We strive to be the most trusted
partner for the world's hardest
engineering problems.

DOCUMENTATION OF LOGICAL EQUIVALENCE CHECKING

Date:10-Feb-25

Prepared by: Masum Saimon

We strive to be the most trusted partner for the world's hardest engineering problems.

Table of Contents

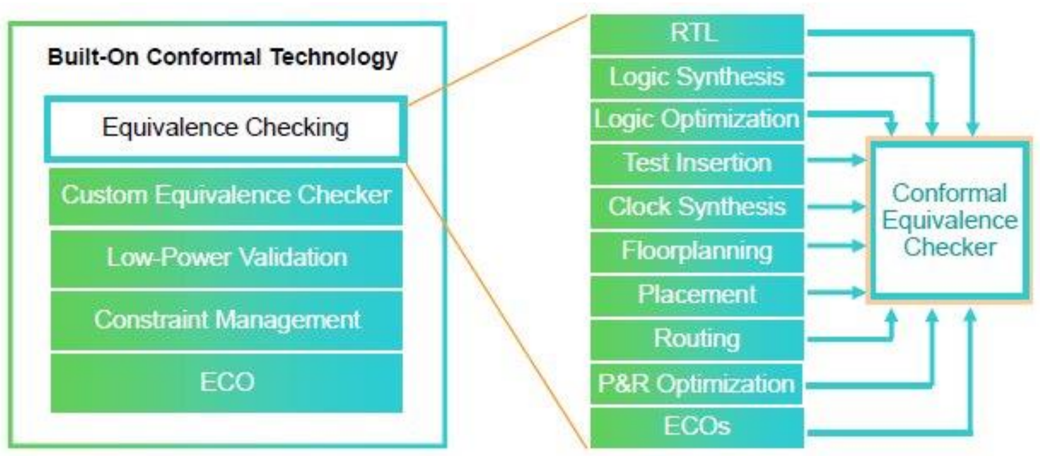
We strive to be the most trusted partner for the world's hardest engineering problems.

Logical Equivalence Check

What Is LEC?

Logical Equivalence Checking (LEC) is a process used in digital design to ensure that two different representations of a design—typically the high-level RTL (Register Transfer Level) code and the lower-level gate-level netlist (resulting from synthesis or optimization)—behave exactly the same. In simple terms, LEC confirms that no matter how the design is implemented, the end functionality remains unchanged.

- During development, a chip design undergoes numerous transformations and iterations before the final layout and each step in this process has the potential to introduce logical bugs
- Conformal/Formality Equivalence Checker compares your designs across each stage. This process is called LEC.



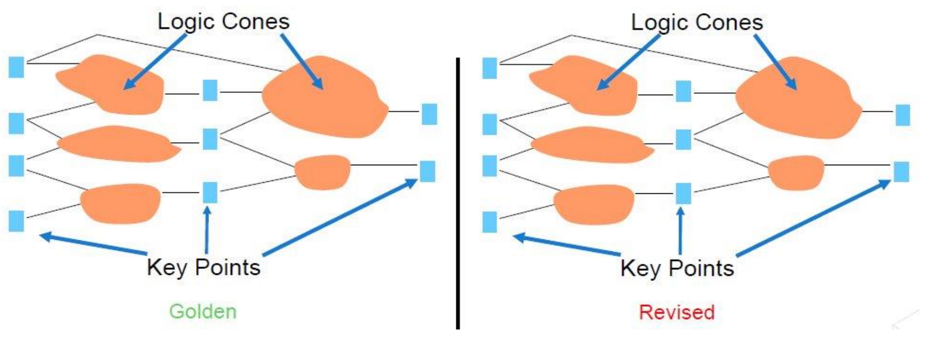
What are the key Points?

The design consists of combinational logic cones bounded by following key points:

- Primary inputs (PI), Primary outputs (PO)
- D flip flops, D latches
- Blackboxes

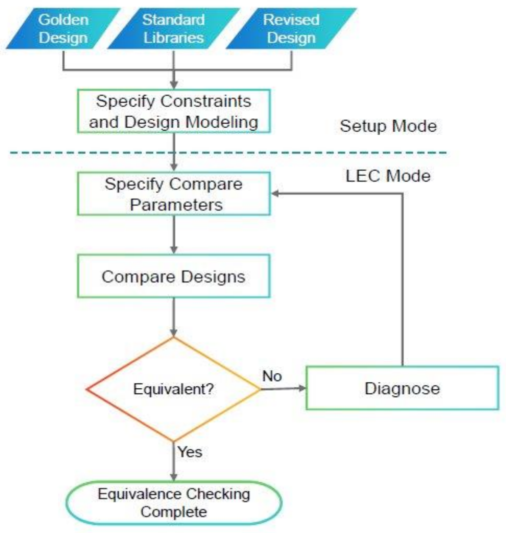
- Tie Z, Tie E, and CUT gates

- **TIE-Z gate:** Conformal creates a Z gate to drive a floating net or pin. It also creates a Z gate when modeling a tristate. The tool assumes that a Z gate can be 0 or 1
- **TIE-E gate:** By default, Conformal creates an E (error) gate for x assignment in the Revised. A Golden x assignment converts to a don't care. The tool assumes that an E gate can be 0 or 1
- **CUT gates:** Combinational loops are cut and one or more points to ease comparison



LEC Flow:

Taking Conformal Flow Diagram as Reference



##Steps to Implement:

- Synthesis:
  - syn_map
  - syn_opt

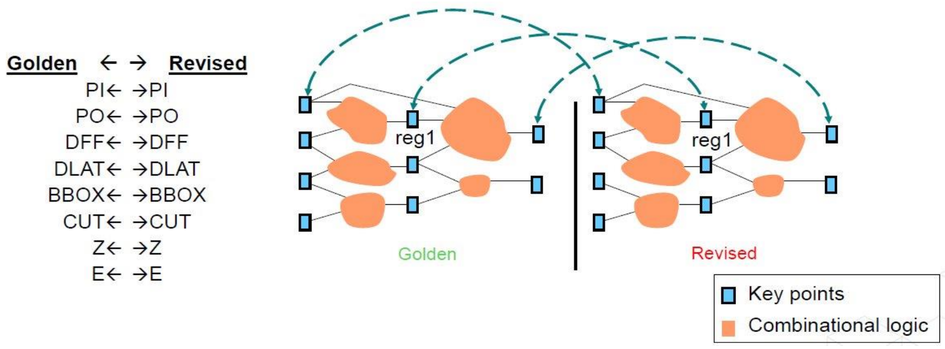
- Innovus:
  - Floorplan
  - Postroute

We strive to be the most trusted partner for the world's hardest engineering problems.

What is Mapping? Why is Mapping Necessary?

- During equivalence checking, you read in two designs to compare them:
  - The design you are comparing against must be considered Golden
  - The modified one must be considered Revised

- Mapping is pairing corresponding Golden and Revised key points
- Mapping is necessary so that the tool knows which Golden combinational logic cone to compare to which Revised combinational logic cone. **When mapping is complete, the two designs are ready for comparison**

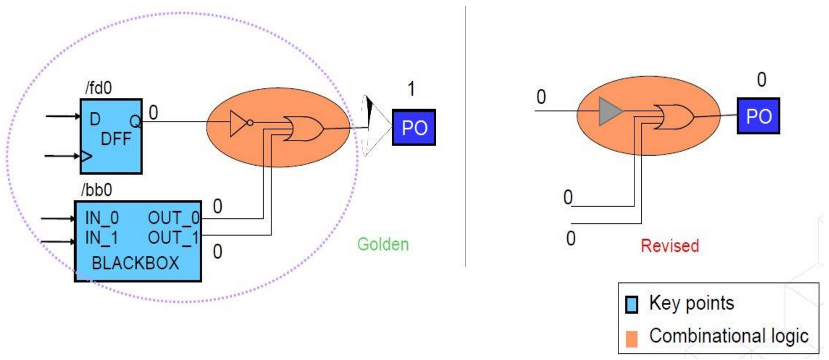


Compare Process and Compare Points

- During the compare process, all the corresponding combinational logic cones in the Golden and Revised designs are compared for equivalency. The two designs are equivalent when ALL corresponding cones are equivalent

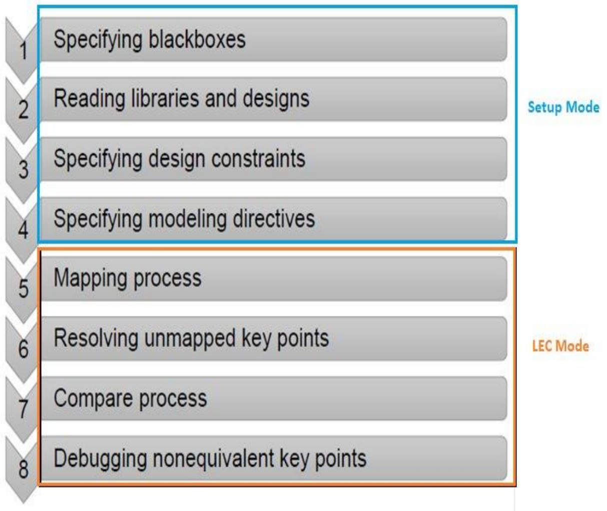
- Compare points are sink points of logic cones. For example: Primary outputs, cut gates, DFFs, D latches, and blackboxes

- The compare points are mapped key points to which a set of vectors are propagated. Then the software determines whether the resulting value matches the value at the same compare point in the other design. Therefore, any sink or end point of a logic cone can be considered as a compare point



How to Compare Designs

- Comparing Designs requires multiple steps
- After the initial setup, you can start comparing, then fix problems during the comparison iteratively until the comparison is clean
- If you have any unmapped points, you resolve them and then compare the two designs
- After comparison, if there are any nonequivalent key points, you debug those nonequivalent key points
- This is the basic flow within the LEC tool


1 Specifying blackboxes
2 Reading libraries and designs
3 Specifying design constraints
4 Specifying modeling directives
Setup Mode
5 Mapping process
6 Resolving unmapped key points
7 Compare process
8 Debugging nonequivalent key points
LEC Mode

Flow Setup Steps:

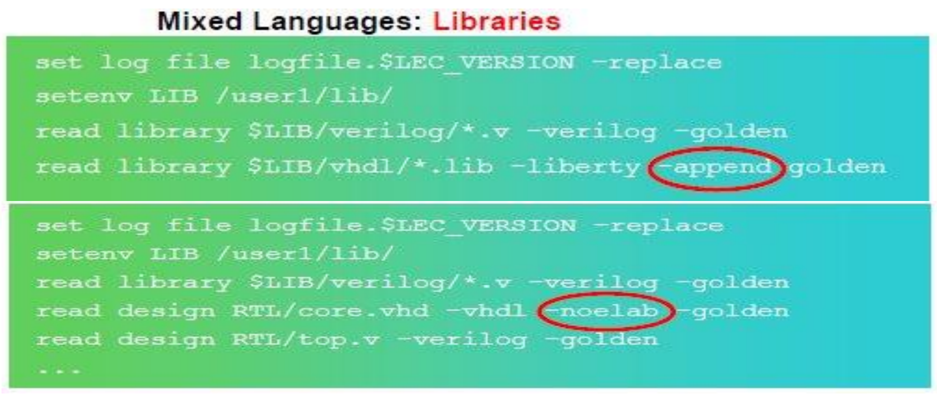
Reading libraries & Designs

- Reading Verilog Designs and libraries
- Reading VHDL Designs & Libraries
- Reading Mixed languages
- View HDL Rule Check Messages
- Enable and Disable Rule Checks
- Run Incremental Rule Checks

Basic commands:

```
add search path /user1/lib/verilog/ -lib revised
read library library.v -verilog -revised
read design revised.v -verilog -revised
```

- When reading mixed designs, use the **-noelab** option to read the VHDL files. When reading in the
- top level design, just don't use the **-noelab** option anymore; the tool will elaborate the design.



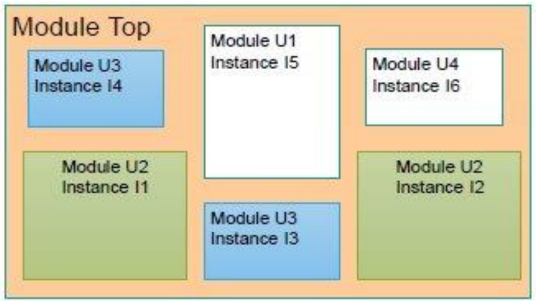
Mixed Languages: Libraries

```
set log file logfile.$LEC_VERSION -replace
setenv LIB /user1/lib/
read library $LIB/verilog/*.v -verilog -golden
read library $LIB/vhdl/*.lib -liberty -append golden
```

```
set log file logfile.$LEC_VERSION -replace
setenv LIB /user1/lib/
read library $LIB/verilog/*.v -verilog -golden
read design RTL/core.vhd -vhdl -noelab -golden
read design RTL/top.v -verilog -golden
...
```

Specifying Design Constraints

- To treat any module or instance as a blackbox, Command:
  ```
  add black box U2 -module -golden
  add black box /Top/I3 –golden
  ```
  - In this graphic, instances I1 and I2 are from the same module. Therefore, the command with the module option causes both of these instances to be treated as blackboxes . The command without the module option only treats one instance as a blackbox.

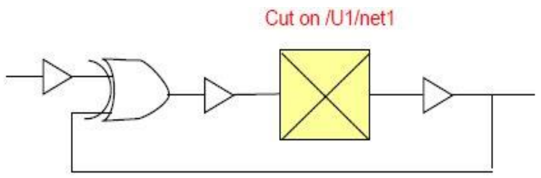


- To specify the global behavior of floating signals in the designs,

  Command:
  ```
  set undriven signal 0 -revised
  ```

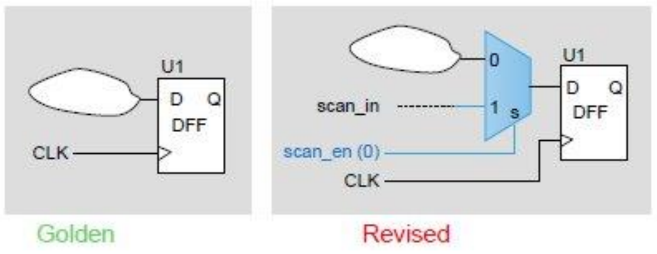
- To specify the cut points for breaking combinational feedback loops,

  Command:
  ```
  add cut point /U1/net1 -revised
  ```



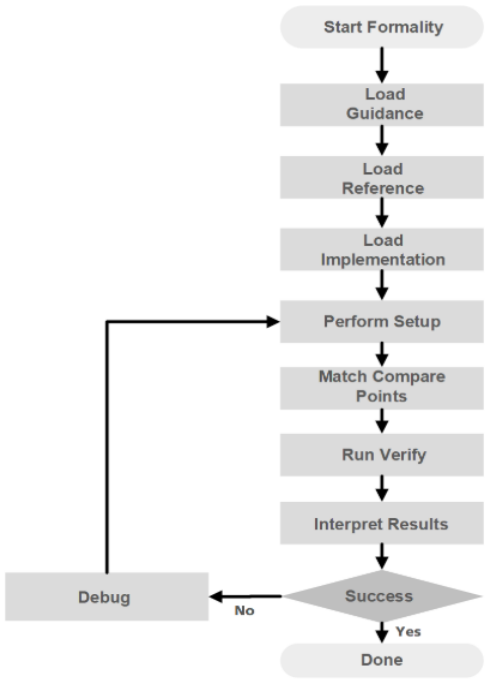
- To specify the mode of circuit operation under which comparison will take place (for example, functional versus scan operation),
  Command:
  ```
  add pin constraint 0 scan_en -revised
  ```

- To apply a logic 0 or a logic 1 to the output of an internal DFF or a D latch, Command:
  ```
  add instance constraint 0 U1 –revised
  ```



Taking Formality Flow as Reference:

- The flow outlines the Formality design verification process flow. It represents specific steps to perform an equivalence check using Formality. Each topic corresponds to one or more steps in the flow.



Load Guidance:

The load guidance step of the Formality process flow is the point at which you can opt to provide setup information about design changes caused by other tools used in the design flow.

```
fm_shell (setup)> set_svf design.svf
```

- Files containing this guidance information are known as SVF files, and they generally have the .svf extension. An SVF file enables the tool to process the content and store data for use during the matching step that follows. Guidance is recommended in a Synopsys design implementation flow, while it is optional when verifying designs modified by third-party tools.

Load Designs:

To perform verification, you must first provide Formality with two designs. The golden design, the one that is known to be functionally correct, is the reference design. The second design is a modified version of the reference design and is known as the implementation design. This is the design that you want to verify against the reference design for functional equivalence.

```
fm_shell (setup)> read_verilog -r top.v
```

- Formality can be used to verify two RTL designs against each other, two gate-level designs against each other, or an RTL design against a gate-level design.
- The design files that you load into Formality can use only synthesizable SystemVerilog, Verilog, or VHDL constructs or can be in the Synopsys internal database format (**.db, .ddc, or Milkyway database**).

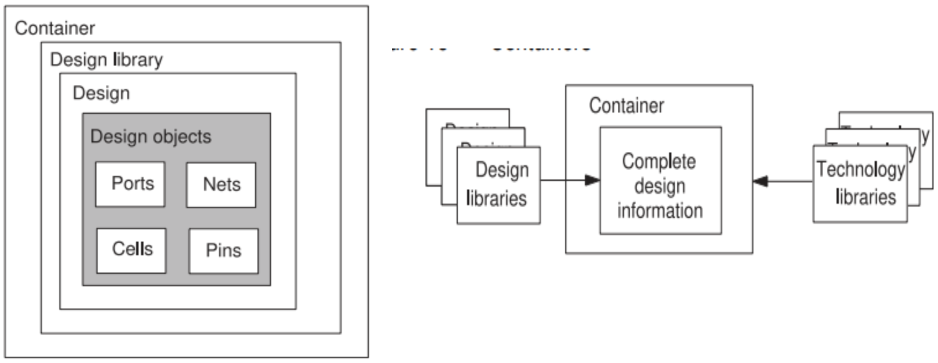
We strive to be the most trusted partner for the world's hardest engineering problems.

- After designs are loaded into Formality in this step of the process flow, you can control certain aspects of the verification process, such as establishing environmental parameters.

1. Setting Up The Designs:

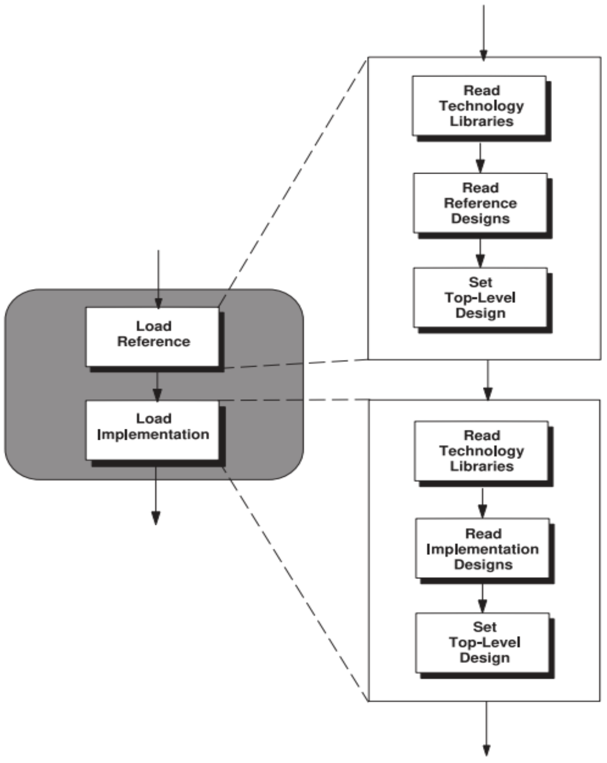
A container is a complete, self-contained space into which Formality reads designs. It is typical for one container to hold the reference design while another holds the implementation design. In general, you do not need to concern yourself with containers. You simply load designs in as either reference or implementation.

A container typically includes a set of related technology libraries and design libraries that fully describe a design that is to be compared against another design. A technology library is a collection of parts associated with a particular vendor and design technology. A design library is a collection of designs associated with a single design effort. Designs contain design objects such as cells, ports, nets, and pins. A cell can be a primitive or an instance of another design.



In Formality, one container is always considered the current container. Unless you specifically set the current container, Formality uses the last container into which a design is read. That container remains the current container until you specifically change it or you create a new container. Many Formality commands operate on the current container by default (when you do not specify a specific container).

- Formality Read Design Process Flow:



2. Design Loading Steps:

Loading designs into Formality consists of three main steps: • Load the technology libraries (optional, as needed) • Load the design files • Set the top-level block to compare These three steps are done for both the reference and implementation designs and are nearly identical in process. This section is, therefore, broken into the following two subsections, with most of the details captured solely in the load reference design section.

• Loading the Reference Design

• Loading the Implementation Design

○ Reading Technology Libraries

As needed, read in the technology libraries that support your reference design. If you do not specify a technology library name with the commands described in the following section, Formality uses the default name, TECH_WORK

➢ Reading Synopsys (.db) Format

Synopsys internal database (.db) library files are shared by default. If you read in a file without specifying whether it applies to the reference or implementation design, it applies to both.

```
To read cell definition information contained in .db format files,
fm_shell
read_db file_list
[-libname library_name ] [-merge ] [-replace_black_box ]
```

We strive to be the most trusted
partner for the world's hardest
engineering problems.

Reading Designs

Read a reference design into Formality based on the language that represents it. At its most basic, (where the -r option indicates the reference design.) Specify one of the following, depending on the design type:

```
fm_shell
read_verilog -r files
read_sverilog -r files
read_vhdl -r files
read_ddc -r files
read_milkyway -r files
read_db -r files
```

Setting The Top level Design:

To set the top-level design for the reference design,

```
fm_shell
Specify: set_top
[-vhdl_arch name ]
[moduleName | designID | -auto ]
[-parameter value ]
```

- If you are elaborating VHDL and you have more than one architecture, use the **-vhdl_arch** option. The **set_top** command tells Formality to set and link the top-level design. If you are using the default r and i containers, this command also sets the top-level design as the reference or implementation design.

Loading the Implementation Design

This section provides an overview of the read-design process flow for the implementation design.

Note: If you already specified a .db library for the reference design, it is automatically shared with the implementation design.

Many Formality shell commands can operate on either the reference or implementation design. These commands all have a switch to indicate which design container is used for that command. The **-r** switch refers to the reference design or container. The **-i** switch refers to the implementation design or container. Use the -i option to specify the implementation container or use the **-container container_name** option to provide a specific container name. From within the GUI, use the Implementation tab to read an implementation design.

Library Loading Order

Formality has the ability to load and manage multiple definitions of a cell, such as synthesis .db format files, simulation .db format files, and Verilog or VHDL netlists. The order in which the library files are loaded determines which library model is used by Formality. If the libraries are not loaded in the correct sequence, it can lead to inconsistent or incorrect verification results. If you are a library provider, you should deliver explicit Formality loading instructions for multiple libraries. One way to do this is to provide a Formality script that loads the library files (such as .db, .v, and .vhd) in the correct order.

We strive to be the most trusted partner for the world's hardest engineering problems.

IEEE Std 1735-2014 Encryption of RTL Files

The Formality tool supports IEEE Std 1735-2014 encryption for SystemVerilog and Verilog files. This provides an industry-standard method to encrypt a file against one or more public keys, such that only those key owners can decrypt the file.

To enable this feature, set the following application variable before reading or analyzing your RTL:

```
set_app_var hdlin_enable_ieee_1735_support true
```

The following tool behaviors apply to encrypted RTL:

• Elaboration errors are reported with protected names instead of a generic message, for example

```
               Error: You are using an identifier '' which is not declared in that
scope. (File: include_pkg_er2.vp Line: 1) (FMR_VLOG-606)
```

The Formality tool suppresses encrypted objects from the following report and find commands only: ◦ ***report_black_boxes*** ◦ ***report_clocks*** ◦ ***report_designs*** ◦ ***report_parameters*** ◦ ***report_fsm*** ◦ ***report_libraries*** ◦ ***report_truth_table*** ◦ ***report_source_path*** ◦ ***report_hierarchy*** ◦ ***find_designs*** ◦ ***find_cells*** ◦ ***find_nets*** ◦ ***find_pins*** ◦ ***find_ports***

3. Reading Technology Cell Libraries

There is a range of optional functionality available to you through use of the containers into which the Formality designs are read. You can use the SVF guidance flow to control specific variables. SystemVerilog, Verilog, and VHDL cell definition information must be in the form of synthesizable RTL or a structural netlist. In general, Formality cannot use behavioral constructs or simulation models, such as VHDL VITAL models.

➢ Reading SystemVerilog, Verilog, and VHDL Cell Definitions

To read cell definition information contained in SystemVerilog, Verilog, or VHDL RTL files, do the following:

```
fm_shell
Specify:
set_app_var hdlin_library_file file
set_app_var hdlin_library_directory directory
```

Performing Setup:

After reading designs into the Formality environment and linking them, set the design specific options for Formality to perform verification. For example, if you are aware of certain areas in a design that Formality cannot verify, you can prevent the tool from verifying the areas. Or, to

We strive to be the most trusted partner for the world's hardest engineering problems.

improve the performance of verification, you can declare blocks in two separate designs black boxes.

Common Operations

Tasks and procedures that are performed often to setup a design for verification are described in the following subsections:

• **Handling Black Boxes**

```
set_app_var hdlin_interface_only ”DRAM* SRAM*”
```

This variable is not cumulative. Subsequent specifications cause Formality to override prior specifications. Therefore, if you want to mark all RAMs with names starting with DRAM* and SRAM* as black boxes, for example, specify both on one line. To report the black boxes, use the `report_black_boxes` command.

• **Specifying Constants**

Formality recognizes two types of constants: design and user-defined. Design constants are nets in your design that are tied to a logic 1 or 0 value.
User-defined constants are ports or nets to which you attach a logic 1 or 0 value, using Formality commands. User-defined constants are especially helpful when several problem areas exist in a circuit and you want to isolate a particular trouble spot by disabling an area of logic. For example, suppose your implementation design has scan logic and you do not want to consider it in the verification process.
You can assign a constant to the scan-enable input port to disable the scan logic and take it out of the verification process.
You can apply a user-defined constant to a port or net. However, if you assign a constant to a net with a driver, Formality displays a warning message.
Formality tracks all user-defined constants and generates reports. You can specify how Formality propagates constants through different levels of the design hierarchy.
You can manage user-defined constants by performing the tasks in the following sections.

To set a net, port, cell, or pin to a constant state of 0 or 1,

```
set_constant [-type type ] instance_path constant_value
```

For constant_value, specify either 0 or 1. If more than one design object shares the same name as that of the specified object, use the -type option and specify the object type (either port or net). You can specify an object ID or instance-based path name for instance_path. Use the latter to apply a constant to a single instance of an object instead of all instances. In addition, you can use wildcards to specify objects to be set constant.

To remove a user-defined constant,

```
remove_constant -all
or
remove_constant [-type ID_type ] object_ID ...
```

• **Specifying External Constraints**

Sometimes, you might want to restrict the inputs used for verification by setting an external constraint. By setting an external constraint, you can limit the potential differences between two designs by eliminating unused combinations of input values from consideration, thereby reducing verification time and eliminating potential false failures that can result from verification with the unconstrained values. When you define the allowed values of, and relationships between, primary inputs, registers, and black box outputs, and allow the verification engine to use this information, the resulting verification is restricted to identify only those differences between the reference and implementation designs that result from the allowed states.

We strive to be the most trusted partner for the world's hardest engineering problems.

Typical constraint types that you can set are
• One-hot: One control point at logic 1; others at logic 0.
• One-cold: One control point at logic 0; others at logic 1.
• Coupled: Related control points always at the same state.
• Mutually exclusive: Two control points always at opposite states.
• User-defined: You define the legal state of the control points.

The following paragraphs describe three cases where setting external constraints within verification is important.

To define an external constraint,
```
set_constraint type_name [-name constraint_name [-map map_list1 map_list2 ]
constraint_type control_point_list [designID]
```

To create a user-defined constraint type and establish the mapping between the ports of a design that define the constraint and control points in the constrained design:
```
create_constraint_type type_name [designID ]
```

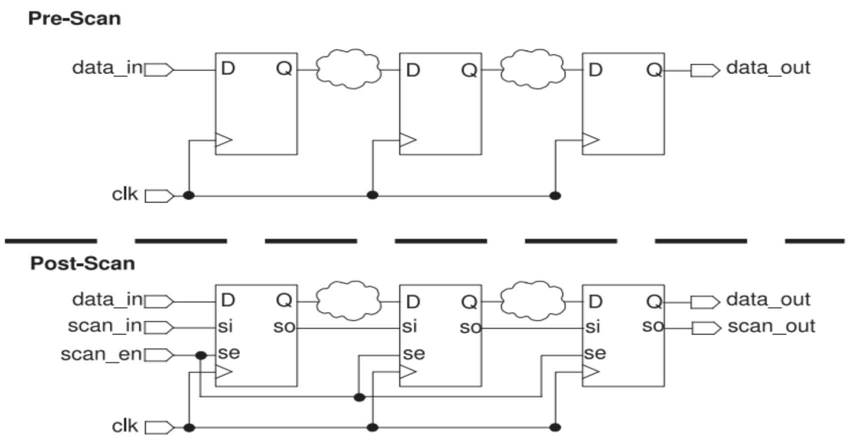
A constraint module has the following characteristics:
• One or more inputs and exactly one output
• Outputs in logic 1 for a legal state; otherwise logic 0
• No inouts (bidirectional ports)
• No sequential logic
• No three-state logic
• No black boxes

• Combinational Design Changes

This section describes how to prepare designs with combinational design changes, such as
• Internal scan insertions
• Boundary-scan insertions
• Clock tree buffers

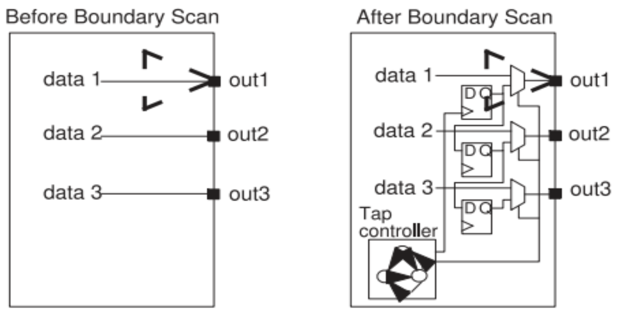
Disabling Scan Logic

Insert internal scan to set and observe the state of the registers internal to a design. During scan insertion, the scan flops replace flip-flops. The scan flops are then connected
Common Operations Feedback into a long shift register. The additional logic added during scan insertion means that the combinational function has changed,



Disabling Boundary Scan in Your Designs:
Boundary scan is similar to internal scan in that it involves the addition of logic to a design. This added logic makes it possible to set and observe the logic values at the primary inputs and outputs (the boundaries) of a chip, at the primary inputs and outputs (the boundaries) of a chip, as shown in Figure 18. Boundary scan is also referred to as the IEEE 1149.1 Std. specification.

Boundary Scan Insertion:



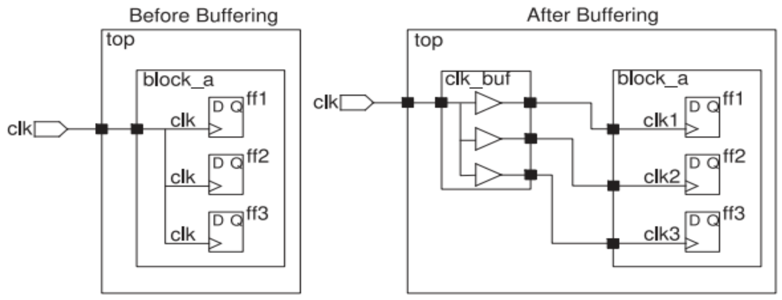
Designs with boundary-scan registers inserted require setup attention because
• The logic cones at the primary outputs differ
• The boundary-scan design has extra state-holding elements

Boundary scan must be disabled in your design in the following cases:
• If the design contains an optional asynchronous TAP reset pin (such as TRSTZ or TRSTN), use set_constant on the pin to disable the scan cells.
• If the design contains only the four mandatory TAP inputs (TAS, TCK, TDI, and TDO), force an internal net of the design with the set_constant command. For example,

```
fm_shell (setup)> set_constant gates:/WORK/TSRTS 0
fm_shell (setup)> set_constant gates:/WORK/alu/somenet 0
```

Managing Clock Tree Buffering
Clock tree buffering is the addition of buffers in the clock path to allow the clock signal to drive large loads.

Clock Tree Buffer



Without the correct setup, verification of block_a fails. However, it would succeed with top-down verification. As shown in the figure, before buffering, the clock pin of ff3 is clk. After buffering, the clock pin of ff3 is clk3. The logic cones for ff3 are different, resulting in a failing point. To manage the clock tree buffering, you must use the set_user_match command to specify that the buffered clock pins are equivalent. With the set_user_match command you can match one object in the reference design to multiple objects in the implementation design (1-to-n matching). For example, if you want to match a clock port, clk, in the reference design to three clock ports in the implementation design, clk, clk1, and clk2, you can use

```
        set_user_match r: /WORK/design/clk i:/WORK/design/clk i:/WORK/ design/
clk1 i:/WORK/design/clk2
```

Alternatively, you can issue multiple commands for each port in the implementation:

```
        set_user_match r: /WORK/design/clk i:/WORK/design/clk
        set_user_match r: /WORK/design/clk i:/WORK/design/clk1
        set_user_match r: /WORK/design/clk i:/WORK/design/clk2
```

If you know a clock port is inverted, use the -inverted option to the set_user_match command. Therefore, if your reference design had a clock port, clk, and your implementation design had a clk port and an inverted clock port, clk_inv, you would use the following command:

```
        set_user_match r:/WORK/design/clk i:/WORK/design/clk
        set_user_match -inverted r:/WORK/design/clk i:/WORK/design/clk_inv
```

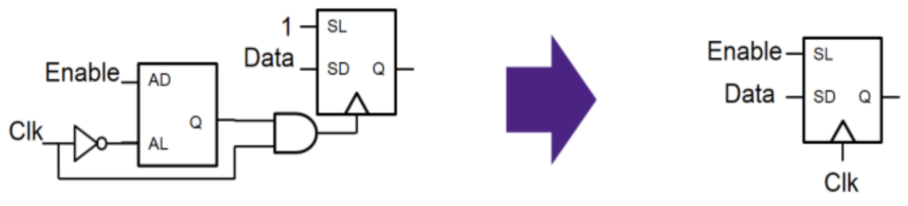
• Sequential Design Changes

Sequential design changes require setup before verification. Sequential design changes include:

- Clock gating
- Automatic clock gating
- Pushing inversions across registers
- Retiming

Reverse Clock Gating

- The Formality tool processes both reference and implementation designs by searching for clock-gating latches. At the beginning of the verification stage, the tool transforms these latches into synchronous load control logic and clock signals on the downstream flip-flops. In the simplest form, the enable pin on the clock gating latch becomes the synchronous load (SL) function of the flip-flop, and the clock-gating latch clock signal is connected to the clock pin on the flip-flop. This reverses any clock gates inserted so that it looks like a design without clock gating for verification purposes.

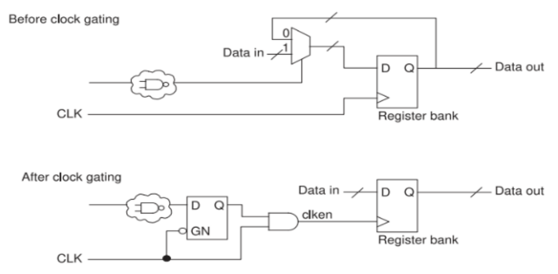


- When the `verification_clock_gate_reverse_gating` variable is set to true, the reverse clock-gating algorithm takes precedence

- over both the `verification_clock_gate_hold_mode`
- and `verification_clock_gate_edge_analysis` variables.

Setting Clock Gating:

- Clock gating applies to synchronous load-enable registers, which are groups of flip-flops that share the same clock and synchronous control signals. Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks. In its simplest form, clock



We strive to be the most trusted partner for the world's hardest engineering problems.

gating is the addition of logic at the register's clock input path that disables the clock when the register output is not changing.

- To verify designs with clock gating in the tool,
  `set_app_var verification_clock_gate_hold_mode [none | low | high | any | collapse_all_cg_cells ]`
- The verification_clock_gate_hold_mode variable has the following values:
  • none (off) is the default.
  • low to specify clock gating that holds the clock low when inactive.
  • high to specify clock gating that holds the clock high when inactive.
  • any to specify both the high and low styles of clock gating within the same design.
  • collapse_all_cg_cells has the same effect as the low value. In addition, if the clock-gating cell is in the fan-in of a register and in the fan-in of a primary output port or black-box input pin, the cell is treated as a clock-gating cell in all of those logic cones.
  The verification_clock_gate_hold_mode variable affects the entire design. It cannot be placed on a single instance and enabling it causes slower runtime.
- **To enable automatic verification of the clock-gate designs,**
  `set_app_var verification_clock_gate_edge_analysis true`

• Low-Power Designs

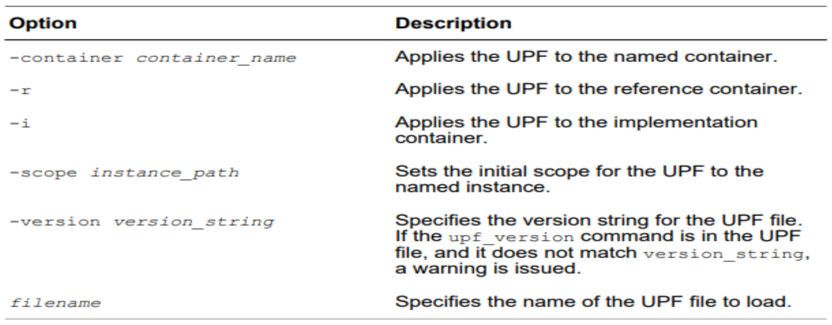
Formality verifies and supports designs that use the IEEE 1801, also known as the Unified Power Format (UPF) standard.
Formality reads UPF files that are created at each stage of the design process, allowing verification of the intermediate netlists produced by Design Compiler and IC Compiler.

In UPF verification flow, the tool verifies designs consisting of
• A design source file with the UPF file
• A Design Compiler netlist with the generated UPF file
• An IC Compiler netlist with the generated UPF file
• An IC Compiler power and ground connected netlist Special steps might be required to handle designs that contain retention registers.

Loading the UPF File

To load and use the UPF information file into Formality, set the top design in the container and issue the following command in the setup mode:

```
load_upf [ -container container_name | -r | -i ] [-scope instance_path ] [ -
version version_string ] filename
```



Option	Description
-container *container_name*	Applies the UPF to the named container.
-r	Applies the UPF to the reference container.
-i	Applies the UPF to the implementation container.
-scope *instance_path*	Sets the initial scope for the UPF to the named instance.
-version *version_string*	Specifies the version string for the UPF file. If the upf_version command is in the UPF file, and it does not match version_string, a warning is issued.
filename	Specifies the name of the UPF file to load.

We strive to be the most trusted partner for the world's hardest engineering problems.

Common areas where LEC fail:

- If multibit flops are used in the design, then the issue of mapping golden netlist versus revised netlist will crop up, as flop names will be changed in revised netlist.
- Clock gating cells not getting mapped after cloning in revised netlist.
- Logical connectivity breaks during timing fixing or while doing manual ECO.
- Functional ECO implementation.
- Missing DFT constraints.

Why Equivalence Checking is Important:

- **Design Confidence:** We want to be sure that when we optimize or make changes, our design still works as intended.
- **Error Prevention:** Catching mismatches early prevents costly mistakes later in manufacturing.
- **ECO Verification:** When making last-minute changes (Engineering Change Orders), LEC ensures those changes haven't broken the design.

Other Benefits:

- Less reliance on gate level simulation.
- Boosted confidence in new tool revisions for synthesis and place & route.
- Watch-dog for poor RTL coding areas in the design.
- Nearly exhaustive proof of equivalence without writing test patterns.
- Decreased risk of missing bugs inserted by the back-end process.

Input Files Required

Before you run an LEC, you must have several input files:

1.1. RTL File (Input Design)

- **Description:** Your original design code written in Verilog, VHDL, or another HDL.
- **How to Generate:**
  - Written by the designer.
  - Stored as .v, .sv, or .vhd files.

1.2. Gate-Level Netlist File (Synthesized Design)

- **Description:** The output from the synthesis tool that translates RTL into a network of gates.

We strive to be the most trusted partner for the world's hardest engineering problems.

- **How to Generate:**
    - Run your synthesis tool (e.g., Synopsys Design Compiler, Cadence Genus) on your RTL.
    - The tool produces a netlist file (often with a extension for Verilog netlists or a proprietary format).

1.3. Mapping/Constraint File

- **Description:** A file that helps the LEC tool map signals between the RTL and the netlist (e.g., clock, reset, and interface signals).
- **How to Generate:**
    - Often manually created or generated as part of the synthesis flow.
    - Contains information on which RTL signals correspond to which netlist signals.

1.4. Library Files

- **Description:** Standard cell libraries and timing libraries used during synthesis.
- **How to Use:**
    - Ensure the LEC tool is pointed to these files so it understands the cell functions.

Process of Running LEC

The overall LEC process can be divided into several stages:

Stage 1: Environment Setup & File Import

Steps:

Launch LEC Tool:

Open your preferred LEC tool (e.g., Synopsys Formality, Cadence Conformal).

Import Input Files:

- **RTL File:** Import your multiplier.v (or similar) into the tool.
- **Gate-Level Netlist:** Import multiplier_netlist.v.
- **Mapping File:** Load the mapping constraints file (mapping.map).
- **Library Files:** Set the paths to the standard cell library files.

Tool Command Example (Pseudo-Command):

```
lec_tool -rtl multiplier.v -netlist multiplier_netlist.v -map mapping.map -lib std_cell.lib
```

Note: The exact command depends on the tool you use.

We strive to be the most trusted partner for the world's hardest engineering problems.

Stage 2: Running the Equivalence Check

Steps:

Setup Verification Options:

- Select hierarchical mode if verifying module-by-module.
- Choose sequential verification if the design includes state elements (flip-flops).

Start the LEC Process:

- The tool will compare the RTL and netlist logic.
- It checks combinational paths, sequential elements, and ensures that all mapped signals match.

Monitor the Process:

- The tool may provide real-time logs or progress indicators.

What Happens Internally:

- The LEC tool cross-references each logic block.
- It uses formal methods and simulation-based techniques to prove equivalence.

Stage 3: Output Files & Results

After running, the tool generates output files containing the results:

3.1. Equivalence Report

- **Description:** A detailed report stating whether the RTL and netlist are functionally equivalent.
- **Contents:**
    - Summary of pass/fail status.
    - List of mismatches (if any) with details about the signal or block.
    - Debug information or counterexamples if discrepancies exist.
- **Example:**lec_report.txt or a tool-specific report file.

3.2. Log Files

- **Description:** Files that log the commands executed, warnings, and errors.
- **Usage:** Helpful for debugging issues if the equivalence check fails.
- **Example:**`lec_log.txt`.

3.3. Waveform Files (Optional)

- **Description:** Some tools can generate simulation waveforms for mismatched signals.
- **Usage:** Visualize the signal behavior differences between RTL and netlist.
- **Example:**`debug_wave.vcd`.

Stage 4: Debugging and Fixes

If mismatches are found:

Examine the Equivalence Report:

- Identify the modules or signals where differences occur.

Use Debug Tools:

- Open waveform files if provided.
- Trace the signal paths to understand why the mismatch exists.

Iterate on the Design:

- Modify the RTL or adjust synthesis constraints.
- Re-run synthesis if needed, then re-run the LEC.
- Update the mapping file if necessary.

Final Step:

- **Re-run LEC:** Once adjustments are made, run the LEC tool again until the report confirms full equivalence.

We strive to be the most trusted partner for the world's hardest engineering problems.

References:

1. Formality and Formality ECO Online Help
2. Genus Synthesis Flows Guide -- Exiting Genus
3. A Guide on Logical Equivalence Checking