SYNOPSYS®

CUSTOMER EDUCATION SERVICES

Design Compiler: RTL Synthesis Workshop

Lab Guide

10-I-011-SLG-024 2017.09

Synopsys Customer Education Services
690 E. Middlefield Road
Mountain View, California 94043

Workshop Registration: http://training.synopsys.com

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Document Order Number: 10-I-011-SLG-024
Design Compiler: RTL Synthesis Lab Guide

1 Data Setup for DC Topographical Mode

Learning Objectives

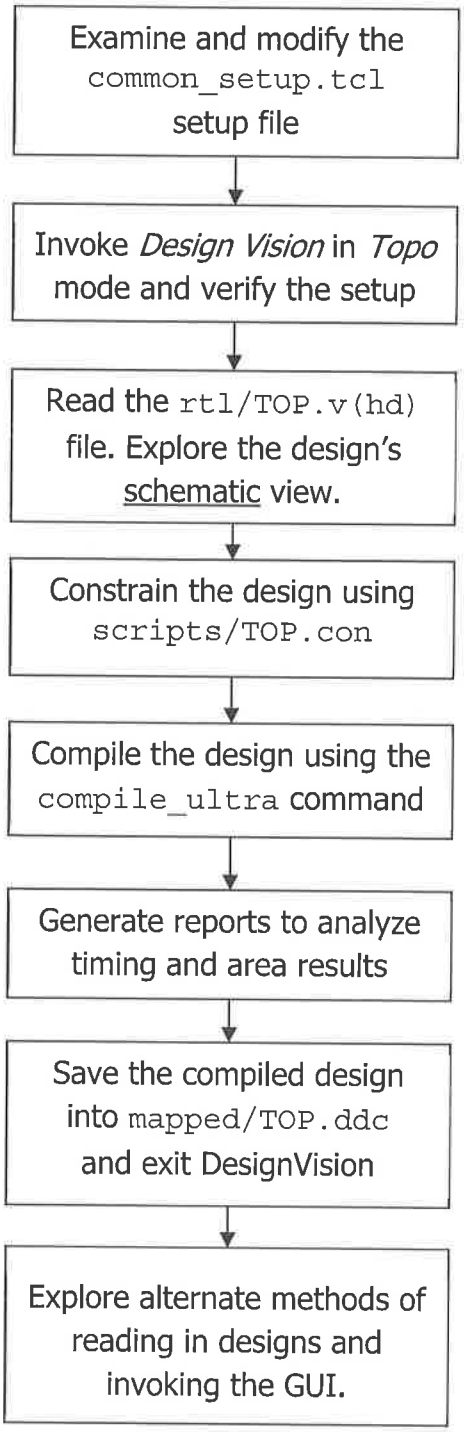
After completing this lab, you should be able to:

- Update the `common_setup.tcl` file to fully specify the logic and physical library and technology files
- Explore the schematic view in *Design Vision*
- Take a design through the basic synthesis steps in *Topographical mode* and generate reports
- Visit *SolvNet* to browse the user manual for Design Vision

Lab Duration: 60 minutes

Lab Flow

Follow the detailed step-by-step **Lab Instructions** on the following pages to perform the steps highlighted in this flow:



Lab Instructions

Task 1. Examine and Modify the Setup Files

1. Make the **lab1** directory your current directory and list the following files:

```
UNIX% cd lab1
UNIX% ls -al .synopsys*
UNIX% ls -al *setup*
```

You are provided with a `.synopsys_dc.setup` file. It defines aliases and sources two other setup files: `common_setup.tcl` and `dc_setup.tcl`.

The `common_setup.tcl` file contains user defined variables (in UPPER CASE letters) which specify technology file and directory names. These variables are used in `dc_setup.tcl`.

The `dc_setup.tcl` file executes commands to load the necessary logic and physical technology data, using variables from `common_setup.tcl`.

2. You will NOT have to modify the `.synopsys_dc.setup` or `dc_setup.tcl` files for this lab. However, verify that the `dc_setup.tcl` file contains the "*****" in the `link_library`.

3. Using a text editor of your choice, edit `common_setup.tcl` and incorporate the missing information (in **bold**) from the following table:

User-defined variable	Directory or File Names
`ADDITIONAL_SEARCH_PATH` Additional `search_path` directories for logic (db) libraries, design files, and scripts	`../ref/libs/mw_lib/sc/LM` **`./rtl`** **`./scripts`**
`TARGET_LIBRARY_FILES` Logic Cell Library file	**`sc_max.db`** (located in the `LM` view of the *Milkyway* cell library)
`SYMBOL_LIBRARY_FILES` Symbol library file	**`sc.sdb`** (located in the `LM` view of the *Milkyway* cell library)

This table is continued on the next page.

User-defined variable	Directory or File Names
MW_DESIGN_LIB Milkyway Design Library Name	**TOP_LIB** (user-defined name)
MW_REFERENCE_LIB_DIRS *Milkyway* physical cell libraries (standard/macro/pad cells)	../ref/libs/mw_lib/**sc**
TECH_FILE Physical Technology file	../ref/libs/tech/cb13_6m.tf
TLUPLUS_MAX_FILE Max TLUPlus file	../ref/libs/tlup/cb13_6m_max.tluplus
MAP_FILE TLUPlus Layer Mapping file	../ref/libs/tlup/cb13_6m.map

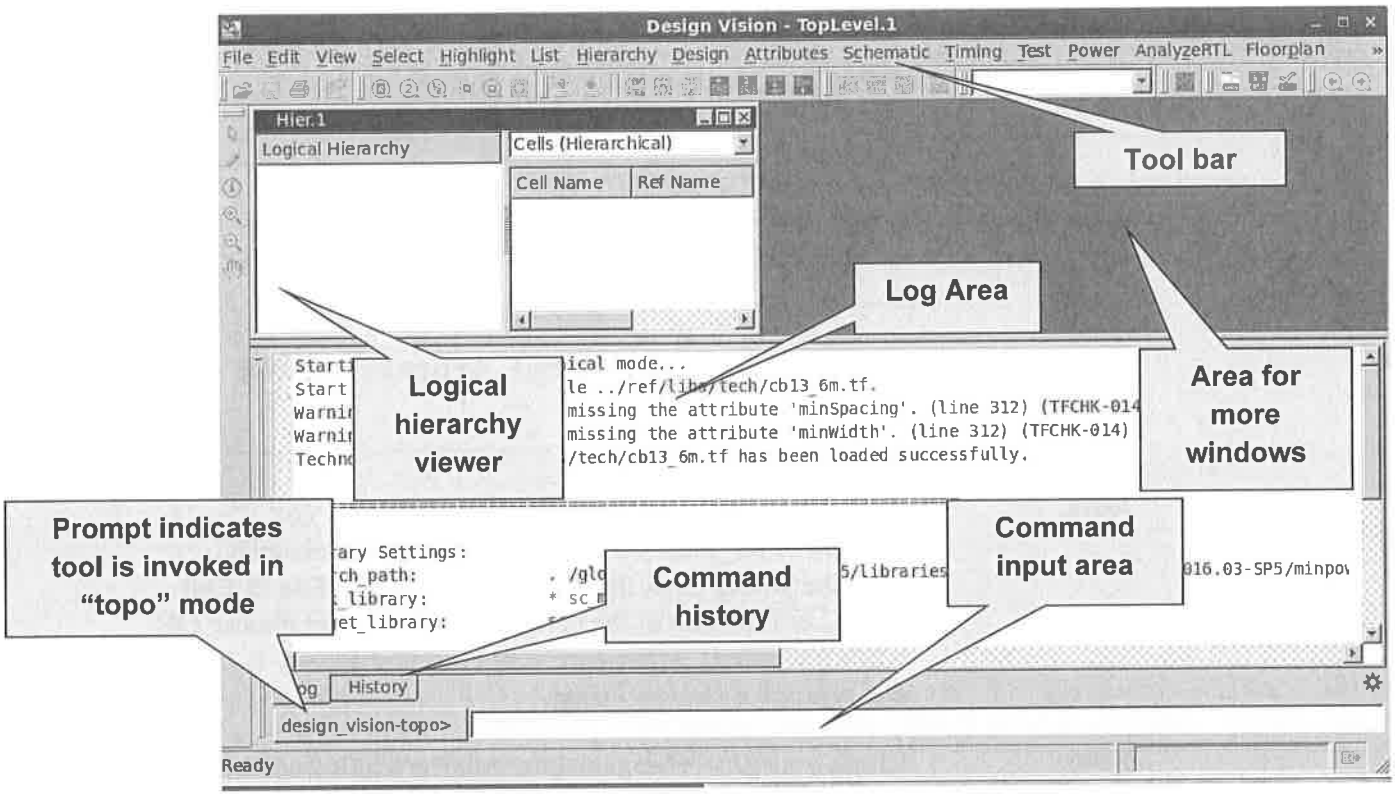
Task 2. Invoke Design Vision

1. Make sure you are still in the **lab1** directory and invoke *Design Vision* in *Topographical mode*. Recall that this step automatically sources the .synopsys_dc.setup file.

```
unix% pwd
unix% design_vision -topo
```

Note: You can ignore the 4.1 that appears at the prompt.

Note: If you get a MWUI-004 error message, "Library 'TOP_LIB' already exists", you can ignore it. This indicates that the TOP_LIB *Milkyway* design library directory was already created when *Design Vision* was first invoked. Other than specifying the TOP_LIB design library name, the setup variables you edited in Task 1 did not affect the design library, so it does not need to be re-created. If there would have been a problem with the design library itself, you would have had to delete the TOP_LIB directory prior to revinvoking the tool, in order to re-create a correct design library.



2. You can confirm that the tool is invoked in "Topographical mode" by noticing `-topo` in the command prompt, or by looking for the startup message that indicates "*Starting shell in Topographical mode...*".

3. View the ***Log Area*** at the bottom of the GUI.

   This area displays all the executed commands, their results, and shows any error messages.

   Scroll up to see the log messages.

   You should see the values of the key `"Library Settings"` variables echoed here, followed by the `"Initializing gui preferences"` message, and the "`gui_start`" command, which invokes the GUI.

4. Choose menu **File → Setup...** and verify that the libraries are set up correctly:

Question 1. What is the value of the `link_library` variable?

.....

Question 2. What is the value of the `target_library` variable?

.....

Question 3. What is the value of the `symbol_library` variable?

.....

Note: If the libraries are called **`your_library.db (or "--")`**, (instead of `sc_max.db`) this indicates you invoked DC from the wrong Unix directory. Exit the GUI (**File → Exit → OK**, or type **exit** at the command prompt, and choose **OK** when prompted). Make sure your current directory is ***`lab1`*** and re-invoke *Design Vision*.

Note: Check your answers against the **Answers/Solutions** section at the end of this lab, and fix your `common_setup.tcl` file accordingly. If you are stuck, compare your setup file with that provided in the ***.solutions*** directory.
If you edit a "setup" file like `common_setup.tcl`, you should exit *Design Vision*, then re-invoke the tool.

5. From **File→Setup...** select the  icon to the right of the ***Search path*** field. This opens up the *Set Search Path* dialog, which shows an expanded list of each search path directory. Verify that the first five entries at the top are as follows – these are the default `search_path` directories:

```
.
/<tool_installation_directory>/libraries/syn
/<tool_installation_directory>/minpower/syn
/<tool_installation_directory>/dw/syn_ver
/<tool_installation_directory>/dw/sim_ver
```

Question 4. What user directories have been added to the `search_path`? (make sure two directories were added)

.....

6. Click **Cancel** twice to close the *Set Search Path* and *Application Setup* dialogs.

7. In the original *Unix* window from which you invoked *Design Vision* (the shell interface), type the following commands at the DC prompt. This is another way to confirm variable settings, as well as user-defined and default aliases.

   Note: Command line editing allows for command, option, variable and file completion. Type a few letters and then hit the **[Tab]** key.

   Note: If your <Backspace> key deletes entire words, use Ctrl-H to back-space one character at a time.

```
printvar target_library
printvar link_library
printvar search_path
```

8. Check the consistency between the logic and physical libraries:

```
check_library
```

   Notice that, at the end of the check output, it reports that there are 4 cells missing in the logic library. There is a table listing the missing cells just below that (`feedth*`, `tap`). These are "feedthrough" and "tap" cells, which are required in the physical layout, but not in the logic design or netlist (also known as "physical-only" cells). The warning can, therefore, be ignored.

9. Check the consistency between the *TLUPus* and the *Technology* files:

```
check_tlu_plus_files
```

   You should see that all three checks "`Passed!`".

Task 3. Read the Design into DC Memory

Design Compiler can read *VHDL*, *Verilog*, as well as *SystemVerilog RTL* files.

1. Click on the **Read...** button  at the top-left of the GUI (or **File → Read**).
2. In the dialog box that appears, double-click on the directory `rtl`, and then again on `TOP.v` or `TOP.vhd`.

   In the "Logical Hierarchy" window on the left side of the GUI (you may need to widen the window), there is now an icon for `TOP`, which is the top-level *design* name. There are also icons for the lower-level hierarchical ("H") instances or *cells*: `I_COUNT`, `I_DECODE`, and `I_FSM`.
3. Select `TOP` (single click with left mouse button), and look at the lower-right corner of the GUI window to verify the selection:
   You should see **Design: TOP**. This ensures that your *current design* is properly set to the top-level design.
4. Select **File → Link Design → OK** to link the design and resolve all references. You should not see any warning or error messages in the *Log Area*.
5. Save the unmapped design in *ddc* format. Type the following in the *Command Input Area* or in the Unix window in which you invoked Design Vision:

   Note: Command line editing allows for command, option, variable and file completion. Type a few letters and then hit the **[Tab]** key.

```
write_file -hier -f ddc -out unmapped/TOP.ddc
```

6. Type the following non-GUI *dc_shell* commands to see a list of designs and libraries in memory:

```
list_designs
list_libs
```

Task 4. Explore the Schematic View

1. Make sure that the lower right corner still shows that **Design: TOP** is selected. If not, select `TOP` with a single click of the left mouse button in the *Logical Hierarchy* window.

2. Select the *schematic view* by clicking the  (**Create Schematic of Selected Objects**) icon in the tool bar. You will see the top-level *interface diagram* of the block called ***TOP***, with its input and output ports. The diagram is labelled `Schematic.1` in the upper-left corner of the new window. Double click on the diagram. The schematic of `TOP` will now expand into a block diagram containing the instances (`I_*`) of `FSM`, `DECODE` and `COUNT`.

   You now have two windows open in the GUI: The Hierarchy and Schematic view windows. Maximize the schematic window. The Logical Hierarchy window is also maximized, but is "behind" the Schematic window that you maximized. You can bring different windows to the foreground by selecting the appropriate tab below the view window. Press the **[F]** key to "fit" the schematic view to the full window.

3. Minimize the *Schematic* view window or select the left-most  *Hier.1* tab to make the *Logical Hierarchy* window visible.

4. Explore the `TOP` design's hierarchy by visiting the **Schematic View** of the various subdesigns:

   Select a sub-design in the **Logical Hierarchy** window and click on the  icon. This opens another schematic view window displaying the interface diagram of the sub-design. Double-click the interface diagram, or, from the top banner click on the down-arrow button  to see the schematic details of the subdesign.

   Note: Because you have not compiled these designs yet, you will see *GTECH* components (e.g. `GTECH_BUF`, `*SELECT_OP*`), not gates from the target cell library.

   Now click on the up-arrow button. You should see the schematic of `TOP`, containing the the three sub-designs (if you see the *interface diagram* instead, double-click to get to the schematic view of `TOP`). Explore the difference between the down-arrow button and double-clicking: From the `TOP` schematic, left-click `I_DECODE` to select it, then click the down-arrow. This displays the internals of `I_DECODE`. Now go back up to the schematic of `TOP`, and instead, double-click on `I_DECODE`: This shows you the internal `I_DECODE` logic, in context of the `TOP` design! Notice the connection to `I_FSM` on the left, and `I_COUNT` on the right. If you now double-click on the other sub-designs, they too will be expanded in context.

Task 5. Explore the Mouse Functions

1. **Click and hold** the **right mouse button** in a **schematic view** to see the available mouse functions.
2. Either select **Zoom Fit All** with the left mouse button, or press the **[F]** key, to maximize the view. Now either repeat **Step 1** and select **Zoom In Tool**, or press **[Z]**. With the left mouse button click and drag the rectangular area you want to zoom into. Press the **[ESC]** key to exit out of the ZOOM mode.
3. Repeat **Step 1** and select **Pan Tool**, or use the **Up/Down/Left/Right arrow** keys to pan around a zoomed-in view.
4. Use **Zoom Out Tool** or **[-]** and **Zoom In Tool** or **[+]** to zoom in and out of the center of the viewing area.
5. Return to **Zoom Fit All** or **[F]**.
6. A quicker way to zoom in and out is using "strokes". Press and hold the middle mouse button on the lower left corner of the rectangle you want to zoom into, then move the mouse while still pressed to the upper right corner. Only then release the middle mouse button. You just performed a zoom in gesture or stroke. By pressing the middle mouse button in place, a menu will appear with the defined strokes. Experiment with some more strokes. If your mouse has a middle scroll button, you can also zoom in and out with it.
7. To select multiple objects, experiment with using your **left mouse button** and the **CTRL key**. Use the left mouse button to select the first object, then **left mouse button and CTRL key** to select additional objects. *Selected objects are highlighted in white.* Click in any black area to un-select the selected objects.

Recall the Basic Steps in Synthesis Flow

The four steps after "read" will be performed in the upcoming tasks:

- Read and translate RTL code (`read_vhdl/read_verilog`)
- Constrain the design (`source` a constraints file)
- Synthesize the design (`compile_ultra`)
- Generate reports (`report_*`)
- Save the resulting netlist (`write_file, write_icc2_files`)

Task 6. Constrain TOP with a Script file

1. Open the **Schematic** view for `TOP`.

   Note: If your *Logical Hierarchy* window is closed, re-open it by selecting **Hierarchy → New Logical Hierarchy View**

2. Constrain the `TOP` design by typing the following at the command prompt on the bottom of the Design Vision window. Remember to take advantage of command completion by hitting the tab key.

```
source -verbose TOP.con
```

   Note: You will see a bunch of `1`'s, as a result of the `-verbose` option: A `1` indicates that a command was successfully applied.

   Note: If the `source` command gives an error message, make sure that the `./scripts` directory has been added to the `ADDITIONAL_SEARCH_PATH` variable in `common_setup.tcl`, then exit and re-invoke DC. Alternatively, type `source scripts/TOP.con` if you do not want to re-invoke the Design Vision.

   You will not be able to "see" the constraints in the view window, but they are there. In upcoming labs you will learn how to generate reports to verify the constraints that have been applied to a design.

Task 7. Compile or Map to Vendor-Specific Gates

1. To compile the design, close the GUI and type the following command at the command prompt on the bottom of the *Design Vision* window:

```
gui_stop
compile_ultra
```

   Monitor the log as *compile* progresses. You will see various tables for the different optimization phases of compile. The "AREA" column indicates the design size. The "WORST NEG SLACK" column indicates by how much the critical or worst path in the design is violating, relative to its constraint (Actual delay – Expected delay). The "TOTAL SETUP COST" is the sum of all the violating path slacks for setup timing. When optimization reaches a point of diminishing returns, or the slack and cost numbers reach zero, which means that there are no timing or DRC violations, `compile_ultra` ends.

2. Once compile is completed, start the GUI:

```
gui_start
```

3. You will notice that the schematic(s) disappeared after compile. Repeat Task 4, steps 1 and 2, to display the compiled schematic view of TOP.

4. Notice that compile_ultra has automatically flattened or "ungrouped" the sub-designs DECODE, FSM and COUNT, to generate optimal timing and area results (you will learn about "auto ungrouping" later in this course).

5. Use the right mouse zoom functions, the [Z] and [F] keys, the middle mouse button "strokes", or the buttons on the command bar to explore the **Schematic View** of TOP. You will now see gates from the target cell library – no longer GTECH gates.

Task 8. Generate Reports and Analyze Timing

1. At the design_vision-topo> prompt type:

```
rc
```

rc is an alias that was specified in the .synopsys_dc.setup file. It executes the following command: report_constraint -all_violators

Note: Use the shell window behind the GUI, to view the report.

This report lists a summary of all constraint violations. You should see several "max_delay/setup" violations. There is also a max_leakage violation – this is because the default max_leakage constraint is 0.

You can also get more detailed timing path information by generating a timing report:

```
rt
```

By default, report_timing shows the timing of the critical path per clock.

Record the following *Worst Timing Violation*:

Slack (VIOLATED): _____

Note: You don't need to worry about any timing violation for this lab, you will learn how to analyze timing violations and choose strategies to improve timing later in this course.

2. Generate an area report, **ra**, and record the following:

Total cell area: _____

3. Locate the histogram buttons  at the top of the window. Hover your mouse over the middle button – a "tool hint" should display "*Create endpoint slack histogram*".

4. Click on the **Create endpoint slack histogram** button, and in the dialog box that appears, select **OK**.

   You will see a histogram window displaying several vertical bins. Each bin represents a number of timing paths to endpoints (output ports or register input pins). A green bin indicates that all timing paths within that bin meet timing. A red bin indicates violating paths.

   When a bin is selected, it turns yellow, and the timing slack (expected delay – actual delay) and end-point of the paths are listed in the right section of the dialog box.

5. Select the red violating bin (left mouse button), and on the right side, select the top-most endpoint (left mouse button).

   You can show the path to the selected endpoint in the schematic, as follows:

   First select the left-most  "**Create Schematic of Selected Objects**" button – this shows the endpoint of the violating path – a register. Now select the middle "**Add Paths to Path Schematic**" button and click **OK** – this adds the path leading up to the endpoint (by default, the worst violating path is shown). You can optionally select the right-most "**Add Fanin/Fanout to Path Schematic**" button to include fan-ins or fan-outs of specific pins (a "green arrow") along the path. Alternatively, for this optional step, you can click the right mouse button and select *Add Logic → Fanin/Fanout.*

Task 9. Save the Optimized Design

After compile, or after any major steps, it is advisable to save the design. The native Design Compiler format is *ddc*, both for unmapped or mapped (compiled) designs.

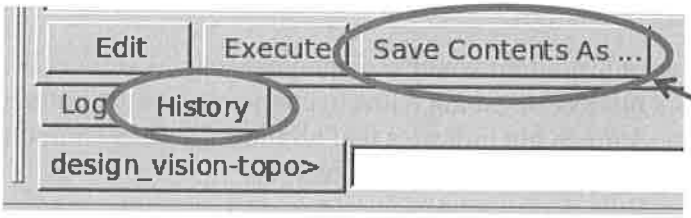
1. Choose menu **File → Save As.**
2. Double click on the `mapped` directory.
3. Enter `TOP.ddc` in the *File name* field.
4. Verify that the **Save all designs in hierarchy** button is selected. This will save the entire design hierarchy into a single (`.ddc`) file.
5. Click **Open.**

   You just saved the gate-level netlist (the entire hierarchy) in 'ddc' format under the `mapped` directory. You can verify that the file was created by entering '`ls mapped`'.

6. Save the gate-level design for IC Compiler II using the command:

```
write_icc2_files -output ./mapped/TOP_icc2
```

7. Next, select the **History** tab in the bottom-left corner of the GUI.



8. **Highlight** some or all of the commands with the left mouse button, then save the command history by selecting "**Save Contents As**" and specifying the file name **dc.tcl** into the **scripts** directory.

Task 10. Remove Designs and Exit Design Vision

1. Remove all designs from DC memory:

```
fr
list_designs
```

Notice that all the icons in Design Vision have been deleted. The "***fr***" alias executes the following command: `remove_design -designs`.

2. List a history of all commands executed since invoking Design Vision:

```
h ; # Alias for "history"
```

These executed commands are automatically logged in the `command.log` file that has been created in your `lab1` project working directory. If you unintentionally exited out of Design Vision, you could recreate everything you did up to that point by doing the following (do not do this now):

- Copy and rename the `command.log` file, and edit the copy to remove any "`exit`" or "`quit`" command at the end

- Execute the log file:
  `UNIX% design_vision -topo -f command_copy.log`

- Alternatively, you can use the **dc.tcl** file that you saved previously:
  `UNIX% design_vision -topo -f scripts/dc.tcl`

3. Exit from Design Vision. Use the menu sequence **File → Exit → OK**, or type `exit` at the command prompt, and choose **OK** when prompted.

4. Now try an alternate way to invoke the *DesignVision* GUI, by launching the interactive DC shell and then invoking the GUI:

```
UNIX% dc_shell -topo
dc_shell-topo> gui_start (or start_gui)
```

Note: You can ignore the Error message:
`Library 'TOP_LIB' already exists.`

5. You can return to the shell mode either from the *Design Vision* GUI: **File → Close GUI**, or from the command line:

```
design_vision-topo> gui_stop (or stop_gui)
```

6. **Exit** the DC shell.

Task 11. (OPTIONAL) Browse Documentation on *SolvNet*

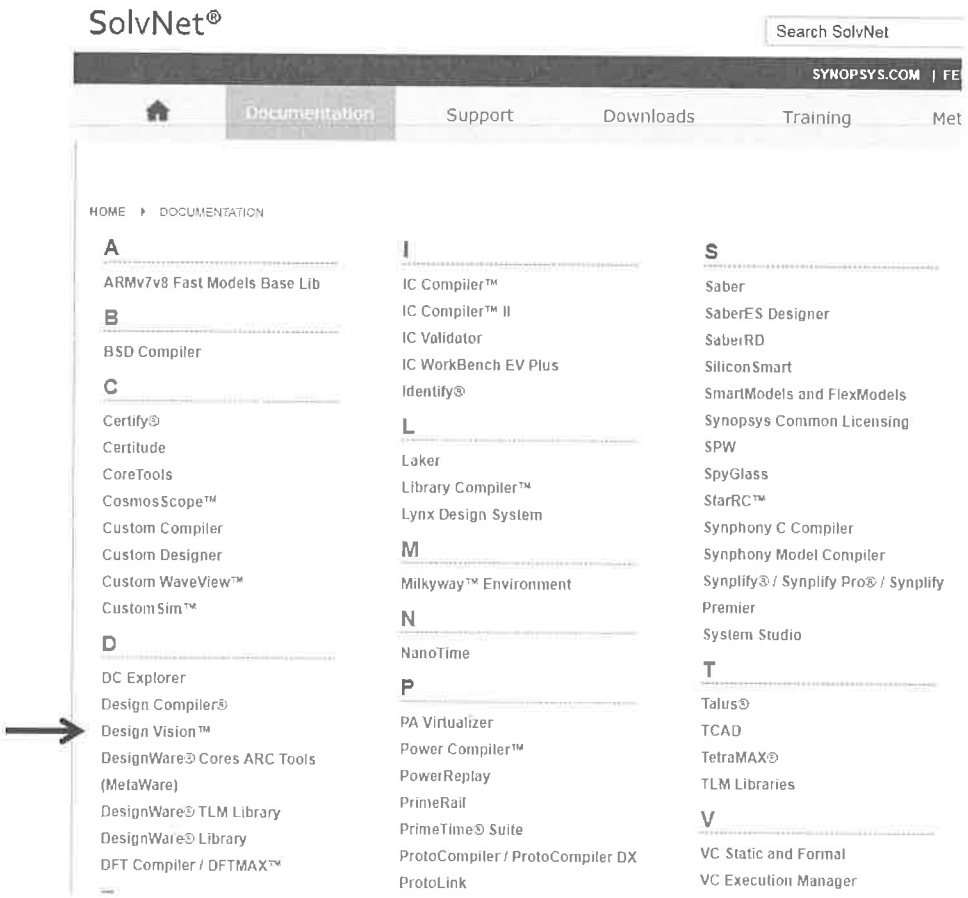
In this task we will pretend that we want to learn how to use Design Vision to print a design schematic. We will browse the on-line documentation to find the needed information. You will need your *SolvNet* ID and password for this task.

1. Open a web browser, enter the URL ***solvnet.synopsys.com***, and log on to *SolvNet* (the link to *SolvNet* is also available from the Synopsys home page):

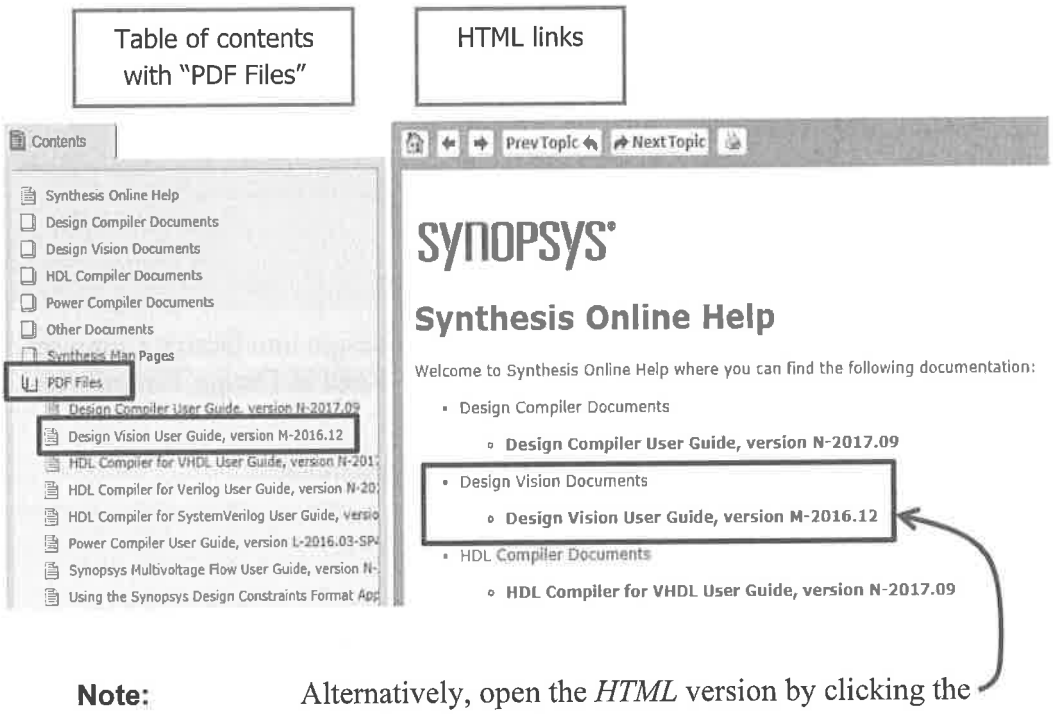
```
unix% firefox &
```

2. You should see several "buttons" at the top of the main page, titled "Documentation", "Support", "Downloads", "Training" etc. Click on "**Documentation**".

3. From the resulting "Documentation" product list, select **Design Vision**.



4. From the left *Contents* tab, expand "**PDF Files**" and open the *Design Vision User Guide*.



Note: Alternatively, open the *HTML* version by clicking the equivalent link on the right.

5. Once the user guide is open, scroll to page vii and select chapter 4, **Performing Basic Tasks**, and then select **Saving an Image of a Window or View**.

   You do not need to save anything - this is just one example of how to use the online documentation.

6. Exit the web browser.

Task 12. (OPTIONAL) Using `analyze` and `elaborate` to read in an HDL design

For this task, you will invoke the tool in the *dc_shell* mode, and you will learn some additional ways to read in HDL files.

1. Invoke DC shell in Topo mode from the `lab1` directory:

```
unix% pwd
unix% dc_shell -topo
```

There are several ways you can read a design into Design Compiler. In the beginning of this lab you used **File → Read** in *Design Vision*, which invokes the `read_verilog/vhdl` command.

2. Read a Verilog file in the *dc_shell* environment:

```
read_verilog ./rtl/TOP.v
current_design TOP
link
```

Note: You can read *VHDL* files using `read_vhdl` and *System Verilog* files using `read_sverilog`.

3. Remove all the designs from the DC memory:

```
fr
```

4. Another method to read a design is to use `analyze/elaborate`. The `analyze` command checks the syntax and synthesizability of the HDL. It also creates a directory called `analyzed`, by default, in which it stores an intermediate binary version of the design. The user can optionally change the directory in which these intermediate files are stored using the `define_design_lib` command (shown on the next page).
The `elaborate` command sets the "current design" to the specified design name (usually the top-level *module* or *entity*), links the design hierarchy, and translates the analyzed results into *ddc*. HDL designs containing parameters can be read in as templates, so that parameters may be redefined during the elaboration stage – an advantage over the `read` command which does not allow this:

```
# Instead of the default "analyzed", write out the
# intermediate files in the directory called "work"
file mkdir ./work
define_design_lib WORK -path ./work
analyze -format verilog -library WORK ./rtl/TOP.v
elaborate TOP
```

Another method to read a design uses a *VCS* style option of the "analyze" command (*VCS* is Synopsys' functional verification tool). This is practical when you have *Verilog* files in different directories and do not know all the sub-design files required to link the design correctly. Here you need to provide only the top module *Verilog* file and `elaborate` will load the other required *Verilog* files by searching specified directory paths.

5. Read the file using the "`-vcs`" option:

```
fr ;# Remove all the designs from DC memory
analyze -vcs "-verilog -y ./rtl +libext+.v" TOP.v
elaborate TOP
```

You have completed the lab.

Answers / Solutions

Question 1. What is the value of the `link_library` variable?

```
* sc_max.db
```

Question 2. What is the value of the `target_library` variable?

```
sc_max.db
```

Question 3. What is the value of the `symbol_library` variable?

```
sc.sdb
```

Question 4. What user directories have been added to the `search_path`?

```
./rtl
./scripts
```

2 Accessing Design and Library Objects

There is NO LAB for this unit

This page is left blank intentionally.

3 Constraints: Reg-to-Reg and I/O Timing

Learning Objectives

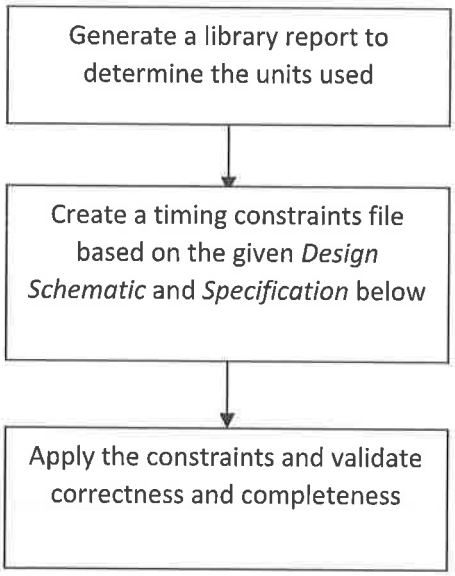
After completing this lab you should be able to:

- Determine the unit of time used in the *target library*
- Create a *Design Compiler* timing constraints file based on a provided schematic and specification
- Verify the syntax of the constraints prior to applying them to a design
- Apply the constraints to a design
- Validate the completeness and correctness of the applied constraints

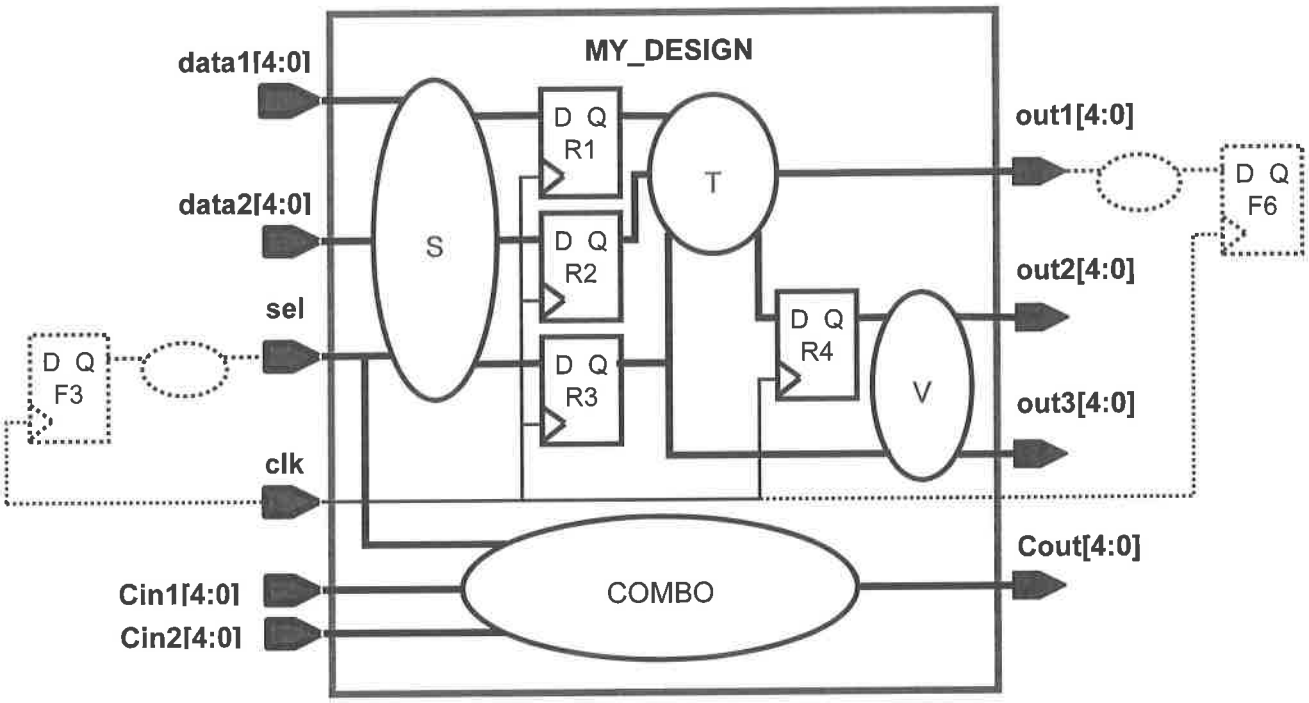
Lab Duration: 80 minutes

Lab Flow

Follow the step-by-step **Lab Instructions** on the following pages to perform the three tasks shown in the lab flow below. Refer to the **Design Schematic** and **Design Specification** sections are needed.



Design Schematic



Design Specification

Hint: **Read carefully!** Some of the specifications are described in "non-DC" language or terms, requiring translation and calculation to derive the DC constraints.

Clock Definition	1. Clock `clk` has a frequency of **333.33 Mhz.** (All inputs and outputs are launched/captured by this same clock – no virtual clocks needed for I/Os) 2. The maximum clock generator delay (located outside `MY_DESIGN`) to the `clk` port is **700ps**. (HINT: *source* latency) 3. The maximum insertion delay from the clock port to all the register clock pins is **300ps +/- 30ps**. (HINT: Treat the 300ps as network latency and the +/-30ps as clock skew) 4. The clock period can fluctuate **+/- 40ps** due to jitter. 5. Apply **50ps** of "setup margin" to the clock period. 6. The worst case rise/fall transition time of any clock pin is **120 ps**.
Register Setup Time	Assume a maximum setup time of **0.2ns** for any register in MY_DESIGN
Input Ports (sequential logic)	1. The maximum delay from ports `data1` and `data2` through the *internal input logic* `S` is **2.2ns**. 2. The latest *F3 data arrival time* at the `sel` port is **1.4ns** absolute time. (HINT: Input delay is specified relative to the launching clock edge)
Output Ports (sequential logic)	1. The maximum delay of the *external* combo logic at port `out1` is **420ps**; `F6` has a setup time of **80ps**. 2. The maximum *internal* delay to `out2` is **810ps** 3. The `out3` port has a **400ps** setup time requirement with respect to its capturing register clock pin.
Combinational Logic	The maximum delay from `Cin1` and `Cin2` to `Cout` is **2.45ns**. (HINT: Use appropriate input and output delay constraints with respect to clock `clk`)

Lab Instructions

Task 1. Determine the Target Library's *Time Unit*

1. Change to the **lab3** UNIX directory.
2. Using a text editor or viewer look at the **common_setup.tcl** file to answer this question:

   Question 1. What is the *target library* file name (specified using the TARGET_LIBRARY_FILES variable)?

   .....

3. Exit the text editor or viewer.
4. Invoke *Design Compiler Topographical* from the **lab3** directory:

```
UNIX% dc_shell -topo | tee -i lab3.log
```

5. Normally the *target* and *link libraries* are loaded into *Design Compiler* memory when a design is read in (with read_verilog, read_vhdl, read_ddc, or analyze/elaborate). In this specific instance we want to load the target library in DC memory so we can determine the unit of time, without loading a design. This can be done as follows:

```
dc_shell-topo> read_db <target_library_FILE_NAME>
```

   Note: You can ignore the warning about "Overwriting design file .../sc_max.db"

   Note: Alternatively, once a design and its libraries arel oaded in DC memory, you can use report_units.

6. Determine the *library* name associated with this library *file*:

```
dc_shell-topo> list_libs
```

   Question 2. What is the target *library* name?

   .....

7. Generate a library report file for the above library:

```
redirect -file lib.rpt {report_lib <LIBRARY_NAME>}
```

8. Use a text editor or viewer to look at the top portion of the **`lib.rpt`** file, and answer the following question:

   Question 3. What is the "`Time Unit`" of the target library?

   .....

9. Exit the text editor or viewer.

10. Exit *Design Compiler*:

```
dc_shell-topo> exit
```

Task 2. Create a Timing Constraints File

1. In the **`scripts`** directory use a text editor to create a new file called **`MY_DESIGN.con`**.

   Question 4. What is the recommended first command for any constraint file?

   .....

2. Using the **Design Specification** and **Design Schematic** on the previous pages, as well as the appropriate time unit, enter the required constraints in `MY_DESIGN.con`.

   Note: Use the ***Job Aid*** and DC's **`help`** and **`man`** commands as needed. Refer to the solution file to check your constraints: **`.solutions/MY_DESIGN.con`**.

   Note: To use DC's `help` and `man` you will need to invoke the DC shell. It is also possible to access the *Design Compiler* "man" pages without invoking DC: The recommended approach is to create a separate UNIX alias, ***dcman***, for example:

```
UNIX% alias dcman '/usr/bin/man -M $SYNOPSYS/doc/syn/man'
UNIX% dcman create_clock
```

   Note: Exit the man page by typing '**`q`**', the lower-case [Q] key.

3. After completing the constraints file, check your constraint syntax, and correct as necessary. The absence of any specific syntax message indicates that the file is syntax clean:

```
UNIX% dcprocheck scripts/MY_DESIGN.con
```

Note: `dcprocheck` is a syntax checking utility that is included with the *Design Compiler* executable. It is available if you are able to launch *Design Compiler* – no additional user setup is required.

Task 3. Apply Constraints and Validate

1. Before invoking *Design Compiler*, type **`ls`** in the **`lab3`** directory and notice the sub-directory called `MY_DESIGN_LIB`: This is the *Milkyway* ***design library***, which was created when you first invoked *Design Compiler* in Task 1.
2. Now invoke the *DC Topographical shell* from the **`lab3`** directory.

   Question 5. Why do you NOT get an error message about trying to create a *Milkyway* design library that already exists?
   (HINT: Look at the `dc_setup.tcl` file)

   .....

   .....

3. ***Read*** the *Verilog RTL* design file **`rtl/MY_DESIGN.v`**.
4. Perform the following "good practice" steps: Ensure the ***current design*** is **`MY_DESIGN`**, then ***link*** and ***check*** the design.

   Note: Use the ***Job Aid*** and DC's **`help`** and **`man`** commands, or refer to **`.solutions/dc.tcl`**, as needed.

5. Apply the constraints file and make any corrections as needed:

```
source scripts/MY_DESIGN.con
```

   Note: If any errors or warnings are reported, re-run the above command using the **`-echo`** option, to help you identify the command(s) to fix.

6. Check that there are no missing or conflicting key constraints – correct as needed:

```
check_timing
```

Note: You can ignore the message "`Warning: there are 21 input ports that only have partial input delay specified. (TIM-212)`". It appears if a `set_input_delay` command has a `-max` option without a corresponding `-min` option. Since we are only constraining for *setup* (`-max`) timing, the warning can be ignored.

7. Verify the clock and port constraints – correct as needed.

   You should see one clock, with the correct *period* and *waveform*, applied to the *source* (port or pin) called `clk`:

```
report_clock
```

   You should see the correct maximum network latency (*Rise Delay/Fall Delay*), setup (*Minus*) uncertainty, max source latency, and max transition:

```
report_clock -skew
```

   You should see correct input and output delays:

```
report_port -verbose
```

Note: The other port report sections will show default (empty or 0) values. Some of these will be modified during later labs.

8. Write out the applied constraints, in expanded form, to a file for further checking:

```
write_script -out scripts/MY_DESIGN.wscr
```

9. Ensure that your constraints are complete and correct by perfoming a UNIX *diff* between your write-script file, and the provided solution file – correct as needed:

   Note: You may see different values for your input and output delays on ports Cin* and Cout* compared to the solution. What matters for these combinational logic path constraints is that the sum of their input and output delays match.

```
UNIX% tkdiff scripts/MY_DESIGN.wscr
.solutions/MY_DESIGN.wscr
  OR
UNIX% diff scripts/MY_DESIGN.wscr
.solutions/MY_DESIGN.wscr
```

10. If the above step uncovers differences which you do not understand, take a look at **`.solutions/MY_DESIGN.con`**: This file contains comments explaining how each constraint was determined.

11. Save the *unmapped* design and exit *Design Compiler*:

```
write_file -format ddc -hier \
           -out unmapped/MY_DESIGN.ddc
exit
```

You have completed the lab.

Answers / Solutions

Question 1. What is the *target library* file name? (specified using the `TARGET_LIBRARY_FILES` variable)?

`sc_max.db`

Question 2. What is the target *library* name?

`cb13fs120_tsmc_max`

Question 3. What is the "Time Unit" of the target library?

1ns

Question 4. What is the recommended first command for any constraint file?

```
reset_design
```

Note: If there are multiple constraint files which are sourced sequentially, then `reset_design` should be used only once, at the top of the constraint file which is first applied.

Question 5. Why do you NOT get an error message about trying to create a *Milkyway* design library that already exists? (HINT: Look at the `dc_setup.tcl` file)

In the `dc_setup.tcl` file, which is *sourced* upon DC start-up by the `.synopsys_dc.setup` file, there is an "`if`" statement which checks for the existence of a *Milkyway* design library, and skips the `create_mw_lib` command if it already exists. Without this check, the `create_mw_lib` command would have generated the following message:
"`Error: Library 'MY_DESIGN_LIB' already exists. (MWUI-004)`".

4

Constraints: Input Transition and Output Loading

Learning Objectives

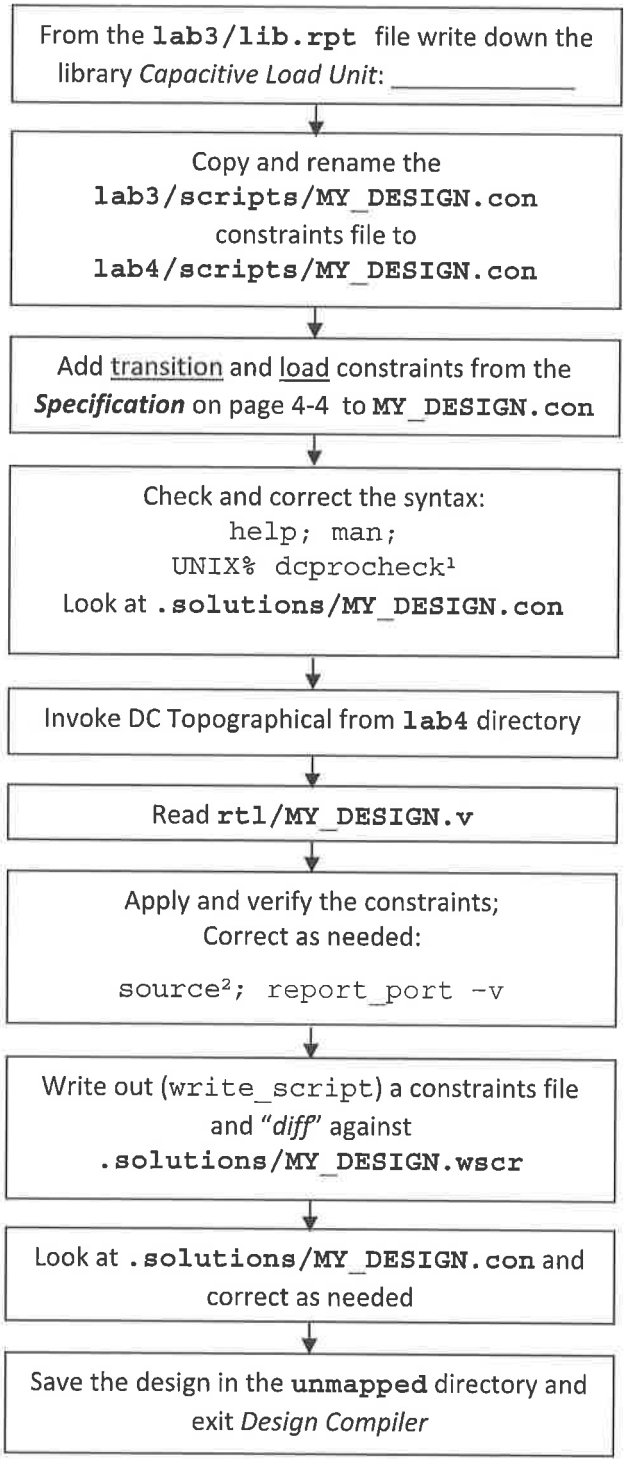
After completing this lab you should be able to:

- Apply and verify input transition and capacitive output loading constraints

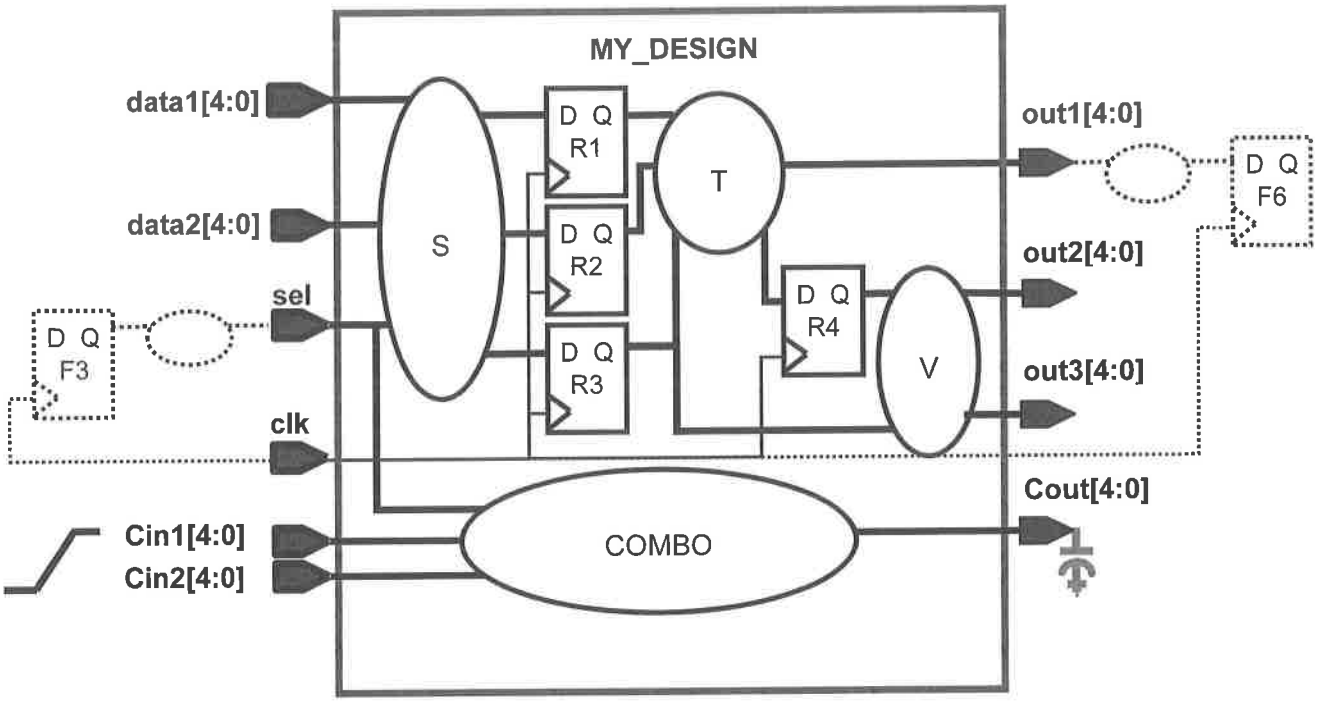
Lab Duration: 45 minutes

Lab Instructions

Perform the steps below. Refer to the *Lab 3* step-by-step instructions as needed.



Design Schematic



Design Specification

Hint: Use the `lib.rpt` file in `lab3` to obtain some of the information needed to apply these specs.

Input Ports (transition)	1. Specify a *driving cell* called `bufbd1` on all inputs, except `clk` as well as `Cin*`. 2. The `Cin*` ports are chip-level inputs and have a **120ps** maximum input transition.
Output Ports (capacitive load)	1. All outputs, except `Cout`, drive a maximum load equivalent to **2** times the capacitance of the "`I`" pin of the cell `bufbd7` (see **Note** below). 2. The `Cout` port drives a maximum load of **25 fF**.

Note: [1] You may get the syntax warning below if you run `dcprocheck`. The checker is overly "picky" compared to *Design Compiler* in this instance – you can ignore the warning:

```
... use curly braces to avoid double substitution
        expr 2 * [load_of lib/cell/pin]
                 ^
```

To avoid the warning enter:

```
expr {2 * [load_of lib/cell/pin]}
```

Note: [2] The warning "`Design rule attributes from the driving cell will be set on the port.`" can be ignored – it is letting you know that `set_driving_cell` also applies any DRCs of the driving cell to the input ports.

You have completed the lab.

5 DC Ultra Synthesis Techniques

Learning Objectives

After completing this lab, you should be able to:

- Apply the recommended *DC Ultra* synthesis techniques to meet the required design specifications
- Verify applied directives and variables before compile
- Analyze the gate-level netlist to:
  a) Ensure that all design constraints have been met
  b) Observe the results of the various optimization techniques invoked
- Use the *LayoutWindow* to verify the applied physical (floorplan) constraints and view the cell placement
- Perform *formal verification* on the synthesized netlist versus the *RTL* design using *Formality*

Lab Duration: 75 minutes

Lab Overview

The objective of this lab is to compile a design called STOTO using the information listed in the **Synthesis Specification** table on page 5. You will accomplish this by following the step-by-step **Lab Instructions** starting on page 6, which will guide you in performing the following tasks:

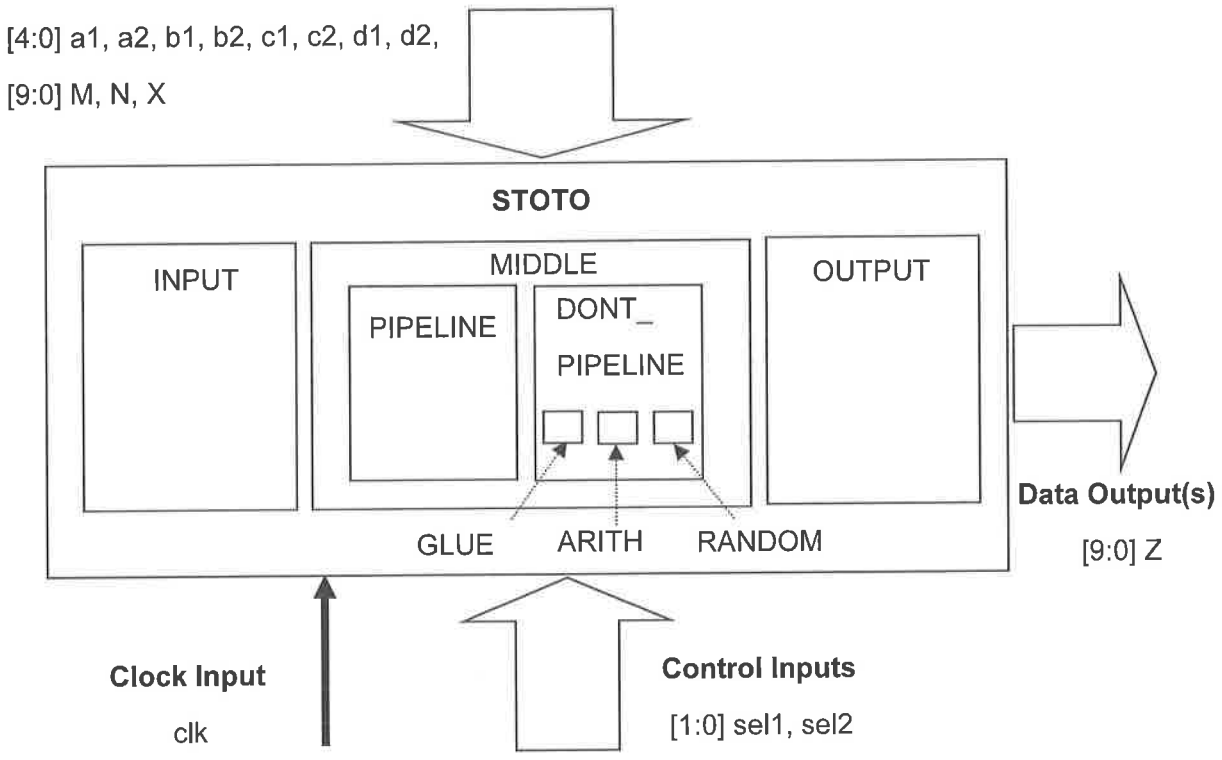
- Use the *DC Topographical* shell interactively to read in and constrain the design
- Interactively apply and execute the appropriate compile flow techniques and commands, to achieve the stated **Design Specification**
- Verify the applied compile directive commands and variables before each compile or optimization
- Analyze the results after each compile or optimization to determine what step, if any, to perform next
- While performing all the above steps interactively you will also create a "run script" so that your steps can be easily corrected and re-applied as necessary
- Use the *LayoutWindow* to verify that the applied physical (floorplan) constraints were honored during compile, and to view the cell placement
- Optionally, perform equivalence checking of the compiled netlist to RTL using *Formality*

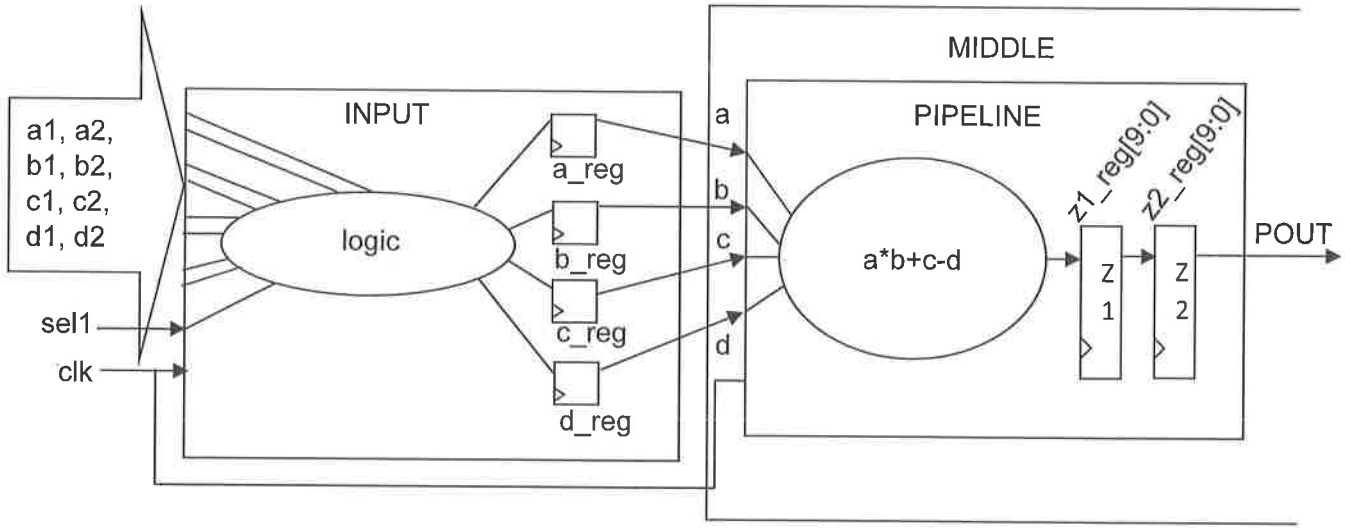
Design Schematic

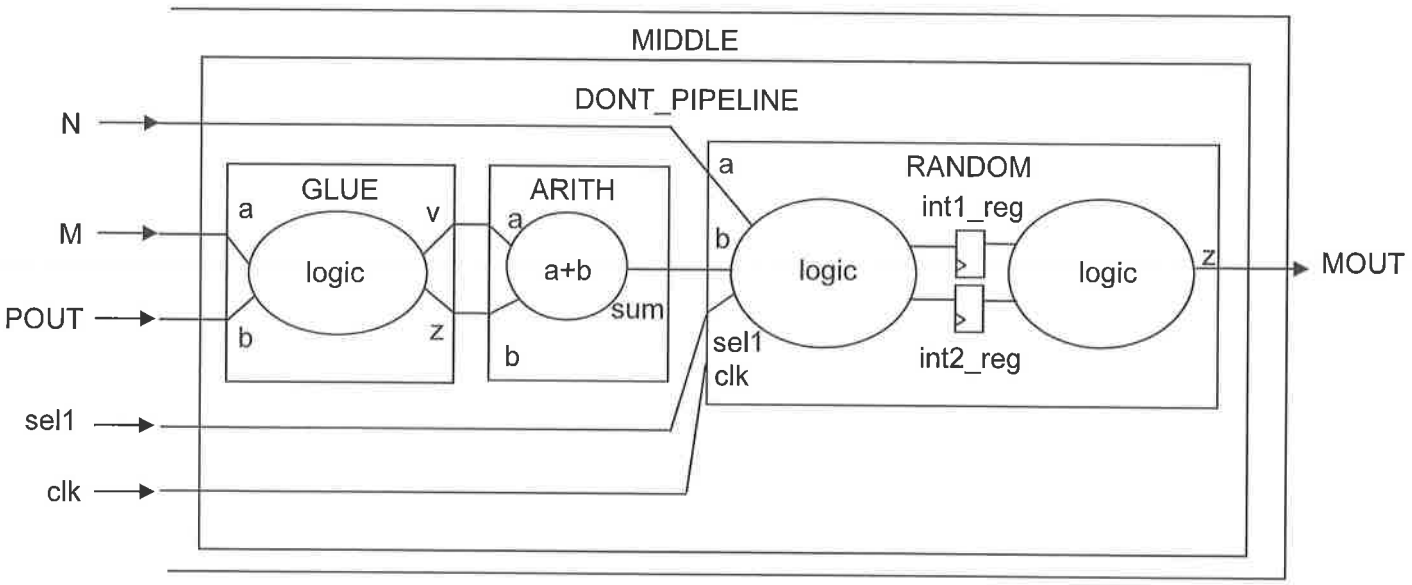
You are provided with the RTL code of the design (`rtl/STOTO.v`) and the design constaints (`scripts/STOTO.con`).

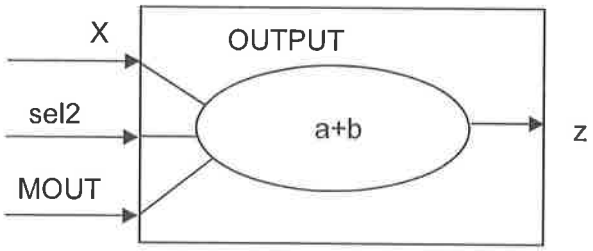
RTL design hierarchy:

Design Name	Cell Name
STOTO	
INPUT	(I_IN)
MIDDLE	(I_MIDDLE)
PIPELINE	(I_PIPELINE)
DONT_PIPELINE	(I_DONT_PIPELINE)
GLUE	(I_GLUE)
ARITH	(I_ARITH)
RANDOM	(I_RANDOM)
OUTPUT	(I_OUT)




a1, a2,
b1, b2,
c1, c2,
d1, d2
sel1
clk
INPUT
logic
a_reg
b_reg
c_reg
d_reg
MIDDLE
a
b
c
d
PIPELINE
a*b+c-d
z1_reg[9:0]
z2_reg[9:0]
Z 1
Z 2
POUT


MIDDLE
DONT_PIPELINE
N
M
POUT
sel1
clk
GLUE
a
b
logic
v
z
ARITH
a
b
a+b
sum
a
b
RANDOM
int1_reg
int2_reg
logic
logic
sel1
clk
z
MOUT


X
sel2
MOUT
OUTPUT
a+b
z

Synthesis Specification

Available Resources	1. You will use a provided script, `num_core.sh`, to determine how many cores/threads to use (up to 8) 2. All Design Compiler features and related licenses are available (1 license per feature)
Design and Constraints Files	1. RTL code location: **`rtl/STOTO.v`** 2. Design to be compiled: **`STOTO`** 3. Constraints file location: **`scripts/STOTO.con`** 4. These files should not be modified
Floorplan	Final floorplan definition: 1. The core size is 150 um X 100 um 2. `STOTO` has no instantiated hard macros 3. Standard cells can be placed anywhere inside the core area except for a 20 um X 20 um area in the upper-left corner, which is reserved for a different block
Design Specification	1. The I/O constraints are estimates and have been conservatively constrained 2. The final compiled design must meet setup timing on all internal register-to-register paths 3. The **`INPUT`** block hierarchy should be preserved to facilitate post-synthesis verification 4. The **`PIPELINE`** block contains a "pure" pipelined design 5. The output (**`POUT[9:0]`**) of **`PIPELINE`** must remain registered 6. The logic positions of registers in the instance or *cell* **`I_DONT_PIPELINE`** are fixed and cannot be modified 7. DRC violations should be fixed unless they prevent setup timing constraints from being met 8. The logic position of registers may be modified to improve timing, except where expressly prohibited or handled differently by above specs 9. Scan insertion will be performed by the "Test group" after the design has met these specifications

Lab Instructions

Task 1. Synthesis of RTL design containing pipelines

1. Using the provided `num_cores.sh` script, determine how many cores and threads are available on your machine:

```
UNIX% ./num_cores.sh
```

Question 1. How many cores/threads are available on your machine?

.....

Note: You are encouraged to check your answers against the ***Answers/Solutions*** section at the end of the lab.

2. Look at the **Available Resources** section of the **Synthesis Specification** table on the (previous page) to answer the following question:

Question 2. How many cores/threads will you be able to use to compile your design?

.....

Note: You will use the above information in an upcoming step to enable multi-core optimization prior to compiling the design

3. Invoke *DC Topographical* from the `lab5` directory.

4. Create a new file called `dc.tcl` in the `scripts` directory. For each of the following steps, whenever you apply a command interactively in the DC-shell environment, copy and paste the command into the `dc.tcl` file.

5. In your *Job Aid*, locate the **Run Script** section and the **Compile Flow** section. Refer to these sections while performing each of the following steps.

6. Before reading in the design, specify an *SVF* file name for *Formality* so that all the retiming (and other *Ultra*) transformations can be captured into a file named `STOTO.svf`:

```
set_svf STOTO.svf
```

7. Refer to the **Design and Constraints Files** section of the **Synthesis Specification** table (page 5) to do the following, interactively, in the DC-shell environment: read, link and check the design `STOTO` (not "`WRONG_DESIGN`"!), then source and check the timing constraints.

   Note: The `check_timing` warning message `TIM-212` about "`74 input ports that only have partial input delay specified`" can be ignored: We have specified a `-max` delay option without its `-min` counterpart, since we do not worry about *hold* timing during synthesis.

   Note: Remember to also "copy and paste" the commands into your run script file `dc.tcl`!

8. Refer to the **Floorplan** section of the **Synthesis Specification** table. Source the following file to apply the specified final floorplan physical constraints:

   Note: Since the floorplan definition is final, treat the synthesis to be performed in this lab as "2nd pass synthesis".

```
source STOTO.pcon
```

9. **Interactively apply** the appropriate commands, up until but NOT including *compile*, to address **Design Specification** #1 through #7.

   Remember to copy and paste into your run script.

   Note: Refer to your *Job Aid* sections called **Run Script** and **Compile Flow**, and to `.solutions/dc.tcl`, as needed.

10. Verify that the appropriate physical constraints, directives and attributes have been applied prior to *compile*:

```
report_physical_constraints
report_path_group
get_attribute [get_designs "INPUT"] ungroup
get_attribute [get_designs "PIPELINE"] \
        optimize_registers
get_attribute [get_cells \
        I_MIDDLE/I_PIPELINE/z2_reg*] dont_retime
get_attribute [get_cells \
        I_MIDDLE/I_DONT_PIPELINE] dont_retime
get_attribute [get_designs "STOTO"] cost_priority
```

Question 3. Are you seeing the expected results for each check?

.....

.....

.....

.....

.....

Note: Check your results against the answers in the *Answers/Solutions* section at the end of this lab. If you do not see the expected results modify your applied commands until you do.

11. Save your design as `unmapped/STOTO.ddc`.
12. Apply the appropriate command to enable **multi-core optimization**, based on your answer to Question #2 (assuming more than 1 core is available).
13. Look at **Design Specification** #8 and #9, and answer the following question:

    Question 4. What `compile_ultra` options will you apply?

    .....

14. **Compile** the design using the options identified in the previous question.
15. While you are waiting for the compile to finish, use the **Design Schematic** on pages 3 and 4 to answer the following two questions:

    Question 5. Is the design well partitioned for synthesis? Explain.

    .....

    .....

    .....

    Question 6. Which sub-designs would need to be ungrouped to obtain ideal partitioning for synthesis?

    .....

    .....

16. After the *compile* is completed, use the compile log "`Information`" messages to answer the following questions:

Question 7. What information message code confirms that multi-core optimization (if specified) was honored? (HINT: Look near the top of the log)

.....

.....

Question 8. What is an indication that the design is being retimed?

.....

.....

Question 9. What design hierarchies are being auto-ungrouped during `compile_ultra`? Do these match your requirements?

.....

.....

.....

17. You can also verify the design's remaining hierarchy with:

```
report_hierarchy -noleaf
```

18. Generate a constraint report (`rc`) to the screen as well as to a file:

```
redirect -tee -file rc_compile_ultra.rpt {rc}
```

Question 10. Are there any register-register setup timing violations? If so, describe them.

.....

.....

.....

Question 11. Should you be concerned about the other violations? Explain.

.....

.....

.....

.....

.....

.....

19. Record the setup timing *WNS* or *worst negative slack* for the `INPUTS` group:

    `max_delay/setup` timing WNS for `INPUTS`: _____

20. Generate a timing report (`rt`):

```
redirect -tee -file rt_compile_ultra.rpt {rt}
```

21. Look at the "**`Startpoint`**" of the **`OUTPUTS`** group. Notice that the startpoint "**`I_MIDDLE/I_DONT_PIPELINE/I_RANDOM`**..." contains the instance names of sub-designs that were auto-ungrouped!

    When ungrouping, *Design Compiler* keeps the original hierarchical path name to a child cell and converts it into a non-hierarchical cell name. This register start-point is actually a leaf cell in the top-level design `STOTO`!! The "slash" no longer denotes a hierarchy separator but is just a character that is part of the cell's new, longer name. This makes it easy to "trace" ungrouped cells to know where they originally came from.

22. Save your design as **`mapped/STOTO.ddc`** for *IC Compiler*.

23. Stop recording design changes in the *SVF* file for *Formality*:

```
set_svf -off
```

Next we are going to determine if registers were affected by *adaptive retiming* or by *register retiming*.

24. The cell name of a pipeline register that has been moved or repositioned by *register retiming* is re-named: **`clockname_r_REG#_S#`**. If the following command returns any cells, this confirms that these registers were moved by *register retiming*:

```
get_cells -hier *r_REG*_S*
```

Look at the returned cells to answer the following questions.

Question 12. In what sub-design are these retimed registers?

.....

Question 13. How can you verify that the output registers in **`PIPELINE`**, called **`z2_reg*`**, were not moved?

.....

.....

25. Non-pipeline registers moved by *Adaptive Retiming* are renamed **`R_##`**.

Enter the following command. If it returns any cells, this confirms that these registers were moved by *adaptive retiming*:

```
get_cells -hier R_*
```

Look at the returned cells to answer the following questions.

Question 14. Were registers in the **`INPUT`** sub-design affected by *adaptive retiming*, and if so, is this due to the `-retime` option?

.....

.....

Question 15. Were registers in the **`PIPELINE`** design affected by *adaptive retiming*, and if so, is this due to the `-retime` option?

.....

.....

Question 16. Were ALL registers in the `INPUT` sub-design affected by *adaptive retiming*?

.....

.....

26. Refer to the right-most section of the "**Compile Flow**" section of the *Job Aid*, after the "**Compile**" step, to answer the following two questions:

Question 17. What are the next suggested steps if there are still violations after the initial compile?

.....

.....

.....

.....

.....

Question 18. Based on **Design Specification #1** and **#2**, does it make sense to perform these steps?

.....

.....

.....

.....

.....

.....

Next you will bring up the *layout window* (discussed in more detail in a later unit) to view the floorplan and cell placement.

27. Report the physical constraints, which were applied from `scripts/STOTO.pcon`:

```
report_physical_constraints
```

28. Start the *DesignVision* GUI:

```
dc_shell-topo> gui_start
```

29. Open a *LayoutWindow* using the pull-down menu:
Window → New Layout Window

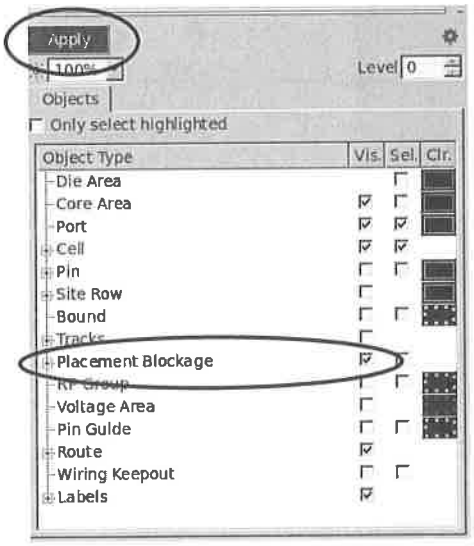
You should see a rectangular shape, which represents the core placement area, as well as the input/output pin locations along the boundary of the core. The placed cells are not visible, by default.

30. Place your cursor near the upper-right corner of the core boundary and look at the (X,Y) coordinates in the bottom-right banner of the *LayoutWindow*.

Question 19. Does the size of the core area match the physical constraint applied in `./scripts/STOTO.pcon`?

.....

31. You should see the *View Settings* control form, shown below, on the left side of the *LayoutWindow*. If you do not, press the F8 key. In the **Object Type** list, check the **Vis.** (Visible) box next to **Placement Blockage** and **Apply**:



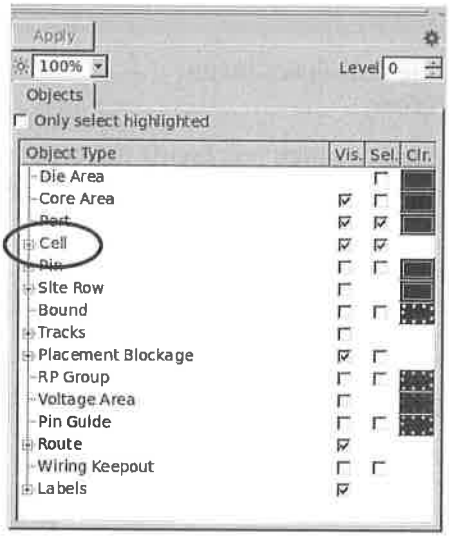
The placement blockage is now visible in the upper-left corner of the placement area.

Note: You can toggle between showing and hiding the *View Settings* form by using the pull down command *View→Toolbars→View Settings* or the F8 key.

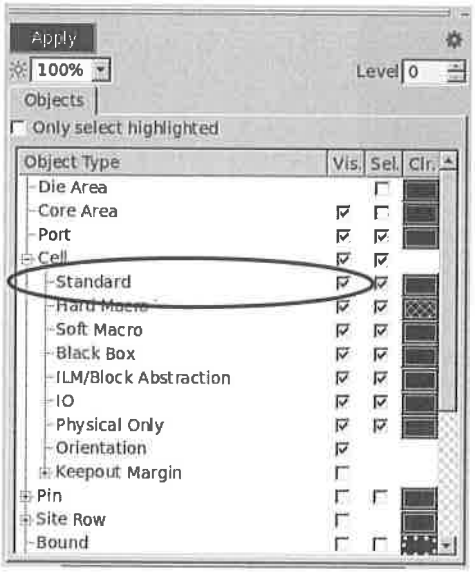
Question 20. Does the location of the placement blockage match the physical constraint applied in `./scripts/STOTO.pcon`?

.....

32. Expand the **Cell** objects by clicking on the "+" icon:



33. Check the "**Vis.**" (Visible) and "**Sel.**" (Selectable) boxes next to "**Standard**" and *Apply*.



The standard cells should now be visible. You can use the "**Z**" and "**F**" keys to zoom in and fit the layout view. Pres **Esc** to exit the zoom mode.

You will notice that many standard cells overlap, and their placement looks "un-organized". Standard cells must eventually be placed in *legal* placement locations (no overlapping) inside *placement rows* (usually running horizontally), so their top and bottom edges align perfectly. This is not the case here because *DC-Topo* performs only *coarse placement*, for quicker placement – it does not perform placement *legalization*. Coarse placement is good enough for purposes of estimating the interconnect or net parasitic R/C's, which is the main goal of *DC-Topographical* synthesis.

34. Exit Design Compiler.

If you have a few extra minutes and the interest, perform the optional *Formal Verification* task on the following page.

Task 2. OPTIONAL: Formal Verification

In this task, you will invoke *Formality* (Synopsys' *Formal Verification* tool) to verify that your RTL design (the "Reference") and the optimized netlist (the "Implementation") are functionally equivalent.

1. Open the `./scripts/fm.tcl` file to get an understanding of the *Formality* flow.

   Question 21. Which file contains *SVF* information written out by DC?

   .....

   Question 22. Which files are the "Reference" and "Implementation" design files that are being verified for functional equivalence?

   .....

2. From the `lab5` directory invoke the *Formality* shell:

```
UNIX% fm_shell
```

3. Execute the **`fm.tcl`** script:

```
fm_shell (setup)> source -echo ./scripts/fm.tcl
```

   Question 23. Does verification succeed or fail?

   .....

   Note: If the verification failed, check if the *svf* file has been read in successfully.

A successful verification means that the gate-level netlist generated by synthesis is logically equivalent to the original RTL design, which is the purpose of equivalency checking. Your work is done!

If you want to get an idea of the changes or transformations that DC performed and were captured in the *SVF* file, look at the last table in the log (generated by the command `report_guidance -summary`). For example, from the listed guidance commands "`retiming`" and "`ungroup`" you know that registers have been retimed, and sub-designs have been ungrouped. These RTL-to-netlist changes are passed on to *Formality*, in the form of "guidance commands", to help *Formality* in verifying logical equivalence. The guidance commands are applied to the "reference" input (the RTL design, in this case)

during the "matching" phase. During the "verification" phase *Formality* compares the functionality between the modified reference input (modified by "accepted" guidance commands) and the "implementation" input (the gate-level netlist, in this case). If the reference input and the implementation input are logically equivalent, the verification passes.

Note that an "accepted" guidance command means that *Formality* was able to verify that a specific transformation performed by DC was logically valid. Your netlist can still fail logical equivalency verification with 100% accepted guidance commands, and conversely, verification can pass even if there are some rejected guidance commands.

If you run *Formality* without providing the *SVF* file, the tool may not be able to verify equivalency.

For more information on *Formality* please refer to the *Formality User's Guide*, available on-line through *SolvNet*.

4. Exit *Formality*.

You have now seen how *Ultra* optimization techniques such as register retiming and adaptive retiming, auto-ungrouping, and datapath optimization can be effectively used to improve timing in a timing-critical design. With the help of the provided *Job Aid* you should now be able to apply the recommended synthesis flow and techniques to any real-world design!

You have completed the lab.

Answers / Solutions

Question 1. How many cores and threads are available on your machine?

Your answer may be different. In the following example, 4 cores and the same number of threads are available:

```
Number of CPU cores:
cpu cores       : 4

Can have 1 or 2 threads per core.
Number of threads:
4
```

Question 2. How many cores/threads will you be able to use to compile your design?

Since you have one license available, this enables you to use up to 8 cores/threads during compile. If the cores have multi-threading enabled, you can specify the # of threads (up to 8) for '`set_host_options -max_cores`'.

Question 3. Are you seeing the expected results for each check?

`report_physical_constraints` should report the *die area* and *placement blockage* location defined in `scripts/STOTO.pcon`.

`report_path_group` should confirm that there are 4 path groups, in addition to the `**default**` group: an input, output, combinational and `clk` path group. The `clk` group should have a *weight* of `5.00` and a *critical range* of `0.21` (10% of the clock period of 2.1ns, from `report_clock`), while the other groups should have `1.00` and `0.00`, respectively. (Spec #1 and #2)

`get_attribute [get_designs "INPUT"] ungroup` should return "**`false`**" as the value of the ungroup attribute. (Spec #3)

`get_attribute [get_designs "PIPELINE"] \ optimize_registers` should return "**`true`**" as a result of applying `set_optimize_registers` on the "`PIPELINE`" block. (Spec # 4)

`get_attribute \ [get_cells I_MIDDLE/I_PIPELINE/z2_reg*] \ dont_retime` should return "**`true true true true true true true true true true`**" as a result of using the `set_dont_retime` command on these registers. (Spec #5)

`get_attribute \ [get_cells I_MIDDLE/I_DONT_PIPELINE] \ dont_retime` should return "**`true`**" as a result of using the `set_dont_retime` command on the "`I_DONT_PIPELINE`" instance or *cell*. (Spec # 6)

`get_attribute [get_designs "STOTO"] \ cost_priority` should return "**`max_delay`**" as a result of applying `set_cost_priority`. (Spec #7)

Question 4. What `compile_ultra` options will you apply?

`compile_ultra` **`-spg -retime`**

From #8 "The logic position of registers may be modified to improve timing ….", you should use **`-retime`**, to enable *adaptive re-timing* on all non-pipelined registers (except the ones with a `dont_retime==true` attribute)

From #9 "Scan insertion will be performed by the "Test group" after the design has met these specifications", you should enable test-ready synthesis with the **`-scan`** option.

Question 5. Is the design well partitioned for synthesis? Explain.

No! Logic optimization will be restricted at the interfaces between the following sub-designs, due to hierarchical partitioning:

```
GLUE   → ARITH
ARITH  → RANDOM
RANDOM → OUTPUT
```

Question 6. Which sub-designs would need to be ungrouped to obtain ideal partitioning for synthesis?

`MIDDLE`, `DONT_PIPELINE`, `GLUE`, `ARITH`, `RANDOM` and `OUTPUT`.

Question 7. What information message code confirms that multi-core optimization (if specified) was honored? (HINT: Look near the top of the log)

```
Information: Running optimization using a
maximum of 4 cores. (OPT-1500)
```

If you enter "`list_licenses`", it will list the licenses which have been checked out:

`BOA-BRT` (required for register and adaptive retiming)

`DC-Expert`

`DC-Extension` (required for `-spg`)

`DC-Ultra-Features` (required for `compile_ultra`)

`DC-Ultra-Opt` (required for `compile_ultra`)

`Design-Compiler`

`DesignWare` (required for `compile_ultra`)

`HDL-Compiler`

`Milkyway-Interface`

`Test-Compiler` (required for `-scan`)

Question 8. What is an indication that the design is being retimed?

```
Information: Retiming is enabled. SVF file
must be used for formal verification. (OPT-
1210)
```

This refers to either register or adaptive retiming.

Also, after the beginning of "**`Mapping Optimizations (Ultra High effort)`**" there are "**`Retiming`**" messages indicating the register retiming steps being performed on the **`PIPELINE`** design.

Question 9. What design hierarchies are being auto-ungrouped during `compile_ultra`? Do these match your requirements?

`OPT-776` messages should confirm that the six designs listed in *Question 6* are ungrouped before "`Pass 1 Mapping`". Per the spec, the "`INPUT`" block should not be ungrouped.

Question 10. Are there any register-register timing violations? If so, describe them.

If all the appropriate optimizations were enabled, there should not be any **`max-delay`** or **`setup`** timing violations in the **`clk`** (or register-register) path group. Setup timing violations may exist in I/O path groups (e.g. `INPUTS`).

Question 11. Should you be concerned about the other violations? Explain.

Since, according to the **Design Specification**, the I/O constraints are conservative, and the goal is to meet reg-to-reg timing, the max-delay violations are not critical (Specs #1 and #2).

Question 12. In what sub-design are these retimed registers?

The cell names all begin with **`I_MIDDLE/I_PIPELINE/clk*`**, which indicates that the retimed registers are in the `PIPELINE` sub-design, or "reference". You can confirm this "cell name" versus "reference name" relationship with `report_cell -nosplit I_MIDDLE/I_PIPELINE`

Question 13. How can you verify that the output registers in **`PIPELINE`**, called **`z2_reg*`**, were not moved?

Since all the register cells end with **`_s1`**, this means that only **`z1_reg`**, the first stage registers, were moved (otherwise there would also be **`_s2`** register names).

This can be further verified with additional checks: `get_cells -hier *z2_reg*` shows that the original register names still exist, which means that *register retiming* did not move them.

Question 14. Were registers in the **INPUT** sub-design affected by *adaptive retiming*, and if so, is this due to the `-retime` option?

The answer to both questions is "**yes**".
The `get_cells -hier R_*` command returns registers called `I_IN/R_*`. We can confirm that **I_IN** is the cell or instance name of the **INPUT** sub-design with `report_cell -nosplit I_IN` (shows cell versus reference name relationship). Furthermore, we know that the **INPUT** registers were moved by *adaptive retiming* because the register cells are named **R_#**, which follows the *adaptive retiming* re-naming convention.

Question 15. Were registers in the **PIPELINE** design affected by *adaptive retiming*, and if so, is this due to the `-retime` option?

No, there are no registers called **R_#** in the **PIPELINE** design. If there were, this would mean that they were moved by adaptive retiming as well. However, this adaptive retiming would NOT have been due to `-retime`, but due to register retiming. The final optimization phase of register retiming includes an implicit *adaptive retiming* phase.

Question 16. Were ALL registers in the **INPUT** sub-design affected by *adaptive retiming*?

No! Using `get_cells I_IN/*_reg*` you can confirm that some of the registers still have their original register names, which confirms that they were NOT affected by *adaptive retiming*.

Question 17. What are the next suggested steps if there are still violations in the register-to-register paths?

1) Apply more focus on violating critical paths, as necessary: `group_path -weight -critical`

2) Perform an *incremental* compile:
`compile_ultra -scan -retime` **`-incremental`**

Question 18. Based on **Design Specification #1** and **#2**, does it make sense to perform these steps?

We have already met the stated **Design Specification #2** that all reg-to-reg setup timing must be met. Since the I/O constraints are estimates and have been conservatively constrained (**Design Specification #1**), it does not make sense to spend any more effort on these I/O violations.

Question 19. Does the size of the core area match the physical constraint applied in `./scripts/STOTO.pcon`?

It should match the die area (= core area, by default) coordinates (150x100 rectangle) listed in the physical constraints report.

Question 20. Does the location of the placement blockage match the physical constraint applied in `./scripts/STOTO.pcon`?

It should match the KEEPOUT LOCATION coordinates {(0, 80), (20, 100)} listed in the physical constraints report.

Question 21. Which file contains SVF information written out by DC?

`STOTO.svf`

The file name was specified using the `set_svf` command prior to synthesis, and is repeated here for *Formality*. Any optimization transformations which can affect formal verification are recorded by DC in this binary file. If the user does not specify a file name, *SVF* data is written to a file named `default.svf`. A text equivalent *SVF* file, called `svf.txt` can be found under the `formality_svf` directory after running *Formality*.

Question 22. Which files are the "Reference" and "Implementation" design files that are being verified for functional equivalence?

The reference RTL design file is **`rtl/STOTO.v`**.
(`read_verilog` **`-r`** `STOTO.v`)

The implementation or gate level design file is **`mapped/stoto.retime.ddc`**.
(`read_ddc` **`-i`** `STOTO.ddc`)

Question 23. Does verification succeed or fail?

Near the end of the log output, under
`*** Verification Results ****` you should see the message "**`Verification SUCCEEDED`**"

This page is left blank intentionally

6 Timing Analysis

There is NO LAB for this unit

This page is left blank intentionally.

7 Constraints: Multiple Clocks and Exceptions

Learning Objectives

The goal of this lab is to give you a better understanding of how static timing analysis works and how single-cycle timing exceptions are properly applied.

After completing this lab, you should be able to:

- Constrain and analyze complex multi-clock designs with mutually exclusive clocks and multicycle paths.

Lab Duration: 60 minutes

Background

The design you will be working with, called "**EXCEPTIONS**", has already been **constrained** and **compiled**, and saved as a "mapped" *ddc* file. The design contains parallel paths shown in the figure below (*resets* not shown). The inputs `adr_i` and `coeff`, and the output `dout` are each connected to both a combinational path as well as a sequential path.

The combinational and sequential paths have **different constraint** requirements. When you perform timing analysis, you will discover that there are initially many violations – all due to **incomplete** constraints. Your objective is to **add** appropriate constraints as dictated by the design requirements. Once the design is completely and correctly constrained, the design should **meet all timing constraints without needing to be re-compiled**.

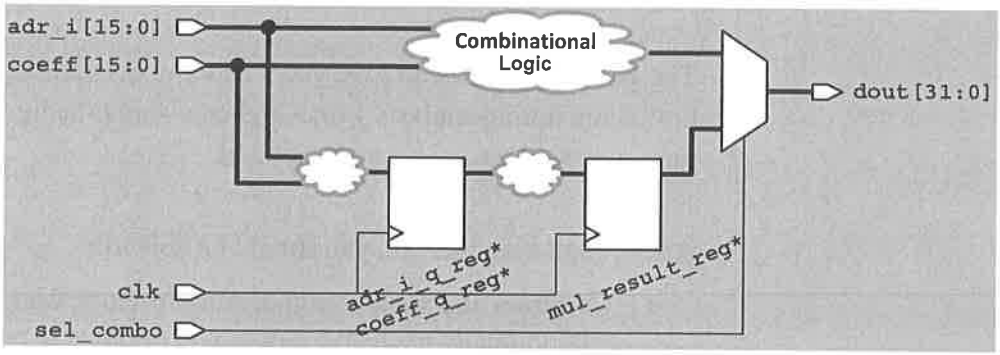


Figure 1. Design "EXCEPTIONS"
(The `adr_i_q*` and `coeff_q*` registers are shown as one to simpilfy the schematic)

If you run into difficulties while performing the various tasks, check the Answers/Solutions section at the back of this lab.

You may also check the file `.solutions/dc.tcl` ***for help.***

Lab Instructions

Task 1. Read a Mapped Design

1. Make sure your current working directory is **`lab7`** and invoke `dc_shell` in *Topographical* mode .

2. Read the compiled design "**`EXCEPTIONS`**" into memory:

```
read_ddc EXCEPTIONS.ddc
current_design EXCEPTIONS
link
```

3. Use the "`view`" procedure to generate and analyze a constraint report and a default timing report.

```
v  rc
v  rt
```

Note: "`v`" is an alias for the Tcl *procedure* called "`view`", which is defined in `../ref/tools/procs.tcl`. This is not a standard *Design Compiler* command. It is very useful for viewing reports in the interactive shell, so we are making it available for your use during the labs.

Question 1. What is the timing slack of the critical path – the *worst negative slack* (WNS)?

.....

Question 2. Describe the start and end points of this violating timing path:

.....

.....

Question 3. What is the available delay for this combinational path (= Clock Period – Input Delay – Output Delay)?

.....

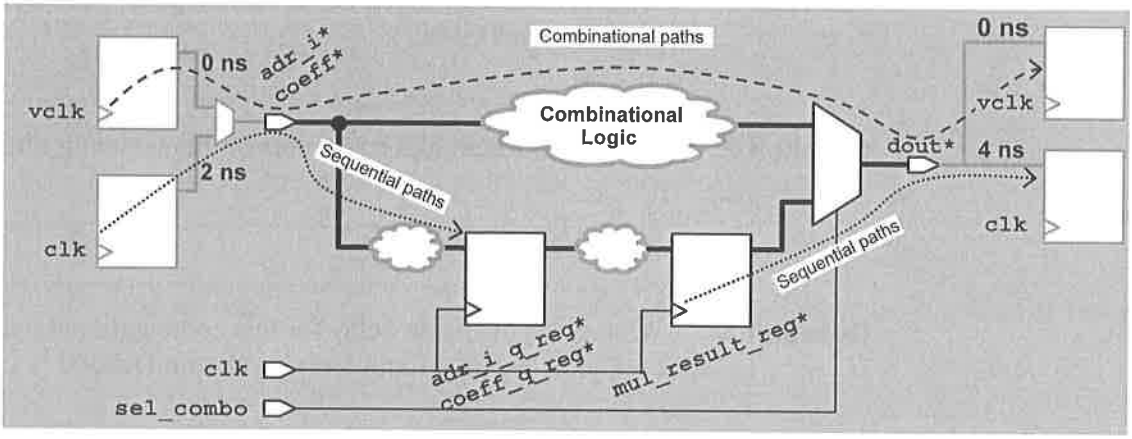
The combinational path is over constrained - it cannot meet a **–1 ns** maximum delay requirement! This design is improperly constrained for demonstration purposes. ***The maximum propagation delay through the combinational path should actually be constrained to 6 ns.*** The existing input and output delay constraints are correct for the sequential paths (from the `coeff*` and `adr_i*` input ports to the `coeff_q_reg*` and `adr_i_q_reg*` registers, and from the `mul_result_reg*` registers to the `dout*` output ports), but clearly not for the purely combinational paths. The constraints must be expanded such that the combinational and sequential paths are constrained independently. The next *Task* will guide you through the steps to accomplish this.

Task 2. Constrain Parallel I/O Paths Separately

The **`adr_i`**, **`coeff`** and **`dout`** I/O ports have parallel combinational and sequential timing paths. In this task you will use a *virtual clock* to enable you to constrain the combinational paths independently from the sequential timing paths.

The existing clock, input and output delay constraints are correct for the sequential input and output paths (see schematic below). In this task you will apply additional constraints so that the combinational logic paths are constrained independent of the sequential logic paths.

To accomplish this, the combinational paths will be constrained using a new *virtual clock* which you will call **`vclk`**. The sequential paths are already correctly constrained using the 5ns design clock **`clk`**, and input and output delays of 2ns and 4ns, respectively. The following schematic illustrates this concept (the input ports, **`adr_i`** and **`coeff`**, are collapsed into one port here to simplify the schematic):



The diagram shows that two clocks, **`clk`** and **`vclk`** are clocking the input and output data of the design. The sequential paths, to and from the internal registers, are constrained using **`clk`**, while the combinational paths (from `adr_i*` and `coeff*`, through the combinational logic, to `dout*`) are to be constrained only by the virtual clock **`vclk`**.

1. The combinational paths must have a maximum delay of 6 ns. Start by creating a **6 ns** virtual clock `vclk`, along with input and output delays with respect to `vclk` that are set to **0 ns**.

   Remember that the input delay constraint should be applied to the ports `coeff*` and `adr_i*`.

   Note: **IMPORTANT**: There are existing input- and output-delay constraints which must be kept as is. Use the `-add_delay` option to ***add*** the new I/O delay constraints above, without overwriting the existing constraints.

   Apply the appropriate commands based on what you learned in lecture. Use the *Job Aid* as needed.

   If you're stuck, check the Answers section, or look at `.solutions/dc.tcl`.

2. Generate a timing report for a timing path constrained by the virtual clock.

   The `-group vclk` option will generate a timing report for the path group `vclk`, which will report the timing path with the worst setup timing slack that is captured by `vclk`.

   ```
   v rt -group vclk
   ```

   Question 4. Does the path violate or meet timing?

   .....

   Question 5. Are the launch and capture clocks correct for the combinational paths?

   .....

   .....

   .....

3. The clocks `clk` and `vclk` should not interact. Apply a *logically exclusive clock group* constraint (or two *false path* constraints, if you prefer) to disable timing analysis between these clocks.

   If you're stuck, check the Answers section, or look at `.solutions/dc.tcl`.

4. Repeat the previous timing report.

Question 6. Does the path violate or meet timing?

.....

Question 7. Are the launch and capture clocks correct now?

.....

5. The combinational path constraints are not complete yet! Generate the following report and answer the following questions:

```
v rt -group clk
```

Question 8. Does this path violate or meet timing?

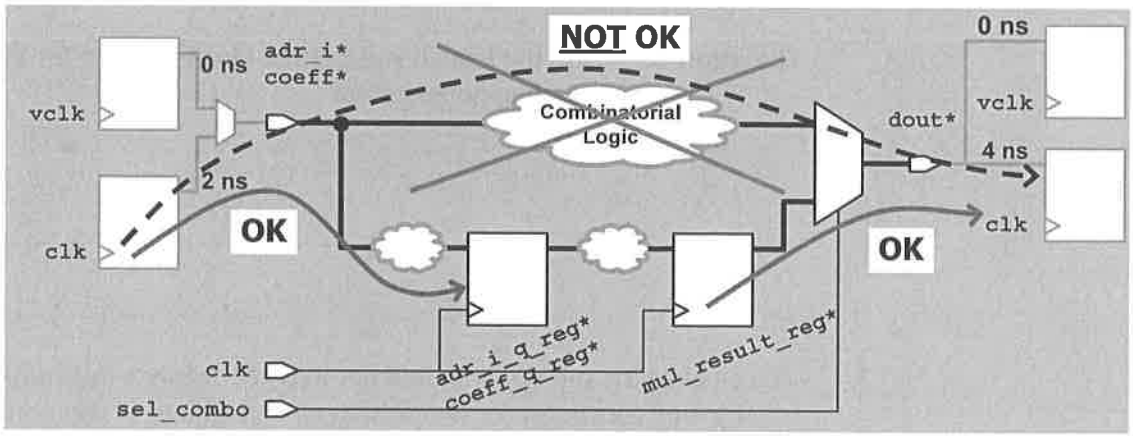
.....

Question 9. Describe the start and end points of this timing path. Does this match your expectations?

.....

.....

.....



6. As the final step, disable timing analysis from **clk** to **clk** through the combinational logic using the *false path* command.

   Hint: In addition to -from and -to options, use two **-through** options to help you distinguish the combinational **clk** to **clk** paths from the sequential **clk** to **clk** paths.

7. Generate a timing report from the coeff* and adr_i* inputs to all outputs and confirm that that these paths meet timing. Also verify that there are *no paths* through the combo logic constrained by **clk**:

```
set in_ports [get_ports "coeff* adr_i*"]
v rt -from $in_ports -to [all_outputs]
v rt -from $in_ports -to [all_outputs] -group clk
```

Task 3. Constrain a Multicycle Path

The paths to **mul_result_reg** are supposed to be multicycle paths, which are allowed to take up to 2 clock cycles (instead of 1, the default) for *setup* timing. The *hold* capture checks should be done at 0ns. These paths are not correctly constrained either.

1. Execute a timing report for the clock **clk**:

```
v rt -group clk
```

Question 10. How large is this violation in comparison to the clock period?

.....

2. Apply a *multicycle path* constraint of 2 cycles for setup to all paths that end at the **D** pin of **mul_result_reg***.

3. Verify that the multicycle path now meets setup timing:

```
v rt -to mul_result_reg*/D
```

Question 11. What is the effective clock period for this timing path (the clock capture edge minus the clock launch edge)?

.....

4. Generate a *hold timing* report to these same end points.

```
!! -delay min
```

Question 12. Do the launch and capture clock edges match the specification described at the beginning of this task?

.....

5. Complete the multicycle path constraint by including a constraint for hold.
6. Generate another *hold* timing report to confirm that the correct launch and capture clock edges are used, and that *hold* timing is met.
7. Generate a final "`report_constraint -all`" to make sure that all violations have been taken care of.
8. **OPTIONAL: If you have time, do the following before exiting the tool:**

   Whenever you apply *false path* or *multicycle path* "timing exceptions", it is highly recommended to check if any exceptions are being ignored by Design Compiler. Generate the following report:

```
report_timing_requirements -ignored
```

   You will notice that many `NONEXISTENT PATH`, `FALSE` paths are reported! If you look carefully at these paths, you will discover that all paths start from `clk`, go through `adr_i` or `coeff`, then through `dout`, and end at `clk`. They are a result of the `set_false_path` command applied in Task 2, step 6. If you look closely, you will discover that the problem has to do with the wildcards "*" that were used in both the input and output port names: Since the I/Os are parallel, bit-sliced, paths the bit "0" inputs only connect to bit "0" outputs, and not to bits 1-15, and bit "1" inputs only connect to bit "1" outputs, etcetera, which is the cause of the non-existent paths. These reported ignored exceptions are expected, and can be safely ignored.
9. **Quit** Design Compiler.

You have completed the lab.

Answers / Solutions

Question 1. What is the timing slack of the critical path – the *worst negative slack* (WNS)?

Both the constraint and the timing report show that the **WNS is -6.78 ns**. Notice from the *data required section* of the timing report that the clock period is 5ns. This means that the WNS violation is larger than 100% of the clock period that is being used!

Question 2. Describe the start and end points of this violating timing path:

From the timing report: The start point is an input port `coeff*`, and the end point is an output port `dout*`, which means that this is a combinational path (see **Figure 1**).

Question 3. What is the available delay for this combinational path (= Clock Period – Input Delay – Output Delay)?

-1ns (5ns – 2ns – 4ns). This is not an achievable constraint!

Solution for Task 2, Step 1:

```
create_clock -name vclk -period 6
set in_ports [get_ports "coeff* adr_i*"]
set_input_delay 0 -clock vclk -add_delay $in_ports
set_output_delay 0 -clock vclk -add_delay [all_outputs]
```

Question 4. Does the path violate or meet timing?

It violates timing by a large amount!

Question 5. Are the launch and capture clocks correct for the combinational paths?

The launch clock is `clk` and the capture clock is `vclk`. This is not correct! The combinational paths should be constrained only by `vclk`. The two clocks should not interact.

Solution for Task 2, Step 3:

```
set_clock_group -name false_grp1 -logically_exclusive \
   -group clk -group vclk
# OR alternatively:
set_false_path -from [get_clocks clk] -to [get_clocks vclk]
set_false_path -from [get_clocks vclk] -to [get_clocks clk]
```

Question 6. Does the path violate or meet timing?

It meets timing.

Question 7. Are the launch and capture clocks correct now?

Yes. Both the launch and capture clocks are now **`vclk`**, as expected.

Question 8. Does this path violate or meet timing?

It violates timing by a large amount!

Question 9. Describe the start and end points of this timing path. Does this match your expectations?

The start point is an input port and the end point is an output port. This should not happen! The combinational paths should only be constrained by **`vclk`**, not by **`clk`**.

Solution for Task 2, Step 6:

```
set_false_path -from [get_clocks clk] \
   -through $in_ports \
   -through [all_outputs] -to [get_clocks clk]
v rt -from $in_ports -to [all_outputs]
```

Question 10. How large is this violation in comparison to the clock period?

The violation is **4.68 ns**, which is almost equal to the clock period of 5 ns.

Solution for Task 3, Step 2:

```
set_multicycle_path 2 -setup -to mul_result_reg*/D
```

Question 11. What is the effective clock period for this timing path (the clock capture edge minus the clock launch edge)?

The effective clock period is 10 ns, 2 times the 5ns clock period of **clk**.

Question 12. Do the launch and capture clock edges match the specification described at the beginning of this task?

No! The launch clock edge is 0ns but the capture clock edge is at 5 ns (one edge preceding the *setup* clock edge). This does not match the original specification, which stated that the *hold* capture checks should be done at 0ns.

Solution for Task 3, Step 5:

```
set_multicycle_path 1 -hold -to mul_result_reg*/D
```

This page is left blank intentionally

8

DC Graphical: SPG Flow, Congestion, Layout GUI

There is NO LAB for this unit

This page is left blank intentionally.

9

OPTIONAL LAB: Constraints: Complex Design Considerations

Learning Objectives

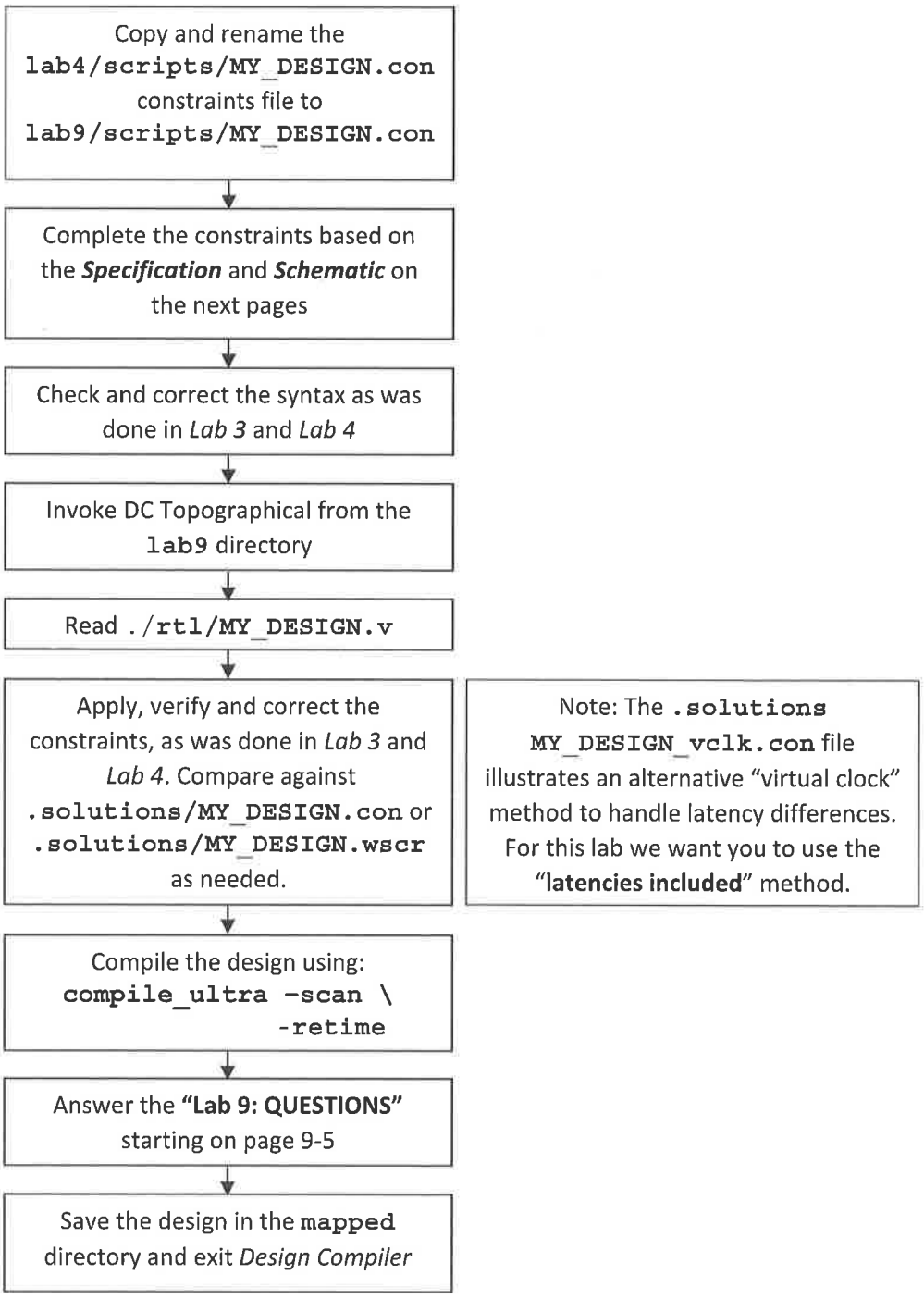
After completing this lab you should be able to:

- Constrain a design with the following characteristics:
  - Contains positive and negative edge-triggered registers
  - Inputs/outputs are being launched/captured by multiple registers
  - External input/output paths have non-default latencies
  - Driving cells on inputs drive additional external loads
- Compile the constrained design
- Relate entries in a timing report to their corresponding constraints

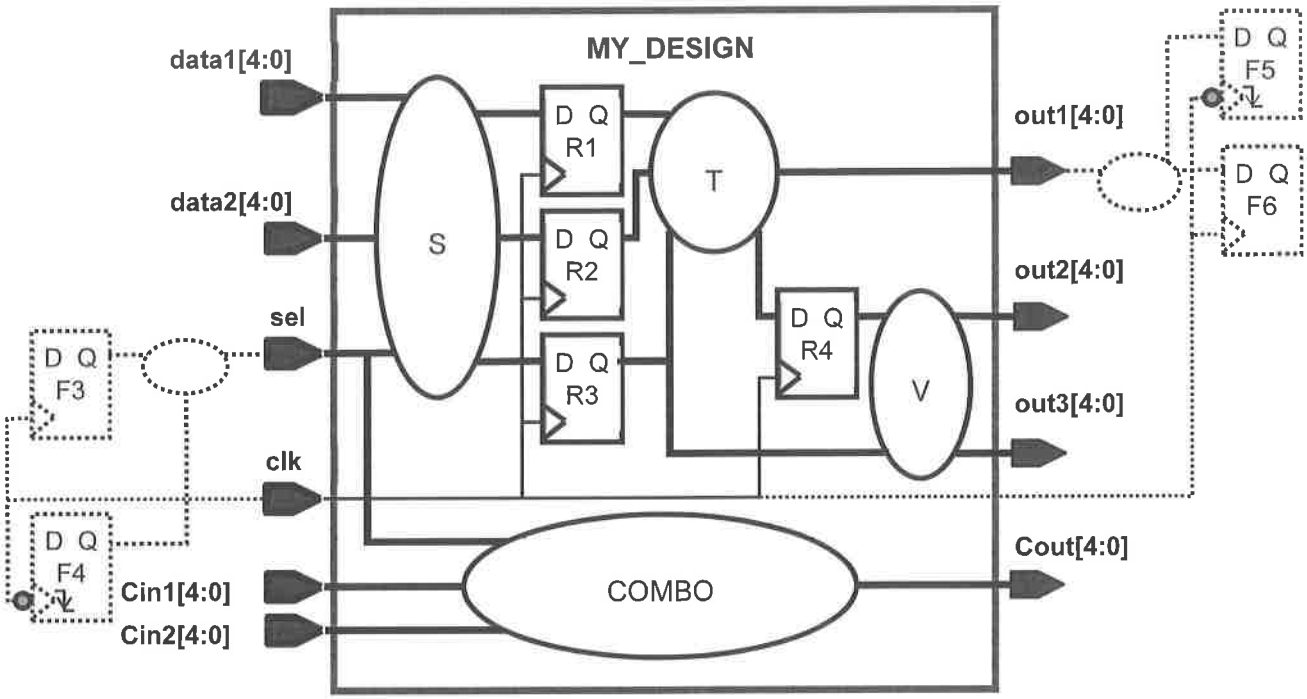
Lab Duration: 75 minutes

Instructions

Perform the steps below:



Design Schematic



Design Specification

Clock Spec	1. Rename the clock from "`clk` " to "`my_clk`" 2. Change the duty cycle to **40 % active high**, with **zero offset**.
Input Port Delays	1. The `sel` port data is being supplied by an additional register, `F4`. The data at `sel` arrives no later than **420ps** after the **negative** edge-triggered launching clock edge of `F4`. 1. The clock pin at `F4` sees a **600ps total latency (source + network)**, which is different than the total latency of `my_clk`. Use the "**latencies included**" method to incorporate this latency.
Output Port Delays	2. The `out1` port data is captured by an additional register, `F5`. The data at `out1` must arrive no later than **260ps** before `F5`'s negative edge-triggered clock edge. 3. The clock pin at `F5` sees a **network latency of 500ps**, (its **source latency** is the same as that of `my_clk`). Use the "**latencies included**" method to incorporate this latency.
External Load, Input Ports	Each input port (except `clk`) fans out to **2 other sub-blocks**, which each drive the equivalent of **3 `bufbd1`** (input pin `I`) buffers internally. Model this external **capacitive loading** on the inputs.

Questions

Before answering the following questions, complete the Instruction steps shown in the flow diagram on page 2, up to and including `compile_ultra`.

When answering the following questions, refer to the "*Answers/Solutions*" section at the end of this lab to verify your answers, or if you need some help.

Note that our answers to these questions are based on the "latencies included" method. If you use the "virtual clocks" method instead, some of your answers will differ.

1. Generate a constraint report:

```
v rc
```

Question 1. Does the design have any timing or DRC violations?

.....

2. Generate a timing report for the path to the `out1` output. Include options to show net *transition times* and *net delays* to *6 decimal places,* as well as net *fanout*:

```
v rt -trans -input -sig 6 -nets -to [get_ports out1]
```

3. Use the report to locate and fill in the following requested data. This data should match your constraints:

Question 2. What is the startpoint of the reported path?

.....

Question 3. From this report record the path's launching clock data. Include the constraint file command(s) which produce each of the report values:

Launching clock **`my_clk`**:
Launch edge time _____ Rising or falling? _____

Command(s) .....

.....

Clock network delay = ...

Command(s) .....

.....

Clock pin transition time = ...

Command(s) .....

.....

Question 4. From the report record the path's data required data. Include the constraint file command(s) which produce each of the report values:

Capturing clock **`my_clk`**:
Capture edge time _____ Rising or falling? _____

Command(s) .....

.....

Clock network delay = ...

Command(s) .....

.....

Clock uncertainty = ...

Command(s) .....

.....

Output external delay = ...

Command(s) .....

.....

Question 5. Why is the output timing with respect to a falling clock edge?

.....

.....

.....

Question 6. Why are the report values for *uncertainty* and *output external delay* the negative of their corresponding constraint values?

.....

.....

4. Generate a timing report for the path from the **`sel`** input to the **`Cout`** output. Include options to show *transition times* and *net delays* to *6 decimal places*. Use the report to locate and fill in the requested data. This data should match your constraints:

```
v rt -trans -input -sig 6 -from [get_ports sel] \
                          -to [get_ports Cout]
```

Question 7. From the report record the path's startpoint data. Include the constraint file command(s) which produce each of the report values:

Clock **`my_clk`**
Launch edge time _____ Rising or falling? _____

Command(s) .....

.....

Clock network delay = ...

Command(s) .....

.....

Input external delay = ...

Command(s) .....

.....

Transition time of **sel** port = ...

Command(s) .....

.....

Question 8. What is causing the "Incr" (incremental) delay on the **sel** input port?

.....

Question 9. How can you verify that the COMBO path from **Cin*** to **Cout** is constrained to the required 2.45ns spec?

.....

5. Save the design in **mapped** and exit Design Compiler.

You are now able to create, verify and identify timing and environmental constraints for any single clock design given the schematic and specification!!!

You have completed lab.

Answers / Solutions – Using the "Latencies Included" method

Question 1. Does the design have any timing or DRC violations?

From `report_constraint -all` you should see that there are no timing or design rule constraints violations. If you have any violations (other than *max_leakage_power* and *Min pulse width* violations):
1) Check and correct your constraints (compare against `.solutions/MY_DESIGN.con`)
2) Remove the design from memory
3) Re-apply the constraints
4) Re-compile the design
5) Generate another constraint report

Question 2. What is the startpoint of the reported path?

The startpoint is the clock pin of a register:
`R_#/CP`

Question 3. From this report record the path's launching clock data. Include the constraint file command(s) which produce each of the report values:

Launching clock **`my_clk`**:
Launch edge time: **0.0ns, Rising**

```
create_clock -period 3.0              \
     -name my_clk -waveform {0 1.2} \
      [get_ports clk]
```

Clock network delay = **1.0ns**

```
set_clock_latency -source -max 0.7  \
          [get_clocks my_clk]
set_clock_latency -max 0.3          \
          [get_clocks my_clk]
```

Clock pin transition time = **0.12ns**

```
set_clock_transition -max 0.12      \
          [get_clocks my_clk]
```

Question 4. From the report record the path's data required data. Include the constraint file command(s) which produce each of the report values:

Capturing clock **my_clk**:
Capture edge time: **1.2ns, Falling**

```
create_clock -period 3.0              \
     -name my_clk -waveform {0 1.2} \
      [get_ports clk]

set_output_delay -max -0.94       \
     -clock my_clk -add_delay     \
     -clock_fall                  \
     -network_latency_included    \
     -source_latency_included     \
      [get_ports out1]
```

Clock network delay = **0.0 ns**

Note: Your results will be different if you did not use the `-network/source_latency_included` options shown.

```
set_output_delay -max -0.94          \
     -clock my_clk -add_delay        \
     -clock_fall                     \
     -network_latency_included       \
     -source_latency_included        \
      [get_ports out1]
```

Clock uncertainty = **-0.15ns**

```
set_clock_uncertainty -setup 0.15 \
      [get_clocks my_clk]
```

Output external delay = **0.94ns**

```
set_output_delay -max -0.94       \
     -clock my_clk -add_delay     \
     -clock_fall                  \
     -network_latency_included    \
     -source_latency_included     \
      [get_ports out1]
```

Question 5. Why is the output timing with respect to a falling clock edge?

The `out1` port is constrained by two registers – a rising (`F6`) and a falling (`F5`) edge-triggered one. DC determined that the timing constraint to the falling edge-triggered register is the more restrictive of the two.

Question 6. Why are the report values for *uncertainty* and *output external delay* the negative of their corresponding constraint values?

The negated values simply mean that the constraint numbers are being subtracted from the "*data required time*".

Question 7. From the report record the path's startpoint data. Include the constraint file command(s) which produce each of the report values:

Launching clock `my_clk`:
Launch edge time: **1.2ns, Falling**

```
create_clock -period 3.0              \
      -name my_clk -waveform {0 1.2} \
       [get_ports clk]

set_input_delay -max 1.02          \
      -clock my_clk -add_delay     \
      -clock_fall                  \
      -network_latency_included    \
      -source_latency_included     \
       [get_ports sel]
```

Clock network delay = **0.0ns**

Note: Your results will be different if you did not use the `-network/source_latency_included` options shown.

```
set_input_delay -max 1.02      \
   -clock my_clk -add_delay    \
   -clock_fall                 \
   -network_latency_included   \
   -source_latency_included    \
            [get_ports sel]
```

Input external delay = **1.02ns**

```
set_input_delay -max 1.02     \
   -clock my_clk -add_delay   \
   -clock_fall                \
   -network_latency_included  \
   -source_latency_included   \
            [get_ports sel]
```

Transition time of **sel** port = **~0.5ns**

```
set_driving_cell                \
   -max -lib_cell bufbd1        \
   [remove_from_collection      \
   [all_inputs]                 \
   [get_ports "clk Cin*"]]
```

Question 8. What is causing the "Incr" (incremental) delay on the **sel** input port?

This represents the additional time for the input signal, with the above transition time, to reach the "switching point". It is not due to net delay, which is reported separately at the input pin of the first gate.

Question 9. How can you verify that the COMBO path from **Cin*** to **Cout** is constrained to the required 2.45ns spec?

```
v rt -from [get_ports Cin*] \
     -to [get_ports Cout]
```

Subtract the data arrival time at the Cin input port (1.30 ns in the "Path" column) from the "data required time" (3.75 ns), to get **2.45ns**.

10 Post-Synthesis Output Data

There is NO LAB for this unit

This page is left blank intentionally.

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Tool Invocation, Basic Help and Setup

Invoking Design Compiler in Topographical Mode

```
unix% dc_shell –topographical                  # Interactive shell
dc_shell-topo> start_gui                       # Open GUI from shell
dc_shell-topo> stop_gui                        # Close GUI, return to shell
unix% design_vision –topographical             # Open GUI directly
unix% dc_shell  -topo –f dc.tcl  |  tee –i dc.log   # Batch mode
```

Getting Help in DC shell

```
help –verbose *clock              # Lists command options
create_clock –help                # Lists command options
man create_clock                  # Displays command man page
apropos –symbols_only period      # List command(s) with key word in command/option
printvar *_library                # Lists variable and values
man target_library                # Displays variable man page
```

Settings and Checks Applied at Startup

```
# These settings can all be included in the .synopsys_dc.setup file, OR,
# they can be split up into "commom_setup.tcl" and "dc_setup.tcl" files, which are
# then sourced by the "run script", called "dc.tcl", as done by the "Reference
# Methodology" generated scripts
set ADDITIONAL_SEARCH_PATH      "./libs/sc/LM  ./rtl ./scripts"
set TARGET_LIBRARY_FILES        sc_max.db
set ADDL_LINK_LIBRARY_FILES     IP_max.db
set SYMBOL_LIBRARY_FILES        sc.sdb
set MW_DESIGN_LIB               MY_DESIGN_LIB
set MW_REFERENCE_LIB_DIRS       "./libs/mw_lib/sc ./libs/mw_libs/IP"
set TECH_FILE                   ./libs/tech/cb13_6m.tf
set TLUPLUS_MAX_FILE            ./libs/tlup/cb13_6m_max.tluplus
set MAP_FILE                    ./libs/tlup/cb13_6m.map

set_app_var search_path "$search_path $ADDITIONAL_SEARCH_PATH"
set_app_var target_library $TARGET_LIBRARY_FILES
set_app_var link_library "* $target_library $ADDL_LINK_LIBRARY_FILES"
set_app_var symbol_library $SYMBOL_LIBRARY_FILES
set_app_var mw_reference_library $MW_REFERENCE_LIB_DIRS
set_app_var mw_design_library $MW_DESIGN_LIB
get_app_var  -list  -only_changed_vars  *
```

Additional Help

```
# Using DC "man" from UNIX prompt
unix% alias dcman '/usr/bin/man –M $SYNOPSYS/doc/syn/man'
unix% dcman target_library
```

Solvnet articles, documentation, support: **solvnet.synopsys.com**

Register for workshops: **training.synopsys.com**

Helpful UNIX-like DC-shell commands

```
pwd; cd;  ls;
history;  !!;  !7;  !report
sh <UNIX_command>
printenv
get_unix_variable <UNIX_variable>
```

Settings and Checks Applied at Startup

```
# Only create new Milkyway design library if it doesn't already exist
if {![file isdirectory $mw_design_library ]} {
     create_mw_lib   -technology $TECH_FILE  \
       -mw_reference_library $mw_reference_library  $mw_design_library}
open_mw_lib     $mw_design_library
check_library
set_tlu_plus_files -max_tluplus $TLUPLUS_MAX_FILE  -tech2itf_map $MAP_FILE
check_tlu_plus_files
history keep 200
set_app_var alib_library_analysis_path  ../ ;# ALIB files
define_design_lib WORK –path ./work
set_svf <my_filename.svf>
set_app_var sh_enable_page_mode false
suppress_message {LINT-28 LINT-32 LINT-33 UID-401}
set_app_var alib_library_analysis_path [get_unix_variable HOME]
alias h history
alias rc "report_constraint -all_violators"
```

SYNOPSYS®

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Tcl Constructs and Collection Commands

TCL Commands and Constructs

```
set PER 2.0
echo $PER            # Result: 2.0
set MARG 0.95
expr $PER * $MARG    # expr: *, /, +, -, >, <, =, <=, >=
set  PCI_PORTS [get_ports A]
set  PCI_PORTS [get_ports "Y??M Z*"]
echo "Effctv P = [expr $PERIOD * $MARGIN]"
# Result with soft quotes:  "Effctv P = 1.9"
echo {Effctv P = [expr $PERIOD * $MARGIN]}
# Result with hard quotes:
# "Effctv P = [expr $PERIOD * $MARGIN]"
# Tcl Comment line
set COMMENT in_line ;   # Tcl inline comment
set MY_DESIGNS "A.v B.v Top.v"
foreach DESIGN $MY_DESIGNS {read_verilog $DESIGN}
for {set i 1} {$i < 10} {incr i} {read_verilog BLOCK_$i.v}
```

Object Retrieval and Manipulation (Collection Commands)

```
get_ports
get_pins
get_designs
get_cells
get_nets
get_clocks
get_nets –of_objects [get_pins FF1_reg/Q]

get_libs <lib_name>
get_lib_cells <lib_name/cell_names>
get_lib_pins <lib_name/cell_name/pin_names>

all_inputs
all_outputs
all_clocks
all_registers
all_connected
all_fanin
all_fanout
all_ideal_nets


set  PCI_PORTS [get_ports pci_*]
echo $PCI_PORTS              # → _sel184
query_objects $PCI_PORTS     # → {pci_1 pci_2 ...}
get_object_name $PCI_PORTS   # → pci_1 pci_2 ...
sizeof_collection $PCI_PORTS # → 37


set  PCI_PORTS [add_to_collection $PCI_PORTS \
                    [get_ports CTRL*]]
set  ALL_INP_EXC_CLK [remove_from_collection \
                    [all_inputs] [get_ports CLK]]
```

Object Retrieval and Manipulation (Collection Commands) (continued...)

```
compare_collections

index_collection

sort_collection


foreach_in_collection my_cells [get_cells -hier * \
                    -filter "is_hierarchical == true"] {

echo "Instance [get_object_name $cell] is hierarchical"

}

  # Filtering operators: ==, !=, >, <, >=, <=, =~, !~
filter_collection [get_cells *] "ref_name =~ AN*"
get_cells * -filter "dont_touch == true"
get_clocks * -filter "period < 10"


  # List all cell attributes and redirect output to a file
redirect –file cell_attr \
          {list_attributes –application –class cell}

 # Grep the file for cell attributes starting with dont_
UNIX% grep dont_ cell_attr | more


  # List the value of the attribute dont_touch
get_attribute <cell_name> dont_touch


# Example: Identify glue cells in the current design
set GLUE_CELLS \
      [get_cells * -filter "is_hierarchical == false"]
```

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Timing and Physical Constraints; Generating Initial Reports

Timing Constraints

```
reset_design
############ CLOCKS ############################
create_clock -period 2 –name Main_Clk [get_ports Clk1]
create_clock –period 2.5 –waveform {0 1.5} [get_ports Clk2]
create_clock –period 3.5 –name V_Clk; # VIRTUAL clock
create_generated_clock –name  DIV2CLK -divide_by 2 -source [get_ports Clk1] \
            [get_pins I_DIV_FF/Q]
set_clock_uncertainty –setup 0.14 [get_clocks *]
set_clock_uncertainty –setup 0.21 –from [get_clocks Main_Clk] –to [get_clocks Clk2]
set_clock_latency –max 0.6 [get_clocks Main_Clk]  ; # Pre layout  ideal clock tree OR
set_propagated_clock [all_clocks] ; # Post CTS (Clock Tree Synthesis) clock tree
set_clock_latency –source –max 0.3 [get_clocks Main_Clk]
set_clock_transition –max 0.08 [get_clocks Main_Clk]
############ CLOCK TIMING EXCEPTIONS ########
set_clock_groups –logically_exclusive | -physically_exclusive | -asynchronous \
                                        –group CLKA –group CLKB
set_false_path -from [get_clocks Asynch_CLKA] -to [get_clocks Asynch_CLKB]
set_multicycle_path –setup 4 –from –from A_reg -through U_Mult/Out –to B_reg
set_multicycle_path –hold 3 –from –from A_reg -through U_Mult/Out –to B_reg
################  I/O TIMING ########################
set_input_delay -max 0.6 -clock Main_Clk \
  –source_latency_included  –network_latency_included [all_inputs]
set_input_delay -max 0.48 –clock V_Clk -clock_fall  –add_delay  [get_ports A]
set_output_delay -max 0.9 -clock Clk2 \
  –source_latency_included  –network_latency_included [get_ports "OUT2 OUT7"]
set_output_delay -max 1.1 –clock V_Clk -clock_fall  –add_delay  [get_ports OUT7]
################ ENVIRONMENT ######################
set_max_capacitance 1.2 [all_inputs]; # User-defined Design Rule Constraint
set_load –max 0.080  [all_outputs]
set_load –max [expr [load_of slow_proc/NAND2_3/A] * 4] [get_ports OUT3]
set_load –max 0.12 [all_inputs]
set_input_transition –max 0.12 [remove_from_collection [all_inputs][get_ports B]]
set_driving_cell –max –lib_cell FD1 –pin Q [get_ports B]
```

Checking and Removing Constraints

```
report_clock; report_clock –skew -attr
report_design
report_port –verbose
report_path_group
report_timing_requirements –ignored
report_auto_ungroup
report_interclock_relation
write_script –output <constraints.tcl>
check_timing
reset_path –from FF1_reg
remove_clock
remove_clock_transition
remove_clock_uncertainty
remove_input_delay
remove_output_delay
remove_driving_cell
```

Syntax Checking

```
Unix% dcprocheck constr_file.con
```

Generating Reports

```
# Generate a library report file
read_db library_file.db
list_libs
redirect \
    –file reports/lib.rpt {report_lib <libname>}
report_hierarchy [-noleaf]
# Arithmetic implementation and  resource-sharing info
report_resources
  # List area for all cells in the design
report_cell [get_cells –hier *]
```

Conservative Output Load Algorithm

Used for "load budgeting" if actual output load values are not known. Finds the largest `max_capacitance` value in the library and applies that value as a conservative output load

```
set LIB_NAME ssc_core_slow
set MAX_CAP 0
set OUTPUT_PINS [get_lib_pins $LIB_NAME/*/*   \
                                    -filter "direction == 2"]
foreach_in_collection pin $OUTPUT_PINS {
  set NEW_CAP [get_attribute $pin max_capacitance]
  if {$NEW_CAP > $MAX_CAP} {
    set MAX_CAP $NEW_CAP
  }
}
set_load –max $MAX_CAP [all_outputs]
```

Correlation between DC and ICC (II)

```
# Load the same ICC (II) floorplan definition in DC

# Automatically change key DC timing variable settings to match ICC (II) defaults:
set_compile_spg_mode  icc | icc2

# Use the "Consistency Checker" to compare
#  settings:   Solvnet Doc Id: 026366, 2112936
```

Physical Constraints

set_aspect_ratio	create_route_guide
set_utilization	create_net_shape
create_die_area	create_user_shape
create_site_row	set_pin_physical_constraints
set_port_side	create_pin_guide
create_terminal	create_via_master
set_cell_location	create_via
create_placement_blockage	create_track
create_voltage_area	set_keepout_margin
create_bounds	compute_polygons

```
report_physical_constraints
reset_physical_constraints
```

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Run Script

Run Script
(called "dc.tcl" in the RM scripts)

```
read_verilog {A.v B.v TOP.v}              or
read_sverilog  {A.sv B.sv TOP.sv}         or
read_vhdl {A.vhd B.vhd TOP.vhd}           or
read_ddc MY_TOP.ddc                       or
analyze –format verilog {A.v B.v TOP.v}
elaborate MY_TOP –parameters "A_WIDTH=8, B_WIDTH=16"

current_design MY_TOP

link

if {[check_design] ==0} {
   echo "Check Design Error"
   exit                  # Exits DC if a check-design error is encountered
}                        # Continue if NO problems encountered
write_file –f ddc –hier –out unmappedd/TOP.ddc


redirect –tee –file reports/precompile.rpt {source –echo -verbose TOP.con}
redirect –append –tee –file reports/precompile.rpt {check_timing}

# Source physical constraints, if available, prior to 1st pass synthesis:
source <Physical_Constraints.tcl>
# Load the floorplan (FP) definition from DEF or floorplan Tcl file(s) prior to 2nd pass:
extract_physical_constraints [-allow_physical_cells] <FP.def>
read_floorplan <FP.tcl>
```

"Compile Flow" commands go here – see next page

Run Script (Continued)
(called "dc.tcl" in the RM scripts)

```
check_design
check_design –html check_design.html
sh firefox check_design.html

report_constraint –all_violators
report_qor
report_timing [ -delay <max | min> ]
              [ -to <pin_port_clock_list> ]
              [ -from <pin_port_clock_list> ]
              [ -through <pin_port_list> ]
              [ -group]
              [ -input_pins ]
              [ -max_paths <path_count> ]
              [ -nworst <paths_per_endpoint_count >]
              [ -nets ]  [ -capacitance ] [-transition]
              [ -significant_digits <number>]

report_area
report_congestion

# Parallel command execution using a maximum of 8 cores
update_timing
parallel_execute [list          "report_constraint -all > rc.rpt"  \
                                "report_timing          > rt.rpt"  \
                                "report_area            > ra.rpt" ]

set_app_var verilogout_no_tri true

change_names –rule verilog –hier ;# See below to avoid bit blasting ports

# Avoid bit-blasted ports in SystemVerilog and VHDL records
define_name_rules verilog –preserve_struct_ports

change_names –hierarchy –rules verilog

write_file –f verilog –hier –out mapped/TOP.v

write_file –f ddc –hier –out mapped/TOP.ddc

write_sdc TOP.sdc

write_scan_def –out TOP_scan.def

write_icc2_files –output TOP_icc2

exit
```

SYNOPSYS®

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Compile Flow

Compile Flow

```
group_path -name CLK1 -critical_range <10% of CLK1 Period> –weight 5
group_path -name CLK2 -critical_range <10% of CLK2 Period> –weight 2
group_path –name INPUTS –from [all_inputs]
group_path –name OUTPUTS –to [all_outputs]
group_path –name COMBO –from [all_inputs] –to [all_outputs]

set_fix_multiple_port_nets –all –buffer_constants


# Enable multi-core optimization (Additional licenses may be required):
set_host_options –max_cores <#>
report_host_options
remove_host_options

# Prevent specific sub-designs from being ungrouped:
set_ungroup <top_level_and/or_pipelined_blocks> false
# Prevent DesignWare hierarchies from being ungrouped:
set_app_var compile_ultra_ungroup_dw false

# If needed: Disable boundary optimization on specific sub-designs:
set_boundary_optimization <cells or designs> false
# Allow only constant propagation
set_app_var compile_enable_constant_propagation_with_no_boundary_opt true
# Disable constant propagation on specific sub-block pins
set_compile_directives –constant_propagation false [get_pins "SUB2/In2 SUB2/In3"]

# If needed: Exclude specific cells/design from adaptive retiming (-retime)
set_dont_retime <cells_or_designs> true


# Manually controlling register replication:
set_register_replication [–num_copies 3 | -max_fanout 40] [get_cells B_reg]
# If needed: Change register replication naming style:
set_app_var register_replication_naming_style "%s_rep%d"
# To enable register replication across hierarchy:
set_app_var  compile_register_replication_across_hierarchy true
```

Compile Flow (Continued)

```
# Select the scan register style for test-ready synthesis, and disable
# automatic shift-register identification, if applicable:
set_scan_configuration \
      -style <multiplexed_flip_flop | clocked_scan | lssd | aux_clock_lssd>
set_app_var compile_seqmap_identify_shift_registers false

# If needed: Maintain registered pipeline outputs if required
set_dont_retime [get_cells U_Pipeline/R12_reg*] true

# Enable register retiming if your design(s) contain(s) pure pipelines;
# Enable threshold optimization if needed:
set_optimize_registers true –design My_Pipeline_Subdesign \
                [-clock CLK1 –delay_threshold]  <clock_period>

# Enable high effort timing optimization
set_app_var compile_timing_high_effort true

# Focus timining-driven placement on TNS reduction (instead of WNS):
set_app_var placer_tns_driven true

# Disable runtime intensive settings
compile_prefer_runtime

# Prioritize setup timing or max-delay over DRC
set_cost_priority –delay

# Disable DRC fixing on the entire clock network
set_auto_disable_drc_nets –on_clock_network true
```

"Net Layer Optimization" commands go here – see next page

Compile Flow (Continued)

```
# The following command and variable settings affect coarse placement
# during compile and incremental compile, which can directly or
# indirectly affect congestion:
set_app_var compile_prefer_mux                                  false | true
set_app_var placer_channel_detect_mode                          false | true
set_app_var placer_congestion_effort                            auto | medium
set_app_var placer_enable_enhanced_router                       false | true
set_app_var placer_enable_enhanced_soft_blockages               false | true
set_app_var placer_enable_redefined_blockage_behavior           false | true
set_app_var placer_max_cell_density_threshold          -1 | (0<=value<=1.0)
set_app_var placer_reduce_high_density_regions                  false | true
set_app_var placer_tns_driven                                   false | true
set_app_var placer_tns_driven_in_incremental_compile            false | true
set_app_var spg_congestion_placement_in_incremental_compile     false | true
set_app_var spg_high_effort_mux_area_structuring                false | true
set_app_var spg_place_enable_precluster                         false | true

set_congestion_options [–max_util] [-layer] [-availability] [-coordinate] ...


#                       Compile:
# Enable/disable optimizations as needed.
# The –spg option is recommended once the floorplan definition
# is complete (2nd pass synthesis):

compile_ultra –scan –retime  [-spg]  [-no_boundary]  [-no_autoungroup]
                [-no_design_rule]  [-no_seq_output_inversion]

# Apply more focus on violating critical paths, as necessary
group_path –name <group_name> -from <path_start> -to <path_end> \
            –critical range <10% of max delay goal> -weight 5

# Perform an incremental ultra compile:
compile_ultra [use same options from 2nd pass compile] –incremental

# Final Area Recovery
optimize_netlist –area
```

SYNOPSYS®

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Net Layer Optimization

Net Layer Optimization

```
# There are 3 different ways to do net layer optimization

# 1a. Automatic net layer optimization using top 2 layers
compile_ultra –spg

# 1b. Enabling automatic net layer optimization using multiple layers
set_app_var spg_enable_zroute_layer_promotion true
compile_ultra –spg


# 2. Pattern based  layer aware optimization
remove_net_search_pattern –all

set PATTERN_FO [create_net_search_pattern –fanout_lower_limit 150.00]
set_net_search_pattern_delay_estimation_options \
        -min_layer_name M6 –max_layer_name M8 –pattern $PATTERN_FO

set PATTERN_NL [create_net_search_pattern -net_length_lower_limit 100]
set_net_search_pattern_delay_estimation_options \
        -min_layer_name M7 -max_layer_name M8 -pattern $PATTERN_NL

set PATTERN_CRIT_IO \
      [create_net_search_pattern -setup_slack_upper_limit "-1.0" -connect_to_port]
set_net_search_pattern_delay_estimation_options \
        -min_layer M7 -max_layer M10 -pattern $PATTERN_CRIT_IO
compile_ultra –spg


# 3. Pattern based  layer aware optimization
set_net_routing_layer_constraints \
        –min_layer_name M5 -max_layer_name M6 [get_net top/sub/net1]
compile_ultra –incremental –spg ;# Optimization
extract_rc –estimate ;# RC estimation

# To see the net layer assignments from DC:
report_net_routing_layer_constraints [get_nets –hier]  –output net_layer.tcl
```

Application note available on SolvNet:
"How to Implement Net Layer Optimization With Design Compiler Graphical"
https://solvnet.synopsys.com/retrieve/037946.html

SYNOPSYS®

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

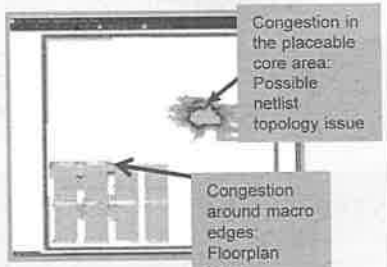
Congestion Analysis

Congestion analysis

Use same global and common *Zroute* settings in DC as ICC (II)
set_route_zrt_global_options ...
set_route_zrt_common_options ...

Congestion can be visually analyzed in the GUI:
`dc_shell-topo> start_gui`
`Design Vision GUI: Windows → New Layout Window`
`Layout Window: View → Map Mode → Reload`
Command equivalent: `report_congestion -build_map`

Congestion can be reported in text format:
`dc_shell-topo> report_congestion`



Overall Congestion

```
Both Dirs: Overflow = 3621
H routing: Overflow = 1724
V routing: Overflow = 1897
```

- Congestion Direction
- Overflow: Total # wires for which a GRC routing resource was not available

Worst Hotspot

```
Max = 13    (1 GRCs)
Max = 13    (1 GRCs)
Max =  8    (1 GRCs)
```

- Maximum overflows for a single GRC
- # GRCs with maximum overflow

Violating GRCs

```
GRCs = 2247  (2.73%)
GRCs = 1139  (1.38%)
GRCs = 1108  (1.34%)
```

- # GRCs with any overflow
- Percentage Violating GRCs to Total GRCs

SYNOPSYS®

Design Compiler: RTL Synthesis 2017.09-SP4 JOB AID

Congestion Alleviation Flow