GCD: VLSI's Hello World

CS250 Laboratory 1 (Version 092509a)
September 25, 2009
Yunsup Lee

For the first lab assignment, you will write an RTL model of a greatest common divisor (GCD) circuit and push it through all the VLSI tools you will use in the class. The deliverables for this lab are (a) your working Verilog RTL checked into a revision control system called Subversion (SVN), (b) build results and reports generated by VCS, DC Compiler, Formality, IC Compiler, PrimeTime PX checked into SVN, and (c) written answers to the questions given at the end of this document. The lab assignment is due at the start of class on Tuesday, September 8. You must submit your written answers electronically by adding a directory titled `writeup` to your lab project directory (`lab1/trunk/writeup`). Electronic submissions must be in plain text or PDF format. You are encouraged to discuss your design with others in the class, but you must turn in your own work.

For this assignment, you will become familiar with the VLSI tools you will use this semester, learn how a design "flows" through the toolchain, and practice Verilog coding.

Figure 1 shows 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 an RTL model 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 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. After place and route, you will generate and simulate the final gate-level netlist using VCS. Finally you will use this gate-level simulation as a final test for correctness and to generate transition counts for every net in the design. Synopsys PrimeTime PX (`pt_shell`) takes these transition counts as input and correlate them with the capacitance values in the final layout to produce estimated power measurements.

Each piece of the toolflow has its own build directory and its own makefile. Please consult the following tutorials for more information on using the various parts of the toolflow.

- *Tutorial 2: Bits and Pieces of CS250's Toolflow*
- *Tutorial 4: Simulation using Synopsys VCS*
- *Tutorial 5: RTL-to-Gates Synthesis using Synopsys Design Compiler*
- *Tutorial 6: Automatic Placement and Routing using Synopsys IC Compiler*
- *Tutorial 7: Power Analysis using Synopsys VCS and Synopsys PrimeTime PX*

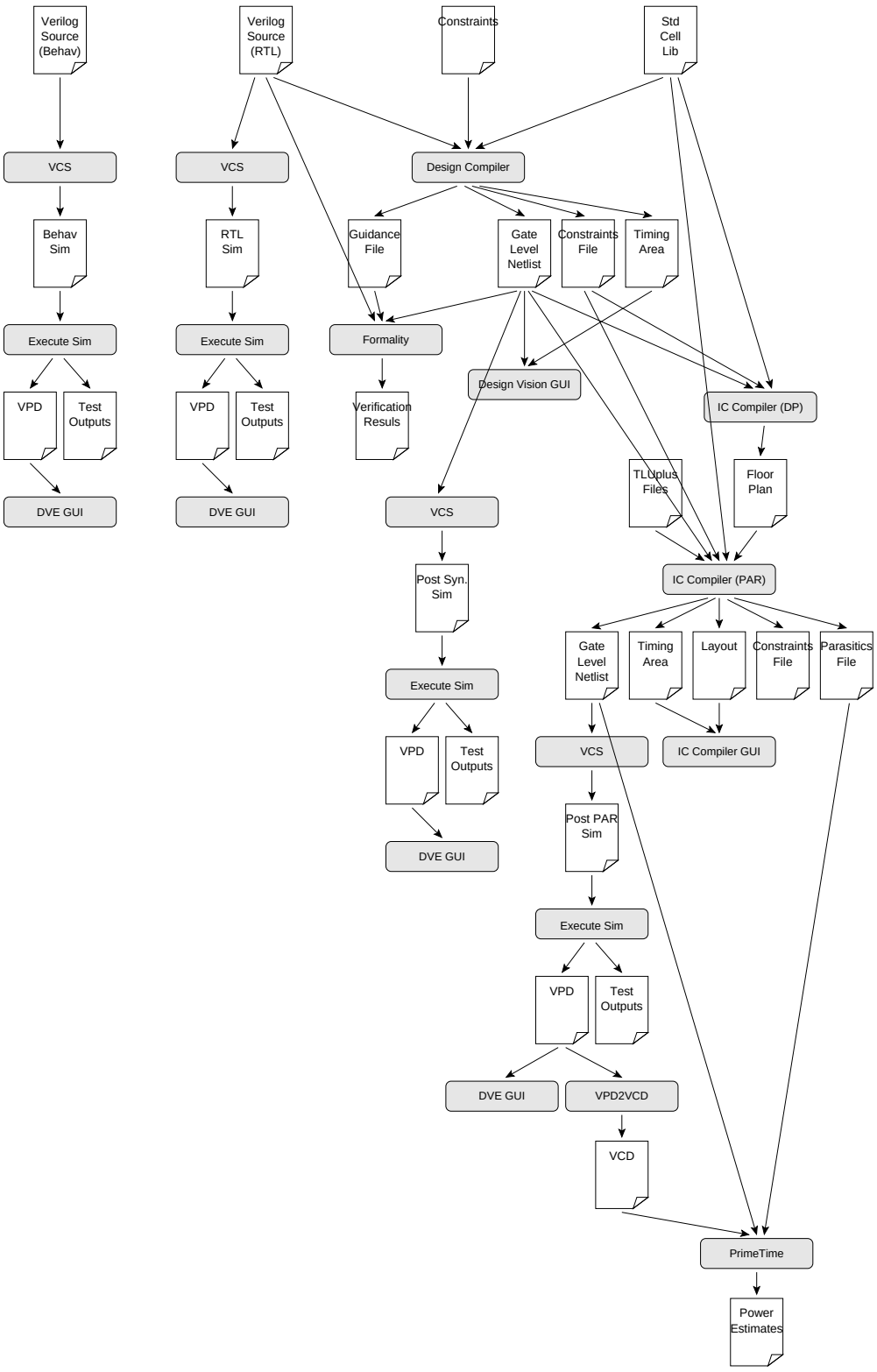

Figure 1: CS250 Toolflow for Lab 1

Block Diagram and Module Interface

The block diagram is shown in Figure 2. Your module should be named as `gcdGCDUnit` and must have 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.

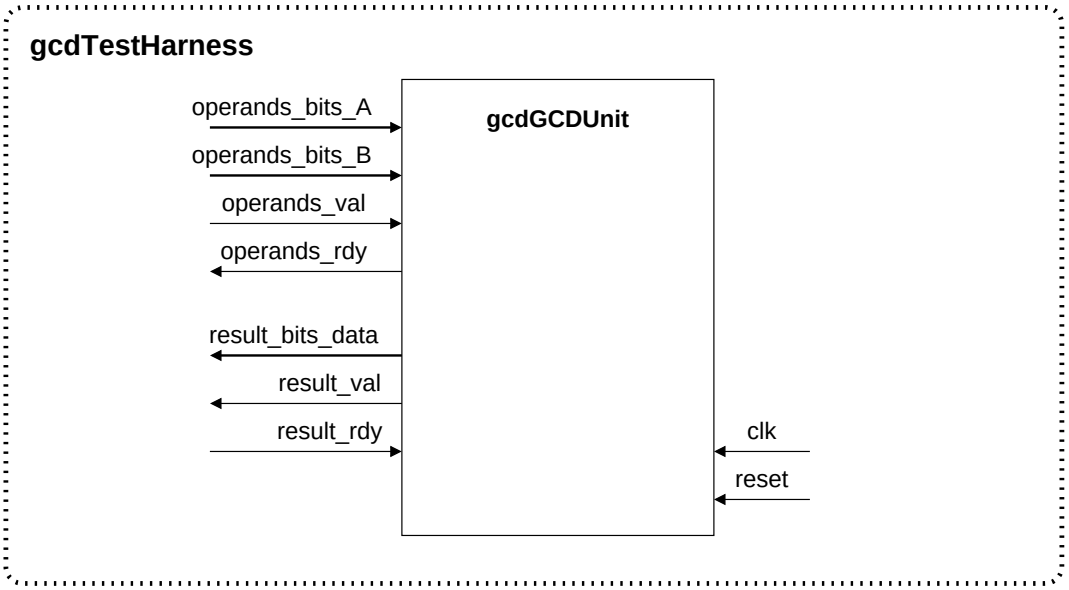

Figure 2: Block diagram for GCD Test Harness

```
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

Getting Started

All of the CS250 laboratory assignments should be completed on an EECS Instructional machine. Please see the course website for more information on the computing resources available for CS250 students. Once you have logged into an EECS Instructional you will need to setup the CS250 toolflow with the following commands.

```
% source ~cs250/tools/cs250.bashrc
```

You will be using SVN to manage your CS250 laboratory assignments. Please see *Tutorial 1: Using SVN to Manage Source RTL* for more information on how to use SVN. Every student has their own directory in the repository which is not accessible to other students. Assuming your username is `yunsup`, you can checkout your personal SVN directory using the following command.

```
% svn checkout $SVNREPO/yunsup vc
```

To begin the lab you will need to make use of the lab harness located in `~cs250/lab1`. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands copy the lab harness into your SVN directory and adds the new project to SVN.

```
% cd vc
% mkdir lab1
% svn add lab1
% cd lab1
% mkdir trunk branches tags
% cd trunk
% pwd
vc/lab1/trunk
% cp -R ~cs250/lab1/v-gcd/* .
% cd ..
% svn add *
% svn commit -m "Initial checkin"
% svn update
```

The resulting `lab1/trunk` project directory contains the following primary subdirectories: `src` contains your source Verilog; `build` contains automated makefiles and scripts for building your design; and `build.manual` is the directory to tryout the VLSI tools manually.

The `src` directory contains the Verilog test harness and other Verilog modules you will need in this lab assignment. The files marked with `(empty)` are the files you need to fill in.

- `gcdGCDUnit_behav.v` - Behavioral implementation of gcdGCDUnit
- `gcdGCDUnit_rtl.v (empty)` - RTL implementation of gcdGCDUnit
- `gcdGCDUnitCtrl.v (empty)` - Control part of the RTL implementation
- `gcdGCDUnitDpath.v (empty)` - Datapath part of the RTL implementation
- `gcdTestHarness_behav.v` - Test harness for the behavioral model
- `gcdTestHarness_rtl.v` - Test harness for the RTL model

The `build` and `build.manual` directory contains the following subdirectories which you will use when building your chip.

- `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 includes its own makefile and additional script files. Once you have all the tools working you can use the toplevel makefile in the `build` directory to run multiple tools at once. For example, once all the scripts are properly setup you should be able to use the following command to synthesize, floorplan, and place and route your design.

```
% pwd
vc/lab1/trunk/build
% make icc-par
```

Pushing the design through all the VLSI Tools

Let's assume you are finished with your RTL implementation adding stuff to the top module which is in `gcdGCDUnit_rtl.v`, control logic to `gcdGCDUnitCtrl.v`, and the datapath to `gcdGCDUnitDpath.v`. You will begin by running several commands manually before learning how you can automate the tools with scripts. You will try out the manual build in the `build.manual` directory and the automated build using makefiles and scripts in the `build` directory.

Synopsys VCS: Simulating your Verilog

VCS compiles source Verilog into a cycle-accurate executable. VCS can compile both Verilog expressed in behavioral models and RTL models. In behavioral models, logic is expressed as higher level behaviors. In RTL models, logic is expressed at register level. Verilog written in behavioral models might not be synthesizable. However, behavioral models can be useful when expressing its functionality, or when expressing a block that you are not interested in synthesizing. The test harness itself is a good example which is written in behavioral Verilog. You will start with simulating the GCD module written in behavioral model.

```
% pwd
vc/lab1/trunk/build.manual
% cd vcs-sim-behav
% vcs -PP +lint=all +v2k -timescale=1ns/10ps \
  ../../src/gcdGCDUnit_behav.v \
  ../../src/gcdTestHarness_behav.v
```

By default, VCS generates a simulator named `simv`. The `-PP` command line argument turns on support for using the `VPD` trace output format. The `+lint=all` argument turns on Verilog warnings.

Since it is relatively easy to write legal Verilog code which is probably functionally incorrect, you will always want to use this argument. For example, VCS will warn you if you connect nets with different bitwidths or forget to wire up a port. Always try to eliminate all VCS compilation errors *and* warnings. Since you will be making use of various Verilog-2001 language features, you need to set the `+v2k` command line option so that VCS will correctly handle these new constructs. 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. You use the `-v` flag 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 ANSI C code which is then compiled using `gcc`. When VCS is finished you should see a `simv` executable in the build directory. Now run the simulator.

```
% ./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 you will use makefiles to help automate the process of building our simulators.

```
% pwd
vc/lab1/trunk/build
% cd vcs-sim-behav
% cat Makefile
...
vclibsrcs = \
        $(vclibdir)/vcQueues.v \
        $(vclibdir)/vcStateElements.v \
        $(vclibdir)/vcMuxes.v \
        $(vclibdir)/vcArith.v \
        $(vclibdir)/vcTest.v \
        $(vclibdir)/vcTestSource.v \
        $(vclibdir)/vcTestSink.v \
...
vsrcs = \
        $(srcdir)/gcdGCDUnit_behav.v \
        $(srcdir)/gcdTestHarness_behav.v \
...
```

```
% make
% make run
```

You can leverage the same makefile to build the simulator for the Verilog written in RTL model.

```
% pwd
vc/lab1/trunk/build
% cd 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 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.

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

Choose *File > Open Database* and pick `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*.

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

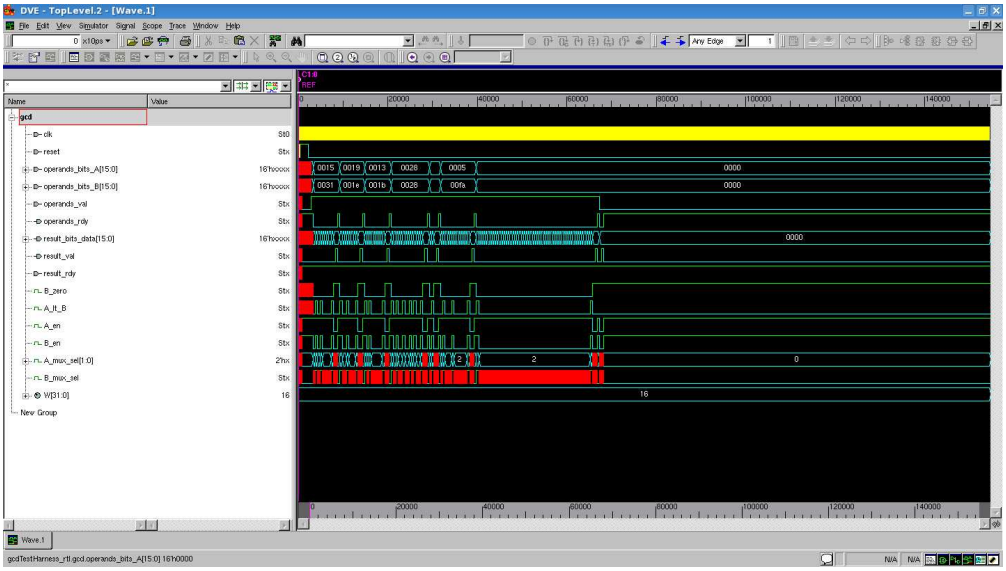

Figure 4: DVE Waveform Window

gate-level netlist is a completely structural description with only standard cells at the leaves of the design.

```
% pwd
vc/lab1/trunk/build.manual
% cd dc-syn
% dc_shell-xg-t
...
Initializing...
dc_shell>
```

You will now execute some commands to setup your environment.

```
dc_shell> set_app_var search_path \
  "~cs250/stdcells/synopsys-90nm/default/db/cells \
   ~cs250/install/vclib \
   ../../src"
dc_shell> set_app_var target_library "cells.db"
dc_shell> set_app_var synthetic_library "dw_foundation.sldb"
dc_shell> set_app_var link_library "* $target_library $synthetic_library"
dc_shell> set_app_var alib_library_analysis_path \
  "/home/ff/cs250/stdcells/synopsys-90nm/default/alib"
dc_shell> define_design_lib WORK -path "./work"
dc_shell> 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`, and `link` commands.

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

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 tool that the pin named `clk` is the clock and that your desired clock period is 1 nanoseconds.

```
dc_shell> 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> compile_ultra -no_autoungroup
...
  Beginning Delay Optimization Phase
  ----------------------------------

   ELAPSED            WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:00:26    6042.9      0.05       0.8       0.0
    0:00:28    6504.7      0.02       0.1       0.0
    0:00:30    6504.7      0.02       0.1       0.0
    0:00:31    6625.6      0.02       0.2       0.0 dpath/A_pf/q_np_reg[14]/D
    0:00:33    6655.7      0.00       0.0       0.0
    0:00:33    6655.7      0.00       0.0       0.0
    0:00:33    6655.7      0.00       0.0       0.0
    0:00:33    6655.7      0.00       0.0       0.0
    0:00:33    6606.7      0.16       4.2       0.0
    0:00:34    6067.8      0.08       1.1       0.0
    0:00:34    6067.8      0.08       1.1       0.0
    0:00:38    6458.3      0.00       0.0       0.0
    0:00:38    6458.3      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 the `compile_ultra` shows the worst negative slack which indicates how much room there is between the critical path in your design and the 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 across all endpoints in the design.

Now you can stop writing the guidance information for formal verification and write the synthesized gate-level netlist and generated constraints as well.

```
dc_shell> set_svf -off
```

```
dc_shell> change_names -rules verilog -hierarchy
dc_shell> write -format ddc -hierarchy -output gcdGCDUnit_rtl.mapped.ddc
dc_shell> write -f verilog -hierarchy -output gcdGCDUnit_rtl.mapped.v
dc_shell> write_sdc -version 1.7 -nosplit gcdGCDUnit_rtl.mapped.sdc
```

Take a look at various reports on synthesis results.

```
dc_shell> 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
dpath/B_pf/q_np_reg_9_/CLK (DFFX1)                   0.00      0.00  0.00 r
dpath/B_pf/q_np_reg_9_/Q (DFFX1)                     0.09      0.26  0.26 f
dpath/B_pf/q_np_9_ (net)                   3                   0.00  0.26 f
dpath/B_pf/q_np_9_ (vcEDFF_pf_W16_1)                           0.00  0.26 f
dpath/n110 (net)                                               0.00  0.26 f
dpath/U104/QN (INVX4)                                0.04      0.03  0.29 r
dpath/n150 (net)                           3                   0.00  0.29 r
...
dpath/n3 (net)                             1                   0.00  0.51 r
dpath/U62/QN (NOR2X4)                                0.07      0.05  0.56 f
dpath/A_lt_B (net)                         4                   0.00  0.56 f
dpath/A_lt_B (gcdGCDUnitDpath_W16)                             0.00  0.56 f
A_lt_B (net)                                                   0.00  0.56 f
ctrl/A_lt_B (gcdGCDUnitCtrl)                                   0.00  0.56 f
ctrl/A_lt_B (net)                                              0.00  0.56 f
ctrl/U4/QN (NAND2X4)                                 0.05      0.04  0.60 r
ctrl/n1 (net)                              1                   0.00  0.60 r
ctrl/U3/QN (INVX8)                                   0.03      0.03  0.63 f
ctrl/A_mux_sel[0] (net)                    3                   0.00  0.63 f

A_mux_sel[0] (net)                                             0.00  0.63 f
dpath/A_mux_sel[0] (gcdGCDUnitDpath_W16)                       0.00  0.63 f
dpath/A_mux_sel[0] (net)                                       0.00  0.63 f
dpath/A_mux/sel[0] (vcMux3_W16)                                0.00  0.63 f
dpath/A_mux/sel[0] (net)                                       0.00  0.63 f
dpath/A_mux/U81/QN (NOR2X4)                          0.07      0.05  0.67 r
dpath/A_mux/n30 (net)                      1                   0.00  0.67 r
...
dpath/A_mux/n60 (net)                      1                   0.00  0.86 f
dpath/A_mux/U101/QN (NAND2X4)                        0.03      0.02  0.88 r
dpath/A_mux/out[8] (net)                   1                   0.00  0.88 r
dpath/A_mux/out[8] (vcMux3_W16)                                0.00  0.88 r
dpath/A_mux_out[8] (net)                                       0.00  0.88 r
dpath/A_pf/d_p[8] (vcEDFF_pf_W16_0)                            0.00  0.88 r
dpath/A_pf/d_p[8] (net)                                        0.00  0.88 r
```

```
dpath/A_pf/U10/QN (NAND2X2)                          0.03      0.03  0.90 f
dpath/A_pf/n83 (net)                        1                  0.00  0.90 f
dpath/A_pf/U8/QN (NAND2X2)                           0.03      0.02  0.93 r
dpath/A_pf/n41 (net)                        1                  0.00  0.93 r
dpath/A_pf/q_np_reg_8_/D (DFFX1)                     0.03      0.00  0.93 r
data arrival time                                                    0.93

clock ideal_clock1 (rise edge)                                      1.00  1.00
clock network delay (ideal)                                         0.00  1.00
dpath/A_pf/q_np_reg_8_/CLK (DFFX1)                                  0.00  1.00 r
library setup time                                                 -0.07  0.93
data required time                                                        0.93
--------------------------------------------------------------------------------
data required time                                                        0.93
data arrival time                                                        -0.93
--------------------------------------------------------------------------------
slack (MET)                                                               0.00
...
```

This report lists the *critical path* of the design. 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. You can see that the critical path starts at bit 9 of the operand B register in the datapath; goes through the comparator; to the control logic; through the operand A mux; and finally ends at bit 8 of operand A register in the datapath. The critical path takes a total of 0.93ns which is less than the 1ns clock period constraint.

```
dc_shell> report_area -nosplit -hierarchy
...
                   Global cell area      Local cell area
                   ------------------  --------------------
Hierarchical cell  Absolute   Percent  Combi-     Noncombi-
                   Total      Total    national   national   Design
-----------------  ---------  -------  ---------  ---------  -------------------
gcdGCDUnit_rtl     6040.3223    100.0     0.0000     0.0000  gcdGCDUnit_rtl
ctrl                377.1080      6.2   291.4000     0.0000  gcdGCDUnitCtrl
ctrl/state_pf        85.7080      1.4    35.9420    49.7660  vcRDFF_pf_2_0
dpath              5663.2163     93.8  2616.9607     0.0000  gcdGCDUnitDpath_W16
dpath/A_mux        1013.3156     16.8  1013.3156     0.0000  vcMux3_W16
dpath/A_pf          983.2397     16.3   578.6608   404.5789  vcEDFF_pf_W16_0
dpath/B_mux         178.7870      3.0   178.7870     0.0000  vcMux2_W16
dpath/B_pf          870.9218     14.4   408.2840   462.6379  vcEDFF_pf_W16_1
-----------------  ---------  -------  ---------  ---------  -------------------
Total                                   5123.3506   916.9829
...
```

This report tells you the post synthesis area results. The units are um^2. You can see that the datapath consumes 93.8% of the total chip area.

```
dc_shell> report_power -nosplit -hier
```

```
...
--------------------------------------------------------------------------------
                                     Switch   Int      Leak     Total
Hierarchy                            Power    Power    Power    Power    %
--------------------------------------------------------------------------------
gcdGCDUnit_rtl                          1.163    2.037 2.82e+07    3.228 100.0
  dpath (gcdGCDUnitDpath_W16)           1.082    1.935 2.60e+07    3.044  94.3
    B_pf (vcEDFF_pf_W16_1)              0.130    0.423 3.08e+06    0.556  17.2
    B_mux (vcMux2_W16)                1.49e-02 4.63e-02 7.71e+05 6.21e-02   1.9
    A_pf (vcEDFF_pf_W16_0)              0.178    0.459 4.53e+06    0.641  19.9
    A_mux (vcMux3_W16)                  0.163    0.212 6.04e+06    0.381  11.8
  ctrl (gcdGCDUnitCtrl)               8.04e-02    0.102 2.18e+06    0.184   5.7
    state_pf (vcRDFF_pf_2_0)          2.95e-02 4.84e-02 5.80e+05 7.85e-02   2.4
...
```

This report tells you about post synthesis power results. The dynamic power units are mW while the leakage power units are pW.

```
dc_shell> report_reference -nosplit -hierarchy
...
****************************************
Design: vcEDFF_pf_W16_0
****************************************
Reference          Library       Unit Area   Count    Total Area   Attributes
--------------------------------------------------------------------------------
AO21X1             saed90nm_typ    10.138000       1    10.138000
DFFX1              saed90nm_typ    24.882999      15   373.244991 n
DFFX2              saed90nm_typ    31.334000       1    31.334000 n
INVX0              saed90nm_typ     5.530000       3    16.590001
INVX2              saed90nm_typ     6.451000       3    19.353001
INVX4              saed90nm_typ     9.216000       9    82.943996
ISOLORX1           saed90nm_typ     7.387000       1     7.387000
NAND2X2            saed90nm_typ     8.798000      31   272.738010
NAND2X4            saed90nm_typ    14.501000       1    14.501000
NBUFFX16           saed90nm_typ    26.700001       2    53.400002
NOR2X0             saed90nm_typ     5.530000       1     5.530000
OR2X1              saed90nm_typ     7.373000       8    58.984001
OR2X2              saed90nm_typ     9.274000       4    37.096001
--------------------------------------------------------------------------------
Total 13 references                                    983.240003
...
```

This report lists the standard cells used in each module. The `vcEDFF` module is made out of 15 `DFFX1` cells, 1 `DFFX2` cell, 31 `NAND2X2` cells, etc. You can also see how much area it is consuming.

```
dc_shell> report_resources -nosplit -hierarchy
...
****************************************
Design : gcdGCDUnitDpath_W16
```

```
****************************************
Resource Report for this hierarchy in file ../../src/gcdGCDUnitDpath.v
===============================================================================
| Cell           | Module         | Parameters | Contained Operations       |
===============================================================================
| lt_x_80_1      | DW_cmp         | width=16   | lt_80                      |
| sub_x_81_1     | DW01_sub       | width=16   | sub_81                     |
===============================================================================
...
```

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.

You can use makefiles and scripts to help automate the process of synthesizing your design.

```
% pwd
vc/lab1/trunk/build
% cd dc-syn
% cat Makefile
...
vclibsrcs = \
        $(vclibdir)/vcStateElements.v \
        $(vclibdir)/vcMuxes.v \
...
vsrcs = \
        $(srcdir)/gcdGCDUnitCtrl.v \
        $(srcdir)/gcdGCDUnitDpath.v \
        $(srcdir)/gcdGCDUnit_rtl.v \
...
% make
```

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

```
% pwd
vc/lab1/trunk/build/dc-syn
% ls -l
drwxr--r-- 2 cs250 cs250 4096 2009-08-15 04:32 alib-52
drwxr-xr-x 6 cs250 cs250 4096 2009-08-26 14:25 build-dc-2009-08-26_14-25
-rw-r--r-- 1 cs250 cs250 1109 2009-08-15 11:25 constraints.tcl
lrwxrwxrwx 1 cs250 cs250   25 2009-08-26 14:25 current-dc -> build-dc-2009-08-26_14-25
drwxr--r-- 2 cs250 cs250 4096 2009-08-26 13:15 dc_scripts
drwxr--r-- 2 cs250 cs250 4096 2009-08-14 09:43 fm_scripts
-rw-r--r-- 1 cs250 cs250 3613 2009-08-26 01:45 Makefile
drwxr--r-- 2 cs250 cs250 4096 2009-08-14 15:27 scripts
% cd current-dc
% ls -l
```

```
-rw-r--r-- 1 cs250 cs250 165351 2009-08-26 14:25 command.log
-rw-r--r-- 1 cs250 cs250   2509 2009-08-26 14:25 common_setup.tcl
-rw-r--r-- 1 cs250 cs250   1109 2009-08-26 14:25 constraints.tcl
-rw-r--r-- 1 cs250 cs250   3757 2009-08-26 14:25 dc_setup.tcl
-rw-r--r-- 1 cs250 cs250   9877 2009-08-26 14:25 dc.tcl
drwxr-xr-x 2 cs250 cs250   4096 2009-08-26 14:25 log
-rw-r--r-- 1 cs250 cs250    534 2009-08-26 14:25 make_generated_vars.tcl
drwxr-xr-x 2 cs250 cs250   4096 2009-08-26 14:25 reports
drwxr-xr-x 2 cs250 cs250   4096 2009-08-26 14:25 results
-rw-r--r-- 1 cs250 cs250     29 2009-08-26 14:25 timestamp
drwxr-xr-x 2 cs250 cs250   8192 2009-08-26 14:25 work
```

Notice that the makefile does not overwrite build directories. It always create new build directories. This makes it easy to change your synthesis scripts or source Verilog, resynthesize your design, and compare your results to previous designs. 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 result. You should avoid using the GUI to actually perform synthesis since you want to use scripts for this. Now launch design vision.

```
% pwd
vc/lab1/trunk/build/dc-syn/current-dc
% design_vision-xg
...
Initializing...
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 5). If you right click on a module and choose *Schematic View* option, the tool will display a schematic of the synthesized logic corresponding to that module.

Synopsys Formality: Formal Verification

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

```
% pwd
vc/lab1/trunk/build.manual
% cd dc-syn
% fm_shell
...
fm_shell (setup)>
```

Execute some commands to setup your environment.

```
fm_shell (setup)> set_app_var search_path \
```

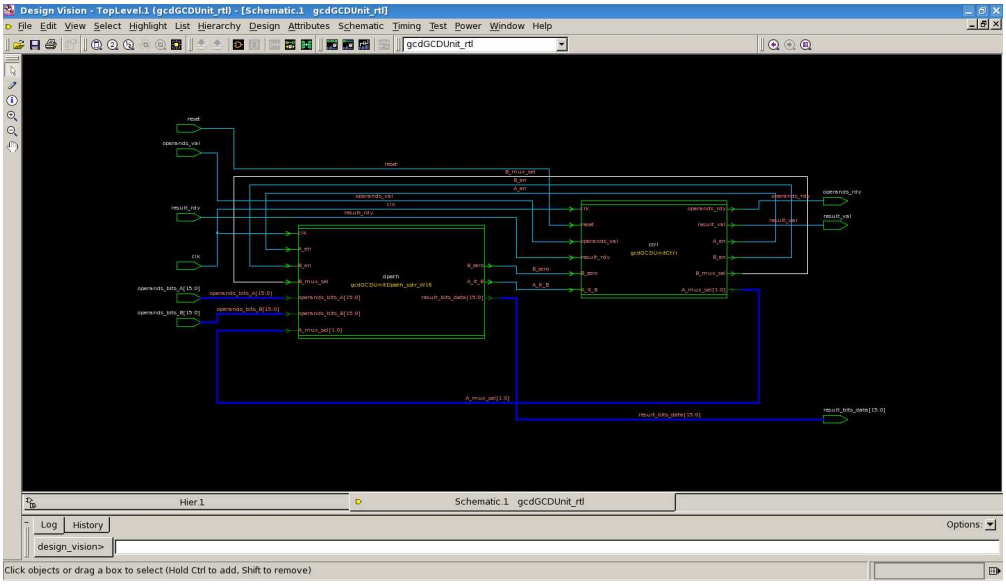

Figure 5: Design Vision Hierarchical View

```
  "~cs250/stdcells/synopsys-90nm/default/db/cells \
   ~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 point to the standard libraries you will be using for the class. Now you can load your original RTL design and the synthesized gate-level netlist to Formality.

```
fm_shell (setup)> read_verilog -r \
  "vcStateElements.v vcMuxes.v \
   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 verify whether or not both design match.

```
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 can also report the current status.

```
fm_shell (verify)> report_status
```

You can use makefiles and scripts to help automate the process of formal verification.

```
% pwd
vc/lab1/trunk/build
% cd dc-syn
% make fm
```

Synopsys VCS: Simulating Post Synthesis Gate-Level Netlist

After obtaining the synthesized gate-level netlist, you will double-check the netlist by running a simulation using VCS.

```
% pwd
vc/lab1/trunk/build.manual
% cd vcs-sim-gl-syn
% vcs -PP +lint=all +v2k -timescale=1ns/10ps \
  -y $UCB_VLSI_HOME/stdcells/synopsys-90nm/default/verilog/cells +libext+.v \
  +incdir+$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/vcMuxes.v \
  -v $UCB_VLSI_HOME/install/vclib/vcArith.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 \
  ../dc-syn/gcdGCDUnit_rtl.mapped.v \
  ../../src/gcdTestHarness_rtl.v
% ./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
...
```

You can use the makefile to build the post synthesis gate-level netlist simulator and run the simulator.

```
% pwd
vc/lab1/trunk/build
% cd vcs-sim-gl-syn
% make
% make run
```

Synopsys IC Compiler: Gate-Level Netlist to Layout

IC Compiler 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. At this point, you can see the instantiated standard cells and routed metals. You will use this tool more interactively, so go ahead and launch the IC Compiler with the GUI enabled.

```
% pwd
vc/lab1/trunk/build.manual
% cd icc-par
% icc_shell -gui
...
Initializing...
icc_shell>
```

Execute some commands to setup your environment.

```
icc_shell> set_app_var search_path \
  "~cs250/stdcells/synopsys-90nm/default/db/cells"
icc_shell> set_app_var target_library "cells.db"
icc_shell> set_app_var link_library "* $target_library"
```

These commands point to the standard libraries you will be using for the class. Before you jump into place and route, you will create your own Milkyway database, the place where you will be saving your place and routed design. Notice while you create the Milkyway database you hand in the technology file which has all the information about the process (e.g., detailed information of the poly and the metal layers), and the Milkyway reference database which captures the standard cell layout. Then you will read your synthesized gate-level netlist from the `dc-syn` directory. Also specify the tlu+ files which has the information you will use when extracting the parasitics of the layout. You will also make power and ground ports.

```
icc_shell> create_mw_lib \
  -tech "~cs250/stdcells/synopsys-90nm/default/techfile/techfile.tf" \
  -bus_naming_style {[%d]} \
  -mw_reference_library "~cs250/stdcells/synopsys-90nm/default/mw/cells/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 "~cs250/stdcells/synopsys-90nm/default/tluplus/cells/max.tluplus" \
  -min_tluplus "~cs250/stdcells/synopsys-90nm/default/tluplus/cells/min.tluplus" \
  -tech2itf_map "~cs250/stdcells/synopsys-90nm/default/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. At this point, you can see the estimated voltage drops on the power rails (Figure 6). The numbers on your right are specified in mW.

```
icc_shell> initialize_floorplan \
  -control_type "aspect_ratio" -core_aspect_ratio "1" \
  -core_utilization "0.7" -row_core_ratio "1" \
  -left_io2core "30" -bottom_io2core "30" -right_io2core "30" -top_io2core "30" \
  -start_first_row
icc_shell> create_fp_placement
icc_shell> synthesize_fp_rail \
  -power_budget "1000" -voltage_supply "1.2" -target_voltage_drop "250" \
  -output_dir "./pna_output" -nets "VDD VSS" -create_virtual_rails "M1" \
  -synthesize_power_plan -synthesize_power_pads -use_strap_ends_as_pads
```

If you have met your power budget, 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 highest fan-out nets, you should route this signal first. The tool will first analyze the clock net and insert buffers to minimize the skew before routing.

```
icc_shell> clock_opt -only_cts -no_clock_route
icc_shell> route_group -all_clock_nets -search_repair_loop "15"
```

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 7).

Go ahead and route the remaining nets.

```
icc_shell> route_opt -initial_route_only
icc_shell> route_opt -skip_initial_route -effort medium -power
```

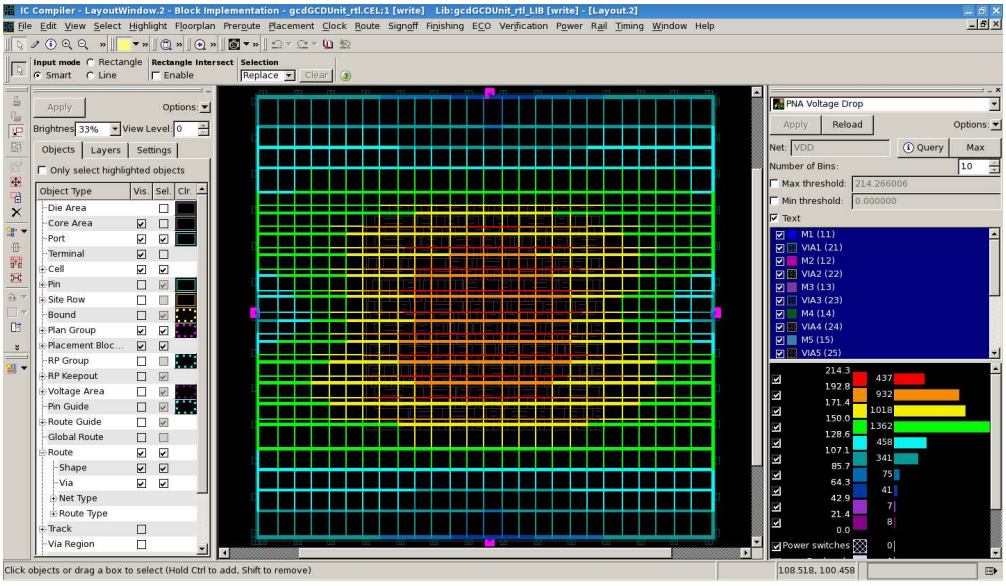

Figure 6: Estimated voltage drops shown in IC Compiler

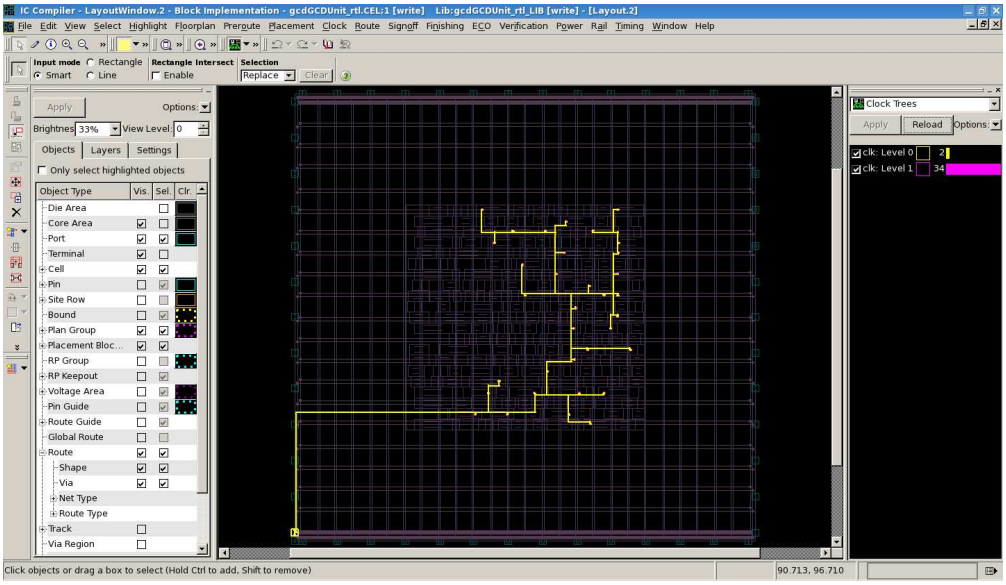

Figure 7: Synthesized clock tree shown in IC Compiler

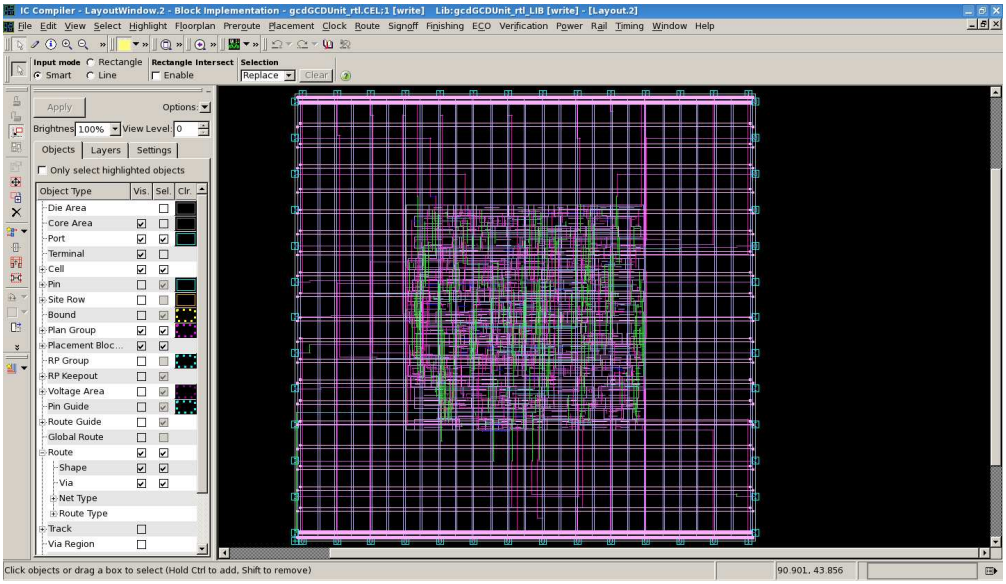

Figure 8: Routed signals shown in IC Compiler

Figure 8 shows the routed signals. Synopsys 90nm process provides nine metal layers (metal 1 is mostly used by the standard cell layout itself) to route your signals.

Notice that there are various holes 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. Notice that you need to rerun routing due to some Design Rule Check (DRC) errors.

```
icc_shell> insert_stdcell_filler \
  -cell_with_metal "SHFILL2" -connect_to_power "VDD" -connect_to_ground "VSS"
icc_shell> route_opt -incremental -size_only
icc_shell> route_search_repair -rerun_drc -loop "2"
```

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

```
icc_shell> change_names -rules verilog -hierarchy
icc_shell> write_verilog "gcdGCDUnit_rtl.output.v"
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> save_mw_cel
icc_shell> close_mw_cel
```

You can automate this process. Notice that the makefile creates new build directories like the one in Design Compiler.

```
% pwd
vc/lab1/trunk/build
```

```
% cd icc-par
% make
% ls -l
-rw-r--r--  1 cs250 cs250 13766 Aug 15 17:11 Makefile
drwxr-xr-x 10 cs250 cs250  4096 Aug 15 04:35 build-icc-2009-08-26_04-32
drwxr-xr-x 11 cs250 cs250  4096 Aug 24 15:07 build-icc-2009-08-26_11-26
drwxr-xr-x  8 cs250 cs250  4096 Aug 15 04:33 build-iccdp-2009-08-26_04-32
drwxr-xr-x  8 cs250 cs250  4096 Aug 24 14:13 build-iccdp-2009-08-26_11-26
lrwxrwxrwx  1 cs250 cs250    26 Aug 15 11:26 current-icc -> build-icc-2009-08-26_11-26
lrwxrwxrwx  1 cs250 cs250    28 Aug 15 11:26 current-iccdp -> build-iccdp-2009-08-26_11-26
drwxr-xr-x  2 cs250 cs250  4096 Aug 15 17:11 icc_scripts
drwxr-xr-x  2 cs250 cs250  4096 Aug 14 17:55 iccdp_scripts
drwxr-xr-x  2 cs250 cs250  4096 Aug 14 09:43 scripts
```

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 get the switching activities in several formats to estimate power. Use `vpd2vcd` and `vcd2saif` to convert a `vpd` format into a `vcd` and a `saif` format.

```
% pwd
vc/lab1/trunk/build.manual
% cd vcs-sim-gl-par
% vcs -PP +lint=all +v2k -timescale=1ns/10ps \
  -y $UCB_VLSI_HOME/stdcells/synopsys-90nm/default/verilog/cells +libext+.v \
  +incdir+$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/vcMuxes.v \
  -v $UCB_VLSI_HOME/install/vclib/vcArith.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 \
  ../icc-par/gcdGCDUnit_rtl.output.v \
  ../../src/gcdTestHarness_rtl.v
% ./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
...
```

```
% vpd2vcd vcdplus.vpd vcdplus.vcd
% vcd2saif -input vcdplus.vcd -output vcdplus.saif
```

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

```
% pwd
vc/lab1/trunk/build
% cd 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. PrimeTime PX supports two types of power analysis modes. They are averaged mode and time-based mode. Averaged mode calculates averaged power based on toggle rates. Time-based mode let's you know the peak power as well as the averaged power using gate-level simulation activity.

```
% pwd
vc/lab1/trunk/build.manual
% cd pt-pwr
% pt_shell
...
pt_shell>
```

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

```
pt_shell> set search_path \
  "~cs250/stdcells/synopsys-90nm/default/db/cells"
pt_shell> set target_library "cells.db"
pt_shell> set link_path "* $target_library"
pt_shell> set power_enable_analysis "true"
```

Read 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
```

Start with the averaged mode power analysis. This mode uses the `saif` format. Read the parasitics generated 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                  4.03e-05 1.43e-04 8.33e-06  1.92e-04 100.0
  dpath (gcdGCDUnitDpath_W16)   3.17e-05 1.32e-04 7.26e-06  1.71e-04  89.0
    A_mux (vcMux3_W16)          9.01e-06 1.27e-05 7.70e-07  2.25e-05  11.8
    B_mux (vcMux2_W16)          7.01e-07 3.68e-06 8.89e-07  5.27e-06   2.7
    B_pf (vcEDFF_pf_W16_1)      1.94e-06 3.76e-05 1.55e-06  4.11e-05  21.4
    A_pf (vcEDFF_pf_W16_0)      6.38e-06 4.61e-05 1.55e-06  5.41e-05  28.2
  ctrl (gcdGCDUnitCtrl)         8.56e-06 1.14e-05 1.07e-06  2.11e-05  11.0
    state_pf (vcRDFF_pf_2_0)    2.70e-07 4.85e-06 2.29e-07  5.35e-06   2.8
...
```

You can see the average power consumption by each module in your design.

Try the time-based power analysis. This mode takes the `vcd` format as an input. Read the parasitics and run `report_power`. You will see the estimated peak power as well as 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> 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                  2.84e-05 1.48e-04 8.09e-06  1.85e-04 100.0
  dpath (gcdGCDUnitDpath_W16)   2.17e-05 1.36e-04 7.06e-06  1.65e-04  89.3
    A_mux (vcMux3_W16)          5.08e-06 9.75e-06 7.76e-07  1.56e-05   8.4
    B_mux (vcMux2_W16)          4.93e-07 3.73e-06 8.87e-07  5.11e-06   2.8
    B_pf (vcEDFF_pf_W16_1)      1.37e-06 4.01e-05 1.55e-06  4.30e-05  23.3
    A_pf (vcEDFF_pf_W16_0)      4.01e-06 5.19e-05 1.55e-06  5.75e-05  31.1
  ctrl (gcdGCDUnitCtrl)         6.62e-06 1.21e-05 1.02e-06  1.98e-05  10.7
    state_pf (vcRDFF_pf_2_0)    2.26e-07 5.61e-06 2.29e-07  6.06e-06   3.3

                                Peak     Peak            Glitch   X-tran
Hierarchy                       Power    Time            Power    Power
------------------------------------------------------------------------------
gcdGCDUnit_rtl                  2.47e-02 785.000-785.001 1.47e-07 7.43e-07
  dpath (gcdGCDUnitDpath_W16)   2.39e-02 785.000-785.001 1.47e-07 6.28e-07
    A_mux (vcMux3_W16)          4.39e-03 165.373-165.374 1.47e-07 3.48e-07
```

```
    B_mux (vcMux2_W16)            1.72e-03 785.209-785.210    0.000    0.000
    B_pf (vcEDFF_pf_W16_1)        1.25e-02 785.000-785.001    0.000 1.13e-07
    A_pf (vcEDFF_pf_W16_0)        1.15e-02 1035.000-1035.001
                                                              0.000 1.67e-07
  ctrl (gcdGCDUnitCtrl)           7.17e-03 1325.254-1325.255
                                                              0.000 1.16e-07
    state_pf (vcRDFF_pf_2_0)      2.33e-03 1335.000-1335.001
                                                              0.000 1.17e-08
...
```

You can automate this process using makefiles.

```
% pwd
vc/lab1/trunk/build
% cd pt-pwr
% make
```

Lab Hints and Tips

Tip 1: Write clean synthesizable code

Place most of your logic in leaf modules and use structural Verilog to connect the leaf modules in a hierarchy. Make sure to separate out datapath and control circuitry.

Tip 2: Make Use of the Verilog Component Library

We have provided you with a very simple Verilog Component Library (VCLIB) which you may find useful for this lab. VCLIB includes simple mux, register, arithmetic, and memory modules. It is installed globally at `~cs250/install/vclib`. Examine the makefile included in the lab harness to see how to link in the library.

You would consider using `vcEDFF_pf`, `vcMux2`, and `vcMux3` in your datapath.

Tip 3: Write Semi-Behavioral Verilog

You might try to design optimized gate-level comparators and a subtractor like the following.

```
vcEQComparator#(16) comp
(
  .in0 (B),
  .in1 (const0),
  .out (B_zero)
);

vcLTComparator#(16) lt_comp
(
  .in0 (A),
  .in1 (B),
```

```
    .out (A_lt_B)
  );

  vcSubtractor#(16) sub
  (
    .in0 (A),
    .in1 (B),
    .out (sub_out)
  );
```

Consider using a more behavioral implementation of the functional units. The synthesize tools might pick up built in Verilog operators and use DesignWare components that might end up in a more efficient design.

```
assign B_zero  = ( B == 0 );
assign A_lt_B  = ( A < B );
assign sub_out = A - B;
```

Questions

Your writing should not exceed one page. Make your writings as crisp as you can!

Q1. gcdGCDUnit

Tell us how your `gcdGCDUnit` works in detail. You might include a diagram of your datapath and control logic.

Q2. W=16 vs. W=32

Throughout the lab, you assumed operand A, B, and the result to be 16 bits wide. We would like to make a GCD unit which takes 32 bits operands and produces a 32 bit result.

- What changes would be necessary for the `gcdGCDUnit_rtl` module?
- Does the Verilog test harness `gcdTestHarness_rtl` work? If not, what changes do you need to make in order to test our new 32-bit `gcdGCDUnit_rtl`? Show us that your 32-bit `gcdGCDUnit_rtl` works.
- How does this affect your chip? Fill in the following table and explain.

Q3. VLSI Tools

Tell us about your experience using the VLSI tools. Did you have any problems checking out a license for any of the tools? Any suggestions?

Read me before you commit!

- As you may have learned in this lab, makefiles for Design Compiler, IC Compiler, PrimeTime PX makes a new build directory for every new run. Don't submit all of them. Make sure that you only turn in the most recent build result.

	Unit	`gcdGCDUnit_rtl (W=16)`	`gcdGCDUnit_rtl (W=32)`
Post Synthesis Area	um^2		
Post Synthesis Power	mW		
Post Place+Route Area	um^2		
Average Power (min)	mW		
Average Power (max)	mW		
Peak Power (min)	mW		
Peak Power (max)	mW		
Total Cell Count (exclude `SHFILL2` Cell)			
`DFFX*` Cell Count			

- Design Compiler saves intermediate representations of the standard cell library to a directory called `alib-52` to speed things up. Please make sure you exclude the `alib-52` when you turn in the build results for synthesis.
- 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. If you have added Verilog sources codes or changed the name of the Verilog source codes, however, you **must** commit the makefile for every `vcs-sim-*` directory.
- You don't need to submit build results which are done manually.
- We will checkout the stuff you committed to the `lab1/trunk` and use that for lab grading. Feel free to take advantage of branches and tags. We will also checkout the version which was committed just before 12:30pm on Tuesday, September 8. Use your late days wisely!
- To summarize, your SVN tree for lab1 should look like the following:

```
/yunsup
  /lab1
    /trunk
      /src: COMMIT STUFF YOU HAVE HERE
      /build
        /dc-syn: COMMIT THE MOST RECENT VERSION YOU HAVE HERE
        /icc-par: COMMIT THE MOST RECENT VERSION YOU HAVE HERE
        /pt-pwr: COMMIT THE MOST RECENT VERSION YOU HAVE HERE
        /vcs-sim-behav: may not need to commit stuff here
        /vcs-sim-gl-par: may not need to commit stuff here
        /vcs-sim-gl-syn: may not need to commit stuff here
        /vcs-sim-rtl: may not need to commit stuff here
      /build.manual: don't need to commit stuff here
      /writeup: COMMIT STUFF YOU HAVE HERE
    /branches: feel free to use branches!
    /tags: feel free to use tags!
```

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

- CS250 VLSI Systems Design (2009) - University of California at Berkeley