JOINT INSTITUTE
交大密西根学院

ECE4810J SoC Design

Fall 2022

Lab #5 ASIC Backend Flow

Due: 11:59pm Nov. 6th, 2022

Logistics:

- This lab is a team exercise.
- Please use the discussion board on Piazza for Q&A.
- All reports and code (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

Contents

1 Overview

In this lab, you will learn about ASIC design flow. The goals of this lab are to:

- Learn how to compile the design with IC Compiler and what kind of commands are needed for it.
- Learn how to use Formality to detect unexpected differences that may have been introduced into a design during development.
- Learn how to use PrimeTime to validate the timing performance of a design by checking all possible paths for timing violations without using logic simulation or test vectors.

2 Physical Design

2.1 Introduction

IC Compiler is a single, convergent netlist-to-GDSII synthesis design tool for chip designers developing very deep submicron designs. It takes as input a gate-level netlist, a detailed floorplan, timing constraints, physical and timing libraries, and foundry-process data, and it generates as output either a GDSII-format file of the layout. IC Compiler provides two user interfaces:

- Shell interface (icc_shell) - The IC Compiler command-line interface is used for scripts, batch mode, and push-button operations.
- Graphical user interface (GUI) - The IC Compiler graphical user interface is an advanced analysis and physical editing tool.

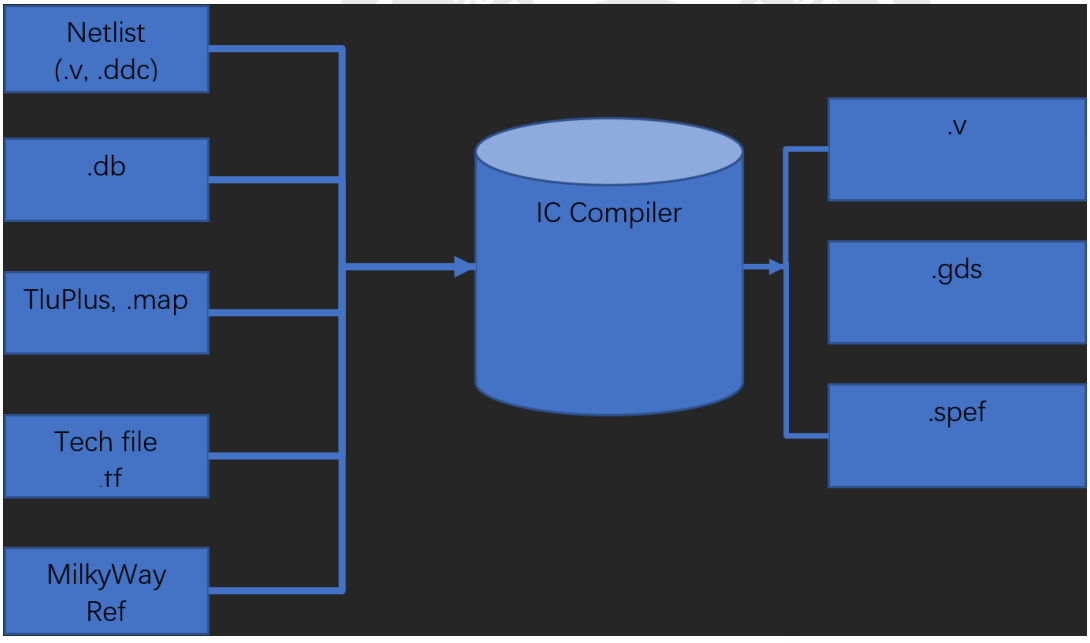


Figure 1: Input and output files of IC compiler.

The cell logic, timing, and power information are typically contained in a set of Synopsys databases (.db).

Tech file - A technology file provides technology-specific information, such as the names and physical and electrical characteristics of each metal layer and the routing design rules. IC Compiler uses the Milkyway design library to access the technology information. A technology file contains process-specific data. **TluPlus** - The parasitic attributes define the metal layer parasitics. In general, IC Compiler gets the parasitic information from the TLUPlus files rather than from the technology file.

We will use Synopsys IC Compiler Version Q-2019.12-SP5 for linux64 in this lab. You can modify your `.cshrc` *to launch the latest Synopsys IC Compiler. However, we will sooner or later migrate to Synopsys IC Compiler II (icc2_shell)*

Synopsys IC Compiler includes

- IC Compiler (TM)
- IC Compiler-PC (TM)
- IC Compiler-XP (TM)
- IC Compiler-DP (TM)
- IC Compiler-AG (TM)

2.2 Simplified IC Compiler Design Flow

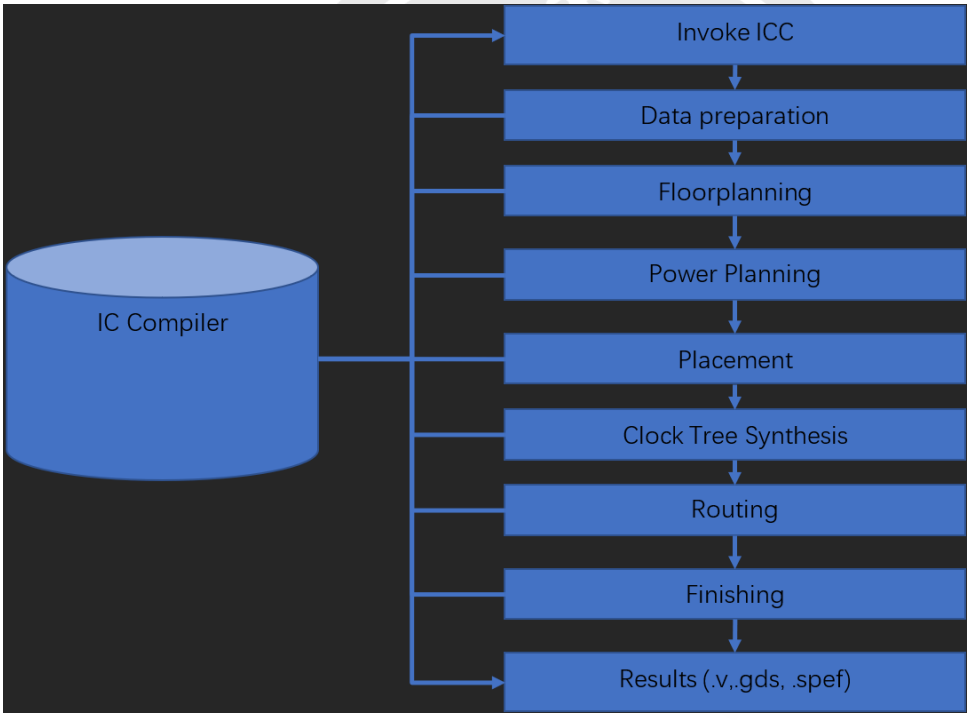


Figure 2: Simplified IC Compiler Design Flow.

1. Start IC Compiler graphical user interface (GUI) from the **work** directory. To start the IC Compiler command-line interface, enter the following commands at the UNIX or Linux prompt:

```
unzip /home/Flow/Synopsys/Scripts/Icc32.zip -d ~/synopsys/syn_tut/
cd ~/synopsys/syn_tut/icc_32/work
icc_shell -gui
```

This opens the IC Compiler top-level GUI window (Figure 3).

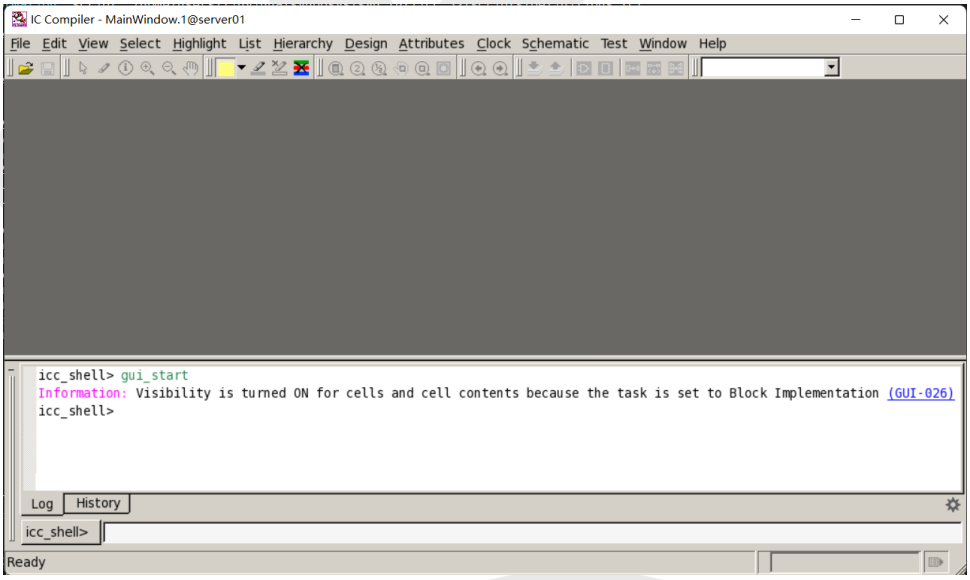


Figure 3: IC Compiler Top-level GUI window.

The libraries are appended to the search path by running the setup.tcl script from the `../scripts` directory, or alternatively, you can set them up manually (Fig. 2). To open the Setup Library window, go to File->Setup->Appliation Setup...

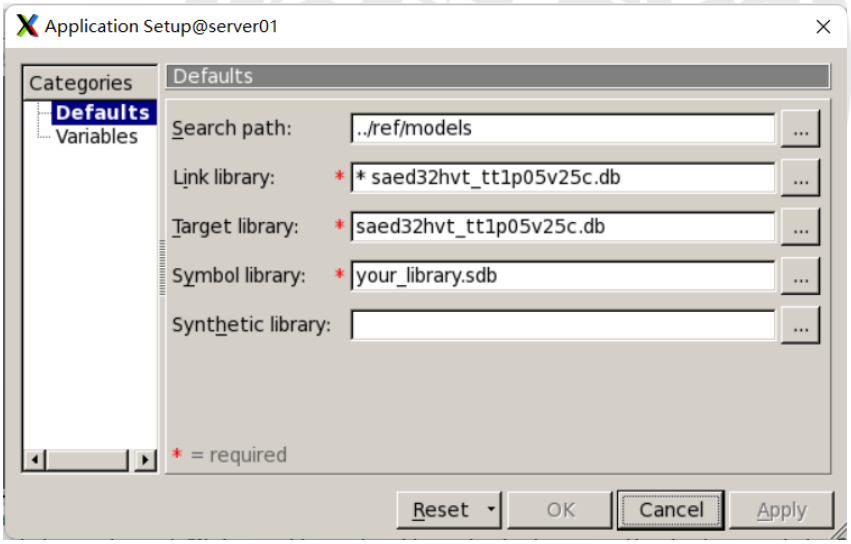


Figure 4: Application setup window.

```
source ../scripts/setup.tcl
```

In IC Compiler, you specify the .db files to use for the design by setting target_library and link_library variables:

- The target_library variable specifies the .db library files containing the logic cells that can be used for optimization, for example, different NAND gates having various areas, drive strengths, delays, and power usage.
- The link_library variable specifies the .db libraries containing all the logic cells that can be used to resolve hierarchical references in the design during the execution of the link command.

2. Use the script created to carry out complex commands. There is `definitions.tcl` script in the `scripts` directory. To source the script, write:

```
source ../scripts/definitions.tcl
```

Alternatively (and recommended the first time you run the flow), you can also run all steps of the definitions.tcl script manually as follows.

(a) To create the Milkyway design library, Choose File > Create Library.

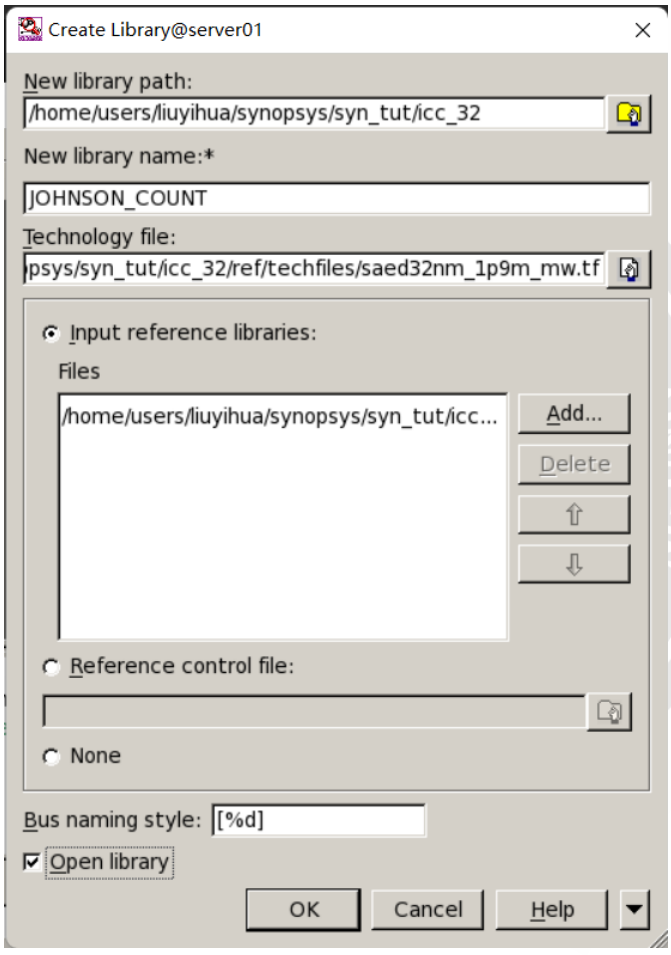


Figure 5: Create library window.

The `create_mw_lib` command and the Create Library dialog box have options to specify the name of the new Milkyway library, the name of the associated technology file, the names of the associated Milkyway reference libraries, and the bus naming style. *Remember to check "Open library". The warnings shown in the output can be ignored*:

```
Start to load technology file /home/users/<username>/synopsys/s
 ↪  yn_tut/icc_32/ref/techfiles/saed32nm_1p9m_mw.tf.
Warning: Layer 'M1' has a pitch 0.152 that does not match the
 ↪  recommended wire-to-via pitch 0.13 or 0.105. (TFCHK-049)
Warning: Layer 'M2' has a pitch 0.152 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M3' has a pitch 0.304 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M4' has a pitch 0.304 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M5' has a pitch 0.608 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M6' has a pitch 0.608 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M7' has a pitch 1.216 that does not match the
 ↪  recommended wire-to-via pitch 0.139. (TFCHK-049)
Warning: Layer 'M8' has a pitch 1.216 that does not match the
 ↪  recommended wire-to-via pitch 0.179 or 0.164. (TFCHK-049)
Warning: Layer 'M9' has a pitch 2.432 that does not match the
 ↪  recommended wire-to-via pitch 1.74. (TFCHK-049)
Warning: Layer 'MRDL' has a pitch 4.864 that does not match the
 ↪  recommended wire-to-via pitch 4.5. (TFCHK-049)
Warning: Layer 'MRDL' has a pitch 4.864 that does not match the
 ↪  doubled pitch 2.432 or tripled pitch 3.648. (TFCHK-050)
Warning: CapModel sections are missing. Capacitance models
 ↪  should be loaded with a TLU+ file later. (TFCHK-084)
Technology file /home/users/<username>/synopsys/syn_tut/icc_32/
 ↪  ref/techfiles/saed32nm_1p9m_mw.tf has been loaded
 ↪  successfully.
{JOHNSON_COUNT}
```

(b) TLUPlus is a binary table format that stores the RC coefficients. The TLUPlus models enable accurate RC extraction results by including the effects of width, space, density, and temperature on the resistance coefficients. Choose File > Set TLU+ (Figure 6).

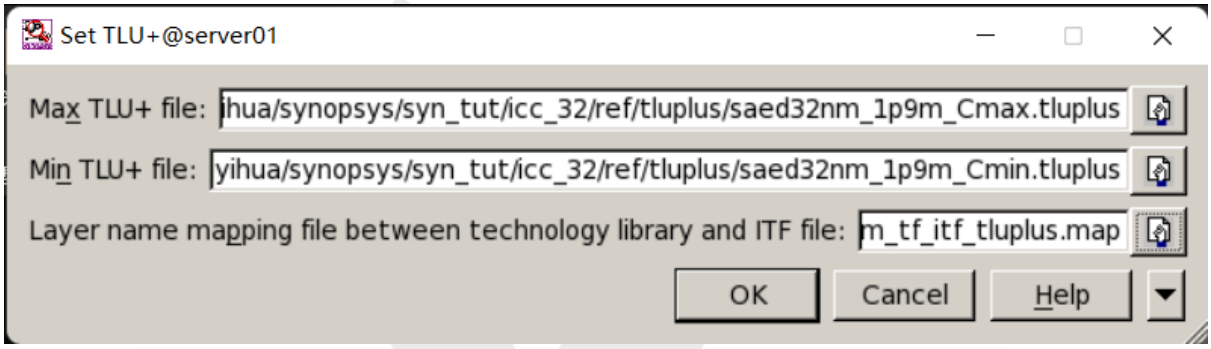


Figure 6: Set TLU+ window.

Type or browse for the map file in the "Map file" text box. Type or browse for the maximum and minimum TLUPlus model files.

(c) Import Design - Read the Verilog file for the design by choosing File > Import > Read Verilog... and specifying the Verilog netlist file, which is a gate-level design in one or more files (Figure 7 and Figure 8).

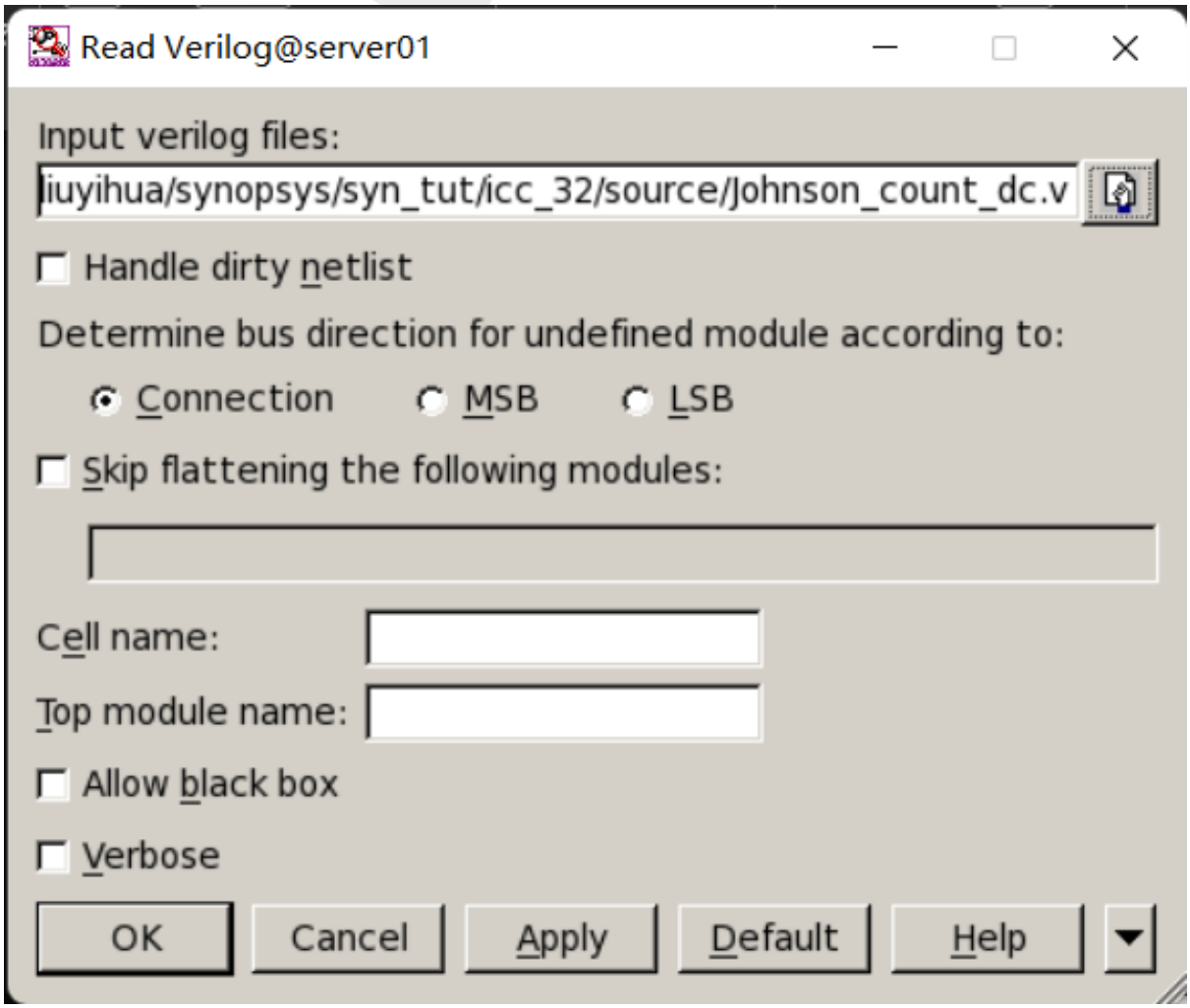


Figure 7: Import design window.

If `import_designs` failed with errors, please go to Appendix B.1. The Cell name and the Top module name can be left empty as IC Compiler will automatically specify them from the Input Verilog files.

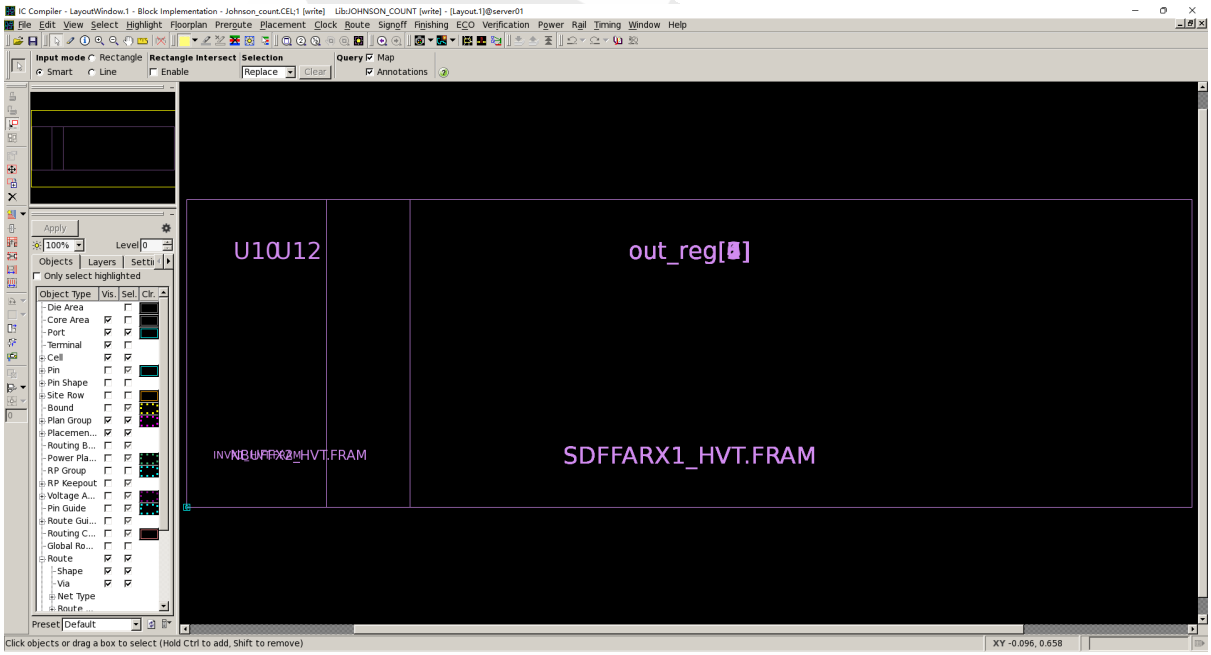


Figure 8: Design view after importing.

(d) To source constraints with .sdc (Synopsys Design Constraints) format:

```
icc_shell> source ../source/constr.sdc
```

3. Use script created to carry out complex commands. There is a script in `scripts` directory. The name of this script: `design_all.tcl`. To source the script, write:

   ```
   icc_shell> source ../scripts/design_all.tcl
   ```

   Now show the steps of `design_all.tcl` script. Note that commands in the script and following steps with GUI are not identical, even though they yield similar layouts.

   - Initialize floorplan; declare power nets and pins
   - Create rectangular rings
   - Create power straps
   - Core Placement and Optimization
   - Core CTS (Clock Tree synthesis) and Optimization
   - Preroute standard cells
   - Core Routing and Optimization
   - Inserting Fillers

(a) Floorplan information includes the core area, top-level ports, and placement sites.
For floorplanning the design choose Floorplan->Create Floorplan... (Figure 9 and Figure 10).

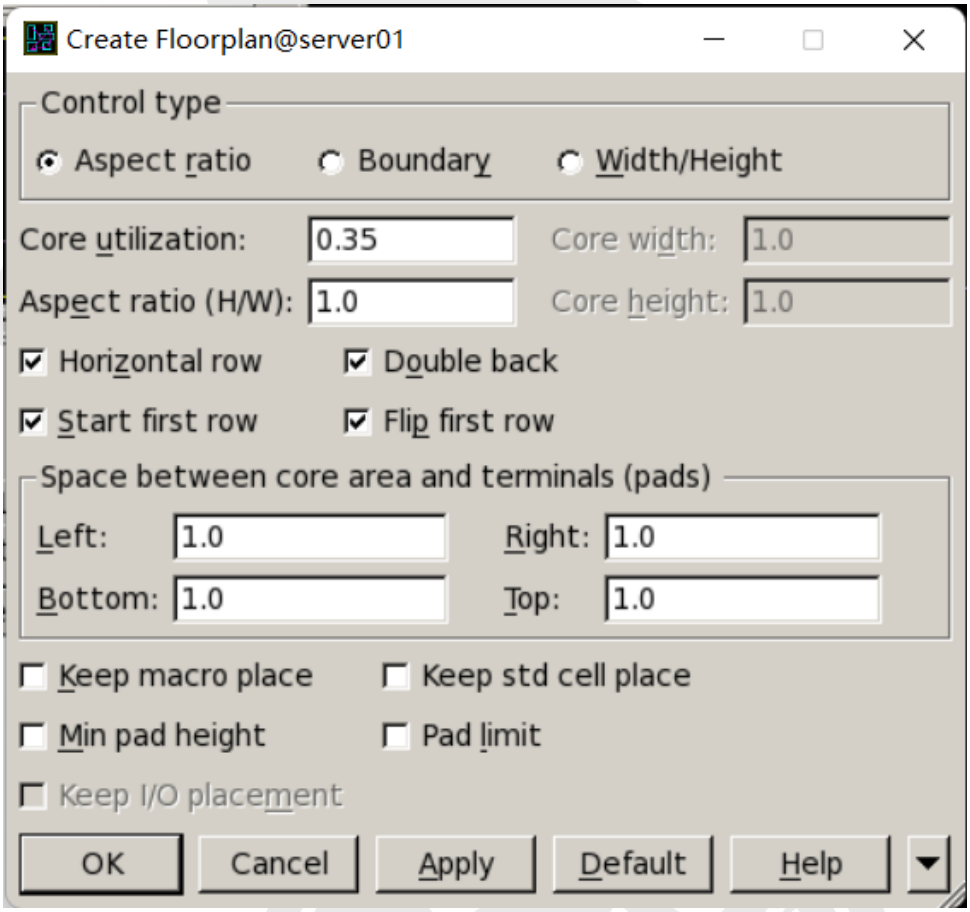


Figure 9: Initialize floorplan window.

Specify the control parameters.
Indicate the method which specifies the size of the core area:

i. Aspect ratio – A ratio of height divided by width (the default)

ii. Width & height – The exact width and height

iii. Row number – A number of rows

iv. Boundary – A fixed size according to the boundary defined in design planning

If the design view does not show the name labels, it is because the name labeling is set to "Both" by default, and there is not enough space to show both the instance name and the reference name. You can zoom in the design view to see the full labels. To only show the instance name, go to the "Settings" tab-

>"Cells" tab, and in the "Name labeling" frame, select "Instance name". You can save the design and reopen it by

```
open_mw_lib
 ↪  /home/users/<username>/synopsys/syn_tut/icc_32/JOHNSON_COUNT
::iccGUI::open_mw_cel  Johnson_count
open_mw_cel Johnson_count
```

If you failed to open the design with error

```
open_mw_cel Johnson_count
INFO: cell is locked by <username> (pid 446498 server01) check
 ↪  again...
INFO: cell is locked by <username> (pid 446498 server01) check
 ↪  again...
INFO: cell is locked by <username> (pid 446498 server01) check
 ↪  again...
INFO: cell is locked by <username> (pid 446498 server01) check
 ↪  again...
INFO: cell is locked by <username> (pid 446498 server01) check
 ↪  again...
INFO: Could not lock the cell for write, giving up
INFO: Please remove the lock to proceed.
Error: Failed to open design:'Johnson_count.CEL;0'for 'w'.
 ↪  (MWUI-007)
```

Please remove `/home/users/<username>/synopsys/syn_tut/icc_32/JOHNSON_COUNT/CEL/Johnson_count:1.lock` file.

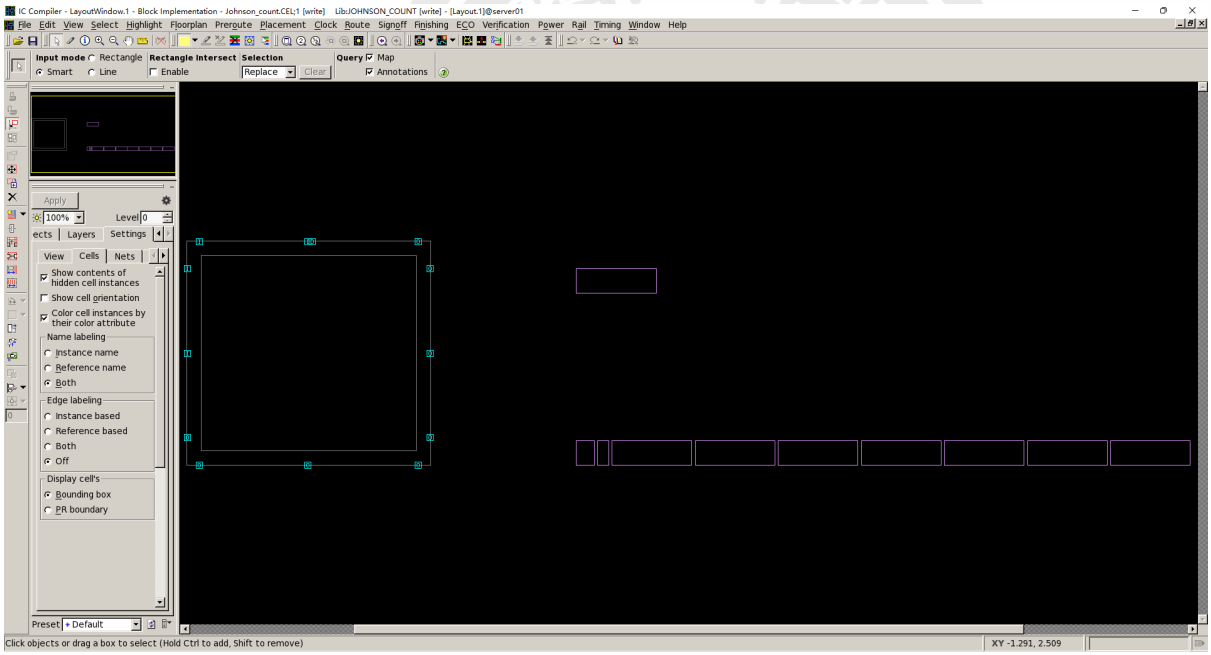


Figure 10: Design view after `initialize_floorplanning`.

Declare power nets and pins.
Type into the ICC shell the following:

```
set power "VDD"
set ground "VSS"
set powerPort "VDD"
set groundPort "VSS"
set mw_logic0_net "VSS"
set mw_logic1_net "VDD"
derive_pg_connection -power_net VDD -ground_net VSS -power_pin
 ↪  VDD -ground_pin VSS
```

(b) Add power and ground rings.
To add power and ground rings, choose PreRoute > Create Rings. Select the Rectangular tab (Figure 11 and Figure 12)

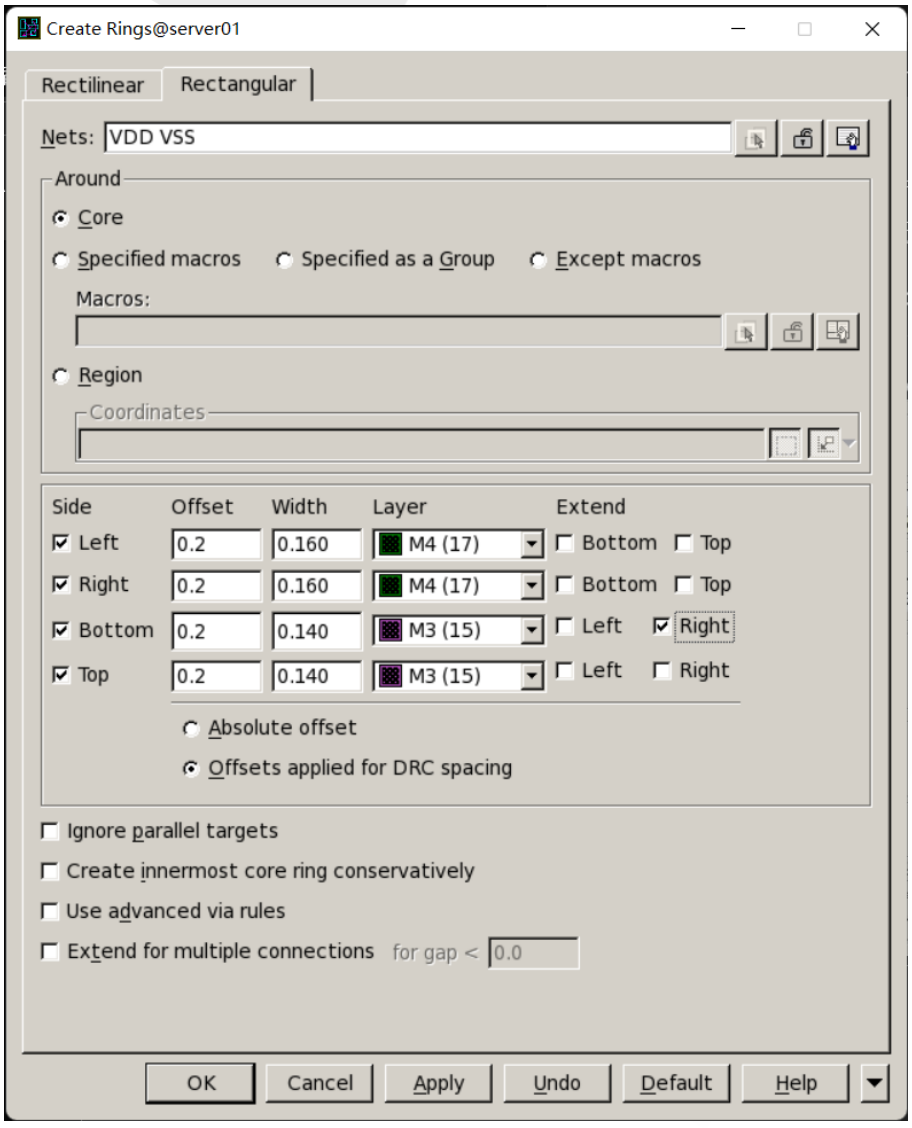


Figure 11: Create rectangular rings window.

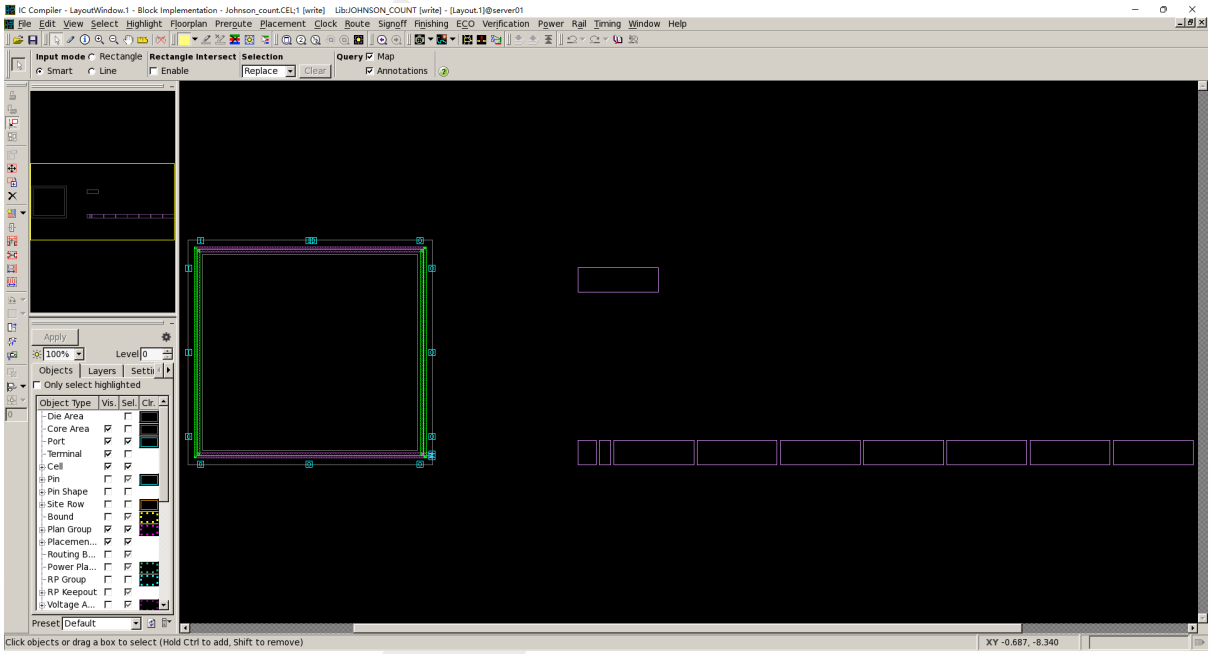


Figure 12: Design view after the creation of rectangular rings.

(c) After the creation of rectangular rings, the straps are automatically connected to the closest power and ground ring at, or beyond, both ends of the straps.

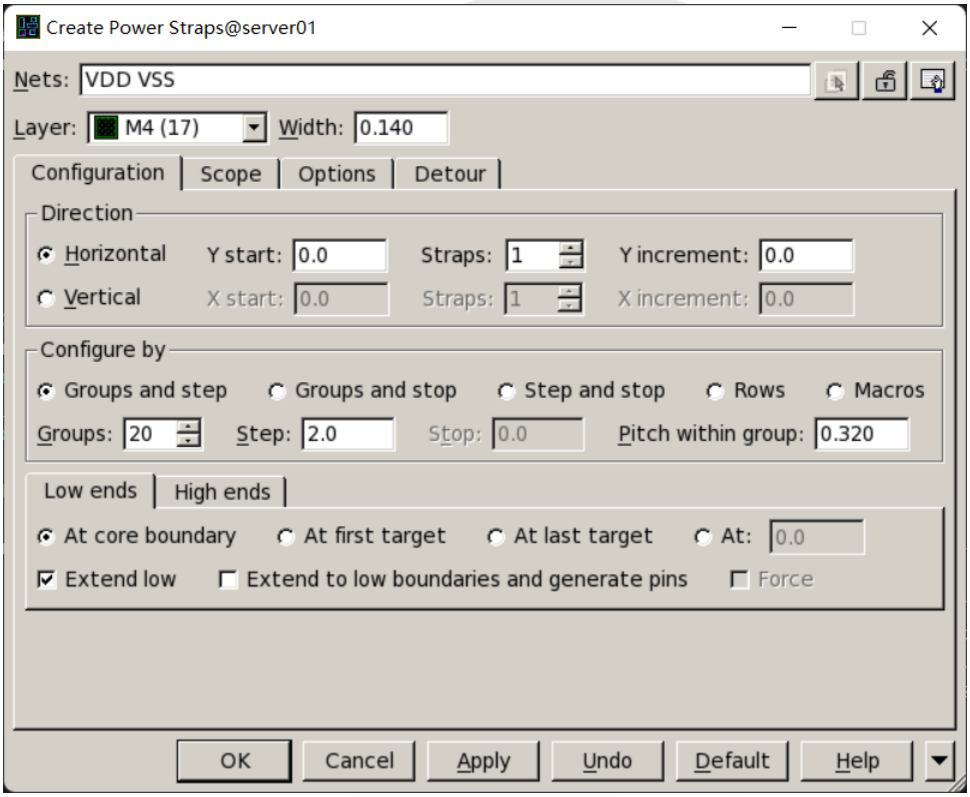


Figure 13: Create object tool window.

The `create_power_straps` command creates power straps in a design. Use a few wide straps rather than many thin straps to improve the placement quality and decrease the placement runtime.
The same as from the Menu bar, choose PreRoute > Create Power Straps... (Figure 13 and Figure 14).
Choose vertical and horizontal metal layers accordingly, and do vertical and horizontal power straps separately.

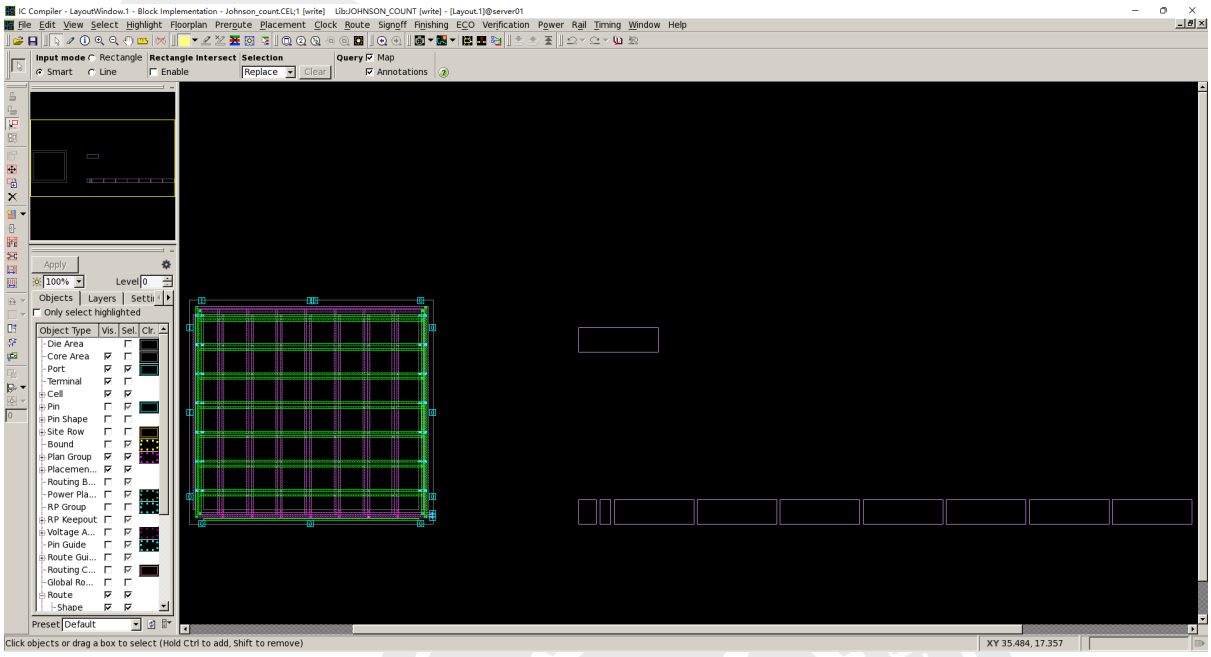


Figure 14: Design view after creation power straps.

You need to choose the core utilization, groups, and the step in a proper range in order to fit the floorplan. Please show your choice.

(d) `place_opt` command performs simulation placement, routing, and optimization on the current design. During the placement phase, this design's standard cells have been automatically placed in horizontal placement rows.

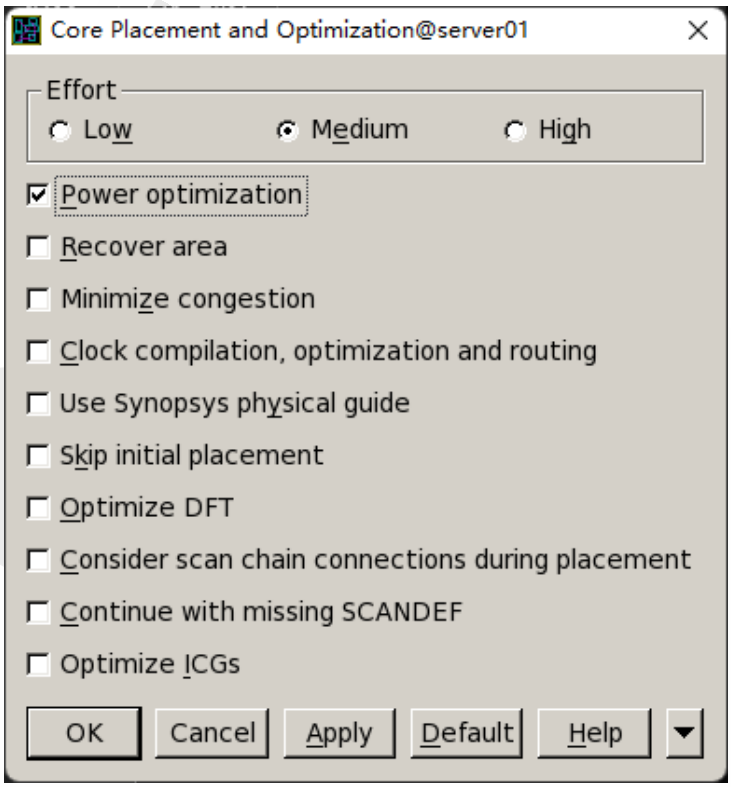


Figure 15: Core placement and optimization window.

To run placement, choose Placement > Core Placement and Optimization... (Figure 15 and Figure 16)

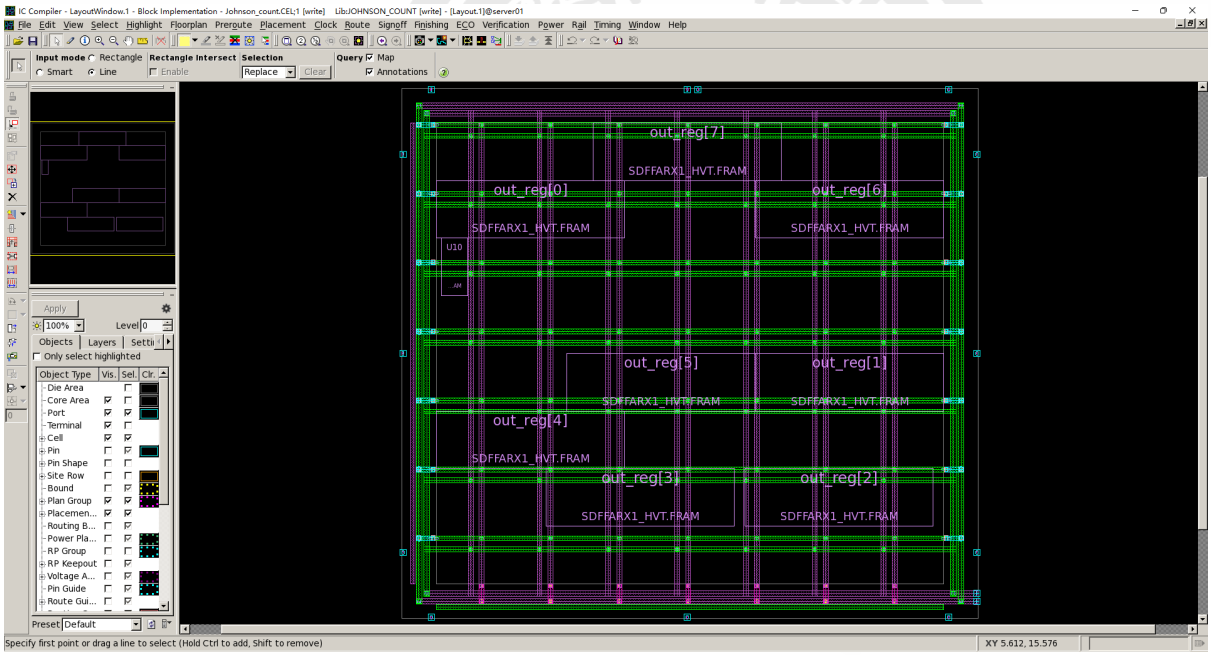
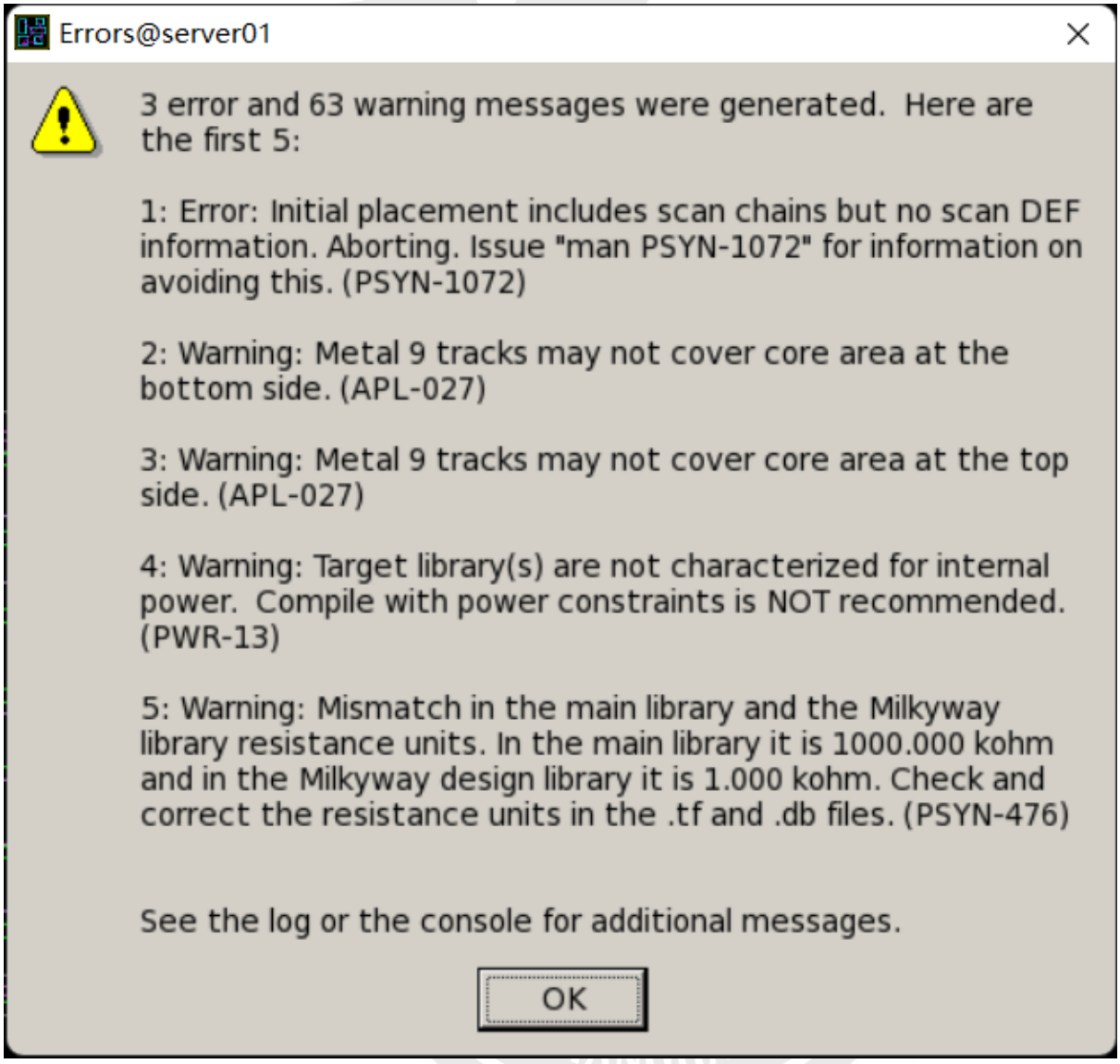


Figure 16: Design view after Core Placement and Optimization.

The "Core Placement and Optimization@server01" dialog box appears. Select "Power optimization". *The effort can be medium or high; also select "Optimize DFT" or "Continue with missing SCANDEF", and optionally "Minimize congestion"; otherwise, you will fail with errors*:

```
Error: Initial placement includes scan chains but no scan DEF
↪  information. Aborting. Issue "man PSYN-1072" for information
↪  on avoiding this. (PSYN-1072)
```



The documentation can be accessed by `man PSYN-1072`. *Or you can also start from the Placement menu->Coarse Placement... and also check the necessary options of configuration. Selecting the effort to high produces less density. If the density is too high, it will fail with errors*:

```
Warning: Density is 99.9% (PSYN-1010)
```

```
Initial legalization:  10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
↪  (0 sec)
Error: Could not find a legal placement.
Legalization complete (0 total sec)
Top 5 lib cells with lowest legal rate;
Error: A legal placement could not be found (PSYN-044)
```

and

```
Warning: Metal 9 tracks may not cover core area at the bottom
↪  side (APL-027)
Warning: Metal 9 tracks may not cover core area at the top side
↪  (APL-027)
Error: An error has occurred in the execution of the detailed
↪  placer. (PSYN-060)

  Placement Optimization complete
  -------------------------------

Error: psynopt has abnormally terminated.  (OPT-100)
Warning: ICC has suggestions for improving your design.  Use
↪  "report_suggestions" for details.  (PSYN-1067)
```

(e) Clock tree synthesis. During clock tree synthesis, IC Compiler builds clock trees that meet the clock tree design rule constraints while balancing the loads and minimizing the clock skew. IC Compiler fixes the placement of the clock sinks, performs incremental logic and placement optimization, and fixes the placement of both the buffers and registers on the clock tree. The `clock_opt` command performs clock tree synthesis, routing of clock nets, extraction, optimization, and optionally hold-time violation fixing on the current design. If clock tree synthesis fails or the routing of clock nets fails, the command returns a value of 0.

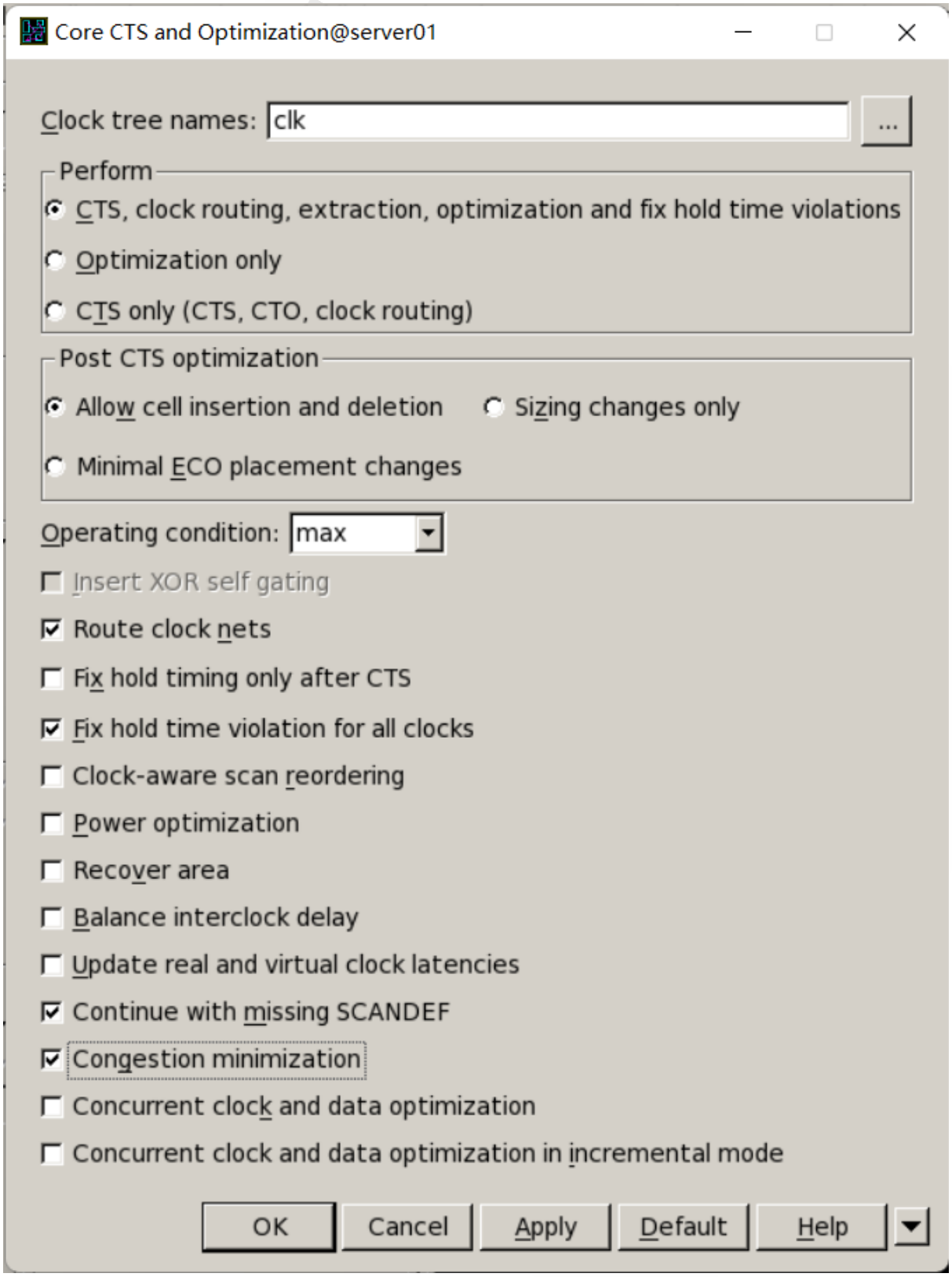
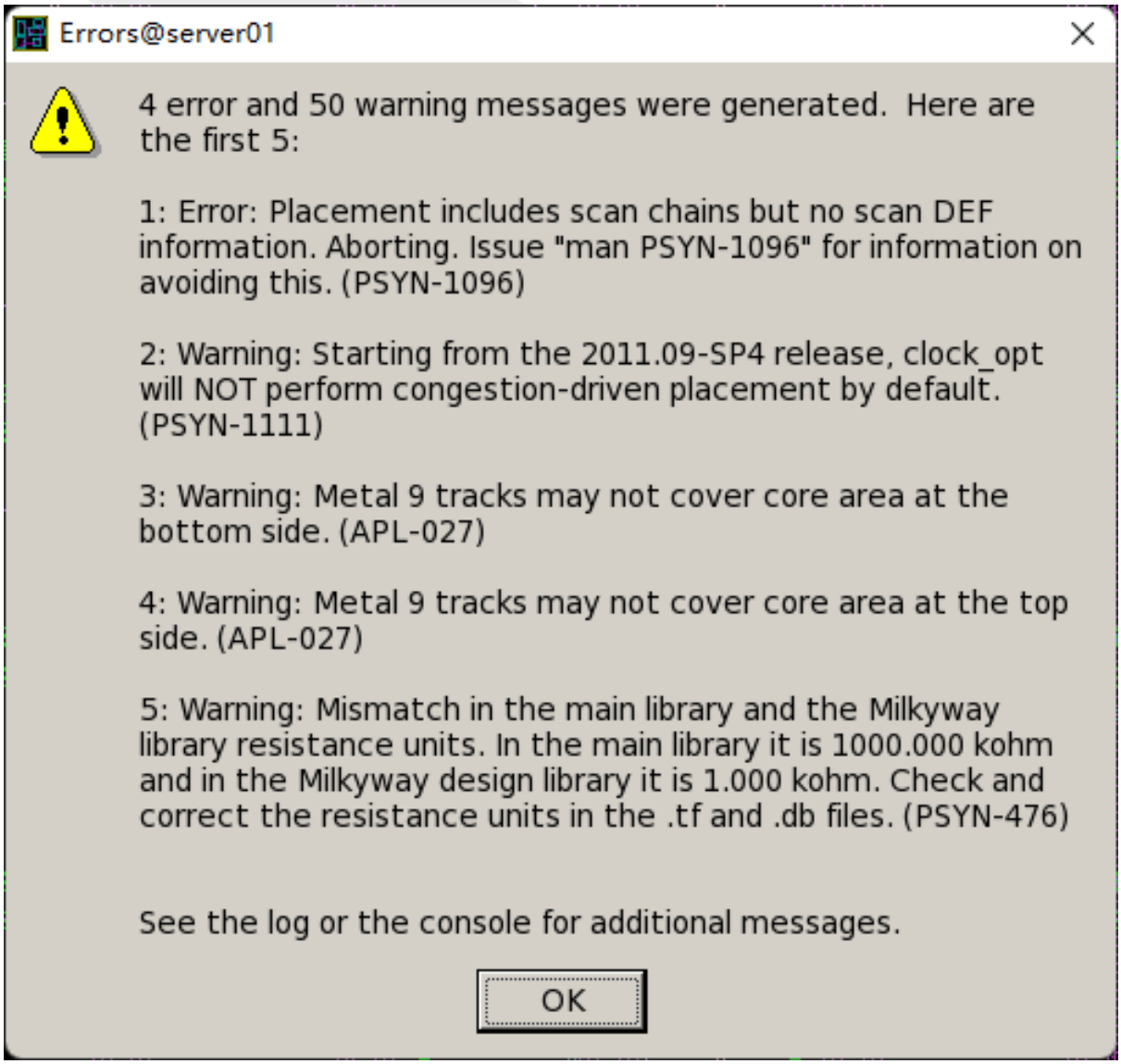


Figure 17: Core CTS and optimization window.

To perform clock tree synthesis and optimization, use the `clock_opt` com-

mand or choose Clock > Core CTS and Optimization... in the GUI (Figure 17 and Figure 18).
The "Core CTS and Optimization@server01" dialog box appears, and select route clock cells. *In the Perform frame, you can select "CTS, clock routing, extraction, optimization and fix hold time violations" or "Optimize only" or previously "Incremental optimization only (clock routing, extraction, optimization, and fix hold time)". You can also optionally check "Congestion minimization" and "Fix hold time violation for all clocks". Also, remember to check "Continue with missing SCANDEF"; otherwise, it will fail with errors*:

```
Error: Placement includes scan chains but no scan DEF
↪ information. Aborting. Issue "man PSYN-1096" for information
↪ on avoiding this. (PSYN-1096)
```



You need to make your own choices to produce a valid design.

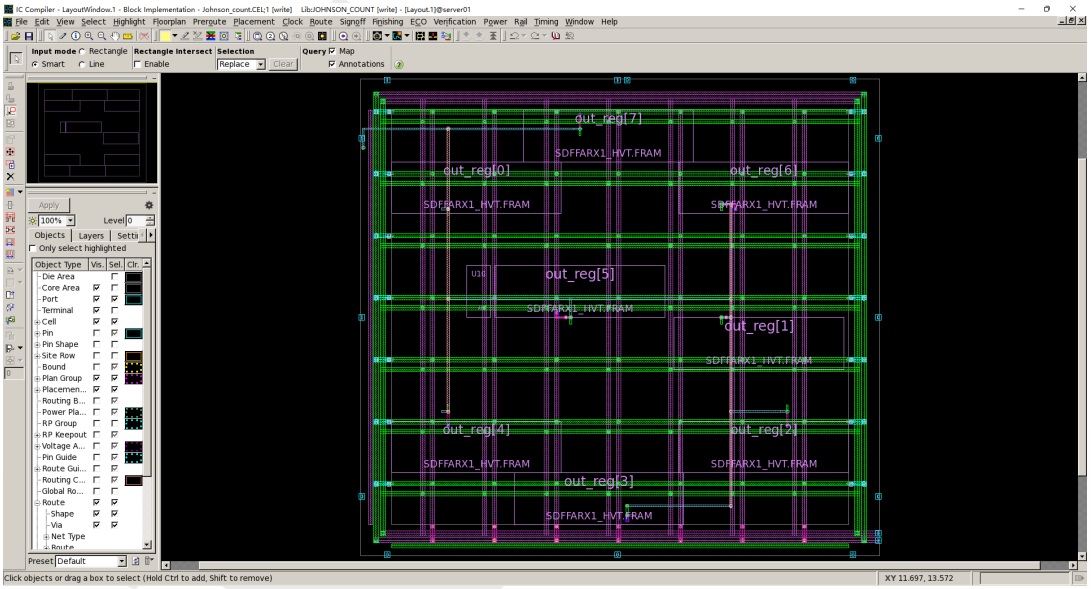


Figure 18: Design view after Core CTS and Optimization.

(f) Connect power and ground pins in standard cells to the straps and rings and connect power and ground rails in the standard cells. To make sure the global router can recognize the routing obstruction, preroute the standard cells before performing global routing.
The `preroute_standard_cells` command connects power and ground pins in the standard cells to the power and ground rings or straps and connects power and ground rails in the standard cells.

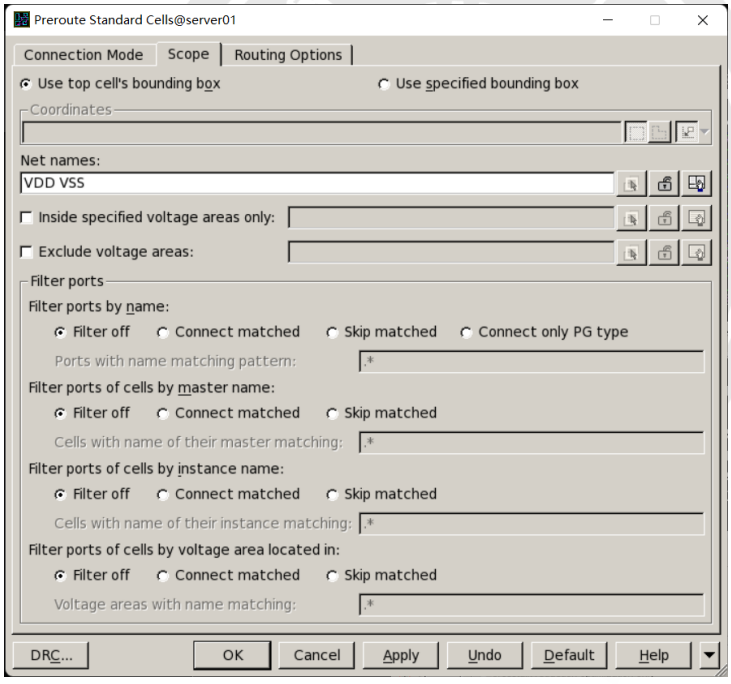


Figure 19: Preroute standard cells window.

The same as from the Menu bar, choose PreRoute > Preroute Standard Cells... (Figure 19 and Figure 20).

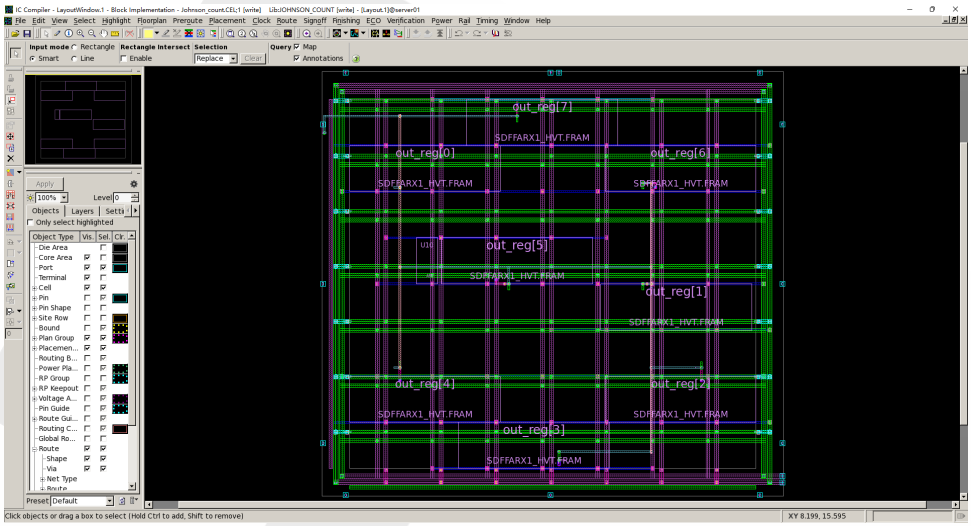


Figure 20: Design view after Prerouting Standard Cells.

(g) The `route_opt` this command performs simultaneous routing and postroute optimization on the current design. The output of this command is a postroute-optimized design.

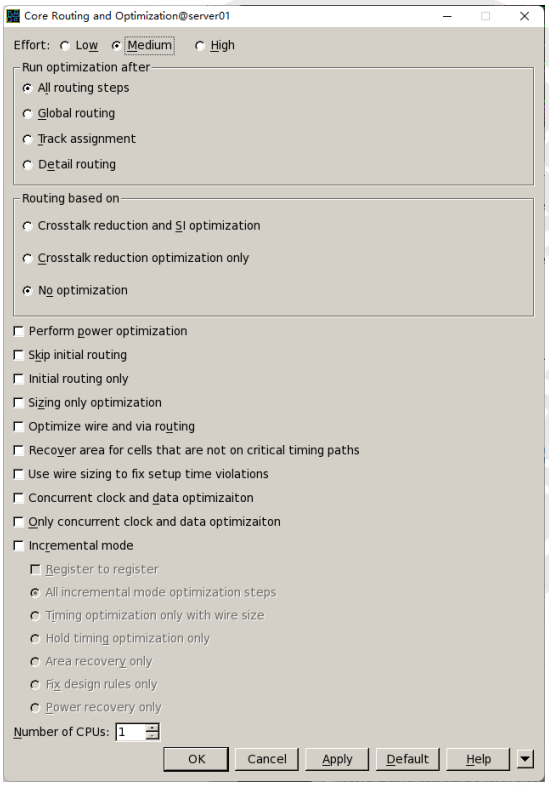


Figure 21: Core routing and optimization window.

Finally, for routing the design, choose Route->Core Routing and Optimization. The "core routing and optimization" dialog box appears (Figure 21 and Figure 22).

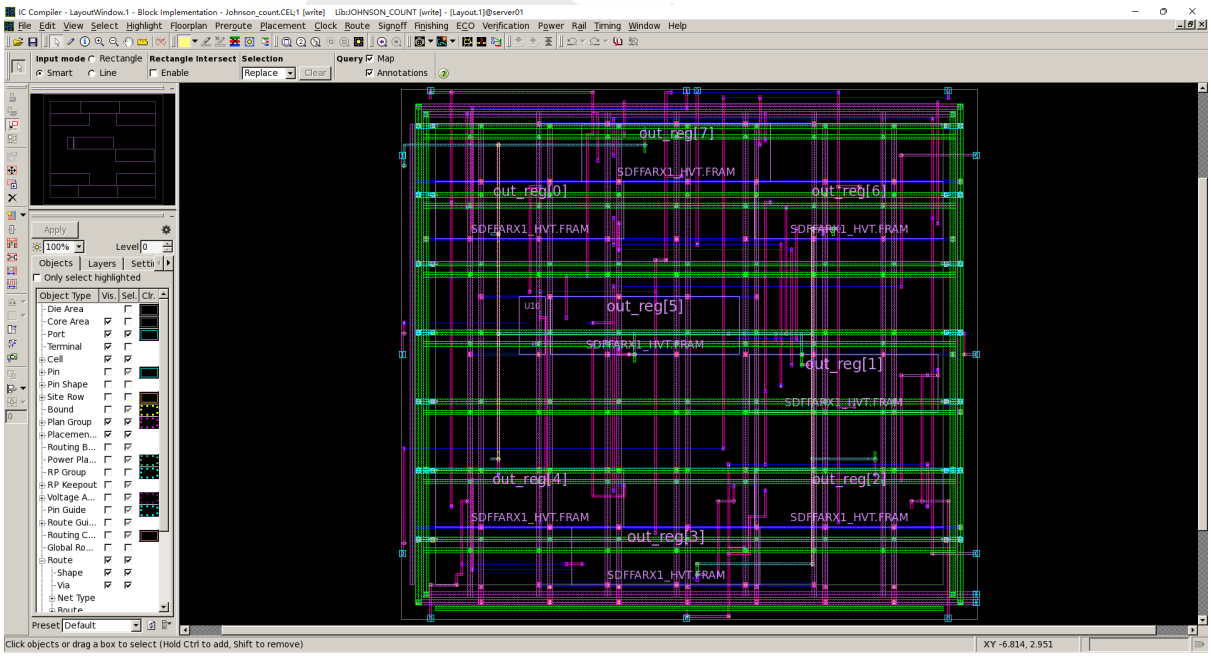


Figure 22: Design view after core routing and optimization.

ICC has `route_eco` command. This command performs routing for the broken nets. It runs the global route, track assignment, and detail route.

(h) `insert_stdcell_filler` this command fills empty spaces in standard cell rows with instances of master filler cells in the library (Figure 23).

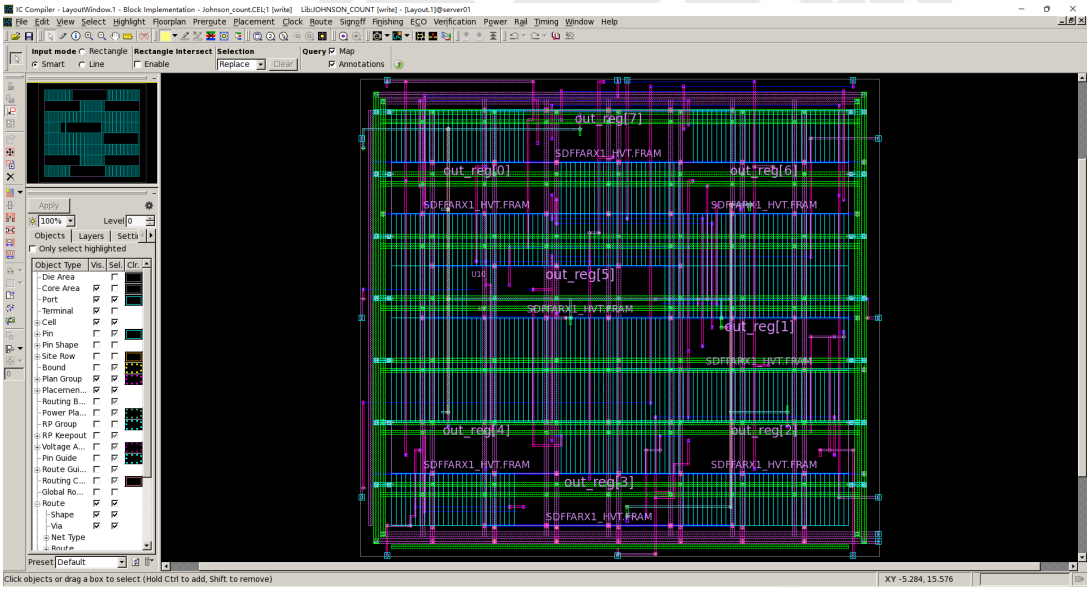
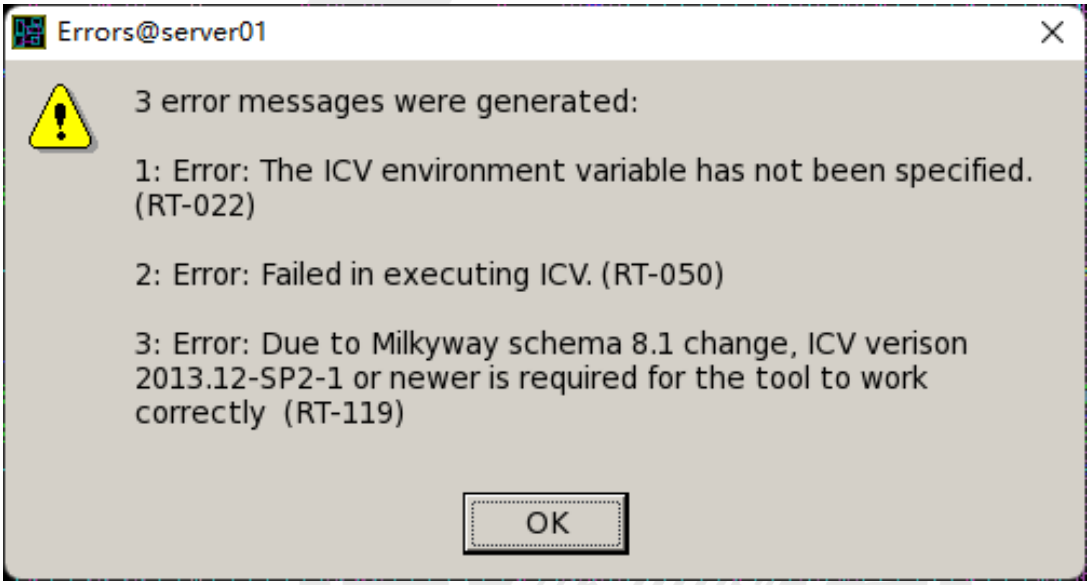


Figure 23: Design view after insert fillers.

Make sure you are inserting correct stdcell filters as for your library. If not, you may get errors:

```
Error: Nothing matched for collection (SEL-005)
ignorePGRail 1 ignoreDpt 1 ignoreBetweenFillers 1
User specify 0 filler masters
```

4. IC Compiler allows checking DRC (Design Rule Checking) and LVS (layout-versus-schematic). To check DRC errors, choose Verification > Signoff DRC... *The DRC requires Synopsys IC Validator (`icv`), which is, however, not set up on the server; thus, we have to skip this step; instead, we should run `extract_rc` to extract the design and derive parasitics.*




```
3 error messages were generated:
1: Error: The ICV environment variable has not been specified.
↪  (RT-022)
2. Error: Failed in executing ICV. (RT-050)
3. Error: Due to Milkyway schema 8.1 change, ICV version
↪  2013.12-SP2-1 or newer is required for the tool to work
↪  correctly (RT-119)
```

To check LVS errors, choose Verification > LVS (Figure 24).

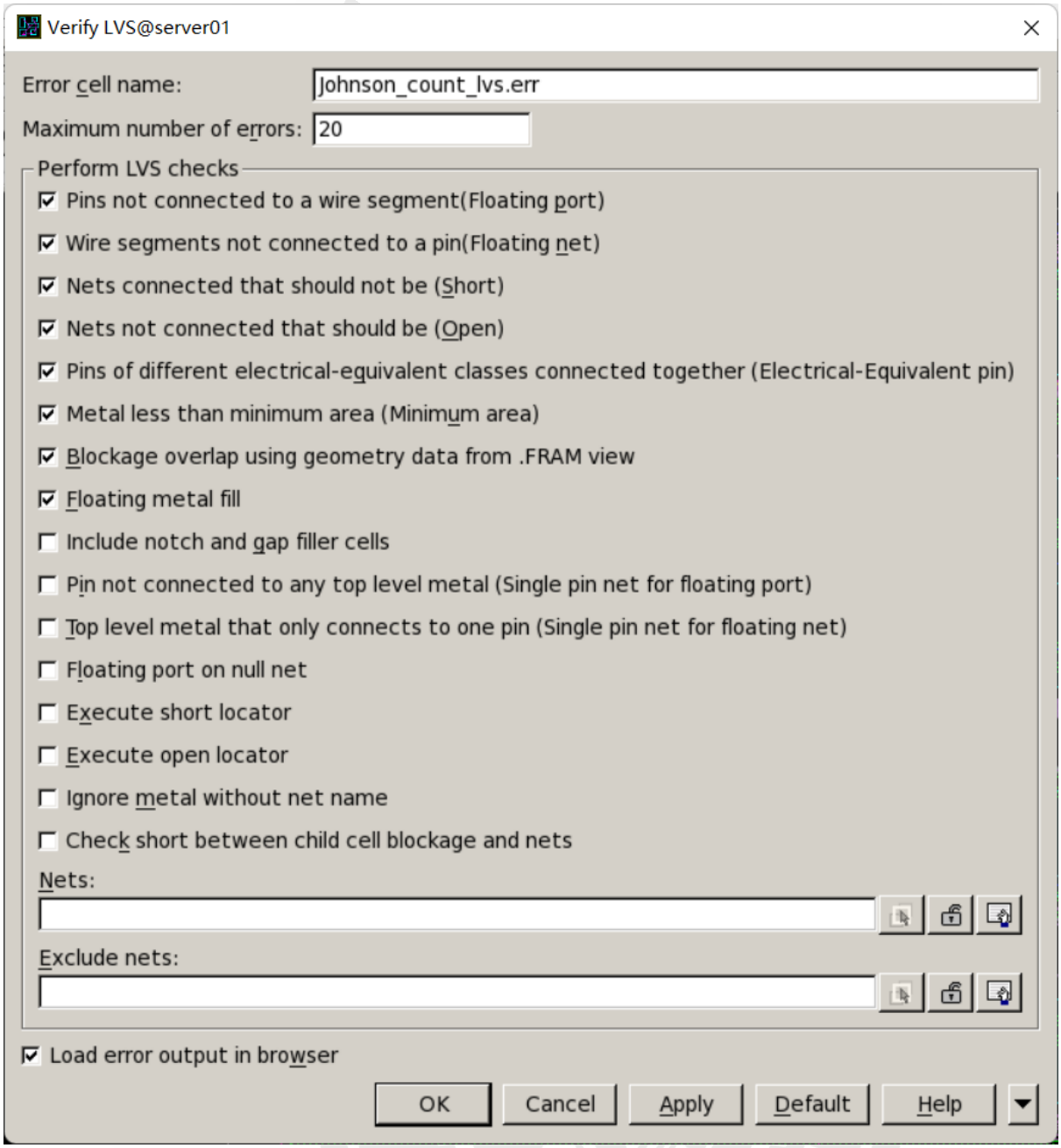


Figure 24: Check LVS window.

Take a screenshot of your Error Browser@server01 and a screenshot of your final design view.

5. Write the design in the following formats:

```
icc_shell> write_verilog -pg -no_physical_only_cells
↪  ../results/Johnson_count.v
```

This command generates a Verilog netlist for the current Milkyway design. By specifying `-pg` and `-no_physical_only_cells` options.
`-pg` - Output power and ground nets and ports for all cell instances and module

instances.
`-no_physical_only_cells` - Do not output physical-only cell instances. The same as from the Menu bar, choose File > Export > Write Verilog... (Figure 25).

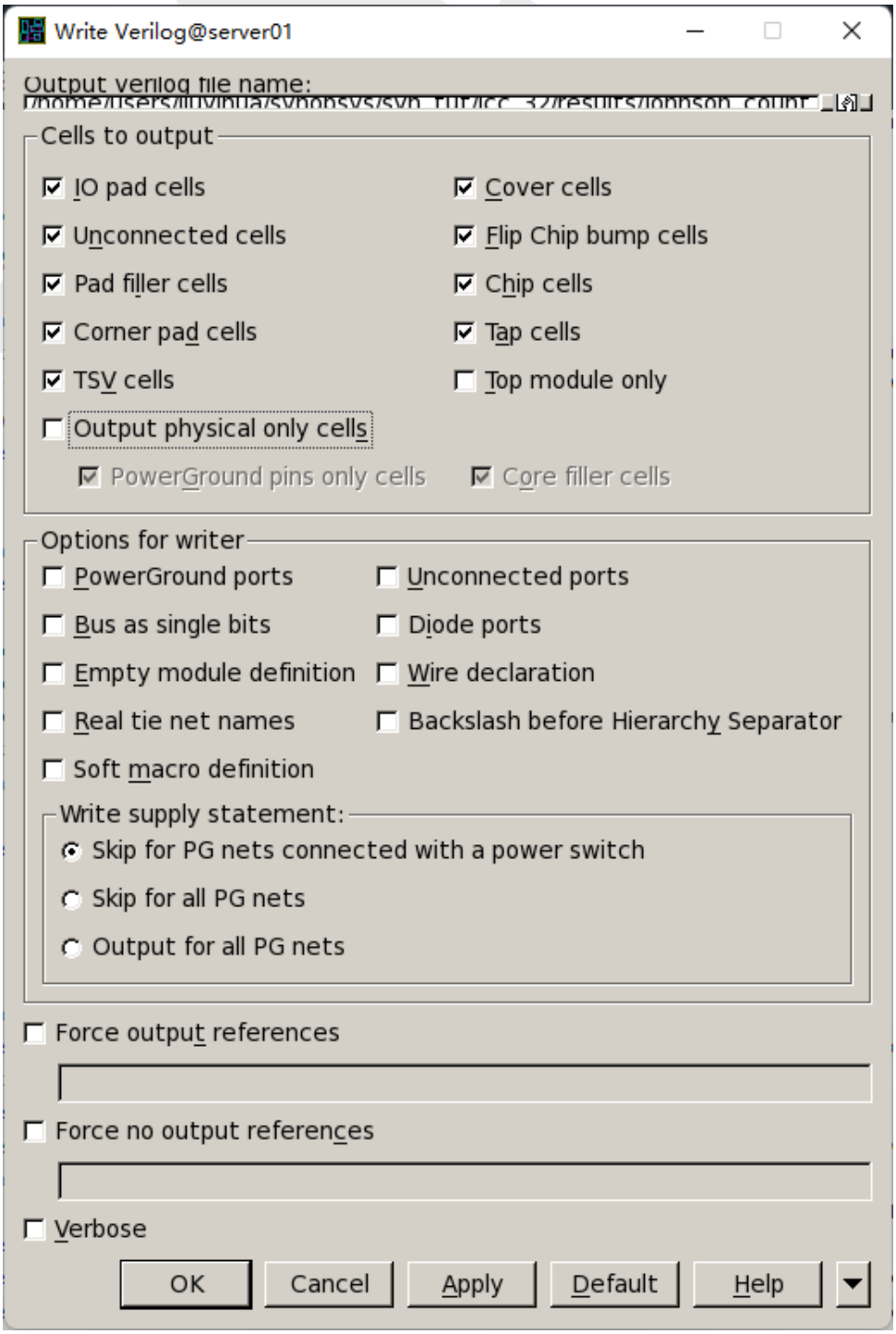


Figure 25: Write Verilog format box.

To write without PowerGround ports Verilog format the following this command:

```
icc_shell> write_verilog -pg -no_physical_only_cells
↪  ../results/Johnson_count.v
```

To write the design with Standard Parasitic Exchange Format (SPEF), use the following command:

```
icc_shell> write_parasitics -output {../results/Johnson_count.spf}
```

This command writes parasitics for the current design to a disk file.
`-output`: Specifies the name of the output file to which parasitics for the current design are written.
The same as from the Menu bar, choose File > Export > Write Parasitics (Figure 26).

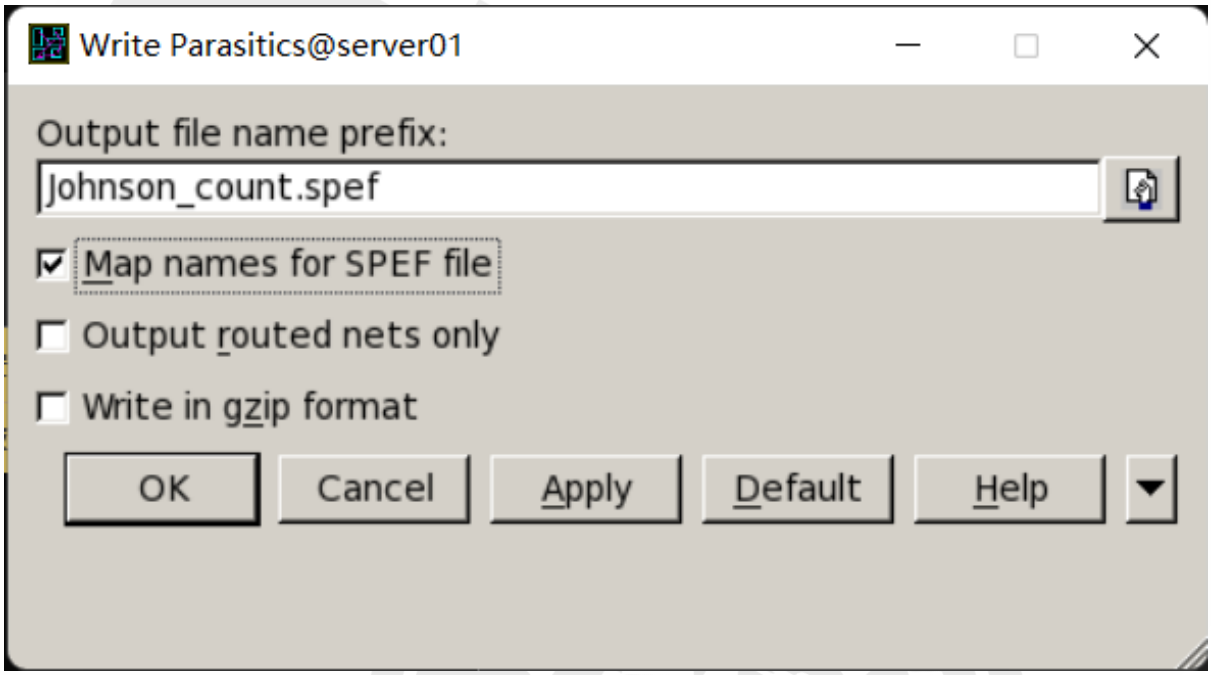


Figure 26: Write .spef format box.

Command `write_stream` writes the data in the specified library to a file in GDS format. The options are set by `set_write_stream_options`.

```
set_write_stream_options -map_layer ../ref/saed90nm.gdsout.map
↪  -output_filling fill        -child_depth
↪  20        -output_outdated_fill -output_pin {text geometry}
write_stream -lib COUNT -format gds -cells Johnson_count
↪  ../results/Johnson_count.gds
```

`-map_layer` Specifies the file that defines the layer mapping from Milkyway to GDSII.
`-output_filling` Specifies the types of filling data to output.
`-child_depth 20` Specifies the hierarchical level to output child cells. To export all child cells, specify a large number, such as 20.

`-output_outdated_fill` Forces the output of fill data even if the fill data is out-of-date.
`-output_pin` Specifies the object types to output for each pin.
The same as from the Menu bar, choose File > Export > Write Stream (Figure 27).

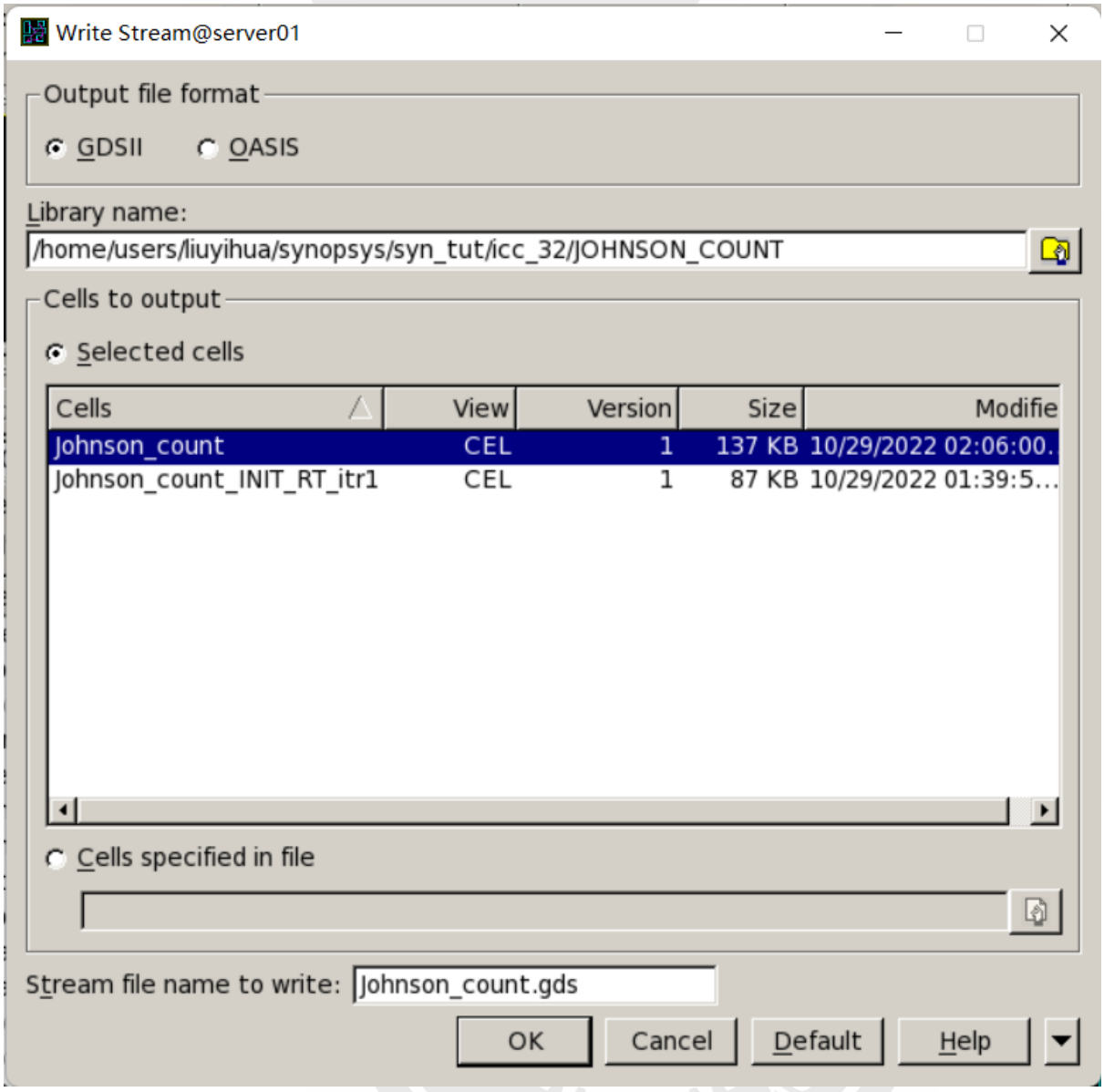


Figure 27: Write .gds format box.

In this design, output results are stored in ../.location/results/ directory.

6. To exit IC Compiler write exit in the command line.

```
exit
```

3 Formal Verification

3.1 Introduction

The purpose of Formality is to detect unexpected differences that may have been introduced into a design during development.

For this lab, you need to copy `/home/Flow/Synopsys/Scripts/formality_32` to the `syn_tut` directory.

```
cd ~/synopsys/syn_tut
cp -r /home/Flow/Synopsys/Scripts/formality_32/ .
cd formality_32
cd pre_lay
```

Before we start Formality, we're going to create a link to the database that we'll be using (we'll need this later). To create this link, navigate to the scripts directory and run the `link_to_database.sh` script:

```
cd scripts
./link_to_database.sh
```

You may need to modify the script if needed. In this lab, we will use Synopsys Formality (R) Version R-2020.09-SP3 for linux64.

To start Formality (as usual use the work directory), enter the following command at the terminal (here we verify the prelayout netlist, for postlayout you just need to use the postlay directory):

```
cd ../work
fm_shell
```

The `fm_shell` command starts the Formality shell environment. From here, start the graphical user interface (GUI) as follows:

```
fm_shell (setup)> start_gui
```

Alternatively, To Start Formality in GUI mode you can also enter:

```
fm_shell -gui
```

This opens the Formality top-level GUI window.

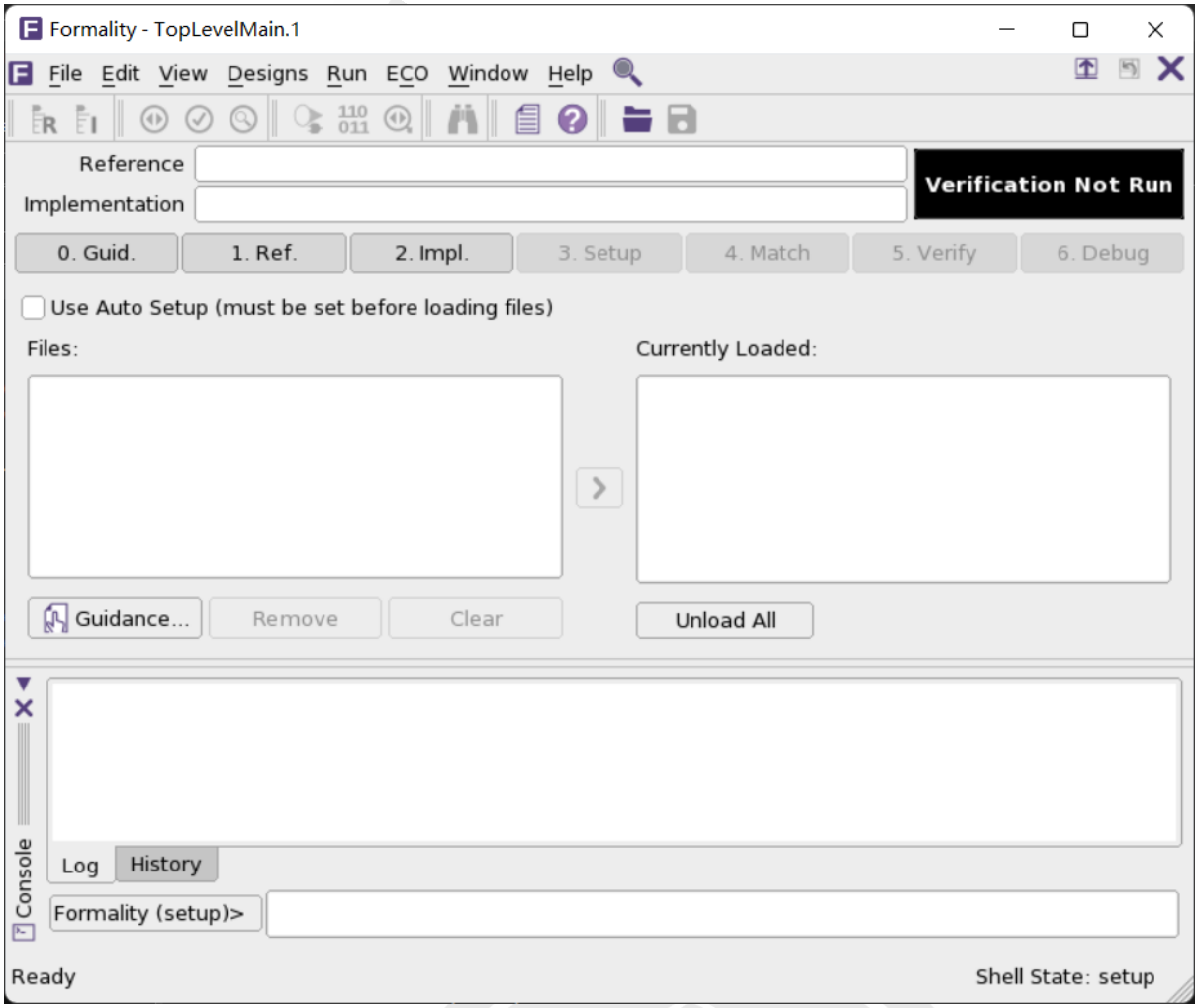


Figure 28: The Main Window of Formality GUI.

In Formality, the following concepts are used:

- Reference design: This design is the golden design, the standard against which Formality tests for equivalence.
- Implementation design: This design is the changed design. It is the design whose correctness you want to prove. For example, a newly synthesized design is an implementation of the source RTL design.
- The setup indicates the mode that is currently in when using commands. The modes are "setup", "match", and "verify".

When Formality is invoked, begin in the setup mode.

3.2 Tasks

If Formality is given the result of DC, then work in the `pre_lay` directory, and if Formality is given the result of ICC, then work in the `post_lay` directory.

This lab is an example where Formality works with the result of DC. Therefore, use the `pre_lay` directory. Running Formality with the result of ICC is similar to running these lab steps.

To understand Formality's general sections, you can also execute the commands step-by-step (recommended for first-time users).

3.2.1 Guidance (Load Automated Setup File)

Before specifying the reference and implementation designs, you can optionally load an automated setup file (.svf) into Formality. The automated setup file helps Formality process design changes caused by other tools used in the design flow. Formality uses this file to assist the compare point matching and verification process. For each automated setup file that is loaded, Formality processes the content and stores the information for use during the name-based compare point matching period.

For this Lab, let us Load the `default.svf` file.

1. In "0. Guid." click Guidance...
2. Locate the `default.svf` file within the Source Directory
3. Click Open
4. Click the right arrow button to load files

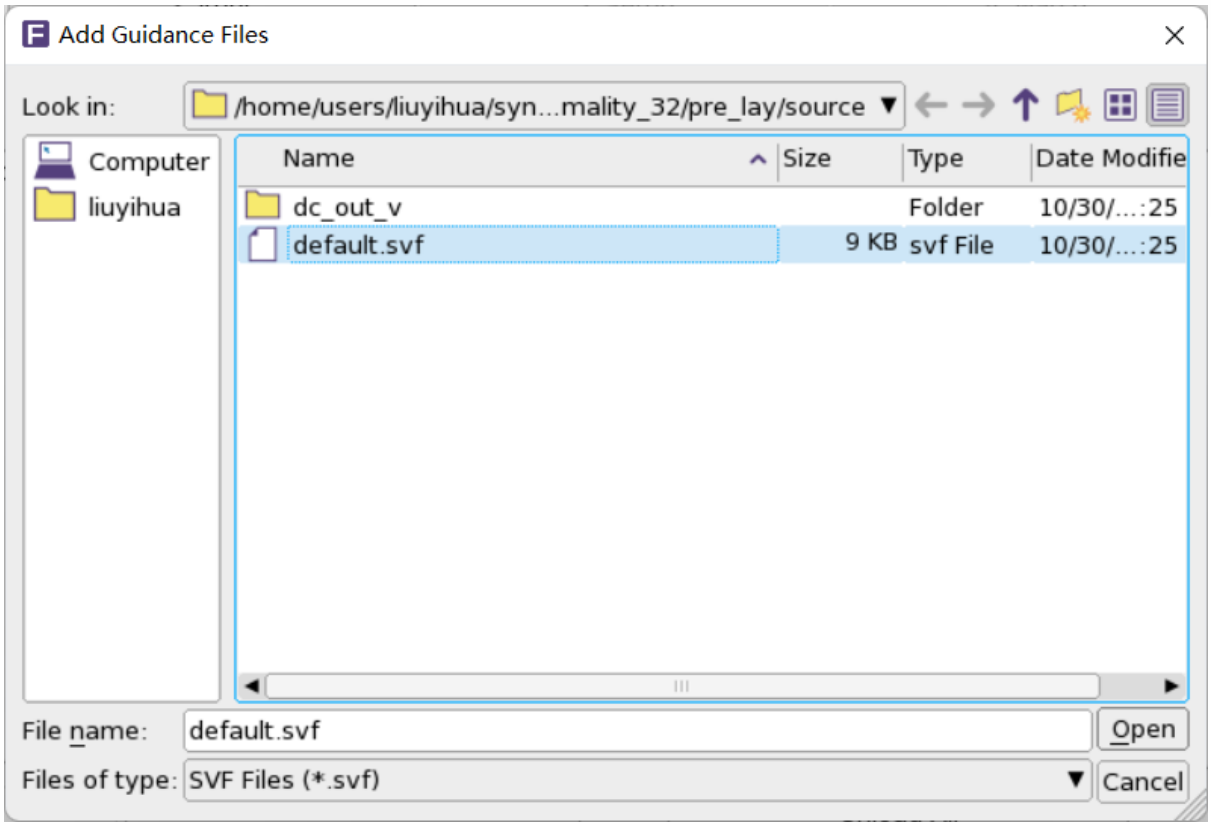


Figure 29: Loading Guidance.

```
set_svf -append { /home/users/<username>/synopsys/syn_tut/formality_32/p
↪ re_lay/source/default.svf
↪ }
```

3.2.2 Reference (Specify the Reference Design)

Specifying the reference design involves the reading in of design files, optionally reading in technology libraries, and setting the top-level design. The reference design is the design with which the transformed (implementation) design is compared. Click the "1. Ref." Tab

1. **Source the Design Entity**
   During this stage of Reference, the files that describe the entirety of the Reference Design will be loaded by Formality.
   (a) In the "1. Read Design Files" tab, in the Verilog tab, in the Preferences frame, select WORK as the Design Library and select the correct Verilog Version, then click Verilog...
   (b) Locate the `Johnson_count.v` File within the Source Directory
   (c) Click Open
   (d) Click the right arrow button to load files

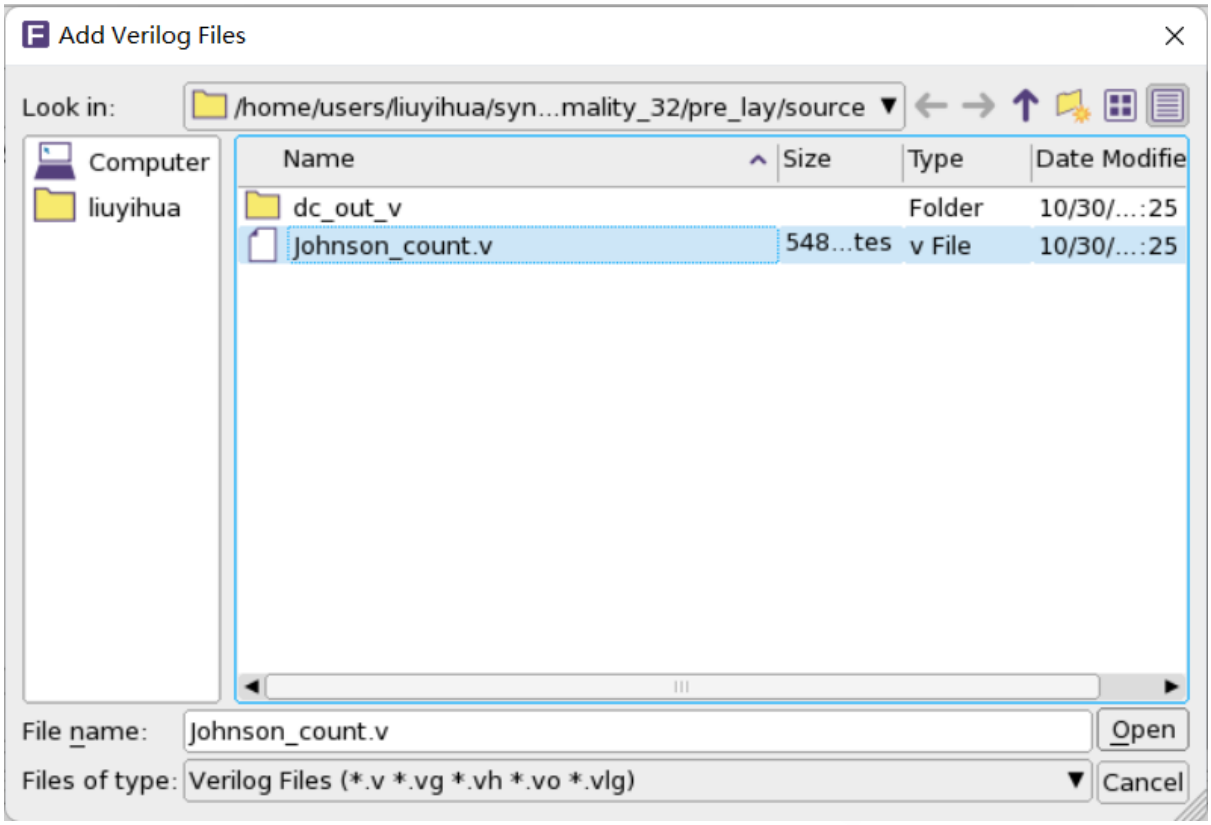


Figure 30: Set up RTL Source.

```
read_verilog -container r -libname WORK -05 { /home/users/<username
↪ >/synopsys/syn_tut/formality_32/pre_lay/source/Johnson_count.v
↪ }
```

2. **Source the DB file**
   During this stage of Reference, an optional technology library may be loaded for Formality, and the design, to use.

   (a) Select the "2. Read DB Libraries" Tab within "1. Ref."

   (b) In the Preferences frame, select WORK as the Design Library, check "Determined by DB" and "Read as a shared library". Click DB...

   (c) Locate `database_link.db` within the `ref/` directory. This is the link to the database that we created earlier.

   (d) Click Open

   (e) Click the right arrow button to load files

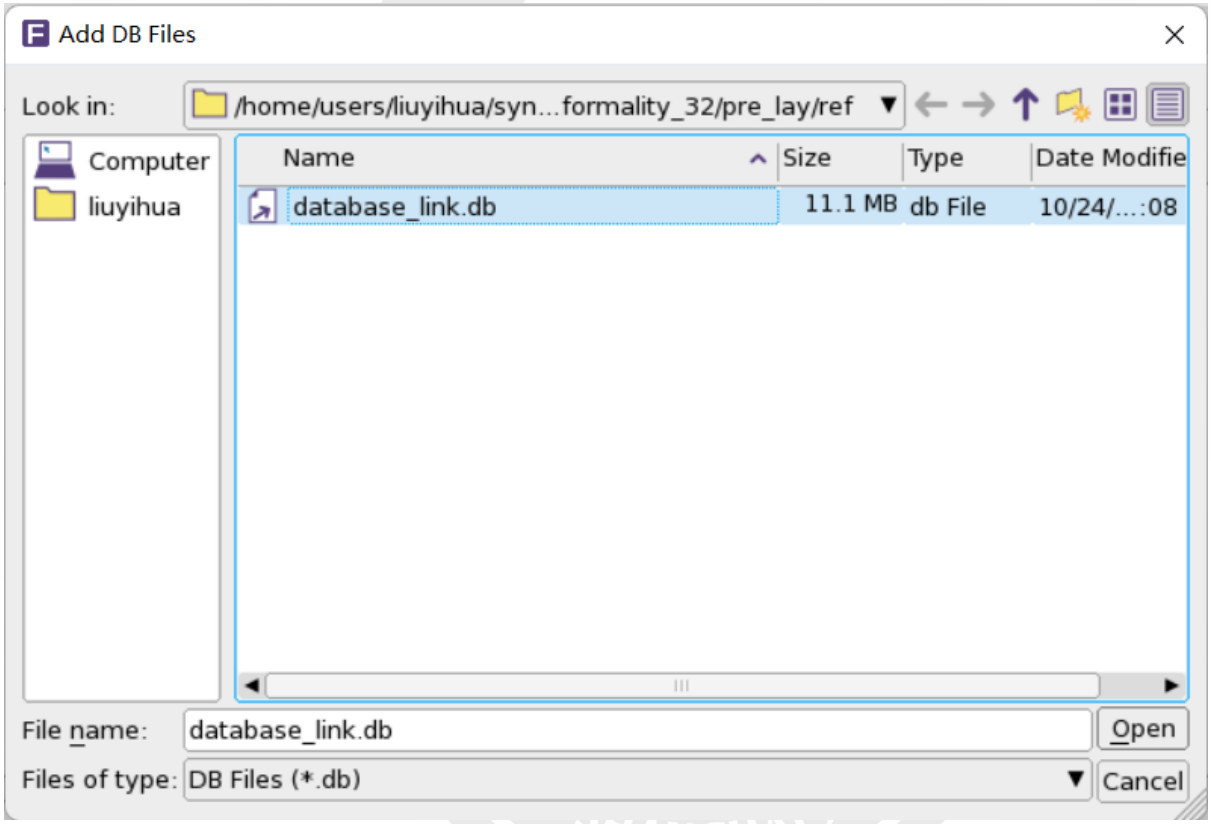


Figure 31: Setting the DB File.

```
read_db { /home/users/<username>/synopsys/syn_tut/formality_32/pre_
↪ lay/ref/database_link.db
↪ }
```

3. **Set Top Design of reference**
   During this stage of Reference, the Top Level Design of the Reference Entity loaded

earlier will be set.

(a) Select the "3. Set Top Design" Tab within "1. Ref."

(b) Select the Library WORK under "1. Choose a Library"

(c) Select `Johnson_count` under "2. Choose a Design"

(d) Click "Set Top" under "3. Set and link the top design"

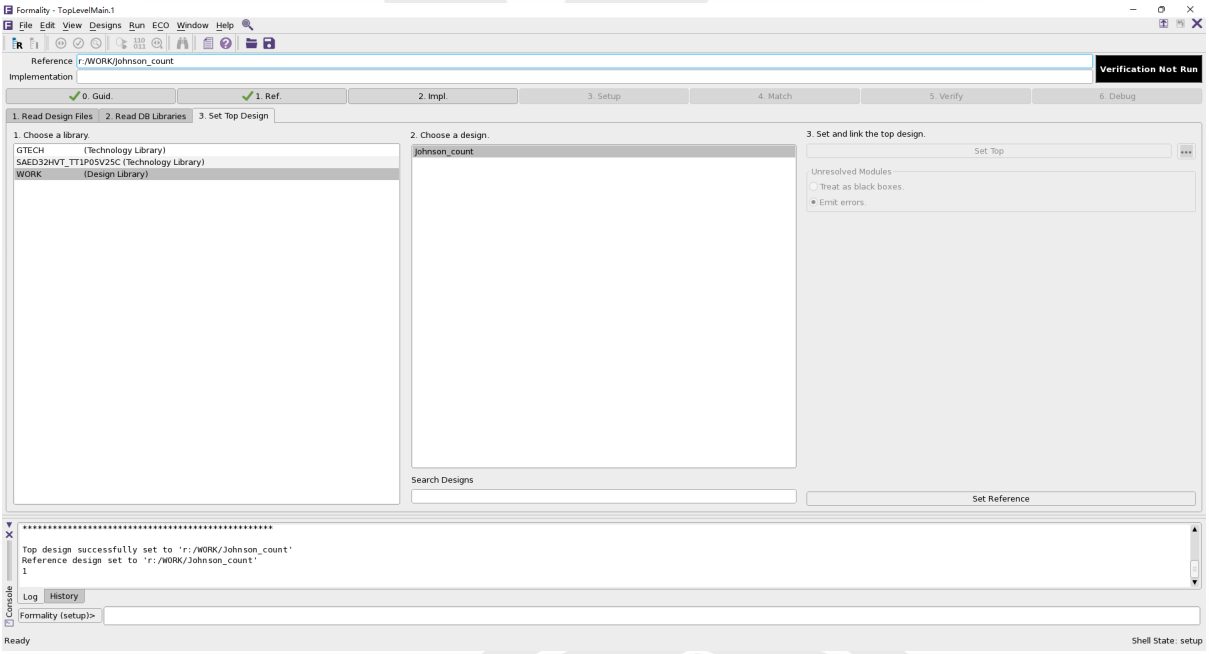


Figure 32: Setting the Top Level Design.

```
set_top r:/WORK/Johnson_count
```

3.2.3 Implementation (Specify the Implementation Design)

The procedure for specifying the implementation design is identical to that for specifying the reference design. The implementation design is the gate-level netlist after the design is compiled. Click the "2. Impl." Tab

1. **Source the Implementation Design**
   During this stage of Implementation, the files that describe the entirety of the Compiled Design will be loaded by Formality.

   (a) In the Verilog tab, in the Preferences frame, select WORK as the Design Library and select the correct Verilog Version, then click Verilog...

   (b) Locate the `Johnson_count_dc.v` file within `source/dc_out_v`

   (c) Click Open

   (d) Click the right arrow button to load files

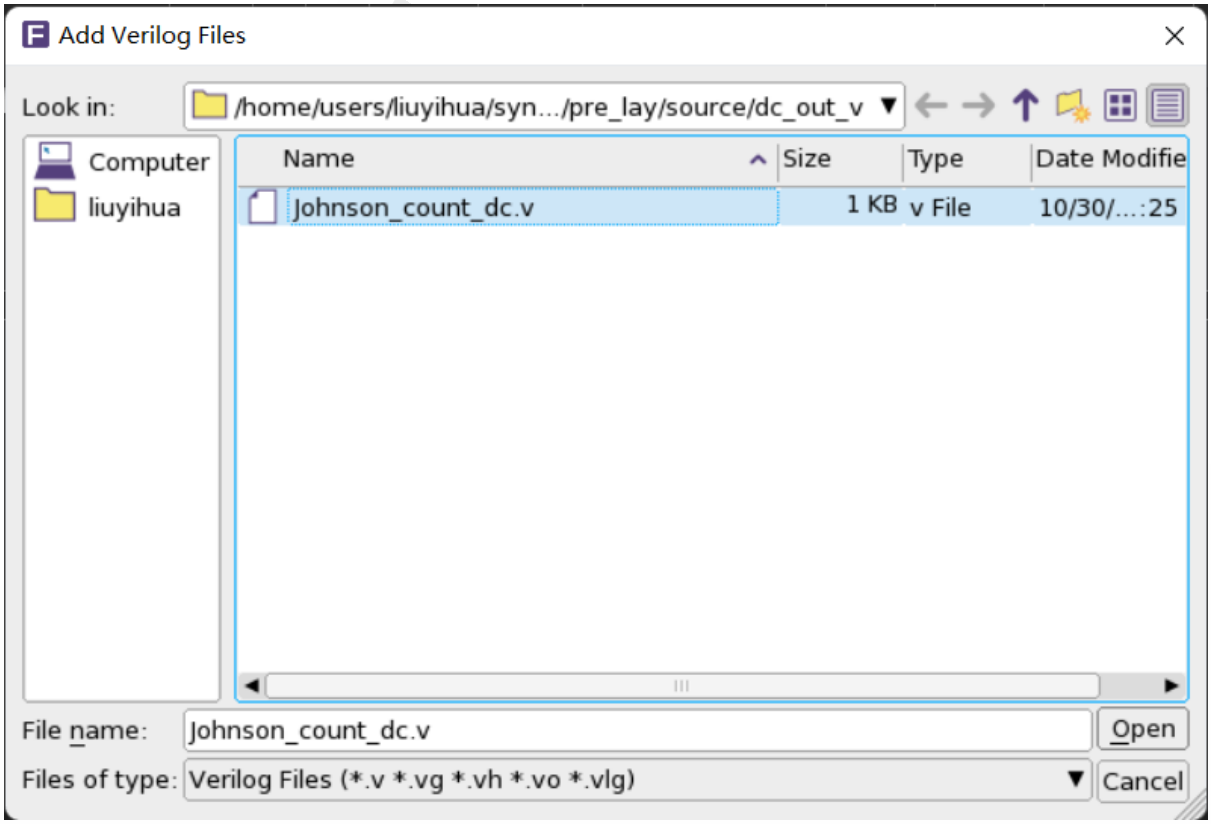


Figure 33: Set Gate Level Verilog.

```
read_verilog -container i -libname WORK -05 {
↪  /home/users/<username>/synopsys/syn_tut/formality_32/pre_lay/so
↪  urce/dc_out_v/Johnson_count_dc.v
↪  }
```

2. **Source the DB file**
   During this stage of Implementation, we will be loading the technology library for use by design, the same as the "1. Ref." Stage.

3. **Set Top Design of reference**
   During this stage of Implementation, the Top Level Design of the Compiled Entity is loaded, the same as the "1. Ref." Stage.

   ```
   set_top i:/WORK/Johnson_count
   ```

3.2.4 Setup (Setup the Design)

After Importing the Reference and Implementation Design, Formality is then Set up for the Comparison between the two. Click the "3. Setup" tab.

Now we add the reference and implementation constants.

1. Click Set...
2. Select Input Ports under the Reference Tab
   You should now see four ports appear underneath the Instance of the Design

Note that the new version changed the "Hide Objects" frame to "Show Objects" frame.

3. Select the SE port
4. Click Apply
5. Repeat process for the r port as well as SE and r under the implementation Tab

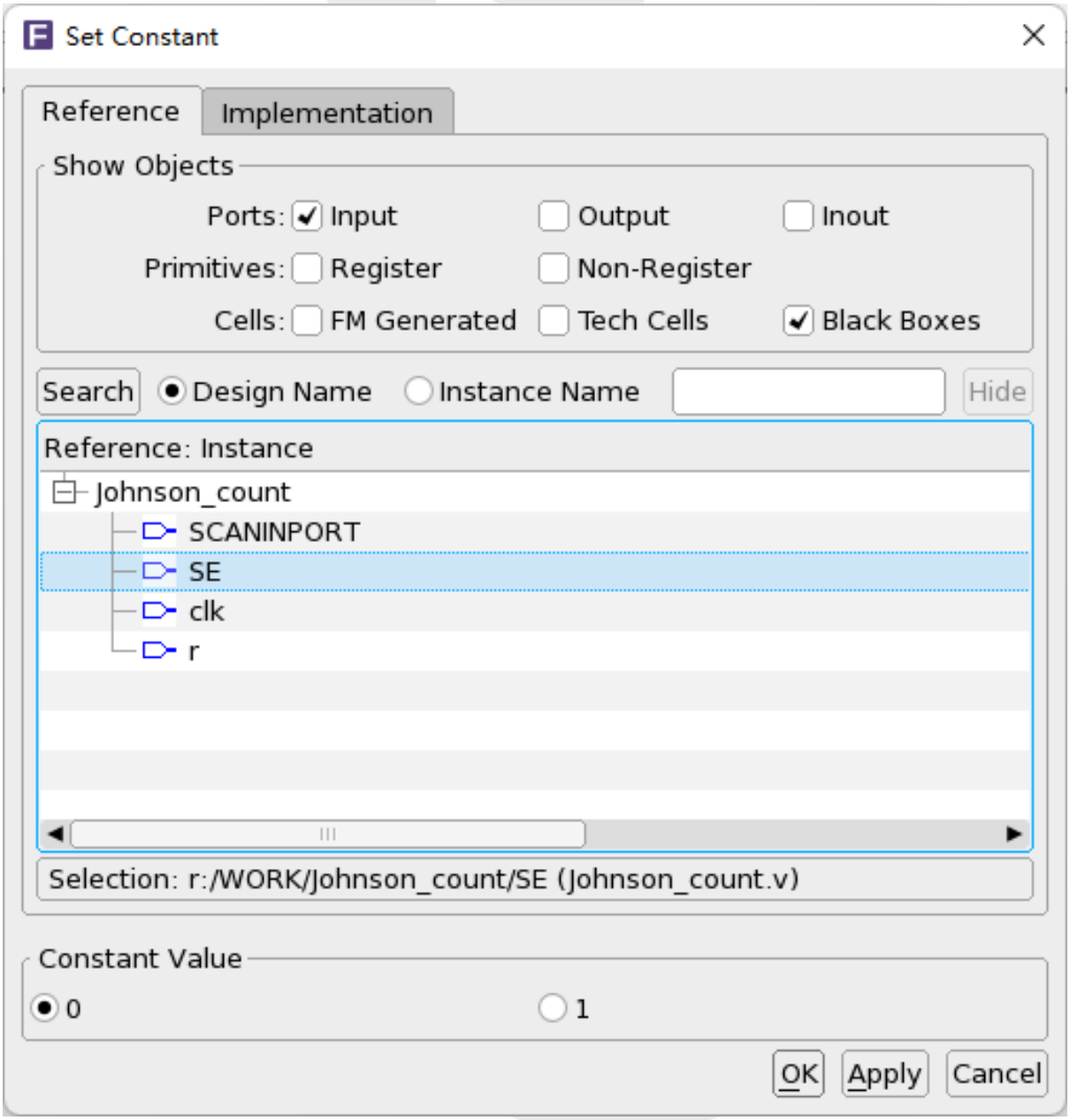


Figure 34: Setup the Design Window.

```
set_constant -type port r:/WORK/Johnson_count/SE 0
set_constant -type port r:/WORK/Johnson_count/r 0
set_constant -type port i:/WORK/Johnson_count/SE 0
```

```
set_constant -type port i:/WORK/Johnson_count/r 0
```

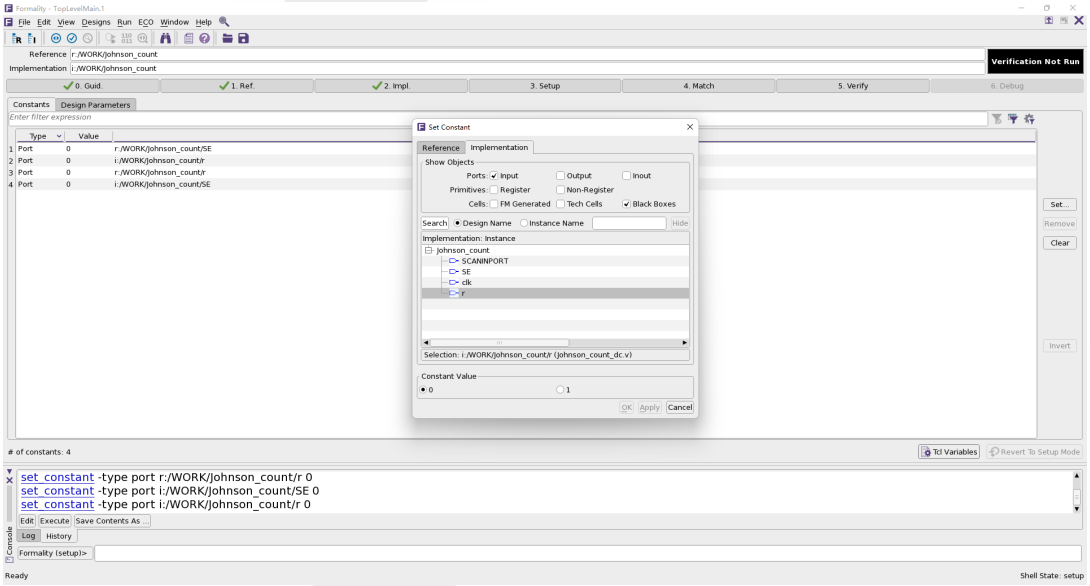


Figure 35: After setting up the Design.

3.2.5 Match (Match Compare Points)

"Match compare points" is the process by which Formality segments the Reference and Implementation Designs into logical units called logic cones. To match compare points between `Johnson_count.v` and `Johnson_count_dc.v` (after DC), do the following:

1. Click upon the "4. Match" Tab to start the Matching Process
2. Click Run Matching

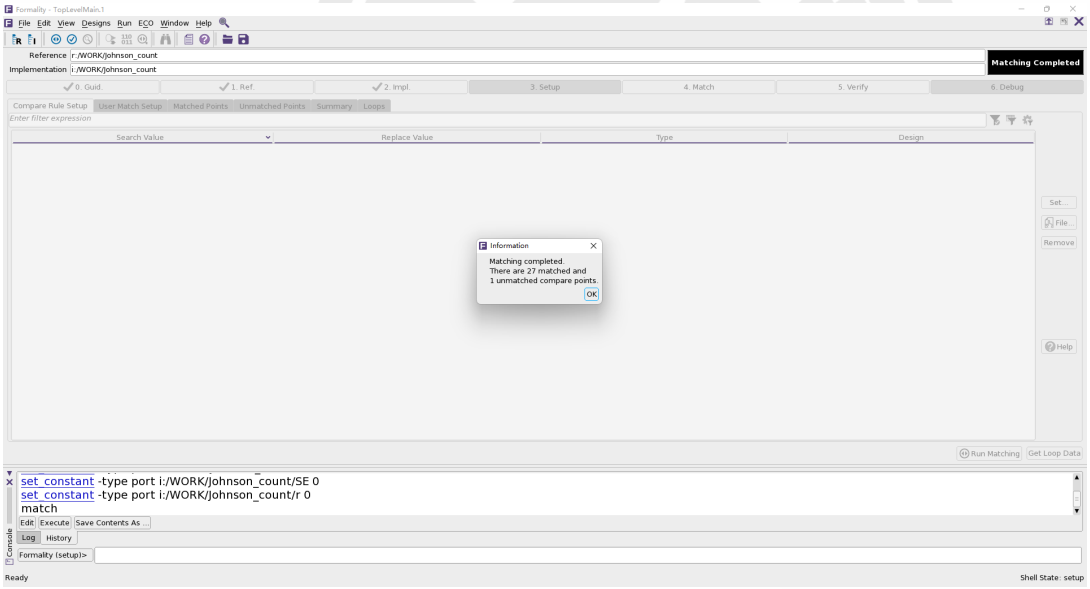


Figure 36: Match compare points.

```
match
```

As seen, Formality matches the logic cones between each Design for later Comparison and Verification. As can be seen, we have one unmatched point between our two implementations. This indicates the designs do not use the same input and output ports or registers, some of the ports are undriven or have similar issues.

1. To see the unmatched points, click the Unmatched Points tab.
   Here we can see our unmatched point.

2. Right Click on the only Reference Object

3. Click View Reference Object
   Here we can see our unmatched point seems to be an undriven port. Zooming out, we can see the entire Reference Design.

4. Closing the Design, Right Click on the Object, and Click View Reference Source
   Here we can see the reference object does not utilize the SCANOUTPORT Output.

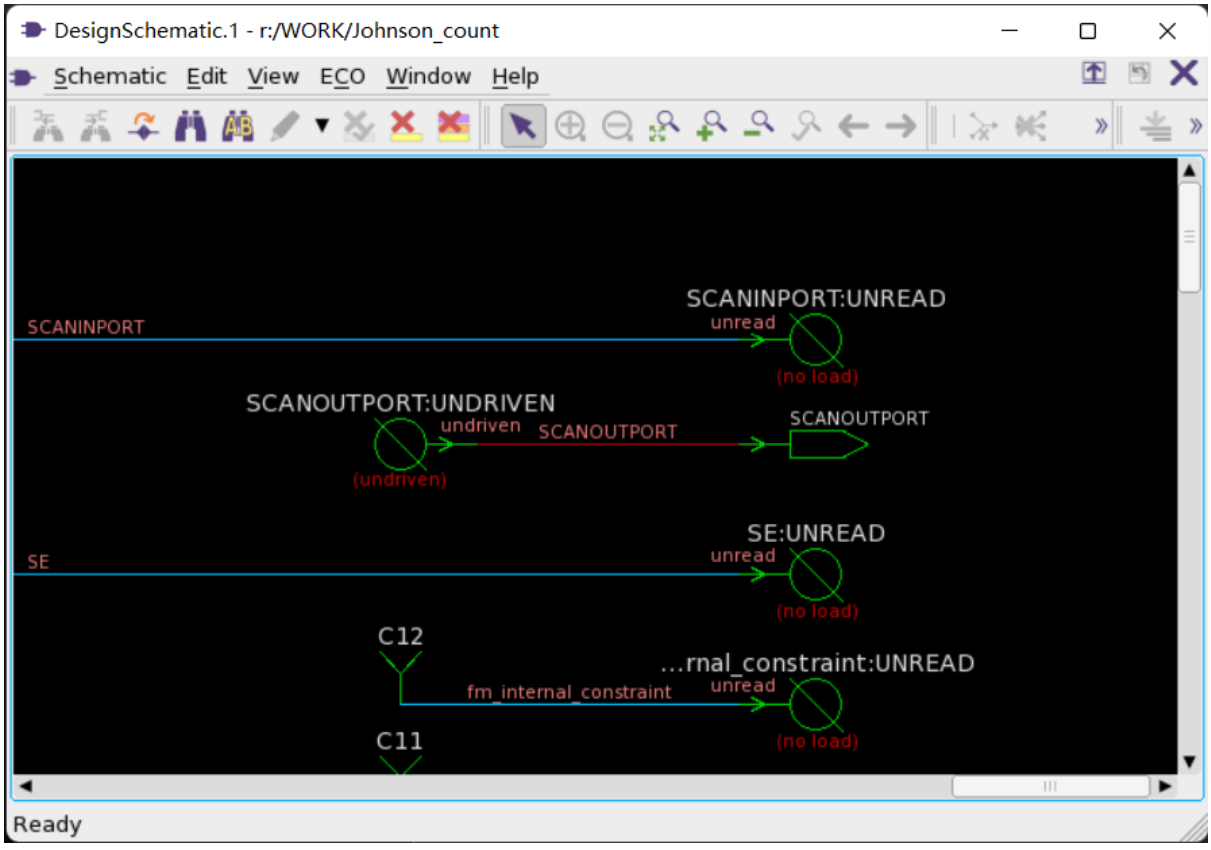
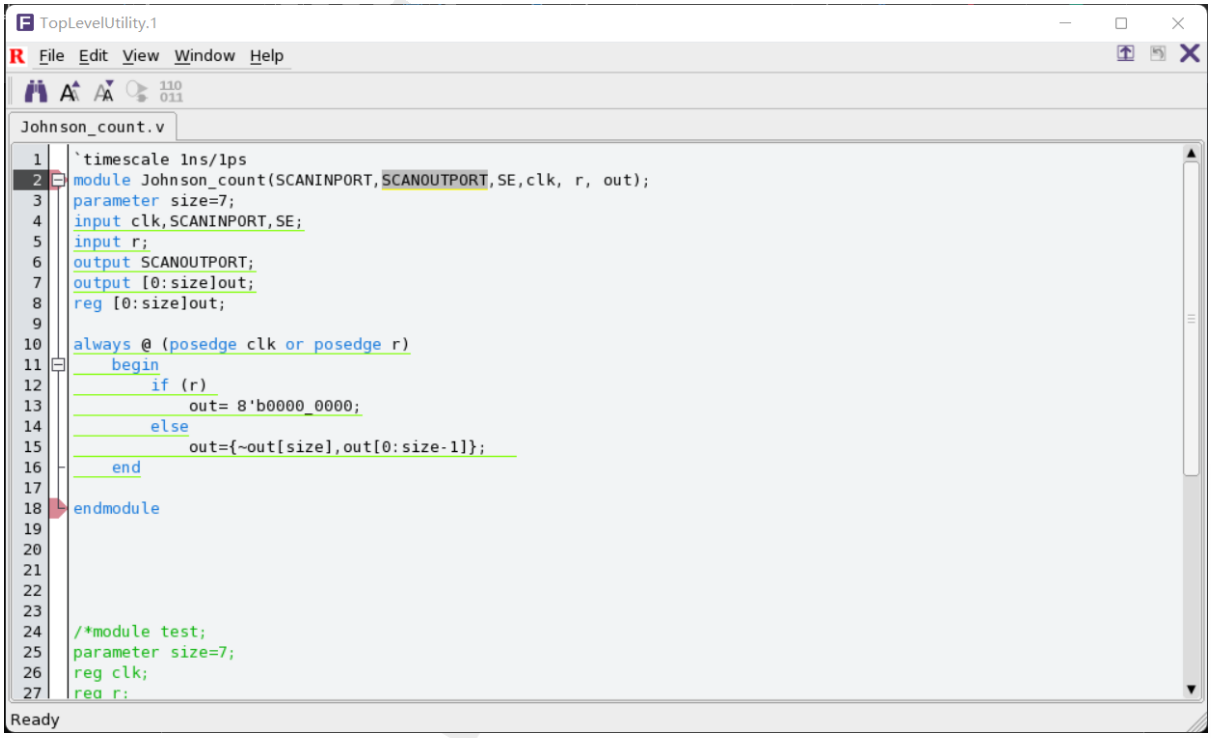


Figure 37: Unmatched Point.



From here, we may continue to verify and debug, which will fail due to our undriven signal. If continued, one may look upon both the Reference and Implementation Design Objects and Sources to look at the differences between the two designs. As seen in Matching, we have an undriven signal, so driving the signal will remedy the issue.

1. Open a New Terminal
2. Navigate to `Johnson_count.v` in `pre_lay/source`
3. Run `gedit Johnson_count.v`
4. Add an instance `NBUFFX2_HVT U12` in the Empty Line Before `endmodule`
5. Save the file

After this, you must go through Reference, Setup, and Match again to Reload the Modified File and Set it up for Verification. After Doing so, when Running Match, no Unmatched Points will appear.

3.2.6 Verify (Verify the Designs)

When using the verify command, Formality attempts to prove design equivalence between an implementation design and a reference design. This section describes how to verify a design or a single compare point, as well as how to perform traditional hierarchical verification.

Click on the "5. Verify" Tab, Then Click on Verify.

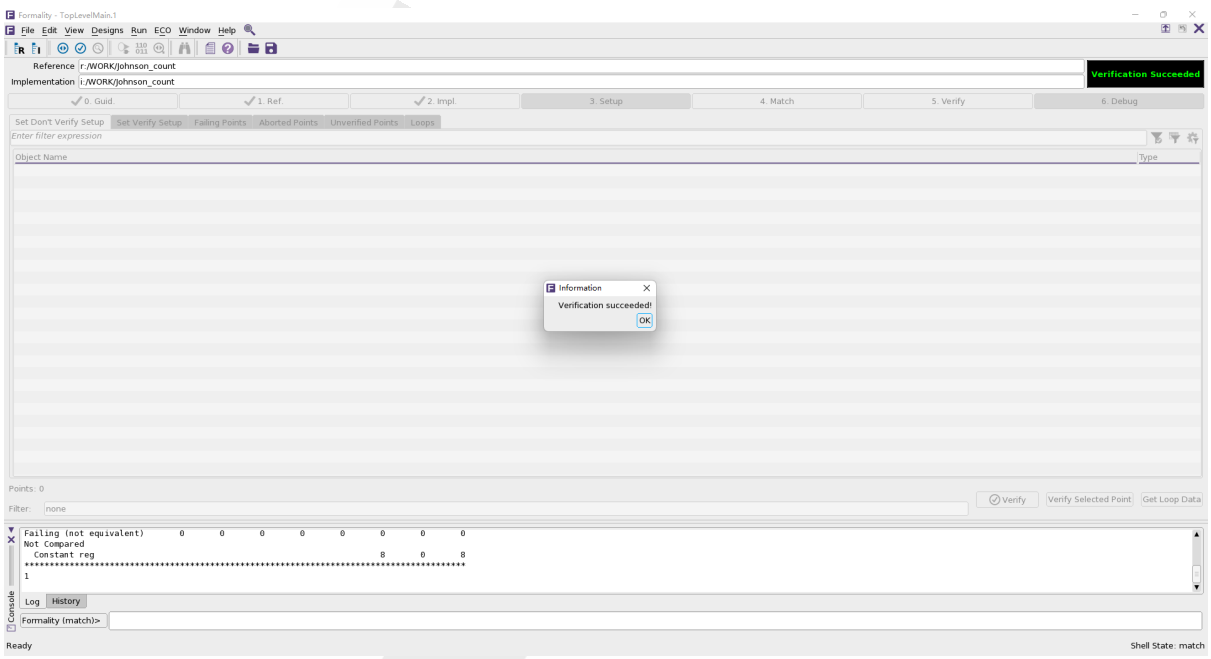


Figure 38: Verify the design.

verify

The verifying process was finished successfully.

Should Verification Fail, ensure that you have updated the reference design, so no more undriven ports exist and the new design has been loaded.

Ensure Setup and Match were Run as well after doing this.

Should "'set_constant' can only be executed in SETUP mode" appear or similar issues, type setup in the formality terminal or command line and attempt the step again.

```
Formality (match)> setup or Formality (verify)> setup or etc.
```

3.2.7 Debug

After Verify is run, if any issues are found, one must find the exact points in the designs that exhibit the difference in functionality and then fix them during this Debugging Stage.

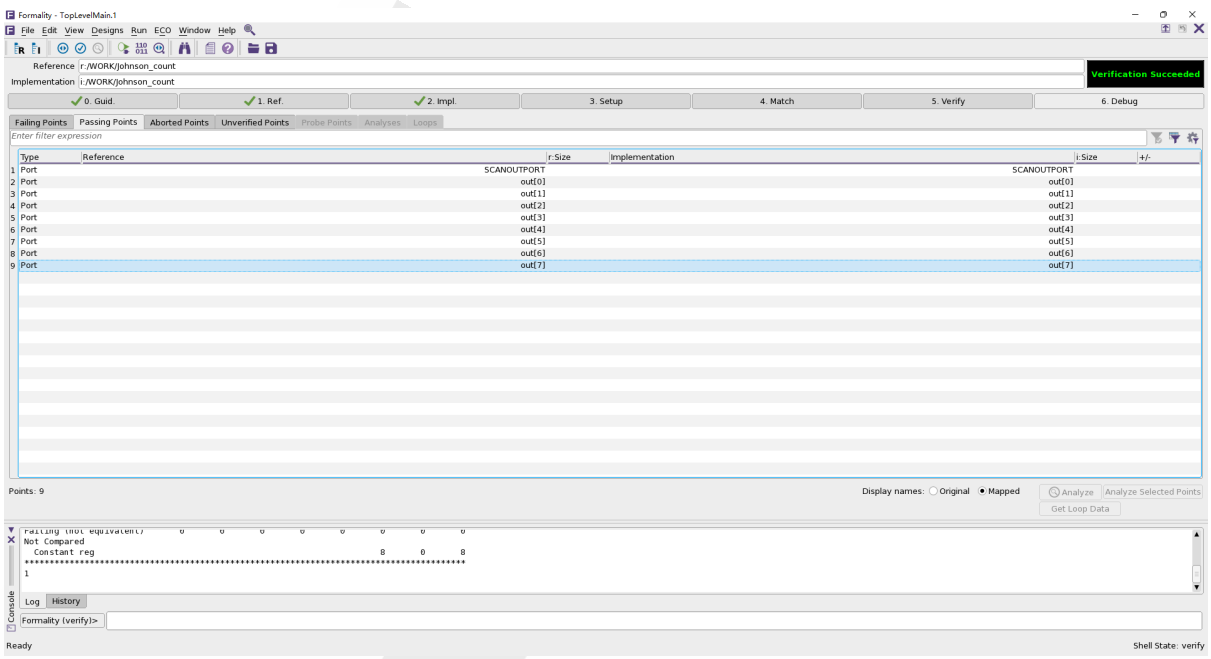
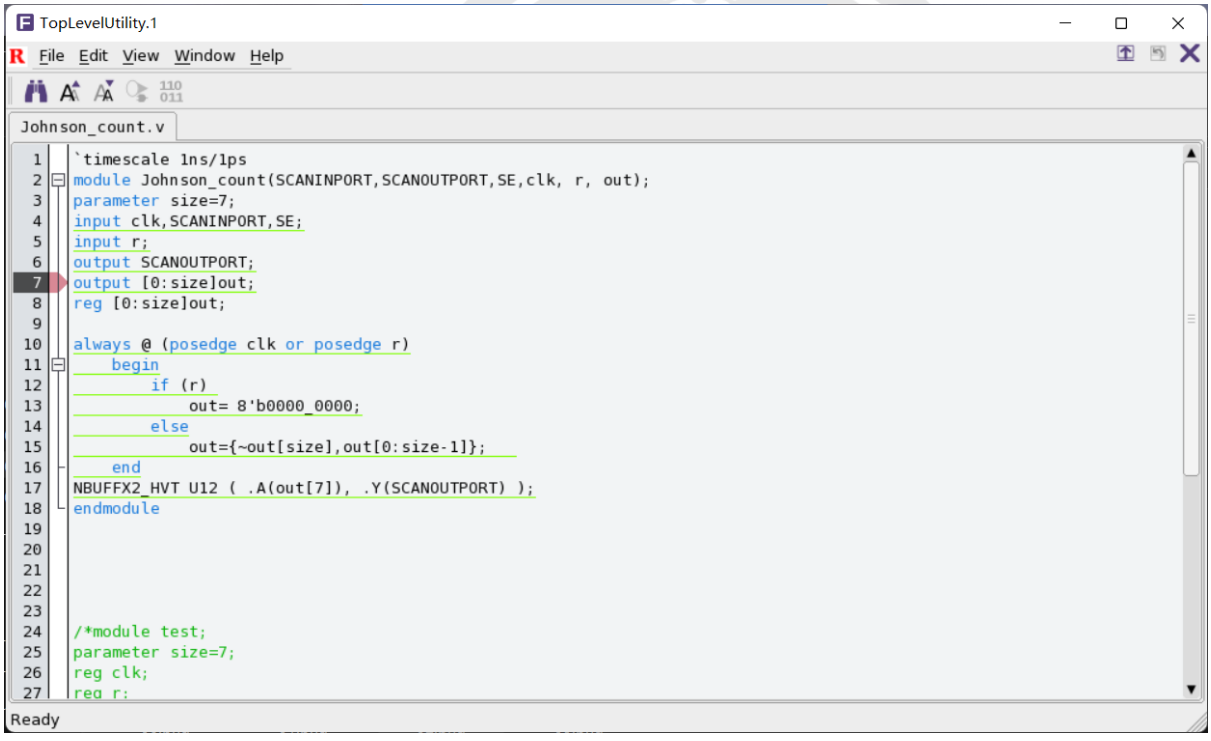
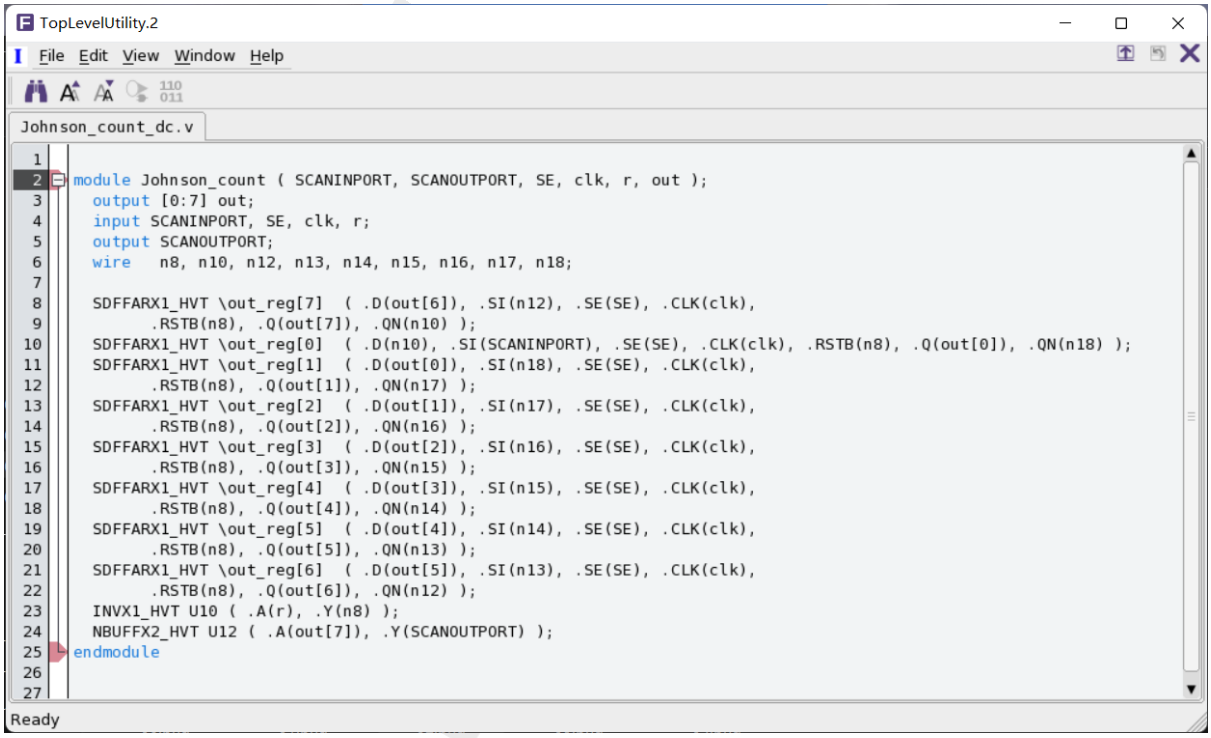


Figure 39: Debugging the Design.

Formality is able to simultaneously display Reference and Implementation Source Views and mark differences and/or similarities. This is done by right-clicking on the object and Clicking View Reference/Implementation Source.



(a) Pre-lay



(b) Post-lay

Figure 40: Implementation and Reference Verilog

Formality can also display a schematic view and highlight reference objects on it. This is done in a similar manner to the sources except clicking View Reference/Implementation Source.

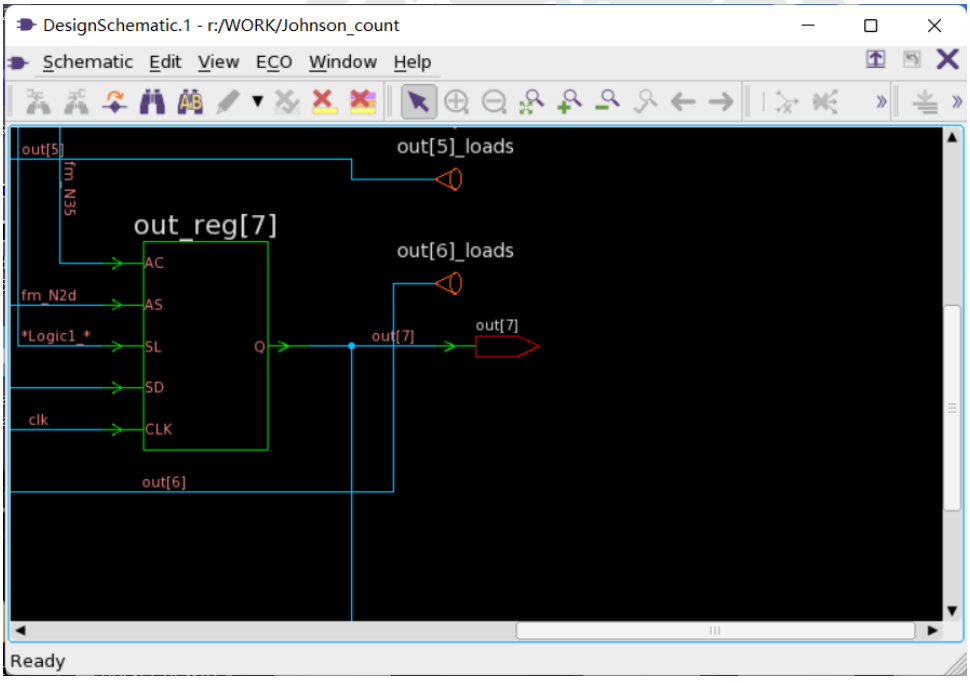


Figure 41: View Reference Object in the schematic.

In this lab work, design output results are stored in the `../results/` directory.

To exit Formality write exit in the command line.

```
Formality (verify)> exit
```

4 Static Timing Analysis with PrimeTime

4.1 Introduction

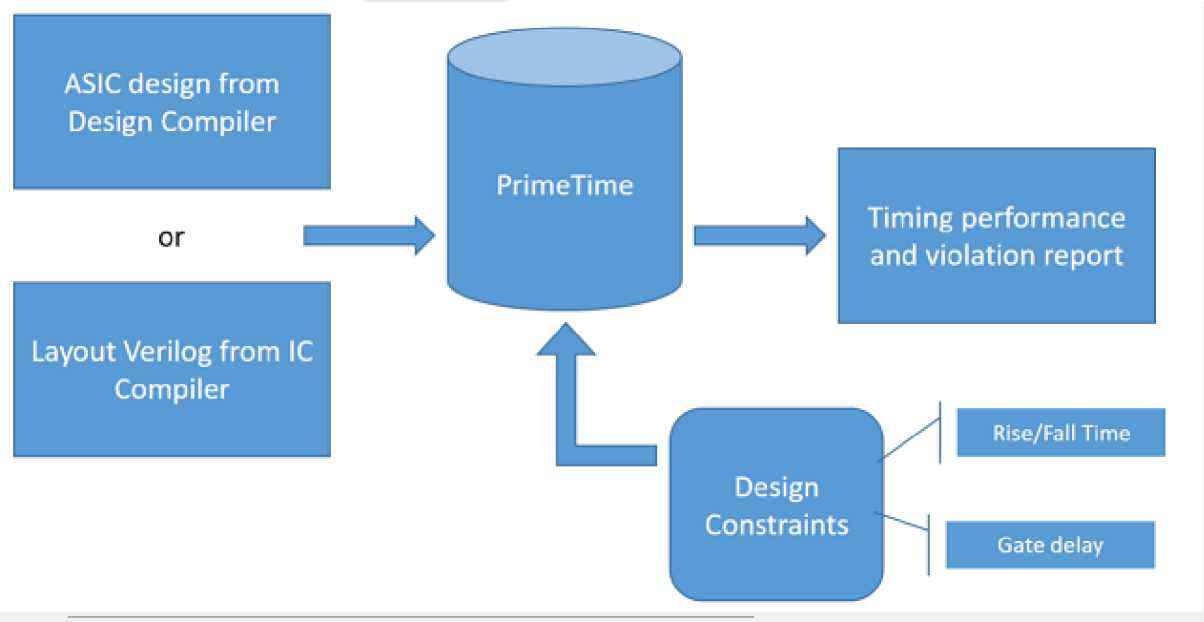
PrimeTime is a full-chip, gate-level Static Timing Analysis (STA) tool that is an essential part of the design and analysis flow for today's large-chip designs.

Why is timing analysis important when designing a chip?

Timing is important because just designing the chip is not enough; we need to know how fast the chip is going to run, how fast the chip is going to interact with the other chips, how fast the input reaches the output etc.

Timing Analysis is a method of verifying the timing performance of a design by checking for all possible timing violations in all possible paths.

4.2 PrimeTime Basic Flow



PrimeTime can be used to perform STA using either the results of the Design Compiler (DC) or Integrated Circuit Compiler (ICC). If the DC results are used, we are doing pre-layout STA. If the ICC results are used, we are doing post-layout STA. PrimeTime accepts as input a Verilog file from the DC or ICC and then a design constraints file specifying the timing constraints of the design. With these inputs, PrimeTime produces timing performance and violation reports.

If you are using results from the Design Compiler, then work in the `pre_lay` directory. If you are using results from the IC Compiler, then work in the `post_lay` directory. This lab will go through pre-layout STA first and then post-layout STA. *In this lab, we will use Synopsys PrimeTime (R) Version R-2020.09-SP1 for linux64.*

4.3 Pre-Layout STA

1. Unzip the `/home/Flow/Synopsys/Scripts/pt_32.zip` file to the `syn_tut` directory. Now, set up the Synopsys tools and start the PrimeTime graphical user interface (GUI) by entering the following:

```
cd ~/synopsys/syn_tut
unzip /home/Flow/Synopsys/Scripts/pt_32.zip -d .
cd pt_32
cd pre_lay
cd work
pt_shell -gui
```

This opens the PrimeTime top-level GUI window (Figure 42).

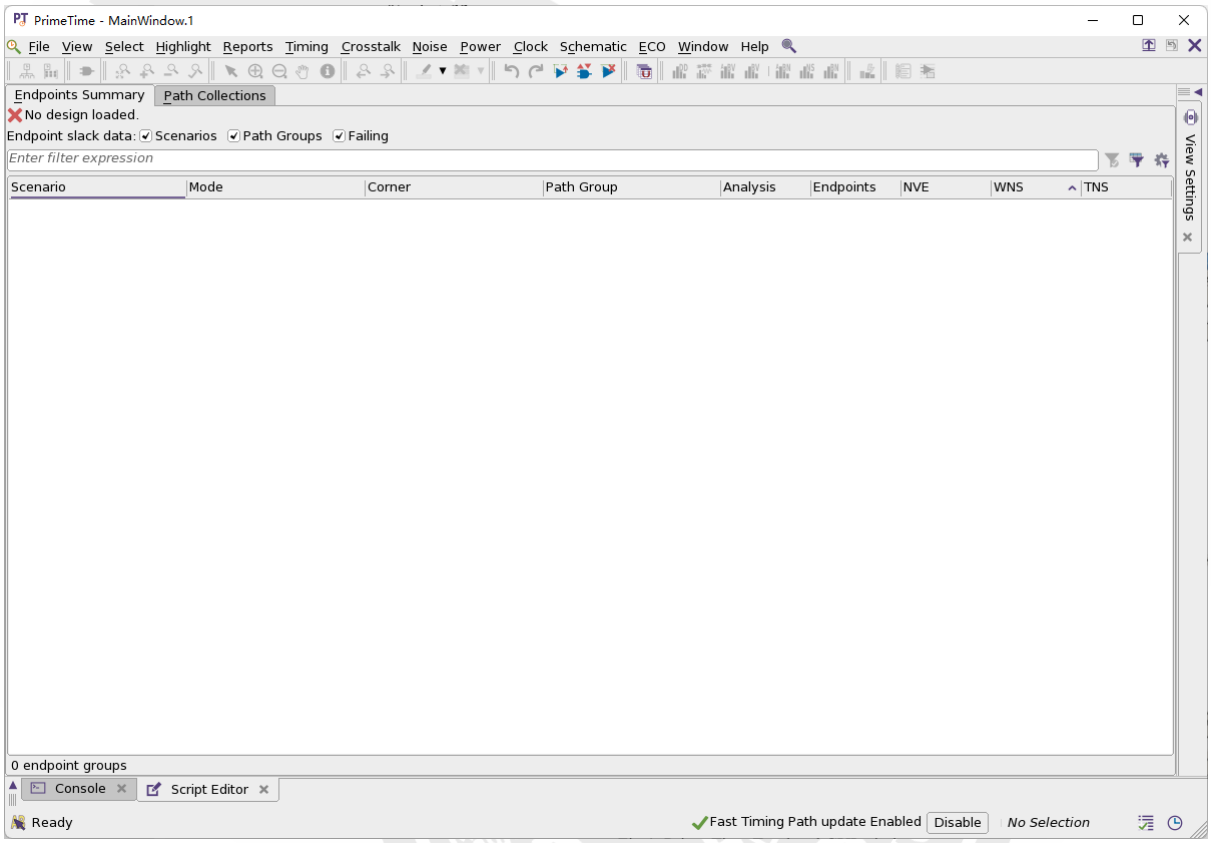


Figure 42: PrimeTime Top-level GUI window.

If it shows an error:

```
Error: Library Compiler executable path is not set. (PT-063)
```

and quitted with error:

```
Suppressed Messages Summary:
Id          Severity      Occurrences   Suppressed
------------------------------------------------------------------------------
```

```
CMD-005     Error                   12          12
Total 1 type of message is suppressed
```

Please read Appendix: Troubleshooting B.2.

2. The paths to the libraries are set up by running the setup.tcl file in the scripts directory. Look at this file using your text editor of choice to see how the libraries are being set up *and make modifications if needed.* To run the script, enter the following command into PrimeTime:

   ```
   pt_shell> source ../scripts/setup.tcl
   ```

3. Next, notice that there is one other script in the scripts directory named `sta_pre_lay.scr`. This script checks both setup and hold timing and generates the appropriate reports. You can run these scripts in the same way that you ran the setup.tcl script to create the timing reports. However, if this is your first experience with PrimeTime, we recommend running the commands individually to get familiar with them. This tutorial leads you through running the individual commands. The overall flow of the scripts is shown below (Figure 43).

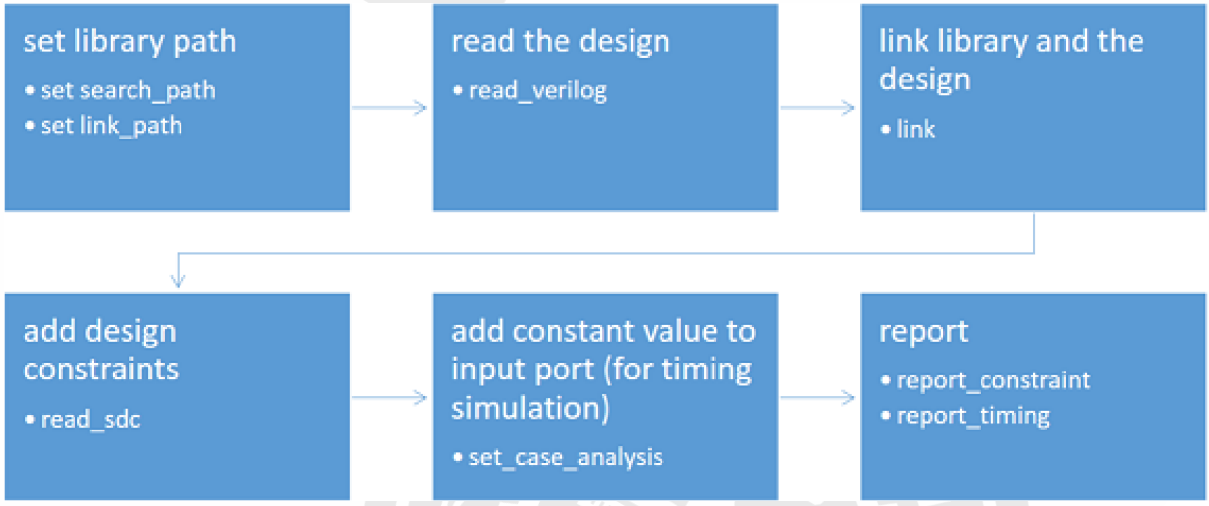


Figure 43: Workflow used in Scripts.

   (a) First, we will read the design netlist. PrimeTime accepts design gate-level netlists in both Verilog and VHDL formats. You can read the design netlist with the command `read_verilog` or `read_vhdl`. Our file is in Verilog, so we will run the command below.

   ```
   pt_shell> read_verilog ../source/Johnson_count_dc.v
   ```

   (b) For a design to be complete, it needs to be connected to all of the library components and designs it references. So to perform a name-based resolution of design references for the current design, we will use the link command. The references must be located and linked to the current design in order for the design to be functional. The purpose of this command is to locate all of the designs and library components referenced in the current design and connect (link) them to the current design.

`link`

(c) Next, we need to read the design constraints. This is done using the `read_sdc` command.

```
pt_shell> read_sdc ../source/constr.sdc
```

(d) Now we will apply a constant value to input ports r and SE with the `set_case_analysis` command. It specifies that a port or pin is at a constant logic value (1 or 0) or is at a transition (rising or falling). The ports in this lab are non-active with the value 0, so the ports r and SE are given a value of 0.

```
set_case_analysis 0 [get_port r]
set_case_analysis 0 [get_port SE]
```

(e) Now, we have supplied PrimeTime with all of the inputs that it needs to perform STA. We can now find whether or not our design meets the timing constraints with the `report_constraint` command. Several flags can be passed to `report_constraint` depending on what kind of timing violations we want to check. For example, `report_constraint -max_delay` will report setup timing information, and `report_constraint -min_delay` will report hold timing information. We want to see all timing violations, so we will run `report_constraint -all_violators`. To produce all timing violations and write the results to a file, run the command below.

```
pt_shell> report_constraint -all_violators -significant_digits 4
↪  > ../results/J_pre_constr.rpt
```

(f) The `report_timing` command is the most flexible and powerful PrimeTime analysis command. The `-delay_type` option specifies the type of timing checks to report. Set the delay type to `max` for setup checks and to min for hold checks. To generate timing reports for setup and hold checks and write them to a file, run the commands below.

```
report_timing -delay_type max -nworst 40 -significant_digits 4 >
↪  ../results/J_pre_max_timing.rpt
report_timing -delay_type min -nworst 40 -significant_digits 4 >
↪  ../results/J_pre_min_timing.rpt
```

In reports, it is possible to have violations, for example, hold or setup violations. To correct the hold violation, add buffers to the respective port. And To correct the setup violation, increase the area of cells. Figure 44 shows an excerpt from a report with no timing violations

```
14
15  Startpoint: out_reg[7] (rising edge-triggered flip-flop clocked by clk)
16  Endpoint: out_reg[0] (rising edge-triggered flip-flop clocked by clk)
17  Last common pin: clk
18  Path Group: clk
19  Path Type: min
20
21  Point                                              Incr       Path
22  ---------------------------------------------------------------------------
23  clock clk (rise edge)                           0.0000     0.0000
24  clock network delay (propagated)                0.0234     0.0234
25  out_reg[7]/CLK (SDFFARX1_HVT)                   0.0000     0.0234 r
26  out_reg[7]/QN (SDFFARX1_HVT)                    0.0000     0.0234 r
27  out_reg[0]/D (SDFFARX1_HVT)                     0.0124     0.0358 r
28  data arrival time                                          0.0358
29
30  clock clk (rise edge)                           0.0000     0.0000
31  clock network delay (propagated)                0.0237     0.0237
32  clock reconvergence pessimism                   0.0000     0.0237
33  clock uncertainty                               0.4000     0.4237
34  out_reg[0]/CLK (SDFFARX1_HVT)                              0.4237 r
35  library hold time                              -0.0275     0.3962
36  data required time                                         0.3962
37  ---------------------------------------------------------------------------
38  data required time                                         0.3962
39  data arrival time                                         -0.0358
40  ---------------------------------------------------------------------------
41  slack (VIOLATED)                                          -0.3604
42
43
44  Startpoint: out_reg[6] (rising edge-triggered flip-flop clocked by clk)
45  Endpoint: out_reg[7] (rising edge-triggered flip-flop clocked by clk)
46  Last common pin: clk
47  Path Group: clk
48  Path Type: min
49
```

Figure 44: Excerpt from Timing Report.

You can view the Timing Path Table by going to the "Path Collections" tab and clicking the "Create..." button and entering "40" in "NWorst paths per endpoint", and clicking "OK", The report can also be seen in the Timing Analysis Driver Console by clicking on one of the rows and then clicking the "Inspect Worst Paths..." button. See Figure 45 and Figure 46.

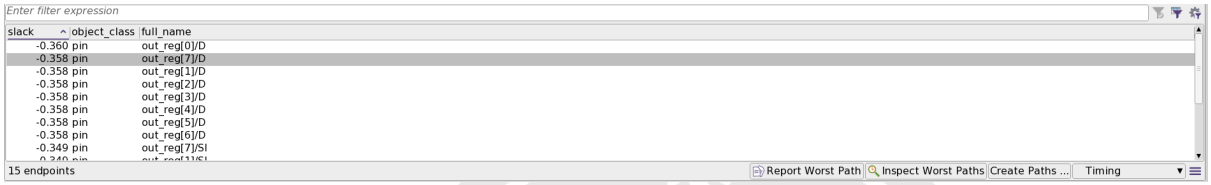


Figure 45: The Timing Path Table.

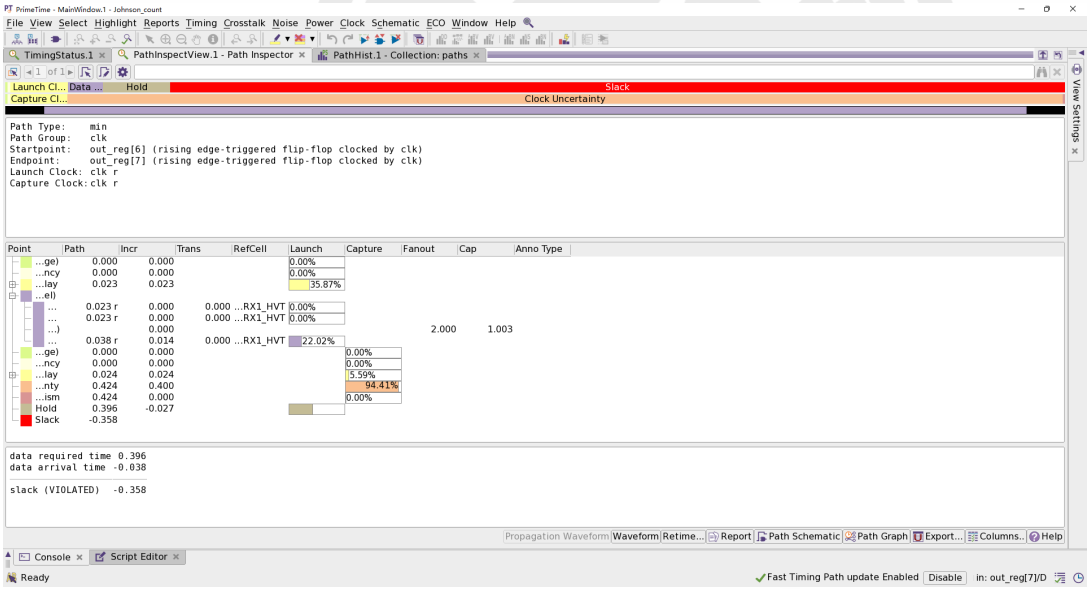


Figure 46: Path Inspector console.

You can view the schematic of the selected path by right-clicking and selecting "Path Schematic" either in the Table view or in the "Inspector" Window, see Figure 6. In addition to the schematic view, you can also view the waveform graph by right-clicking and selecting "Waveform" in the Inspector window. See Figure 7.

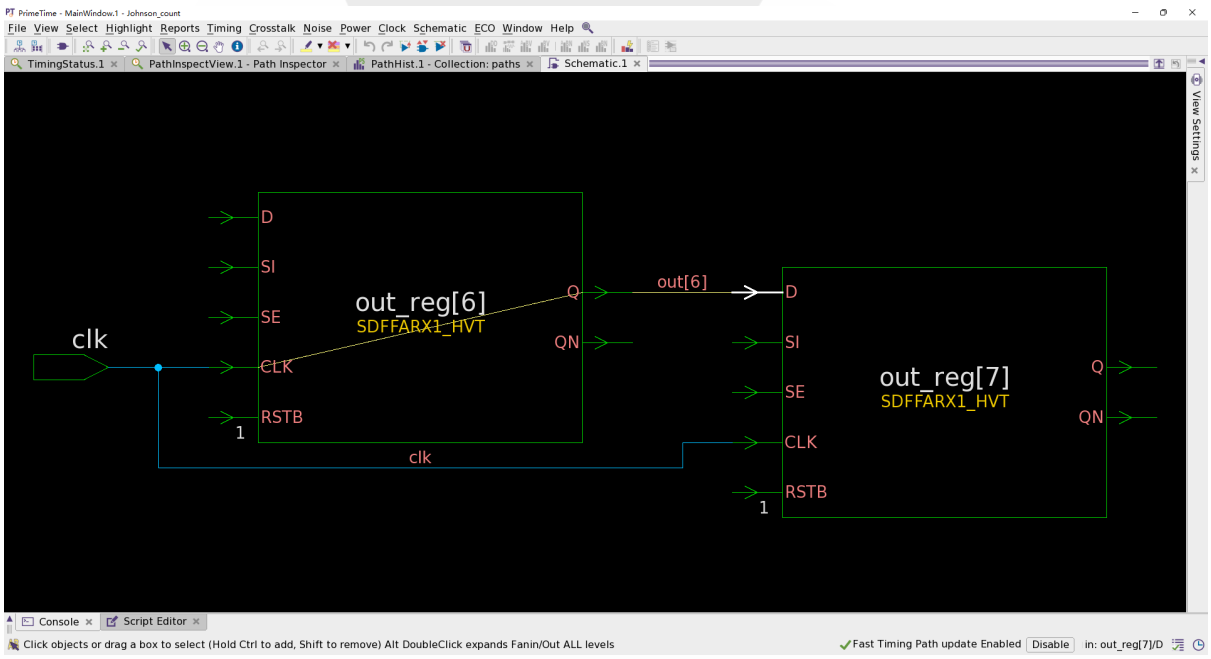


Figure 47: Path Schematic.

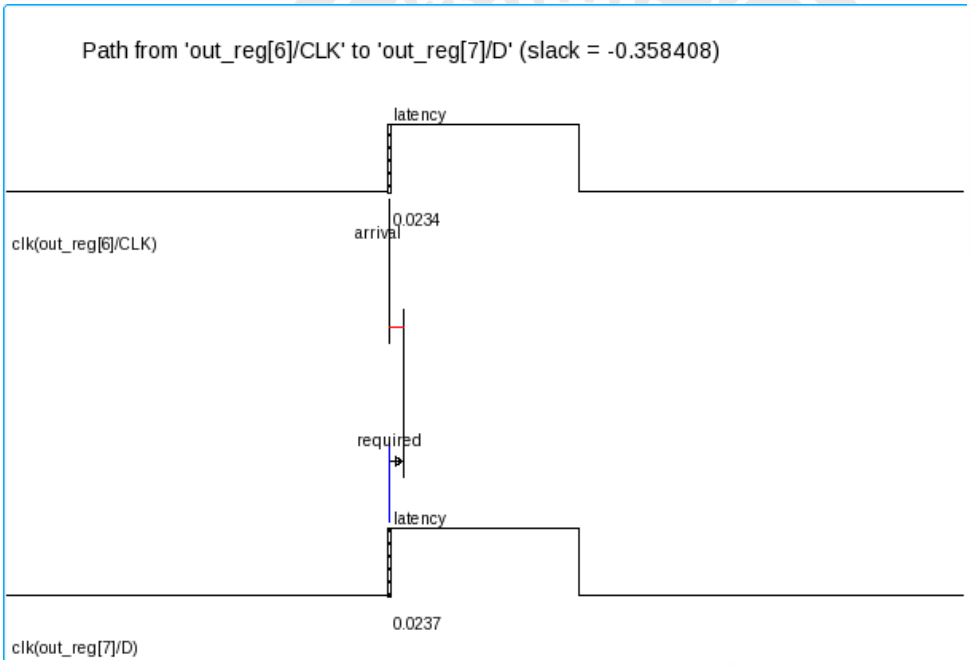


Figure 48: Path Waveform.

You are required to complete the previous steps from "out_reg[7] (rising edge-triggered flip-flop clocked by clk)" to "out[7] (output port)".

(g) PrimeTime can also output results in Standard Delay Format (.sdf). This includes delay information, such as pin-to-pin cell delays and net delays, and timing checks, such as setup, hold, recovery, and removal times. Run the following command to produce an SDF file:

```
pt_shell> write_sdf ../results/Johnson_count_pre.sdf
```

4. We exit PrimeTime by using the `exit` command:

```
pt_shell> exit
```

4.4 Post-Layout STA

1. First, we will navigate to the `post_lay` directory. This directory is very similar to the `pre_lay` directory. The most notable difference is that the Verilog file in the source directory is the output from the IC Compiler and not the Design Compiler. This means that we are using a post-layout gate-level netlist. Once inside the `post_lay` directory, we will go to the `work` directory and start the PrimeTime GUI as we did before.

```
cd ~/synopsys/syn_tut/pt_32/post_lay/
cd work
pt_shell -gui
```

2. To set up the libraries, we can use the setup.tcl script in the scripts directory. Since the libraries we are using do not change based on whether we are doing a pre-layout or post-layout STA, we can use the same setup.tcl file we used before. If you like, you can open the setup.tcl script with a text editor to verify that it is the same script we used for pre-layout STA.

```
pt_shell> source ../scripts/setup.tcl
```

3. Notice that the only other script in the scripts directory is `sta_post_lay.scr`. This script is identical to the script used for pre-layout STA except for the two things mentioned below. Open the script with a text editor to verify this.

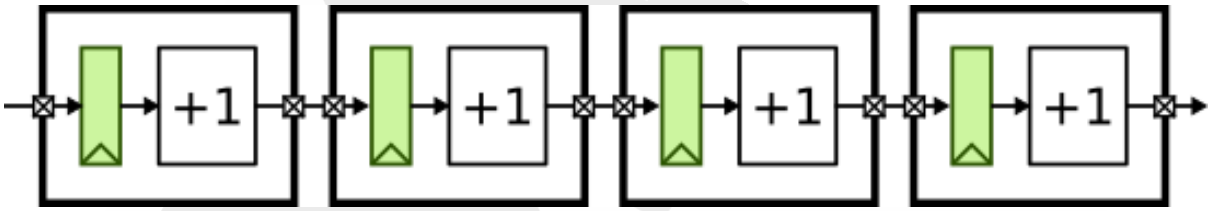
   - The name of the Verilog file is different since we are using the output from the IC Compiler instead of from the Design Compiler. So, the `read_verilog` command reads a different file.
   - Since we are doing post-layout STA, we use slightly different names for the reports that we generate to indicate this.

4. If you like, you may run the `sta_post_lay.scr` script line by line like we did for the pre-layout STA. The steps will be identical other than the two differences mentioned above. If you followed the pre-layout STA section above, however, you will not learn much from doing this, and you may just run the script using the command below.

```
pt_shell> source ../scripts/sta_post_lay.scr
```

5 (Optional) PyMTL3

Our goal in this section is to generate a gate-level netlist for the following four-stage registered incrementer:



We will take an incremental design approach. We will start by implementing and testing a single registered incrementer, and then we will write a generic multi-stage registered incrementer.

5.1 Implement, Test, and Translate a Registered Incrementer

In `RegincrRTL.py`, set the `rtl_language` variable to `verilog` use Verilog for your RTL design and set it to `pymtl` to use PyMTL for your RTL design. Also, set `TOPDIR` to your workspace.

Now let us run all of the tests for the registered incrementer:

```
mkdir -p $TOPDIR/sim/build
cd $TOPDIR/sim/build
pytest ../regincr
```

The tests will fail because we need to finish the implementation. Let's start by focusing on the basic registered incrementer module.

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrRTL_test.py
```

Your task is to finish the implementation both in Verilog and in PyMTL3.

Once you have finished the implementation, let us rerun the tests:

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrRTL_test.py -sv
```

The `-v` command line option tells `pytest` to be more verbose in its output, and the `-s` command line option tells `pytest` to print out the line tracing. Make sure you understand the line tracing output. You can also dump VCD files for waveform debugging with `gtkwave`:

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrRTL_test.py -sv --dump-vcd
gtkwave regincr.RegIncrRTL_test__test_small.vcd
gtkwave regincr.RegIncrRTL_test__test_small_top.verilator1.vcd
```

GTKWave is the most popular open-source waveform viewer. However, I would suggest you to try the new terminal-based waveform viewer sootty. To run `sootty`, you need a terminal that fully supports the display of SVG. Currently, there are only two choices: `iTerm2` for Mac and `kitty` for Mac/Linux (WSL is also OK).

PyMTL supports automatically translating PyMTL RTL into Verilog RTL so we can then use that Verilog RTL with the ASIC flow. To test the translated Verilog RTL, you can use the `-test-verilog` command line option:

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrRTL_test.py --test-verilog
ls *.v
less RegIncrRTL__pickled.v
```

You should use `-test-verilog` regardless of whether or not you implemented your design using PyMTL RTL or Verilog RTL. Take a look at the generated Verilog RTL. If you did your design in Verilog RTL, then it won't generate any Verilog since we still have the Verilog RTL that you wrote by hand. If you did your design in PyMTL RTL, hopefully, it should be clear the one-to-one mapping from PyMTL RTL to Verilog RTL.

5.2 Implement, Test, and Translate Multi-Stage Registered Incrementer

Now let's work on composing a single registered incrementer into a multi-stage registered incrementer. We will be using *static elaboration* to make the multi-stage registered incrementer generic. In other words, our design will be parameterized by the number of stages so we can easily generate a pipeline with one stage, two stages, four stages, etc. Let's start by running all of the tests for the multi-stage registered incrementer.

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrNstageRTL_test.py
```

Use `geany` or your favorite text editor to open the implementation and uncomment the static elabroation logic to instantiate a pipeline of registered incrementers.

Before re-running the tests, let us take a look at how we are doing the testing in the corresponding test script. Use `geany` or your favorite text editor to open up RegIncrNstageRTL_test.py. Notice how PyMTL enables sophisticated testing for highly parameterized components. The test script includes directed tests for two and three-stage pipelines with various small, large, and random values and also includes random testing with 1, 2, 3, 4, 5, and 6 stages. Writing a similar test harness in Verilog would likely require 10x more code and be significantly more tedious!

Let us re-run a single test and use line tracing to see the data moving through the pipeline:

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrNstageRTL_test.py -sv -k 4stage_small
```

And now let us run all of the tests both without and with translation:

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrNstageRTL_test.py -sv
pytest ../regincr/RegIncrNstageRTL_test.py --test-verilog
ls *.v
less RegIncr4stageRTL__pickled.v
```

Notice how we have generated a wrapper which picks a specific parameter value for this instance of the multi-stage registered incrementer.

Finally, we are going to run all of the tests with the `-dump-vtb` option, which will generate a Verilog test-bench that can then be used for RTL and gate-level simulation.

```
cd $TOPDIR/sim/build
pytest ../regincr/RegIncrNstageRTL_test.py --test-verilog --dump-vtb
ls *.v
less RegIncr4stageRTL_test_4stage_random_tb.v
less RegIncr4stageRTL_test_4stage_random_tb.v.cases
```

5.3 Simulate Multi-Stage Registered Incrementer

Test scripts are great for verification, but when we want to push a design through the flow we usually want to use a simulator to drive that process. A simulator is meant for evaluating the area, energy, and performance of a design as opposed to verification. We have included a simple simulator called `regincr-sim`, which takes a list of values on the command line and sends these values through the pipeline. Let's see the simulator in action:

```
cd $TOPDIR/sim/build
../regincr/regincr-sim 0x10 0x20 0x30 0x40
less RegIncr4stageRTL__pickled.v
```

We now have the Verilog RTL that we want to push through the next step in the ASIC front-end flow.

5.4 Using Synopsys VCS for Fast-Functional Gate-Level Simulation

Recall that PyMTL3 simulation of PyMTL3 RTL or Verilog RTL uses two-state simulation. To help catch bugs due to uninitialized state (and also just to help verify the design using another Verilog simulator), we can use Synopsys VCS for four-state RTL simulation. This simulator will make use of the Verilog test-bench generated by the `-dump-vtb` option from earlier (although we could also write our own Verilog test-bench from scratch). Here is how to run VCS for RTL simulation:

```
mkdir -p $TOPDIR/asic/synopsys-vcs-rtl-sim
cd $TOPDIR/asic/synopsys-vcs-rtl-sim
vcs -full64 -sverilog +lint=all -xprop=tmerge -override_timescale=1ns/1ps \
    +incdir+../../sim/build \
    +vcs+dumpvars+vcs-rtl-sim.vcd \
```

```
    -top RegIncr4stageRTL_tb \
    ../../sim/build/RegIncr4stageRTL__pickled.v \
    ../../sim/build/RegIncr4stageRTL_test_4stage_random_tb.v
```

You should see a `simv` binary which is the compiled RTL simulator that you can run like `./simv`. It should pass the test. View the resulting waveforms by `sootty`. Browse the signal hierarchy and view the waveforms for one of the four registered incrementers. Note how the signals are initialized to X and only become 0 or 1 after a few cycles once we come out of reset. If we improperly used an initialized value, then we would see X-propagation, which would hopefully cause a failing test case.

6 Questions

1. What are the main differences between VCD (Value Change Dump) and extended VCD (EVCD) waveform format?
2. List three other waveform formats and annotate where they are used and generated.
3. What are the main differences between gate-level simulation and RTL simulation?
4. What does fast-functional simulation mean?
5. What are the main differences between forward-annotated and backward-annotated simulation?
6. What are the main differences between post-synth power analysis and post-place-and-route power analysis?
7. Would you prefer working on the ASIC flow front end or back end if you have an opportunity to design an SoC in the future, and why? (No Standard Answer)

7 Deliverables

Please add comments or explanations to all the deliverables.

1. Section 2.2: your "Create Floorplan", "Create Power Straps@server01", and "Core placement and optimization window" configuration (screenshot or Tcl script), the Console output of creating power straps command, the design view after creating power straps, the design view after Core Placement and Optimization, the screenshot of your Error Browser@server01, and the final design view.
2. Section 3.2.5: the modified `Johnson_count.v`.
3. Section 3.2.6: the output of `verify`.
4. Section 3.2.7: the screenshot of View Implementation Object in the schematic.
5. Section 4.3: `J_pre_constr.rpt`, `J_pre_max_timing.rpt`, `J_pre_min_timing.rpt`, and `Johnson_count.sdf` under the `results` directory, the Path Inspector console, Path Schematic, and Path Waveform from "out_reg[7] (rising edge-triggered flip-flop clocked by clk)" to "out[7] (output port)".
6. Section 4.4: the screenshot of the Endpoints Summary in Step 4.
7. Section 6: answers to the questions.
8. (Optional) Section 5.1: the modified `RegIncrPRTL.py`, `RegIncrVRTL.py`, and the generated `RegIncrRTL__pickled.v`.
9. (Optional) Section 5.2: the modified `RegIncrNStagePRTL.py`, `RegIncrNStageVRTL.py`, and the generated `RegIncrNStageRTL__pickled.v`, `RegIncr4stageRTL_test_4stage_random_tb.v`, `RegIncr4stageRTL_test_4stage_random_tb.v.cases`, the waveform view of `regincr.RegIncrRTL_test__test_small.vcd` and `regincr.RegIncrRTL_test__test_small_top.verilator1.vcd` in `sootty`.
10. (Optional) Section 5.3: `RegIncrRTL__pickled.v`.
11. (Optional) Section 5.4: the generated VCD file and the waveform view in `gtkwave`.

8 Grading policy

Factors	Percentage
Section 2	36%
Section 3	25%
Section 4	25%
Section 6	14%
Section 5	20% (bonus)

A Peer Evaluation Form

Part	Your work	Your partner's work	Your score	Your partner's score
Section 2				
Section 3				
Section 4				
Section 6				
Section 5				

B Troubleshooting

B.1 IC Compiler Import Designs

If the `import_designs` failed with error:

```
Loading db file '/home/users/<username>/synopsys/syn_tut/icc_32/ref/mode
↪  ls/saed32hvt_tt1p05v25c.db'
Information: Using CCS timing libraries. (TIM-024)
Warning: Unit conflict found: Milkyway technology file resistance unit
↪  is kOhm; main library resistance unit is MOhm. (IFS-007)
Loading db file '/apps/synopsys/icc-2019.03/libraries/syn/gtech.db'
Loading db file '/apps/synopsys/icc-2019.03/libraries/syn/standard.sldb'
Warning: /home/users/<username>/synopsys/syn_tut/icc_32/ref/saed32nm_hvt
↪  _1p9m: bus naming style _<%d> is not consistent with main lib.
↪  (MWNL-111)

*****  Verilog HDL translation! *****

*****    Start Pass 1 *****
Compiling source file /home/users/<username>/synopsys/syn_tut/icc_32/sou
↪  rce/Johnson_count_dc.v

*****  Pass 1 Complete *****
Elapsed =    0:00:00, CPU =    0:00:00

*****  Verilog HDL translation! *****

*****    Start Pass 2 *****
Compiling source file /home/users/<username>/synopsys/syn_tut/icc_32/sou
↪  rce/Johnson_count_dc.v
Error:   /home/users/<username>/synopsys/syn_tut/icc_32/source/Johnson_c
↪  ount_dc.v:9:  module SDFFARX1_HVT is not
↪  defined.
 (VER-500)
Error: Module 'SDFFARX1_HVT' is not defined.  (MWNL-297)
```

```
Error:   /home/users/<username>/synopsys/syn_tut/icc_32/source/Johnson_c
↪  ount_dc.v:9: ERROR: near line 9: Port connection
↪  failed.
 (VER-500)

Error: Verilog parser cannot parse the /home/users/<username>/synopsys/s
↪  yn_tut/icc_32/source/Johnson_count_dc.v source file.
↪  (MWNL-047)
No such file or directory
Error: Current design is not defined. (UID-4)
0
```

Please go to your `icc_32/ref/saed32nm_hvt_1p9m` CEL and FRAM directories and check whether the suffices of files are an underscore and a number or a colon and a number. If the suffices are an underscore and a number, then you are probably using `/home/Flow/Synopsys/Scripts/icc_32` or unzipped `Icc32.zip` in Windows. Please unzip `Icc32.zip` on the server to your workspace directory [1]. This is because the Windows file system does not support colons in file names, so the colons are converted to underscores, which are, however, not supported by the default `-bus_naming_style {[%d]}` argument.

B.2 PT-063 Library Compiler

Please run [2]

```
setenv SYNOPSYS_LC_ROOT /apps/synopsys/lc-2019.03
```

C Change Log

Fall 2022: Yihua Liu

- created this lab

References

[1] Simon0827. *Problem of ICC's create_mw_lib*. Jan. 24, 2013. URL: https://bbs.eetop.cn/thread-374092-1-1.html.

[2] soniqec and puxiancheng. *pt_shell Startup Error: Library Compiler executable path is not set*. July 4, 2021. URL: https://bbs.eetop.cn/thread-906741-1-1.html.