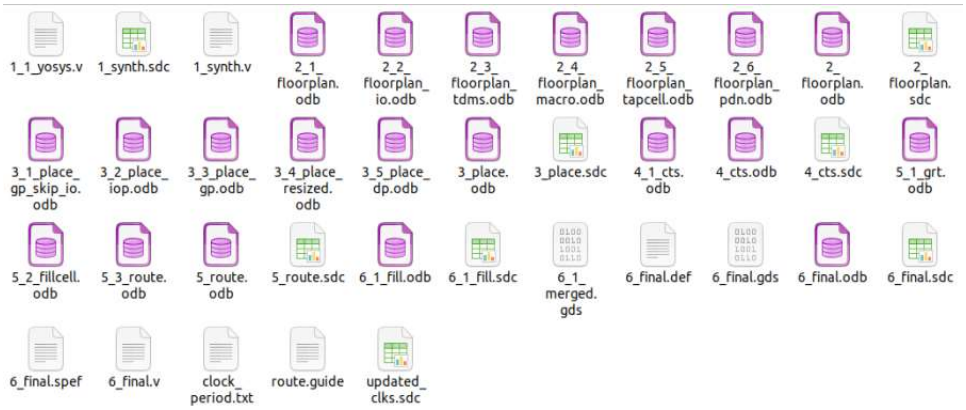
perspective, as it allows one to easily modify specific portions of the flow using only a couple of files.

To install this, please follow the instructions outlined in the link below:

https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/blob/master/docs/user/BuildLocally.md

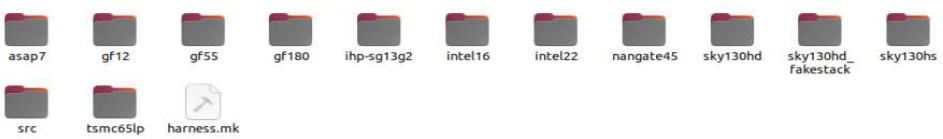
Again, there are couple of things to note – firstly, to fully install and demo the openROAD flow, one only needs to run the commands in the first three code blocks. However, the "git clone..." command should be run wherever you desire to install the repository. Also, these installations are rather finicky, and I would highly recommend leaving the virtual machine open and focused when installing and not doing anything else on the base computer – it has gotten permanently stuck for me before when attempting to leave it running in the background. Finally, the "source ./env.sh" must be run every time a new command window is launched so that the tool is able to run yosys and openroad from the command line.

If you were able to install openROAD-flow-scripts successfully and ran the final make, then your ".../OpenROAD-flow-scripts/flow/results/nangate45/gcd/base/" folder should look like this:




These files are outputs from each stage of the flow. In particular, the GDS files describe the physical masks, the SDC files describe constraints (that are usually copied from an original SDC file), while the ODB files are an internal file format that openROAD uses to communicate between stages. One might also notice that there are several Verilog files – the ones near the upper left corner describe the design after synthesis, while the ones near the bottom describe the design after it is fully implemented.

At this point, it is useful to discuss how the flow is controlled. If one travels to ".../OpenROAD-flow-scripts/flow/designs/", they will be greeted by an assortment of folders, each corresponding to a specific PDK:



For our designs, we are interested in using the Skywater 130nm process, which applies to the "sky130hd" and "sky130hs" folders. For now, let's focus on the "sky130hd" folder.

Opening this, we see the following folders:


These are various designs that are pre-configured for openROAD. Let's look at "aes":


There a couple of things to note here – the "config.mk" file is the main configuration file for the flow, and controls most of the major functionality. The "constraint.sdc" file imposes external restrictions on the design, such as maximum logic delays. The JSON files are there for additional functionality and are beyond the scope of this discussion.

Let's now focus our attention on the config file:

```
export DESIGN_NICKNAME = aes
export DESIGN_NAME = aes_cipher_top
export PLATFORM    = sky130hd

export VERILOG_FILES = $(sort $(wildcard ./designs/src/$(DESIGN_NICKNAME)/*.v))
export SDC_FILE      = ./designs/$(PLATFORM)/$(DESIGN_NICKNAME)/constraint.sdc

export PLACE_PINS_ARGS = -min_distance 4 -min_distance_in_tracks

export CORE_UTILIZATION = 20
export CORE_ASPECT_RATIO = 1
export CORE_MARGIN = 2

export PLACE_DENSITY = 0.6
export TNS_END_PERCENT = 100
```

The first three lines tells the flow which folders and modules to use. Specifically, the first line outlines where to find the Verilog and constraint files, the second line outlines what the top-level module is called, and the third line outlines which PDK to use.

The two lines after this tell the flow where to look for the Verilog and constraint files. Please note that the tool expects that the Verilog files are to be kept within a ".../designs/src/aes" folder, while it expects that the constraint files are to be kept in the same folder as the config file.

The line after this tells the flow how to space the pins – in this case they should be 4 tracks apart.

The three lines after this configure the dimensions of the chip. Specifically, the first line sets which percentage of the core (main circuit) area is dedicated to cell placement, while the remaining percentage is dedicated to routing. The second line sets the aspect ratio of the core

area, and the third line sets the amount of area that is left for the inputs. From these, the tool can automatically figure out how big the chip must be.

The second-to-last input line sets the placement density, which controls how close cells should be placed together, and the last input line sets the percentage of setup time violations to repair.

This config file is one of many included with the flow. Feel free to explore other files in the sky130hd folder. Some documentation on various config variables can be found below:

https://openroad-flow-scripts.readthedocs.io/en/latest/user/FlowVariables.html

Let's also have a look at the constraints file:

```
current_design aes_cipher_top

set clk_name  clk
set clk_port_name clk
set clk_period 5.6
set clk_io_pct 0.2

set clk_port [get_ports $clk_port_name]

create_clock -name $clk_name -period $clk_period $clk_port

set non_clock_inputs [lsearch -inline -all -not -exact [all_inputs] $clk_port]

set_input_delay  [expr $clk_period * $clk_io_pct] -clock $clk_name $non_clock_inputs
set_output_delay [expr $clk_period * $clk_io_pct] -clock $clk_name [all_outputs]
```

This constraints file abides by the Synopys Design Constraints format. Documentation for this can be found online, but we will explain the above lines in the following paragraph.

The first line tells the SDC file the top-level module name. The next four lines set up various variables that will be used for the rest of the file – clk_name is the name the SDC file calls the clock, while clk_port_name is the name of the clock within the top-level module. clk_period is the duration of the clock in nanoseconds, while clk_io_pct is the percentage of the clock that we expect the inputs and outputs to be delayed by. The latter is particularly important, as input delays could lead to setup time violations at the first layer of flip-flops, while output delays could lead to setup time violations outside the circuit. The last four lines of the file simply implement the functionality outlined by the first lines.

To implement our own custom design, we will have to do a few things. First, we need a Verilog file to implement. An example 8-bit counter is shown below, and provided in the repository in the “Verilog” folder, but anything that is syntheziable should do:

```
module counter(input clk,
               input rst,
               output reg [7:0] out);

    always @ (posedge rst or posedge clk) begin
        if (rst == 1) begin
            out <= 0;
        end else begin
            out <= out + 1;
        end
    end
endmodule
```

Please place your Verilog file in a ".../designs/src/nickname/" folder, where "nickname" is a name of your choosing.

Next, please copy the configuration file of aes to a ".../designs/nickname" folder, where "nickname" is the same as above. From there, please change the "DESIGN_NICKNAME" variable to the "nickname" variable that was previously used and the "DESIGN_NAME" variable to the name of your top-level module. In our counter example, I just used "counter" for both names, resulting in the following config file:

```
export DESIGN_NICKNAME = counter
export DESIGN_NAME = counter
export PLATFORM    = sky130hd

export VERILOG_FILES = $(sort $(wildcard ./designs/src/$(DESIGN_NICKNAME)/*.v))
export SDC_FILE      = ./designs/$(PLATFORM)/$(DESIGN_NICKNAME)/constraint.sdc
export PDN_TCL       = ./designs/$(PLATFORM)/$(DESIGN_NICKNAME)/pdn.tcl

export PLACE_PINS_ARGS = -min_distance 4 -min_distance_in_tracks

export CORE_UTILIZATION = 20
export CORE_ASPECT_RATIO = 1
export CORE_MARGIN = 2

export PLACE_DENSITY = 1
export TNS_END_PERCENT = 100
```

Also, one may note that there is a line specifying "PDN_TCL". This is needed to ensure that power pins will be created by openROAD, which is necessary for SPICE simulations later down the line. The PDN file is the exact same as the one supplied in the ".../flow/platforms/sky130hd" folder, but with the key difference being that the pins option is specified in the creation of the PDN grid:

```
####################################
# standard cell grid
####################################
define_pdn_grid -name {grid} -voltage_domains {CORE} -pins {met1 met4 met5}
add_pdn_stripe -grid {grid} -layer {met1} -width {0.48} -pitch {5.44} -offset {0} -followpins
add_pdn_stripe -grid {grid} -layer {met4} -width {1.600} -pitch {27.140} -offset {13.570}
add_pdn_stripe -grid {grid} -layer {met5} -width {1.600} -pitch {27.200} -offset {13.600}
add_pdn_connect -grid {grid} -layers {met1 met4}
add_pdn_connect -grid {grid} -layers {met4 met5}
```

If you wish to run SPICE simulations later, make sure that either the platform PDN.tcl file has this change, or that there is a copy in the design folder with the change.

Also, copy over the constraints file to the same folder and change the current_design variable value to the name of your top-level module – the file used for our counter is shown below:

```
current_design counter

set clk_name  clk
set clk_port_name clk
set clk_period 5.6
set clk_io_pct 0.2

set clk_port [get_ports $clk_port_name]

create_clock -name $clk_name -period $clk_period $clk_port

set non_clock_inputs [lsearch -inline -all -not -exact [all_inputs] $clk_port]

set_input_delay  [expr $clk_period * $clk_io_pct] -clock $clk_name $non_clock_inputs
set_output_delay [expr $clk_period * $clk_io_pct] -clock $clk_name [all_outputs]
```

(The config , PDN, and constraints files for the counter can also be found on the repository under the “config” folder.)

Finally, when all of the above is complete, one must modify the Makefile (found in “.../OpenROAD-flow-scripts/flow/Makefile”) such that the “DESIGN_CONFIG” variable points towards our config file – the file is configured for our counter below:

```
# DESIGN_CONFIG=./designs/intel16/aes/config.mk
# DESIGN_CONFIG=./designs/intel16/gcd/config.mk

# DESIGN_CONFIG=./designs/intel22/aes/config.mk
# DESIGN_CONFIG=./designs/intel22/gcd/config.mk
# DESIGN_CONFIG=./designs/intel22/ibex/config.mk
# DESIGN_CONFIG=./designs/intel22/jpeg/config.mk

# DESIGN_CONFIG=./designs/gf180/aes/config.mk
# DESIGN_CONFIG=./designs/gf180/ibex/config.mk
# DESIGN_CONFIG=./designs/gf180/jpeg/config.mk
# DESIGN_CONFIG=./designs/gf180/riscv32i/config.mk
# DESIGN_CONFIG=./designs/gf180/uart-blocks/config.mk

#DESIGN_CONFIG=./designs/ihp-sg13g2/aes/config.mk
#DESIGN_CONFIG=./designs/ihp-sg13g2/ibex/config.mk
#DESIGN_CONFIG=./designs/ihp-sg13g2/gcd/config.mk
#DESIGN_CONFIG=./designs/ihp-sg13g2/spi/config.mk
#DESIGN_CONFIG=./designs/ihp-sg13g2/riscv32i/config.mk

# Default design
#DESIGN_CONFIG ?= ./designs/nangate45/gcd/config.mk
DESIGN_CONFIG ?= ./designs/sky130hd/counter/config.mk

# Default TNS_END_PERCENT value
export TNS_END_PERCENT ?=5

# If we are running headless use offscreen rendering for save_image
ifndef DISPLAY
export QT_QPA_PLATFORM ?= offscreen
endif
```

Once this is all complete, running “make” in the flow folder (after ensuring the environment script has already been ran in the terminal) should run through the process. If successful, you should see results appear in the “.../flow/results/sky130hd/nickname/base” folder. To remove the flow outputs, one can run “make clean_all”.

Once the above is complete, we have a complete physical design. However, there are several checks that should be ran to ensure that the design is able to be manufactured and correct. For the former, we run what is called DRC (design rule check.) This checks a variety of rules to ensure that the design can be manufactured. For example, if two metal lines are too close together, there is a risk that a manufacturing error could cause them to short. As it turns out, openROAD has native support for DRC, which can be ran using the “make DRC” command (please note that, while this is supported for the “sky130hd” platform, it may not be supported for others. If it is not supported, the response will say so.) Running DRC will generate a

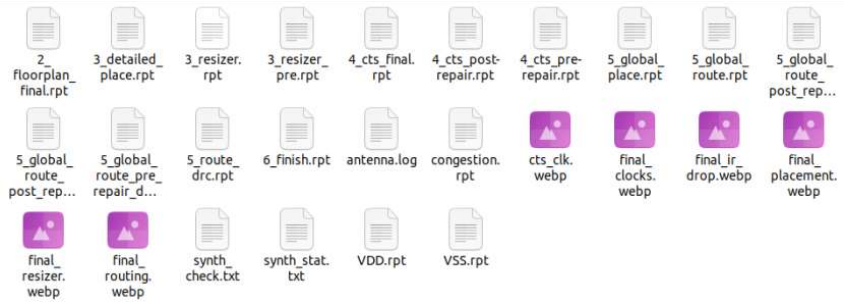
".../flow/reports/sky130hd/counter/base/6_drc_count.rpt" file. If this file just contains "0", then your design has passed DRC. If your design does not pass DRC, you could try to fix it manually, but it is likely a better idea to increase the amount of space available on the chip for now.

In terms of LVS, unfortunately, the native toolflow does not support LVS for this platform. To do this, we will need to install the SKY130 PDK directly. This will be the focus of the section after the next.

## openROAD Tips and Tricks

This section contains some pertinent information about additional desired functionality and certain issues one might encounter using the flow.

- Reports – It is worth noting that during the process flow, certain reports are generated in the ".../flow/reports/sky130hd/nickname/base" folder – our counter example is shown below:




- These reports are mostly timing reports – the most important of which is "6_finish.rpt". For this file, we will notice that, at the top, the worst slack is reported. If this value is positive, it means that the data makes it "in time", and no violations take place. However, if it is negative, timing was violated, meaning that the specified clock was too restrictive. Thus, one should be sure to check this file after the flow is complete, and if timing is violated, consider relaxing the clocking constraint.

- The demonstrated method of declaring the core size may be undesirable as it does not give the user direct control over the process. Also, for smaller designs, this method is more prone to failure due to edge cases. Either way, one can manually declare the exact size of the core and die in the following manner (the dimensions are in micrometers):

```
export DIE_AREA    = 0 0 300 300
export CORE_AREA   = 20 20 280 280
```

- If the flow quits during placement due to a utilization error, there wasn't enough area allocated on the chip to accommodate all sites. To fix this, either increase the core area, or decrease utilization, depending on the method used.

- If the flow quits during placement due to a density error, try changing the placement density to the suggested density. If that doesn't work, or leads to routing issues, try increasing the area of the chip using one of the methods mentioned above.
- If the flow quits during routing, it is likely due to not allocating enough space. Again, try increasing the area of the chip using one of the methods mentioned above.

If you encounter an error that is not mentioned above, openROAD's official documentation is linked below:

https://openroad.readthedocs.io/en/latest/index.html

Also, the “issues” section of both the openROAD and openROAD-flow-scripts repositories is a good place to check:

https://github.com/The-OpenROAD-Project/OpenROAD/issues

https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/issues

Finally, if one searches the command in question in the source code of openROAD, they should be able to find a document explaining the exact functionality of the options, which may be helpful in resolving the error:

https://github.com/The-OpenROAD-Project/OpenROAD

## Magic VLSI and SKY130 Installation

To do LVS, and later, SPICE simulation, we must install the SKY130 PDK. In particular, to save on memory, we will only install the sky130hd cell library in the SKY130A variant. With this, we only need around 2 GB of space. By contrast, if we were to install the entire PDK, it would take around 50 GB.

However, before we can install the PDK, we first must install Magic. This is because the PDK files that are necessary for Magic to function with it require Magic to already be installed (We need Magic to do circuit extraction for LVS and SPICE simulation later.) To install Magic, we first clone the repository:

git clone https://github.com/RTimothyEdwards/magic.git

After this, we run the following commands (after changing directory to access the cloned repository):

./configure

make

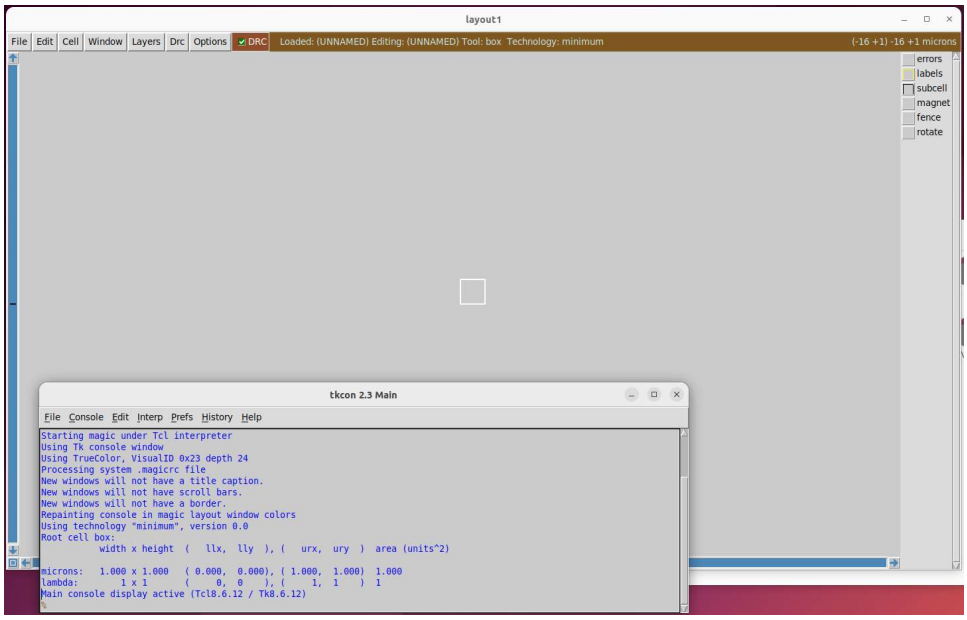
sudo make install

If you encounter issues with the above commands, please consult the website below:

http://opencircuitdesign.com/magic/

Pay close attention to the ./configure command – if it is reporting that some requirement is not present, such as Tk, then you likely did not run the "sudo apt install..." at the end of the Ubuntu section.

To confirm that the installation was successful, please run "magic" in your terminal. You should see the following windows appear:




Now that we have Magic installed, we can install the SKY130 PDK. To do this, we will use the "Open-PDKs" repository, which contains a variety of scripts that will automatically install specified PDKs. It will also create files that help integrate the PDKs with open-source tools. To make use of this repository, we must first clone it, as shown below:

git clone https://github.com/RTimothyEdwards/open_pdks.git

The following instructions show how to install a small segment of what the above repository is capable of. If you wish to install other PDKs, feel free the consult the installation documentation below:

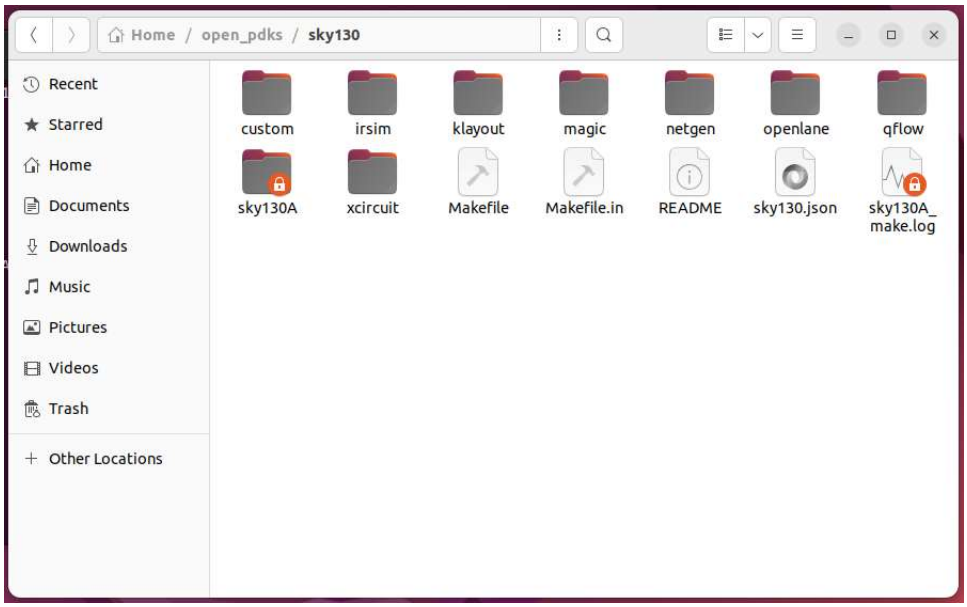
http://opencircuitdesign.com/open_pdks/install.html

Before installing the PDK, we must configure which ones we wish to install. We do this below:

./configure --enable-sky130-pdk --with-sky130-variants=A --enable-primitive-sky130 –disable-io-sky130 --disable-sc-hs-sky130 --disable-sc-ms-sky130 --disable-sc-ls-sky130 --disable-sc-lp-sky130 --disable-sc-hdll-sky130 --disable-sc-hvl-sky130 –disable-gf180mcu-pdk

(We also install the primitives library for SPICE simulations later)

Next, run “sudo make” in the same directory. After this, your “.../open_pdks/sky130/” folder should look like this:




The “sky130A” folder is where all of the PDK files were installed. Finally, run “sudo make install” to copy the files to a different directory, which, importantly, should be independent of where the repository was cloned so the following instructions will work.

After we have installed the SKY130 PDK, we must ensure that Magic has access to the TCL files. We do this by creating a symbolic link below:

sudo ln -s /usr/local/share/pdk/sky130A/libs.tech/magic/* /usr/local/lib/magic/sys/

After the above command is run, try running “Magic -T sky130A”. If you are successful, the top of the layout window should report “Technology: sky130A”.

## Netgen Installation and LVS

Now that we have Magic working with the SKY130 PDK, we can discuss how we will do LVS. However, before discussing how the process works, it is worth mentioning that the LVS process is rather convoluted. As such, we have created a script called “LVS.sh” in the LVS folder that will automatically do the steps outlined in the following paragraphs. However, if this is your first time doing LVS using the toolflow, I would highly recommend following the below steps so that you obtain a working understanding of how it works, such that, if there are issues with the script in the future, you are able to debug and fix them.
That being said, let’s continue with the step-by-step explanation. By using a certain script in the openROAD toolflow, one can generate a CDL file, which is a type of schematic description of

what the tool thinks it has implemented on the chip (it is likely different from the synthesized description since the design most likely had to modified during placement and routing.) To check that this is actually what was implemented, we will use Magic to extract a SPICE file from the GDS file (which is the file containing the output masks) and use a program called “Netgen” to do LVS. Thus, the next logical step is to install Netgen. To do so, we first clone the repository below:

git clone https://github.com/RTimothyEdwards/netgen.git

Then, we install netgen by running the following in its folder:

./configure

make

sudo make install

If a window shows when one runs “netgen”, then installation was successful. Also, documentation for it can be found below in case one runs into problems using it:

http://opencircuitdesign.com/netgen/

Next, we shall use Magic to extract a SPICE file from the GDS file. We do this by using the below TCL script:

```
# Read the GDS file
gds read "/home/tutorial/openroad_script_files/OpenROAD-flow-scripts/flow/results/sky130hd/
counter/base/6_final.gds"

# Load the module that was just read
load counter

# Extract the module to an .ext file
extract

# Configure the ext2spice command for LVS
ext2soice LVS

# Extract the module to a SPICE file
ext2spice counter

# Terminate
exit
```

(This can also be found under the “LVS” folder.)

There are a couple of change for this script to adapt it to your own module. Firstly, the top line loads the GDS file, so please change this to whatever path that your design was generated on. The second line loads the module into the viewer, so please change the module name to whatever was set to your “DESIGN_NAME” variable in your config file. Finally, the second-to-last line converts the “.ext” file to a SPICE file so again, please change “counter” to the “DESIGN_NAME” value.

To run this, simply do “magic -T sky130A -noconsole -dnull Magic_Extraction.tcl”, where “Magic_Extraction.tcl” is the TCL script. If successful, a SPICE file should appear in the folder, along with around thirty “.ext” files.

Next, we need to generate the CDL file that represents our schematic. We do this by running the openROAD program on the following TCL script:

```
#Read in the final output of the toolflow
read_db /home/tutorial/openroad_script_files/OpenROAD-flow-scripts/flow/results/sky130hd/counter/
base/6_final.odb

#Write the CDL file
write_cdl -masters /home/tutorial/openroad_script_files/OpenROAD-flow-scripts/flow/platforms/
sky130hd/cdl/sky130hd.cdl 6_final.cdl
```

(This can also be found under the “LVS” folder.)

Again, there are a couple of changes that need to be made. The read_db command reads the output “odb” file, so change that accordingly, also, the write_cdl command needs a “masters” file, which can be found in the platforms folder, so change the line accordingly. Finally, the script currently outputs to the user directory, so if that is not desired, feel free to change the second argument. The script can be ran by running:

.../OpenROAD-flow-scripts/tools/install/OpenROAD/bin/openroad -exit cdl.tcl

If successful, you should now have a CDL file – however, we must have it reference the PDK SPICE file, otherwise the LVS will fail. To do this, please make sure the following include directive is at the top of the CDL file:

```
1 * CDL Netlist generated by OpenROAD
2
3 .INCLUDE /usr/local/share/pdk/sky130A/libs.ref/sky130_fd_sc_hd/spice/sky130_fd_sc_hd.spice
4
5 *.BUSDELIMITER [
6
7 .SUBCKT counter clk out[0] out[10] out[11] out[12] out[13]
8 + out[14] out[15] out[16] out[17] out[18] out[19] out[1] out[20]
9 + out[21] out[22] out[23] out[24] out[25] out[26] out[27] out[28]
10 + out[2] out[3] out[4] out[5] out[6] out[7] out[8] out[9] rst
```

(A python script that automatically does this can also be found under the “LVS” folder)

Finally, we are ready to do LVS! To do so, first open netgen by running “netgen” in a command prompt. Then, run the following command:

lvs {SPICE_FILE DESIGN_NAME} {CDL_FILE  DESIGN_NAME} /usr/local/share/pdk/sky130A/libs.tech/netgen/sky130A_setup.tcl comp.out

where SPICE_FILE is the extracted SPICE file, CDL_FILE is the CDL file outputted by openROAD, and comp.out is a log file that will be saved in the location in which netgen was invoked. If LVS succeeds, the following message should appear at the end of execution:

```
Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

At this point, we have conducted both DRC and LVS, confirming that our design is both manufacturable and physically does what it should do. However, it is also critical to simulate our design with parasitics to get a more accurate picture of how it might function in the real world. This is the focus of the next section.

## SPICE Simulation

Before doing SPICE simulation, we must extract the equivalent circuit using Magic VLSI. We can do this by using the script below:

```
#Read in the design
gds read /home/tutorial/openroad_script_files/OpenROAD-flow-scripts/flow/results/sky130hd/
counter/base/6_final.gds

#Load and flatten the design - necessary for resistance extraction
load counter
flatten design_flatten
load design_flatten

#Select the top cell and do extraction
select top cell
extract all

#Convert the extracted file to a sim file
ext2sim labels on
ext2sim

#Extract resistance
extresist

#Convert extraction to SPICE with resistances
ext2spice lvs
ext2spice cthresh 0
ext2spice extresist on
ext2spice

#Exit
exit
```

(This script can be found under the “NgSpice” folder.)

To adapt this file to the design of interest, there are two necessary changes to be made. On the first line, the GDS file is loaded into Magic – as such, one should change the path so that it points to their design. Further, when we load the design in, it is assumed to be called “counter”, but this should be changed to whatever the “DESIGN_NAME” variable was in the config file. Also, it is worth noting that this design will export a SPICE file called “design_flatten.spice”, regardless of what the design is called. We then run this script using “magic -T sky130A -noconsole -dnull Magic_Extraction_PEX.tcl”, where “Magic_Extraction_PEX.tcl” is the TCL script.

To do SPICE simulation, we shall use NgSpice, an open-source SPICE simulator that is directly supported by the SKY130 PDK. The developers have also provided an excellent user manual,

which should serve as an effective reference to implement more advanced functionality than what this tutorial will show, if desired.

To help get started with simulations, we have provided a simple Python script, called "SPICE_formatter.py" to automatically custom simulation based on user input. This script also requires a script template, called "template.cir", as well as the final Verilog and SPICE files to detect ports. Both the "SPICE_formatter.py" and "template.cir" files can be found on the repository under the "NgSpice" folder.

To use this file, there are several settings that must be changed. Firstly, the "unmodified_sim_file" variable must correspond to the "template.cir" file found on Github. Secondly, the "modified_sim_file" variable must correspond to the path of the file that you wish to generate. Thirdly, the "SPICE_file" must correspond to the SPICE file that was just extracted. Fourthly, the "verilog_file" variable must correspond to the "6_final.v" file found in the results folder.

Once these variables are set, the code may be run. Doing so, the code will first prompt the user as to whether the design has a clock, and, if so, ask for its name. If it has a clock, the code will assume that the design also has a reset and ask for its name – since, usually, unless there is a very specific functionality that one wishes to target, synchronous designs must have a reset. However, if you are sure that a reset is not necessary, then just specify some other input as the reset, and adjust it based off the information provided later (please note that the following sentences may mention that the code will ask for information relating to the clock or reset, but if you do not initially specify it as such, then it will not.) The code will also ask if the reset is an active high, meaning that a reset value of '1' resets the circuit. It will then ask the user for the desired clock period, i.e. how often the clock should pulse, and reset time, i.e. how long the reset should be held at '1' or '0', for active high and active low resets, respectively. After that, it will ask for simulation time and precision. What one puts down for precision will affect simulation time and accuracy, but I have found that 40 ps is about the highest that is reasonable, so use this if unsure. Next, if there are other inputs than just the clock and reset, the code will ask the user if they wish to statically power the input, i.e. just set it to '1' or '0', or to set it to a custom source that they can freely enter. Finally, the code will ask if the user wishes to plot the clock and reset of the course of operation, as well as the number of output signals to plot and their names. They will all be displayed on the same graph.

If successful, one should obtain an output file like below:

```
Test netlist
.lib "/home/tutorial/open_pdks/sky130/sky130A/libs.tech/ngspice/sky130.lib.spice" tt
.include "/home/tutorial/openroad_script_files/OpenROAD-flow-scripts/flow/results/sky130hd/counter/base/counter.spice"

X0 VDD VSS clk out.0 out.10 out.11 out.14 out.15 out.17 out.18 out.19 out.1 out.21 out.26 out.27 out.2 out.3 out.8 out.9

V0 VSS 0 DC 0
V1 VDD 0 DC 1.8V
V2 clk 0 DC PULSE(1.8V 0 0ns 1fs 1fs 5.0ns 10.0ns)
V3 rst 0 DC PULSE(1.8V 0 18.0ns 1fs 1fs 1000000ns 1000000ns)

.tran 40.0ps 100.0ns uic

.control
    run
    let POWER_VECTOR = -i(V1) * v(VDD)
    meas tran AVG_POWER AVG POWER_VECTOR from=1ns to=99.0ns
    plot clk rst out.0 out.1
.endc

.end
```

While the script that we have provided is a useful tool to get started with simulation, if one wishes to implement more complicated functionality, or just wishes to make a slight change, such as increasing the simulation time, it is far better to just edit the output file directly. Thus, we will provide a brief explanation on what each line in the above file does. For more-depth explanations, feel free to consult the manual.

The first line of the file is just the title of the netlist – this is unimportant, but one can change it to be more distinctive if they desire. Next, the ngspice library that is installed within the openPDK is included. The final “tt” corresponds to the process corner, which are manufacturing configurations of mobility. “tt” corresponds to “typical-typical”, “ff” corresponds to “fast-fast”, and “ss” corresponds to “slow-slow”, all of which are self-explanatory. After the library is included, the design is also included, and shortly after which, it is instantiated. The next two lines set up the VSS and VDD power supplies. After this, the clock and reset are configured using the PULSE voltage source – the clock signal is a repeating pulse, while the reset pulses just once to ‘0’ and stays there for 1 million nanoseconds. It is very unlikely that the desired simulation time will need to be higher than this, but if so, this value will need to be increased. Also, if there are input voltages, they will appear in this block. However, in this case, they are not here. After this block, the transient simulation is configured – the first time corresponds to the desired precision of the simulation, while the second corresponds to the desired simulation time. The final ‘uic’ parameter tells the simulation to use specified initial conditions and, if they are not found, zero voltage for initial capacitor values. After transient simulation is specified, we arrive at the control block. Inside this, the simulation is run, then the next two lines calculate the average power and report it in the console. Finally, the signals are plotted and the control block and design are closed.

To run the simulation, first execute the “ngspice” command. This will open up the ngspice console. In it, one must run “set ngbehavior=hsa” for compatibility reasons with Magic, and, if desired “set num_threads=X”, where X is the number of threads, for better performance if the virtual machine was allocated multiple cores. Finally, running “source out.cir”, where “out.cir” was the file generated by the Python script, will start simulation. It this, the tool will eventually

display a line showing the “Reference value”, which is just the current simulated time in seconds. Immediately after the simulation concludes, the tool will print out the average power in Watts, and a new window will open showing the desired plot. Also, the console will still be present after simulation – this is useful, as it allows you to create other plots and see other data in case your original specified plot was unclear.