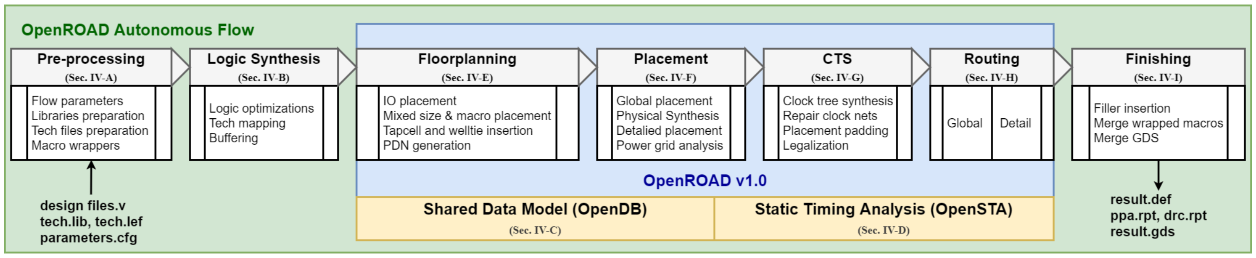
Productivity Tools for IC Design (EE2510). Oct-Nov 2025. Exercise Set 4.

Section 1 Internals of OpenRoad

I assume that you have run the flow for at least one design in the examples:

```
$ cd OpenROAD-flow-scripts/
$ cd ./flow/
#/ edit makefile to enable a specific design:
DESIGN_CONFIG=./designs/nangate45/gcd/config.mk
$ ./utils/docker_shell /bin/bash    # start the docker container to use the flow
container> make
#/ once make completes, the design results - .def files, and others are placed in ./results,
whereas reports (summary of results) are kept in ./reports folder.
```



1. OpenRoad Internal Usage:
As discussed in class a
“LEF defines the elements of an IC process technology and associated library of cell models. DEF defines the elements of an IC design relevant to physical layout, including the netlist and design constraints. LEF and DEF inputs are in ASCII form.”
“Library Exchange Format (LEF) file contains library information for a class of designs. Library data includes layer, via, placement site type, and macro cell definitions.” and
“A Design Exchange Format (DEF) file contains the design-specific information of a circuit and is a representation of the design at any point during the layout process. ..DEF conveys logical design data to, and physical design data from, place-and-route tools. Logical design data can include internal connectivity (represented by a netlist), grouping information, and physical constraints. Physical data includes placement locations and orientations, routing geometry data, and logical design changes for backannotation. Place-and-route tools also can read physical design data, for example, to perform ECO changes.”

OpenRoad uses OpenDB database format to store both the LEF and DEF files. “The structure of OpenDB is based on the Cadence Design Systems text file formats LEF (library) and DEF (design) formats version 5.6. OpenDB supports a binary file format to save and load the design much faster than using LEF and DEF.”

OpenRoad includes commands to read and write openDB files. Find a .lef file in the flow/ folder,

```
$ cp platforms/sky130hd/lef/sky130_fd_sc_hd_merged.lef ./test.lef
$ openroad
Openroad> set db [odb::dbDatabase_create]
Openroad> set lib [odb::read_lef $db "test.lef"]
Openroad> set tech [$lib getTech]
# # now you can print information read from the .lef file.
puts "LEF version: [$tech getLefVersion]"
puts "LEF version string: [$tech getLefVersionStr]"

puts "manufacturing grid size: [$tech getManufacturingGrid]"
puts "case sensitive: [$tech getNamesCaseSensitive]"
puts "num routing layers: [$tech getRoutingLayerCount]"
puts "num vias: [$tech getViaCount]"
puts "num layers: [$tech getLayerCount]"
puts "units: [set units [$tech getLefUnits]]"

##See:https://github.com/The-OpenROAD-Project/DAC-2020-Tutorial/tree/master/2_database
_access
```

2. We can use the information contained in .lef and .def files to generate reports. (see https://github.com/The-OpenROAD-Project/DAC-2020-Tutorial/tree/master/4_generating_reports) For this analysis, first copy the 6_final.def file from the design's result folder and constraint.sdc file from design folder. For example (assume we are in flow folder)

```
$ cp ./results/nangate45/gcd/base/6_final.def ./platforms/nangate45/
$ cp ./designs/nangate45/gcd/constraint.sdc ./platforms/nangate45/
$ cd ./platforms/nangate45/
$ openroad
openroad> read_lef "./lef/NangateOpenCellLibrary.tech.lef"
read_lef "./lef/NangateOpenCellLibrary.macro.mod.lef"
read_liberty "./lib/NangateOpenCellLibrary_typical.lib"
read_def "6_final.def"
read_sdc "constraint.sdc"
set_propagated_clock [all_clocks]
report_checks -path_delay min -fields {slew cap input nets fanout} -format
full_clock_expanded
report_checks -path_delay max -fields {slew cap input nets fanout} -format
full_clock_expanded
report_power
report_design_area
```

3. The openroad workflow tool Klayout is a GDS file editor and viewer. Here we use it to see the final generated GDS. Starting in the folder "flow", do the following:

```
$ cd ./results/nangate45/gcd/base/
$ klayout    # launch klayout
In the klayout window open the file "6_1_merged.gds" from the current folder, it will visualize
the design.
```

Section 2: Exercises

0. For these experiments we want to have a design which gets compiled in lowest possible time. Write some shell commands to sort on size, the verilog files available in flow/designs/src folder. Pick up the smallest design and use it below.
For illustration, I am using nangate7 with gcd design.

The following exercises are the modified versions of the exercises present in:
https://github.com/The-OpenROAD-Project/micro2022tutorial

1. In the design flow, placement density and core utilization are two important parameters. Core utilization refers to the fraction of the core area that is occupied by standard cells (logic gates, flip-flops, etc.) after placement, not including routing, buffers, or whitespace. We don't want this to be too low (say <50%) or too high (>80-90%). In the former case silicon area is wasted, whereas in latter case, the tighter packing can cause routing congestion, DRC (Design Rule Check) violations, longer wires or higher delay, difficulty in inserting clock tree or filler cells. It is specified during the floor planning stage:
> initialize_floorplan -utilization 70 -aspect_ratio 1.0 -core_space 1000
Will try to get 70% utilisation in core area of 1000 um^2.

Placement density refers to the local cell packing density used during placement, essentially, how tightly cells can be packed within each placement bin (region). Lower placement density can ease routing and allow better timing.

In this exercise, you are going to change the utilisation of the design in config.mk file (designs/nangate45/gcd/config.mk) and collect statistics for these utilisation. To do this write following commands:
A. Use sed command to replace "export CORE_UTILIZATION ?= 50" with different values (50, 60, 70, 80, 90).
B. Run make
C. Store the relevant files from the generated results and reports to a folder named "exercise1".
D. Go back to A till the list is over.
E. Analyse the data collected to find out worst slack, power and design_area and compare them.
F. Write a makefile recipe to do this automatically.

2. In this exercise, we have following verilog code and its constraints:
alu.v

```
module alu #(
  parameter WIDTH = 12
)(
  input  wire             clk,   // Clock
  input  wire             rst,   // Active-high reset
  input  wire [WIDTH-1:0] a, b,  // ALU operands
  input  wire [      3:0] sel,   // ALU function select
  output reg  [WIDTH-1:0] out    // ALU output
);

  // Define functions
  localparam ADD  = 4'h0; // Addition
  localparam SUB  = 4'h1; // Subtraction
  localparam MULT = 4'h2; // Multiplication
  localparam DIV  = 4'h3; // Division
  localparam LSL  = 4'h4; // Logical shift left
  localparam LSR  = 4'h5; // Logical shift right
  localparam RL   = 4'h6; // Rotate left
  localparam RR   = 4'h7; // Rotate right
  localparam AND  = 4'h8; // Bitwise AND
  localparam OR   = 4'h9; // Bitwise OR
  localparam XOR  = 4'ha; // Bitwise XOR
  localparam NOR  = 4'hb; // Bitwise NOR
  localparam NAND = 4'hc; // Bitwise NAND
  localparam XNOR = 4'hd; // Bitwise XNOR
  localparam GT   = 4'he; // Signed greater than
  localparam EQ   = 4'hf; // Equal

  always @(posedge clk) begin
    if(rst) begin
      out <= WIDTH'b0;
    end else begin
      case(sel)
        ADD : out <= a + b;
        SUB : out <= a - b;
        MULT: out <= a * b;
        DIV : out <= a / b;
        LSL : out <= a << 1;
        LSR : out <= a >> 1;
        RL  : out <= {a[WIDTH-2:0],a[WIDTH-1]};
        RR  : out <= {a[0],a[WIDTH-1:1]};
        AND : out <= a & b;
        OR  : out <= a | b;
        XOR : out <= a ^ b;
        NOR : out <= ~(a | b);
        NAND: out <= ~(a & b);
        XNOR: out <= ~(a ^ b);
        GT  : out <= $signed(a) > $signed(b);
        EQ  : out <= a == b;
      endcase
    end
  end

endmodule
```

constraint.sdc:

```
set clk_period 7
```

```
# Don't change below here
current_design alu
set clk_name core_clock
set clk_port_name clk
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]
```

A. Create a folder in ./design folder corresponding to the above design. This is under nangate45.
B. Compile to see if it produces any error. Use config.mk similar to the one used in gcd. Find the effect as the placement_density is varied from 0.4 to 0.9.
C. Now change the frequency of operation to observe the slack and timing violations. To do this, use this config.mk

```
export DESIGN_NAME = alu
export PLATFORM    = nangate45

export VERILOG_FILES = ./design/src/alu/$(DESIGN_NAME).v
export SDC_FILE      = ./design/nangate45/alu/constraint.sdc
export ABC_AREA      = 1

export CORE_UTILIZATION = 45
export PLACE_DENSITY = 0.50
export CORE_ASPECT_RATIO = 1
export CORE_MARGIN = 1.0
```

Then modify the clk_period in constraint.sdc from 3 to 10 ns.
Generate reports for power, area and max frequency used by the design under different clk_period. Max frequency is determined by : fmax = 1 / (clk_period − worst_slack).
Plot these quantities as a function of the clk_period.
D. Write a makefile recipe for things done in C.

3. Follow https://openroad-flow-scripts.readthedocs.io/en/latest/tutorials/FlowTutorial.html for use of GUI and tcl commands for optimisations of the designs.