Using VLSI Design Flow Outputs, Part 2

EE241B Tutorial
Written by Brian Zimmer (2013)
Updated by Bonjern Yang (2018)
Updated by Sean Huang (2019)

1 Overview

In the last tutorial, we imported a design into Cadence Virtuoso. In this tutorial, we will extract the RC parasitics from the place-and-routed design and compare to the timing reports given by the digital flow. There are a lot of abstractions in the VLSI design flow, so it is interesting to compare against the extracted simulation.

2 Extraction

To run parasitic extraction in Virtuoso, first change the name of the layout cell view to "decoder" (or copy to a cell view called "decoder"). Open the layout view and go to StarRC — Parasitic Generation Cockpit. The setting should be automatically loaded for you as shown in Figure **??**, but you will need to change a few things: Change the "LVS FLOW" to "ICV". From the Run Cockpit tab, uncheck "LVS Clean". Click on the "Output Parasitics" tab, and check the box next to "Ports Annotation", then enter in your library and cell name. If you do not do this, you will not be able to netlist the design. Some other options are useful to understand.

- Change "LVS FLOW" to "ICV"
- From the Run Cockpit tab:
  - Uncheck "LVS Clean"
- From the Device Extraction tab:
  - ICV Runset: "/home/ff/ee241/hammer-stuff/SAED_PDK32nm/icv/drc/drc/saed32nm_1p9m_lvs_rules
- From the Extract Parasitics tab:
  - EXTRACTION: RC *(Extracts both resistance and capacitance)*
  - COUPLE TO GROUND: NO *(Includes coupling between nets)*
  - Additional Options for setting thresholds to limit the number of extracted devices
- Output Parasitics tab:
  - Check "Ports Annotation"

Now click "Apply" and wait a minute while extraction runs. When it finishes, go to your new library and open the "starrc" view of the decoder cell. If you zoom in, you can see the annotated parasitics.

3 Simulation in Virtuoso

Go to File — New — Cell View, call it decoder_tb, and make the view and type "schematic". Press 'i', then insert the symbol view of your newly imported decoder. Using the "vdc" and "vpwl" components in analogLib and the "vdd", "gnd", "noConn" components in basic, hook up VDD, VSS, and the inputs as shown in Figure 1. Set the "DC voltage" of the vdc instance to 1.05. Note that you need to instantiate an array of noConn by pressing q while the noConn block is selected and changing the instance name, as shown in Figure 2. Set the VPWL as follows:

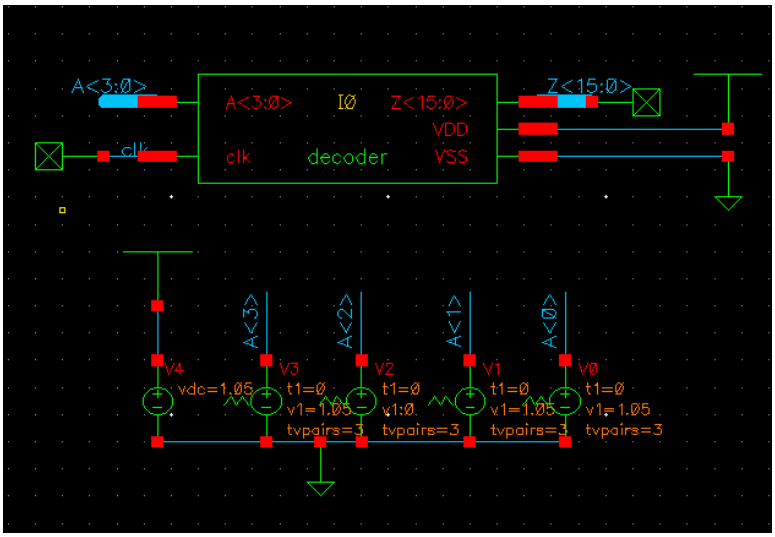


Figure 1: Testbench to measure a timing path.

Figure 2: Creating an array of noConn instances.

- A[3]: 3 pairs of points, T1: 0, V1: 1.05, T2: 500p, V2: 1.05, T3: 510p, V3: 1.05
- A[2]: 3 pairs of points, T1: 0, V1: 0, T2: 500p, V2: 0, T3: 510p, V3: 1.05
- A[1]: 3 pairs of points, T1: 0, V1: 1.05, T2: 500p, V2: 1.05, T3: 510p, V3: 1.05

- A[0]: 3 pairs of points, T1: 0, V1: 1.05, T2: 500p, V2: 1.05, T3: 510p, V3: 1.05

Go to File — Check and Save, and there should be no errors or warnings. Next, we need to make a "config" view that will let us tell the simulator whether to simulate with the schematic view or the starrc view. Go to File — New — Cellview, and enter

- Cell: decoder_tb (the testbench you just created)
- View: config

Then click enter the settings shown in Figure 3. When the next window opens, go to File — Save. Now in the schematic window, go to Launch — ADE L. Go to Setup — Simulator, and make sure HSPICE is chosen. Click on Setup — Design, and choose View: config, and press ok. Next, go to Analysis — Choose, and set the stop time to 1n, and click ok. Then go to Outputs — To be Plotted — Select on Schematic, and click on both the A and Z bus (and select all of their signals), then press "Esc" when finished. Last, go to Simulation — Netlist and Run, and you should see a result like Figure 4.

To measure the path we are measuring (A[2] rising to Z[15] rising), mouse over the first edge at the 50% point and press "a", then go to the second edge and press "b".

Your result is only measuring the devices. Now, we would like to simulate this same path, but also annotate the parasitics we calculated during extraction.

Open the config view of decoder_tb, (click "yes" for Configuration config), then in the "View to Use" column for the "decoder" instance, enter starrc as shown in Figure 5. Then go to File — Save.

Go back to ADE L and rerun the simulation. The delay should now increase because the parasitics of the wires are now included. Go to File — Save State, then save inside "cellview" so you can open this simulation later. By going to Simulation — Netlist — Display, you can make sure the parasitics are being included within the spice deck.

Note that IC Compiler and Primetime perform timing on all paths. However, many paths in the decoder are false paths. For example, the path A[3] falling to Z[2] falling is a false path. Because it is a one hot decoder, a high A[3] means that only Z[15:8] could be high, so there is no possible way for Z[2] to fall.

3.1 Energy measurement

At the beginning of the tutorial when you typed "make pt-pwr", you should simulated the post-place-and-route netlist with annotated parasitics and performed power analysis on your design. Go back to the simulation to see what activity the testbench generated.

```
% cd $LABROOT/build-rvt/vcs-sim-gl-par
% make convert
% dve -vpd vcdplus.vpd &
```

Then plot both the A and Z signals. You should see the decoder is driven with A counting from 0000 to 1111 as shown in Figure 6. By running "make convert", the switching activity of every

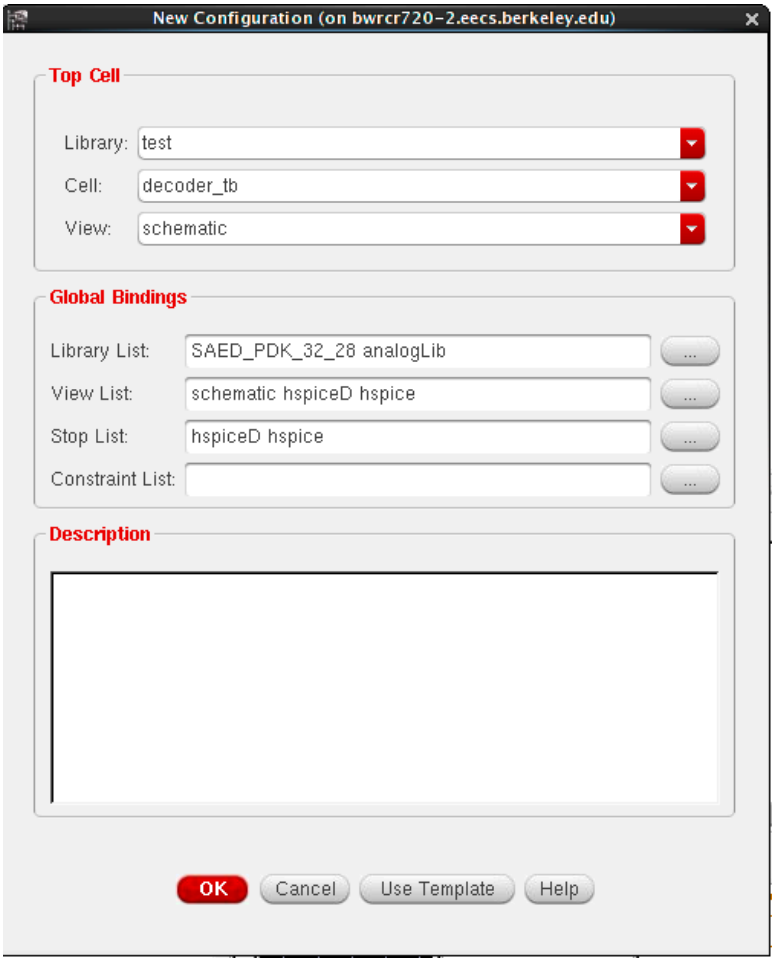


Figure 3: Settings for the new config view.

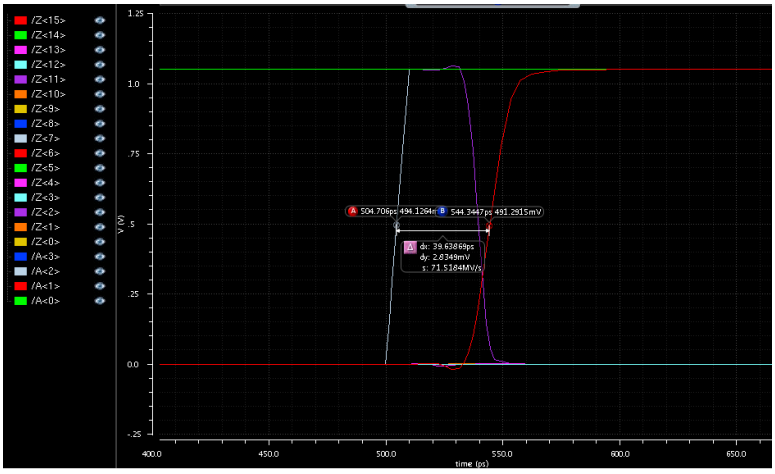
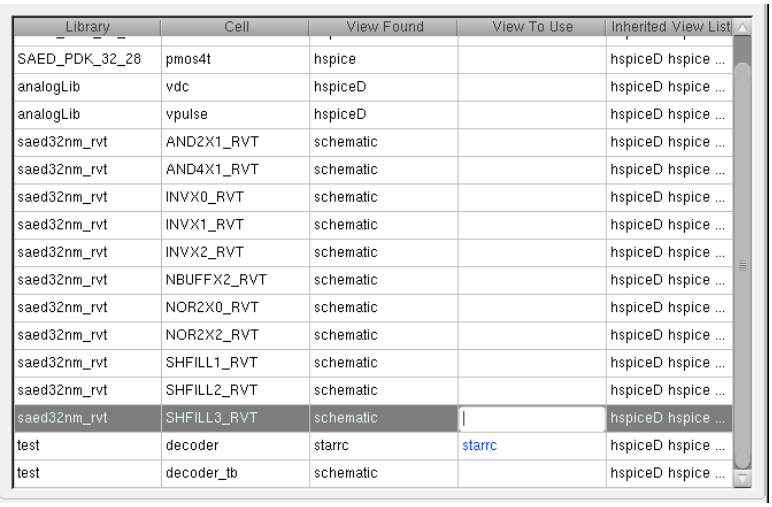


Figure 4: A[2] rise to Z[15] rising timing path for schematic simulation.



Library	Cell	View Found	View To Use	Inherited View List
SAED_PDK_32_28	pmos4t	hspice		hspiceD hspice ...
analogLib	vdc	hspiceD		hspiceD hspice ...
analogLib	vpulse	hspiceD		hspiceD hspice ...
saed32nm_rvt	AND2X1_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	AND4X1_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	INVX0_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	INVX1_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	INVX2_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	NBUFFX2_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	NOR2X0_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	NOR2X2_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	SHFILL1_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	SHFILL2_RVT	schematic		hspiceD hspice ...
saed32nm_rvt	SHFILL3_RVT	schematic		hspiceD hspice ...
test	decoder	starrc	starrc	hspiceD hspice ...
test	decoder_tb	schematic		hspiceD hspice ...

Figure 5: Change the config view to use the extracted netlist.

node is annotated in a SAIF file. Then Primetime (in pt-pwr) combines this with the icc-par netlist and parasitics file (SBPF) to accurately measure energy consumption. For example, look at the generated SAIF file (switching activity).

```
% cd $LABROOT/build-rvt/vcs-sim-gl-par
% make convert
% cat vcdplus.saif
(INSTANCE U34
      (NET
        (A1
          (T0 1600) (T1 1400) (TX 0)
          (TC 1) (IG 0)
        )
        (A2
          (T0 1400) (T1 1590) (TX 10)
          (TC 14) (IG 0)
  ....
```

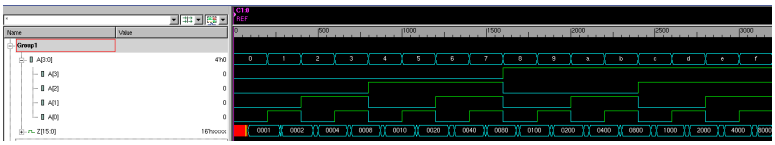


Figure 6: Test vector generated by Verilog testbench.

TC counts the number of toggles, and is used to calculate dynamic energy. T1, T0, and TX correspond to time at logic 1, 0, and X and are used to calculate leakage energy (leakage will be state dependent).

Now look at the Primetime energy report:

```
% cd $LABROOT/build-rvt/pt-pwr
% make
% cat current-pt/reports/vcdplus.power.avg.max.report

                                        Switch    Int       Leak      Total
Hierarchy                               Power     Power     Power     Power     %
------------------------------------------------------------------------------
decoder                                 3.06e-05  2.68e-05  1.70e-06  5.90e-05  100.0
```

So for the 3.2ns testbench, the average power was 59uW for the decoder.

Now, back in Virtuoso, recreate the same testbench inside decoder_tb as shown in Figure 7 using vpulse with periods of 400ps, 800ps, 1600ps, and 3200ps for A[0], A[1], A[2], and A[3], and pulse time as half of the period. You might want to backup your old testbench. Launch ADE again, and run a transient simulation for 3.2ns. Probe the current through the VDD current source. Then go to Outputs — Setup, click New Expression, and set the Name to be "power", then next to calculator click Open. Click the "it" button, then select the positive port of the VDD voltage source. Then in the function panel, click "Average", then append "1.05" to the expression in the textbox. Without closing the calculator, go back to the Setup Output Options dialog and click "Get Expression". The box will look like Figure 8. Click "Add", then "Ok". Now run the simulation. In the outputs pane, you should see a number for your power. This is the power measured by a spice simulation of the extracted netlist.

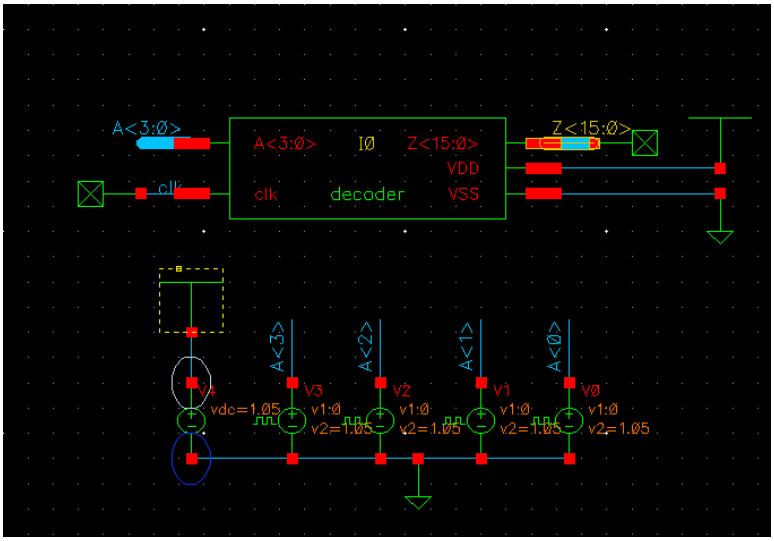


Figure 7: Test bench inside Virtuoso that emulates the Verilog testbench.

4 Mixed-mode simulation

For small designs, full HSPICE simulation in Virtuoso using handcrafted testbenches is easy. However, for larger designs, it is very useful to stimulate the simulation with a Verilog testbench, using

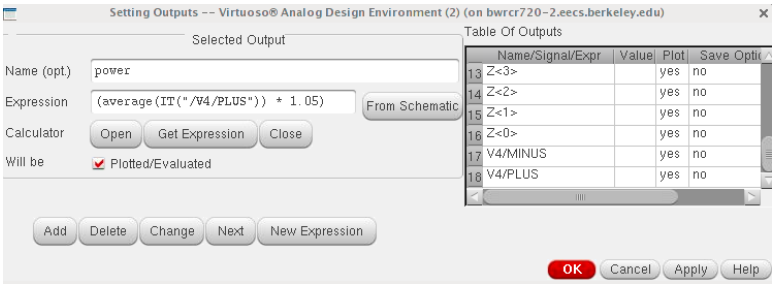


Figure 8: Create a measure statement to calculate average power.

a faster simulator, such as XA. Then critical analog components such as PLLs, SRAMS, and asynchronous interfaces can be simulated at the transistor level while the basic digital logic can be simulated more quickly with VCS.

Another reason to run mixed-mode simulations is to measure the effects of voltage scaling on energy. While Primetime works well for measuring power of digital logic, voltage scaling studies require all of the standard cells to be characterized at different voltages, and these cells are not always available.

In ADE L, go to Simulation — Netlist — Create. A netlist with parasitics should appear. Now to go File — Save As, and place this design in the $LABROOT/build-rvt/vcs-sim-tl-pex directory as decoder.sp. This directory is a new directory that will run a mixed simulation.

Look at the Makefile in this directory.

```
  % cd $LABROOT/build-rvt/vcs-sim-tl-pex
  % cat Makefile

  srcdir = $(basedir)/src
  vsrcs = \
    $(srcdir)/decoder_tb.v

  ...

$(vcs_sim) : Makefile $(vsrcs)
  $(VCS) $(VCS_OPTS) +incdir+$(srcdir) -o $(vcs_sim) \
         +define+CLOCK_PERIOD=$(vcs_clock_period) \
        $(vsrcs)

  ...

  VCS_OPTS = -notice -PP -line +lint=all,noVCDE,noTFIPC,noIWU,noOUDPE \
    +v2k -timescale=1ns/1ps \
    -P ../icc-par/current-icc/access.tab -debug \
    +neg_tchk \
    -ad=vcsAD.init -lca
```

The -ad=vcsAD.init line is the magic line that enables mixed signal simulation. Think of the vcsAD.init file as the config file we used in Virtuoso to choose between schematic only and parasitic simulation. You can replace $(vcs_clock_period) to define a clock period in terms of ns. In our case, replace this with 0.4 in order to compare with the extracted virtuoso simulation. If you do not replace this with a real number, outputs in xa.meas will not be properly computed.

The only Verilog source is the same testbench as before, which instantiates the decoder with:

```
decoder DUT0 (.A(A), .Z(Z), .clk(clk));
```

Now look at the configuration file. This defines the analog to digital interface, tells VCS to use the Synopsys XA simulator (a transistor level simulator like HSPICE but optimized for speed), and to use spice on the "decoder" cell corresponding to the instantiation above. Also, Verilog has bus format for signals (eg. A[3:0]) but SPICE has no buses, so the tool must know how to convert between formats to hook up the ports correctly. xa_options tells the simulator to merge all probed signals with the VPD file so you can see digital and analog signals together in the same tool.

```
% cat vcsAD.init
choose xa -hspice decoder.sp -c xa_options -o xa;
use_spice -cell decoder;
bus_format <%d>;
d2a rf_time=10p rise_time=20p fall_time=20p delay=0 hiv=1.05 lov=0.0 node=*;
a2d loth=0.3v hith=0.7v node=*;
```

Next we need to hookup the supplies within the decoder. While there are cleaner ways to do this, we will do something easy for now. Edit decoder.sp. Delete all lines (except the first line) before the .subckt definition, and all lines after the .ends at the end of the circuit. *Make sure the first line is a comment.* Note that if your power measurment fails, you need to make sure your FROM and TO values are within the simulation time. Then add a path to the library, and place voltage sources as shown below (make sure the voltage sources are within the subcircuit definition):

```
% cat decoder.sp

*  Decoder
.lib /home/ff/ee241/synopsys-32nm/hspice/saed32nm.lib TT

.subckt decoder a<3> a<2> a<1> a<0> vdd vss z<15> z<14> z<13> z<12> z<11>
+ z<10> z<9> z<8> z<7> z<6> z<5> z<4> z<3> z<2> z<1> z<0> clk
Vvdd vdd 0 DC=1.05
Vvss vss 0 DC=0
.probe tran i(Vvdd)
.measure pavg AVG par('1.05*i(Vvdd)') FROM=0 TO=3.2n

    cg1431 u26|mp3|drn 0 c=85.1142e-18
```

Last, run the simulation and see the power measured by XA.

```
% make
```

```
% make run
% cat xa.meas
% dve -vpd vcdplus.vpd
```

5 HAMMER, the Third

So far, we've pushed a GCD accelerator through the basic HAMMER VLSI flow, then applied power straps to the design. However, our GCD still doesn't have pins to connect to other blocks, so this isn't a very effective accelerator. To change that, this tutorial will introduce the concept of custom hooks.

5.1 HAMMER Hooks

As you may have noticed, HAMMER is a relatively new tool (in fact, the power strapping API we used in the last lab was still in review when the last lab released). Several functions that a VLSI flow automation tool would be expected to have simply haven't been developed yet, or are too specific for the project to be made into a generic option. For these custom-made features, HAMMER gives you the ability to write custom functions and features and apply them seamlessly into the existing flow.

As part of the power strapping process, the last HAMMER flow actually made use of such a hook to place decoupling capacitor cells as a reference for the power routing. This was the reason why the command changed to use `inst-vlsi.py` as opposed to `hammer-vlsi`, so let's take a look at that file.

```
def place_tap_cells(x: hammer_vlsi.HammerTool) -> bool:
  x.append('''
set WT_INT 25
set_db add_well_taps_cell DCAP_HVT
add_well_taps -cell_interval $WT_INT
  ''')
  return True

def define_pins(x: hammer_vlsi.HammerTool) -> bool:
  x.verbose_append("edit_pin -pin * -spread_type SIDE -side LEFT -layer M5
      ↪ -fixed_pin".format(top=x.top_module))
  return True

class InstDriver(CLIDriver):

  def action_map(self) -> Dict[str, Callable[[hammer_vlsi.HammerDriver, Callable[[str],
      ↪ None]], Optional[dict]]]:
    """Return the mapping of valid actions -> functions for each action of the
        ↪ command-line driver."""
    par_action = self.create_par_action(custom_hooks=[
      hammer_vlsi.HammerTool.make_replacement_hook("place_tap_cells", place_tap_cells)
    ])

    new_dict = dict(super().action_map())
    new_dict.update({
```

```
      "par": par_action
      })
    return new_dict

  def par_action(self, driver: hammer_vlsi.HammerDriver, append_error_func:
      ↪ Callable[[str], None]) -> Optional[dict]:
    if not driver.load_par_tool():
      return None
    success, par_output = driver.run_par(hook_actions=[
      hammer_vlsi.HammerTool.make_replacement_hook("place_tap_cells", place_tap_cells),
      hammer_vlsi.HammerTool.make_post_insertion_hook("power_straps", define_pins)
      ])
    return par_output

if __name__ == '__main__':
    InstDriver().main()
```

The `action_map` function updates the overall steps HAMMER takes. In this case, we haven't redifined the synthesis actions, so we don't update them in this function. However, we have a few changes made to the place and route action. This is defined here with the `create_par_action` and then added to the dictionary of actions. Of particular note here is how the `custom_hooks` option has been defined. Right now the only hook we've added is `place_tap_cells`, but we will quickly be adding new ones. The hook is added to the flow by means of a hook insertion method, which can be `make_pre_insertion_hook`, `make_post_insertion_hook`, and `make_replacement_hook`. These add the hook step before, after, or replaces an existing step in the flow, respectively.

Above the `action_map`, we have the hook function itself. The easiest way to write these hooks is to define the actual Tcl string that is added to the tcl script. We can do fancier things with calculating parameters from the input YAML file, but in this case, we just need to define some tap cells and add them to the design.

Now for the fun part. Change branches to `lab3`.

Look at `inst-vlsi.py` again. Some things have changed a bit! First of all, there are many more hooks now, and all of them have moved to the `par_action` function, where hooks can also be defined. For now, there are no differences between the two, but defining custom hooks in the action map takes precendence. The bigger difference, however, is that there are no more hooks defined in the file anymore! They've migrated to the `inst_hooks.py` file, which is where we will be heading next.

This file is pretty complicated, but the hook in particular that we care about is `inst_define_pins`. This function is defining the pin locations around the border of the chip. I'll go through the syntax for the `edit_pin` command as this may be important for using HAMMER for projects.

```
def inst_define_pins(x: hammer_vlsi.HammerTool) -> bool:
    # x.verbose_append("edit_pin -pin clk -spread_type SIDE -side LEFT -layer M5
        ↪ -fixed_pin".format(top=x.top_module))
    x.append('''
edit_pin -pin clk -side LEFT -layer 3 -fix_overlap 1 -assign 0.0 0.0 -use clock
edit_pin -side LEFT -layer 3 -pin [get_db [get_db ports -if {.name == operands_bits_A*}]
    ↪ .name] -spread_type start -unit track -spread_direction clockwise -spacing 15
```

```
    ↪ -start {0 2}
edit_pin -side TOP -layer 2 -pin [get_db [get_db ports -if {.name == operands_bits_B*}]
    ↪ .name] -spread_type start -unit track -spread_direction clockwise -spacing 15
    ↪ -start {2 35}
edit_pin -side BOTTOM -layer 2 -pin {{operands_rdy} {operands_val} {reset}} -spread_type
    ↪ start -unit track -spread_direction counterclockwise -spacing 10 -start {2 0}
edit_pin -side BOTTOM -layer 2 -pin {{result_rdy} {result_val}} -spread_type start -unit
    ↪ track -spread_direction clockwise -spacing 10 -start {28 0}
edit_pin -side RIGHT -layer 3 -pin [get_db [get_db ports -if {.name ==
    ↪ result_bits_data*}] .name] -spread_type start -unit track -spread_direction
    ↪ clockwise -spacing 15 -start {28 35}
    ''')
    return True
```

`edit_pin`:

- `-pin`
  - This can either be defined specifically by name, or can be specified by another command.
  - An example: `[get_db [get_db ports -if .name == operands_bits_A*] .name]` would select all the pins that begin with `operands_bits_A`. The two `get_db` are necessary to convert the result of the command to a set of strings of pin names.
- `-side`
  - Defines the side of the chip to place the pins
- `-layer`
  - Defines the layer on which to place the pin shapes
- `-spread_type`
  - Defines the spread paradigm. This defines how the pins are spread around the side. For example, `start` uses the first pin in the list as the starting point and proceeds in the direction specified in `spread_direction`. Another valid option is `center`, which uses the first pin as the center of the distribution and spreads the pins around accordingly.
- `-spread_direction`
  - Direction of pin spread, defined as either `clockwise` or `counterclockwise`
- `-unit`
  - Defines unit of spacing. This can either be in microns or track widths, default to microns if not specified.
- `-spacing`
  - Number of spacing units between pins.
- `-start`
  - Starting coordinate for the first pin. If there are obstructions, Innovus will place this at the closest legal placement location if possible.

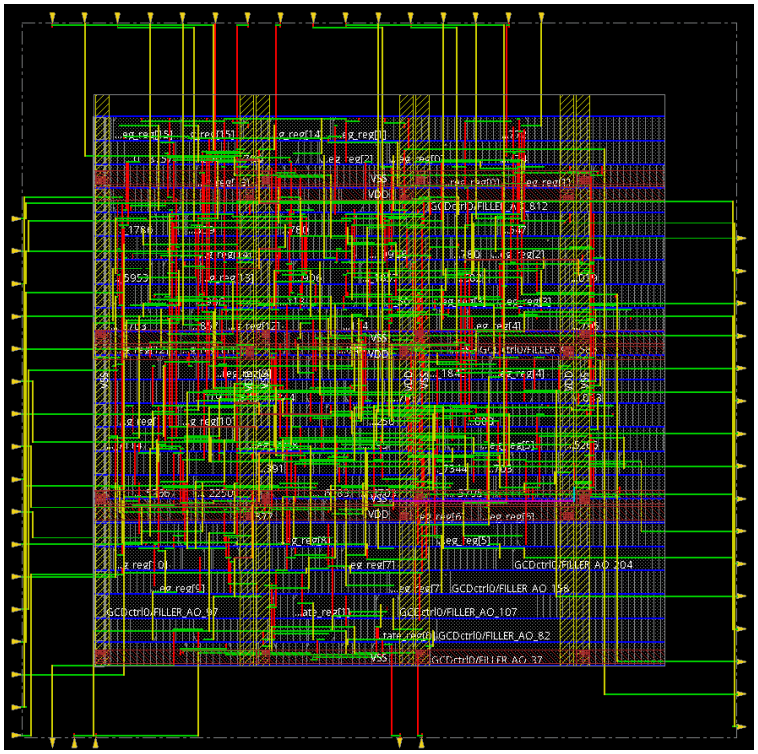


Figure 9: GCD With Pins

Go ahead and run the new lab 3 flow through HAMMER using the same commands as lab 2 and take a look (in Innovus) at the finished design by running

```
% obj/par-rundir/generated-scripts/open_chip
```

6 Conclusion

In this tutorial, you investigated the accuracy of IC Compiler and Primetime reports. Using Synopsys StarRCXT you extracted parasitics from the layout. With this information, you can simulate a path and compare its delay between schematic-only simulation, extracted simulation, and IC Compiler estimates. Also, you can measure energy based on extracted simulation results, primetime, and a mixed signal simulation of an extracted netlist through a Verilog testbench. You also learned how to write custom hooks for HAMMER for specific functions or for features not yet implemented, so (ideally) you can now use HAMMER for anything!