GCD: VLSI's Hello World

CS250 Laboratory 1 (Version 090913)
Written by Yunsup Lee
Updated by Brian Zimmer, Rimas Avizienis, and Ben Keller

Overview

The goal of this assignment is to get you familiar with the VLSI CAD tools you will be using throughout the semester. You will learn about each of the stages in the ASIC tool flow, and get a chance to do some Chisel coding. Specifically, you will write an RTL model of a greatest common divisor (GCD) circuit, synthesize and place and route the design, verify its correctness by simulating at each stage (RTL, post-synthesis, post-place-and-route), and perform power analysis.

Deliverables

This lab is due **Friday, September 13th at 2PM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) build results and reports generated by VCS, DC Compiler, Formality, IC Compiler, and PrimeTime PX checked into your git repo (results and reports only!)
- (c) written answers to the questions given at the end of this document checked into your git repository as `writeup/report.pdf` or `writeup/report.txt`

You are encouraged to discuss your design with others in the class, but you must write your own code and turn in your own work.

VLSI Toolflow Introduction

Figure 1 illustrates the toolflow you will be using for the first lab. You will use Synopsys VCS (`vcs`) to *simulate* and *debug* your RTL design. After you get your design right, you will use Synopsys Design Compiler (`dc_shell-xg-t`) to *synthesize* the design. Synthesis is the process of transforming RTL into a gate-level netlist. You will use Synopsys Formality (`fm_shell`) to *formally verify* that the RTL model and the gate-level model *match*. VCS is used again to simulate the synthesized gate-level netlist. After obtaining a working gate-level netlist, you will use Synopsys IC Compiler (`icc_shell`) 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 traces on the various metal layers, with vias providing connections between the metal layers. The tools will provide feedback on the performance and area characteristics of your design after both synthesis and place-and-route (PAR). The results from PAR are more realistic but take much longer to produce. After placing and routing, you will generate and simulate the final gate-level netlist using VCS. You will use this gate-level simulation not only as a final verification step, but also to generate transition counts for every net in the design. Synopsys PrimeTime PX (`pt_shell`) takes these transition counts and correlates them with the capacitance values extracted from the final layout to estimate the power consumed by a design. The diagram below illustrates how the tools work together.

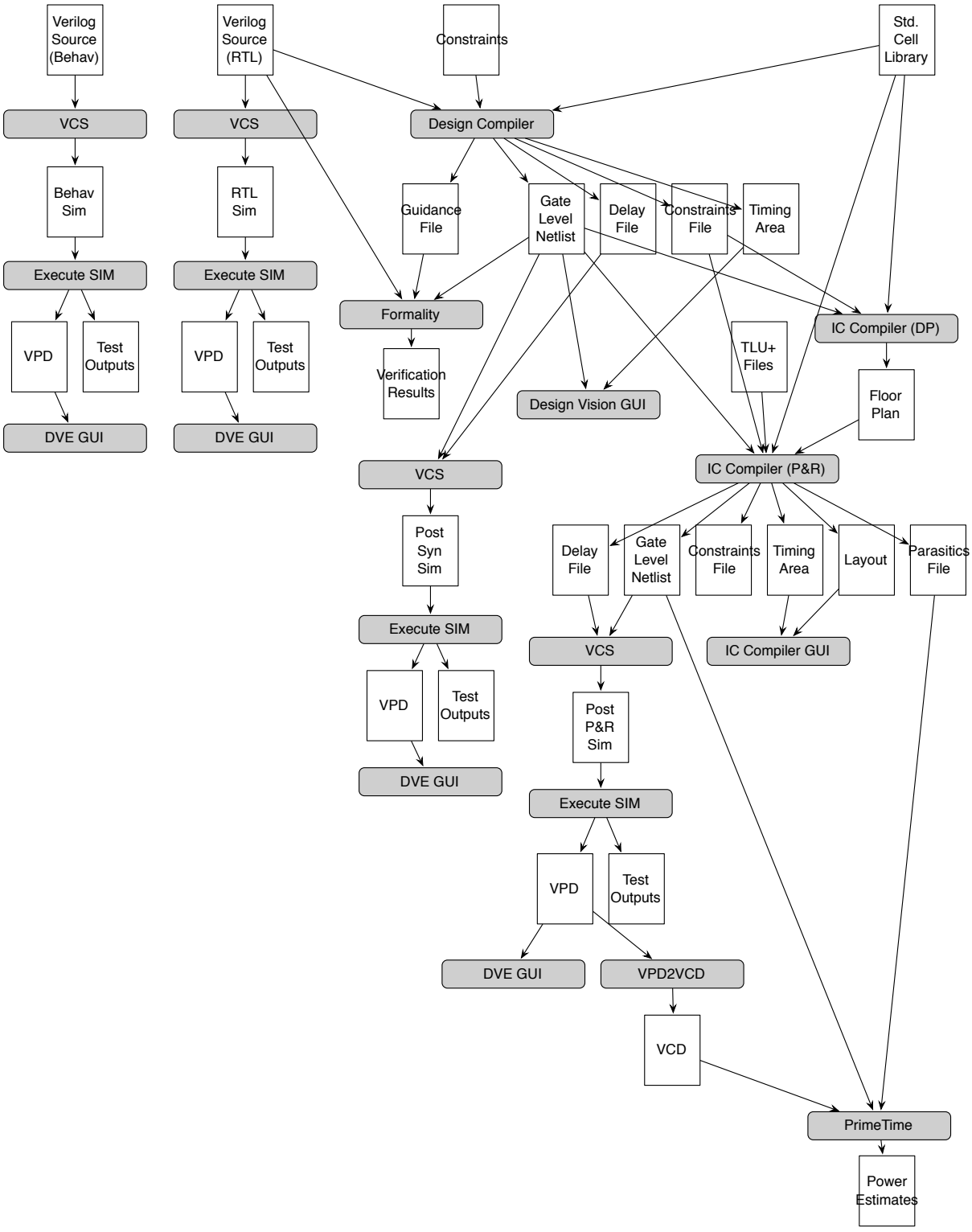

Figure 1: CS250 Toolflow for Lab 1

Prerequisites

As you can see from the diagram, many different tools are needed to take even a simple design from RTL all the way to a 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 learn a bit about each one.

Each tool has a GUI interface. However, most of the commands you'll want to execute will be the same for every design iteration and become repetitive to type, so TCL scripts are used to automate these processes. When you use the GUI, the terminal window will display the command line that corresponds to each mouse action or menu item, and these commands can be added to scripts. To keep files organized, each piece of the toolflow has its own build directory and its own Makefile. The Makefile initializes the program and points it at the setup scripts. A top-level Makefile runs each program in succession so that a single command can ideally take an RTL design all of the way through the flow without any user intervention. This will facilitate design space exploration, by making it easy for you to modify some part of your design and see how the change impacts the power, area, and performance characteristics of the resulting chip.

Getting Started

You can follow along through with this lab by typing in the commands marked with a '%' symbol at the shell prompt. To cut and paste commands from this lab into your bash shell (and make sure bash ignores the '%' character) just use an alias to "undefine" the '%' character like this:

```
% alias %=""
```

Note: OS X Preview may not copy newlines correctly. If you have problems, try using Adobe Reader.

All of the CS250 laboratory assignments should be completed on one of the EECS instructional machines allocated for the class. Please follow the setup instructions on the course website before attempting this lab. In particular, you will need to source a setup script before you can run the CAD tools. This script specifies the location of each tool and sets up necessary environment variables.

You will be using Git to manage your CS250 laboratory assignments. Please see the Git tutorial posted on the course website for more information about how to use Git. Each student will have a private git repository hosted on github.com. If you don't already have a Github account, you will need to create one. Once you have an account, you must post your Github account name and CS250 class account username on Piazza. Once you do this, your TA will create a private repository for you and you will be able to access the lab materials.

The lab materials we provide will be hosted in a *template* repository. You will clone this template repository to a directory on the machine you're working on. Afterwards, you will set your remote repository to point at your private repository. This will create a local repository that is linked to two different remote repositories (one which is managed by the staff and is read only, while the other is your private repository). If any updates are made to the *template* repository, you should be able to easily merge the changes into your local repository.

As the CAD tools generate a lot of data and your class account home directories have only a small disk quota (not to mention access speed issues with network mounted filesystems), we will need to use the local disk of the machine to store the outputs of the CAD tools. By default, the permissions on a directory that you create in `/scratch` will be set to that its contents are only readable by your class account. You will use git to backup your design files to a server hosted by github. Assuming your username is `cs250-ab` (change this to your own class account username), you can create a local working git directory on one of the EECS instructional machines using the following commands:

```
% cd /scratch
% mkdir cs250-ab
% cd cs250-ab
% git init
% git remote add template https://github.com/ucberkeley-cs250/lab-templates.git
% git remote add origin https://github.com/ucberkeley-cs250/cs250-ab.git
```

To do this the lab you will make use of some infrastructure that we have provided. The infrastructure includes Makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands fetch these files from the *template* repository, and then copy them into your private 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.

```
  % cd /scratch/cs250-ab
  % git pull template master
Username for 'https://github.com': <github username>
Password for 'https://<username>@github.com':
remote: Counting objects: 191, done.
remote: Compressing objects: 100% (136/136), done.
remote: Total 191 (delta 41), reused 188 (delta 41)
Receiving objects: 100% (191/191), 185.87 KiB | 293 KiB/s, done.
Resolving deltas: 100% (41/41), done.
From https://github.com/ucberkeley-cs250/lab-templates
 * branch            master     -> FETCH_HEAD

  % git pull origin master
  ...
  % git push origin master
  ...
  % cd lab1-verilog
  % LABROOT=$PWD
```

The two remote repositories are named *template* and *origin*. *origin* points to your private repository and *template* points to the read-only staff account. If the provided lab files are ever updated, a simple `git pull template master` should merge in these changes with your own local versions of the files.

Please run `git push origin master` frequently. `/scratch` is only intended as temporary storage and is not backed up. `/scratch` lives on a local drive, so if you ever decide to work on a different machine, you can push/pull your design files to/from Github to move your design files from one machine's local drive to the other's. Follow the instructions below to move files between machines (assuming all the files of interest have already been committed to your local repository). This procedure will not move your build directories (you will need to rerun synthesis or place-and-route to regenerate the files on the new machine), so only switch machines if there is a good reason to do so.

```
  (on machine A)
  % git push origin master
  (on machine B)
  % git pull origin master
```

The resulting `$LABROOT` directory contains the following subdirectories: `src` contains your source Verilog; `build` contains makefiles and scripts that automate the process of building your design; and `build-unscripted` is a directory for you to use when invoking the VLSI tools manually. The `src` directory contains the a Verilog test harness and other Verilog modules you will be using in this lab assignment. Please read through these files and understand how the design works. We have supplied a Verilog solution for you. Figure 2 shows each directory that you have been provided and includes comments about what they do.

We have provided two Verilog designs, a "behavioral" version, and an "RTL" version. The behavioral design somewhat resembles C code. It only describes the behavior of a module, and is not intended to be synthesized into hardware. Testbenches are typically written in a behavioral style.

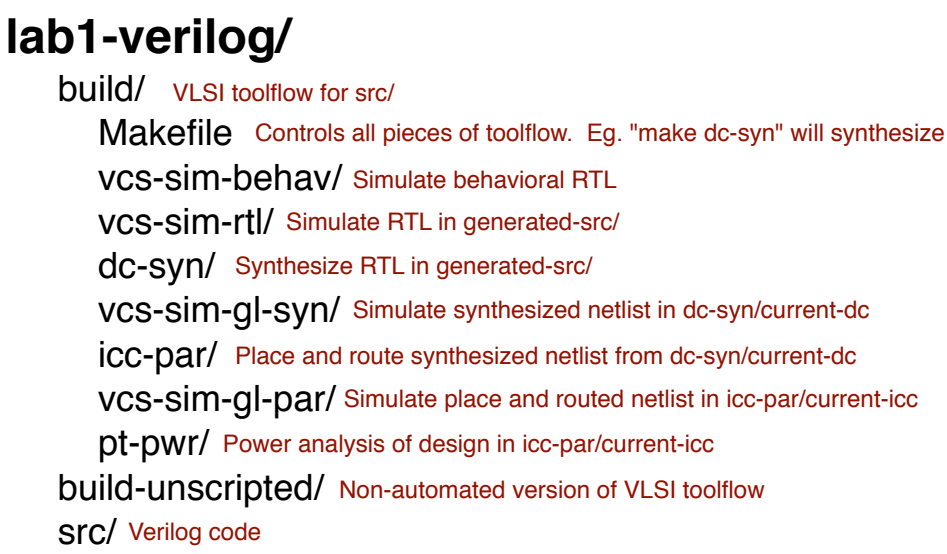

Figure 2: Directory organization for lab1-verilog/

The RTL design is a register transfer level (combinational logic and registers) description of the design. This RTL level description will be synthesized into a collection of logic gates and state elements, and eventually used to generate a complete ASIC layout.

Behavioral:

- `src/gcdGCDUnit_behav.v` - Behavioral implementation of gcdGCDUnit
- `src/gcdTestHarness_behav.v` - Test harness for the behavioral version

RTL:

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

A block diagram of the design is shown in Figure 3. The top level module is named `gcdGCDUnit` and has the interface shown in Figure 4. We have provided you with a test harness that will drive the inputs and verify the outputs of your design.

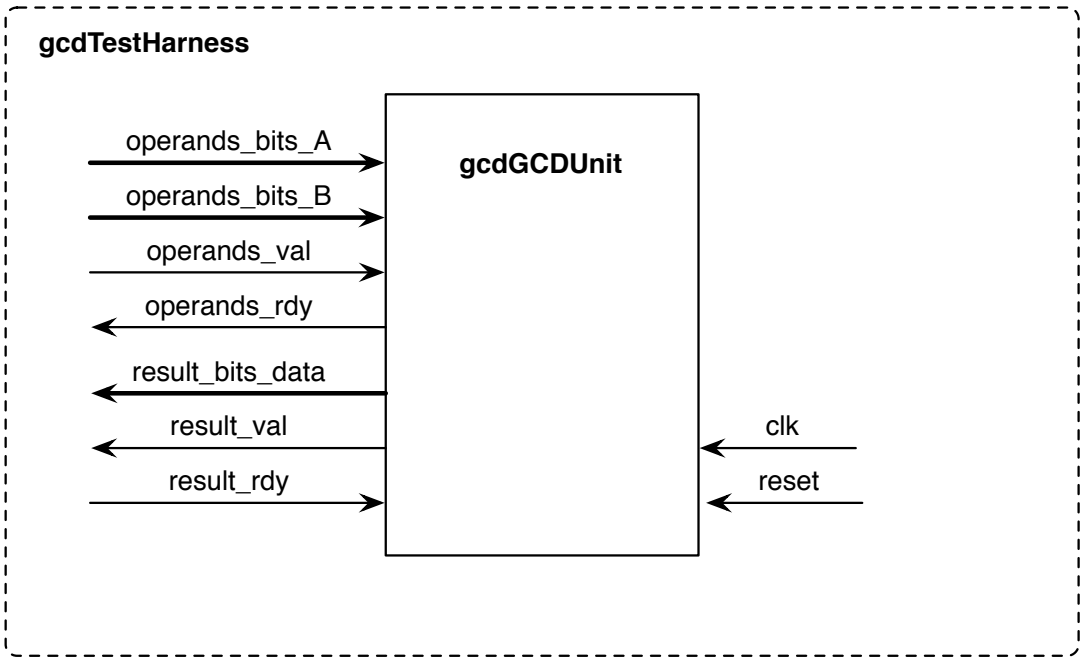

Figure 3: Block diagram for GCD Test Harness

```
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 4: Interface for the GCD module

The `build` and `build-unscripted` directory contain the following subdirectories which you will use when building your chip. The order in which they are listed is the order in which they occur in the toolflow.

- `vcs-sim-behav` - Behavioral simulation using Synopsys VCS
- `vcs-sim-rtl` - RTL simulation using Synopsys VCS
- `dc-syn` - Synthesis using Synopsys Design Compiler
- `vcs-sim-gl-syn` - Post synthesis gate-level simulation using Synopsys VCS

- `icc-par` - Automatic placement and routing using Synopsys IC Compiler
- `vcs-sim-gl-par` - Post place and route gate-level simulation using Synopsys VCS
- `pt-pwr` - Power analysis using Synopsys PrimeTime PX

Each subdirectory contains a Makefile and some contain script files as well. Running the "make" command in a subdirectory will launch the corresponding tool. For example, to perform synthesis using Design Compiler (DC):

```
% cd $LABROOT/build
% cd dc-syn
% make
```

Note: you must follow the ordering given in the list above.

Once you have verified that each of the steps in the flow succeeds, you can use the toplevel Makefile in the `build` directory to perform multiple of the steps in the flow in sequence. For example, once all the design files and scripts are set up properly, you should be able to use the following command to synthesize, floorplan, and place and route your design. You specify the name of the latest step in the flow you would like to run, and the Makefile ensures that every step preceding that one is performed first.

```
% cd $LABROOT/build
% make icc-par
```

Pushing a Design Through the VLSI Tools

First, you will invoke several of the tools manually and enter commands interactively, before learning how to automate the process with scripts. You will perform the manual build in the `build-unscripted` directory, and the automated build using makefiles and scripts in the `build` directory.

Synopsys VCS: Simulating your Verilog

VCS compiles Verilog source files into a native binary that implements a simulation of the Verilog design. VCS can simulate both behavioral and RTL level Verilog modules. In behavioral models, a module's functionality can be described more easily by using higher levels of abstraction. In RTL descriptions, a module's functionality is described at a level that can be mapped to a collection of registers and gates. Verilog behavioral models may not be synthesizable, but they can be useful in constructing testbenches and when simulating external devices that your design interfaces with. The test harness we have provided for this lab is a good example of how behavioral Verilog can be used. You will start by simulating the GCD module implemented in a behavioral style.

```
% cd $LABROOT/build-unscripted/vcs-sim-behav
% vcs -full64 -PP +lint=all +v2k -timescale=1ns/10ps \
  ../../src/gcdGCDUnit_behav.v \
  ../../src/gcdTestHarness_behav.v
```

By default, VCS produces a simulator binary called `simv`. The `-PP` command line option turns on support for using the `VPD` trace output format. The `+lint=all` argument turns on all Verilog warnings. Since it is quite easy to write legal Verilog code that doesn't behave as intended, you should always enable all warnings to help you catch mistakes. For example, VCS will warn you if you try to connect two nets with different bitwidths or don't wire up a port on a module. Always try to eliminate all VCS compilation errors *and* warnings. The `+v2k` command line option tells VCS to enable Verilog-2001 language features. Verilog allows a designer to specify how the abstract delay units in their design map into real time units using the `` `timescale `` compiler directive. To make it easy to change this parameter you will specify it on the command line instead of in the Verilog source. After these arguments you list the Verilog source files. The `-v` flag is used to indicate which Verilog files are part of a library (and thus should only be compiled if needed) and which files are part of the actual design (and thus should always be compiled). After running this command, you should see text output indicating that VCS is parsing the Verilog files and compiling the modules. Notice that VCS actually generates C++ code which is then compiled using `gcc`. When VCS is finished there should be a `simv` executable in the build directory. Now try running it:

```
% ./simv
...
 Entering Test Suite: exGCD_behav
  [ passed ] Test ( gcd(27,15) ) succeeded, [ 0003 == 00000003 ]
  [ passed ] Test ( gcd(21,49) ) succeeded, [ 0007 == 00000007 ]
  [ passed ] Test ( gcd(25,30) ) succeeded, [ 0005 == 00000005 ]
  [ passed ] Test ( gcd(19,27) ) succeeded, [ 0001 == 00000001 ]
  [ passed ] Test ( gcd(40,40) ) succeeded, [ 0028 == 00000028 ]
  [ passed ] Test ( gcd(250,190) ) succeeded, [ 000a == 0000000a ]
```

```
    [ passed ] Test ( gcd(0,0) ) succeeded, [ 0000 == 00000000 ]
  ...
```

Typing in all the Verilog source files on the command line can be very tedious, so we will use Makefiles to automate the process of compiling our simulator binaries.

```
% cd $LABROOT/build/vcs-sim-behav
% cat Makefile
...
vsrcs = \
        $(srcdir)/gcdGCDUnit_behav.v \
        $(srcdir)/gcdTestHarness_behav.v \
...
% make
% make run
```

You can leverage the same makefile to build the simulator for the RTL level Verilog version of the design.

```
% cd $LABROOT/build/vcs-sim-rtl
% cat Makefile
...
vsrcs = \
        $(srcdir)/gcdGCDUnitCtrl.v \
        $(srcdir)/gcdGCDUnitDpath.v \
        $(srcdir)/gcdGCDUnit_rtl.v \
        $(srcdir)/gcdTestHarness_rtl.v \
...
% make
% make run
./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
...
```

Where should you start if a design doesn't pass all your tests? The answer is to debug your RTL code using the Discovery Visualization Environment (DVE) GUI to generate a waveform view of signals in your design. The simulator already has already written a trace of the activity of every net in your design to the `vcdplus.vpd` file. DVE can read the `vcdplus.vpd` file and visualize the wave form.

```
% ls
csrc  Makefile  simv  simv.daidir  timestamp  vcdplus.vpd
% dve -full64 -vpd vcdplus.vpd &
```

To add signals to the waveform window (see Figure 5) you can select them in the hierarchy window and then right click to choose *Add To Waves > New Wave View*.

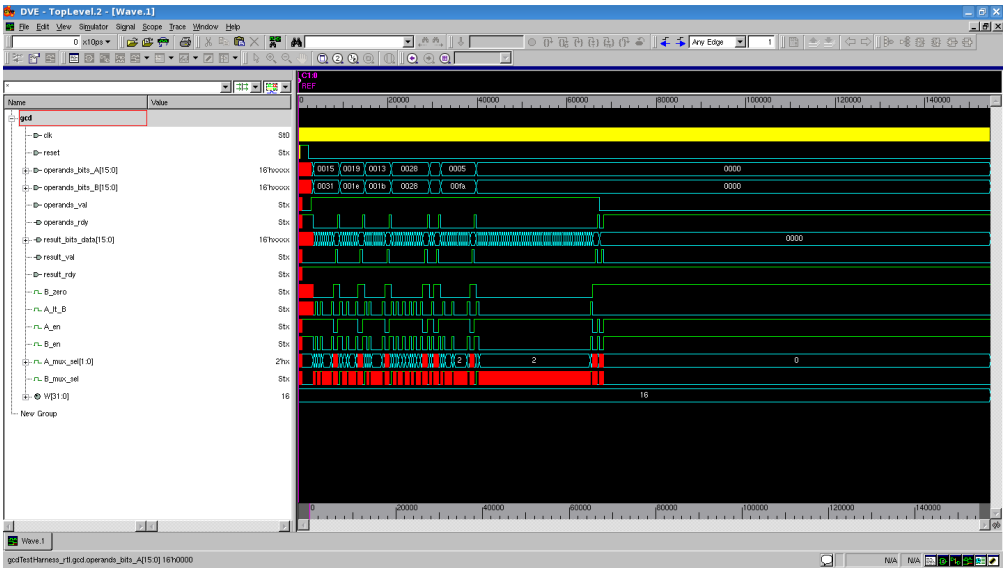

Figure 5: DVE Waveform Window

Synopsys Design Compiler: RTL to Gate-Level Netlist

Design Compiler 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 of the design, with only standard cells (and later on, possibly SRAM macros) as leaves in the design hierarchy. To cut and past commands from this lab into your Design Compiler shell and make sure Design Compiler ignores the `dc_shell-topo>` string, we will use an alias to "undefine" the `dc_shell-topo>` string.

```
% cd $LABROOT/build-unscripted/dc-syn
% dc_shell-xg-t -64bit -topographical_mode
...
Initializing...
alias "dc_shell-topo>" ""
```

You will now execute some commands to setup your environment.

```
dc_shell-topo> set ucb_vlsi_home [getenv UCB_VLSI_HOME]
dc_shell-topo> set stdcells_home \
  $ucb_vlsi_home/stdcells/synopsys-32nm/typical_rvt
dc_shell-topo> set_app_var search_path \
  "$stdcells_home/db $ucb_vlsi_home/install/vclib ../../src"
```

```
dc_shell-topo> set_app_var target_library "cells.db"
dc_shell-topo> set_app_var synthetic_library "dw_foundation.sldb"
dc_shell-topo> set_app_var link_library "* $target_library $synthetic_library"
dc_shell-topo> set_app_var alib_library_analysis_path "alib"
dc_shell-topo> set_app_var mw_logic1_net "VDD"
dc_shell-topo> set_app_var mw_logic0_net "VSS"
dc_shell-topo> create_mw_lib -technology $stdcells_home/techfile/techfile.tf \
  -mw_reference_library $stdcells_home/mw/cells.mw "gcdGCDUnit_rtl_LIB"
dc_shell-topo> open_mw_lib "gcdGCDUnit_rtl_LIB"
dc_shell-topo> check_library
dc_shell-topo> set_tlu_plus_files \
  -max_tluplus $stdcells_home/tluplus/max.tluplus \
  -min_tluplus $stdcells_home/tluplus/min.tluplus \
  -tech2itf_map $stdcells_home/techfile/tech2itf.map
dc_shell-topo> check_tlu_plus_files
dc_shell-topo> define_design_lib WORK -path "./work"
dc_shell-topo> set_svf "gcdGCDUnit_rtl.svf"
```

These commands point to your Verilog source directory, create a Synopsys work directory, and point to the standard libraries you will be using for the class. The `set_svf` command is used to set up a guidance file which is used by Synopsys Formality. Now you can load your Verilog design in to Design Compiler with the `analyze`, `elaborate`, `link`, and `check_design` commands.

```
dc_shell-topo> analyze -format verilog \
  "gcdGCDUnitCtrl.v gcdGCDUnitDpath.v gcdGCDUnit_rtl.v"
dc_shell-topo> elaborate "gcdGCDUnit_rtl"
dc_shell-topo> link
dc_shell-topo> check_design
```

Before you can synthesize your design, you must specify some constraints; most importantly you must tell the tool your target clock period. The following command tells the Design Compiler that the pin named `clk` is the clock and that your target clock period is 1 nanosecond.

```
dc_shell-topo> create_clock clk -name ideal_clock1 -period 1
```

Now you are ready to use the `compile_ultra` command to actually synthesize your design into a gate-level netlist. `-no_autoungroup` is specified in order to preserve the hierarchy during syntehsis.

```
dc_shell-topo> compile_ultra -gate_clock -no_autoungroup
 ...
 Beginning Delay Optimization
 ----------------------------
   0:00:04    3113.2      0.02       0.1       0.0
   0:00:04    3142.7      0.00       0.0       0.0
   0:00:04    3142.7      0.00       0.0       0.0
   0:00:04    3222.8      0.00       0.0       0.0
   0:00:04    3222.8      0.00       0.0       0.0
   0:00:04    3222.8      0.00       0.0       0.0
   0:00:04    3222.8      0.00       0.0       0.0
```

```
  0:00:04    3222.8      0.00       0.0       0.0
  0:00:04    3222.8      0.00       0.0       0.0
  0:00:04    3222.8      0.00       0.0       0.0
...
```

The `compile_ultra` command will report how the design is being optimized. You should see Design Compiler performing technology mapping, delay optimization, and area reduction. The fragment from `compile_ultra` above shows the worst negative slack which indicates how much room there is between the critical path in your design and your specified clock constraint. Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a great amount. Total negative slack is the sum of all negative slack summed over all the endpoints (register inputs or top level input/output ports) in the design.

Now you can generate the guidance information required by the formal verification tool, and produce the synthesized gate-level netlist and derived constraints files.

```
dc_shell-topo> set_svf -off
dc_shell-topo> change_names -rules verilog -hierarchy
dc_shell-topo> write -format ddc -hierarchy -output gcdGCDUnit_rtl.mapped.ddc
dc_shell-topo> write -f verilog -hierarchy -output gcdGCDUnit_rtl.mapped.v
dc_shell-topo> write_sdf gcdGCDUnit_rtl.mapped.sdf
dc_shell-topo> write_sdc -nosplit gcdGCDUnit_rtl.mapped.sdc
dc_shell-topo> write_milkyway -overwrite -output "gcdGCDUnit_rtl_DCT"
dc_shell-topo> source ./find_regs.tcl
dc_shell-topo> find_regs gcdTestHarness_rtl/gcd
```

Take a look at various reports describing the synthesis results.

```
dc_shell-topo> report_timing -transition_time -nets -attributes -nosplit
...
Point                                    Fanout     Trans      Incr       Path
---------------------------------------------------------------------------------
clock ideal_clock1 (rise edge)                                 0.00       0.00
clock network delay (ideal)                                    0.00       0.00
GCDdpath0/A_reg_reg_2_/CLK (DFFARX1_RVT)             0.00      0.00       0.00 r
GCDdpath0/A_reg_reg_2_/Q (DFFARX1_RVT)               0.04      0.11       0.11 r
GCDdpath0/result_bits_data[2] (net)        5                   0.00       0.11 r
GCDdpath0/sub_x_2_U56/Y (NAND2X0_RVT)                0.08      0.06 *     0.17 f
GCDdpath0/sub_x_2_n44 (net)                3                   0.00       0.17 f
 ...
GCDdpath0/sub_x_2_U11/Y (OAI21X1_RVT)                0.02      0.07 *     0.37 r
GCDdpath0/sub_x_2_n16 (net)                1                   0.00       0.37 r
GCDdpath0/sub_x_2_U9/CO (FADDX1_RVT)                 0.03      0.06 *     0.43 r
GCDdpath0/sub_x_2_n15 (net)                1                   0.00       0.43 r
...
GCDdpath0/sub_x_2_n14 (net)                1                   0.00       0.49 r
GCDdpath0/sub_x_2_U7/CO (FADDX1_RVT)                 0.03      0.06 *     0.55 r
GCDdpath0/sub_x_2_n13 (net)                1                   0.00       0.55 r
GCDdpath0/sub_x_2_U6/CO (FADDX1_RVT)                 0.04      0.06 *     0.61 r
```

```
GCDdpath0/sub_x_2_n12 (net)               1                  0.00       0.61 r
GCDdpath0/sub_x_2_U5/CO (FADDX1_RVT)                0.03     0.06 *     0.67 r
...
GCDdpath0/sub_x_2_n9 (net)                1                  0.00       0.80 r
GCDdpath0/sub_x_2_U1/Y (XOR2X1_RVT)                 0.03     0.08 *     0.87 f
GCDdpath0/sub_out[15] (net)               1                  0.00       0.87 f
GCDdpath0/U38/Y (AO222X1_RVT)                       0.02     0.04 *     0.91 f
GCDdpath0/A_next[15] (net)                1                  0.00       0.91 f
GCDdpath0/A_reg_reg_15_/D (DFFARX1_RVT)             0.02     0.00 *     0.91 f
data arrival time                                                       0.91

clock ideal_clock1 (rise edge)                                 1.00       1.00
clock network delay (ideal)                                    0.00       1.00
GCDdpath0/A_reg_reg_15_/CLK (DFFARX1_RVT)                      0.00       1.00 r
library setup time                                            -0.03       0.97
data required time                                                      0.97
--------------------------------------------------------------------------------
data arrival time                                                      -0.91
--------------------------------------------------------------------------------
slack (MET)                                                             0.06
...
```

This report shows the *critical path* of the design. The critical path has the longest propagation delay between any two registers in the design and therefore sets an upper bound on the design's operating frequency. In this report, we see that the critical path begins at the output of bit 2 of the operand A register in the datapath, goes several subtractors in the datapath, and ends at output register 15. The critical path takes a total of 0.91ns which is less than the 1ns clock period constraint. This is reflected by the final line declaring that the positive slack has met timing.

```
dc_shell-topo> report_area -nosplit -hierarchy
...
                          Global cell area          Local cell area
                          ------------------  ----------------------------
Hierarchical cell         Absolute   Percent  Combi-     Noncombi-  Black
                          Total      Total    national   national   boxes
------------------------  ---------  -------  ---------  ---------  ------
gcdGCDUnit_rtl             656.1998    100.0     3.0497     0.0000  0.0000
GCDctrl0                    50.8288      7.7    37.6133    13.2155  0.0000
GCDdpath0                  602.3213     91.8   362.9176   227.7130  0.0000
...
------------------------  ---------  -------  ---------  ---------  ------
Total                                         403.5807   252.6191  0.0000
...
```

This report tells you about the post synthesis area results. The units are um^2. You can see that the datapath accounts for 91.8% of the total chip area.

```
dc_shell-topo> report_power -nosplit -hier
...
```

```
--------------------------------------------------------------------------------
                                  Switch   Int      Leak     Total
Hierarchy                         Power    Power    Power    Power    %
--------------------------------------------------------------------------------
gcdGCDUnit_rtl                    57.283  209.307 6.53e+07  331.874 100.0
  GCDctrl0 (gcdGCDUnitCtrl)        3.988   16.366 4.55e+06   24.901   7.5
  GCDdpath0 (gcdGCDUnitDpath_W16) 51.012  192.593 6.02e+07  303.794  91.5
...
```

This report contains post synthesis power estimates. The dynamic power units are uW while the leakage power units are pW.

```
dc_shell-topo> report_reference -nosplit -hierarchy
...
****************************************
Design: gcdGCDUnitDpath_W16
****************************************
Reference                    Library       Unit Area   Count    Total Area
--------------------------------------------------------------------------------
AND2X1_RVT          saed32rvt_tt1p05v25c     2.033152       1     2.033152
AND2X2_RVT          saed32rvt_tt1p05v25c     2.287296       1     2.287296
AO22X1_RVT          saed32rvt_tt1p05v25c     2.541440      13    33.038720
AO222X1_RVT         saed32rvt_tt1p05v25c     3.303872      16    52.861954
AOI21X1_RVT         saed32rvt_tt1p05v25c     3.049728       4    12.198912
AOI22X1_RVT         saed32rvt_tt1p05v25c     3.049728       1     3.049728
DFFARX1_RVT         saed32rvt_tt1p05v25c     7.116032      32   227.713028
FADDX1_RVT          saed32rvt_tt1p05v25c     4.828736       7    33.801151
INVX1_RVT           saed32rvt_tt1p05v25c     1.270720      11    13.977920
MUX21X1_RVT         saed32rvt_tt1p05v25c     3.303872      16    52.861954
NAND2X0_RVT         saed32rvt_tt1p05v25c     1.524864      27    41.171327
NAND3X0_RVT         saed32rvt_tt1p05v25c     1.779008       2     3.558016
NAND4X0_RVT         saed32rvt_tt1p05v25c     2.033152       4     8.132608
NOR2X0_RVT          saed32rvt_tt1p05v25c     2.541440      11    27.955840
NOR2X2_RVT          saed32rvt_tt1p05v25c     2.795584       2     5.591168
NOR4X1_RVT          saed32rvt_tt1p05v25c     3.049728       1     3.049728
OA22X1_RVT          saed32rvt_tt1p05v25c     2.541440       1     2.541440
OAI21X1_RVT         saed32rvt_tt1p05v25c     3.049728       7    21.348095
SNPS_CLOCK_GATE... saed32rvt_tt1p05v25c     5.845312       1     5.845312
SNPS_CLOCK_GATE... saed32rvt_tt1p05v25c     5.845312       1     5.845312
XNOR2X1_RVT         saed32rvt_tt1p05v25c     4.320448       4    17.281792
XOR2X1_RVT          saed32rvt_tt1p05v25c     4.320448       5    21.602240
XOR2X2_RVT          saed32rvt_tt1p05v25c     4.574592       1     4.574592
--------------------------------------------------------------------------------
Total 23 references                                              602.321285
...
```

This report lists the standard cells used to implement each module. The `gcdGCDUnitDpath` module consists of 13 `AO22X1` cells, 32 `DFFARX1` cells, and so on. The total area consumed by each type of cell is also reported.

```
dc_shell-topo> report_resources -nosplit -hierarchy
...
Resource Report for this hierarchy in file ../../src/gcdGCDUnitDpath.v
===============================================================================
| Cell           | Module         | Parameters | Contained Operations       |
===============================================================================
| sub_x_28_1     | DW01_sub       | width=16   | sub_28                     |
| lt_x_46_1      | DW_cmp         | width=16   | lt_46                      |
===============================================================================

Implementation Report
===============================================================================
|                    |                  | Current            | Set            |
| Cell               | Module           | Implementation     | Implementation |
===============================================================================
| sub_x_28_1         | DW01_sub         | pparch (area,speed)                 |
| lt_x_46_1          | DW_cmp           | apparch (area)     |                |
===============================================================================
...
```

Synopsys provides a library of commonly used arithmetic components as highly optimized building blocks. This library is called Design Ware and Design Compiler will automatically use Design Ware components when it can. This report can help you determine when Design Compiler is using Design Ware components. The `DW01_sub` in the module name indicates that this is a Design Ware subtractor. This report also gives you what type of architecture it used.

You can use makefiles and scripts to help automate the process of synthesizing your design. Type `exit` to leave the DC shell. Then type:

```
% cd $LABROOT/build/dc-syn
% cat Makefile
...
vsrcs = \
        $(srcdir)/gcdGCDUnitCtrl.v \
        $(srcdir)/gcdGCDUnitDpath.v \
        $(srcdir)/gcdGCDUnit_rtl.v \
...
% make
```

Go ahead and take a look what the automated build system produced.

```
% cd $LABROOT/build/dc-syn
% ls -l
-rw-r--r-- 1 cs250-ab cs250 4555 Aug 29 22:15 Makefile
drwxr-xr-x 7 cs250-ab cs250 4096 Aug 29 22:15 build-dc-2010-08-29_22-15
-rw-r--r-- 1 cs250-ab cs250 1108 Aug 28 12:06 constraints.tcl
lrwxrwxrwx 1 cs250-ab cs250   25 Aug 29 22:15 current-dc -> build-dc-2010-08-29_22-15
drwxr-xr-x 2 cs250-ab cs250 4096 Aug 29 21:39 rm_dc_scripts
drwxr-xr-x 2 cs250-ab cs250 4096 Aug 28 12:00 rm_notes
```

```
drwxr-xr-x 2 cs250-ab cs250 4096 Aug 29 21:50 rm_setup
% cd current-dc
% ls -l
drwxr-xr-x 2 cs250-ab cs250   4096 Aug 29 22:15 WORK
-rw-r--r-- 1 cs250-ab cs250     47 Aug 29 22:15 access.tab
-rw-r--r-- 1 cs250-ab cs250 235827 Aug 29 22:15 command.log
-rw-r--r-- 1 cs250-ab cs250   5141 Aug 29 22:15 common_setup.tcl
-rw-r--r-- 1 cs250-ab cs250   1108 Aug 29 22:15 constraints.tcl
-rw-r--r-- 1 cs250-ab cs250  18996 Aug 29 22:15 dc.tcl
-rw-r--r-- 1 cs250-ab cs250   4621 Aug 29 22:15 dc_setup.tcl
-rw-r--r-- 1 cs250-ab cs250   4625 Aug 29 22:15 dc_setup_filenames.tcl
-rw-r--r-- 1 cs250-ab cs250   2730 Aug 29 22:15 find_regs.tcl
-rw-r--r-- 1 cs250-ab cs250   4439 Aug 29 22:15 force_regs.ucli
drwxr-xr-x 3 cs250-ab cs250   4096 Aug 29 22:15 gcdGCDUnit_rtl_LIB
drwxr-xr-x 2 cs250-ab cs250   4096 Aug 29 22:15 log
-rw-r--r-- 1 cs250-ab cs250   1087 Aug 29 22:15 make_generated_vars.tcl
drwxr-xr-x 2 cs250-ab cs250   4096 Aug 29 22:15 reports
drwxr-xr-x 2 cs250-ab cs250   4096 Aug 29 22:15 results
-rw-r--r-- 1 cs250-ab cs250     29 Aug 29 22:15 timestamp
```

Notice that the Makefile does not overwrite build directories. It creates a new build directory every time you run make. This makes it easy to change your synthesis scripts or source Verilog, resynthesize your design, and compare your results to those from an earlier design. You can use symlinks to keep track of various build directories. Inside the `current-dc` directory, you can see all the tcl scripts as well as the directories named `results` and `reports`. `results` contains your synthesized gate-level netlist, and `reports` contains various post synthesis reports.

Synopsys provides a GUI front-end for Design Compiler called Design Vision which you will use to analyze the synthesis results. You should avoid using the GUI to actually perform synthesis since scripting the process is more efficient. Start Design Vision, and open the .ddc file to load your synthesized design. (.ddc is a proprietary binary format used by Synopsys to encapsulate all post-synthesis data.)

```
% cd $LABROOT/build/dc-syn/current-dc
% design_vision-xg -64bit
...
Initializing...
design_vision> alias "design_vision>" ""
design_vision> source dc_setup.tcl
design_vision> read_file -format ddc "results/gcdGCDUnit_rtl.mapped.ddc"
```

You can browse your design with the hierarchical view (see Figure 6). If you right click on a module and select the *Schematic View* option, the tool will display a schematic view of the standard cells used to implement that module.

Synopsys Formality: Formal Verification

Formality formally verifies whether or not the RTL and the synthesized gate-level netlist match.

```
% cd $LABROOT/build-unscripted/dc-syn
```

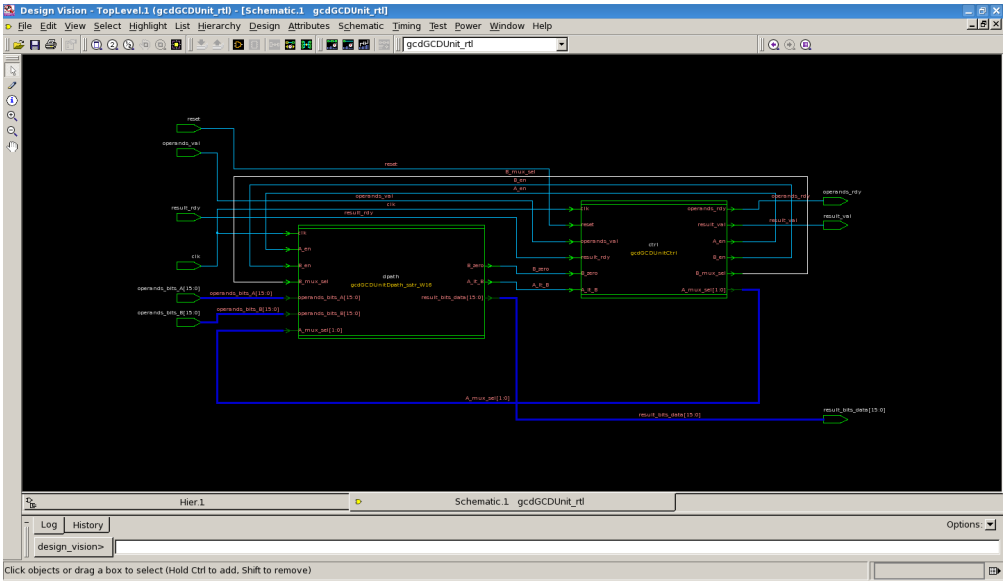

Figure 6: Design Vision Hierarchical View

```
% fm_shell -64bit
...
fm_shell (setup)> alias "fm_shell" ""
fm_shell (setup)> alias "(setup)>" ""
fm_shell (setup)> alias "(match)>" ""
fm_shell (setup)> alias "(verify)>" ""
```

Execute some commands to setup your environment.

```
fm_shell (setup)> set ucb_vlsi_home [getenv UCB_VLSI_HOME]
fm_shell (setup)> set stdcells_home \
  $ucb_vlsi_home/stdcells/synopsys-32nm/default
fm_shell (setup)> set_app_var search_path \
  "$stdcells_home/db ~cs250/install/vclib ../../src"
fm_shell (setup)> set_app_var synopsys_auto_setup "true"
fm_shell (setup)> set_svf "gcdGCDUnit_rtl.svf"
fm_shell (setup)> read_db -technology_library "cells.db"
```

These commands point to your Verilog source directory, the svf file you generated during synthesis, and the standard libraries you will be using for the class. Now go ahead and open your original RTL design and the synthesized gate-level netlist in Formality.

```
fm_shell (setup)> read_verilog -r \
  "gcdGCDUnitCtrl.v gcdGCDUnitDpath.v gcdGCDUnit_rtl.v" \
  -work_library WORK
fm_shell (setup)> set_top r:/WORK/gcdGCDUnit_rtl
fm_shell (setup)> read_ddc -i "./gcdGCDUnit_rtl.mapped.ddc"
fm_shell (setup)> set_top i:/WORK/gcdGCDUnit_rtl
```

You are ready to begin verification to determine whether or not the two representations of the design match are equivalent.

```
fm_shell (setup)> match
fm_shell (match)> report_unmatched_points
fm_shell (match)> verify
...
************************ Verification Results ************************
Verification SUCCEEDED
   ATTENTION:  synopsys_auto_setup mode was enabled.
               See Synopsys Auto Setup Summary for details
-----------------------------------------------------------
 Reference design: r:/WORK/gcdGCDUnit_rtl
 Implementation design: i:/WORK/gcdGCDUnit_rtl
 52 Passing compare points
-------------------------------------------------------------------------
Matched Compare Points   BBPin  Loop BBNet   Cut  Port   DFF   LAT TOTAL
-------------------------------------------------------------------------
Passing (equivalent)         0     0     0     0    18    34     0    52
Failing (not equivalent)     0     0     0     0     0     0     0     0
*************************************************************************
...
```

You may have noticed that while Formality is able to match all of the points in the reference and implementation design, it also identified several objects in the implementation design that are missing from the reference:

```
-------------------------------------------------------------------------------
Unmatched Objects                                              REF         IMPL
-------------------------------------------------------------------------------
 Registers                                                       0            2
   Clock-gate LAT                                                0            2
*******************************************************************************
```

These clock-gating latches are automatically inserted by Design Compiler. They idle parts of the clock network that are not needed to save power on a cycle-by-cycle basis, and should not affect the functionality of the design. Nonetheless, benign mismatched objects such as these show the importance of using simulations as well as formal verification to check a synthesized design.

We have provided you with Makefiles and scripts which automate the formal verification procedure.

```
% cd $LABROOT/build/dc-syn
% make fm
```

This will not generate a separate build directory, but will verify the latest design and write a report to its `reports` directory.

Synopsys VCS: Simulating Post Synthesis Gate-Level Netlist

After generating a synthesized gate-level netlist, you will verify that the netlist behaves as expected by running another simulation using VCS.

```
% cd $LABROOT/build-unscripted/dc-syn
% sdfcorrect.py gcdGCDUnit_rtl.mapped.sdf gcdGCDUnit_rtl.mapped.corrected.sdf
% cd $LABROOT/build-unscripted/vcs-sim-gl-syn
% vcs -full64 -PP +lint=all +v2k -timescale=1ns/10ps \
  -P ../dc-syn/access.tab -debug \
  +neg_tchk +sdfverbose \
  -sdf typ:gcdGCDUnit_rtl:../dc-syn/gcdGCDUnit_rtl.mapped.corrected.sdf \
  +incdir+$UCB_VLSI_HOME/install/vclib \
  -v $UCB_VLSI_HOME/install/vclib/vcQueues.v \
  -v $UCB_VLSI_HOME/install/vclib/vcStateElements.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTest.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTestSource.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTestSink.v \
  -v $UCB_VLSI_HOME/stdcells/synopsys-32nm/default/verilog/cells.v \
  ../dc-syn/gcdGCDUnit_rtl.mapped.v \
  ../../src/gcdTestHarness_rtl.v \
  +define+functional=1 \
  +define+CLOCK_PERIOD=1.0
% ./simv -ucli +verbose=1
ucli% source ../dc-syn/force_regs.ucli
ucli% run
...
  + 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
...
```

Makefiles are provided which build the simulator and execute the testbench.

```
% cd $LABROOT/build/vcs-sim-gl-syn
% make
% make run
```

Synopsys IC Compiler: Gate-Level Netlist to Layout

IC Compiler performs placement and routing of the standard cells in your design. It takes a gate-level netlist, standard cell library, floorplan information, and technology files describing the metal layers available for routing as input and produces a layout as an output. After this step, you can visually inspect the standard cells and routes on the metal layers. For this step, you will use IC Compiler in an interactive fashion, so go ahead and launch it with its GUI enabled.

```
% cd $LABROOT/build-unscripted/icc-par
```

```
% icc_shell -64bit -gui
...
Initializing...
icc_shell> alias "icc_shell>" ""
```

Execute the following commands to setup your environment:

```
icc_shell> set ucb_vlsi_home [getenv UCB_VLSI_HOME]
icc_shell> set stdcells_home \
  $ucb_vlsi_home/stdcells/synopsys-32nm/default
icc_shell> set_app_var search_path "$stdcells_home/db"
icc_shell> set_app_var target_library "cells.db"
icc_shell> set_app_var link_library "* $target_library"
```

These commands point the tool to the standard libraries you will be using in this class. Before you begin place-and-route, you will create a Milkyway database in which to store your placed and routed design. Note that when you create the Milkyway database you specify the technology file which describes the design rules of the target process (e.g., detailed information about allowed thicknesses and spacings on the poly and metal layers), and the Milkyway reference database which contains the layouts of the standard cells. Next, load your synthesized gate-level netlist from the `dc-syn` directory. You must also specify the TLU+ files which include information necessary to perform parasitic extraction for calculating wire delays. You also need to create power and ground ports.

```
icc_shell> create_mw_lib \
  -tech "$stdcells_home/techfile/techfile.tf" \
  -bus_naming_style {[%d]} \
  -mw_reference_library "$stdcells_home/mw/cells.mw" \
  "gcdGCDUnit_rtl_LIB"
icc_shell> open_mw_lib gcdGCDUnit_rtl_LIB
icc_shell> import_designs "../dc-syn/gcdGCDUnit_rtl.mapped.ddc" \
  -format "ddc" -top "gcdGCDUnit_rtl" -cel "gcdGCDUnit_rtl"
icc_shell> set_tlu_plus_files \
  -max_tluplus "$stdcells_home/tluplus/max.tluplus" \
  -min_tluplus "$stdcells_home/tluplus/min.tluplus" \
  -tech2itf_map "$stdcells_home/techfile/tech2itf.map"
icc_shell> derive_pg_connection \
  -power_net "VDD" -power_pin "VDD" -ground_net "VSS" -ground_pin "VSS" \
  -create_ports "top"
```

Make an initial floorplan and synthesize power rails. We are setting up the power rails in M8 and M9, the top two metal layers in the design. At this point, you can see a visualization of the estimated voltage drops on the power rails, as shown in (Figure 7). Note that we are assuming a 100mW power consumption for our GCD unit. Mouse over the different colored rails to see the voltage droops predicted by ICC at different parts of our chip (specified in mV).

```
icc_shell> create_floorplan \
  -core_utilization 0.6  -flip_first_row  -start_first_row \
  -left_io2core 5 -bottom_io2core 5 -right_io2core 5 -top_io2core 5
```

```
icc_shell> create_fp_placement
icc_shell> set_fp_rail_constraints -add_layer -layer M9 \
   -direction horizontal -min_strap 4
icc_shell> set_fp_rail_constraints -add_layer -layer M8 \
   -direction vertical -min_strap 4
icc_shell> synthesize_fp_rail \
  -power_budget "100" -voltage_supply "1.05" -target_voltage_drop "100" \
  -output_dir "./pna_output" -nets "VDD VSS" -create_virtual_rails "M1" \
  -synthesize_power_plan -synthesize_power_pads -use_strap_ends_as_pads
```

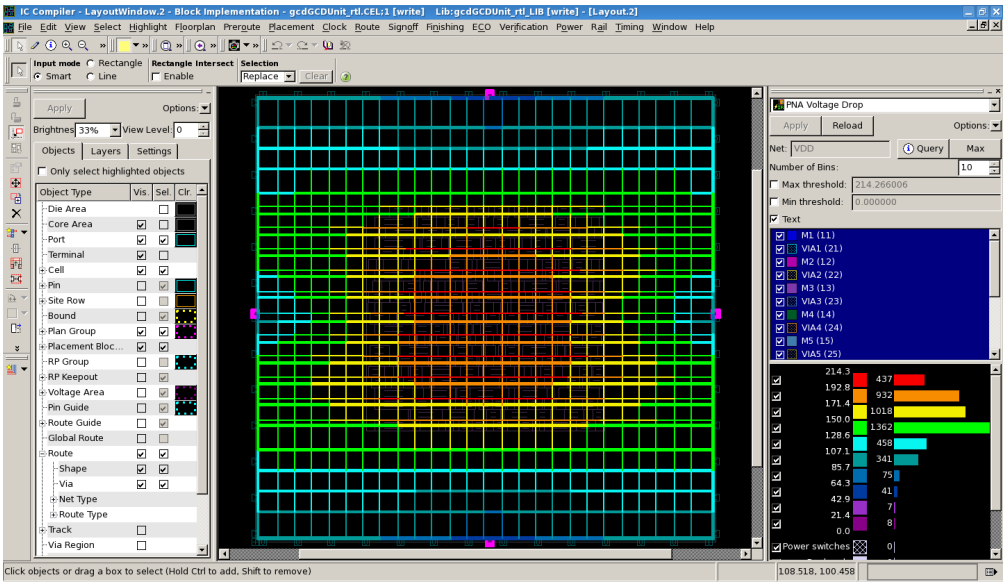

Figure 7: Estimated voltage drops shown in IC Compiler

In a new design, you might have to make changes to your power network based on this information. Here, everything should be reasonable, so go ahead and commit the power plan.

```
icc_shell> commit_fp_rail
```

You will now perform clock tree synthesis. Because the clock is one of the most critical and highest fan-out nets in the design, it must be routed first. The tool will analyze the clock net and insert buffers to minimize clock skew before performing routing.

```
icc_shell> clock_opt -only_cts -no_clock_route
icc_shell> route_zrt_group -all_clock_nets -reuse_existing_global_route true
```

Take a look at the generated clock tree. Choose *Clock* > *Color By Clock Trees*. Hit *Reload*, and then hit *OK* on the popup window. Now you will be able to see the synthesized clock tree (Figure 8).

Go ahead and route the remaining nets.

```
icc_shell> route_opt -initial_route_only
icc_shell> route_opt -skip_initial_route -effort low
```

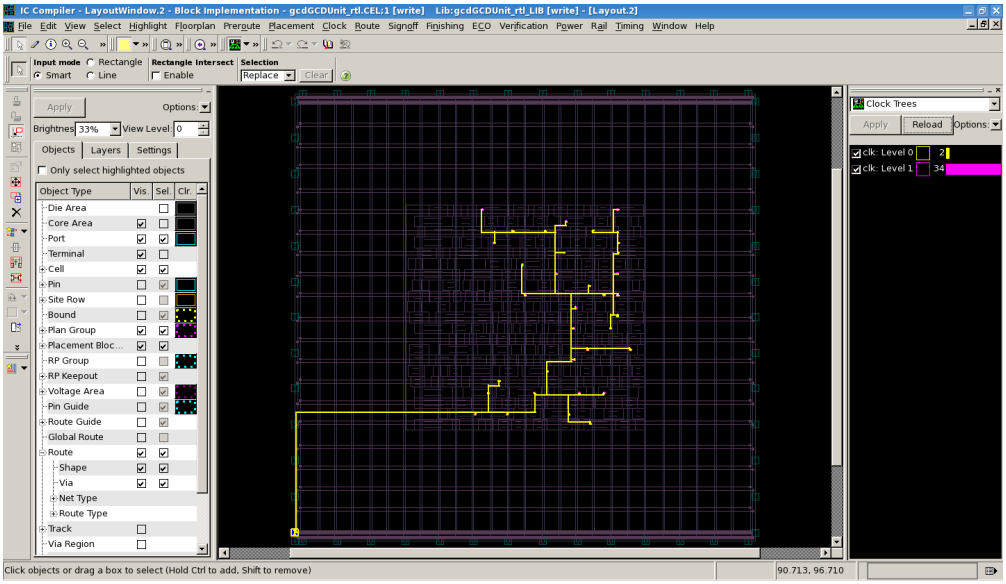

Figure 8: Synthesized clock tree shown in IC Compiler

Figure 9 shows the routed signals. The Synopsys 32nm process provides nine metal layers (metal 1 is mostly used by the standard cell layout itself) for routing your signals, although we used the top two layers for the power grid.

Notice that there are various gaps in the placement. You will add *filler* cells to fill up these spaces. Filler cells are just empty standard cells which connect the power and ground rails, but are necessary for the chip to be manufactured correctly. Notice that you need to rerun routing due to some Design Rule Check (DRC) errors.

```
icc_shell> insert_stdcell_filler \
  -cell_with_metal "SHFILL1_RVT SHFILL2_RVT SHFILL3_RVT" \
  -connect_to_power "VDD" -connect_to_ground "VSS"
icc_shell> route_opt -incremental -size_only
```

Congratulations! Now you have your design mapped onto silicon. Go ahead and generate the post place and route netlist and the constraint file. You also need to generate parasitics files necessary to estimate the power consumption of your design.

```
icc_shell> change_names -rules verilog -hierarchy
icc_shell> write_verilog "gcdGCDUnit_rtl.output.v"
icc_shell> write_sdf "gcdGCDUnit_rtl.output.sdf"
icc_shell> write_sdc "gcdGCDUnit_rtl.output.sdc"
icc_shell> extract_rc -coupling_cap
icc_shell> write_parasitics -format SBPF -output "gcdGCDUnit_rtl.output.sbpf"
icc_shell> source ./find_regs.tcl
icc_shell> find_regs gcdTestHarness_rtl/gcd
icc_shell> save_mw_cel
icc_shell> close_mw_cel
```

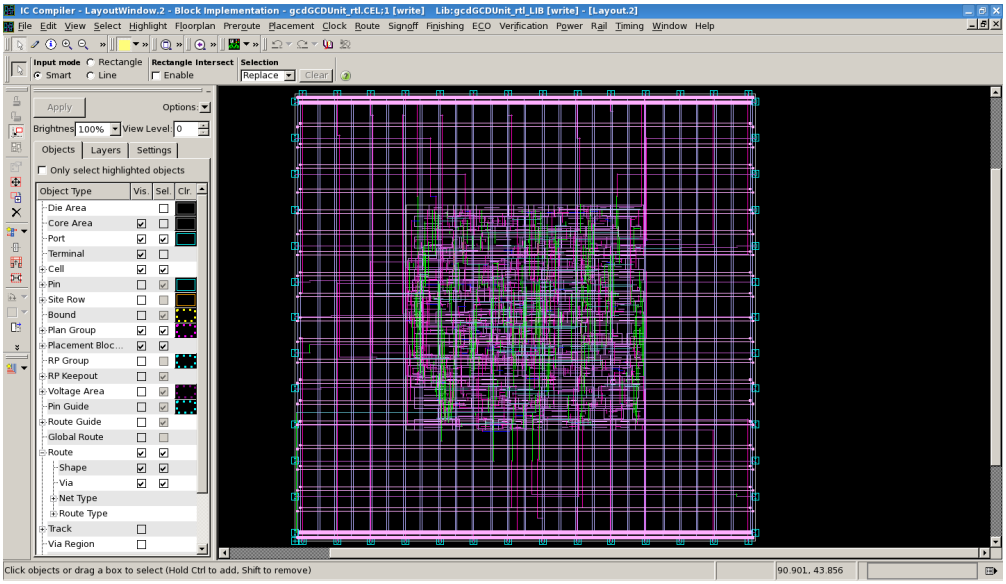

Figure 9: Routed signals shown in IC Compiler

This process can be automated using Makefiles. Notice that the Makefile creates new build directories like the one in Design Compiler. However, for the ICC flow steps, the Makefile creates two different directories. `current-icc` points to the completed design, while `current-iccdp` saves only the initial "design planning" steps. This allows you to re-run several place-and-route jobs off of the same initial floorplan to save time.

```
% cd $LABROOT/build/icc-par
% make
% ls -l
-rw-r--r-- 1 cs250-ab cs250 15232 Aug 28 18:40 Makefile
drwxr-xr-x 6 cs250-ab cs250  4096 Aug 29 23:42 build-icc-2010-08-29_23-41
drwxr-xr-x 8 cs250-ab cs250  4096 Aug 29 23:41 build-iccdp-2010-08-29_23-41
lrwxrwxrwx 1 cs250-ab cs250    26 Aug 29 23:41 current-icc -> build-icc-2010-...
lrwxrwxrwx 1 cs250-ab cs250    28 Aug 29 23:41 current-iccdp -> build-iccdp-...
drwxr-xr-x 2 cs250-ab cs250  4096 Aug 29 23:19 rm_icc_dp_scripts
drwxr-xr-x 2 cs250-ab cs250  4096 Aug 29 23:29 rm_icc_scripts
drwxr-xr-x 2 cs250-ab cs250  4096 Aug 29 23:38 rm_icc_zrt_scripts
drwxr-xr-x 2 cs250-ab cs250  4096 Aug 28 16:20 rm_notes
drwxr-xr-x 2 cs250-ab cs250  4096 Aug 29 23:34 rm_setup
```

Place-and-route is the most resource-intensive step of the VLSI flow. While it runs quickly for a small design like GCD, place-and-route can take many hours to complete on a large design.

Synopsys VCS: Simulating Post Place and Route Gate-Level Netlist

After you obtain the post place and route gate-level netlist, you will double-check the netlist by running a simulation using VCS. You also need to produce switching activity information in several

formats for power estimation. Use `vpd2vcd` and `vcd2saif` to convert a `vpd` file into a `vcd` and an `saif` file.

```
% cd $LABROOT/build-unscripted/icc-par
% sdfcorrect.py gcdGCDUnit_rtl.output.sdf gcdGCDUnit_rtl.output.corrected.sdf
% cd $LABROOT/build-unscripted/vcs-sim-gl-par
% vcs -full64 -PP +lint=all +v2k -timescale=1ns/10ps \
  -P ../icc-par/access.tab -debug \
  +neg_tchk +sdfverbose \
  -sdf typ:gcdGCDUnit_rtl:../icc-par/gcdGCDUnit_rtl.output.corrected.sdf \
  +incdir+$UCB_VLSI_HOME/install/vclib \
  -v $UCB_VLSI_HOME/install/vclib/vcQueues.v \
  -v $UCB_VLSI_HOME/install/vclib/vcStateElements.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTest.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTestSource.v \
  -v $UCB_VLSI_HOME/install/vclib/vcTestSink.v \
  -v $UCB_VLSI_HOME/stdcells/synopsys-32nm/default/verilog/cells.v \
  ../icc-par/gcdGCDUnit_rtl.output.v \
  ../../src/gcdTestHarness_rtl.v \
  +define+CLOCK_PERIOD=1.0
% ./simv -ucli +verbose=1
ucli% source ../icc-par/force_regs.ucli
ucli% run
...
  + 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
...
% vpd2vcd vcdplus.vpd vcdplus.vcd
% vcd2saif -input vcdplus.vcd -output vcdplus.saif
```

You can use the provided Makefile to build the post synthesis gate-level netlist simulator, run, and convert the switching activity file into a `vcd` and a `saif` format.

```
% cd $LABROOT/build/vcs-sim-gl-par
% make
% make run
% make convert
```

Synopsys PrimeTime PX: Estimating Power

PrimeTime PX is an add-on feature to PrimeTime that analyzes power dissipation of a cell-based design. While Design Compile and ICC both report power numbers, Primetime PX produces the most accurate numbers because it takes into account both the parasitics resulting from the physical design and the expected switching activity of each node. PrimeTime PX supports two power analysis modes: averaged and time-based. Averaged mode estimates average power consumption based on toggle rates. Time-based mode estimates the peak power consumption as well as the averaged power using gate-level simulation activity.

```
% cd $LABROOT/build-unscripted/pt-pwr
% pt_shell -64bit
...
pt_shell> alias "pt_shell>" ""
```

Go ahead and execute some commands to setup your environment. Enable the power analysis mode.

```
pt_shell> set ucb_vlsi_home [getenv UCB_VLSI_HOME]
pt_shell> set stdcells_home \
  $ucb_vlsi_home/stdcells/synopsys-32nm/default
pt_shell> set search_path "$stdcells_home/db"
pt_shell> set target_library "cells.db"
pt_shell> set link_path "* $target_library"
pt_shell> set power_enable_analysis "true"
```

Load the post place and route gate-level netlist into PrimeTime PX.

```
pt_shell> read_verilog "../icc-par/gcdGCDUnit_rtl.output.v"
pt_shell> current_design "gcdGCDUnit_rtl"
pt_shell> link
```

First try averaged mode power analysis. This mode takes the `saif` format file as input. Load in the parasitics information produced by IC Compiler before you run `report_power`.

```
pt_shell> set power_analysis_mode "averaged"
pt_shell> read_saif "../vcs-sim-gl-par/vcdplus.saif" \
  -strip_path "gcdTestHarness_rtl/gcd"
pt_shell> report_switching_activity -list_not_annotated
pt_shell> read_parasitics -increment \
  -format sbpf "../icc-par/gcdGCDUnit_rtl.output.sbpf.max"
pt_shell> report_power -verbose -hierarchy
...
                                Switch   Int      Leak      Total
Hierarchy                       Power    Power    Power     Power    %
------------------------------------------------------------------------
gcdGCDUnit_rtl                   0.000 6.88e-03 7.86e-05  6.96e-03 100.0
  GCDctrl0 (gcdGCDUnitCtrl)      0.000 6.89e-08 1.54e-05  1.54e-05   0.2
  GCDdpath0 (gcdGCDUnitDpath_W16) 0.000 6.88e-03 6.27e-05  6.95e-03  99.8
...
```

This shows you the average power consumption of your design, broken down by module.

Now try time-based power analysis. This mode takes a `vcd` format file as input. Load the parasitics information and run `report_power`. The report will include the estimated peak power as well as average power. In some cases, the peak power can be much larger than the average power.

```
pt_shell> set power_analysis_mode "time_based"
pt_shell> read_vcd "../vcs-sim-gl-par/vcdplus.vcd" \
  -strip_path "gcdTestHarness_rtl/gcd"
pt_shell> report_switching_activity -list_not_annotated
pt_shell> report_power -verbose -hierarchy
...
                                Switch   Int      Leak      Total
Hierarchy                       Power    Power    Power     Power    %
------------------------------------------------------------------------------
gcdGCDUnit_rtl                   0.000 6.30e-03 7.75e-05  6.37e-03 100.0
  GCDctrl0 (gcdGCDUnitCtrl)      0.000 3.36e-07 1.51e-05  1.54e-05   0.2
  GCDdpath0 (gcdGCDUnitDpath_W16) 0.000 6.30e-03 6.19e-05  6.36e-03  99.7

                                  Peak     Peak          Glitch   X-tran
Hierarchy                         Power    Time          Power    Power
------------------------------------------------------------------------------
gcdGCDUnit_rtl                   7.15e-03   0.025-0.026    0.000 2.90e-07
  GCDctrl0 (gcdGCDUnitCtrl)      4.59e-05   0.025-0.026    0.000 2.90e-07
  GCDdpath0 (gcdGCDUnitDpath_W16) 7.10e-03   0.019-0.020    0.000    0.000
  ...
```

Makefiles can be used to automate this procedure.

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

Introduction to Chisel

Chisel is a hardware description language currently under development at UC Berkeley. In this section, your assignment is to re-implement GCD in Chisel, and then to push the Chisel generated Verilog representation of the design through the toolflow. So far, we have been working in the `lab1-verilog/` directory, but now we will start working in `lab1-chisel/`.

For our purposes, Chisel adds one extra step to the beginning of the traditional tool flow. The Chisel code is used to generate Verilog files, after which point the rest of the flow stays the same. One major difference arises when debugging a design: when doing so, we only modify the Chisel code and never edit the Chisel generated Verilog files directly. The Verilog serves purely as an intermediate representation of the design.

Next, we will describe the directory structure used for the Chisel based design. `src/` is where Chisel code should go, and `src/work.scala` is the top level file that instantiates the modules in your design. We have provided an incomplete implementation of GCD in the `src/gcd.scala` file which you should fill in with your Chisel code. The interface between the module and the test harness is provided, so you only need to implement body of the component. If you like, you can use the Verilog implementation as a guide and do an almost direct translation, or you can experiment with some of Chisel's language features to make the implementation more clear and concise. Typing `make vlsi` in the top level project directory will invoke the Chisel compiler to produce a Verilog representation of your design and place the resulting file in the `vlsi/generated-src` directory.

To push this code through the VLSI toolflow, go to `vlsi/build`, which is essentially the same as `lab1-verilog/build` and push the design through the flow by following the same steps outlined earlier in the lab.

Using the Chisel tool flow

Before starting, make sure you have the most recent version of the lab template.

```
% cd /scratch/cs250-ab/lab1-chisel/
% git pull template master
```

Figure 10 shows each directory in the lab1-chisel hierarchy and includes comments about what each one contains.

lab1-chisel/

- Makefile — "make verilog" takes .scala from src/ and generates vlsi/generated-src/*.v
- .gitignore — Tells Git to ignore generated-src/, dynamically generated files in sbt/
- sbt/ — "Magic" directory for chisel
- src/ — Chisel code (*.scala). top.scala is a mandatory top level file that instantiates design
- emulator/ — Chisel target: C emulator
  - Makefile — Automate emulator testing
  - generated-src/ — C code generated by Chisel
  - testbench/ — Testbench that drives Chisel C emulator
- vlsi/ — Chisel target: VLSI
  - build/ — VLSI toolflow for vlsi/generated-src
    - Makefile — Controls all pieces of toolflow. Eg. "make dc-syn" will synthesize
    - vcs-sim-rtl/ — Simulate RTL in generated-src/
    - dc-syn/ — Synthesize RTL in generated-src/
    - vcs-sim-gl-syn/ — Simulate synthesized netlist in dc-syn/current-dc
    - icc-par/ — Place and route synthesized netlist from dc-syn/current-dc
    - vcs-sim-gl-par/ — Simulate place and routed netlist in icc-par/current-icc
    - pt-pwr/ — Power analysis of design in icc-par/current-icc
  - generated-src/ — Verilog code generated by Chisel (*.v)
  - Makefile — Does nothing by default. Will be used later for design space exploration

Notation:
blue means that these files generated dynamically, and are not stored in the repository

Figure 10: Directory organization for lab1-chisel/

Change your `$LABROOT` environment variable to point to the top level directory of the Chisel version of this design.

```
% cd /scratch/cs250-ab/lab1-chisel/
% LABROOT=$PWD
% cd src
% vim work.scala
```

You are now looking at the top level Chisel file. This is just boilerplate code that instantiates the top level module in your design. This is also where the interfaces to testbenches are defined. Next, open the code describing the GCD module that is being instantiated in `top.scala`

```
% vim gcd.scala
```

This is where you need to fill in the design with some of your own Chisel code. When you are ready to test your code, there are two methods from which to choose. First, the Chisel compiler can produce C++ code which implements a cycle-accurate simulation of your design. To generate the C++ code, compile the simulator, and run the testbench, run the following commands:

```
% cd $LABROOT
% make emulator
% cd emulator
% make run
```

In addition to a C++ description of a simulator, the Chisel compiler can also generate Verilog code that can be used as input to an ASIC flow like the one you experimented with earlier in this lab. As long as the components in Chisel have the same names as their counterparts in the Verilog implementation (which is true in this case because we supplied the top-level interface to you with the same naming scheme) you can re-use the same Verilog testbench for your Chisel-generated Verilog code. Once your design passes the tests in simulation, go ahead and push it through the rest of the steps in the flow.

```
% cd $LABROOT
% make vlsi
% cd vlsi/build/vcs-sim-rtl
% make run
```

Debugging

To debug your Chisel design, you can use either the C++ simulator, or simulate the generated Verilog files using VCS (in the `vcs-sim-rtl` directory).

To debug using the C++ simulator, you can add printf statements to `testbench/GCD-emulator.cpp`. All internal signals are renamed as `modulename__oldsignalname` unless there are duplicates (e.g., `GCD_ctrl__io_result_ready`, and the top level input/outputs are always accessible. You can look in `generated-src/GCD.h` to find the new signal names. Another technique is to use GDB, a traditional debugger for C/C++ programs.

Here is an example of how you might use GDB to debug your Chisel design:

```
% cd $LABROOT/emulator
% vim generated-src/GCD.cpp
(find the line number of the last line in the clock_lo function)
% gdb GCD-emulator
gdb% break GCD.cpp:linenumYouFound
gdb% run
gdb% p/x GCD__signalname
gdb% p/x GCD__signalname_shadow (the value signal will have during the next cycle)
gdb% c (go to next cycle)
```

To debug using the Verilog, you can use the procedure described previously to open vcs simulation results with DVE. Notice that the design will contain many signals with the `T_` prefix, which hold intermediate values produced by the Chisel compiler.

```
% cd $LABROOT
% cd vlsi/build/vcs-sim-rtl
% make run
% dve -full64 -vpd vcdplus.vpd &
```

Questions

Your writeup should not exceed two pages in length. Make your writing as crisp as you can!

Q1. W=16 vs. W=32 (For Verilog implementation only)

Throughout the lab, you assumed that the operands A, B, and the result were 16 bits wide. Now we'd like to build a GCD unit which operates on 32 bit values and produces a 32 bit result.

- What changes would you need to make to the `gcdGCDUnit_rtl` module?
- Does the Verilog provided test harness `gcdTestHarness_rtl` work with the 32-bit version of GCD? If not, what changes do you need to make in order to test our new 32-bit `gcdGCDUnit_rtl` module? Demonstrate that your 32-bit `gcdGCDUnit_rtl` functions correctly.
- How does this change effect the area/power/performance properties of the chip? Fill in the boxes in the following table by finding the values in the reports generated by the tools. An optional (but highly recommended) way to collect this data is to write a Python script to go through all the reports and extract the data you need. Use the output of your script to fill in the boxes in the table. Running a script to gather the data should look something like this:

```
% cd $LABROOT
% ./collect_data.py
```

	Unit	`gcdGCDUnit_rtl (W=16)`	`gcdGCDUnit_rtl (W=32)`
Post Synthesis Critical Path Length	ns		
Post Synthesis Area	um^2		
Post Synthesis Power	mW		
Post Place+Route Critical Path Length	ns		
Post Place+Route Area	um^2		
Post Place+Route Power	mW		
Average Power (max)	mW		
Peak Power (max)	mW		
Total Cell Count (exclude `SHFILL*` Cell)			
`DFF*` Cell Count			

Q2. Chisel vs. Verilog

Tell us how the results of pushing your Chisel generated Verilog version of GCD differ from the results produced when building the Verilog RTL version of GCD we provided. Be sure to describe the differences between key metrics such as performance, energy, and area. If there are differences, can you determine the cause?

Read me before you commit!

- Commiting is not enough for us to grade this lab, you will also need to push your changes to github with the following command: `git push origin master`
- If you are using one or more late days for this lab, please make a note of it in your writeup. If you do not, your TA will assume that whatever was committed at the deadline represents your submission for the lab, and any later commits will be disregarded.
- Please note in your writeup if you discussed or received help with the lab from others in the course. This will not affect your grade, but is useful in the interest of full disclosure.
- Please note in your writeup (roughly) how many hours you spent on this lab in total.
- As you may have learned in this lab, makefiles for Design Compiler, IC Compiler, and PrimeTime PX make a new build directory for every new run. Don't submit multiple build directories. Make sure that you only turn in the most recent build result.
- We have included several `.gitignore` files in the repository to keep the repo clean and to avoid accidental commits of peripheral or generated files. Normally, we do not want to commit the results of builds, only the source files needed to generate those builds. In this case, you need to commit certain build results that are targeted by a `.gitignore` file. To add an ignored file or directory to a git repo, use `git add -f`.
- You don't need to submit results for the VCS simulation. We will build and run simulation based on your committed source code or gate-level netlist.
- When you commit a symbolic link in Git, it stores the link as a file, and if the link is to a folder, it does not commit that folder. For example, committing the `current-dc` symlink will not commit your results. You also need to commit the build directory referenced by the symlink.
- You don't need to submit build results that you generated "by hand" in the `build-unscripted` directory.
- To summarize, your Git tree for lab1 should look like the following (use the Github web browser to check that everything is there):

```
/cs250-ab
  /lab1-verilog
    /src: no change
    /build:
      /dc-syn: COMMIT THE MOST RECENT BUILD VERSION
      /icc-par: COMMIT THE MOST RECENT BUILD VERSION
      /pt-pwr: COMMIT THE MOST RECENT BUILD VERSION
      /vcs-sim-behav: original files only
      /vcs-sim-gl-par: original files only
      /vcs-sim-gl-syn: original files only
      /vcs-sim-rtl: original files only
    /build-unscripted: original files only
  /lab1-chisel
    /src: COMMIT CHISEL CODE
    /vlsi:
      /build:
        /dc-syn: COMMIT THE MOST RECENT BUILD VERSION
        /icc-par: COMMIT THE MOST RECENT BUILD VERSION
```

```
        /pt-pwr: COMMIT THE MOST RECENT BUILD VERSION
        /vcs-sim-behav: original files only
        /vcs-sim-gl-par: original files only
        /vcs-sim-gl-syn: original files only
        /vcs-sim-rtl: original files only
    /writeup: COMMIT REPORT
    COMMIT PYTHON SCRIPT (Python script is not required, but highly recommended)
```

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. Original contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Versions of this lab have been used in the following courses:

- CS250 VLSI Systems Design (2009-2013) - University of California at Berkeley
- CSE291 Manycore System Design (2009) - University of California at San Diego