Using VLSI Design Flow Outputs

EE241 Tutorial
Written by Brian Zimmer (2013)
Updated by Bonjern Yang (2018)
Updated by Sean Huang (2019)

1 Overview

In this tutorial, we will start with a fully place-and-routed 4-to-16 decoder created using the Synopsys VLSI design flow, import this design into Cadence Virtuoso, extract the design, and simulate it at the transistor level to verify predicted timing and energy results. The VLSI design flow uses a series of models and abstractions to generate circuits, so to truly understand the VLSI flow, it is instructive to remove these abstractions and compare the results in terms of timing and energy. Figure 1 shows a flowchart showing all of the conceptual steps in the tutorial.

2 Getting Started

2.1 Synopsys VLSI Flow

First, we will need to setup working directories for both the VLSI design flow. Run the commands below to setup the VLSI flow. On BWRC machines, replace ∼ee241 with /tools/designs/ee241 throughout this tutorial.

```
% source ~ee241/tutorials/ee241.bashrc
% cd /scratch/userA
% git clone ~ee241/tutorials/decoder_analysis
% cd decoder_analysis
% LABROOT=$PWD
```

This repository already contains a working 4-to-16 decoder, but you need to build the design yourself again. The following commands will run the entire flow to generate a place-and-routed design.

```
% cd $LABROOT/build-rvt
% make pt-pwr
```

Remember if you need to rerun any step in the process and the makefile claims that everything is up-to-date, you can run:

```
% make clean
```

2.2 Cadence Virtuoso

Next, run the commands below to setup a working directory for Cadence Virtuoso. Unlike the VLSI flow directory which will need to be copied for every different design, this directory only needs to be setup once and can be used for all of your projects.

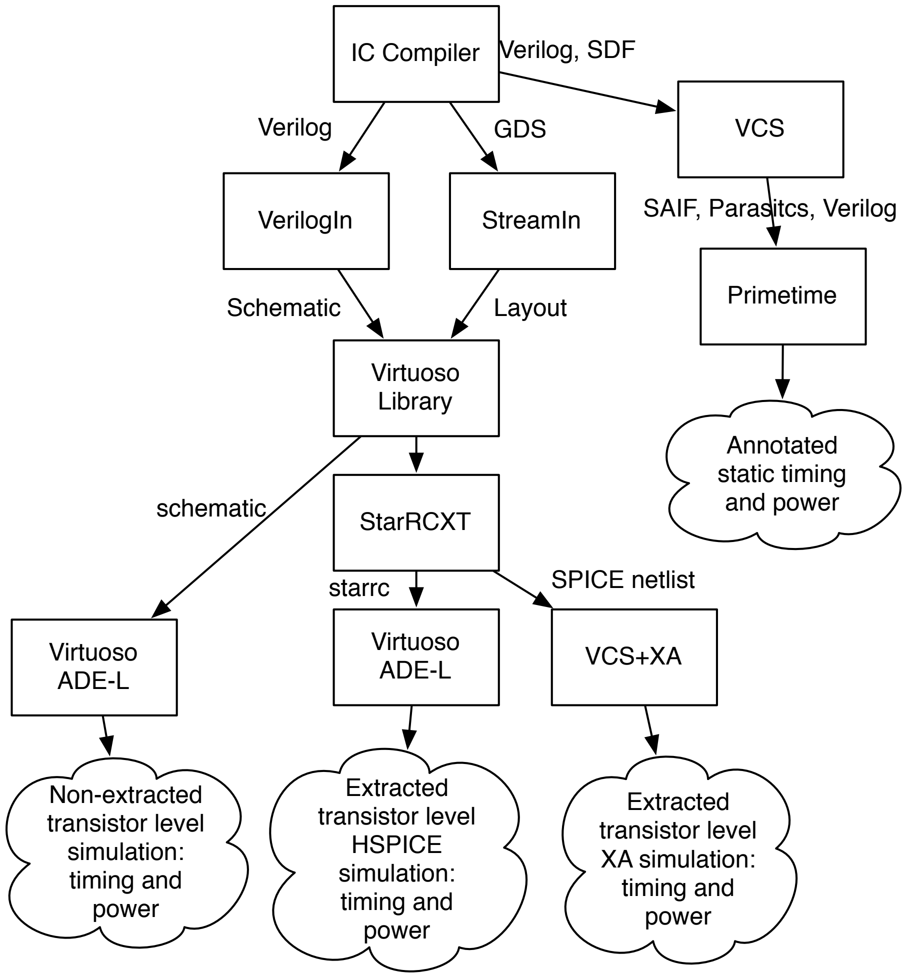

IC Compiler
Verilog, SDF
VCS
Verilog
GDS
VerilogIn
StreamIn
SAIF, Parasitcs, Verilog
Primetime
Schematic
Layout
Virtuoso Library
Annotated static timing and power
schematic
StarRCXT
starrc
SPICE netlist
Virtuoso ADE-L
Virtuoso ADE-L
VCS+XA
Non-extracted transistor level simulation: timing and power
Extracted transistor level HSPICE simulation: timing and power
Extracted transistor level XA simulation: timing and power

Figure 1: EE241 Toolflow for Lab 2

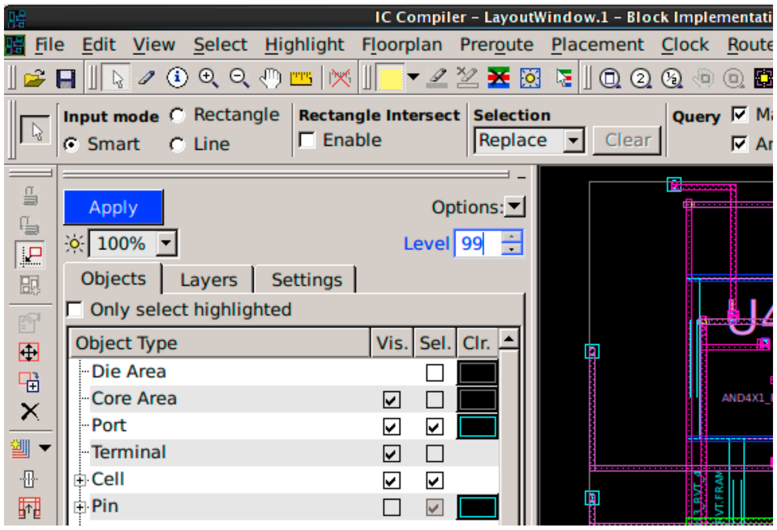
```
% cd /scratch/userA
% git clone ~ee241/ee241_virtuoso
% cd ee241_virtuoso
% VIRTUOSOLABROOT=$PWD
```

3 Translating layout

Open up the design in IC Compiler.

```
% cd $LABROOT/build-rvt/icc-par/current-icc
% ./start_gui
```

Notice that all can be seen is the wiring–the standard cells are all hidden. To expose the design, on the left panel increase the level to "99" and click "Apply", as shown in Figure 2.



Input mode: Smart [✓] / Rectangle / Line
Selection: Replace
100%
Level 99

Object Type	Vis.	Sel.	Clr.
Die Area		☐	■
Core Area	☑	☐	■
Port	☑	☑	■
Terminal	☑	☐	
Cell	☑	☑	
Pin	☐	☑	■

Figure 2: Exposing additional levels.

You can now see the metalization inside each standard cell, as shown in Figure 3 (use "z" and the mouse to the zoom). But notice you still cannot see any devices–there are wells and poly but no active area. This will also be true of an IP blocks you place, such as memories. These views are only "black-box" views and only contain enough information to perform routing. To tape-out this design or perform transistor level extraction and simulation, we need all of the layers.

While IC Compiler is open, there are a few nice tools to use to better understand the output. For example, to highlight a path, go to Timing — New Timing Analysis Window. You can click "Apply" to see the worst paths, or set a specific path by setting the "From:" pin and "To:" pin. The TimingWindow will show the actual arrival time vs. the required arrival time, and report the difference as the "slack." Negative slack will mean there is a violated setup time. Click on a path, and click "Inspector" to see the path as part of the schematic. Then, to highlight the path on the layout, click on Highlight — Inspected Path (or Selected Path).

Now, open Cadence Virtuoso.

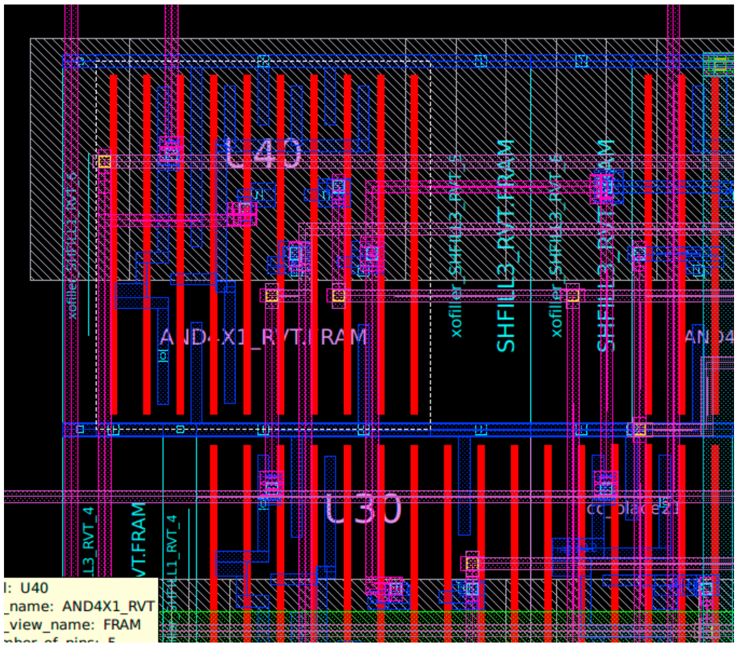


Figure 3: Blackbox view in IC Compiler.

```
% cd $VIRTUOSOLABROOT
% virtuoso &
```

Open the library saed32nm_rvt, then open the "layout" view of AND2X1_RVT. The layout is shown in Figure 4. Now press the ∼ key to view only the frontend layers. The views available inside Virtuoso include all of the needed layers and are called "full-views." You can cycle through other layers using the keys 1, 2, 3, etc to show that layer of metal and the via above it. Click "AV" to show all of the layers again.

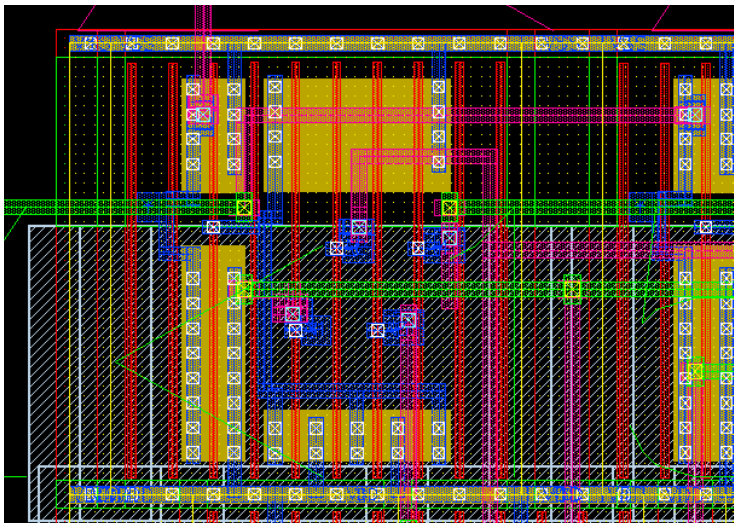


Figure 4: Full view in Virtuoso.

We will use a GDS file format as an intermediate format to move from IC Compiler to Virtuoso. The GDS file will include instantiations of every standard cell, and we will intercept these instantiations and point them to use the "full-views" available in Virtuoso. The GDS file has already been generated inside icc-par/current-icc/results/decoder.gds.

In Virtuoso, make a new library by going to File — New — Library. Give it a name, then select "Attach to an existing technology library" and choose SAED_PDK_32_28. For the rest of this lab document, "test" will refer to this new library.

Then go to File — Import — Stream..., then enter:

- "Stream File": icc-par/current-icc/results/decoder.gds
- "Library": The library you just created
- "Top Level Cell": change_names_icc
- "View": layout

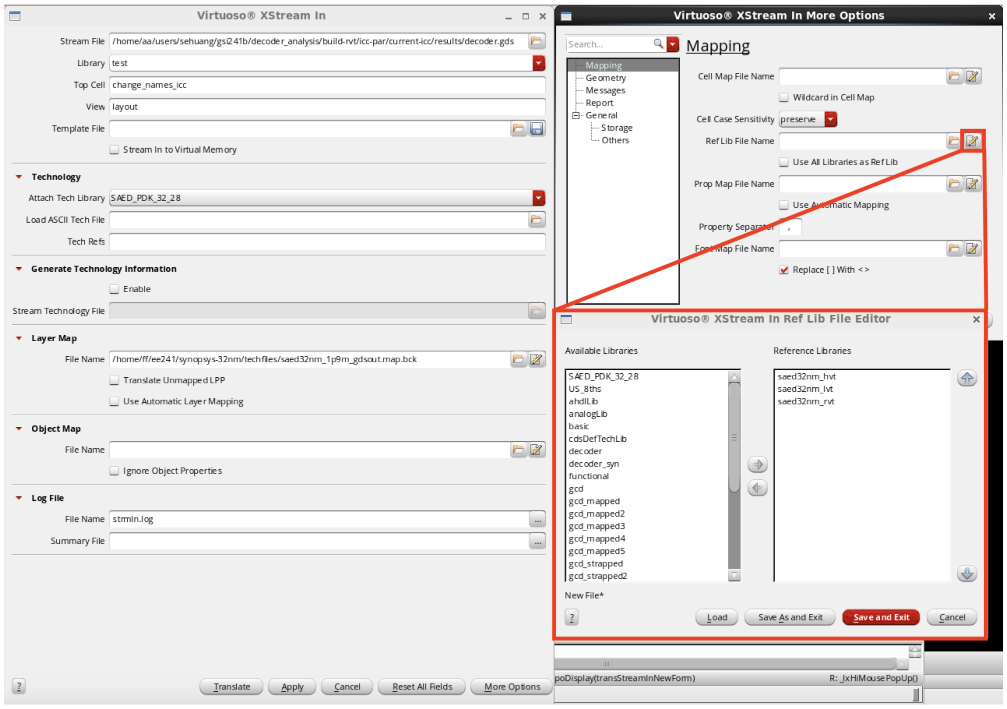
There are a few subfields we will need to fill out as well:

- "Technology"
  - "Attach Tech Library": SAED_PDK_32_28
- "Layer Map"
  - "File Name": ~ee241/synopsys-32nm/techfiles/saed32nm_1p9m_gdsout.map.bck

Select "More Options":

- Ref Lib File Name: Click the far right icon, and select
  - saed32nm_hvt
  - saed32nm_rvt
  - saed32nm_lvt
- Select "Save As and Exit", and give it a name
- Click "OK" to close the dialog box

Click apply (if it prompts you to save your settings, enter a filename and save them), then open the layout view of the design in your new library. Press Shift - + to expose the standard cells so you can see the complete design, as shown in Figure 6.


Virtuoso® XStream In

Stream File: /home/aa/users/sehuang/gsi241b/decoder_analysis/build-rvt/icc-par/current-icc/results/decoder.gds
Library: test
Top Cell: change_names_icc
View: layout
Template File:
Stream In to Virtual Memory

Technology
Attach Tech Library: SAED_PDK_32_28
Load ASCII Tech File:
Tech Refs:

Generate Technology Information
Enable
Stream Technology File:

Layer Map
File Name: /home/ff/ee241/synopsys-32nm/techfiles/saed32nm_1p9m_gdsout.map.bck
Translate Unmapped LPP
Use Automatic Layer Mapping

Object Map
File Name:
Ignore Object Properties

Log File
File Name: strmIn.log
Summary File:

Virtuoso® XStream In More Options

Mapping
Cell Map File Name:
Wildcard in Cell Map
Cell Case Sensitivity: preserve
Ref Lib File Name:
Use All Libraries as Ref Lib
Prop Map File Name:
Use Automatic Mapping
Property Separator: ,
Font Map File Name:
✓ Replace [ ] With < >

Virtuoso® XStream In Ref Lib File Editor

Available Libraries: SAED_PDK_32_28, US_8ths, ahdlLib, analogLib, basic, cdsDefTechLib, decoder, decoder_syn, functional, gcd, gcd_mapped, gcd_mapped2, gcd_mapped3, gcd_mapped4, gcd_mapped5, gcd_strapped, gcd_strapped2

Reference Libraries: saed32nm_hvt, saed32nm_lvt, saed32nm_rvt

Figure 5: GDS Stream In Configuration

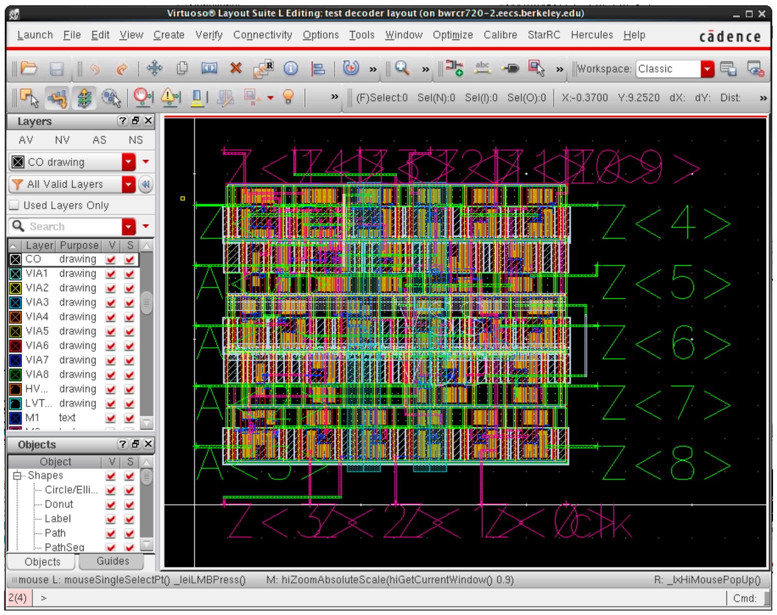

Virtuoso® Layout Suite L Editing: test decoder layout (on bwrcr720-2.eecs.berkeley.edu)
Launch File Edit View Create Verify Connectivity Options Tools Window Optimize Calibre StarRC Hercules Help
cādence
Workspace: Classic
(F)Select:0 Sel(N):0 Sel(I):0 Sel(O):0 X:-0.3700 Y:9.2520 dX: dY: Dist:
Layers
CO drawing
All Valid Layers
Used Layers Only
Objects
Shapes
Guides
Z<4>
Z<5>
Z<6>
Z<7>
Z<8>
mouse L: mouseSingleSelectPt() _leiLMBPress() M: hiZoomAbsoluteScale(hiGetCurrentWindow() 0.9) R: _lxHiMousePopUp()

Figure 6: Entire decoder imported into Virtuoso.

4 Translating schematics

In order to run extraction or LVS on this design, we need to generate schematics for this design.

Note that this design started as RTL level verilog. This is useless as a schematic, as there are no standard cells yet. We could use synthesis output because this includes structural Verilog, yet there is no clock tree and ICC can change cells in order to fix timing, so it will not match the layout. Therefore we need to use the structural Verilog generated by IC Compiler. Open this output in your favorite editor with the following commands

```
% cd $LABROOT/build-rvt/icc-par/current-icc/results
% vim decoder.output.v
```

Notice that each standard cell does not include power and ground. Power and ground was not in the input netlist, and power is treated specially in IC Compiler.

Go back to Virtuoso, and open the schematic and symbol views of the AND2X1_RVT cell in saed32nm_rvt. Notice that this view expects VDD and VSS, because a transistor level netlist needs to know what net is connected to the transistor sources. Therefore our Verilog description needs to include information about power as well. There is another file in the output of IC Compiler that includes this information.

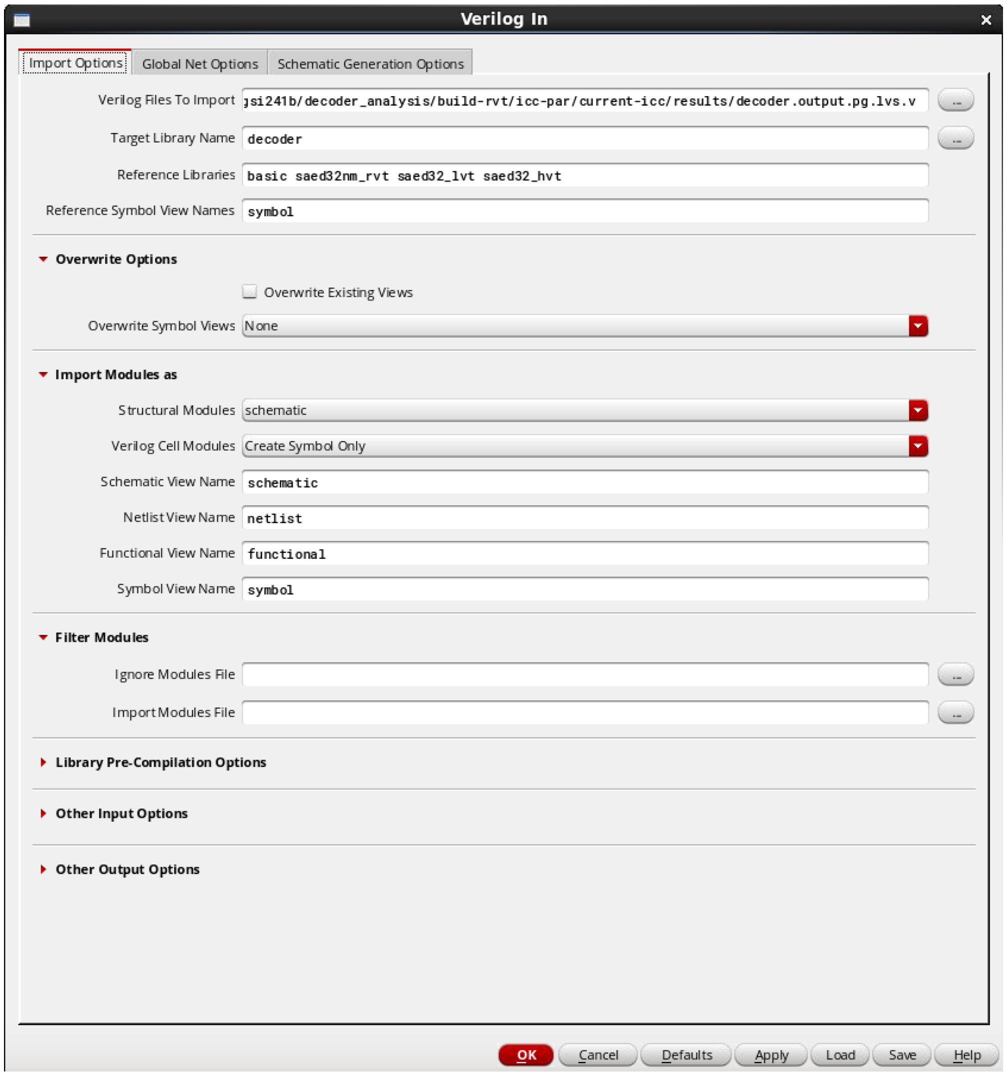
```
% cd $LABROOT/build/icc-par/current-icc/results
% vim decoder.output.pg.lvs.v
```

Notice that this is still a Verilog netlist. Just like the layout from IC Compiler has no information about the actual devices, this Verilog netlist has no information about transistors. However, Virtuoso's schematics are a "full-view" that include the devices, so during import to Virtuoso, we will need to reference the standard cell library as we did for the layout import.

Go to File — Import — Verilog, then and load "verilogin_template". The form should look like Figure 7.

You will need to change:

- Verilog Files to Import: icc-par/current-icc/results/decoder.output.pg.lvs.v
- Target Library Name: your new library name
- Reference Libraries: basic saed32nm_rvt saed32nm_lvt saed32nm_hvt



Verilog Files To Import: `gsi241b/decoder_analysis/build-rvt/icc-par/current-icc/results/decoder.output.pg.lvs.v`
Target Library Name: `decoder`
Reference Libraries: `basic saed32nm_rvt saed32_lvt saed32_hvt`
Reference Symbol View Names: `symbol`

Overwrite Options
Overwrite Existing Views: ☐
Overwrite Symbol Views: None

Import Modules as
Structural Modules: schematic
Verilog Cell Modules: Create Symbol Only
Schematic View Name: `schematic`
Netlist View Name: `netlist`
Functional View Name: `functional`
Symbol View Name: `symbol`

Filter Modules
Ignore Modules File:
Import Modules File:

Figure 7: VerilogIn dialog settings.

Click the "Global Net Options" tab and set "Power Net Name" and "Ground Net Name" to something other than VDD and VSS, e.g. VDD_TOP and VSS_TOP. This prevents the schematic and symbol from using global VDD and VSS pin names.

Click apply, then open the generated schematic. Virtuoso has instantiated all of the standard cells and routed all of the nets.

5 HAMMER, Part 2

Last time, we placed and routed a GCD accelerator, but we stopped short of viewing the design. Now that we know how to stream in a GDS file into Virtuoso, we can take a look at what we created. The HAMMER flow has already streamed out a GDS file of your GCD in `obj/par_rundir/gcdGCDUnit_rtl.gds`. Go ahead and import this into Virtuoso following the same procedure as in the lab.

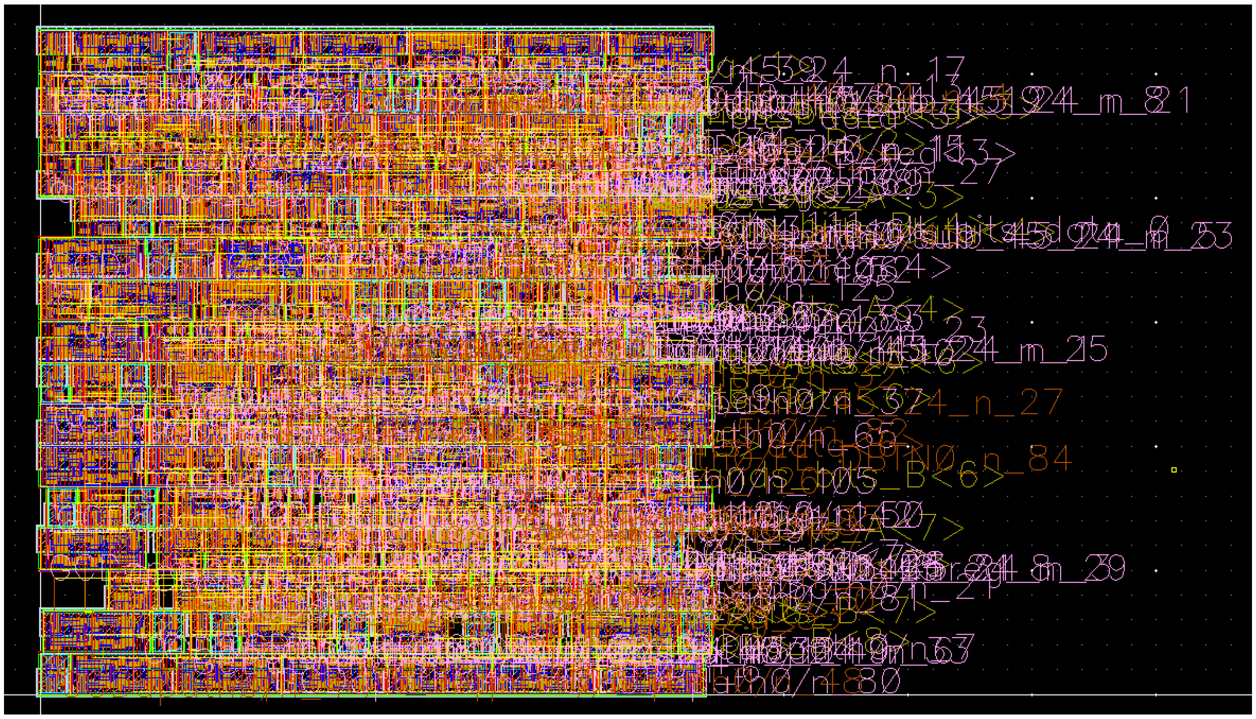


Figure 8: GCD Import

This looks pretty crazy! There are standard cells everywhere and a mess of pin labels, all on the wrong layers. This design also has no power connections or I/O, so this would definitely fail LVS. Let's fix this a bit. First, let's switch to a brand new branch of the lab repo.

```
  % git pull
  % git checkout lab2
```

For a first taste of the exciting new things in store with this iteration of the lab, take a look at `gcdGCDUnit_rtl.yml`.

```
# EE241B Decoder Example on SAED32
# Example to highlight and test placement constraints.

vlsi.core:
  technology: saed32
  node: 33
  technology_path: ["./hammer-saed32-plugin"]
  technology_path_meta: append

vlsi.inputs:
  supplies:
    VDD: "1.05 V"
  power_spec_mode: "manual"
  power_spec_type: "cpf"
  power_spec_contents: "gcdGCDUnit_rtl.cpf"
  power_spec_contents_meta: "transclude"
  #clocks: [
    #{name: "clk", period: "100ns", uncertainty: "5ns"}
  #]
  placement_constraints:
  - path: "gcdGCDUnit_rtl"
    type: "toplevel"
    x: 0
    y: 0
    width: 30
    height: 30
    margins:
      left: 0
      right: 0
      top: 0
      bottom: 0

synthesis.inputs:
  input_files: ["src/lab1_gcd/gcdGCDUnit_rtl.v", "src/lab1_gcd/gcdGCDUnitCtrl.v", "src/lab1_gc
  top_module: "gcdGCDUnit_rtl"

technology.core:
  stackup: "saed32_1p9m_nominal"
  std_cell_rail_layer: "M1"
  tap_cell_rail_reference: "DCAP_HVT"

par:
  inputs.gds_map_mode: manual
  inputs.gds_map_file: "/home/ff/ee241/hammer-stuff/saed32nm_1p9m_inno_gdsout.map"
  power_straps_mode: generate
  generate_power_straps_method: by_tracks
  generate_power_straps_options:
    by_tracks:
      blockage_spacing: 0.0
      strap_layers:
        - M2
        - M3
        - M4
        - M5
        #- M6
        #- M7
        #- M8
        #- M9
      track_width: 4
      track_spacing: 2
      power_utilization: 0.25
      power_utilization_M4: 1.0
      power_utilization_M5: 1.0

synthesis.genus:
  genus_bin: "${cadence.cadence_home}/GENUS${synthesis.genus.version}/bin/genus"
  version: "171"

par.innovus:
  innovus_bin: "${cadence.cadence_home}/INNOVUS${par.innovus.version}/bin/innovus"
  version: "171"
```

There's a lot more going on in here! There are still some familiar fields from last time, but the exciting part comes after `par:`. Here we are describing the power strapping parameters for HAMMER. This file describes the layers to draw power straps, the spacing, the utilization percentage, and also a map file which is important for streaming out the finished design.

The process we used before of running `hammer-vlsi` isn't equipped to handle the additional features from power strapping, so let's update our workspace a bit.

```
  % cd hammer
  % git pull
  % git checkout stackup
  % cd ../hammer-cad-plugins
  % git pull
  % git checkout powerstraps
  % cd ../hammer-saed32-plugins
  % git pull
  % cd ..
```

Now that we have our workspace set up, the new commands for running the HAMMER flow are as follows.

```
  # Synthesis
  % ./inst-vlsi.py syn -p gcdGCDUnit_rtl.yml -p saed32se.yml -o syn-output.json

  # Syn-to-Par
  % ./inst-vlsi.py syn_to_par -p gcdGCDUnit_rtl.yml -p saed32se.yml \
      -p syn-output.json -o par-run.json

  # Place and Route
  % ./inst-vlsi.py par -p par-run.json
```

The reason we are using a different command (a Python file even!) is because we need to define custom hooks to perform additional actions not defined in the default configuration for HAMMER.

HAMMER is extremely flexible as every user is potentially a developer for the framework. If a feature does not exist in HAMMER, any user can write their own custom hook to do it. That being said, let's run the flow through HAMMER with the new commands and see what we get.

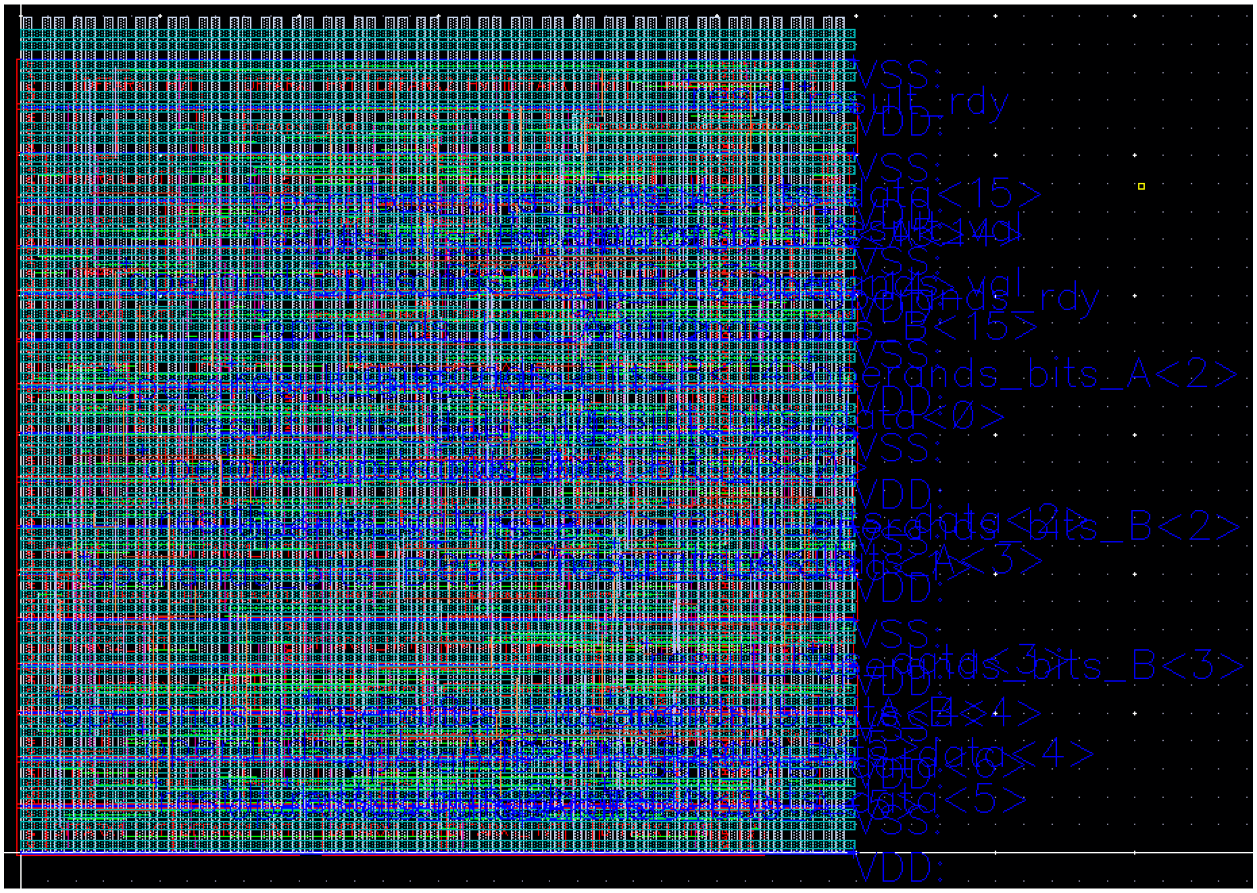


Figure 9: GCD Import With Power Straps

Now we have our power nets connected! Our HAMMERed GCD is looking closer and closer to the result from the Synopsys flow. Try setting up the flow for the decoder example from this past lab, and see the power strapping happen like magic!

6 Conclusion

In this tutorial, you investigated the accuracy of IC Compiler and Primetime reports. In order to run a fully-extracted transistor-level simulation, you need to import both the GDS and Verilog netlist generated by IC Compiler into a Virtuoso schematic and layout. Using HAMMER, you then added power straps to the flow, and we're getting closer to having a DRC/LVS clean design.