Hardware and Embedded Security

Lab 1: Digital Design Flow

The aim of this lab is to introduce the digital design flow by using open source Computer Aided Design (CAD) instruments and technologies.

1 Open technology

In order to design an Integrated Circuit (IC) we need a technology made of rules, cells, ... in other words a Process Design Kit (PDK). There are few open PDK available today. One of the most famous is the Google-Skywater 130 nm PDK [1].

2 Open CAD tools

Despite most of manufactured ICs are developed with commercial CAD tools, there is a growing interest in open CAD tools. Indeed in 2022 the IEEE/ACM International Symposium on Microarchitecture, hosted a tutorial on the *OpenROAD* project [2].

3 Platform setup

Even if it is possible to install from scratch both open CAD tools and technology, we can save time by exploiting the already available Virtual Machine (VM), which has been prepared for the IEEE Micro 2022 tutorial [3] with Virtual Box [4], and available on DropBox [5]. The VM is based on an Ubuntu distribution and comes with a user, who is in the `sudo` list. The user is `user` and the password is `openroad` (useful to enable `sudo` operations). Indeed, to start working with the VM we need to add few other elements, in particular a user-friendly editor and a Hardware Description Language (HDL) simulator. A simple editor, which can work for the purpose, is `gedit`. It is part of a standard package, so it can be installed as: `sudo apt-get install gedit`.

Other editors (such as *Emacs*) offer further features, such as language *statements*[1]. *Icarus Verilog* is a simple and free HDL simulator for Verilog and is available as a standard package. Since it has no integrated waveform viewer, it is possibile to dump signals from the HDL model in Value Change Dump (VCD) and view them with a *vcd* viewer, such as *GTKWave*, which is available as a standard package too:

```
sudo apt-get install iverilog
sudo apt-get install gtkwave
```

Finally, we need a complete PDK library, which can be dowloaded from the course website (file *sky130_fd_sc_hd.tar.gz*) and unpacked in your home directory (`/home/user`)[2].

3.1 PDK structure

The Sky130 PDK comes with several libraries, which are named using the scheme *Process Name_Library Source Abbreviation_Library Type Abbreviation_Library Name*. In our case the *Process Name* is `sky130`, the *Library Source Abbreviation* is `fd` (SkyWater Foundry), the *Library Type Abbreviation* is `sc` (Standard Cells) and the *Library Name* is `hd` (High Density).

3.2 Standard cells

Standard cells are the basic blocks we use to build a digital IC. For each cell the PDK provide users with several different views, such as simulation (verilog - .v) models, timing models (liberty - .lib), geometry models (library exchange format - .lef) and even the layout (graphycal design system - .gds). Several formats (e.g. .v, .lib, .lef) are ASCII representations, so they can be viewed with a text editor. On the contrary, other formats (e.g. .gds) are binary representations, so a tool is needed to view them. Moreover, each cell comes with diffenrent strenghts, namely to amount of current it can drive at the output. Strenght equal to one is minumum and corresponds to transistor minimum width.

[1] sudo apt-get install emacs

[2] You can unpack it with `tar xvzf sky130_fd_sc_hd.tar.gz`

3.2.1 Verilog model

The file `sky130_fd_sc_hd/verilog/sky130_fd_sc_hd.v` contains the simulation models of all the cells. As an example there are different modules for the inverter (`inv`) as there are inverters with different output strenghts (e.g. `inv_1`, `inv_2`). However, they all implement the same logic function. Moreover, these models rely on primitives (see the file `primitives.v`) which are part of the simulation model.

3.2.2 Liberty model

The delay of a logic gate depends on several parameters related to variabilities in the manufacturing process (e.g. variations in the doping concentration). Since gates are made of n-type Metal Oxide Semiconductor (MOS) Field Effect Transistors (FETs) and p-type MOSFETs, one has a typical condition (T) for nMOSFETs and pMOSFETs, leading to the TT condition, and four corner cases. The corner cases are the four combinations one can obtain with nMOSFETs/pMOSFETs faster than expected (F), nMOSFETs/pMOSFETs slower than expected (S). Usually only the best (FF) and worst (SS) corners are considered. Moreover, the dealy depends also on the temperature and power supply. For these reasons, the folder `sky130_fd_sc_hd/lib` contains files obtained for different corners, tempertaure and power supply (e.g. `tt_025C_1v80` is typical-typical, at a temperature equal to 25 Celsius and power supply equal to 1.80 V).

3.2.3 LEF models

Variability of the manufacturing process reflects also on variation of the electrical resistance of the layes. For this reason there are different LEF models, taking into account the different corner cases (see the `sky130_fd_sc_hd/techlef` folder). Moreover, geometry information of each cell is available in the `sky130_fd_sc_hd/lef` folder.

3.2.4 GDS model

To view the *layout* of the cells, we use *klayout*. As an example from `/home/user` you can issue:

`klayout ./sky130_fd_sc_hd/gds/sky130_fd_sc_hd.gds`
When the tool opens, on the left-hand side you have the list of available cells. You can select `sky130_fd_sc_hd__inv_1`, then with the right button you select *Show As New Top* and the layout of the inverter opens. You can see the nMOSFET and pMOSFET. If you open `sky130_fd_sc_hd__inv_2`, you can see that the transistors have a larger width than `sky130_fd_sc_hd__inv_1`.

4 Project structure

To ease the different steps, it is recommended to have a clear organization of files and folders. As an example `lab` as the main folder of the project, `src` as the sub-folder for HDL sources, `tb` as the sub-folder for the test-bench, `sim` as the sub-folder for the simulation, `netlist` as the sub-folder for the result of the logic synthesis, `pnr` as the sub-folder for the place and route.

4.1 Design simulation

Let's assume you want to describe and simulate an n-bit counter, where n is a parameter, which is fixed at design time. The corresponding Register Transfer Level (RTL) Verilog descrption is shown in Listing 1. We assume it is placed in the `src` sub-folder as `counter_gen.v`.

Listing 1: n-bit counter

```
module counter_gen
  #(
    parameter NBIT = 4)
   (
    input                  clk,
    input                  rstn,
    output reg [NBIT-1:0] out);

   always @ (posedge clk, negedge rstn) begin
      if (! rstn)
        out <= 0;
      else
        out <= out + 1;
   end

endmodule
```

In order to simulate the counter, we need to stimulate it, so we have to generate the required inputs. In this example the inputs

are only the clock (`clk`) and the asynchronous reset (`rstn`). Inputs generation to stimulate the circuit is obtained by writing a test-bench, namely an HDL file, where the we instantiate the circuit and describe how input signals change as function of the time. An example is shown in Listing 2. We assume it is placed in the `tb` sub-folder as `tb_counter_gen.v`.

Listing 2: Test-bench for an n-bit counter

```
`timescale 1ns/1ns
module tb_counter_gen();

    localparam TCLK_2 = 5;
    localparam TCLK = TCLK_2*2;

    localparam RST_CYCLES = 2;
    localparam RST_TIME = RST_CYCLES*TCLK;

    localparam SIM_CYCLES = 20;
    localparam SIM_TIME = SIM_CYCLES*TCLK;

    localparam NBIT = 8;

    reg                     clk, rstn;
    wire [NBIT-1:0]         cnt;

    counter_gen #(.NBIT(NBIT) ) dut(clk, rstn, cnt);

    initial begin

        $dumpfile("tb_counter_gen.vcd");
        $dumpvars;

        clk=0;
        forever #(TCLK_2) clk=~clk;

    end

    initial begin

        rstn=0;
        #(RST_TIME);
        rstn=1;
        #(SIM_TIME)
        $finish ;

    end

endmodule
```

As is can be observed, the test-bench includes two statements to open a VCD file (`$dumpfile(``tb_counter_gen.vcd'');`) and to dump signals (`$dumpvars;`).

To simulate the design we rely on *Icarus Verilog*, which requires two steps: i) compile the Verilog files, ii) execute the simulation. In

the first step the tool checks the syntax and generates a .vvp file, which is the one to be launched in the second step to perform the simulation. From the `sim` sub-folder you can issue:

```
iverilog -o tb_counter_gen.vvp ../src/counter_gen.v \
../tb/tb_counter_gen.v
vvp tb_counter_gen.vvp
```

The simulation will generate the file `tb_counter_gen.vcd`, which contains the waveforms of the simulation. You can view it with *GTKWave*:

```
gtkwave tb_counter_gen.vcd
```

4.2 Design implementation

4.2.1 Logic synthesis

The first step to implement the counter is the logic synthesis. We assume you are in the `netlist` sub-folder. To start the logic synthesis we first call the logic synthesizer:

```
yosys
```

This opens the shell of the logic synthesizer, where we can issue commands. Listing 3 shows the sequence of commands required to synthesize the counter and map it to the standard cells.

Listing 3: Yosys synthesis commands

```
read_liberty -lib /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
read_verilog -defer ../src/counter_gen.v
chparam -set NBIT 8
synth -top counter_gen
dfflibmap -liberty /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
abc -liberty /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
write_verilog ./counter_gen.v
```

First we provide the logic synthesizer with the standard cells information by reading the liberty file (`read_liberty`). Then, we read the verilog RTL model (`read_verilog`) giving the possibility to specify the value of the parameters later (`-defer`). We set the `NBIT` parameter to 8 (8-bit counter) and we run the synthesis on the *counter_gen* design. Then, we map the flip-flops and internal gates on the ones available in the technology library (`dfflibmap` and `abc`). Finally, we write the netlist in the file named `counter_gen.v`.

In order to check that the netlist still behaves as the RTL de-

scription, we can reuse the test-bench, *Icarus Verilog* and *GTKWave* to simulate the netlist. Of course this step requires to use the `sky130_fd_sc_hd.v` file, namely the behavioural model of the standard cells instantiated in the netlist. When compiling the model of the cells we have to define a symbol specifying that we want to use *functional* models (`FUNCTIONAL`); we also specify a value for the clock-to-output delay of flip-flops (`UNIT_TIME`). Since the `NBIT` parameter has been fixed for the synthesis, the compiler will generate a warning stating that the parameter assigned to the unit under test by the test-bench is not present. Listing 4 shows the sequence of commands to compile the HDL and run the simulation.

Listing 4: Netlist simulation

```
NET_DIR=../netlist
DESIGN=counter_gen
TB_DIR=../tb
TB_NAME=tb_counter_gen

TECH_NAME=sky130
TECH_TYPE=fd_sc_hd
LIB_NAME=${TECH_NAME}_${TECH_TYPE}
LIB_DIR=/home/user/${LIB_NAME}/verilog

iverilog -DFUNCTIONAL -DUNIT_DELAY=#1 -o ${TB_NAME}.vvp ${LIB_DIR}/primitives.v \
         ${LIB_DIR}/${LIB_NAME}.v ${NET_DIR}/${DESIGN}.v ${TB_DIR}/${TB_NAME}.v \
         -s ${TB_NAME}
vvp ${TB_NAME}.vvp
```

The simulation will generate the file `tb_counter_gen.vcd`, which contains the waveforms of the simulation. You can view it with *GTKWave*:

```
gtkwave tb_counter_gen.vcd
```

4.2.2 Place and route

In the following we assume that the place and route flow, including all the commands, is run from the `pnr` sub-folder.

Setup and initialization Once we have the netlist, we can generate the layout of the circuit by placing the cells and connecting them together. However, this step requires several sub-steps to be completed. First, we can launch the place and route tool (*OpenROAD*). In order to have a graphical view of the result we can activate the graphic user interface:

`openroad -gui`

Like most of the CAD tools, *OpenROAD* accepts tool command language (tcl) commands and scripts. As shown in Listing 5, first we have to read the liberty and LEF models of the library (`read_liberty` and `read_lef`). Then, we read the netlist (`read_verilog`) and we create a database for the current design (`link_design`). Finally, we provide the tool with a set of timing constraints in the form of a Synopsys Design Constraint (SDC) file (see Listing 6), where we set the clock period (3.74 ns in this example) and the maximum delay on all inputs (except the clock) and outputs (748 ps in this example).

Listing 5: Place and route initialization

```
read_liberty /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
read_lef /home/user/sky130_fd_sc_hd/techlef/sky130_fd_sc_hd__nom.tlef
read_lef /home/user/sky130_fd_sc_hd/lef/sky130_fd_sc_hd.lef

read_verilog ../netlist/counter_gen.v
link_design counter_gen
read_sdc counter_gen.sdc
```

Listing 6: Synopsys Design Constraint file

```
set clk_name clk
set clk_port_name clk
set clk_period 3.74
set clk_period_factor 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 delay [expr $clk_period * $clk_period_factor]
set_input_delay $delay -clock $clk_name $non_clock_inputs
set_output_delay $delay -clock $clk_name [all_outputs]
```

The sequence of commands shown in Listing 5 can be directly written in the TCL shell of *OpenROAD*. Otherwise, we can create a TCL file, such as *design_init.tcl*, containing the sequence of commands, and run in the TCL shell of *OpenROAD* as:
`source ./design_init.tcl`

Floorplanning In order to obtain the final layout of the circuit we first have to specify how the available surface is organized. This step is referred to as *floorplanning*. As shown in Listing 7, with

the `initialize_floorplan` command we specify the area of the *die* (i.e. the piece of semiconductor wafer used for the chip) and the area of the *core* (i.e. the part of the *die* used for implement the circuit). The areas are rectagular shapes and are specified as lower left x/y and upper right x/y coordinates. Finally, we specify the *site*, the smallest unit of placement where the smallest cell can be placed. Furthermore, we have to set the tracks (see the script *make_tracks.tcl*, namely the lines on which metal layers are drawn. Their geometry must comply with the *offset* and *pitch* values defined layer by layer in the LEF files.

Listing 7: Floorplanning

```
initialize_floorplan -site "unithd" -die_area {0 0 50 50} -core_area {5 5 45 45}
source /home/user/sky130_fd_sc_hd/make_tracks.tcl
```

Placement According with the design flow suggested in *OpenROAD*, the placement relies on different sub-steps, which can be roughly grouped in *pre-placement*, *placement* and *post-placement adjustment and optimization*, as shown in Listing 8.

Listing 8: Placement

```
# Pre-placement
tapcell -distance 14 -tapcell_master sky130_fd_sc_hd__tapvpwrvgnd_1

source /home/user/sky130_fd_sc_hd/sky130hd.pdn.tcl
pdngen

# Placement
global_placement -skip_io -density 0.65 -pad_left 1 -pad_right 1

place_pins -hor_layer met3 -ver_layer met2

global_placement -density 0.65 -pad_left 1 -pad_right 1

set dont_use_cells "sky130_fd_sc_hd__probe_p_8 sky130_fd_sc_hd__probec_p_8 \
    sky130_fd_sc_hd__lpflow_bleeder_1 \
    sky130_fd_sc_hd__lpflow_clkbufkapwr_1 \
    sky130_fd_sc_hd__lpflow_clkbufkapwr_16 \
    sky130_fd_sc_hd__lpflow_clkbufkapwr_2 \
    sky130_fd_sc_hd__lpflow_clkbufkapwr_4 \
    sky130_fd_sc_hd__lpflow_clkbufkapwr_8 \
    sky130_fd_sc_hd__lpflow_clkinvkapwr_1 \
    sky130_fd_sc_hd__lpflow_clkinvkapwr_16 \
    sky130_fd_sc_hd__lpflow_clkinvkapwr_2 \
    sky130_fd_sc_hd__lpflow_clkinvkapwr_4 \
    sky130_fd_sc_hd__lpflow_clkinvkapwr_8 \
    sky130_fd_sc_hd__lpflow_decapkapwr_12 \
    sky130_fd_sc_hd__lpflow_decapkapwr_3 \
    sky130_fd_sc_hd__lpflow_decapkapwr_4 \
    sky130_fd_sc_hd__lpflow_decapkapwr_6 \
```

```
    sky130_fd_sc_hd__lpflow_decapkapwr_8 \
    sky130_fd_sc_hd__lpflow_inputiso0n_1 \
    sky130_fd_sc_hd__lpflow_inputiso0p_1 \
    sky130_fd_sc_hd__lpflow_inputiso1n_1 \
    sky130_fd_sc_hd__lpflow_inputiso1p_1 \
    sky130_fd_sc_hd__lpflow_inputisolatch_1 \
    sky130_fd_sc_hd__lpflow_isobufsrc_1 \
    sky130_fd_sc_hd__lpflow_isobufsrc_16 \
    sky130_fd_sc_hd__lpflow_isobufsrc_2 \
    sky130_fd_sc_hd__lpflow_isobufsrc_4 \
    sky130_fd_sc_hd__lpflow_isobufsrc_8 \
    sky130_fd_sc_hd__lpflow_isobufsrckapwr_16 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_hl_isowell_tap_1 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_hl_isowell_tap_2 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_hl_isowell_tap_4 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_isowell_4 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_isowell_tap_1 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_isowell_tap_2 \
    sky130_fd_sc_hd__lpflow_lsbuf_lh_isowell_tap_4"

set_dont_use $dont_use_cells

# Post-placement adjustment and optimization
repair_design

set_placement_padding -global -left 0  -right 0

detailed_placement

improve_placement

optimize_mirroring
```

Often, in modern technologies (tapless standard cells), the p-substrate of nMOSFETs and the n-well of pMOSFETs in a cell are not directly connected to the ground and to the power supply to save area. Instead, libraries have special cells for the purpose (*tap cells*) which are placed in a regular interval in each row of placement. The `tapcell` command is the first command in the *pre-placement* phase and is used to place *tap cells*, where the suggested cell is the *tapvpwrvgnd_1*.

Then, we add the power grid to distribute the power supply voltage and the ground with the *sky130hd.pdn.tcl* script and the `pdngen` command.

The *placement* phase relies on the `global_placement` and `place_pins` commands, where we specify the density (i.e. the amount of space utilised for placement of cells out of the total available space). It is recommended to keep the density not too high (roughly no more than 70%) to have enough space for routing. Similarly, cells padding leaves some space in mulitples of the *site* width to reduce placement

congestion.

Finally, the *post-placement* phase starts defining which cells should not be used in current design. Indeed, the library contains also cells for insering probe points (the *probe_p_8* and *probec_p_8*) and for multi-power domains (the ones which name starts with *lpflow*), which are not useful in this case. Then, the following commands fix timing violations, cells usage and legal placement.

Clock tree synthesis The distribution of the clock signal is very important to be sure that synchronous elements are receiving it with a controlled delay (skew). This is usually achieved by structuring the clock distribution as a binary tree structure, with buffers/inverters to balance the delay for all clock inputs. Listing 9 shows the commands to synthesize the clock tree.

Listing 9: Clock tree synthesis

```
repair_clock_inverters
clock_tree_synthesis -root_buf sky130_fd_sc_hd__clkbuf_4 \
                     -buf_list sky130_fd_sc_hd__clkbuf_4 \
                     -sink_clustering_enable \
                     -sink_clustering_size 30 \
                     -sink_clustering_max_diameter 100 \
                     -balance_levels

set_dont_use $dont_use_cells
repair_clock_nets

set_placement_padding -global -left 0 -right 0
detailed_placement

check_placement -verbose
```

As it can be oserved the first command (`repair_clock_inverters`) is a directive to place inverters in the clock tree to balance the delay. Then, the `clock_tree_synthesis` command synthesizes the clock tree using the *clkbuf_4* cell as both the tree root and the buffer in clock wire segments. Other parameters are used to further optimize the clock tree synthesis. All the sinks with the same input timing are grouped in clusters (`sink_cluster_enable`). To limit the number of sinks (*size*) and the distance among sinks in a cluster (*diameter*) the `sink_clustering_size` and `sink_clustering_max_diameter` are set. Finally, we suggest the clock tree synthesis tool (*TritonCTS*) to try to keep a similar number of levels in the clock tree across clock-gates (buffers, inverters, ...)

Then, we check: i) that during the clock tree synthesis only "allowed" cells are used; ii) if it is required to insert buffers in the wire from the clock input pin to the clock root buffer; iii) if all the clock tree cells have legal placement.

Routing Before rounting the circuit we place *fillers* (see Listing 10), which are dummy cells required to continue the power supply and ground rails, n-wells, ... First, we define the set of cells used as *fillers*, namely the ones which name starts with *fill*. Then, we place the *fillers* (`filler_placement`).

Listing 10: Fillers placement

```
set fill_cells "sky130_fd_sc_hd__fill_1 sky130_fd_sc_hd__fill_2
                sky130_fd_sc_hd__fill_4 sky130_fd_sc_hd__fill_8"

filler_placement $fill_cells
check_placement
```

The routing step (see Listing 11) defines wiring connection among cells exploiting metal layers available in the PDK. We can constraint the router to use only certain metal layers during the routing phase (`set_routing_layers`) and we can instruct the router to reduce the routing resources for certain layers by a certain percentage, such as 30% (`set_global_routing_layer_adjustment`). Then, we start the *global routing* (`global_route`) specifying the maximum number of iterations to solve congestion problems.

Before running the *detailed routing* (`detailed_route`) we check that there is no *antenna violations* (`check_antennas`). Antenna violations come from the fact that during the chip manufacturing there can be large amounts of charge in some metal wires. If this wires are connected to the gate of MOSFETs, the charge might damage the gate oxide layer of MOSFETs. The LEF file contains *antenna* rules, namely on a layer-by-layer basis the maximum ratio between the area of wire and the area of the gates connected to the wire.

Listing 11: Routing

```
set_routing_layers -signal met1-met5
set_global_routing_layer_adjustment met1-met5 0.3

global_route -congestion_iterations 100 -verbose

check_antennas -report_violating_nets
```

```
detailed_route
```

Finishing To finish the flow few other steps are required (see Listing 12. Stemming from the design rules imposed by the PDK, when metal layers are manufactured, one of the steps in the process is physical polishing of the fill layers in order to grind down the surface and attain a uniform flatness (`density_fill`). *OpenROAD* receives this information as a *json* file, which is part of the *sky130* PDK.

Then, we check i) the timing by analysing the worst *slack* i.e. the available timing margin with respect to the required clock period; ii) the total area of the circuit.

Finally, we write the verilog model, which can be simulated with *Icarus Verilog* and *GTKWave* as we did after the logic synthesis.

Listing 12: Finishing

```
density_fill -rules /home/user/sky130_fd_sc_hd/fill.json

report_worst_slack
report_design_area

write_verilog design_pnr.v
```

References

[1] https://github.com/google/skywater-pdk.

[2] https://theopenroadproject.org/.

[3] https://the-openroad-project.github.io/micro2022tutorial/content/0-prep.html.

[4] https://www.virtualbox.org/.

[5] https://www.dropbox.com/s/vnql935fqf6yqk9/openroad-tutorial-micro2022.ova?dl=0.