

# EECS 151/251A ASIC Lab 4: Floorplanning, Placement and Power

Written by Nathan Narevsky (2014, 2017) and Brian Zimmer (2014)  
Modified by John Wright (2015,2016) and Taehwan Kim (2018)

## Overview

This lab consists of two parts. For the first part covered in the lab session, you will then learn how the tools can create a floorplan, placement standard cells, and route power nets. By the end you should have a mostly complete layout of the design, including pads to connect the chip to the external world. However, the design you will write has many IO pins, so we will use a modified version with fewer pins (provided for you) to complete this task.

Then, you will be writing a GCD coprocessor as an assignment to practice Verilog and also understand some of the concepts that could be useful for the construction of general-purpose CPU (like your final project).

To begin this lab, get the project files by typing the following command

```
git clone /home/ff/eecs151/labs/lab4
cd lab4
```

If you have not done so already you should add the following line to your `bashrc` file (in your home folder) so that every time you open a new terminal you have the paths for the tools setup properly.

```
source /home/ff/eecs151/tutorials/eecs151.bashrc
```

Note: after doing this, you need to manually source your own `.bashrc` file or open a new terminal. You can test whether your path is setup properly or not by typing `which dc_shell` and making sure that it does not say `command not found` or `no dc shell in <path>`.

## Understanding the File Structure

In the `lab4` folder there are a few subfolders, some of which you have seen in the previous lab. The `dc-syn` folder contains the files to run through design compiler, which maps the design into logic gates from our technology library. The `vcs-sim-rtl` folder contains the files necessary to simulate the behavioral verilog design. The `vcs-sim-g1-syn` folder contains the files necessary to simulate the design after it has been mapped into logic gates. In most real designs this is only done to ensure that the logic is equivalent, it is usually not done with timing enabled until after running place and route. The `vcs-sim-g1-par` folder is where that simulation would be run after the design is run through the place and route tool. The `icc-par` folder is where the design is placed and routed, which is where we will be focusing on this lab.

## Understanding the Example Design

Now, let's take a look at how to run the placement flow. We'll be using provided verilog for this exercise. Take a look at the `gcd.v` file. There is a new module that is instantiated called `gcd_scan`, and the inputs and outputs to this module have changed. Previously the `gcd` module had 16-bit inputs for A and B, but now we are using a scan chain to cut down on the number of pads. Let us look at the contents of the `gcd_scan.v` file:

```

module gcd_scan#( parameter W = 16)
(
    input scan_in,
    input scan_clock,
    output scan_out,
    input scan_enable, //to avoid muxing clock between
    // scan mode and update mode for the result

    input [W-1:0] result_bits_data,
    output [W-1:0] operands_bits_A,
    output [W-1:0] operands_bits_B
);

reg [W-1:0] scan_result_bits_data;
reg [W-1:0] scan_operands_bits_A;
reg [W-1:0] scan_operands_bits_B;

always @(posedge scan_clock) begin
    if(scan_enable) begin
        scan_operands_bits_A <= {scan_in,scan_operands_bits_A[W-1:1]};
        scan_operands_bits_B <= {scan_operands_bits_A[0],scan_operands_bits_B[W-1:1]};
        scan_result_bits_data <= {scan_operands_bits_B[0],scan_result_bits_data[W-1:1]};
    end else begin
        scan_result_bits_data <= result_bits_data;
    end
end

assign scan_out = scan_result_bits_data[0];
assign operands_bits_A = scan_operands_bits_A;
assign operands_bits_B = scan_operands_bits_B;

endmodule

```

In this file we take in an enable, a data bit, and a clock, as well as output a data bit to read out data from the module and ensure that it is operating properly. The structure of the scan chain itself simply takes in a serial stream of bits and outputs a parallel bus of bits. When `scan_enable` is low, the scan chain loads in the output of the `gcd` module so that when it is done computing it can shift those bits out serially.

This is not the top level design anymore, since we are going to include connections for the top level module signals to pads. If you look at the `gcd_io.v` file you can see the connections to these pads, and the instantiation of the `B4I1025_NS` pad cell. Do not worry about the details about this cell, just keep in mind that the signals are going to be connected to them.

The final top level design that we are going to be running through the tools is found in `top.v`. If you scroll to the bottom of the file, you should be able to see that it instantiates a `gcd` module as well as a `gcd_io` module, and connects them together.

## Simulating the Design

To simulate the design you simply need to go to the `vcs-sim-rtl` folder and type:

```
make run
```

This should print out some output telling you that the simulation passed correctly. If you want to look at more of the signals you should open up the waveform viewer like in the previous labs with the command:

```
dve -full64 -vpd vcdplus.vpd &
```

Take a note of the difference in clock frequencies between the clock that is connected to the scan chain and the clock that is connected to the GCD module. The clock for the scan chain will be significantly slower than the one that connects to the GCD module. The reason behind this is that the IO cells are very slow, so in the actual design the scan path that is directly connected to them cannot operate as quickly. The GCD module gets its inputs from the outputs of the scan chain, so we need to be sure to synchronize them properly since they are crossing clock domains. We will discuss later in this lab how the constraints are setup so that the tools will handle this properly.

## Compiling the Design

Now that you understand (at least at a high level) the changes to the design, it is time to compile the new design. Since we covered Design Compiler in detail in the last lab, all you need to do this time is the following:

```
cd dc-syn  
make
```

This will run through the steps to compile the design into logic gates, and will create the files necessary to start the physical implementation of the design. After that finishes, simply go to the `icc-par` folder with the following command:

```
cd ../icc-par
```

## Initializing the Design

Run the following command:

```
make init_design_icc
```

This will create a new build folder and copy over the necessary files, as well as run through a few setup commands for IC Compiler. It will return you to the `icc_shell` so that you can interactively walk through the procedure.

If you run `start_gui` from the shell you can load up the gui, and you should see something similar to the picture below.

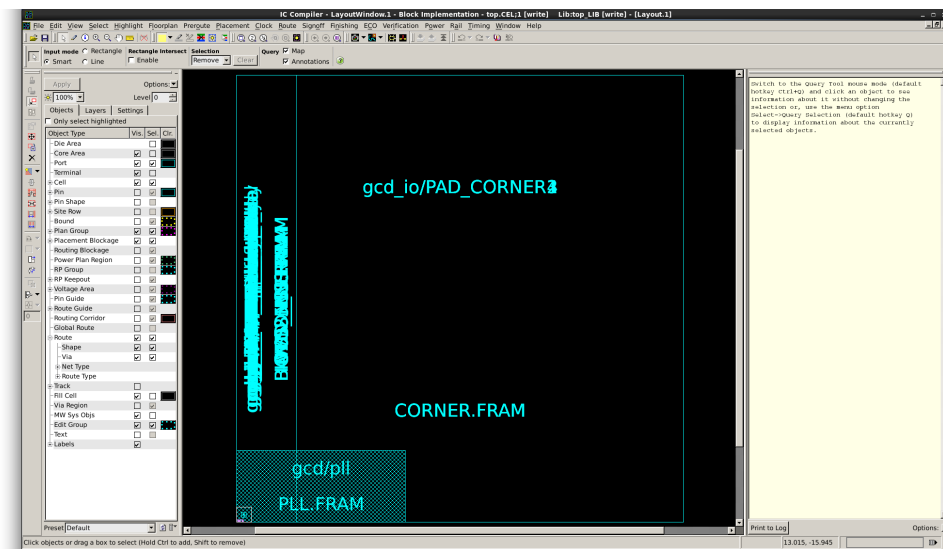


Figure 1: IC Compiler Window

This window shows all of the cells placed on top of each other since we have done nothing to place them. This does get us to a starting point where we issue commands and see how those change the design. The following line is necessary to tell the design how to connect the supply pins, since verilog has no notion of supply connections by default.

```
derive_pg_connection -power_net $MW_POWER_NET -power_pin $MW_POWER_PORT \  
-ground_net $MW_GROUND_NET -ground_pin $MW_GROUND_PORT -create_port top
```

Now it is time to actually floorplan the design. These commands are contained in the `floorplan.tcl` file if copying and pasting from the lab is troublesome. This file can be found in the `floorplan` folder from within `icc-par` at the top level.

The first thing that needs to happen is to set the physical constraints on the pads. You can do this by running the following command:

```
source -echo pads.tcl
```

This runs through all of the commands in the pads.tcl file. Below are the first two lines from that file:

```
set_pad_physical_constraints -pad_name gcd_io/PAD_CORNER1 -side 1
set_pad_physical_constraints -pad_name gcd_io/PAD_pll_ref_clk -side 1 -order 1
```

The `set_pad_physical_constraints` command sets the physical constraints for pad instances. The side number tells the command which side of the chip to place things on. For a rectangular shaped chip, side 1 is the left side, 2 is the top, 3 is the right, and 4 is the bottom. These numbers are in fact only valid for rectangular floorplans. The order flag as seen in the second line sets up an ordering constraint for ordering of pads on a particular side. The default is 0, meaning unordered so the `PAD_CORNER1` cell will be placed last in the ordering. For the left and top sides this ordering is clockwise, and for the right and bottom sides this is counterclockwise. Therefore, the `PAD_pll_ref_clk` cell will be placed in the bottom left (first in priority for the left side clockwise) and the `PAD_CORNER1` cell will be placed in the top left (last in priority for the left side clockwise).

Next we need to run the following commands:

```
derive_pg_connection -power_net VDD -power_pin VDD -create_port top
derive_pg_connection -ground_net VSS -ground_pin VSS -create_port top

set_preferred_routing_direction -layers {M1 M3 M5 M7 M9 MRDL} -direction vertical
set_preferred_routing_direction -layers {M2 M4 M6 M8 MRDL} -direction horizontal
```

The first two commands derive the power and ground connections for the cells in the design. You may see these commands throughout the script multiple times, this is because anytime the tool adds a new cell it may need to update its notion of supply connections for those cells. Please note that this will not actually route the power and ground nets, but just updates the internal notion of what pins need to be connected to these power and ground nets.

The two lines below that setup the routing for different metal layers. IC Compiler wants to route on a grid where the direction of the metal layers alternates as you go through the stackup. Metal 1 (M1) is the lowest layer in this process, which can connect directly to the standard cells which are below all of the metal. The MRDL layer is the highest layer in this process, although that specific metal layer should not be used for much routing. This is due to the fact that it is the top-most layer, and in most processes the parameters of this layer are not optimal for digital routing.

Now that the constraints are set, it is time to actually create the floorplan. The following command does this:

```
create_floorplan \  
    -core_utilization 0.6 \  
    -flip_first_row \  
    -start_first_row \  
    -start_first_row
```

```
-left_io2core 10 \  
-bottom_io2core 10 \  
-right_io2core 10 \  
-top_io2core 10 \  
-row_core_ratio 1
```

The `create_floorplan` command tells the tool to create the area in which to place all of the standard cells, and all of the wires for routing. The `-core_utilization` flag means that you want the design to be occupied by 60% actual standard cells, with 40% filler cells. The reason that you cannot push this too high is that most designs will not be limited by the area of the standard cells themselves, it will be limited by routing which gets exponentially worse the closer packed together things are. The `-flip_first_row` flag specifies to flip the first row at the bottom of the core area. Standard cell rows are abutted together to share either a common VDD or a common GND with the row below them. The orientation of the next row is flipped relative to the one below it since you want two VDD connections together rather than shorting VDD to GND.

The multiple `io2core` flags specify the extra space in micrometers on specific sides of the core to the io cells. The `-row_core_ratio` flag with the value of 1 tells the core area to only include standard cells, and leave no space for routing tracks.

This should look like the following at this point:

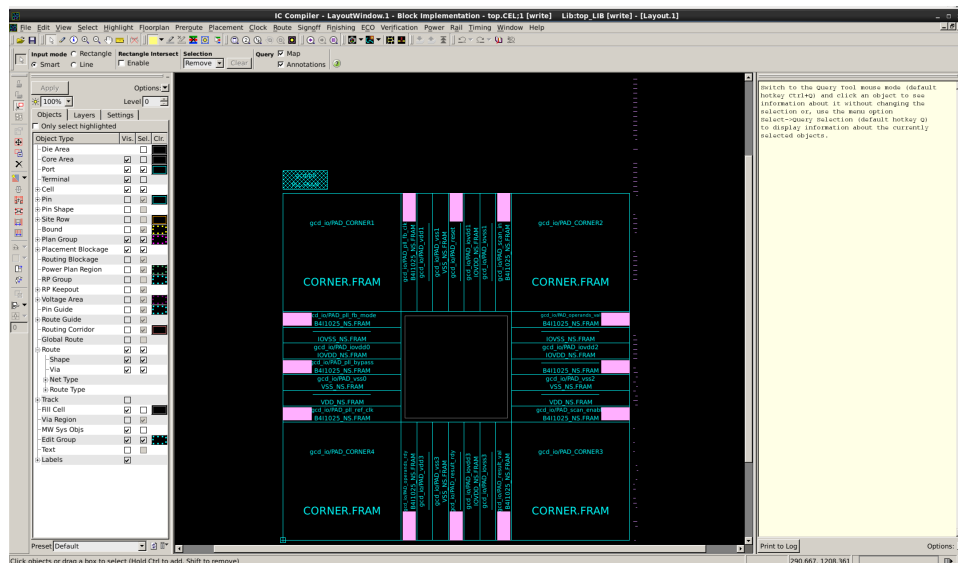


Figure 2: After Creating the Floorplan

Next we need to create the power routing. There are many ways to do this,

```
create_rectilinear_rings -nets {VDD VSS} -width {4 4} -space {1 1} -layers {M8 M9}
```

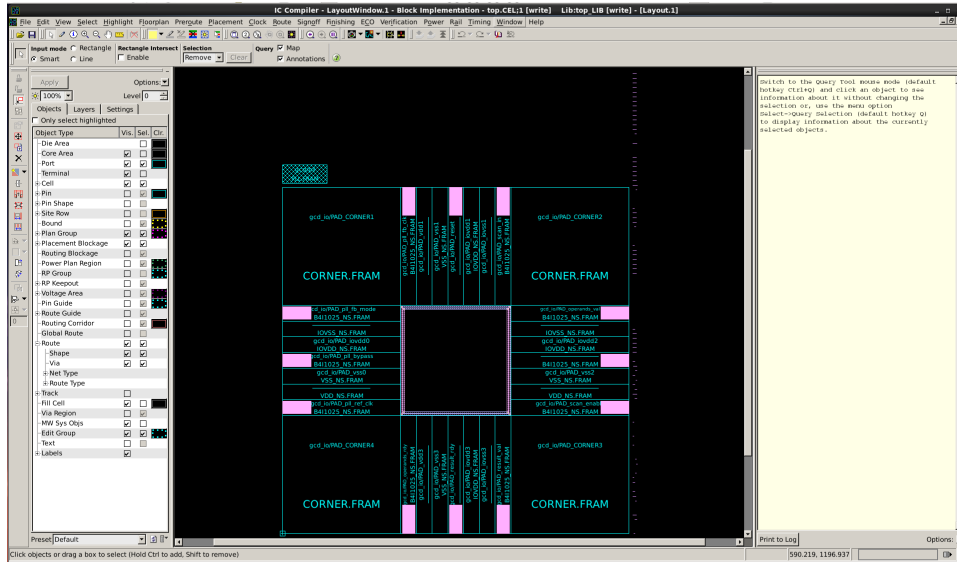


Figure 3: Rectilinear Rings

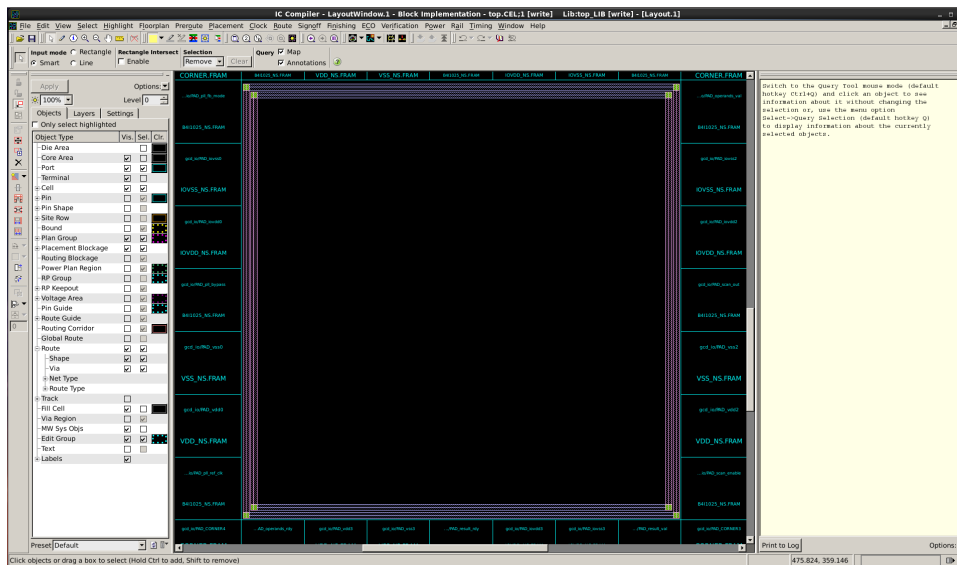


Figure 4: Zoomed in on the rectilinear rings

The following commands will remove the standard cell rows and regenerate them, since for some reason this is necessary in the design kit that we are using.

```
cut_row -all
add_row \
  -within [get_attr [get_core_area] bbox] \
  -direction horizontal \
```

```
-flip_first_row \  
-tile_name unit \  
-bottom_offset 0.2
```

```
create_fp_placement
```

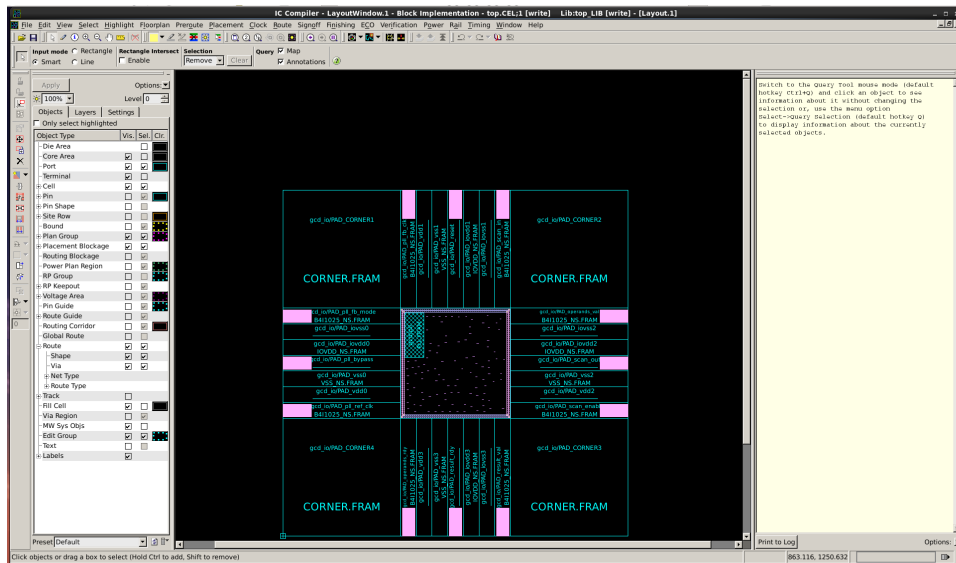


Figure 5: After Creating the Placement

After creating the placement, it is time to create the power straps. The following commands will do that:

```
set_preroute_drc_strategy \  
-min_layer M2 \  
-max_layer MRDL  
  
set_power_plan_strategy core_m3 \  
-nets {VDD VSS} \  
-core \  
-template saed_32nm.tpl:m3_mesh(0.5) \  
-extension {stop: design_boundary}  
  
set_power_plan_strategy core \  
-nets {VDD VSS} \  
-core \  
-template saed_32nm.tpl:m89_mesh(2.0,2.0) \  
-extension {stop: outermost_ring}  
  
compile_power_plan
```



If we look at the template we see the following code:

```
template: m3_mesh(w1) {
  layer : M3 {
    direction : vertical
    width : @w1
    spacing : 1
    pitch : 6
    offset :
  }
}

template: m89_mesh(w1, w2) {
  layer : M8 {
    direction : horizontal
    width : @w1
    pitch : 6
    spacing : 1
    offset :
  }
  layer : M9 {
    direction : vertical
    width : @w2
    spacing : 1
    pitch : 6
    offset :
  }
}
```

This code for the metal 3 mesh is being passed a width of 0.5 ( $\mu m$ ), and the mesh of metal 8 and 9 is being passed a width of 2.0 ( $\mu m$ ). By using this template structure you can easily create a mesh network for the power without a lot of extra commands. Compiling the power plan will translate these constraints into actual pieces of metal. Keep these commands in mind for later, there will be a question on how to modify the power network later in the lab.

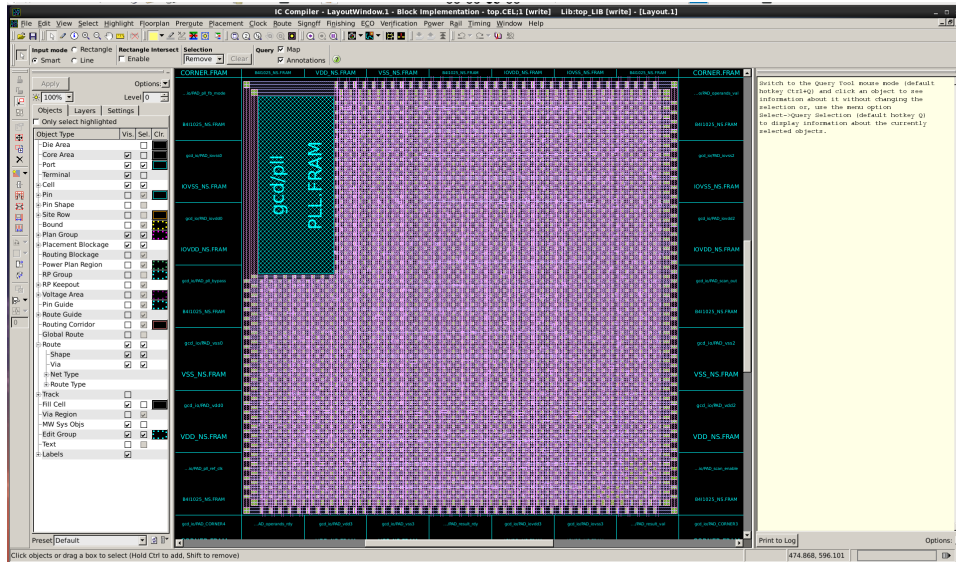


Figure 6: After compiling power plan

After we put in the power straps, we insert filler cells to connect up all of the supply rails. These filler cells are the same size as the standard cells, but do not contain actual transistors. They are there to ensure that the power connections are made and that density of the layers involved in the transistor are maintained.

```
insert_stdcell_filler \
  -cell_without_metal "SHFILL128_RVT SHFILL64_RVT SHFILL3_RVT SHFILL2_RVT SHFILL1_RVT" \
  -connect_to_power {VDD} \
  -connect_to_ground {VSS}
```

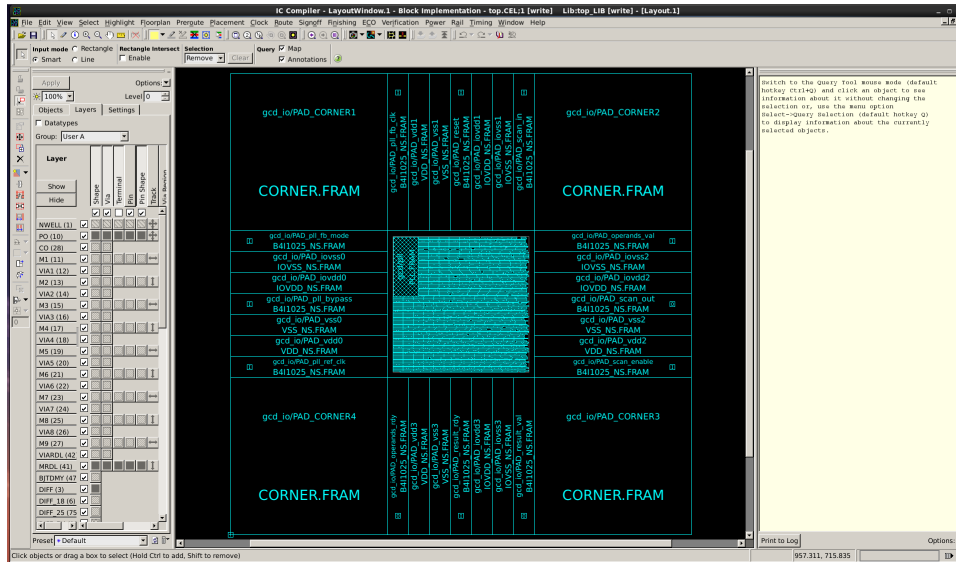


Figure 7: After inserting Filler Cells

Inserting the standard cells does not directly connect the supply rails in metal 1 to the grid that we created earlier with the power plan. To do that we need the following commands:

```
set_preroute_drc_strategy \
  -min_layer M2 \
  -max_layer M5

# preroute standard cell rails
preroute_standard_cells -connect horizontal \
  -nets {VDD VSS} \
  -port_filter_mode off \
  -route_pins_on_layer M1 \
  -cell_master_filter_mode off \
  -cell_instance_filter_mode off \
  -voltage_area_filter_mode off
```

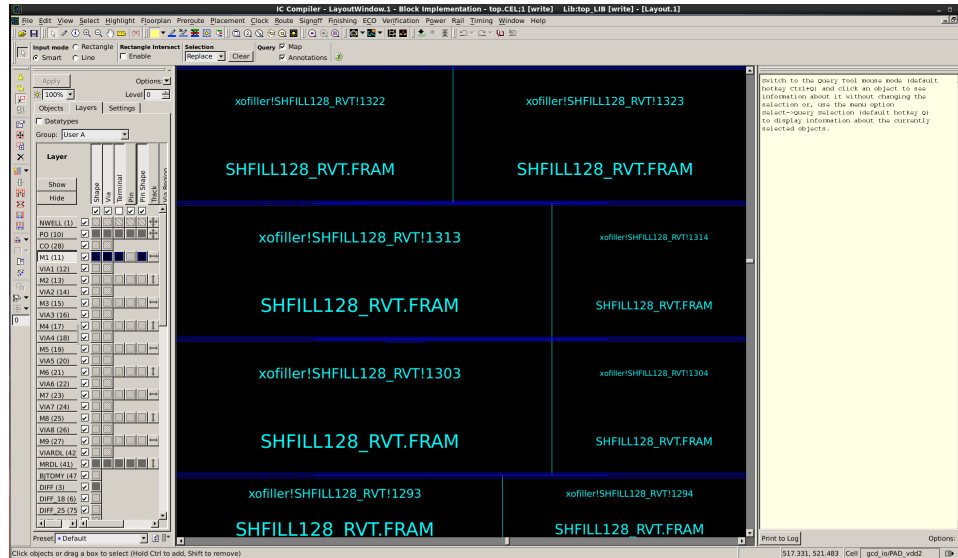


Figure 8: After prerouting the standard cells

We then remove the filler cells so that later steps can insert actual cells if necessary.

```
# verify connectivity of power/ground nets
verify_pg_nets
# get rid of filler cells
remove_stdcell_filler -stdcell
```

```
# verify connectivity of power/ground nets
verify_pg_nets
```

Power/ground connectivity verification will give you several errors, and that is due to the fact that the IO cells are not currently connected to the synthesized power rails. Let's ignore this for now. Now you should have a design that is complete, and looks like the following:

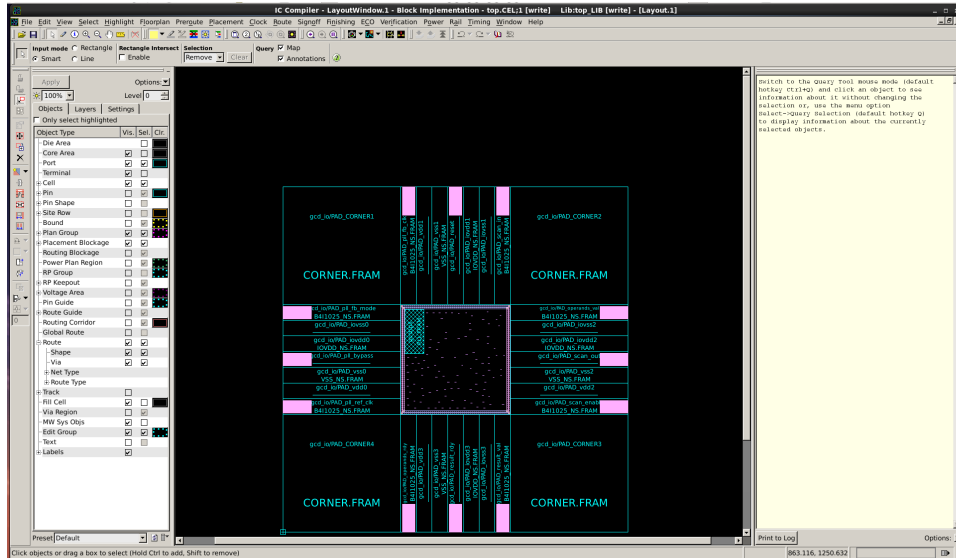


Figure 9: At the end

Alternatively, you could run the following:

```
source -echo floorplan.tcl
```

which will run all the commands in the file.

## Automated Design Flow

As in the previous lab, we have included a fully automated design flow to run through all of the interactive commands. To do this, you need to edit the file `icc_scripts/init_design.icc.tcl` to comment out line 166 by placing a `#` before it. After you do that you can automate the entire design flow by simply typing:

```
make
```

This runs through a few more steps after running through the placement and power routing which will be covered in detail in the next lab, but are necessary to have a fully completed design. One key part that we will cover in this lab is how the constraints are setup for the multiple clock domains. Open up the file `setup/constraints.tcl`. You should see the following lines:

```
# create clock
create_clock -name fast_clk -period ${CLOCK_PERIOD} [get_pins gcd/p11/CLK_4X]
create_clock -name slow_clk -period 20 [get_pins gcd_io/PAD_p11_ref_clk/DOUT]
set_clock_uncertainty ${CLOCK_UNCERTAINTY} [get_clocks fast_clk]
```

```
set_clock_groups -asynchronous \
  -group { fast_clk } \
  -group { slow_clk } \
```

The first two lines create the two different clocks, named `fast_clk` and `slow_clk`. These are both examples of why you do not want to ungroup the hierarchy inside of Design Compiler, since you still want access to the pins of hierarchical submodules at this point in the design. The next line sets up clock uncertainty, which tells the tools some notion of how much uncertainty there is in the clock edges, due to noise and other effects such as variations in buffers driving the clock port. The last command shown here is the most important in this particular design, since it tells the tools that the two clock domains are in fact asynchronous. This will make the tools ignore timing paths that start from one clock domain and end in the other, and vice versa.

To load up the result of this you need to run the following commands:

```
cd current-icc
./start_gui
```

Now we will be looking at some of the reporting/highlighting features that are available within the gui that can give us more information about the power network.

To generate the information for these reports, simply navigate to Preroute — Analyze Power Network.

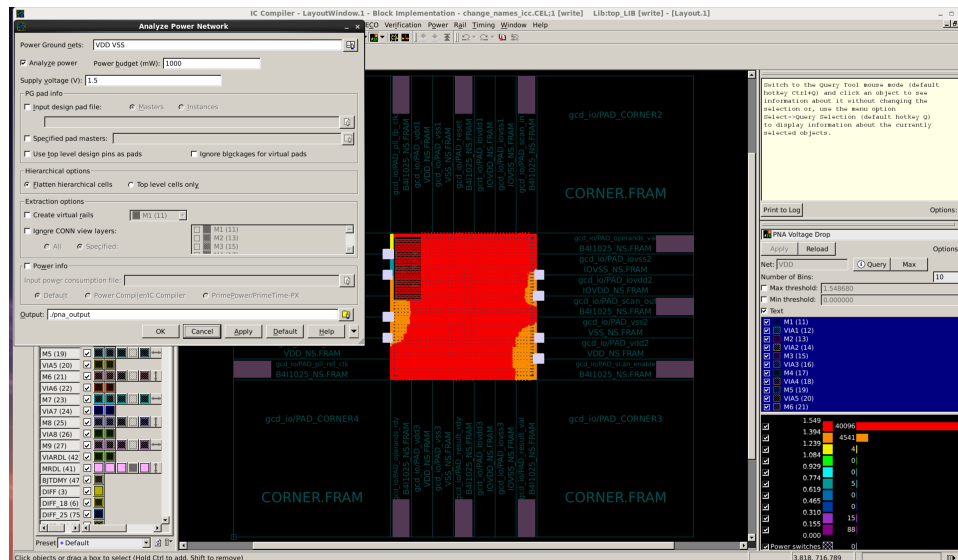


Figure 10: Analyze Power Network

Now that the power network has been analyzed, you can look at some of the interesting features in the gui. Start with the “Power Network Voltage Drop Map” from the same Preroute menu.

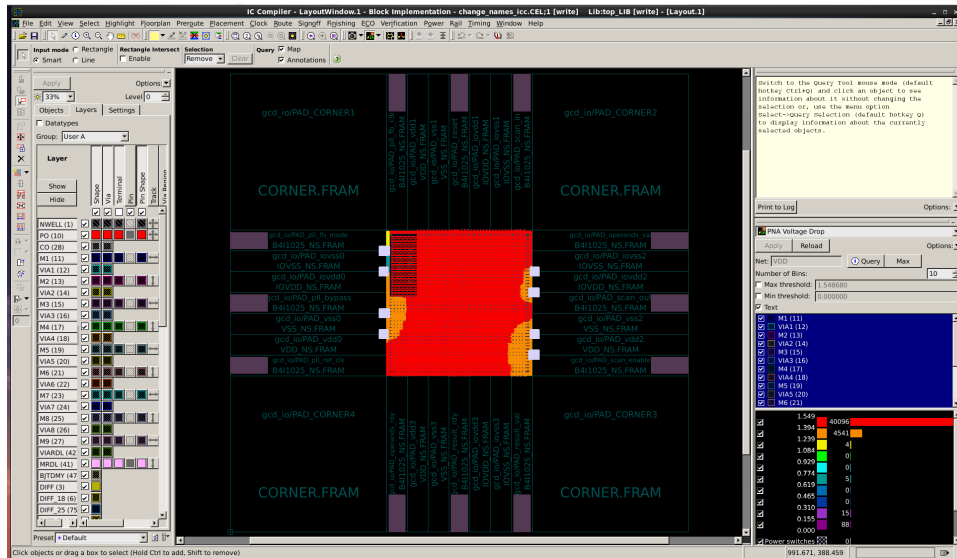


Figure 11: Power Network Voltage Drop Map

Another interesting report to look at is the “PNA Resistance” found in the same menu.

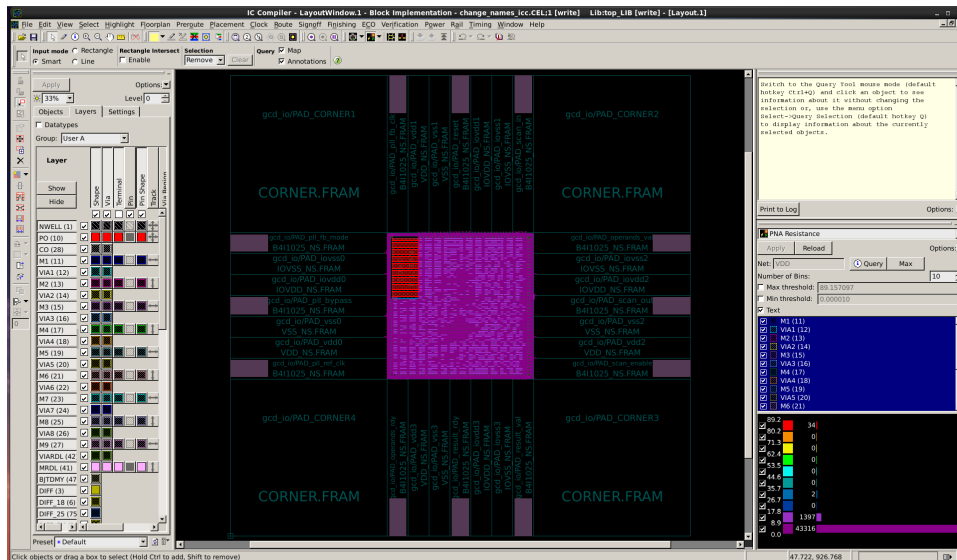


Figure 12: PNA Resistance

There are other reports related to placement as well. Go to Placement and click on the “Color By Hierarchy” report.

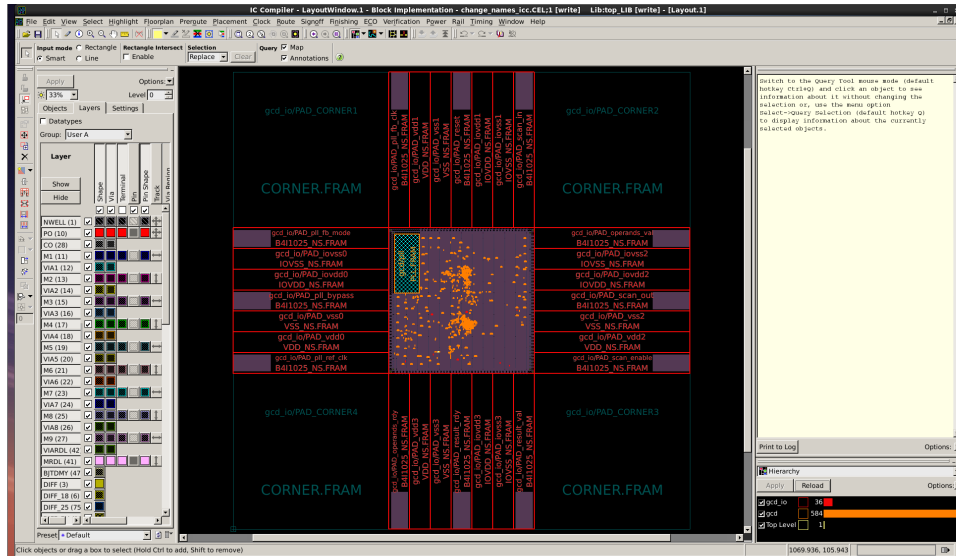


Figure 13: Color By Hierarchy

## Changing the Power Routing

In the previous section we discussed how creating the power plan worked. Now we are going to use that information to regenerate the power plan and the voltage drop map. Be sure to run `make clean` before trying to rerun the design in this section and the next.

### Question 1: Power Routing Changes

Change the power grid to use  $4\ \mu m$  wide wires with a pitch of  $10\ \mu m$  instead of the current setup of  $2\ \mu m$  wide wires with a pitch of  $6\ \mu m$

- Report the changes you need to make to change the power grid this way
- Re-run the design through the tools and generate the PNA Voltage Drop Map. Does the new design look better?
- What would be another potential benefit of routing the power in this manner?

## Changing the Floorplan

### Question 2: Floorplan changes

Now that you understand how to use the tools, change the floorplan so that on the bottom and top sides there are only 4 pads (plus the corners) instead of the current 7, placing the rest of the pads on the left and right sides.

Push the new design through the tools, and report its critical path, cell area, and maximum operating frequency from the reports. Also report the changes you had to make in order to change the floorplan.



## Writing Your Coprocessor - assignment

Take a look at the `gcd_coprocessor.v` file in the `src` folder. You will see the following empty verilog module.

```

module gcd_coprocessor #( parameter W = 32 ) (
    input clk,
    input reset,

    input operands_val,
    input [W-1:0] operands_bits_A,
    input [W-1:0] operands_bits_B,
    output operands_rdy,

    output result_val,
    output [W-1:0] result_bits,
    input result_rdy
);

// You should be able to build this with only structural verilog!

// Define wires

// Instantiate gcd_datapath
// Instantiate gcd_control
// Instantiate request FIFO
// Instantiate response FIFO

endmodule

```

First notice the parameter `W`. `W` is the data width of your coprocessor; the input data and output data will all be this bitwidth. Be sure to pass this parameter on to any submodules that may use it! You should implement a coprocessor that can handle 4 outstanding requests at a time. For now, you will use a FIFO (First-In, First-Out) block to store requests (operands) and responses (results).

A FIFO is a sequential logic element which accepts (enqueues) valid data and outputs (dequeues) it in the same order when the next block is ready to accept. This is useful for buffering between the producer of data and its consumer. When the input data is valid (`enq_val`) and the input data is ready (`enq_rdy`), the input data is enqueued into the FIFO. There are similar signals for the output data. This interface is called a "decoupled" interface, and if implemented correctly it makes modular design easy (although sometimes with performance penalties).

This FIFO is implemented with a 2-dimensional array of data called `buffer`. There are two pointers: a read pointer `rptr` and a write pointer `wptr`. When data is enqueued, the write pointer is incremented. When data is dequeued, the read pointer is incremented. Because the FIFO depth

is a power of 2, we can leverage the fact that addition rolls over and the FIFO will continue to work. However, once the read and write pointers are the same, we don't know if the FIFO is full or empty. We fix this by writing to the `full` register when they are the same and we just enqueued, and clearing the `full` register otherwise.

A partially written FIFO has been provided for you in `fifo.v`. Using the information above, complete the implementation so that it behaves as expected.

A testbench has been provided for you (`gcd_coprocessor_testbench.v`). You can run the testbench to test your code by typing `make run` in `vcs-sim-rtl-coprocessor`.

### Question 3: Design

- a) Submit your code (`gcd_coprocessor.v` and `fifo.v`) with your lab assignment.