RTL-to-Gates Synthesis using Synopsys Design Compiler

Contents

1   Introduction

In this tutorial you will gain experience using Synopsys Design Compiler (DC) to perform hardware synthesis. A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. The resulting gate-level netlist is a completely structural description with standard cells only at the leaves of the design. Figure 1 illustrates the basic DC toolflow and how it fits into the larger ECE5745 flow.

Internally, a synthesis tool performs many steps including high-level RTL optimizations, RTL to unoptimized boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells. Good RTL designers will familiarize themselves with the target standard cell library so that they can develop an intuition on how their RTL will be synthesized into gates.

In this tutorial you will use Synopsys Design Compiler to elaborate the RTL for our example greatest common divisor (GCD) cicruit, set optimization constraints, synthesize the design to gates, and prepare various area and timing reports. You will also learn how to read the various DC text reports and how to use the graphical Synopsys Design Vision tool to visualize the synthesized design.

Note that this tutorial is by no means comprehensive. Synopsys documentation is located on the public course webpage (`http://www.csl.cornell.edu/courses/ece5745/syndocs`) and can be accessed using the username and password distributed in lecture.

2   Getting The Tutorial Code

All of the ECE5745 tutorials should be run on the BRG compute servers **brg-01.ece.cornell.edu** and **brg-02.ece.cornell.edu**. Before proceeding further, please log into one of these machines.

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

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

Once you have logged into a BRG machine you will need to setup the ECE5745 toolflow with the following commands.

Figure 1: DC Toolflow

```
% source setup-ece5745.sh
```

For this tutorial you will be using a GCD circuit as your example RTL design. If you don't already have the source files from Tutorial 1, create an ece5745 folder in your home directory and clone the tutorial files from the git repository:

```
% mkdir ${HOME}/ece5745
% cd ${HOME}/ece5745
% git clone git@github.com:cornell-ece5745/ece5745-tut-asic.git
% cd ece5745-tut-asic/tutorial
% TUTROOT=$PWD
```

Before starting, take a look at the subdirectories in the project directory. Note that there are directories for your RTL source (`src`) and for generated content (`build`). The `build` directory has subdirectories for each major step in the ECE5745 toolflow, these subdirectories contain scripts and configuration files necessary for running the tools. For this tutorial you will work exclusively in the `dc-syn` subdirectory.

3 Important Files

Design Compiler is an extremely complicated tool that requires many pieces to work correctly. Attempts at synthesis without providing the tools with properly formatted configuration scripts, constraint information, and numerous technology files for the target standard cells will only be met with pain and sadness.

While the ECE5745 TAs have already done most of this setup work for you, it is important to have an understanding of all the components required in the synthesis process. Additionally, this will serve as a useful reference should you ever need to set up your own toolflow.

Below is a list of important library files needed by the synthesis tool. Note that these files are specific to the Synopsys 90nm Educational Library we are using for the course. When synthesizing to a different standard cell library or technology process, you will need to replace these files with files provided by the vendor of the new cell library and process.

- `cells.db` - Synopsys 90nm digital standard cell model library. Contains timing and area information for each standard cell.
- `cells_cg.db` - Additional standard cell models for clock gating cells.
- `milkyway.fr` - Milkyway Reference database for the 90nm standard cell library. Contains wireload models for the standard cells.
- `techfile.tf` - Technology file containing charactersitics and design rules for the 90nm process.
- `tech2itf.map` - Technology to ITF (interconnect technology) mapping file.
- `max.tluplus` - Models containing advanced process effects.
- `min.tluplus` - Models containing advanced process effects.

4 Manual Design Compiler Build Process

We will first go through the commands for the tool manually from the commandline so that you can see all the different steps that are required to make the tool work. Since this is extremely tedious, we will only do this once, and later we will use scripts to automate the steps in this portion of the flow for us.

DC can generate a large number of output files, so you will be running DC within a build directory beneath `dc-syn`. Use the following commands to create a build directory for DC and to start the DC shell:

```
% cd $TUTROOT/build/dc-syn
```

```
% mkdir manual
% cd manual
% dc_shell-xg-t -64bit -topographical_mode
```

You should be left at the DC shell prompt from which you can can execute various commands to load in your design, specify constraints, synthesize your design, print reports, etc. You can get more information about a specific command by entering `man <command>` at the `dc_shell` prompt.

Execute the following commands manually in the `dc_shell-topo>` prompt. The first command will create an alias to "undefine" the `dc_shell-topo>` string, which will allow you to cut and paste commands from this tutorial into Design Compiler.

```
# Create an alias for copy and paste.

dc_shell-topo> alias "dc_shell-topo>" ""

# Setup the environment: point to your Verilog source directory, create a
# work directory for the tool, and point to the Synopsys process libraries.

dc_shell-topo> set stdcells_home /research/brg/install/bare-pkgs/noarch/synopsys-90nm/toolflow

dc_shell-topo> set_app_var search_path \
  "$stdcells_home ../../../../vclib/src ../../../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 \
  "/research/brg/install/bare-pkgs/noarch/synopsys-90nm/toolflow/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.tf \
  -mw_reference_library $stdcells_home/milkyway.fr "gcdGCDUnit_rtl_LIB"

dc_shell-topo> open_mw_lib "gcdGCDUnit_rtl_LIB"
dc_shell-topo> check_library

dc_shell-topo> set_tlu_plus_file \
  -max_tluplus $stdcells_home/max.tluplus \
  -min_tluplus $stdcells_home/min.tluplus \
  -tech2itf_map $stdcells_home/tech2itf.map

dc_shell-topo> check_tlu_plus_files
dc_shell-topo> define_design_lib WORK -path "./work"

# Our environment should be setup, now load your Verilog design into DC with
# the analyze, elaborate, link, and check design commands. Executing these
# commands will result in a great deal of log output as the tool elaborates
# Verilog constructs and starts to infer some high-level components.

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

```
# During elaboration DC will report all state inferences. This is a good way
# to verify that latches and flip-flops are not being accidentally inferred.

dc_shell-topo> elaborate "gcdGCDUnit_rtl"
dc_shell-topo> link

# The check_design command checks that the design is consistent. You will not
# be able to synthesize your design until you eliminate all ERRORS. Many
# WARNINGS are not an issue, but it is still useful to skim through this output.

dc_shell-topo> check_design

# Set the constraints.  Before we can synthesize our design, we must specify
# some constraints like the target clock period. The following command tells
# the tool that the pin named clk is the clock and that the desired clock
# period is 1 nanosecond.

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

# The compile_ultra command begins the actuall synthesis process that
# transforms your design into a gate-level netlist. The -no_autoungroup
# flag is specied in order to preserve the hierarchy during synthesis.

dc_shell-topo> compile_ultra -gate_clock -no_autoungroup

# The compile_ultra command may take a while!

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 ../rm_dc_scripts/find_regs.tcl
dc_shell-topo> find_regs gcdTestHarness_rtl/gcd

# Report results, these reports will be explained later in the tutorial,
# just browse them for now.

dc_shell-topo> report_timing -transition_time -nets -attributes -nosplit
dc_shell-topo> report_area -nosplit -hierarchy
dc_shell-topo> report_power -nosplit -hier
dc_shell-topo> report_reference -nosplit -hierarchy
dc_shell-topo> report_resources -nosplit -hierarchy
```

You can now use various commands to examine timing paths, display reports, and further optimize your design. Using the shell directly is useful for finding out more information about a specific command or playing with various options, but for reproducibility and convenience reasons you will primarily use TCL scripts to control this tool.

The final step in the Manual Synthesis process is exiting the DC shell and deleting your build directory:

```
dc_shell-topo> exit
```

```
% cd $TUTROOT/build/dc-syn
% rm -rf manual
```

The remainder of this section is dedicated to discussing in further detail some of the more salient phases of the Design Compiler synthesis flow.

Analysis and Elaboration

The `analysis` command checks your HDL design for proper syntax and synthesizable logic, and then translates this design into an intermediate format inside the specified WORK directory. Elaboration then takes this intermediate representation and begins the task of turning your RTL description into actual hardware. This includes replacing HDL arithmetic operators with synthetic operators (ie. DesignWare components), determining correct bus size, and inferring the presence of state elements such as latches and flip-flops. As a result, the output generated by the analysis and elaboration phases is a good way to quickly assess whether the tools are honoring your design intent.

For more information on the output from the `elaborate` command, and more generally how DC infers combinational and sequential hardware elements, you may want to see the *HDL Compiler for Verilog User Guide* (`dc-user-guide-verilog.pdf`).

Check Design

The `check_design` command is used to check the internal representation of your design and to correct certain design problems. Some of the errors you may encounter when running `check_design` include unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, and recursive hierarchy denitions.

As mentioned earlier, while many WARNINGs are not an issue, you will not be able to synthesize your design until you eliminate all ERRORS. As a best practice you should always do your best to minimize as many WARNINGs as possible, and understand any remaining WARNINGs.

Constraints

DC makes a best effort attempt to synthesize your design while still meeting the two types of constraints: *user specified constraints* and *design rule constraints*. User specified constraints can be used to constrain the clock period (as was done with the `create_clock` command) as well as the arrival of certain input signals, the drive strength of the input signals, and the capacitive load on the output signals. Design rule constraints are fixed constraints which are specified by the standard cell library. For example, there are restrictions on the loads specific gates can drive and also on the transition times of certain pins.

The clock period constraint is one of the most important constraints you must consider because it must be set carefully. If the period is unrealistically small, then the tools will spend forever trying to meet timing and ultimately fail. If the period is too large, then the tools will have no trouble but you will get a very conservative implementation.

Note that designs encountered later in the course will need to have constraints set on their input and output delays as well. In fact, in order to get the correct critical path through your GCD design, it may be necessary to specify these delays. An example of specifying these input and output delays can be seen below:

```
dc_shell-topo> set_output_delay -clock ideal_clock1 2.3 imemreq_bits_addr
dc_shell-topo> set_output_delay -clock ideal_clock1 2.3 imemreq_val
dc_shell-topo> set_input_delay -clock ideal_clock1 0.6 imemresp_bits_data
```

For more information about constraints consult the *Synopsys Timing Constraints and Optimization User Guide* (`dc-user-guide-tco.pdf`) and additionally the *Synopsys Design Constraints Format Application Note* (`dc-application-note-sdc.pdf`).

Synthesis

The `compile_ultra` command will synthesize your design into a gate level netlist and during the compilation process report how the design is being optimized. Some of the steps you should observe DC performing are technology mapping, delay optimization, and area reduction. Figure 2 shows a fragment from some example compile output.

```
dc_shell-topo> compile_ultra -gate_clock -no_autoungroup
...
 ELAPSED              WORST NEG TOTAL NEG  DESIGN                     LEAKAGE
   TIME      AREA       SLACK     SLACK   RULE COST    ENDPOINT       POWER
 --------- --------- --------- --------- --------- ------------- ---------
   0:00:25  56092.3      0.68     193.7       0.0                 308556192.0000

Beginning Delay Optimization
----------------------------
   0:00:28  51442.8      0.28       5.4       0.0                 274241280.0000
   0:00:28  51589.3      0.23       4.5       0.0                 275486656.0000
   0:00:28  51589.3      0.23       4.5       0.0                 275486656.0000
   0:00:33  52530.3      0.12       2.1       0.0                 281122144.0000
   0:00:33  52530.3      0.12       2.1       0.0                 281122144.0000
   0:00:35  54100.7      0.00       0.0       0.0                 295563680.0000
   0:00:35  54100.7      0.00       0.0       0.0                 295563680.0000
   0:00:35  54100.7      0.00       0.0       0.0                 295563680.0000
...
```

Figure 2: Example output from the Design Compiler `compile_ultra` command

You'll see in the compile output of that each line represents an optimization pass. The area column is in units specific to the standard cell library (which is um^2, but for now you should just use the area numbers as a relative metric). The worst negative slack column shows how much room there is between the critical path in your design and the clock constraint. The total negative slack is the sum of all negative slack across all endpoints in the design. A few useful rules of thumb about negative slack are:

- Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a greater amount.
- If the total negative slack is a large negative number it indicates that not only is the design not making timing, but it is possible that *many* paths are too slow.
- If the total negative slack is a small negative number, then this indicates that only a few paths are too slow.

The design rule cost is a indication of how many cells violate one of the standard cell library design rules constraints. Figure 2 shows that on the first iteration, the tool makes timing but at a high area cost, so on the second iteration it optimizes area but this causes the design to no longer meet timing. The tool continues to optimize until it meets the constraints.

The flags you will most commonly use with the `compile_ultra` command include:

- `-gate_clock` enables clock gating.
- `-no_autoungroup` is preserves the hierarchy during synthesis or disables inter-module optimizations.
- With no options given, `compile_ultra` will optimize across module boundaries.

For more information on the `compile_ultra` command you can consult the *Design Compiler User Guide* (`dc-user-guide.pdf`) or use `man compile_ultra` at the DC shell prompt.

5   Automated Design Compiler Build Process

Typing each command via the shell is a tedious and error-prone process, and should typically be avoided. Instead, we make use of scripts to automate the process of executing our tools for us. The various TCL scripts and makefiles used to automate synthesis are described below:

- `Makefrag` - Makefile fragment, this contains the clock period and some other references. Notice that it uses a shorter clock period (90% of the desired clock period) for synthesis in order to leave some slack for place and route.
- `dc-syn/Makefile` - Makefile for driving synthesis with the TCL scripts.
- `dc-syn/rm_setup/dc_setup.tcl` - TCL fragment defines various library variables, creates the search paths, sets the work directory, loads `make_generated_vars.tcl`.
- `dc-syn/rm_dc_scripts/dc.tcl` - Primary TCL script which contains the DC commands. Contains many of the commands executed manually in the previous section, plus several more.
- `dc-syn/rm_dc_scripts/find_regs.tcl` - Script which finds all registers in the design. This is used by gate-level simulation to initalize all state to a known value at startup.
- `dc-syn/rm_setup/common_setup.tcl` - TCL fragment for various common variables.
- `dc-syn/rm_setup/dc_setup_filenames.tcl` - TCL fragment for various filename variables
- `dc-syn/constraints.tcl` - User specified constraints, such as assumed input drive strength (minimum sized inverters) and output load (4 fF of capacitance).
- `make_generated_vars.tcl` - Automatically generated by the makefile, this TCL fragment contains variables which are defined in the makefile and used by the TCL scripts.

Thanks to all the configuration provided by the Makefiles and the TCL scripts, the synthesis process can be completely automated. The following commands will automatically synthesize your design and save several text reports to the build directory:

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

You should see Design Compiler start and then execute the commands located in the `dc.tcl` script. The results of synthesis are stored in new build directories with the format `build-<date_time>`. Each build directory contains copies of all TCL scripts used to perform the synthesis so that it's possible to go back and check the configuration of a specific build. The current-dc symlink will always point to the most recent build directory.

If you try running `make dc` after a previous synthesis without changing any configuration files, the makefile should detect that no modifications occurred (all Verilog source files and DC scripts are the same) and do nothing. If you change the the clock period constraint in `Makefrag` and try running `make dc` one more time, DC should correctly detect that the clock period changed and resynthesize the design.

Note that the makefile does *not* overwrite build directories, it always creates new ones. 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 what various build directories correspond to. For example, the following commands label two build directories with their clock period constraints:

```
% cd $TUTROOT/build/dc-syn
% ln -s build-dc-2012-01-12_11-11 build-1ns
% ln -s build-dc-2012-01-12_11-13 build-1.5ns
```

Every so often you should delete old build directories to save space. The `make clean` command will delete *all* build directories so use it carefully. Sometimes you want to really force the makefile to resynthesize the design but for some reason it may not work properly. To force a resynthesis without doing a `make clean` simply remove the `current` symlink. The following commands will force a resynthesis without actually changing any of the source TCL scripts or Verilog:

```
% cd $TUTROOT/build/dc-syn
% rm -f current-dc
% make dc
```

Eventually you will need to modify the build system to synthesize new designs you come up with for the labs. To modify the Makefile so that it will build different sources, you will need to change:

- all Verilog files containing hardware for your designs.

  NOTE: do not include the test harness! They are not-synthesizable!
- the name of the toplevel Verilog module in your design
- the name of the toplevel module instance in your design
- the name of the test harness module

6 Interpreting the Gate-Level Netlist and Synthesis Reports

The primary output from the synthesis scripts is the synthesized gate-level netlist which is contained in `dc-current/results/gcdGCDUnit_rtl.mapped.v`. Inside the gate-level netlist you'll see that the RTL module hierarchy is preserved since you did not flatten any part of your design. From the gate-level netlist you can determine all the different cells the tool synthesized for various parts of the design. To find out more about each of these cells, you can refer to the databook for the Synopsys 90nm Standard Cell Library (`SAED_Digital_Standard_Cell_Library.pdf`).

In addition to the actual synthesized gate-level netlist the `dc.tcl` also generates several reports, which generally have the `rpt` filename suffix. The following is a list of the synthesis reports:

- `reports/*.mapped.area.rpt` - Area information for each module instance
- `reports/*.mapped.clock_gating.rpt` - Clock gating information
- `reports/*.mapped.power.rpt` - Power information for each module instance
- `reports/*.mapped.qor.rpt` - QoR (Quality of Result) information and statistics
- `reports/*.mapped.reference.rpt` - Information on references
- `reports/*.mapped.resources.rpt` - Information on Design Ware components
- `reports/*.mapped.timing.rpt` - Contains critical timing paths
- `log/dc.log` - Log file of all output during DC run

The `area.rpt` report contains area information for each module in the design. Figure 3 shows a fragment from `area.rpt` for the GCD module. You can use the area report to measure the relative area of the various modules. The report clearly shows that the majority of the processor area is in the datapath.

	Global cell area		Local cell area		
Hierarchical cell	Absolute Total	Percent Total	Combi- national	Noncombi- national	Black boxes
gcdGCDUnit_rtl	3083.6736	100.0	21.1968	0.0000	0.0000
ctrl	217.4976	7.1	167.7312	49.7664	0.0000
dpath	2844.9792	92.3	1978.6752	796.2624	0.0000
dpath/clk_gate_A_reg_reg	35.0208	1.1	12.9024	22.1184	0.0000
dpath/clk_gate_B_reg_reg	35.0208	1.1	12.9024	22.1184	0.0000
Total			2193.4080	890.2656	0.0000

Figure 3: Fragment from `gcdGCDUnit_rtl.mapped.area.rpt`

```
****************************************
Design: gcdGCDUnitCtrl
****************************************
Reference          Library       Unit Area  Count   Total Area   Attributes
---------------------------------------------------------------------------
DFFX1              saed90nm_typ  24.883200       2   49.766399 n
INVX0              saed90nm_typ   5.529600       6   33.177601
INVX1              saed90nm_typ   6.451200       1    6.451200
NAND2X0            saed90nm_typ   5.529600       8   44.236801
NOR2X0             saed90nm_typ   5.529600      12   66.355202
NOR2X2             saed90nm_typ   9.216000       1    9.216000
NOR3X0             saed90nm_typ   8.294400       1    8.294400
---------------------------------------------------------------------------
Total 7 references                                 217.497603
```

Figure 4: Fragment from `gcdGCDUnit_rtl.mapped.reference.rpt`

You can use the `reference.rpt` report (Figure 6) to gain insight into how various modules are being implemented. This provides similar information to the gate-level verilog, but in a more user friendly format.

The see the *critical path* of the design, we can look at the path timing information provided `timing.rpt`. The critical path is the slowest logic path between any two registers, and therefore is the limiting factor preventing a designer from decreasing the clock period constraint (increasing clock speed). Note that the timing report is generated from a purely static worst-case timing analysis (i.e. independent of the actual signals which are active when the processor is running).

You can see an example fragment from the `timing.rpt` in Figure 5. The first column lists various nodes in the design (several have been left out for the sake of brevity), the middle column shows the incremental delay added by that node, and the last column lists the total cumulative delay to the node.

The `resources.rpt` report identifies when DC is using Design Ware components in your design. Design Ware is a Synopsys provided library of highly optimized building blocks used to implement commonly used arithmetic components. Whenever possible, DC will automatically use Design Ware components in order to improve the performance of your design.

In the `resources.rpt` fragment in Figure ?? you can see DC has instantiated two differen Design Ware components: a subtractor and a comparator. Additionally, the report describes why the specific Design Ware component was chosen over another. In this case this comparator architecture was chosen for area reasons, while the subtractor was chosen for both area and speed reasons.

Try playing around with the cycle time constraint of the GCD modules and see if this changes which Design-Ware modules are instantiated. To find out more about the components available in Design Ware, refer to the Building Block IP described in Chapter 2 of the *Design Ware Quick Reference Guide* (`designware-quick-reference.pdf`). To find out more information about specfic Design Ware components, you can refer to the corresponding Design Ware datasheet located in `/classes/ece5745/docs`.

7 Using Design Vision to Analyze the Gate-Level Netlist

Synopsys provides a GUI front-end to 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 you want to use

```
Startpoint: dpath/B_reg_reg_10_
            (rising edge-triggered flip-flop clocked by ideal_clock1)
Endpoint: dpath/clk_gate_A_reg_reg/latch
          (positive level-sensitive latch clocked by ideal_clock1')
Path Group: ideal_clock1
Path Type: max

Point                                        Fanout      Cap    Trans     Incr       Path
------------------------------------------------------------------------------------------
clock ideal_clock1 (rise edge)                                           0.0000     0.0000
clock network delay (ideal)                                             0.0000     0.0000
dpath/B_reg_reg_10_/CLK (DFFX1)                                0.0000   0.0000     0.0000 r
dpath/B_reg_reg_10_/Q (DFFX1)                                  0.0369   0.1792     0.1792 f
dpath/B_reg[10] (net)                           2      5.2088           0.0000     0.1792 f
dpath/U43/IN2 (NAND2X0)                                        0.0369   0.0000 *   0.1792 f
.
.
.
ctrl/U6/QN (NAND2X0)                                           0.0524   0.0315     0.7993 f
ctrl/en_A (net)                                 1      1.8077           0.0000     0.7993 f
ctrl/en_A (gcdGCDUnitCtrl)                                              0.0000     0.7993 f
en_A (net)                                             1.8077           0.0000     0.7993 f
dpath/en_A (gcdGCDUnitDpath_W16)                                       0.0000     0.7993 f
dpath/en_A (net)                                       1.8077           0.0000     0.7993 f
dpath/clk_gate_A_reg_reg/EN (SNPS_CLOCK_GATE_HIGH_gcdGCDUnitDpath_W16_1)   0.0000     0.7993 f
dpath/clk_gate_A_reg_reg/EN (net)                      1.8077           0.0000     0.7993 f
dpath/clk_gate_A_reg_reg/latch/D (LATCHX1)                    0.0524   0.0000 *   0.7993 f
data arrival time                                                                 0.7993

clock ideal_clock1' (rise edge)                                         0.4500     0.4500
clock network delay (ideal)                                            0.0000     0.4500
dpath/clk_gate_A_reg_reg/latch/CLK (LATCHX1)                           0.0000     0.4500 r
time borrowed from endpoint                                            0.3493     0.7993
data required time                                                                0.7993
------------------------------------------------------------------------------------------
data required time                                                                0.7993
data arrival time                                                               -0.7993
------------------------------------------------------------------------------------------
slack (MET)                                                                       0.0000
```

Figure 5: Fragment from gcdGCDUnit_rtl.mapped.rpt

```
Resource Report for this hierarchy in file ../../../src/gcdGCDUnitDpath.v
=========================================================================
| Cell           | Module         | Parameters | Contained Operations    |
=========================================================================
| sub_x_1        | DW01_sub       | width=16   | sub_28                  |
| lt_x_4         | DW_cmp         | width=16   | lt_46                   |
=========================================================================


Implementation Report
===============================================================================
|                |                | Current          | Set              |
| Cell           | Module         | Implementation   | Implementation   |
===============================================================================
| sub_x_1        | DW01_sub       | pparch (area,speed)                 |
| lt_x_4         | DW_cmp         | apparch (area)   |                  |
===============================================================================
```

Figure 6: Fragment from `resources.rpt`

scripts for this. To launch Design Vision and read in your synthesized design, move into the appropriate working directory and use the following commands:

```
% cd $TUTROOT/build/dc-syn/current-dc
% design_vision-xg -64bit
design_vision> source dc_setup.tcl
design_vision> read_ddc results/gcdGCDUnit_rtl.mapped.ddc
```

You can browse your design with the hierarchical view, then right click a module and select the *Schematic View* option to display a schematic of the module's synthesized logic. Figure 7 shows the schematic view for a synthesized design, notice that you can see the individual standard cells such as `MUX21X1`.

You can use Design Vision to examine various timing data. The *Timing > Paths Slack* menu option will create a histogram of the worst case timing paths in your design. You can use this histogram to gain some intuition on how to approach a design which does not meet timing: if there are a large number of paths which have a very large negative timing slack then a global solution is probably necessary, while if there are just one or two paths which are not making timing a more local approach may be sufficient. You can click on a bin and the tool will report critical paths in the bin. Figure 8 shows an example of using these two features. You can also right click to choose *Path Inspector*, which will show the actual components on the critical path (Figure 9).

It is sometimes useful to examine the critical path through a single submodule, this can be done by performing the following steps:

- Right click on the module in the hierarchy view and select the *Characterize* option.
- Check the timing, constraints, and connections boxes.
- Click OK.
- Choose the module from the dropdown list on the toolbar (aka the *Design List*).
- Select *Timing > Report Timing Path* from the menu bar.

Note that the resulting critical path is based on the the constraints imposed on the submodule by the overall design. This may not be the same as the critical path of the submodule if it were tested in isolation.

For more information on Design Vision consult the *Design Vision User Guide* (`dc_dv-user-guide.pdf`).
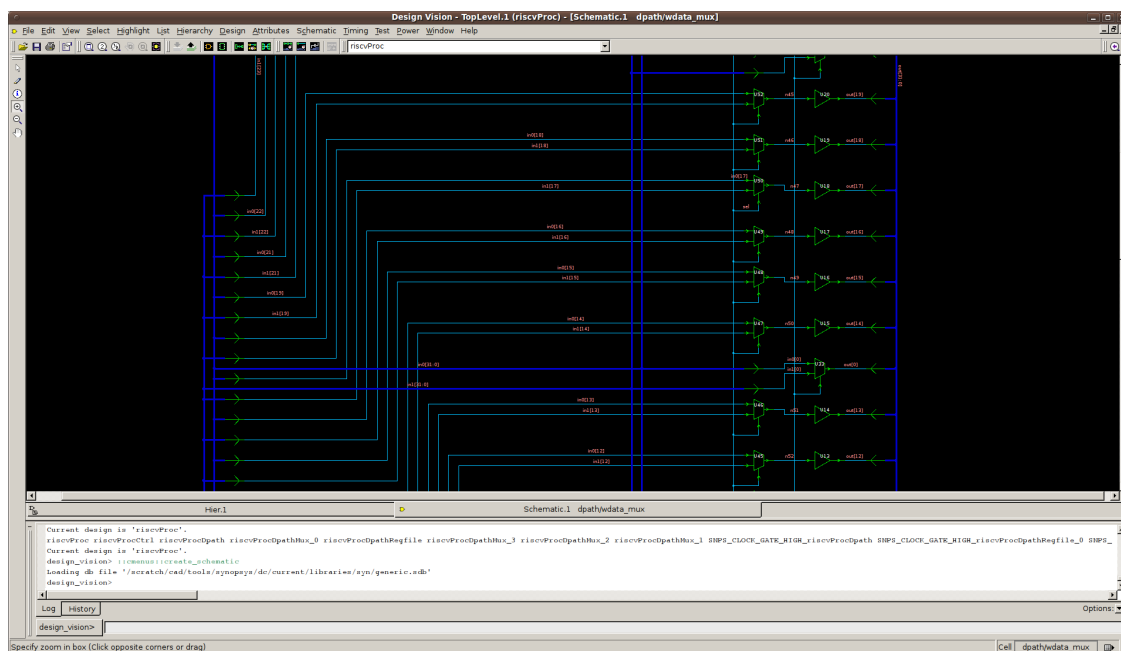
Figure 7: Screen shot of a schematic view in Design Vision

8   Acknowledgements

Many people have contributed to versions of this tutorial over the years. The tutorial 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 tutorial have been used in the following courses:

- CS250 VLSI Systems Design (2009-2011) - University of California at Berkeley
- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CSE291 Manycore System Design (2009) - University of California at San Diego
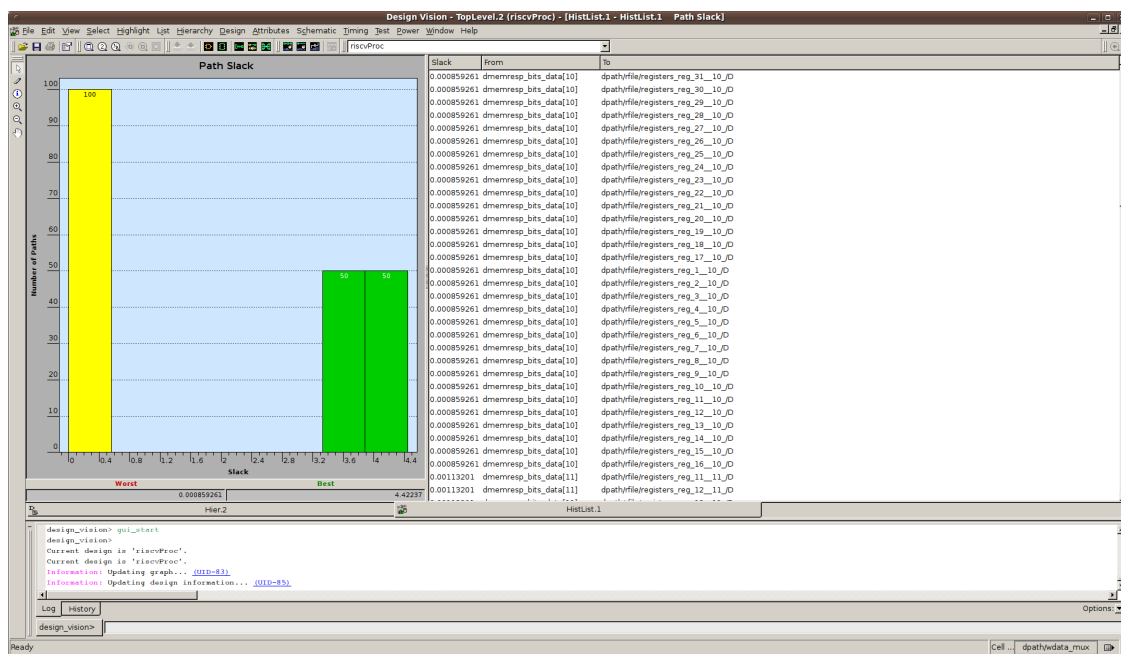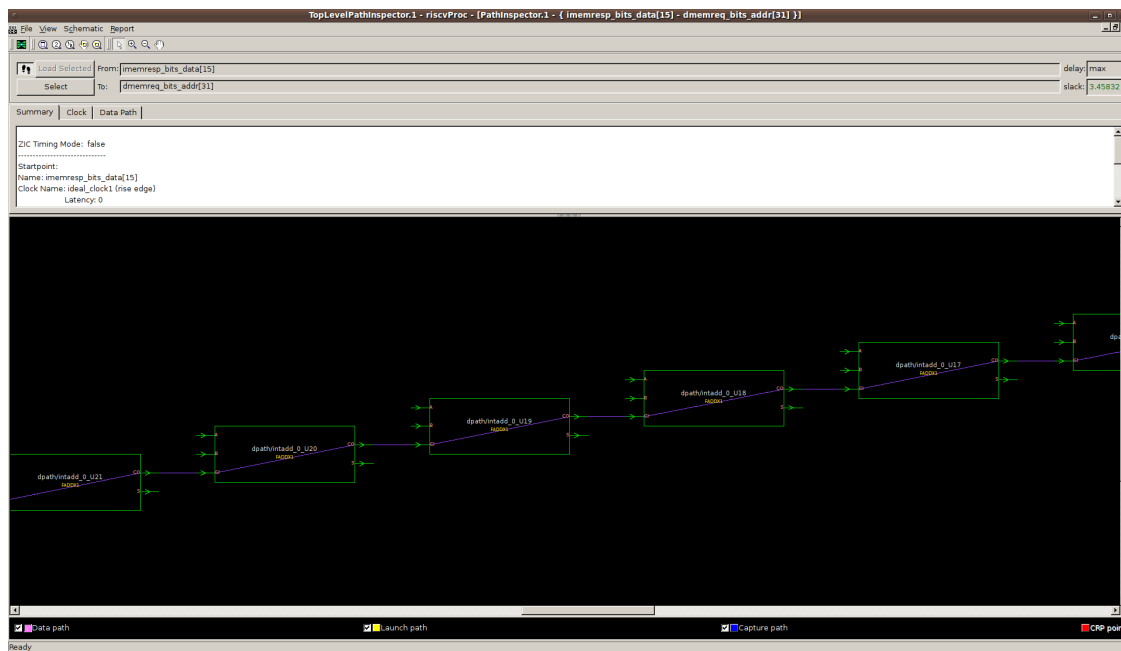
Figure 8: Screen shot of timing results in Design Vision

Figure 9: Screen shot of the path inspector in Design Vision