TOSHIBA

ASIC
Design
Manual

Formality Design Kit User Guide

2002

Formality Design Kit User Guide

Published in September, 2002
Document ID: 45117M-0209

The information contained herein is subject to change without notice.

TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property. In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications. Also, please keep in mind the precautions and conditions set forth in the "Handling Guide for Semiconductor Devices," or "TOSHIBA Semiconductor Reliability Handbook" etc.

The Toshiba products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.). These Toshiba products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury ("Unintended Usage"). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc. Unintended Usage of Toshiba products listed in this document shall be made at the customer's own risk.

Toshiba does not take any responsibility for incidental damage (including loss of business profit, business interruption, loss of business information, and other pecuniary damage) arising out of the use or disability to use the product.

The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patents or patent rights of TOSHIBA or others.

The products described in this document may include products subject to the foreign exchange and foreign trade laws.

Design Compiler and Formality are trademarks of Synopsys, Inc. All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Preface

This user guide describes how ASIC designers can use the Synopsys Formality product to formally verify that a revised gate-level design is functionally equivalent to a reference gate-level design.

This guideline assumes that designers are experienced users of Formality. Experience of Synopsys Design Compiler is helpful, but not required.

Technology Supported

TC240 and later technologies are supported by this design kit.

Formality Version Supported

Formality 2000.11-FM1.0-SP2 to 2002.05 are supported by this design kit.

Other Sources of Information

- Synopsys Manuals
  - Formality User Guide
  - Formality Reference Manual (*formality_install_dir*/doc/fm/*.pdf)
- Toshiba Documentation

  "*User Guide*" and "*Command Reference*" manuals of the following sign-off systems.
  - *PrimeTime Sign-Off System*

- *VSO*
- *VITALSO*

Contents

Style Conventions

The following syntax and notation conventions are used throughout this manual:

Courier Bold	Indicates any reference to a command. In many cases, it indicates keywords or command line options that you must enter literally (or exactly as shown).
Courier Oblique	Indicates variable information, or user-defined arguments for which you must substitute a name or a value.
Courier Plain	In examples, shows text from files and reports printed by the system.
...	Elements preceding ellipses may be repeated any number of times.
[**A**]	Indicates an optional argument.
[**A**|**B**|**C**]	Indicates a list of choices from which you can choose one.
{**A**|**B**|**C**}	Indicates a list of choices from which you must choose one.
underscore|**B**|**C**	Indicates a default option or argument when there is more than one choice.

You must type all other punctuation symbols such as the comma, colon, slash, backslash, and quotation mark, as shown.

CHAPTER 1 Overview

The Formality Design Kit integrates the formal verification products into the ASIC design flow. This chapter introduces the concepts of formal verification and logic equivalence checking.

What is Formal Verification?

Formal verification (FV) is a process used to determine if a design conforms to its specification. Logic equivalence checking is the most widely used formal verification technique today.

Logic equivalence checking employs mathematical methods to comprehensively prove the functional equivalence between two versions of a design. It ensures that a design retains the same behavior throughout the design flow. It is not a replacement for simulation because the functionality of a reference design must first be thoroughly verified by logic simulation.

Also, FV does not verify timing. Therefore, FV cannot exist by itself in a sign-off flow. Static timing analysis (STA), FV, and logic simulation are three types of verification that are complementary in the verification flow for complex designs.

Basic Equivalence Checking Concepts

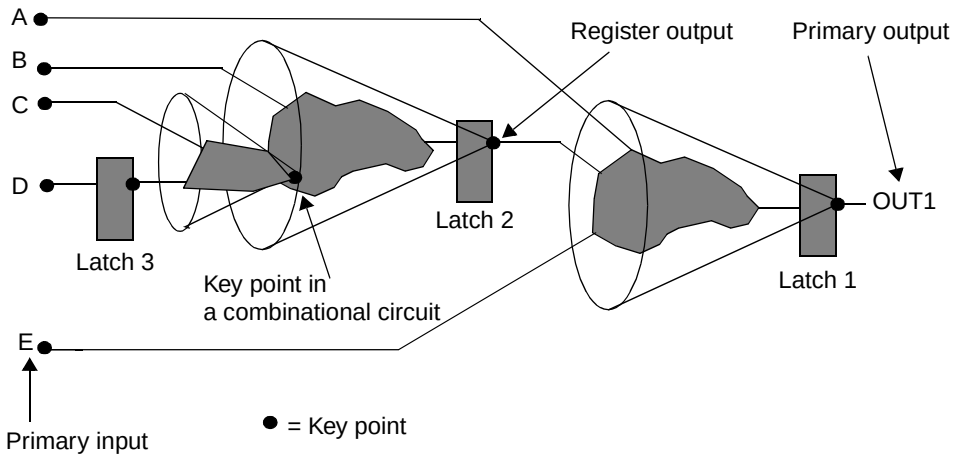
Logic Cones

Typically, FV divides a design into units of logic cones. The definition of a logic cone can vary slightly between tools, but the basic concept remains the same. In FV, a logic cone consists of a logic that drives an end key point. A logic cone can have multiple source key points.

Key Point

Key points (KP) are points in a design used by FV to compare logic. They can be primary inputs, primary outputs, nodes that hold their values over time, for example, outputs of flip-flops or latches. When verification is not completed due to a large combinational circuit, FV may generates key points in a combinational circuit to divide a logic cone. The method how to generate logic cones varies between tools. Examples of logic cones and key points are given in Figure 1–1.

Figure 1–1 Logic Cones and Key Points



Excision

Excision is the removal, from a design, of specific blocks of logic that do not need verification. The most common candidates for excision are RAMs, ROMs, PLLs, IP cores (embedded DRAMs, microprocessors, and so forth), and analog blocks.

In general, any blocks that have been verified by the customer, Toshiba, or the IP provider should be excised. During netlist compilation, inputs to the excised model become primary outputs of the top-level main block, and outputs become primary inputs.

Constraints

Constraints are logic values that are applied to signals and that limit those signals to specified values. This, in turn, can eliminate logic that might exist in one version of a design but not the other. One example is the comparison of a pre-scan inserted design and a post-scan inserted design. This type of comparison is done by limiting signals to specified values from the outside world so that a design can behave in the normal operation mode.

Mapping and Comparing

During netlist compilation in FV, a design is divided into logic cones, and key points are identified. Each logic cone is expressed as a logic equation in the formal verification database. The verification process consists of two main tasks:

- Mapping
- Comparing

First, FV tries to map key points in one design to key points in the other. FV uses two methods to map key points:

- Name-based Mapping

  Match signal names between two designs by their names. This method requires very little work in FV and therefore is fast, but can not be used for two designs containing different signal names or design hierarchies.

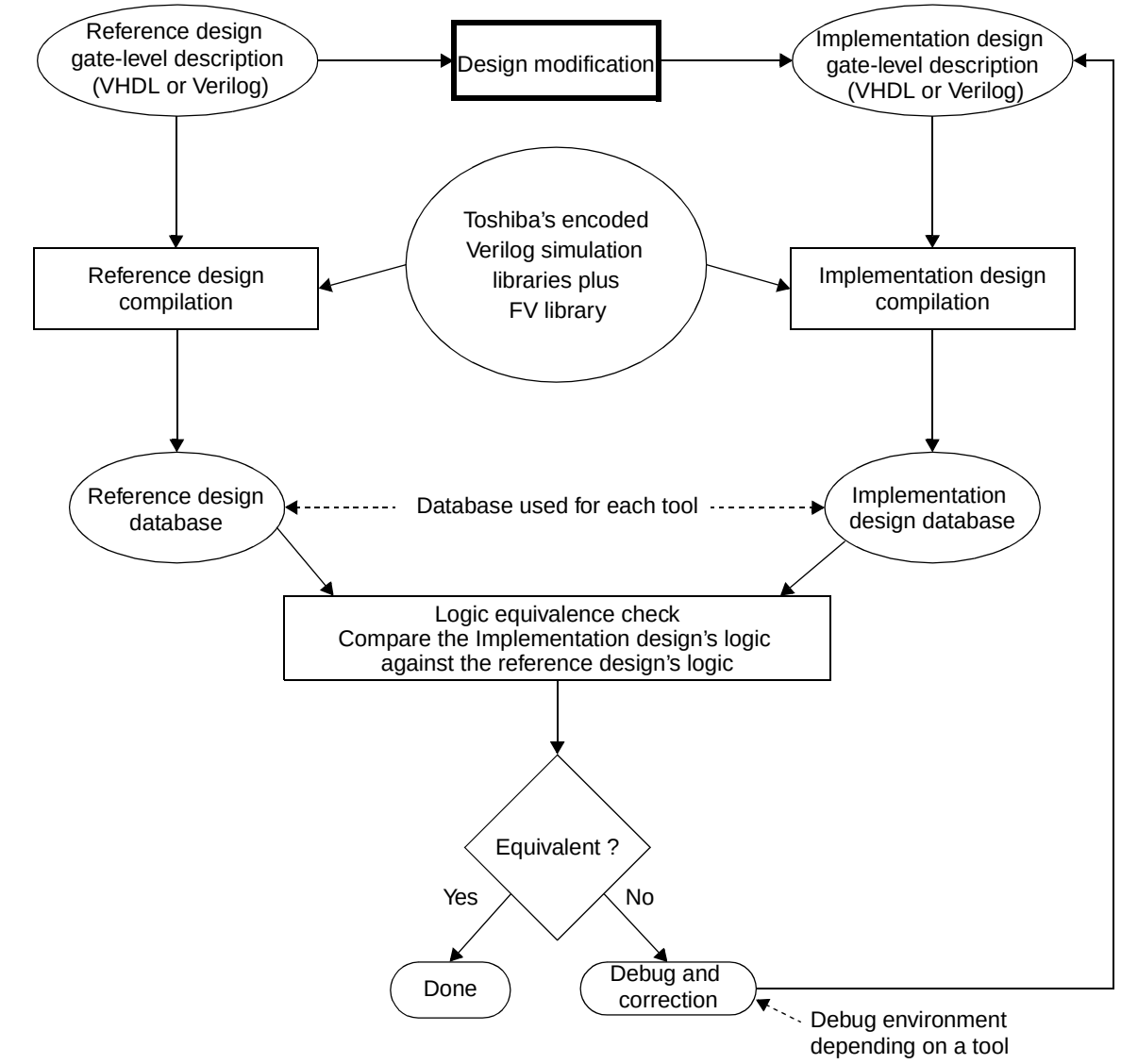
- Nameless mapping

  When the signal names are different in two designs, the mapper must find matching key points through rigorous search and proof. This method can be used for designs containing different signal names or design hierarchies. Since this method can be time-consuming for two designs containing many different signal names or different design hierarchies, you should help the mapper by providing a name mapping rule file whenever applicable.

After mapping a pair of key points between two designs, FV runs an equivalence check on logic cones driving key points. FV performs mathematical proofs on the two symbolic logic equations to determine whether they are equivalent. The proof is comprehensive because the logic is evaluated under all possible conditions. (That corresponds to 100% toggle coverage during logic simulation.)

After all logic cones have been proven to be equivalent or not equivalent, an analysis report is generated which contains all the important information about the comparison. Figure 1–2 shows the FV flow for gate-to-gate comparisons.

Figure 1–2 FV Flow for Gate-to-Gate Comparison



Basic Equivalence Checking Flow

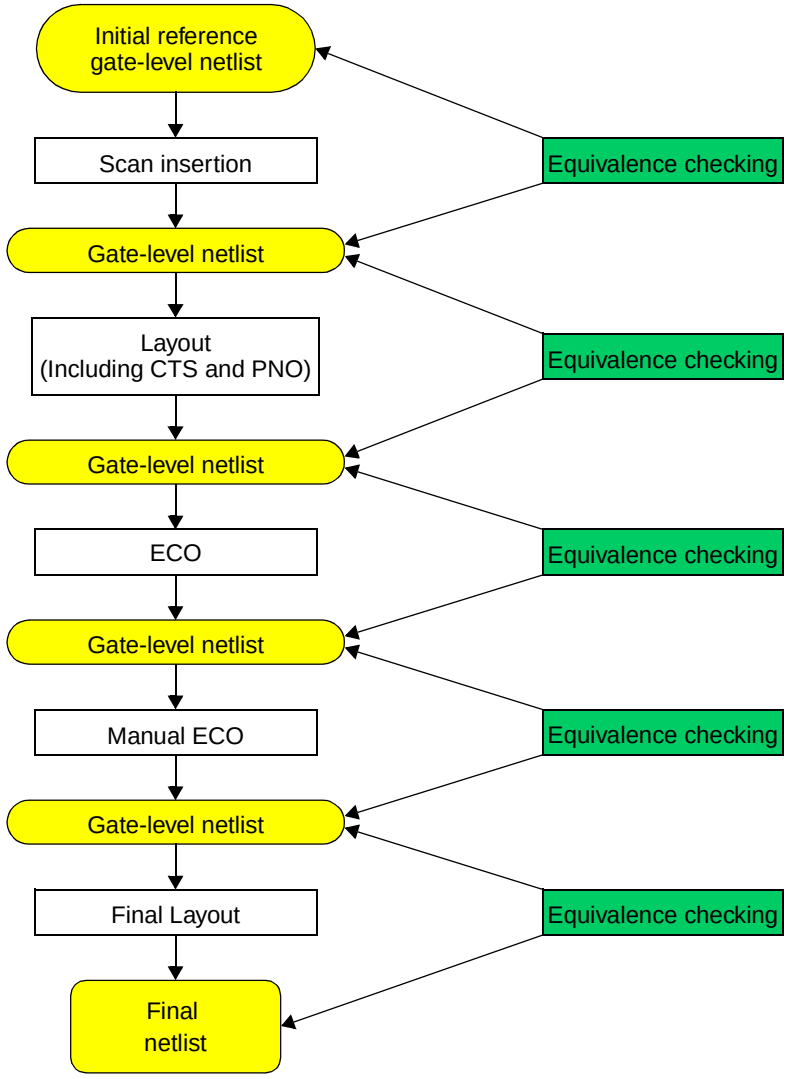
Logic equivalence checking is used to prove whether two given design descriptions are identical in functionality. Typical applications of gate-level equivalence checking in a typical design flow are shown in Figure 1–3.

The gate-level netlist can be in either VHDL or Verilog, but the library used is always in Verilog. Toshiba supports only the library in Verilog, which can be used with gate-level netlist in both VHDL and Verilog.

The detailed flow for the equivalence checking box is given in Figure 1–2. In general, the customer or the Toshiba ASIC Design Center engineer uses FV to verify designs that have been modified by hand or by commercial or in-house software. The initial reference netlist is the synthesized netlist from a synthesis tool.

As described in "What is Formal Verification?" on page 1, STA, FV, and logic simulations are three main components in the FV flow. FV can only point out the differences between two designs. It is not a replacement for a logic simulator, and it does not understand timing. Functionality for a reference and all new logic design must be thoroughly verified before a run of an equivalence check. For timing of a design, verify it with STA or logic simulation which takes delays into account.

Figure 1–3 Typical Applications of Gate-Level Equivalence Checking



Notes on Logic Equivalence Checking

When You Should Run an Equivalence Check

If an equivalence check is to be used in the ASIC design flow, it must be used to verify all versions of the design netlist, starting with the initial reference gate-level netlist (the first functionally correct, synthesized gate-level netlist). Use FV for each stage of netlist transformation. The golden rule of FV is “compare early and compare often.”

Mapping of Primary Input Pins

For the two designs to be compared, all primary inputs must be the same or pre-mapped, including those that are result of excised models. If a design contains extra primary inputs, you can constrain them to a value in order to disable the extra logic to which they are connected.

Differences in Signal Names

FV allows pairs of designs to include some different key point names or signal names. However, if the comparison takes too long or cannot be completed because there are too many differences in signal names, then provide additional mapping information by hand to help the tool with name mapping.

Designs That Can Not Be Handled by Equivalence Checking

FV does not consider timing. Timing-dependent circuits (combinational circuits containing feedback loops, circuits containing bidirectional signal conflicts and floats, etc.) must be checked using other methods, such as static timing analysis (STA) or timing simulation.

Since comparisons are done on the basis of logic cones, FV in principle cannot compare designs with state machine re-encoding or pipeline re-timing because the number of key points or the logic cones themselves might differ between the two designs. The method to handle them varies between tools. For details, see Chapter 2 and later, or the manual for individual tool.

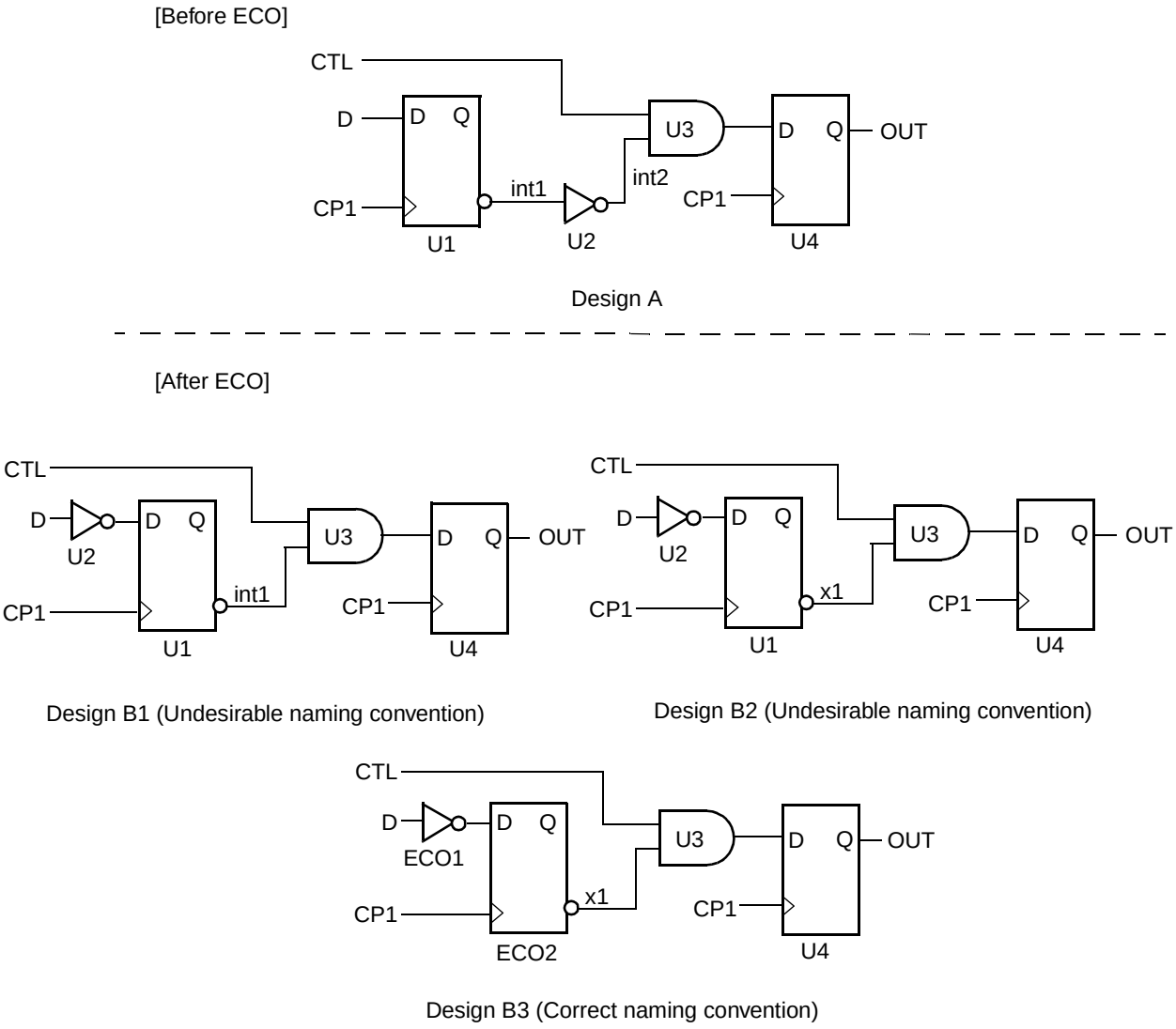
Notes on Manual ECO

When performing manual ECO, use unique signal and instance names for the designs.

An example is shown in Figure 1–4. Design A is the reference design, and designs B1, B2, and B3 are the revised designs after manual ECO. Design A has two key points, `int1` and `OUT`. The polarity of `int1` is inverted in the design after ECO. In design B1, the logic is changed, but the names of key points and instances all remain the same. In B2, `int1` is re-named to `x1`, but the instance name remains the same. In B3, the ECO guideline stated above has been followed because all new and modified logic uses unique signal and instance names.

When B1 and B2 are checked in the name-based comparison mode, FV maps identical instance names and therefore can produce false errors. In the nameless comparison mode, FV proves that B1 and B2 are equivalent. However, we recommend to follow ECO guideline stated above because mapping in the name-based mode takes less time.

Figure 1–4 ECO Guideline



Constraints

You can use FV most effectively when you understand the designs being compared and set correct constraints on them. When two designs are different due to DFT and gated clock, a lot of time can be saved if you can constrain the inputs in such a way that the difference is removed. For example, suppose the flip-flops in the reference design are driven by a primary input clock pin, and the same flip-flops in the implementation design are driven by a gated input clock pin that is controlled by an enable pin. Because the clock is part of the symbolic logic equation of the flip-flop, all of the flip-flops are considered to be different between two designs by FV. A very long run time can result while FV proves that every logic cone is different. To save time, it is highly recommended that you disable the enable control pin so that the clock logic between the two designs remains the same. In this case, you must be aware that FV does not verify the design including gated clocks.

Comparison of Two Synthesized Versions

You must also be aware of the fact that even if two RTL designs have been proven equivalent, it does not necessarily follow that the two synthesized versions of a design are also equivalent. This is because the optimization result of don’t-care conditions between two RTL designs might differ.

CHAPTER 2 Formality Methodology

This chapter describes logic equivalence check with Synopsys' Fomality and the requirements to run a logic equivalence check. For the terms of Formality used in this chapter, see *Formality User Guide* or *Formality Reference Manual*.

What Is Formality?

Synopsys' Formality is a logic equivalence verification tool. Formality reads two designs in RTL or netlist form and converts them to internal symbolic representations. Formality verifies them for equivalence and reports whether they are equivalent.

Notes on Running Formality

Dual-clocked Scan

When a design is verified with dual-clocked scan flip-flops contained in Verilog simulation library provided by Toshiba, an error may occurs where there is no real chance of an error. Toshiba has qualified a special library containing models of

these cells for use by Formality, with which errors do not occur. Running the Toshiba Formality command `read_toshlib` automatically sets to use the special library, which is for use by Formality, for the dual-clocked scan flip-flops.

Combinational Logic Containing a Feedback Loop

Because Formality can not verify a combinational logic containing a feedback loop with which signal values are retained, you need to insert a pseudo-black-box into a loop and set a key point in it to cut a loop. The following command must be entered.

```
create_cutpoint_blackbox black_box_name object_ID
                         -type object_ID_type
```

where,

black_box_name Specifies the name to give a part of a design to be turned into a black box.

object_ID Specifies the full pathname in a container for a signal where a keypoint is to be set.

The following shows the syntax to specify the *object_ID*.

container_ID:*/design_ID/object_name*

object_ID_type Specifies `net` or `port` to *object_ID* where a keypoint is to be set.

Example:

```
create_cutpoint_blackbox mybb A:/WORK/test1/wire3
                         -type net
```

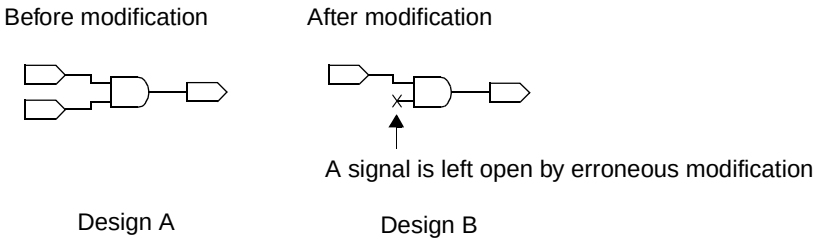
Formality checks only design configuration and static equivalence of cells' connection. Formality can not prove dynamic behavior including signal propagation delays. Therefore, a combinational logic containing a feed back loop requires logic simulation which takes delays into account together with equivalence checking.

Setting Reference and Implementation Designs

Formality differentiates strictly between the reference and implementation designs. If you confuse two designs, Formality might not detect mismatches.

The following shows why Formality can not detect mismatches with a sample design in Figure 2–1. In this example, a signal is erroneously left open by design modification.

Figure 2–1 Erroneously Modified Signal



Formality has the following two interpretation modes for the X value in the *reference* design.

- consistency mode (default)

  Like interpretation by a logic synthesis tool, the X value is converted to a don’t care, which means it can be compared to 0 or 1.

- equality mode

  Like interpretation by a logic simulator, the X value is converted to an unknown, which means it can not be compared to 0 or 1.

Because the signal in the design B is left open by design modification, the output of the design B becomes an X. When you set the design A to the reference design, and the design B to the implementation design, Formality in the consistency mode (default) reports a mismatch. This is because 0 or 1 of the output signal in the design A is compared to the X of the output in the design B. However, when you set the design B to the reference design, and the design A to the implementation design, Formality report a match because the output X in the design B is converted to a don’t care, which can be compared to both 0 and 1 in the design A.

In the equality mode, Formality can detect mismatches whichever design you set to the reference design. The X value is interpreted to an unknown during a gate-level equivalence check. Therefore, we recommend to run Formality in the equality mode to check equivalence between gate-level models. Enter the following command to set the equality mode.

```
set verification_passing_mode equality
```

The other method to resolve the problem that Formality can not detect mismatches is to set an undriven signal to the Z value. Set the following to set an undriven signal to the Z value.

```
set verification_set_undriven_signals z
```

(By default, the undriven signal value is X.)

Checking Designs Containing Asynchronous Signals

When running Formality on the design containing registers with asynchronous control signals, you may be required to set following variable.

```
set verification_asynch_bypass true
```

Note: When checking the design containing dual-clocked scan flip-flops with asynchronous set/reset (FD3SF), you must set true to the above variable.

Checking Designs with Identical Hierarchy (by Formality 2000.11-FM1.0-SP2)

When checking two designs containing the identical design hierarchy, Formality 2000.11-FM1.0-SP2 runs by the bottom-up methodology by default, that means Fomality checks the bottom level first. When Formality proves equivalence on a level, the level is turned into a black box upon an equivalence check on the upper level, that can reduce checking time. When checking an entire design, Formality runs by the top-down methodology.

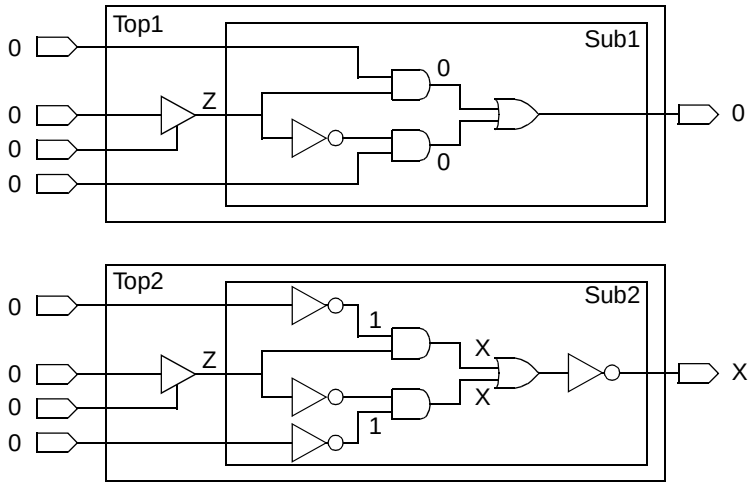
For the design containing 3-state buffers, you must be aware that checking results may have differences between the bottom-up and top-down methodologies.

The following shows why differences are produced with a sample design in Figure 2–2 which contains a selector circuit including AND and OR cells.

The equivalence checking result of the sample design is as follows.

Bottom-up methodology: **match**
Top-down methodology: **mismatch**

Figure 2–2 Design Containing a Selector Circuit



Running a top-down equivalence check on the entire designs of Top1 and Top2 results in a mismatch between them. A mismatch is produced since Xs propagate to the output in Top2 when the 3-state buffer positioned before a selector circuit is in the high-impedance state.

In case of a bottom-up hierarchical equivalence check, equivalence checking is performed first between the selector circuits Sub1 and Sub2 in Top1 and Top2. Because all inputs to Sub1 and Sub2 are treated as primary output signals, equivalence checking results in a match between Sub1 and Sub2. Next, run an equivalence check on Top1 and Top2 by turning both of selector circuits into block boxes, that results in a match between them.

The above example shows that equivalence checking on the entire design containing a 3-state buffer may result in a mismatch though hierarchical equivalence checking results in a match. Therefore, you must be aware that you may miss mismatches which can be produced during a logic simulation on an entire design, if you check functional equivalency of a circuit containing a 3-state buffer with a run of a only hierarchical equivalence check.

Enter the following command to run an equivalence check by the top-down methodology.

```
set_parameters -flatten object_ID
```

Note: Since by default Formality 2001.08 or later runs an equivalence check by the top-down methodology, the set_parameters -flatten option is not required.

Turning Modules into Black Boxes

Since Formality can not check RAMs, ROMs, PLLs, IP cores (embedded DRAMs, microprocessor, etc.), and analog blocks, they are handled as black boxes. In this case, when net information of the module is described in the dummy module file, inputs to the black box module become primary outputs of the top-level main block, and outputs become primary inputs, that assures net connection to the black boxes.

When handling a module as a black box, you need to create a dummy module file containing only input and output declarations statements of a module. If input or output pins to be required are not declared, you must be aware that the incorrect net to the black box can not be detected because these pins are handled as bidirectional pins.

Notes on Using Formality up to 2001.08

Formality automatically regards all undefined designs, e.g. designs of which cell library does not exist, as black boxes in which all I/O pins are bidirectional pins. Therefore, if Formality can not read a design correctly, it can not check the design correctly because the design is handled as a black box. In this case, bear in mind that two designs are not always equivalent even when Formality reports they are equivalent.

You must carefully check the log obtained by a run of Formality in order to verify a design correctly. When the following message is issued, a black box has been created.

```
Warning: 1 (1) black-box references found in reference
(implementation) design; see formality.log for list (FM-182)
```

You can check whether a black box has been created by running `report_libraries`. When a black box has been created, the library named FM_BBOX is created in a container. The designs contained in the FM_BBOX library are handled as block boxes by Formality. Check the contents of the library. If the contents of the library includes mistakes, you have Formality re-read a design correctly.

When you use Toshiba megacells or set black boxes, see "Handling Megacells / IP Cores" on page 23.

Notes on Using Formality 2002.03 and 2002.05

When Formality finds an undefined design, it halts processing with a link error. When an error message is issued, make Formality read a design and library correctly over again since an undefined design exists. An example of an error message is given below.

```
Error: Design 'EAS1D080008A' is not defined but cell
'/WORK/DESIGN/EAS1D080008A' references it. (FM-234)
```

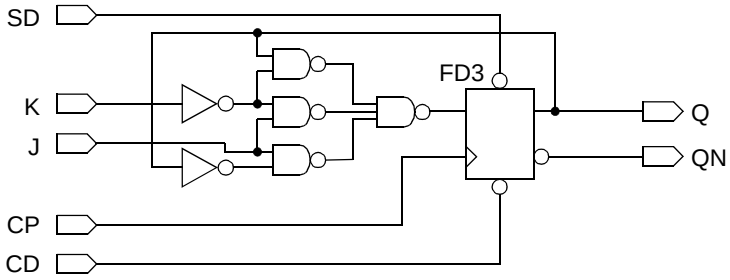
Notes On JK-F/F (JK3) With Asynchronous Set/Reset

Formality constraints a register cell in the library to be a single cell, one key point. Because Toshiba JK-F/F (JK3) can not represent behavior correctly with one cell and one key point, it might not be checked correctly. Care is required when you run an equivalence check on the design containing JK3 with Formality.

When comparing a combinational design containing a flip-flop with asynchronous set/reset (FD3) in Figure 2–3 to JK3, Formality evaluates two designs are equivalent. However, the static analysis shows that behavior differs between two designs. After CD=SD=0 is changed to CD=SD=1, for example, QN is inverted from Q in case of J=K=0 on the rising edge of CP in Figure 2–3. Whereas, Q=QN=0 is obtained in case of JK-F/F.

Note: This is the result obtained with a static analysis, which is not always identical to the behavior during simulation. In the above case, a run of simulation results in a mismatch.

Figure 2–3 Design Containing FD3



Design Flow for Potential Applications

Formality supports all applications listed in this section. For more detailed information, refer to *Formality User Guide* or *Formality Reference Manual*.

Clock Tree Insertion / Post-Layout Netlist Optimization

An example of post-layout netlist optimization is the in-place optimization offered by the Synopsys Design Compiler or Floorplan Manager. Formality can handle this type of netlist transformation and clock tree insertion.

Scan Insertion

Formality ensures functional equivalence before and after scan insertion in the normal operation mode. To verify designs before and after scan insertion, you must constrain the scan control signals so that the scan logic is disabled. The correct syntax for this is given below.

set_constant *object_ID state* **-type** *object_ID_type*

where,

object_ID Specifies the full path name within a container to the signal to which a constraint is set. The syntax to specify the object ID is as follow.

container_ID:/*design_ID*/*object_name*

When you omit *container_ID* and *design_ID*, *container_ID* and *design_ID* which are set currently are applied.

state Specifies `0` or `1`.

object_ID_type
Specifies `net` or `port` to the object ID to which a constraint is set.

Example:

```
set_constant B:/WORK/MBIST_EN 0 -type port
```

When a scan output pin exists in one design and does not exist in the other design, excise the scan output pin from a design by setting the followings.

remove_compare_point *object_ID* **-type** *object_ID_type*

Example:

```
remove_compare_point A:/WORK/MBIST_ON -type port
```

Note: Note is required to check a design which uses a register containing asynchronous control signal (FD3SF). For the detailed descriptions, see "Checking Designs Containing Asynchronous Signals" on page 14.

BIST Insertion

Formality ensures functional equivalence before and after BIST insertion in the normal operation mode. To successfully verify two designs before and after BIST insertion, you must excise both the megacells and the collars surrounding the megacells. The BIST inputs should be statically constrained so that the test logic is disabled, and the BIST outputs should be excised from the design with `remove_compare_point`. When there are designs controlled with internal signals of BIST, which is excised during a check, you need to constrain the internal signals so that these designs can behave in the normal operation mode.

Gate-Eating

Formality optimizes out redundant logic gates. Therefore you can verify two designs before and after gate-eating.

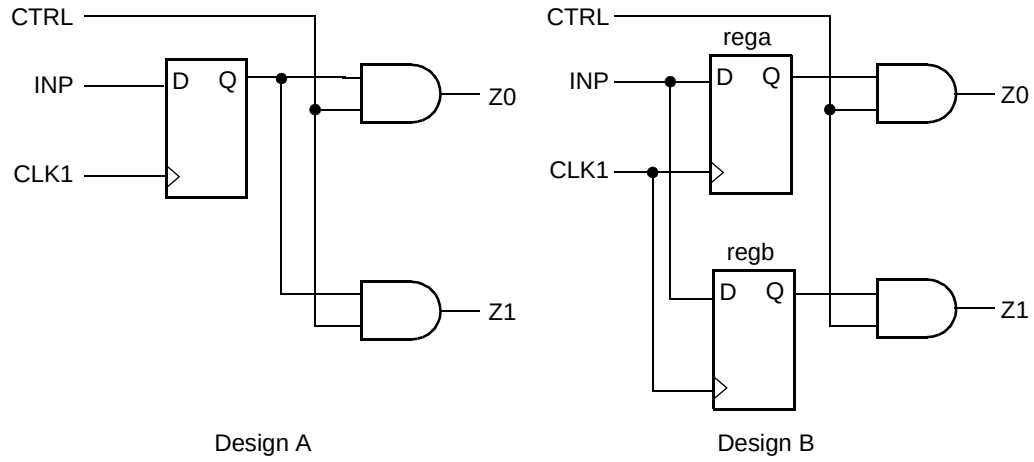
Duplicated Logic

Combinational duplicated gates can be handled automatically. For duplicated flip-flops and latches as shown in Figure 2–4, you must set the parameter to these gates since the number of key points increases. In order to correctly compare design A with design B, the following parameter must be set.

set_parameters -retimed *design_ID*

The above parameter is used to check the design before and after retiming. Because `-retime` is the option to perform verification by taking replacement of registers into account, it can be specified in case of duplicated logic as well as retiming.

Figure 2–4 Comparing Designs with Duplicated Registers for High Drive



Scan Chain Reordering

After scan chain reordering, you need to disable the scan mode. If this guideline is not followed, then the scan outputs of all reordered scan flip-flops are reported as mismatches, and the design before and after scan chain reordering can not be compared.

Flat vs. Hierarchical Comparisons (by Formality 2000.11-FM1.0-SP2)

When you compare the design before and after flattening by Formality 2000.11-FM1.0-SP2, set the following parameter to the design before flattening in order to increase in verification performance.

```
set_parameters -flatten design_ID
```

Note: When a design is flattened, checking performance is decreased largely if hierarchical names are not maintained in the signal names of the flattened design.

Checking Re-timed Designs

Formality can handle a re-timed design on which the `balance_registers` and `optimize_registers` commands of Design Compiler have been executed. You need to set the following parameter.

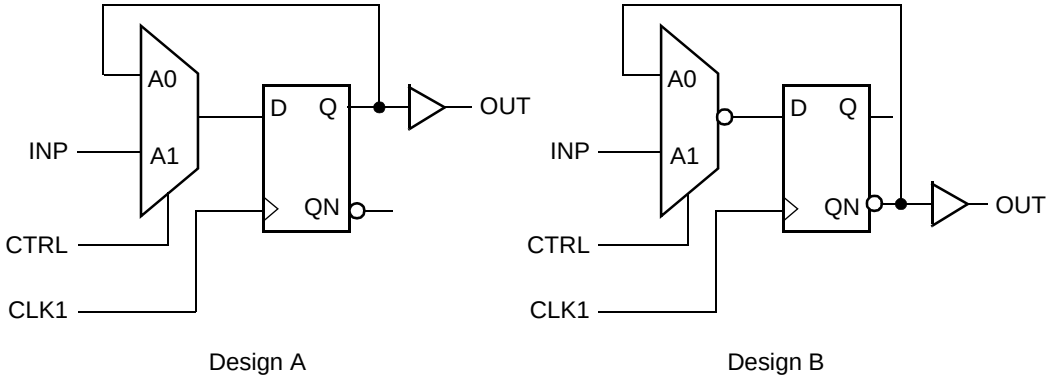
```
set_parameters -retimed design_ID
```

When a re-timed design is created by manual, the possibility of running an equivalence check depends on the contents of re-timed design. Formality can not compare designs with pipeline re-timing across logic cones.

In case that a design change involves polarity inversion, which is performed upon output from a register and input to a register, set the following parameter.

```
set verification_inversion_push true
```

Figure 2–5 Design Change Involving Polarity Inversion



Checking Designs With State Machine Re-encoding

Because comparisons are done on the basis of logic cones, Formality cannot compare designs with state machine re-encoding because the number of key points or the logic cones themselves might differ between the two designs.

However, comparison might be possible when the number of states and state name are identical in the two designs. In this case, you can set the state vector and state encoding with the following two methods.

- Using the FSM state file

  In case of enormous number of states, we recommend to create the FSM state file. Figure 2–6 shows an example of the FSM state file.

Figure 2–6 Sample FSM State File

```
****************************************
Report : fsm
Design : fsm_m
Version: 2001.08
Date   : Thu Jan 24 16:43:47 2002
****************************************
Clock              : clk         Sense: rising_edge
```

```
Asynchronous Reset : Unspecified
Encoding Bit Length: 2
Encoding style     : Unspecified

State Vector: { s_reg[1] s_reg[0] }

State Encodings and Order:

S0         : 00
S1         : 01
S2         : 10
S3         : 11
```

The FSM state file has the format same as the one reported with the `report_fsm` command of Synopsys Design Compiler. The FSM state file contains names of flip-flops consisting of state vectors, state names and encoding. The state name must be identical between the reference and implementation designs so that Formality can check designs with state machine re-encoding.

Enter the following command to have Formality read the FSM state file

read_fsm *file_name design_ID*

♦ Using the Formality command

The Formality command can be used to set the state vector and encoding.

Enter the following command to set the names of flip-flops consisting of FSM state vectors.

set_fsm_state_vector {*F/F_list*} *design_ID*

Example:

```
set_fsm_state_vector { s_reg[1] s_reg[0]}
                     A:/WORK/fsm_m
```

Enter the following to set the state name and encoding.

set_fsm_encoding {*encoding_list*} *design_ID*

Example:

```
set_fsm_encoding {"S0=2#00" "S1=2#01" "S2=2#10"}
                 A:/WORK/fsm_m
```

Handling Megacells / IP Cores

Since Formality can not check RAMs, ROMs, PLLs, IP cores (embedded DRAMs, microprocessor, etc.), and analog blocks, they are handled as black boxes. You can turn Toshiba megacells supported as standard packages into black boxes by generating megacell models for FV with Toshiba module generator (`MDLGEN`) and having Formality read the megacell models with the Toshiba Formality command `read_toshmega`. For the `read_toshmega` command, see Chapter 5, "Toshiba Formality Commands". For the `MDLGEN` command, see *Sign-Off System x.x.x Command Reference*.

The following shows the procedures to check designs by excising Toshiba megacells from the design.

1. **Set up the environment to run `MDLGEN`.**

   ```
   setenv TOSH_ROOT "installation_directory"
   set TOOLS_PATH="${TOSH_ROOT}/toshiba_common/bin"
   set path=($TOOLS_PATH $path)
   ```

2. **Run `MDLGEN`.**

   ```
   mdlgen -tech technology -target fv -mg MGDATA_filename
   ```

3. **Run the `read_toshmega` command in the Formality fm_shell environment to have Formality read the megacell models for FV.**

   Example:

   ```
   fm_shell> read_toshmega ./*.ver \
             -container container_ID
   ```

   The following shows the command to have Formality read multiple files.

   Example:

   ```
   fm_shell> read_toshmega ./*.ver \
             SPECIAL_tsbdummy.v \
             -container container_ID
   ```

4. **When a Toshiba megacell is turned into a black box and net information is read correctly, the following warning message is issued.**

   ```
   Warning: Design A:/MEGA/EAS1A040004A is a black box and
   there are cells referencing it(FM-160)
   ```

If you get the following warning message together with the above one, Formality could not read the net information correctly. Check the dummy module file, and rerun Formality from the stage 3.

```
Warning: 16 (16) black-box pins of unknown direction
found in reference (implementation) design; see
formality.log for list (FM-230)
```

Note: When you want to turn designs other than Toshiba megacells into black boxes, create the dummy module files for the designs separately, or turn the designs into black boxes with the following command in the fm_shell environment.

Formality up to 2001.08:	remove_design
Formality 2002.03 or 2002.05:	set_black_box

CHAPTER 3 Getting Started

The chapter describes how to install and configure the Toshiba Formality Design Kit.

System Requirements

See the Formality Installation Guide for the hardware and operating system requirements.

Installation

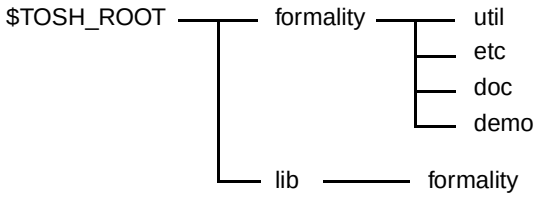
See the Formality Installation Guide for the Formality installation instructions.

Contact your local Toshiba design center to find out how to receive and install the Toshiba Formality Design Kit.

Directory Hierarchy

After installing the Formality Design Kit, you should see the following hierarchy.

Figure 3–1 Formality Design Kit Directory Hierarchy



- The `etc` directory contains the setup file and the `.synopsys_fm.setup` file which is used to set Toshiba Formality commands.
- The `lib` directory contains Toshiba Formality library.
- The `util` directory contains the program that is the Toshiba command. For the details of how to run the Toshiba Formality commands, see Chapter 4, "Checking Equivalence".
- The `demo` directory contains the sample design which is used in Chapter 4, and execution shell.

Environment Setup

Do the following before running Formality.

1. **Modify $TOSH_ROOT/formality/etc/cshrc.fm file for your environment as follow:**
   - Change `FM_HOME` to your Synopsys' Formality tools installation directory.
   - Change `LM_LICENSE_FILE` to your local Formality license file.
   - Change `TOSH_ROOT` to your local setup.
2. **Source the modified `$TOSH_ROOT/formality/etc/cshrc.fm` file.**
3. **Copy the `$TOSH_ROOT/formality/etc/.synopsys_fm.setup` file to the run directory.**

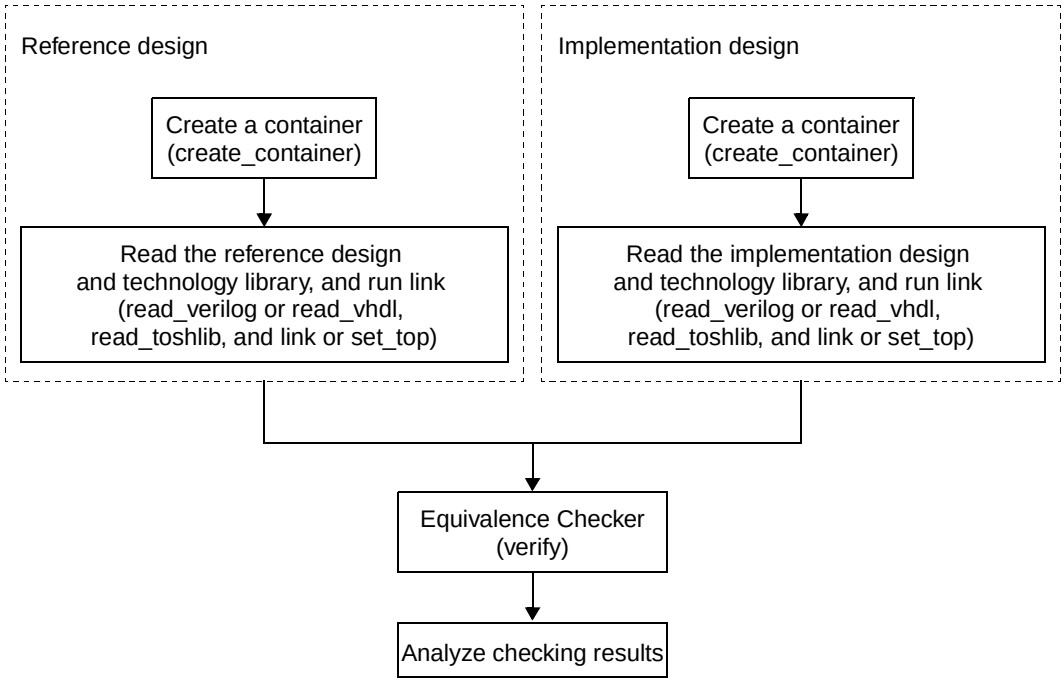
CHAPTER 4 Checking Equivalence

This chapter describes how to run Formality to check the functional equivalence of two design. For more detailed information, see *Formality User Guide* and *Formality Reference Manual*.

Formality Flow

Formality creates the database memory space called a container to store the information on reference and implementation designs. After having read the required information on designs and libraries into containers, Formality checks functional equivalence of two designs. Equivalence checking results obtained with Formality are displayed. Figure 4–1 shows the Formality design flow.

Figure 4–1 Formality Design Flow



Note: For the link command, "link" is used with Formality up to 2001.08, and "set_top" is used with Formality 2002.03 or 2002.05.

Running an Equivalence Check

This section describes the Formality basic functional equivalence check procedures with a sample design.

Using Formality up to 2001.08

1. **Set a run script of equivalence check.**

   Figure 4–2 shows a run script of logical equivalence check for a sample design. A run script of a sample design is used to run an equivalence check of two netlists, `sample0.v` and `sample1.v` which use TC240C technology.

Figure 4–2 Sample Run Script for Equivalence Checking

```
read_toshlib -tech TC240C

create_container ref
read_verilog -container ref -netlist ./sample0_240.v
link ref:/WORK/sample

create_container imp
read_verilog -container imp -netlist ./sample1_240.v
link imp:/WORK/sample

set verification_passing_mode equality
set_reference_design ref:/WORK/sample
set_implementation_design imp:/WORK/sample
verify
exit
```

2. Set the `.synopsys_fm.setup` file.

You need to add setting to your `.synopsys_fm.setup` file so that Toshiba Formality commands can be used. Add setting by referencing the sample file which resides in the `$TOSH_ROOT/formality/etc` directory. Figure 4–3 shows setting (sample file) which is added to your `.synopsys_fm.setup` file.

Figure 4–3 Additional Setting to .synopsys_fm.setup file (Sample File)

```
# set up TOSH_ROOT
set TOSH_ROOT $env(TOSH_ROOT) ;
# read toshiba utility
source $TOSH_ROOT/formality/util/tosh_util.fms
```

3. Read the technology Library with `read_toshlib`.

Formality can read the library used for Toshiba Verilog simulation. For dual-clocked scan flip-flops, Formality is required to read the RTL models contained in the special library for use by Formality.

We provide the `read_toshlib` command which automatically sets to use the special library for use by Formality. When you run `read_toshlib`, the technology library is read in the appropriate format to run Formality. Enter the following command to run `read_toshlib`.

```
read_toshlib -tech target_technology_name
             -container container_ID
```

When you omit `-container`, the technology library is read as the share library which is used with all containers. If both designs use the same technology, Formality is required to read a technology library once, not twice when the technology library is read as the share library for both designs by omitting

`-container`. If the technologies are different, use `-container` to store the technology libraries in the separate containers. `-container` can be abbreviated as `-c`.

4. **Create containers with `create_container`.**

   Containers store the information on designs and libraries used with Formality. Therefore, you need to create containers before Formality reads netlists. For typical equivalence checking, create containers for the reference and implementation designs separately. Enter the following command to create a container.

   ```
   create_container container_ID
   ```

5. **Read netlists with `read_verilog` or `read_vhdl`.**

   Formality can read netlists in Verilog, VHDL, EDIF, and Synopsys' database (`.db`) formats. Enter the following command to read netlist in individual format.

   ```
   read_verilog Verilog_netlist -container container_ID
   -lib library_ID

   read_vhdl VHDL_netlist -container container_ID
   -lib library_ID

   read_edif EDIF_netlist -container container_ID
   -lib library_ID

   read_db DB_netlist -container container_ID
   -lib library_ID
   ```

   Specify `-lib` to the name of a library in the container, where the netlist is to be stored. When you omit `-lib`, netlists are read into the default library named `WORK`. ***For better performance in Verilog gate-level netlist read, use the `-netlist` option with the above command.***

6. **Link to the information in a container with `link`.**

   Link to the information in a container so that Formality can use the required data in it. You can check whether all required data is stored in a container by running `link`. When descriptions in RTL form have been read, elaboration is performed with `link`. Enter the following command to run `link`.

   ```
   link design_ID
   ```

Formality checks whether all required data resides by running `link` automatically during an equivalence check. However, we recommend running `link` separately so that you can identify the source of a mismatch, which is either `link` or verification (`verify`). When Formality finds that all the data required for an equivalence check does not reside upon a run of `link`, it automatically turns undefined part of a design into a black box by issuing the following message. Check the data included in a container, and store the required data correctly in it.

```
Warning: 16 black-box designs were created for missing
references. (FM-064)
```

When a library named FM_BBOX has been generated in a container, Formality handles all designs stored in FM_BBOX as black boxes. Check the contents in the library, and modify them if required.

7. Set up checking environment.

Formality contains several parameters to set up checking environment. When running an equivalence check with Formality, you need to understand the characteristics of the reference design as well as the implementation design, and set the parameters correctly. For the notes on equivalence checking with Formality, required parameters, and how to set up checking environment, see Chapter 2, "Formality Methodology" or *Formality Reference Manual*.

8. Run an equivalence check with `set_reference_design`, `set_implementation_design` and `verify`.

When all data for the reference and implementation designs is prepared, set the reference and implementation designs, then run an equivalence check. When you set the reference and implementation designs, specify the top level module name (*`design_ID`*). Enter the following command to set the reference design.

```
set_reference_design design_ID
```

Enter the following command to set the implementation design.

```
set_implementation_design design_ID
```

Enter the following command to run an equivalence check.

```
verify
```

> **Note:** Formality maps key points by name-based mapping method. When two designs contain different instance names, mapping is performed based on the names of nets driving output signals. When you use nameless mapping method, set false to name_match_based_on_nets.

9. Classify and analyze the equivalence checking results.

Formality displays the checking results. When an equivalence check has been completed, one of the following comments appears.

- `SUCCEEDED`

  Formality proves functional equivalence of the reference and implementation designs. Equivalence checking has been performed on all compare points.

- `FAILED`

  Formality does not prove functional equivalence of the reference and implementation designs. Function is not equivalent on one or more compare points, or compare points are not paired between two designs.

- `INCONCLUSIVE`

  Formality can not check functional equivalence of the reference and implementation designs. This comment is issued when equivalence can not be checked on the mapped compare points in such cases as a combinational logic containing a feedback loop. This comment is also issued when checking is aborted, or checking exceeds the time limit you set.

10. Save the log file.

Formality generates two types of log files.

- `formality.log`

  This file contains the detailed information on the warning and information messages which were not displayed.

- `fm_shell_command.log`

  This file contains logs of command runs during Formality. You can reproduce the same checking operation by running `source` with this file.

Since the comments, that are `SUCCEEDED`, `FAILED` and `INCONCLUSIVE`, are not saved in a log file, use the UNIX `tee` command to save them. The following is an example of the command.

```
fm_shell|tee fm.log
```

Using Formality 2002.03 or 2002.05

This section describes functional equivalence check procedures with Formality 2002.03 or 2002.05. The following discusses procedures with sample commands, and differences from Formality up to 2001.08.

1. **Set a run script of equivalence check.**

   Figure 4–4 shows a run script of logical equivalence check for a sample design.

   Figure 4–4 Sample Run Script for Equivalence Checking

```
create_container ref
read_toshlib -tech TC240C -container ref
read_verilog -container ref ./sample0_240.v
set_top ref:/WORK/sample
set_reference_design ref:/WORK/sample

create_container imp
read_toshlib -tech TC240C -container imp
read_verilog -container imp ./sample1_240.v
set_top imp:/WORK/sample
set_implementation_design imp:/WORK/sample

set verification_passing_mode equality

verify
exit
```

2. **Set the `.synopsys_fm.setup` file.**

   Same procedures as Formality up to 2001.08 shown on page 29.

3. **Create a container for the reference design with `create_container`.**

   Enter the following command to create a container for the reference design.

```
create_container ref
```

4. **Read the technology Library for the reference design with** `read_toshlib`**.**

   Formality 2002.03 or 2002.05 needs to read the technology library for every container.

   ```
   read_toshlib -tech TC240C -container ref
   ```

   If `-container` is not specified, Formality halts processing by issuing the following message.

   ```
   ERROR: You must specify container name for 2002.03 or
   newer.
   ```

5. **Read the netlist of the reference design with** `read_verilog` **or** `read_vhdl`**.**

   Formality reads the netlist of the reference design. The following shows an example of the `read_verilog` command to read the Verilog netlist.

   ```
   read_verilog sample0_240.v -container ref
   ```

   Formality 2002.03 or 2002.05 automatically discriminates between gate-level and RTL netlists, and sets the `-netlist` option when the gate-level netlist is applied.

6. **Link to the netlist of the reference design with** `set_top`**.**

   `set_top` is the link command used with Formality 2002.03 or 2002.05.

   ```
   set_top ref:/WORK/sample
   ```

   When Formality finds an undefined design, it halts processing by issuing the following message. Check the error message and set the required data in the container.

   ```
   Error: Design 'xxxx' is not defined but cell
   '/WORK/xxxx/xxxxx' references it. (FM-234)
   ```

7. **Set the reference design.**

   Set the reference design with the `set_reference_design` command.

   ```
   set_reference_design ref:/WORK/sample
   ```

8. **Create the container for the implementation design with** `create_container`**.**

```
create_container imp
```

9. **Read the technology library for the implementation design with** `read_toshlib`**.**

   Formality 2002.03 or 2002.05 needs to read the technology library for every container.

```
read_toshlib -tech TC240C -container imp
```

   When `-container` is not specified, Formality halts processing by issuing the following error message.

```
ERROR: You must specify container name for 2002.03 or
newer.
```

10. **Read the netlist for the implementation design with** `read_verilog` **or** `read_vhdl`**.**

    The following shows an example of the `read_verilog` command to read the Verilog netlist.

```
read_verilog sample1_240.v -container imp
```

    Formality 2002.03 or 2002.05 automatically discriminates between gate-level and RTL netlists, and sets the `-netlist` option when the gate-level netlist is applied.

11. **Link to the netlist of implementation design with** `set_top`**.**

    `set_top` is the link command used with Formality 2002.03 or 2002.05.

```
set_top imp:/WORK/sample
```

    When Formality finds an undefined design, it halts processing by issuing the following message. Check the error message and set the required data in the container.

```
Error: Design 'xxxx' is not defined but cell
'/WORK/xxxx/xxxxx' references it. (FM-234)
```

12. **Set the implementation design.**

    Set the implementation design with the `set_implementation_design` command.

    ```
    set_implementation_design imp:/WORK/sample
    ```

13. **Set up checking environment.**

    Same procedures as Formality up to 2001.08. See "Set up checking environment." on page 31.

14. **Run an equivalence check with `verify`.**

    Enter the following command to run an equivalence check.

    ```
    verify
    ```

    > **Note:** Formality maps key points by name-based mapping method. When two designs contain different instance names, mapping is performed based on the names of nets driving output signals. When you use nameless mapping method, set false to name_match_based_on_nets.

15. **Classify and analyze the equivalence checking results.**

    Same procedures as Formality up to 2001.08. See "Classify and analyze the equivalence checking results." on page 32.

16. **Save the log file.**

    Same procedures as Formality up to 2001.08. See "Save the log file." on page 32.

CHAPTER 5 Toshiba Formality Commands

The chapter describes Toshiba Formality commands, `read_toshlib` and `read_toshmega`.

read_toshlib (fm_shell command)

The `read_toshlib` command has Formality read the Toshiba Formality library. Upon reading the Toshiba Formality library, Formality reads the Verilog simulation library provided by Toshiba. For dual-clocked scan flip-flops causing an error where there is no real chance of an error, Formality can read RTL models automatically which are contained in the special library for use by Formality. When you use `read_toshlib`, see `read_simulation_library` and `read_verilog`. The following shows the `read_toshlib` command syntax.

```
read_toshlib -tech target_technology_name
             -container container_ID
```

where,

`-tech` *`target_technology_name`*
Specifies the technology name of a design.

`-container` *`container_ID`*
Specifies the name of a container where a library is to be stored. The abbreviation is `-c`.

Using Formality up to 2001.08

When you omit `-container`, the library is stored in the container as the share library which can be used with both the reference and implementation designs. If both designs use the same technology, Formality is required to read a library once, not twice, when it is read as the share library for both designs. If the technologies are different, use `-container` to store the technology libraries in the separate containers.

Using Formality 2002.03 or 2002.05

The container name is required. Even when two designs built in the same technology are compared, the same technology library must be read into individual container.

If `-container` is not specified, Formality halts processing by issuing the following message.

```
ERROR: You must specify container name for 2002.03 or
newer.
```

After Formality has read the Toshiba Verilog simulation library, a library with the specified technology name is created in the container, and the library cell information is stored in that library.

The following shows an example of the `read_toshlib` command.

```
fm_shell> read_toshlib -tech TC240C -container ref
```

> **Note:** read_toshlib is the command to run in the fm_shell environment.

read_toshmega (fm_shell command)

The `read_toshmega` command has Formality read the megacell model files for FV generated with module generator (`MDLGERN`) or user-created dummy module files. The following shows the `make_toshmega` command syntax.

```
read_toshmega [-vh] filename
              -container container_ID
```

where,

-[vh] Specify this option only when user-created dummy module file is in VHDL format. When FV reads the Verilog-HDL dummy module file and megacall model for FV, this option is not required.

filename Specifies the name of the megacell model for FV or user-created dummy module file. A wildcard (*) can be included in the filename.

-container *container_ID*
Specifies the name of a container where the megacell model for FV or dummy module file is to be stored. The abbreviacion is -c.

Using Formality up to 2001.08

When you omit -container, the megacell model for FV or dummy module file is stored in the container as the share library which can be used with both the reference and implementation designs. After Formality has read the megacell model for FV or the dummy module file, the library named MEGACELL is created in the container, and the megacell connection information is stored in that library.

Using Formality 2002.03 or 2002.05

The container name is required. Read the megacell model for FV and dummy module file into individual container.

If -container is not specified, Formality halts processing by issuing the following message.

```
ERROR: You must specify container name for 2002.03 or
newer.
```

The following shows an example of the read_toshmega command.

```
fm_shell> read_toshmega EAS2A040004A_fv.ver -container ref
```

Note: read_toshmega is the command to run in the fm_shell environment.