GCD: VLSI's Hello World

EE241B Tutorial
Updated by Daniel Grubb (2020)
Based on work by Yunsup Lee (2010), Brian Zimmer (2011, 2013),
Angie Wang (2017), Sean Huang (2019)

Overview

For this tutorial, you will become familiar with the VLSI tools you will use throughout this semester and learn how a design "flows" through the toolflow. Specifically, given an RTL model of a simple greatest common divisor (GCD) circuit, you will synthesize and place and route the design and simulate it. We will be using Hammer, a Berkeley-developed VLSI flow framework, as our main flow driver, but you will also get to touch the tools directly and see how a flow is run in other cases. The main goal of this lab is to give you an overview of a typical VLSI flow, to identify what is important for the designer to analyze, and to introduce a few practical aspects that a new VLSI designer may not have experienced before.

VLSI Toolflow Introduction

Figure 1 shows an overview of a Synopsys-tool-based VLSI toolflow. In this lab, you will use Synopsys VCS (`vcs`) to *simulate* and *debug* your RTL design. We will then be using primarily Cadence tools to run through the actual VLSI flow. You will use Cadence Genus to *synthesize* the design. Synthesis is the process of transforming an RTL model into a gate-level netlist. After obtaining a working gate-level netlist, you will use Cadence Innovus to *place and route* the design. Placement is the process by which each standard cell is positioned on the chip, while routing involves wiring the cells together using various metal layers. The tools will provide feedback on the performance and area of your design after both synthesis and place and route. The results from place and route are more realistic but require much more time to generate. Cadence Voltus can then take these outputs and give more accurate estimated power measurements and can help validate your supply network.

Prerequisites

As you can easy tell from the diagram, many different tools are needed to take even a simple design from RTL all the way to transistor-level implementation. Each tool is immensely complicated, and many engineers in industry specialize in only one. In order to produce a VLSI design in a single semester we will need to understand a little about every one.

Each tool has a GUI interface, but .tcl scripts are typically used as the primary means of driving the tool. When you use the GUI, in the terminal window you will see the textual equivalent of each click, and these commands can be added to scripts. In this lab, we will use Hammer which creates an abstraction on top of this which allows the user to create generalized APIs for the physical design flow. More information on Hammer is provided below. You will have the opportunity to interact with the tools directly, from the GUI and the command line, as well as through Hammer.

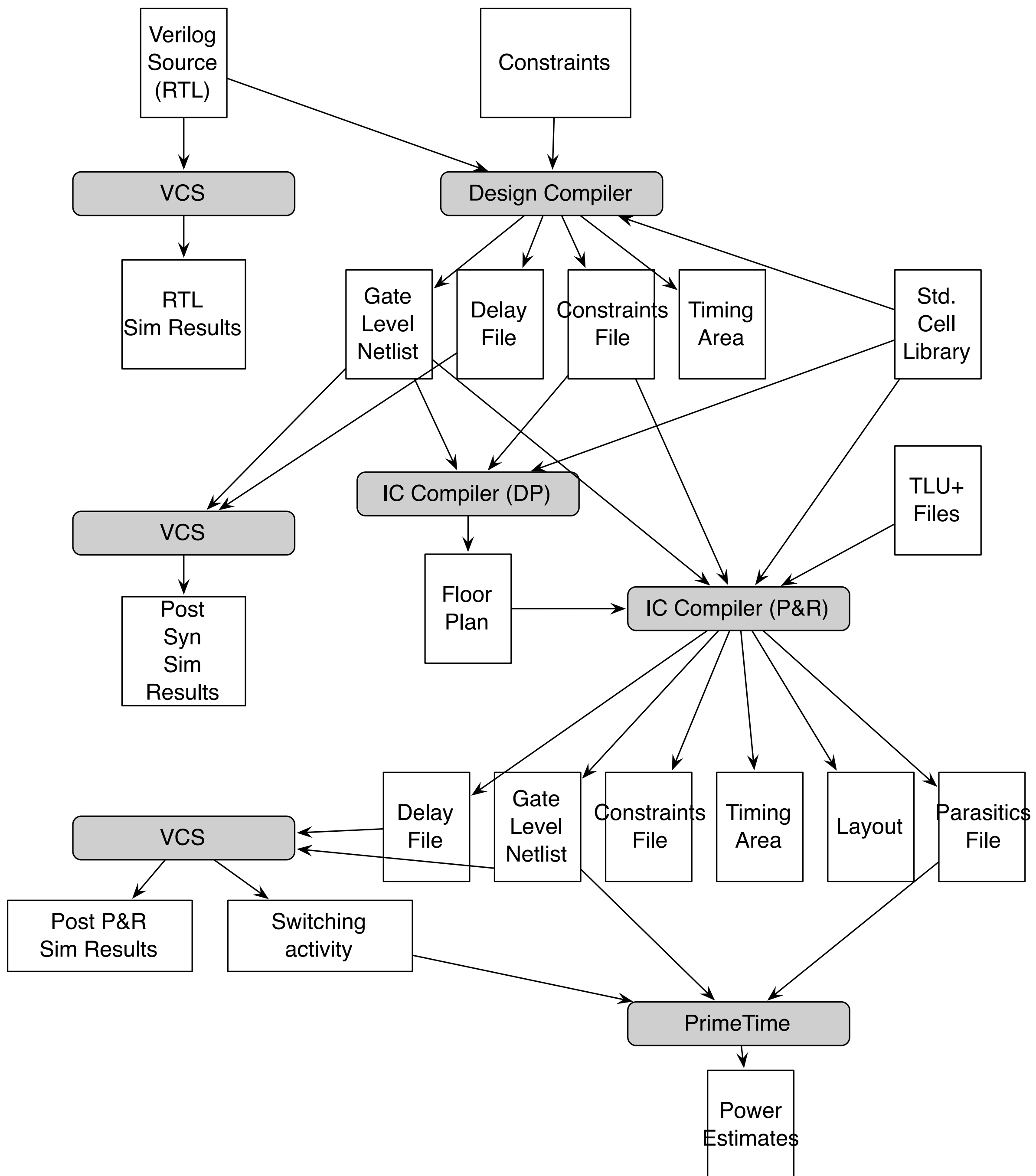

Figure 1: Synopsys VLSI Toolflow

Getting Started

All of the EE241B laboratory assignments should be completed on an EECS Instructional machines. Please see the course website and follow all of the instructions for setting up your computing resources. Remember, you will need to source a setup script in order for these instructions to work. This bash script contains the location of each tool's binary, and also sets up important environment variables. Make sure you have followed class setup instructions before starting (these are posted on the website).

Use the c125m-{1-16} or eda-{1-8}.eecs.berkeley.edu machines for these labs.

As these tools generate lots of data and home directories have too low of a disk quota, we will need to use the local disk of one of the available. Assuming your username is `userA` (change this to your own username), you can create your personal git directory using the following command.

```
% cd /scratch/
% mkdir userA
```

To begin the lab you will need to make use of a provided lab harness. This lab harness provides makefiles, scripts, and the Verilog test harness required to complete the tutorial. The following commands grab these files from the class repository. To simplify the rest of the lab we will also define a '$LABROOT' environment variable which contains the absolute path to the project's top-level root directory. If you're not using bash shell, you can switch to it by typing bash.

```
% bash
% cd /scratch/userA
% git clone ~ee241/spring20-labs/ee241bS20
% cd ee241bS20
% git submodule update --init --recursive (only needed once)
% source sourceme.sh
```

Every time you want to start the tools, you must source the sourceme in the lab directory. This will setup the environment needed for the lab tools to run.

Note: `scratch/` is a local drive, so if you every need to do work on another machine, you will need to rsync files between machines.

The resulting `$LABROOT` directory contains a src/ subdirectory which contains the verilog source code for the GCD design we will be pushing through the flow. There are 3 Hammer related submodules in the directory (hammer, hammer-cadence-plugins, and hammer-synopsys-plugins). Note that you need to be given access to the hammer-cadence-plugins and hammer-synopsys-plugins repos, since they are private. DO NOT PUBLISH THESE PUBLICLY. CLONE THEM ON THE INSTRUCTIONAL MACHINES ONLY.

RTL:

- `src/gcdGCDUnit_rtl.v` - RTL implementation of gcdGCDUnit
- `src/gcdGCDUnitCtrl.v` - Control part of the RTL implementation
- `src/gcdGCDUnitDpath.v` - Datapath part of the RTL implementation
- `src/gcdTestHarness_rtl.v` - Test harness for the RTL model

The block diagram is shown in Figure 2. Your module is named `gcdGCDUnit` and has the interface shown in Figure 3. We have provided you with a test harness that will drive the inputs and check the outputs of your design.

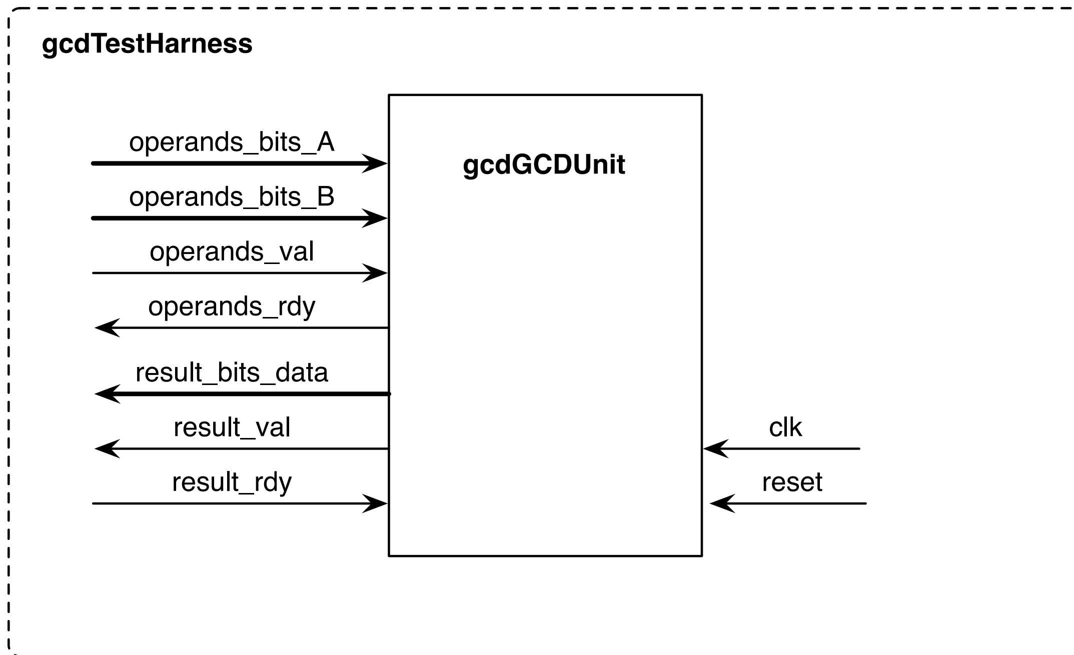

Figure 2: Block diagram for GCD Test Harness

```
module gcdGCDUnit#( parameter W = 16 )
(

  input clk, reset,

  input  [W-1:0] operands_bits_A,   // Operand A
  input  [W-1:0] operands_bits_B,   // Operand B
  input          operands_val,      // Are operands valid?
  output         operands_rdy,      // ready to take operands

  output [W-1:0] result_bits_data,  // GCD
  output         result_val,        // Is the result valid?
  input          result_rdy         // ready to take the result

);
```

Figure 3: Interface for the GCD module

Technology: ASAP7

The technology we will be using this semester is ASAP7. It is an open (free to download for academic institutions) predictive 7nm FinFET PDK developed at Arizona State University. This means that you would not be able to actually tapeout a chip in this technology, but since it is open, you can get experience working in a simulated advanced node without dealing with any pesky NDAs from a real foundry.

There are some quirks with this technology. We have created some scripts that will take the PDK and hack it (through a set of Python utilities) to make it more realistic and fully featured. Furthermore, even though the technology is listed as 7nm, it has a 4x scale factor so that the VLSI tools do not require any special licenses for running at such an advanced technology node. That means that the scripts we provide you will do more automatic hacking of your design to scale it correctly at the end.

The technology is extracted at **~ee241/spring20-labs/asap7PDK_r1p5** and **asap7libs_24**. These directories contain all of the technology files needed by a VLSI flow. In particular, look through some of the files in the **asap7libs_24** directory. Contained here are the LIBs (Liberty Timing File; contains timing information for cell delays, transitions, setup/hold times), LEFs (Library Exchange Format; contains info about physical dimensions, pin locations, metal layers, blockages), and verilog models (used in simulation) for all of the standard cells. The tools require all of this design collateral in order to place and time your design correctly. Furthermore, pretty much all real chips also include analog components or other physical macros. To integrate these blocks into your digital top flow, you need to supply the tools with all of these different files which come from characterizing the blocks in question. You will learn more about some of these formats and the different models used in lecture.

Take a closer look at **~ee241/spring20-labs/asap7libs_24/techlef_misc/asap7_tech_4x_170803.lef**. This file is called the tech lef and it specifies the physical properties of the technology itself. It specifies the layers of the design and spacing design rules for each metal (as described in lecture, there are physical manufacturing limits, for instance on how close metal can be put next to each other). The tools will use this info to make your design as DRC (Design Rules Checker) clean as possible. If you have never looked at a tech lef before, scroll through it and see what is in it. You do not need to understand all of it now, but a key part of VLSI design is being able to process the information supplied by the technology and the tools. Answer the following questions from the techlef (a good reference is located here: /share/instsww/cadence/INNOVUS181/doc/lefdefref/lefdefref.pdf):

1. **Q: How many layers are specified in the ASAP7 technology?**
2. **Q: What are the unit standard cell dimensions? (hint: take a look at the SITE entry)**
3. **Q: What is the preferred routing direction on metal layer 5?**
4. **Q: On metal 8, if there are two wires that are 0.8 um wide and run parallel to each other for 5 um, what is the minimum spacing between the two wires?**

This was just a quick intro to ASAP7 and looking at technologies in general. When running through the rest of the lab, think about which tools require which information and how it is used.

Pushing the design through all the VLSI Tools

You will now go through the entire tool flow and inspect the results after each step.

Hammer Crash Course

As briefly mentioned before, Hammer is a Python-based framework for physical design generators. One of the core driving ideas is the separation of concerns. All portions of the VLSI flow require information specific to the design, the technology used, and the CAD tools used. Hammer is designed to be a single backend that exposes a set of APIs that are then implemented in a design, technology, and tool specific way by different "plugins" supplied to it. We have a set of plugins developed for some Cadence (Genus, Innovus), Synopsys (VCS), and Mentor (DRC/LVS) tools. The Hammer ASAP7 plugin is located at **hammer/src/hammer-vlsi/technology/asap7/**. Hammer enables reuse across projects by enforcing this separation of concerns and allows for the creation of more powerful APIs which let the designer express their design more powerfully.

As these flows can be complicated, there will always be special cases that don't fit specifically into what Hammer has exposed to the user. So Hammer is design to be very flexible and configurable such that the designer can override any default steps in the flow and add new ones.

A Hammer call looks like this (first one is a general call):

```
% hammer-vlsi step -e env.yml -p input.yml --obj_dir build
% hammer-vlsi synthesis -e env.yml -p input.yml --obj_dir build
```

hammer-vlsi is the Hammer driver program (**hammer-vlsi** is the default one, but you can write a driver that extends the provided base class to insert custom steps into your flow). **step** is flow step that you want to run (eg. synthesis/syn, place-and-route/par, drc, lvs, etc.). **-e env.yml** is including env.yml as an environment config. **-p input.yml** is including input.yml as a design input. **obj_dir build** is specifying that the outputs should be placed in a subdirectory called build.

Hammer inputs are specified as keys in defined namespaces in yml and/or json files. A list of default Hammer keys can be found in **hammer/src/hammer-vlsi/defaults.yml**, and the inputs specific to this lab can be found in inst-env.yml and gcd.yml.

Here is a link to the Hammer documentation if you want to know more about it.

In this lab, we have supplied you with a Makefile which sets up the commands for you (Hammer even emits a Make include file to help with this). You will primarily execute the Make commands, but you can easily see which Hammer command is actually being run.

As mentioned before, at the end of the day, Hammer emits TCL to drive the tools, the same as any other flow. You will examine the TCL that Hammer emits to let you see behind the scenes and also interact with the GUI. Hopefully, this lab will give you some perspective on why such a tool was created, even though important features, like support for bottom-up hierarchical design, are outside the scope of this lab.

Synopsys VCS: Simulating your Verilog

VCS compiles source Verilog into a cycle-accurate executable for simulation. VCS can compile Verilog expressed behaviorally, at the RTL level, or as structural verilog (a netlist). Behavioral Verilog cannot be synthesized and should only be used in testbenches. RTL-level Verilog expresses behavior as combinational and sequential logic at a higher level. Structural-level Verilog expresses behavior as specific gates wired together. You will start with simulating the GCD module RTL (before it is synthesized).

```
% make rtl-sim
hammer-vlsi sim -e env.yml -p gcd.yml -p sim_config.yml -p rtl_sim_config.yml
./simv +verbose=1
...
  + Running Test Case: gcdGCDUnit_rtl
     [ passed ] Test ( vcTestSink ) succeeded, [ 0003 == 0003 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0007 == 0007 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0005 == 0005 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0001 == 0001 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0028 == 0028 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 000a == 000a ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0005 == 0005 ]
     [ passed ] Test ( vcTestSink ) succeeded, [ 0000 == 0000 ]
     [ passed ] Test ( Is sink finished? ) succeeded
...
```

All of the simulation outputs are placed into build/sim-rundir/. Where should you start if all of your tests didn't pass? The answer is debug your RTL using Discovery Visualization Environment (DVE) GUI looking at the trace outputs. The simulator already logged the activity for every net to the `vcdplus.vpd` file. DVE can read the `vcdplus.vpd` file and visualize the wave form.

```
% dve -vpd build/sim-rundir/vcdplus.vpd &
```

To add signals to the waveform window (see Figure 4) you can select them in the hierarchy window and then right click to choose *Add To Waves > New Wave View*.

Cadence Genus: RTL to Gate-Level Netlist

Genus performs hardware synthesis. A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as an output. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design.

gcd.yml is the main Hammer input config for this design. Take a look at some of the specified keys. There is some setup info to specify the technology being used (asap7) and tool names and versions (Innovus 18.1). There are some other important specifiers such as the power network and the clock for the design.

Q: What is the clock frequency that is initially specified for the design?

Note that the clock uncertainty is also specified in the same Hammer key. Clock uncertainty is included to add margin to design timing to account for certain clock non-idealities. This value can be selected based off of knowledge of your clock source.

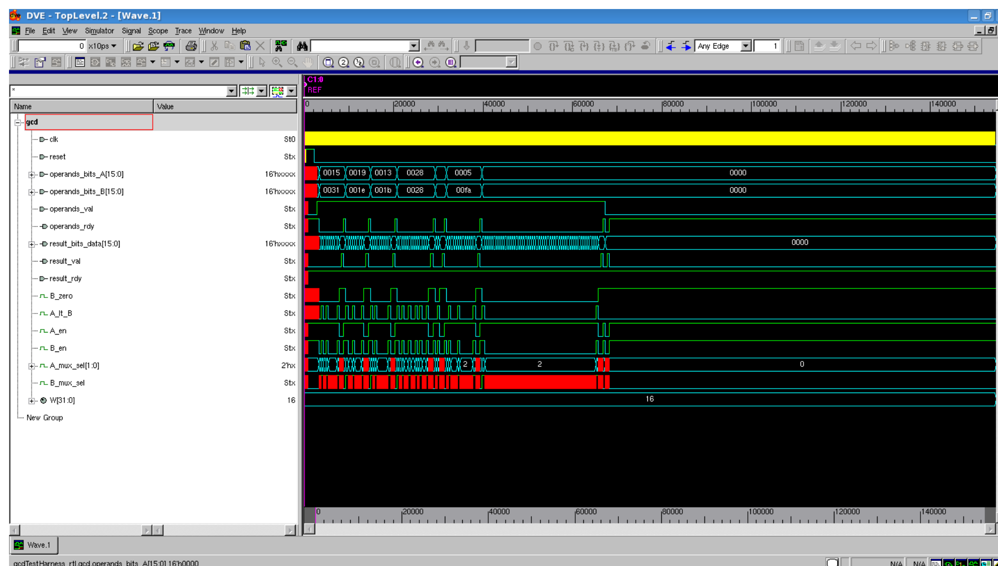

Figure 4: DVE Waveform Window

Now, run the following commands (the corresponding Hammer commands are displayed below the make command you should run):

```
% make buildfile
// hammer-vlsi build -e env.yml -p gcd.yml --obj_dir build
% make syn
// hammer-vlsi syn -e env.yml -p gcd.yml --obj_dir build
```

The buildfile command makes Hammer generate a Make include file (build/hammer.d) which contains all of the Make commands we will use for the rest of the lab. Running *make syn* actually synthesizes the design. The results are place into build/syn-rundir/. At the beginning of the Hammer output, you can see the default steps involved in our synthesis flow.

Go ahead and take a look at what Hammer produced. First look at build/syn-rundir/syn.tcl. This contains the TCL produced by Hammer that is then passed to Genus.

In build/syn-rundir/reports/, there are various reports for the design. Take a look at:

build/syn-rundir/reports/final_time_0P63V_100C.setup_view.rpt

This report specifies the timing results of the design. It lists the *critical path* of the design first. The critical path is the slowest logic between any two registers and is therefore the limiting factor preventing you from decreasing the clock period constraint. Answer the following questions:

1. **Q: What are the start and endpoints of the critical path?**
2. **Q: Does the design meet timing with the current clock frequency? If not, how much *negative slack* is there?**

3. **Q: The nominal supply voltage for this technology is 0.7V. Why does the timing report we care about specify 0.63V (0P63V), 100C (temperature), and setup_view in its name?**

If any of your paths say (VIOLATED) instead of (MET), you need to modify `gcd.yml` to increase the clock period. If you do not do this, your design will not function properly. Note also: if you make your clock period too tight for a given design, the tools (synthesis and place and route) will have trouble doing their jobs and the run time will increase!

After changing the clock frequency, re-run `make syn` and check the report again.

Q: Once your design passes timing, report the new clock period and how much positive slack there is.

Additionally, you can get a sense of the synthesis results by looking at the other final_*.rpt's in the reports directory. These contain information about area and power estimates, as well as a gate mapping summary. These reports provide important feedback about the design which helps root out problems before moving on to place-and-route.

Finally, you should also take a look at the synthesis log itself located at `build/syn-rundir/genus.logx` (where x is the highest current number). The tool outputs a lot of information, including many warnings, some benign, others not. Getting a sense of which parts of the log file are important to pay attention to is an important part of managing your VLSI flow.

Cadence Innovus: Gate-Level Netlist to Layout

Innovus performs place and route. This tool takes a synthesized gate-level netlist and a standard cell library as input and produces a layout as an output. There are many steps in the P&R process, but here we will only focus on high level concepts. If you have further interest in the specifics and how what you have learned in previous circuits classes actually gets implemented in the flow, you can start by digging into the logs, Hammer steps, and by getting more hands-on experience!

Floorplanning is a critical part of the P&R process. Floorplanning, at a high level, involves setting your boundaries, placing SRAMs and other macros, among other things. We have no macros in this design, but you can see some of the floorplanning specifications in `gcd.yml`, such as the power staps specification. In a real design, you will end up spending a considerable amount of time floorplanning to get a design with good QOR, minimal DRC violations, and limited congestion.

Q: What are the initial dimensions of the P&R boundary, as specified in gcd.yml? (hint: the values are given in microns)

Now, let's run P&R. It's pretty similar to our process for synthesis:

```
% make par
// hammer-vlsi par -e env.yml -p build/par-input.json --obj_dir build
```

A quick Hammer aside: before the Hammer **par** step is run, a Hammer transition step **syn-to-par** is run which takes the json output of synthesis and automatically transforms it into the inputs for P&R, **par-input.json**. This is one way in which Hammer is a VLSI flow tool. This is a common pattern in Hammer in terms of transitioning between different steps (you can imagine Hammer as

a graph, with steps like syn and par as nodes, and the syn-to-par, etc. steps as the edges). You can see this step in build/hammer.d.

After a few minutes, routing should have finished. Browse the results directory files in **build/par-rundir/** to see similar outputs as Genus. Look at par.tcl (there is also floorplan.tcl and mmmc.tcl) to see some of the commands that Hammer output to run with Innovus. You can see that Hammer creates an Innovus database after each Hammer step is run. This is done so that the designer can edit one of these steps manually and then continue on with the rest of the automated flow with the custom change. Just like Genus, this directory also contains **innovus.logx**. Again, there are a lot of outputs in this log, but getting to know which warnings are important is critical to becoming an effective designer.

Q: By looking at the Innovus log, did the design meet timing? How much slack is there? (hint: search for "opt_design Final Summary")

Q: By looking at the Innovus log, what is the design density? (hint: look in the same place)

There are also some timing reports you may look at, from after certain steps in the flow, in *build/par-rundir/timingReports/*.

Now open the GUI:

```
% cd $LABROOT/build/par-rundir/generated-scripts/
% ./open_chip
```

Figure 5 shows the routed signals. The Innovus GUI allows you to browse the design, which will allow you to visualize things like congestion, pin locations, and DRC violations. In the upper right corner, select *Physical View*. Then in the right-hand sidebar, select the *Cell V* checkbox, then under *Layer* deselect layers M8, M9, V7, and V8 (since those are dominated by power straps and are not used for routing). If you zoom in, you can see the locations of your standard cells and the routing. You can also see the power straps located on some of the middle routing layers. If you zoom in closer to an intersection of power straps on adjacent layers, you can see the vias between them.

The yellow triangles in the design are the top level pins of the GCD design. If this block were to be placed in a larger design, those are the locations where those locations would be routed to/from.

Q: Submit a screenshot of your design with the above visual settings set.

Take a look at the generated clock tree. Choose *Clock > CCOpt Clock Tree Debugger*. Choose *Ok*. In this diagram, the source node is the clock source and the leaf nodes are the sequential elements that the clock is distributed to. By look at the scale on the left of the diagram, you can see the insertion delay and general skew between the different registers. You can click on these nodes to highlight the nets in the physical view. (Figure 6). We typically want to see a balanced clock tree. This means that the clock will be distributed evenly across the design to the sequential elements. Having an (unintentionally) unbalanced clock tree can cause a lot of timing issues in the design. Sometimes a design will require careful clock tree design that make use of structures like H-trees.

Q: Submit a screenshot of your clock tree debugger.

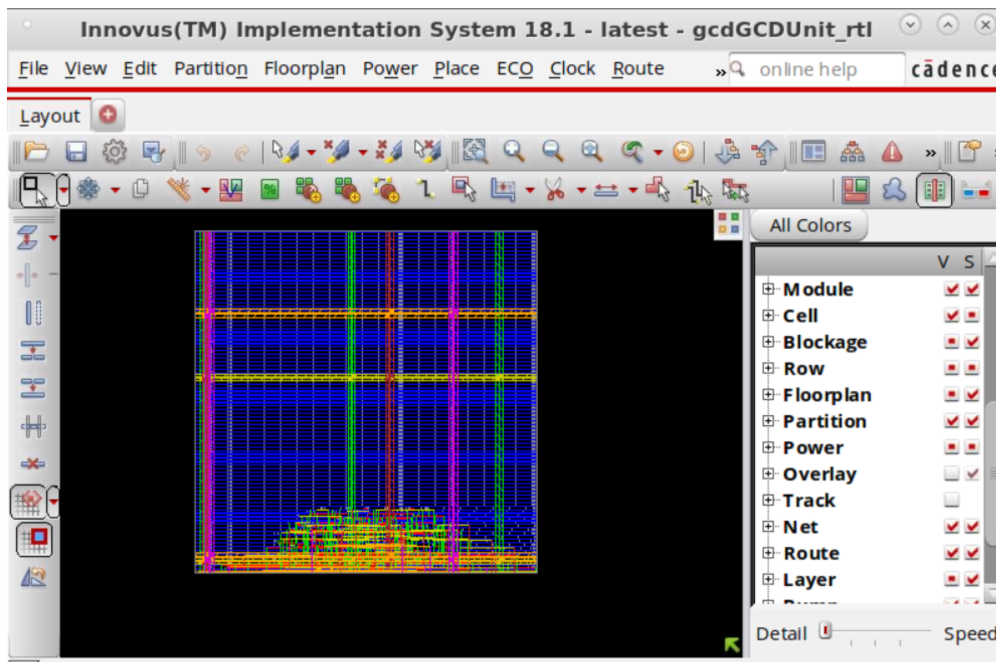

Figure 5: Routed signals shown in Innovus with larger core area

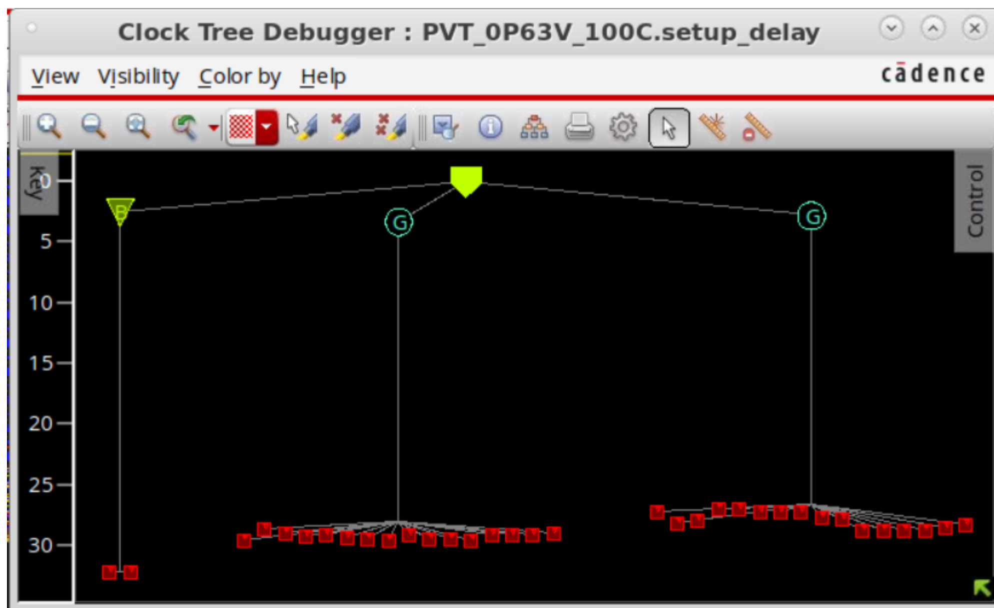

Figure 6: Clock tree debugger shown in Innovus

Remember how Hammer has Innovus save a database after every step? Well, you can open up the design at each of these steps. You can restore an Innovus design by choosing *File > Restore Design*. Choose the *Innovus* radio button, then *Restore Design File* and select the desired database, then *Ok*. The database will then pop up.

The post place-and-route netlist can be found as `build/par-rundir/gcdGCDUnit_rtl.sim.v` and the final GDS can be found at `build/par-rundir/gcdGCDUnit_rtl.gds`.

Summary

You have now taken an RTL-level description in Verilog, then simulated, synthesized, and place-and-routed the design. You used Hammer as a VLSI flow driver to create this design in the ASAP7 technology. This was just a taste to give you a baseline experience with a VLSI flow. As described before, there are many more steps and tools (and file formats to familiarize yourself with) in the flow beyond the scope of this lab to even get something that works, not to mention the many advanced power and clocking techniques used to create a good design. The lectures of the course cover many of these concepts, but the best way to get familiar with them is with hands-on experience. Hopefully, this lab gave you a baseline which you can build upon.

Some other steps in the flow not covered in this lab include gate level simulation which can help verify design functionality and timing (if it is back-annotated), static/active power and rail analysis for accurate power and to verify your power network (Cadence Voltus), static timing analysis (Cadence Tempus), logical equivalence checking (LEC), among many others. DRC (design rule checking), which checks if you design is legal, and LVS (layout vs schematic), which checks if your layout matches your netlist provide a baseline signoff for your chip.

To get exposed to more parts of the Berkeley chip-design ecosystem, check out the Chipyard project, which includes examples on how to create a Chisel-based project with a RISC-V core, integrate IP, and emulate the design on FPGAs, in addition to a slightly more advanced Hammer VLSI example.

Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Updated verssions have been developped for the EE241B Advanced Digital Circuits course, taught by Borivoje Nikolić. Contributors over the years include Brian Zimmer, Angie Wang, Bonjern Yang, Sean Huang, Daniel Grubb, and Borivoje Nikolić.

Versions of this lab have been used in the following courses:

- CS250 VLSI Systems Design (2009-2010) - University of California at Berkeley
- EE241 abd EE241B Advanced Digital Circuits - University of California, Berkeley
- CSE291 Manycore System Design (2009) - University of California at San Diego