PrimeTime®
Modeling
User Guide

Version C-2009.06, June 2009

SYNOPSYS®

Contents

What's New in This Release x

About This User Guide x

Customer Support. xii

1. Introduction to Static Timing Models

Timing Model Types 1-2

Generated Timing Models. 1-4

Interface Logic Modeling 1-4

Model Extraction 1-5

Other Types of Timing Models 1-6

Synopsys Technology Libraries 1-6

Quick Timing Models 1-7

2. Quick Timing Models

Quick Timing Model Design Flow 2-2

Quick Timing Model Parameters. 2-2

Setting Global Parameters. 2-3

Specifying Model Information 2-3

Computing Arc Delays 2-4

Creating a Quick Timing Model. 2-6

Creating the Model and Setting Global Parameters 2-7

Specifying Model Information 2-8

Viewing the New Model 2-9

Saving a Model 2-9
Converting STAMP Models to .lib Format 2-10
Saving a Model as .db Wrapper and Core. 2-10
Saving a Model as .db Library Cell 2-10

Using a Quick Timing Model in a Design 2-10

Quick Timing Model Command Summary 2-11

3. Interface Logic Models

Overview of Interface Logic Modeling. 3-2

Benefits, Flow, and Limitations of ILM 3-3

How ILMs Are Used 3-3

Generating an ILM 3-4

Generating ILMs Using the create_ilm Command 3-5

Specific Setups with ILM 3-6
High Fanout Ports 3-6
Latches 3-7

Recommended Options for the create_ilm Command 3-7
Model Validating an ILM 3-7
ILM Timing Matches Block-Level Netlist Timing 3-8
Scope Checking for ILM Within Chip-Level Design 3-10
Model Validating Issues and Resolutions 3-12
Modeling Issues 3-12
Scope Checking Issues 3-12

Using the ILM in a Top-Level Netlist 3-13

ILMs with PrimeTime SI 3-14
Generating PrimeTime SI ILM 3-14
Model Validating for PrimeTime SI ILM 3-16
Using a PrimeTime SI ILM in a Top-Level Netlist. 3-17

4. Extracted Timing Models

Model Extraction Overview 4-3

Extraction Requirements. 4-3

Extraction Process 4-4

Extracting Timing Paths 4-6
Boundary Nets 4-6
Internal Nets 4-6
Paths From Inputs to Registers 4-7
Paths From Inputs to Outputs 4-7
Paths From Registers to Outputs 4-7
Paths From Registers to Registers 4-8
Clock Paths 4-8
Interface Latch Structures 4-8
Minimum Pulse Width and Minimum Period 4-9
False Paths 4-10
Clock-Gating Checks 4-10
Back-Annotated Delays 4-10
Block Scope 4-11
Noise Characteristics 4-11
Other Extracted Information 4-12
Limitations of Model Extraction 4-14
Limitations of Extracted Models in Design Compiler 4-16
Limitations of Extracted Models in .lib Format 4-16
Extracted Model Types 4-16
Extraction Variables 4-17
Variable Setting Guidelines 4-18
Delay Table Generation 4-18
Preparing for Extraction 4-20
Setting the Extraction Variables 4-20
Setting Operating Conditions 4-21
Defining Wire Load Models 4-22
Defining Clocks 4-22
Setting Up Latch Extraction 4-23
Identifying False Paths 4-23
Removing Internal Input and Output Delays 4-24
Controlling Mode Information in the Extracted Model 4-24
Performing Timing Checks 4-24
Preparing for Crosstalk Analysis 4-24
Model Extraction Options 4-25
Extracted Model Examples 4-26

Extracting Clock-Gating Checks 4-30
Extraction of Combinational Paths Through Gating Logic 4-30
Extraction of Clock-Gating Checks As No-Change Arcs 4-31
Extracting Constant Values. 4-32
Extracting Timing Exceptions 4-33
Restricting the Types of Arcs Extracted 4-36
Back-Annotated Delay and Layout Information. 4-37
Back-Annotated Delay on Nets and Cells 4-37
Internal Nets 4-38
Boundary Nets 4-38
Cells With Annotated Delays 4-39
Guidelines for Back-Annotation 4-39
Performing Model Extraction. 4-40
Loading and Linking the Design 4-40
Preparing to Extract the Model 4-40
Generating the Model 4-41
PrimeTime User-Defined Attributes in .lib. 4-41
Merging Extracted Models 4-42

5. Model Validation

Model Validation Overview 5-2
Model Validation Commands. 5-2
Model Generation and Validation Procedure. 5-2
Slew Propagation. 5-4
write_interface_timing Report. 5-4
Arc Types. 5-6
Time Borrowing by Transparent Latches 5-7
compare_interface_timing Report. 5-8
Comparison Tolerance 5-9
Absolute Tolerance 5-10
Percentage Tolerance 5-10
Debugging Timing Arc Comparison Failures 5-11
Validation Flow With Timing Arc Debugging 5-11

Debugging Report Example. 5-13
Causes of Validation Failure 5-14

Debugging Other Comparison Failures 5-14
Transition Time 5-14
Capacitance. 5-15
Design Rules 5-15

6. Hierarchical Scope Checking

Scope Checking Overview 6-2

Types of Block Scope Checking 6-5
Clock Arrival, Transition, and Waveform 6-6
Interclock Skew and Uncertainty 6-6
Data Input and Output Conditions for Crosstalk 6-7

Block Scope Files 6-8
Generating a Block Scope File 6-8
Reporting Block Scope Files 6-9
Updating Block Scope Data 6-10

Checking the Block Scope at the Chip Level 6-10

Appendix A. Extracting Internal Pins

Clocks on Internal Pins of a Design A-2

Generated Clocks A-2

Check Pins for Clock Networks. A-2

Check Pins for Bidirectional Ports. A-3

Appendix B. Non-Shielded PrimeTime SI ILM Flow

Hierarchical Crosstalk Analysis. B-2

Generating the Block Context B-5

Context Files B-7

Top-Level Design Files B-8

Blocks Below the Top Level B-9

Block-Level Analysis and Timing Model Generation. B-10

Block-Level Analysis B-10
Generating ILMs for a Block Instance B-11
Annotating Arrival and Slew Information B-12
Validating an Interface Logic Model B-13

Top-Level Analysis B-14

Analysis Iterations Using Annotated Arrival and Slew B-15

Using an ILM. B-16

Index

Preface

This preface includes the following sections:

- What’s New in This Release
- About This User Guide
- Customer Support

What’s New in This Release

Information about new features, enhancements, and changes, along with known problems and limitations and resolved Synopsys Technical Action Requests (STARs), is available in the *PrimeTime Release Notes* in SolvNet.

To see the *PrimeTime Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.
2. Select PrimeTime Suite, then select a release in the list that appears at the bottom.

About This User Guide

The *PrimeTime Modeling User Guide* describes how to create, use, and verify chip-level timing models, including quick timing models, interface logic models, interface timing models, and extracted models.

Audience

This user guide is for design engineers who use PrimeTime for static timing analysis.

Related Publications

For additional information about PrimeTime, see Documentation on the Web, which is available through SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to refer to the documentation for the following related Synopsys products:

- PrimeTime SI, PrimeTime PX, and PrimeTime VX
- Design Compiler
- Library Compiler
- Library NCX

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* `[`*`pin2 ... pinN`*`]`
|	Indicates a choice among alternatives, such as `low	medium	high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet, go to the SolvNet Web page at the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar or in the footer.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to https://solvnet.synopsys.com (Synopsys user name and password required), and then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to Static Timing Models

This manual describes how to generate and use timing models for hierarchical timing analysis. You can create timing models by various methods, including quick timing models, interface logic models, and extracted timing models.

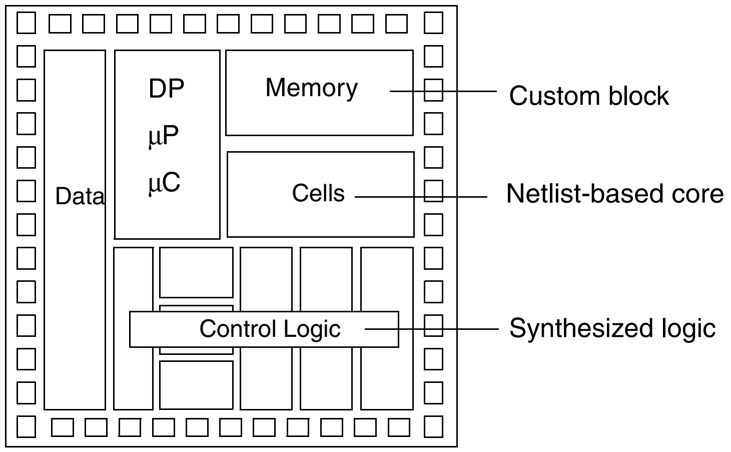
This introductory chapter covers the following topics:

- Timing Model Types
- Generated Timing Models
- Other Types of Timing Models

Timing Model Types

A typical chip can contain synthesized logic, netlist-based cores, and predesigned custom blocks, as shown in Figure 1-1.

Figure 1-1 Components of a Typical Chip



Before you can perform static timing analysis on a chip using PrimeTime, every leaf cell must have a timing model. For static timing purposes, a leaf cell can be a simple macrocell (such as a NAND, NOR, or flip-flop) or a complex block (such as a RAM or microprocessor).

Synopsys provides timing model capabilities that are appropriate for different types of leaf cells:

- Synopsys technology libraries compiled by Library Compiler
- Interface timing specification models compiled by Library Compiler
- Generated interface logic models
- Extracted models created by PrimeTime
- Quick timing models created by PrimeTime

Synthesized logic is modeled as a netlist containing gates such as NANDs, NORs, and flip-flops. The gates are modeled using the standard Synopsys modeling language. They are then compiled into a technology library database (.db) file using Library Compiler. This technology library is the same library used by Design Compiler.

A netlist-based core is a predefined functional block designed at the gate level. For chip-level timing analysis, you can model a netlist-based core using the gate-level netlist. From the gate-level netlist you can generate a flattened Verilog netlist that contains the interface logic of a block, or you can extract a timing model.

A predesigned custom block is defined at the transistor level and imported into the chip as a fixed unit, such as a RAM or microprocessor block. Because a gate-level netlist does not exist for this type of block, the Liberty modeling language can be used to describe the timing behavior of the block.

Interface timing specification models can also be used to model custom blocks. However, the Liberty modeling language offers additional features, such as mode-dependent timing arcs, internally generated clocks, and internal pins.

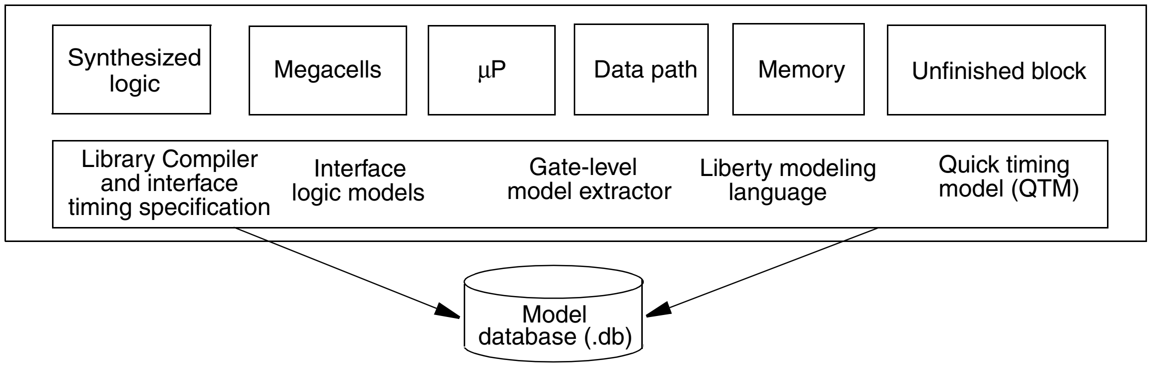
Table 1-1 lists the four methods of model generation that are used for the blocks in a typical chip.

Table 1-1 Model Generation for Blocks in Typical Chips

Block	**Method of model generation**
Synthesized logic	Gate-level netlist
Interface logic	Flattened netlist
Netlist-based core	Model extraction
Custom block	Liberty or interface timing specification

Some methods of model generation store model information in the database (.db) format, as shown in Figure 1-2.

Figure 1-2 Modeling Capabilities

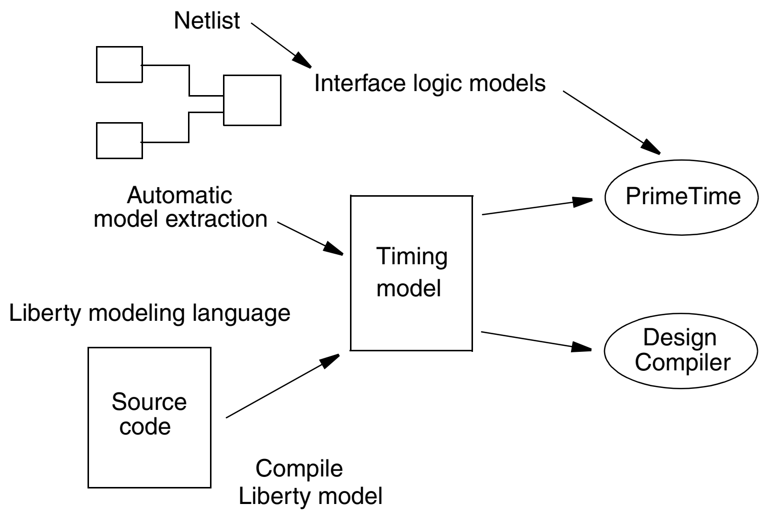


Generated Timing Models

PrimeTime is a chip-level timing analysis tool. You can create timing models for use by PrimeTime in three ways (see Figure 1-3):

- Interface logic modeling
- Model extraction
- Liberty modeling language

Figure 1-3 Methods for Creating a Timing Model

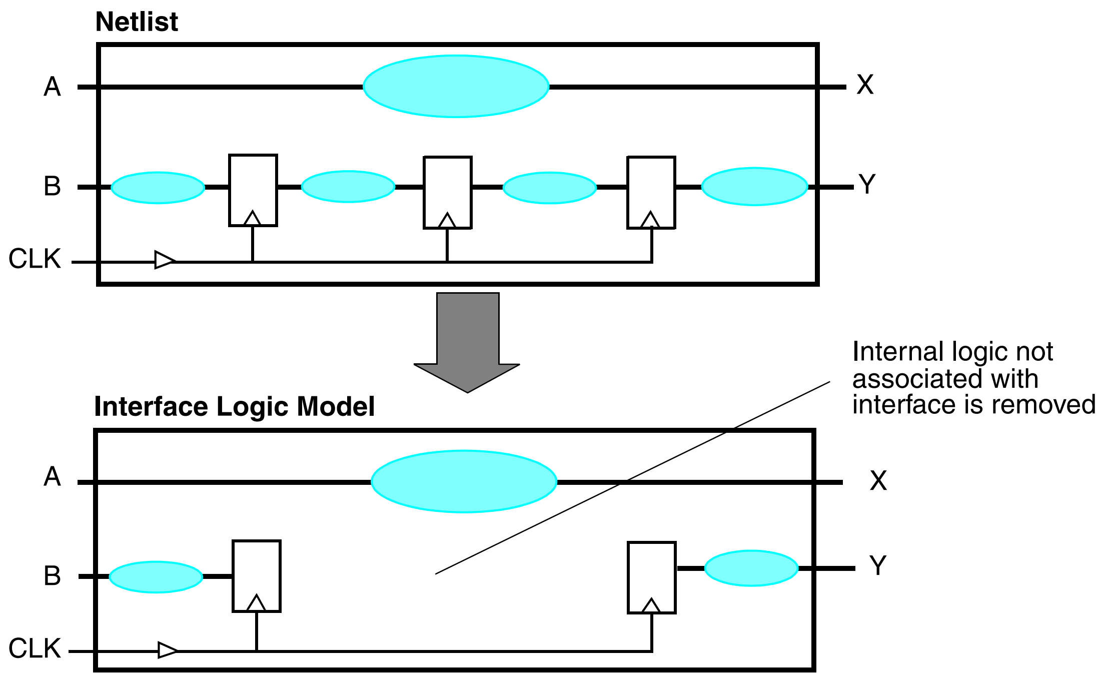


Interface Logic Modeling

Interface logic modeling generation uses a structural approach, where the original circuit is modeled by a smaller circuit that represents the interface logic of the block. Interface logic model generation runs significantly faster than model extraction, which uses a behavioral approach that models the underlying logic through timing arcs between internal pins and ports on a block.

Interface logic model generation takes as input the gate-level netlist for a block and generates a flattened Verilog netlist that contains only the interface logic and eliminates the internal register-to-register logic, as illustrated in Figure 1-4.

Figure 1-4 Generated Interface Logic Model



A generated interface logic model is context-independent, which means that the model is accurate for a range of operating environments. When the model is used in a design, the timing behavior of the model is different for different operating environments. For more information, see Chapter 3, “Interface Logic Models.”

Model Extraction

If the block you are attempting to model has a corresponding technology-mapped gate-level netlist, you can use the PrimeTime model extraction function to automatically generate a timing model for that block.

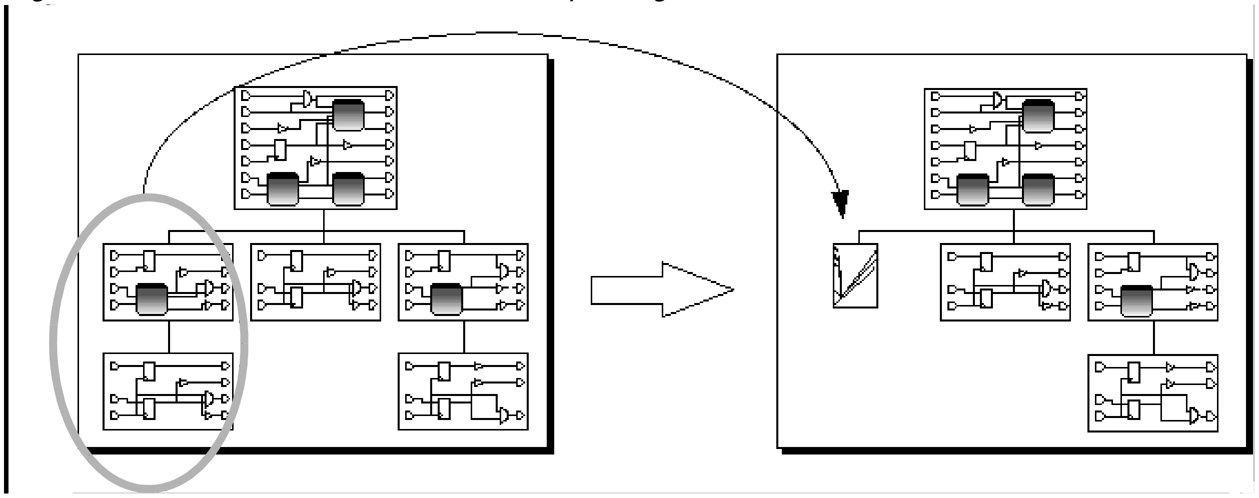
You can use model extraction for the following reasons:

- Reduce the runtime and memory for full-chip analysis

  You can run chip-level analysis with extracted models in place of the gate-level netlist for some modules, as shown in Figure 1-5.

- Extract a model to protect the intellectual property of a netlist-based core

Figure 1-5 Extracted Model of a Bottom-Up Design



Extraction uses a technology-mapped netlist as input and generates a context-independent timing model in the Synopsys .db format or Liberty format.

The generated model contains the same timing behavior as the original netlist. The delay values of arcs in the model are within the user-defined tolerance of the original path delays. For information about model extraction, see Chapter 4, “Extracted Timing Models.”

Other Types of Timing Models

PrimeTime supports other types of timing models, such as Synopsys technology libraries in .db format and quick timing models.

Synopsys Technology Libraries

Synopsys technology libraries (or synthesis libraries) specify the timing and function of macrocells in an ASIC technology using Library Compiler. Design Compiler uses these libraries during the synthesis and timing steps of the design flow.

A technology library contains cell descriptions, which provide specific information about each component within an ASIC technology. Cell descriptions include

Structure

Cell, bus, and pin structure that describes each cell’s connection to the outside world.

Function

The logical function of every output pin contained in the cell used by synthesis.

Timing

Timing analysis and design optimization information, such as the parameters for pin-to-pin timing relationships and timing constraints for sequential cells.

Other synthesis parameters

Parameters that describe area, power, and design rules (such as the maximum capacitance allowed for an output pin).

For more information about technology library cells, see the Library Compiler reference manuals.

Quick Timing Models

Using PrimeTime commands, you can quickly create an approximate timing model for a block without going through the effort of writing a detailed model using the interface timing specification.

You can use quick timing models early in the design cycle to describe rough, initial timing of a block. Later in the cycle, you can replace each quick timing model with a netlist block to obtain more accurate timing. See the information about generating quick timing models in Chapter 2, “Quick Timing Models.”

2

Quick Timing Models

A quick timing model is a temporary timing model you create with PrimeTime commands for a block that has no netlist available. You can use quick timing models early in the design cycle to describe the rough initial timing of blocks that have not yet been designed.

Quick timing models are described in the following sections:

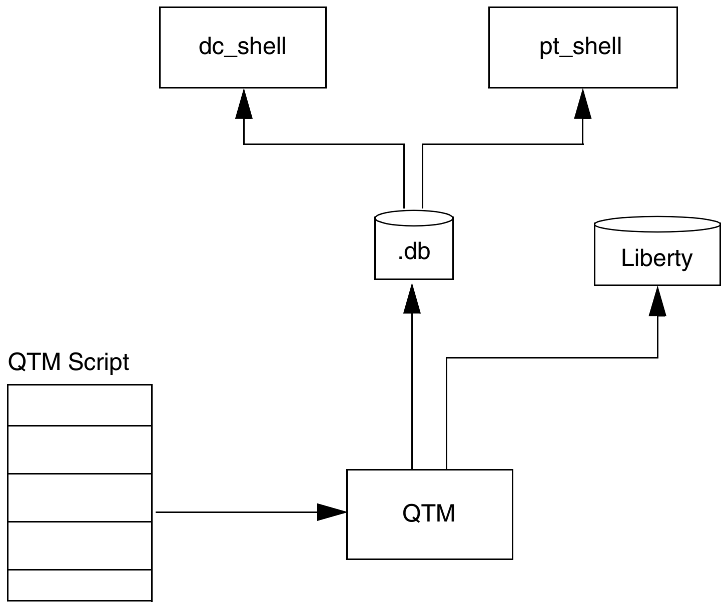
- Quick Timing Model Design Flow
- Quick Timing Model Parameters
- Creating a Quick Timing Model
- Using a Quick Timing Model in a Design
- Quick Timing Model Command Summary

Quick Timing Model Design Flow

To create a quick timing model, you use a series of PrimeTime commands to specify the model ports, the setup and hold constraints on the inputs, the clock-to-output path delays, and the input-to-output path delays. You can also specify the loads on input ports and the drive strength of output ports.

PrimeTime uses the information in the quick timing model to create a Synopsys library cell in .db format with the appropriate arcs. Figure 2-1 shows the design flow for using quick timing models.

Figure 2-1 Design Flow for Using Quick Timing Models



You can save a quick timing model in the Synopsys .db format, then instantiate the quick timing model in a design just as you would instantiate library cells or interface timing specification models. Both Design Compiler and PrimeTime accept designs with instantiated quick timing models.

Quick Timing Model Parameters

PrimeTime provides a two-step procedure for creating quick timing models:

- Set global parameters
- Specify model information, such as timing arcs

Setting Global Parameters

Specify the quick timing model global parameters before you specify model information. The quick timing model global parameters are

Technology library

The `set_qtm_technology` command specifies the technology-specific library to use for creating the quick timing model. Information includes the name of the technology library, the maximum transition time, the maximum capacitance, and the wire load model information.

Path type

The `create_qtm_path_type` command defines a path type. A library cell with an average fanout count for each level constitutes a path type. You can specify any number of path types. Use these types when you specify the timing arc in the quick timing model (for example, four levels of the NAND2 path type). For each level, the quick timing model computes the delay for each of those path types when the path is defined. You can also specify the timing arcs in terms of the actual numbers.

Flip-flop setup time, hold time, and clock-to-output delay

The `set_qtm_global_parameter` command specifies the global setup time, hold time, and clock-to-output delay for a flip-flop in the model. You specify these parameters globally because it is unlikely that you need to specify different times for different arcs.

If you want to specify a different setup time, hold time, or clock-to-output delay, adjust the timing arcs you specify for the model. Specify timing arcs by providing a library element (such as a flip-flop) or by specifying a particular time value (such as 2 ns).

Load type

The `create_qtm_load_type` command defines load types that are used to specify the net capacitance of input ports. A library cell in the library constitutes a load type. You can label each of the library cell types and use this label when specifying the capacitance of the input ports (such as two NAND2 loads).

Drive type

The `create_qtm_drive_type` command creates drive types. You can set the drive type for each output port. The drive type can be any library cell in the library (similar to the load type). This command enables you to specify an output drive of a port that might be different from the library cell you use in the path types.

Specifying Model Information

Specify the model information after you specify the global model parameters. The types of model information are

Model port

The `create_qtm_port` command specifies model ports.

Input port capacitance

The `set_qtm_port_load` command specifies input port capacitance in terms of the load type defined in the global parameter section.

Output port drive

The `set_qtm_port_drive` command specifies output port drive in terms of the drive type global parameter.

Timing arcs

Use the `create_qtm_delay_arc` command to specify delay arcs between ports of the model (clock to output and input to output). Use the `create_qtm_constraint_arc` command to specify constraint arcs (setup and hold). To create a generated clock, use `create_qtm_generated_clock`.

You can specify constraint and delay arcs in terms of levels of path types specified in the global parameters. The calculation of the delay for each arc is explained in the following section.

Computing Arc Delays

You can specify path delays and constraint values in terms of the delay arcs of any gate in the current technology library. For example, you can specify that the path delay of an input port to an output is equivalent to a path containing five NAND gates with an average fanout of three for each gate. PrimeTime computes the delay of this path and generates the corresponding arc.

Similarly, you can specify that the typical setup value is equal to the setup time of the D flip-flop in the library or that the output drive of the out1 port is equivalent to that of a particular buffer in the library.

PrimeTime computes the constraint and delay values for each timing arc from the arc specification. The path type and the number of levels of logic are stored for each timing arc. The global parameters contain setup, hold, and clock-to-output delays. The delay and level are computed from the path type.

Using the information you provide, PrimeTime computes the delay for each arc as follows.

- For a constraint arc:

```
delay = (#levels) * (delay/level) + setup time
```

- For a delay arc (launch type from-clock-to-output-port):

```
delay = (#levels) * (delay/level) + CLK-Q delay + output
```

```
load-dependent delay
```

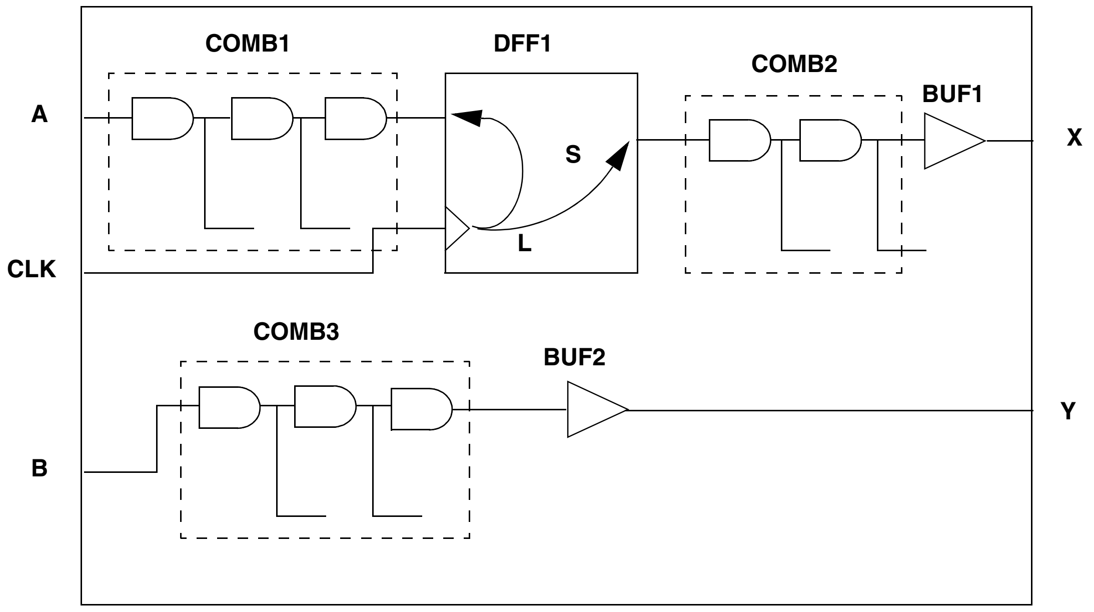
- For a delay arc (combinational arc):

```
delay = (#levels) * (delay/level) + output load-dependent
delay
```

Example

Figure 2-2 shows the quick timing model for an undesigned block.

Figure 2-2 Quick Timing Model for an Undesigned Block



In Figure 2-2,

- Port A is constrained by CLK using the setup requirement. This is modeled as a combinational path, COMB1, plus the setup to a flip-flop. The flip-flop setup time is the global setup time.
- The path from CLK to port X is modeled as the clock-to-output delay of a flip-flop plus the delays of combinational path COMB2 and driver BUF1.
- The path from input port B to output port Y is modeled as combinational path COMB3 plus the delay of the output driver BUF2.

To define this quick timing model,

1. Define a path type (path1) that is a two-input AND with a fanout of two.
2. Define a drive type, named BUF1.

3. Define another drive type, named BUF2.
4. Define the global setup time, which is equivalent to the setup time of DFF1.
5. Define the global clock-to-output delay, which is equivalent to the launch time of DFF1.

The setup time of port A relative to CLK is

```
delay of COMB1 + setup time of DFF1 (denoted by arc ‘S’) =
3 * (delay of path1) + global setup time
```

The delay from DFF1 to output port X is

```
launch time of DFF1 (denoted by arc ‘L’) + delay of COMB2 +
load-
dependent delay of BUF1 =
launch time of DFF1 + 2 * (delay of path1) + delay of BUF1
```

The delay from input port B to output port Y is

```
delay of COMB3 + load-dependent delay of BUF2 =
3 * delay of path1 + delay of BUF2
```

For input arcs, if the fanout of the input port is high, you must take into account the delay caused by buffering. If there is an input port with a high fanout, insert a chain of buffers before driving the library cells, or split the fanout among several buffers. You need to account for such delays by adding to the number of levels when specifying the input arcs.

Creating a Quick Timing Model

Figure 2-3 shows a quick timing model representation of a block. Constraint arcs appear as dashed lines and delay arcs appear as solid lines. Port CLK is a clock port, ports A and B are input ports, and ports X and Y are output ports.

The arcs are

SetupA

Constrains port A; the constraining port is clock CLK.

HoldA

Constrains port A; the constraining port is clock CLK.

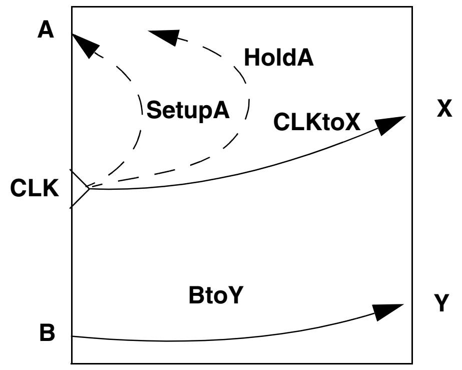
CLKtoX

Delay arc from CLK to X.

BtoY

Delay arc from B to Y.

Figure 2-3 Quick Timing Model Representation of a Block



The tasks for specifying the quick timing model for Figure 2-3 are

- Create the quick timing model and set global parameters. See "Creating the Model and Setting Global Parameters" on page 2-7.
- Specify model information. See "Specifying Model Information" on page 2-8.
- View the model. See "Viewing the New Model" on page 2-9.
- Save the model. See "Saving a Model" on page 2-9.

You need to use the quick timing model commands after `create_qtm_model` and before `save_qtm_model`.

Creating the Model and Setting Global Parameters

To create a model and set the global parameters,

1. Inform PrimeTime that a new model named "example" is being created.

   ```
   pt_shell> create_qtm_model qtm_example
   ```

2. Specify the technology library to use with your design.

   ```
   pt_shell> set_qtm_technology -library lib_new
   ```

3. Define a path type.

   ```
   pt_shell> create_qtm_path_type path1 -lib_cell \
                nand21 -fanout 2
   ```

   In this command, path1 is a two-input NAND with a fanout of two.

4. Define a load type.

```
pt_shell> create_qtm_load_type load1 -lib_cell and22
```

PrimeTime selects the input port of a library cell to use at the input load.

5. Define the drive type.

```
pt_shell> create_qtm_drive_type drive1 -lib_cell buf1

pt_shell> create_qtm_drive_type drive2 -lib_cell buf2
```

6. Define a global setup time.

```
pt_shell> set_qtm_global_parameter -param setup \
          -lib_cell DFF1 -clock CLK -pin D
```

In this command, the setup time is equivalent to the setup time of the DFF1 library cell.

7. Define a hold time.

```
pt_shell> set_qtm_global_parameter -param \
          hold -value 0.0
```

8. Define a clock to the output delay time.

```
pt_shell> set_qtm_global_parameter -param \
          clk_to_output -lib_cell DFF1 -clock CLK -pin Q
```

In this command, the output delay time is equivalent to the launch time of the DFF1 library cell.

Specifying Model Information

After you create a quick timing model and define its parameters, specify the various ports, attributes, and arcs for the model:

1. Create a clock port.

```
pt_shell> create_qtm_port {CLK} -type clock
```

2. Create the input ports.

```
pt_shell> create_qtm_port {A B} -type input
```

3. Create the output ports.

```
pt_shell> create_qtm_port {X Y} -type output
```

4. Set the load1 load type on the A and B ports.

```
pt_shell> set_qtm_port_load {A B} -type load1 -factor 2
```

5. Set a load of three capacitance units on CLK.

```
pt_shell> set_qtm_port_load {CLK} -value 3
```

6. Set a drive on the output ports.

```
pt_shell> set_qtm_port_drive X -type drive1

pt_shell> set_qtm_port_drive Y -type drive2
```

7. Define the setup and hold arcs.

```
pt_shell> create_qtm_constraint_arc -setup -edge rise \
          -name SetupA -from CLK -to A -path_type \
          path1 -path_factor 2

pt_shell> create_qtm_constraint_arc -hold -edge rise \
          -name HoldA -from CLK -to A -path_type \
          path 1 -path_factor 2
```

8. Create the delay arcs.

```
pt_shell> create_qtm_delay_arc -name BtoY -from B \
         -to Y -path_type path1 -path_factor 3

pt_shell> create_qtm_delay_arc -name CLKtoX \
          -from CLK -to X -path_type path1 -path_factor 2
```

Viewing the New Model

Generate a report that shows the defined global parameters, ports, and arcs in the quick timing model. Use the `report_qtm_model` command. For example, enter:

```
pt_shell> report_qtm_model
```

Saving a Model

Save the model in .db format or .lib format.

```
pt_shell> save_qtm_model -output file_name \
          -format {db}
```

If you do not use the `-format` option, PrimeTime creates a model in .db format only.

You can choose to create either a library cell or wrapper and core by using or not using the `-library_cell` option in the `save_qtm_model` command. A library cell is easier to use but is not compatible with certain analysis flows. The `-format` option only controls the output format for a library cell; PrimeTime always writes a wrapper in .db format.

Converting STAMP Models to .lib Format

The `translate_stamp_compiler` command provides a mechanism for converting individual models from STAMP to .lib format, which can then be compiled with Library Compiler. You need to provide the model and data files as inputs and the name of the output file.

Saving a Model as .db Wrapper and Core

To save a quick timing model as a .db wrapper and core, use:

```
pt_shell> save_qtm_model -format db file_name
```

A command in this form writes two files: *file_name*_lib.db containing a library cell *model_name*_core, and *file_name*.db containing wrapper design *model_name* (where *model_name* is the name specified in the `create_qtm_model` command).

Saving a Model as .db Library Cell

To save a quick timing model as a .db library cell, use:

```
pt_shell> save_qtm_model -format db file_name -library_cell
```

A command in this form writes one file: *file_name*_lib.db containing a library cell *model_name*.

Using a Quick Timing Model in a Design

To use a quick timing model in a design,

1. Instantiate the model in your design the same way you instantiate a leaf library cell.
2. Add the library file name to the `link_path` variable. Ensure that the path name of the directory containing this file appears in the `search_path` variable.
3. If your quick timing model has a wrapper, use the `read_db` command to read the wrapper file.
4. Link the design that uses the model.
5. Define the clocks and other timing assertions for the design containing the model.
6. Use the PrimeTime `report_timing` command to get reports on the timing of the design.

Quick Timing Model Command Summary

Table 2-1 summarizes the quick timing model commands that define, report, and save quick timing models.

Table 2-1 Quick Timing Model Commands Summary

Task	Command
Create a constraint arc.	`create_qtm_constraint_arc`
Create a delay arc for a quick timing model.	`create_qtm_delay_arc`
Create a drive type in a quick timing model description.	`create_qtm_drive_type`
Create a load type for a quick timing model description.	`create_qtm_load_type`
Begin defining a quick timing model.	`create_qtm_model`
Create a path type in a quick timing model.	`create_qtm_path_type`
Create a quick timing model port.	`create_qtm_port`
Create a generated clock for the model.	`create_qtm_generated_clock`
Report model data.	`report_qtm_model`
Save the quick timing model.	`save_qtm_model`
Select a quick timing model port.	`select_qtm_port`
Set a global parameter.	`set_qtm_global_parameter`
Set drive on a port.	`set_qtm_port_drive`
Set load on ports.	`set_qtm_port_load`
Set various technology parameters.	`set_qtm_technology`

3

Interface Logic Models

This chapter describes interface logic models (ILMs) and the flow that is used in PrimeTime.

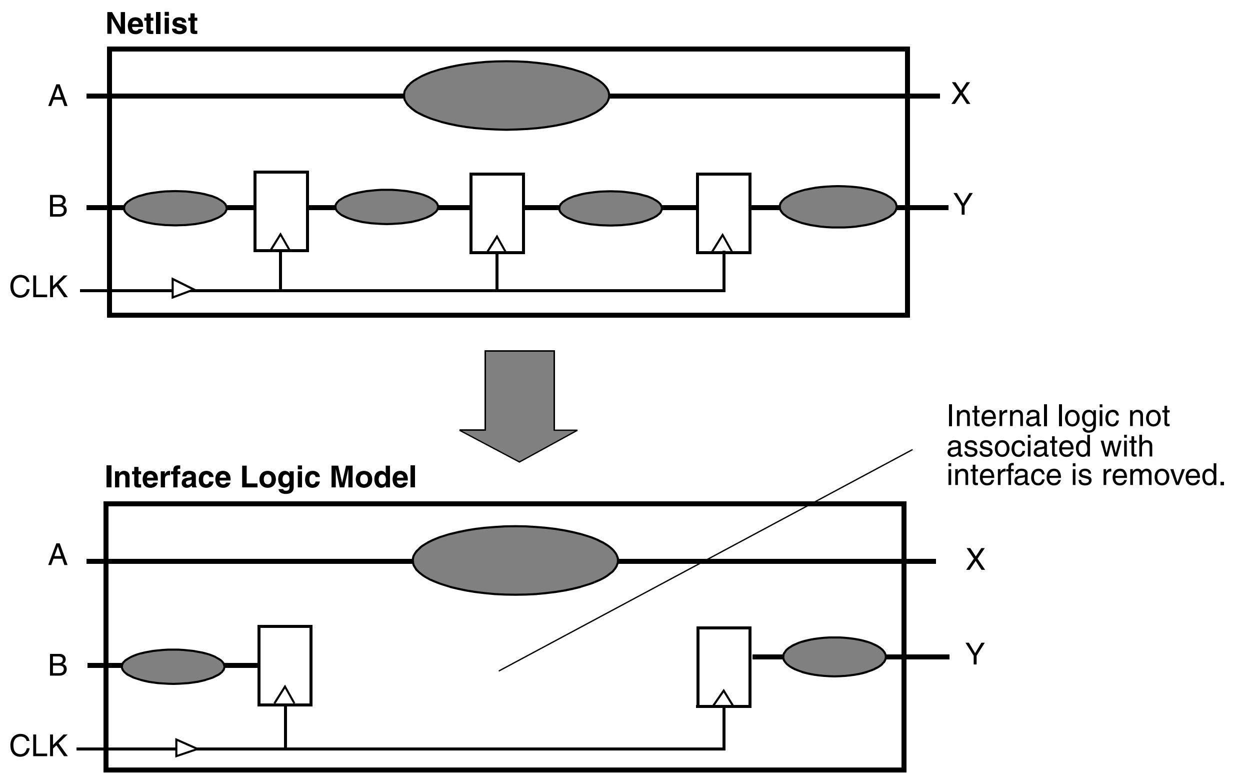
- Overview of Interface Logic Modeling
- Benefits, Flow, and Limitations of ILM
- How ILMs Are Used
- Generating an ILM
- Generating ILMs Using the create_ilm Command
- Specific Setups with ILM
- Recommended Options for the create_ilm Command
- Using the ILM in a Top-Level Netlist
- ILMs with PrimeTime SI

Overview of Interface Logic Modeling

An ILM is a partial netlist that retains the combinational logic from each input port to the first stage of sequential elements of the block. It is also the combinational logic from the last stage of sequential elements to each output port of the block. The clock paths to these sequential elements are also retained. Combinational paths from the input ports that do not encounter a sequential element and pass directly to an output port are also retained in the ILM.

The intent of an ILM is to produce a model that closely resembles the timing of the interface to that of the block-level netlist. The ILM generator takes as input the gate-level netlist for a block and generates a flattened Verilog netlist that contains only the interface logic, without the internal register-to-register logic, as shown in Figure 3-1.

Figure 3-1 Generated Interface Logic Model



The ILM partial netlist is instantiated in place of the block-level netlist within the chip (top-level) netlist. The methodology outlined in this chapter provides the steps needed to generate, validate, and use ILMs in a hierarchical static timing analysis (STA) flow.

Benefits, Flow, and Limitations of ILM

An ILM partial netlist can provide a smaller memory footprint, faster runtimes, and an extremely accurate timing of the interface logic of a block.

You can use ILMs in Standard Delay Format (SDF) or Standard Parasitic Exchange Format (SPEF) based flows. In addition, you can use ILMs in PrimeTime SI using both crosstalk and noise analysis.

To use the model in a hierarchical static timing analysis methodology, you must validate an ILM. This validation requires that the timing and scope match the block-level netlist usage within the design. A certain amount of time is needed to validate an ILM.

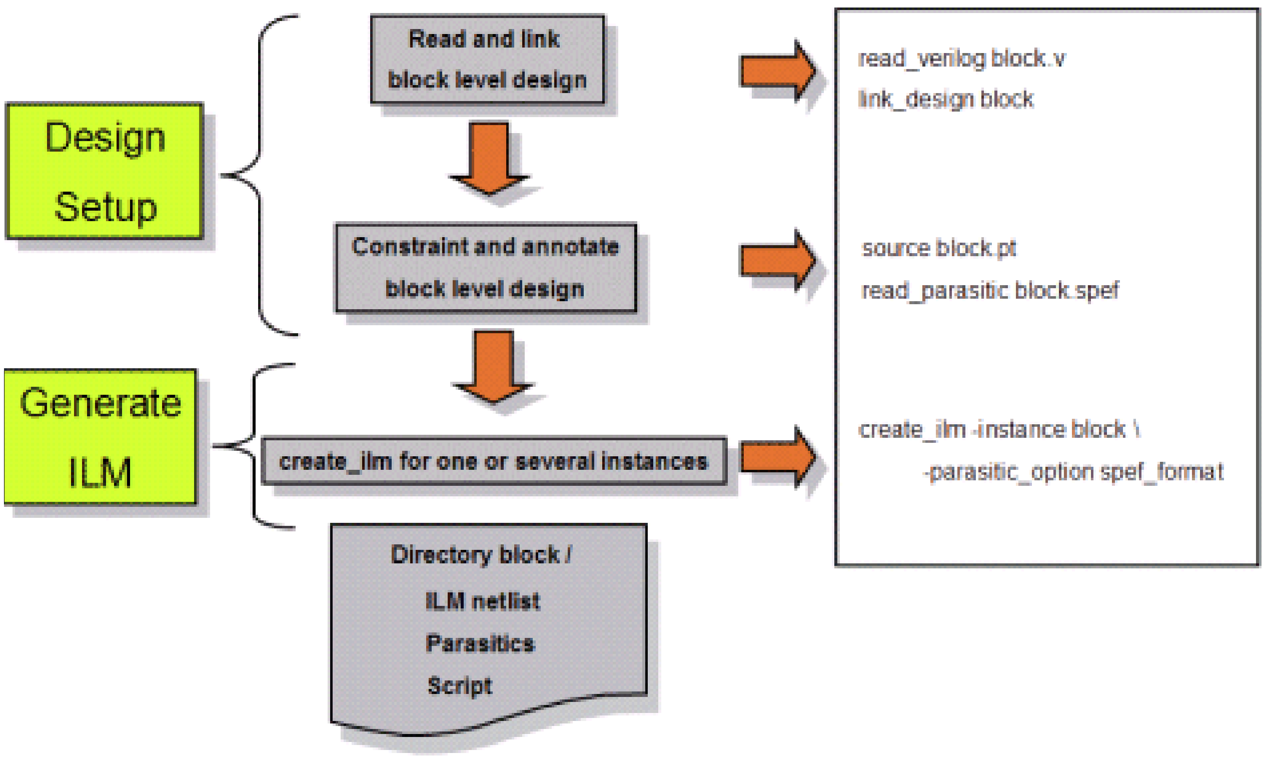
How ILMs Are Used

To generate and use an ILM in a hierarchical static timing analysis flow, do the following:

1. Generate an ILM
2. Validate an ILM
3. Use the ILM in a top-level netlist

Figure 3-2 outlines the steps used when generating an ILM:

Figure 3-2 ILM Usage Flow: Block Level


Design Setup
Generate ILM
Read and link block level design
Constraint and annotate block level design
create_ilm for one or several instances
Directory block / ILM netlist Parasitics Script
read_verilog block.v
link_design block
source block.pt
read_parasitic block.spef
create_ilm -instance block \
-parasitic_option spef_format

Generating an ILM

To generate an ILM, do the following

1. Properly constrain the block.
2. Generate the ILM for the block.

To constrain a block, you must define all the clocks and a conservative range of minimum and maximum values for bounding the input and output ports of the block used during the top-level static timing analysis.

These values should include bounding both the arrival times and transition times of each port of the block. This bounding is accomplished using the following commands:

- `set_input_delay -min/max`
- `set_output_delay -min/max`
- `set_input_transition -min/max`

- `set_clock_latency -dynamic`

During scope checking of the block within the top-level static timing analysis, these values are verified to ensure that the block is operating within the context range used during extraction.

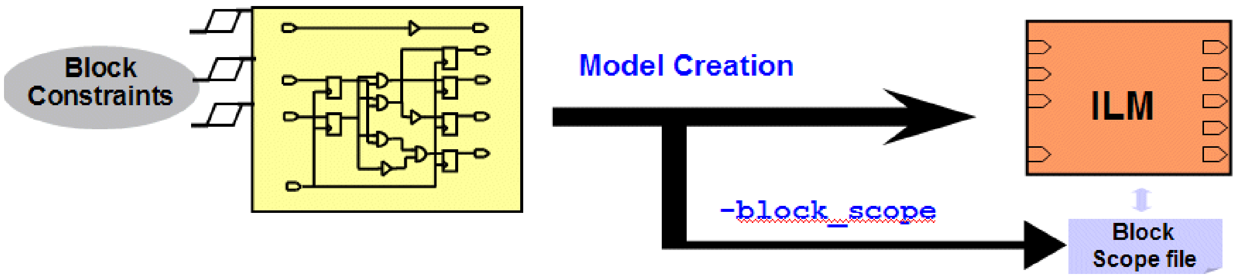
Generating ILMs Using the create_ilm Command

The `create_ilm` command creates an ILM for the current block-level netlist. By default, the command creates a model that includes all of the block interface logic and excludes all the internal register-to-register logic of the block.

A basic set of options for generating an ILM are:

1. To produce a file that reflects the range of signal arrivals and slews entering and leaving the block, use the `-block_scope` option of the `create_ilm` command. Run the `check_block_scope` command during the top-level analysis to check to see if the range is honored. Figure 3-3 shows the process for checking the block level scope.

Figure 3-3 Checking the Block Level Scope



2. To produce the files needed to verify the timing of the ILM, use the `-verification_script` option of the `create_ilm` command.
3. To produce a SPEF file for the nets within the ILM, use the `-parasitic_options` option of the `create_ilm` command.

For example, the following sequence of commands reads in a design called A.v and creates an ILM for that block:

```
pt_shell> set search_path \
            " . ./db ./verilog ./spef ./libraries"
pt_shell> set link_path " * my_lib.db"
pt_shell> read_verilog A.v
pt_shell> link_design A
pt_shell> read_parasitics A.spef
pt_shell> source A.sdc; # script applies constraints
pt_shell> create_ilm -block_scope \
          -verification_script \
          -parasitic_options {spef input_port_nets constant_nets}

pt_shell> write_interface_timing net_tim.rep
```

In this example, the `create_ilm` command produces the following files in the A subdirectory:

A/ilm.v - Verilog netlist

A/ilm_inst.pt.gz - ILM constraint file

A/ilm.spef.gz - ILM SPEF file

A/ilm.scope - ILM block scope file

A/ilm_verif.pt.gz - ILM verification file

Specific Setups with ILM

You can generate an ILM for a block-level netlist that has the following:

- High Fanout Ports (scan_enable, reset, and set)
- Latches

High Fanout Ports

When generating an ILM, each input port that is not defined as a clock is considered as a data port. When the tool generates an ILM, it traces each input port to the first level of the sequential element. For input ports, such as scan_enable, this means that each flip-flop in the block netlist is kept.

To address this, you can use either the `-auto_ignore` switch or the `-ignore_ports` option. Both allow for special handling of these input ports. The `-auto_ignore` switch allows the `create_ilm` command to check to see if an input port goes to more than a specific percent of the sequential elements in the block. The percentage is governed by the `ilm_ignore_percentage` variable with the default being 25%. You can also use the `-ignore_ports` options. This option allows for special handling of the `scan_enable` in this specific instance. For example,

```
pt_shell> create_ilm -ignore_ports [get_ports scan_enable]
```

When using the `-ignore` or `-auto_ignore` options, it can be useful to add the `-keep_ignore_fanout` option. In this case, the fanout from ignored input ports to interface logic is maintained in the model, making the ILM validation easier.

Latches

For designs with latches, you should define the number of levels of borrowing for the latches at the interface. You should set the value of the `-latch_level` option of the `create_ilm` command to match the number of levels of latch borrowing. For example,

```
pt_shell> create_ilm -latch_level number of levels of transparent latches
```

Recommended Options for the create_ilm Command

In the SPEF-based flows, you should use the `-parasitic_options` option with the `create_ilm` command to generate parasitics for ILMs, as follows:

```
pt_shell> create_ilm -parasitic_options {spef input_port_nets \
          constant_nets}
```

To aid in validating the ILM, you should produce both the verification and block scope files by using the `-block_scope` and `-verification_script` options of the `create_ilm` command.

To add pins to the interface logic that otherwise would not be included, use the `-include_pins` option of the `create_ilm` command. For example, if you wanted to keep an internal clock gating logic, you can add these pins for this internal clock gating logic:

```
pt_shell> set pins [get_pins -of_objects [get_net w12]]
pt_shell> create_ilm -include_pins $pins
```

In summary, in a SPEF-based flow, the recommended command to create an ILM is as follows:

```
pt_shell> create_ilm -parasitic_options \
{spef input_port_nets constant_nets} -block_scope -verification_script
```

Model Validating an ILM

There are two checks that help ensure that the ILM block works within the static timing analysis hierarchical flow:

- ILM Timing Matches Block-Level Netlist Timing

- Scope Checking for ILM Within Chip-Level Design

ILM Timing Matches Block-Level Netlist Timing

The first check validates that the interface timing in the ILM matches the block-level netlist. In the flow, both the ILM and block-level netlist produce a file describing the timing at its interface. The `compare_interface_timing` command checks the two files and reports the results.

The scope check verifies that the block at the top-level falls within the ranges of I/O constraints that were set during block-level analysis.

The `create_ilm` command with the `-verification_script` option produces the ilm_verif.pt.gz file with the constraints for the block. You produce the interface timing for both the ILM and the full block netlist and then compare this timing.

To produce the interface timing for the ILM block, you should use the following syntax:

```
pt_shell> set search_path \
            " . ./db ./verilog ./spef ./libraries"
pt_shell> set link_path " * my_lib.db"
pt_shell> read_verilog A/ilm.v
pt_shell> link_design block
pt_shell> source A/ilm_verif.pt.gz
pt_shell> read_parasitics A/ilm.spef.gz
pt_shell> update_timing -full
pt_shell> write_interface_timing ilm_tim.rep
```

The output of the timing report file is similar to the following:

```
****************************************
Command: write_interface_timing
        -significant_digits 6
Design : A
Version: B-2008.06
Date   : Fri Jul 25 11:08:14 2008
****************************************
*********************************************
Section: slack
Info   : Worst-case slack for each port and path group
Design : A
*********************************************

Generated Clock and Source Info:

Attributes:
   L<n> - latch level where <n> is 0 for first level
```

```
                                Arc                 Worst-case
From           To               Type                Slack
-----------------------------------------------------------
EN(r)          clock1(f)        setup                3.826993
EN(f)          clock1(f)        setup                3.805498
EN(r)          clock1(f)        hold                 5.903111
EN(f)          clock1(f)        hold                 5.923214
ain[0](r)      clock1(r)        setup               13.671270
ain[0](f)      clock1(r)        setup               13.650400
ain[0](r)      clock1(r)        hold                -3.937757
ain[0](f)      clock1(r)        hold                -3.929484
```

For the full netlist, you now have the net_tim.rep timing file and for the ILM, you have the ilm_tim.rep timing file.

```
pt_shell> compare_interface_timing net_tim.rep ilm_tim.rep
```

Slight differences in the slack value can be due to slew propagation differences for unconnected pins within the ILM. Therefore, you should set a reasonable range for these types of differences, such as within 2 percent of the clock period.

The `compare_interface_timing` command compares the values in the two timing report files to each other. The format of the report is as follows:

```
****************************************
Command: compare_interface_timing
         net_tim.rep A/ilm_tim.rep
         -session full
Design : A
Version: B-2008.06
Date   : Fri Jul 25 14:11:23 2008
****************************************

                    Arc                    Slack
From        To          Type               Ref         Cmp           Diff    Status
-------------------------------------------------------------------------------------
EN(r)       clock1(f)   setup              3.83        3.83          0.00     PASS
EN(f)       clock1(f)   setup              3.81        3.81          0.00     PASS
EN(r)       clock1(f)   hold               5.90        5.90          0.00     PASS
EN(f)       clock1(f)   hold               5.92        5.92          0.00     PASS
ain[0](r)   clock1(r)   setup             13.67       13.67          0.00     PASS
ain[0](f)   clock1(r)   setup             13.65       13.65          0.00     PASS
ain[0](r)   clock1(r)   hold              -3.94       -3.94          0.00     PASS
ain[0](f)   clock1(r)   hold              -3.93       -3.93          0.00     PASS
…
            Totals        Slack       Transition Time     Capacitance        Rules
-------------------------------------------------------------------------------------
Passed     1098           258                280              560              -
Failed        0             0                  0                0              0
Total      1098           258                280              560              0
```

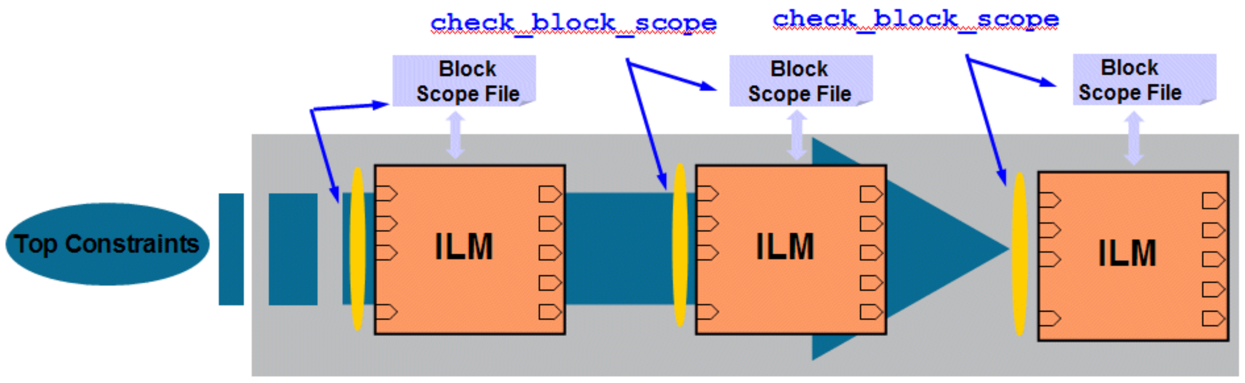
By validating that the slack, capacitance, and transition times match between the full netlist and the ILM netlist, you are assured that the ILM matches the full netlist.

Scope Checking for ILM Within Chip-Level Design

Scope checking verifies that external timing conditions are within the ranges specified for block-level analysis. To ensure that the block context is valid for the original block-level timing analysis, you can check the block scope at the higher level.

While generating the ILM, the `-block_scope` option produces the block scope file. The `check_block_scope` command compares the actual arrivals and slews in the top-level static timing analysis to the values in the block scope file. Figure 3-4 shows the process for checking the top level scope.

Figure 3-4 Checking the Top Level Scope



The `check_block_scope` command checks the following by default:

- Clock consistency between block-level and top-level clock definitions, including existence, period, and mode (ideal/propagated).
- Clock arrival times at block boundary pins
    - Scope contains absolute numbers
    - The `check_block_scope` command transforms these absolute numbers to relative numbers to check clock and edge relationships.
    - The `check_block_scope` command determines a shift computed to minimize the number of scope violations and applies it to all the arrivals.
- Clock transition at block boundary pins
- Interclock relationships at block boundary

From the `create_ilm -block_scope` command, the A/ilm.scope file is created.

To validate that the chip-level context is within the range of values from which the block was extracted, use the following commands:

```
pt_shell> set search_path \
            " . ./db ./verilog ./spef ./libraries"
pt_shell> set link_path "* my_lib.db"
pt_shell> read_verilog chip.v A/ilm.v
pt_shell> link_design chip
pt_shell> read_parasitics -path inst1 A/ilm.spef -increment

pt_shell> current_instance inst1
pt_shell> source A/ilm_inst.pt.gz
pt_shell> current_instance
pt_shell> source chip.pt
pt_shell> update_timing

pt_shell> check_block_scope -instances inst1 A/ilm.scope
```

To report the captured block scope data, use the `report_scope_data` command. The resulting block scope report is similar to the following:

```
****************************************
Report : scope_check
     -instance elvis
     -scope_scenario default
     -relative_clock_arrival
Design : top
Version: B-2008.06
Date   : Fri Jul 25 16:33:16 2008
****************************************

1. Checking analysis environment settings.

  No global derating violations.

2. Matching top-level and block-level clocks.

   (Information) Found top-level clock 'clock_bob' matching block-level
clock 'clock1'.
   (Information) Found top-level clock 'clock2' matching block-level
clock 'clock2'.

   Clock mappings
       Top-level       Block-level     Ref_pin
   -----------------------------------------------------------------
       clock2          clock2          elvis/clk_buf2/A    (MET)
       clock_bob       clock1          elvis/clk_buf1/A    (MET)

3. Checking inter-clock relationships at block boundary.

    No violation to report or not enabled for checking.

4. Checking arrival and transition times at block boundary pins.

      No violation to report or not enabled for checking.
```

You should verify that both the timing and scope checking is correct before proceeding to use the ILM block in the top-level chip level static timing analysis.

Model Validating Issues and Resolutions

The model validating issues are explained in the following sections:

- Modeling Issues
- Scope Checking Issues

Modeling Issues

For ILMs, the `compare_interface_timing` command identifies interface paths that are not matching the block-level netlist.

One possible source of a slack issue can be due to slew propagation of unconnected input pins of the interface cell. To validate this difference, you should use the `report_timing -tran` command on both the block-level netlist compared with the ILM. The report shows the differences in the slew at the output pins of the cells. You can also see the difference by using path-based analysis by using the `report_timing -recalculate` command for both the block-level netlist and the ILM.

You can add these additional unconnected pins to the ILM by using the `create_ilm -include_pin` command. This makes the new ILM larger.

Scope Checking Issues

Scope checking issues with clock definitions for the ILM are identified when you execute the `check_block_scope` command. You should ensure that the clock definitions match from the block-level to the top-level netlist.

Another potential context issue has to do with signal latencies and transition times entering the block. The latencies or transition times entering the block from the top-level does not fall within the range defined while extracting the block.

There are two solutions to this issue. The first solution is to correct the top-level design to match the range set during the extraction. The second solution is to review the ranges set during the block-level analysis. You can then revise these signal ranges during block-level analysis prior to producing a new block scope file for the block.

Using the ILM in a Top-Level Netlist

After you have generated and verified the ILM, you can use it at the chip level in place of the block-level netlist for timing analysis.

To use the generated model,

1. Read in the netlist for each interface model. For example, enter

```
pt_shell> read_verilog A.v
```

2. Read and link the chip-level design:

```
pt_shell> read_verilog chip.v
pt_shell> link_design chip
```

3. For each ILM, read in the SDF or parasitics for the block. To specify the hierarchical path name leading to the instance, use the `-path` option with the `read_sdf` and `read_parasitics` commands. Enter one of the following:

```
pt_shell> read_parasitics -path inst1 -increment \
              A/ilm.spef.gz
```

or

```
pt_shell> read_sdf -path inst1 A/ilm.sdf
```

4. For each ILM, set the current instance to the block and source the script containing assertions and exceptions defined on the block.

```
pt_shell> current_instance inst1
pt_shell> source ilm_inst.pt.gz
```

5. To go back to the top level netlist, enter

```
pt_shell> current_instance
```

6. Perform chip-level timing analysis.

```
pt_shell> check_timing -verbose
pt_shell> report_timing
```

7. Perform scope checking.

```
pt_shell> check_block_scope -instances inst1 A/ilm.scope
```

ILMs with PrimeTime SI

This section discusses the shielded PrimeTime SI ILM flow. A shielded block is where the designer has ensured that there are no wires routed either on top of or in close proximity to the block; therefore, crosstalk effects from the rest of the design into the block are not produced.

The methodology and flow for the PrimeTime flow is comparable to that of the PrimeTime SI ILM flow. The steps are as follows:

1. Generating PrimeTime SI ILM
2. Model Validating for PrimeTime SI ILM
3. Using a PrimeTime SI ILM in a Top-Level Netlist

Generating PrimeTime SI ILM

For generating a PrimeTime SI ILM, provide the transition times (minimum and maximum) for each input port. PrimeTime recommends that you use the `set_driving_cell` command. This should reflect the cell driving the input port at the chip-level. For example,

```
pt_shell> set_driving_cell  -min/max
```

You must add the following to your block-level constraints to enable PrimeTime SI analysis:

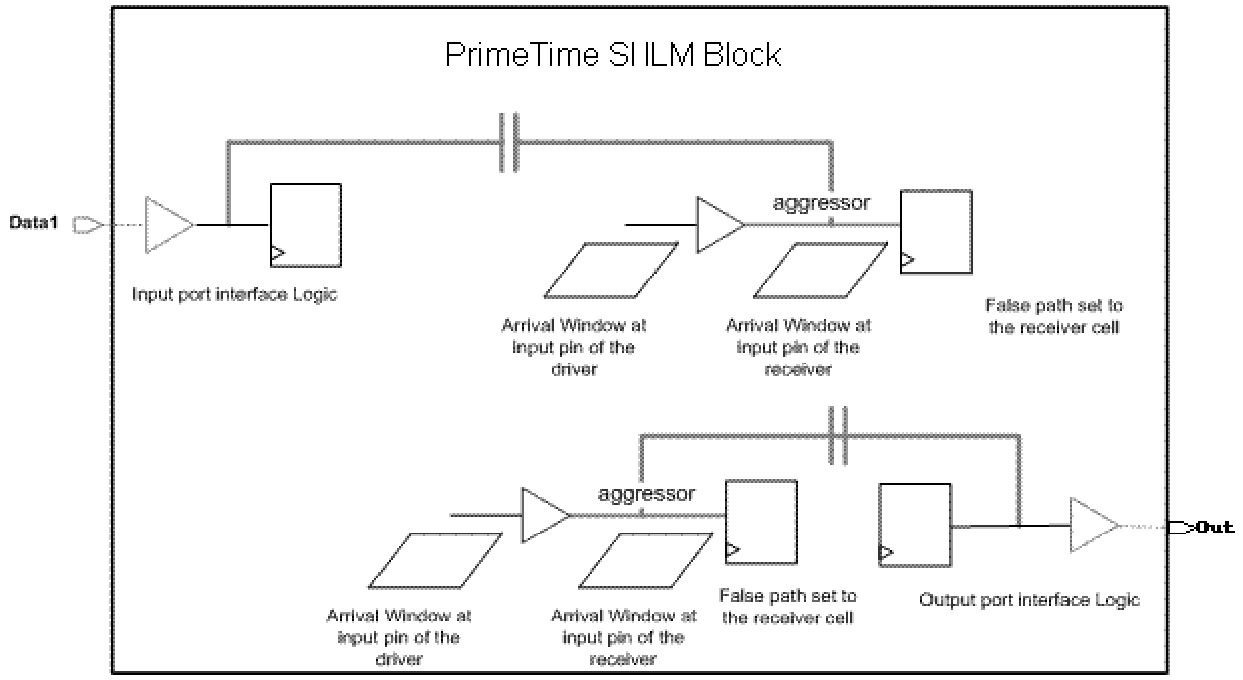
```
pt_shell> set si_enable_analysis true
pt_shell> read_parasitics -keep_capacitive_coupling
```

To include the PrimeTime SI delay analysis, use the `si_delay_pins` command. To include the PrimeTime SI noise analysis, use the `si_noise_pins` command. You can use the following command for analyzing both PrimeTime SI delay and noise analysis:

```
pt_shell> create_ilm -include {si_delay_pins si_noise_pins}
```

These internal nets that are cross-coupled to the interface nets are retained in the PrimeTime SI ILM Verilog netlist. Figure 3-5 shows the PrimeTime SI Block flow.

Figure 3-5 PrimeTime SI ILM Block


PrimeTime SI ILM Block
Data1
Input port interface Logic
aggressor
Arrival Window at input pin of the driver
Arrival Window at input pin of the receiver
False path set to the receiver cell
aggressor
Arrival Window at input pin of the driver
Arrival Window at input pin of the receiver
False path set to the receiver cell
Output port interface Logic
Out

You need to produce an additional file with the arrival and transition times for the cross coupled internal nets. You accomplish this by using the following command:

```
pt_shell> write_arrival_annotations -design
```

Therefore, to produce a PrimeTime SI ILM, the following commands from the previous section are required:

```
pt_shell> create_ilm -block_scope \
           -verification_script \
           -ignore_ports [get_port reset] \
           -include {si_delay_pins si_noise_pins} \
           -parasitic_options {spef input_port_nets constant_nets}

pt_shell> write_arrival_annotations -design
```

The subdirectory has an additional file as follows:

```
A/ilm.v              (Verilog Netlist)
A/ilm.pt.gz     (PrimeTime SI ILM arrival annotations for internal nets)
A/ilm.txt            (Contains list of added pins)
    A/ilm_inst.pt.gz (PrimeTime SI ILM constraint file)
A/ilm.spef.gz   (PrimeTime SI ILM specification file)
A/ilm.scope          (PrimeTime SI ILM block scope file)
A/ilm_verif.pt.gz (PrimeTime SI ILM verification file)
```

The ilm.pt.gz file contains commands similar to the following:

```
alias sat_r "set_annotated_transition -rise"
alias sat_f "set_annotated_transition -fall"
alias sad_r "set_input_delay -rise -add_delay"
alias sad_f "set_input_delay -fall -add_delay"

set pin [get_pin inst1/A1]
sad_r -max -reference_pin clkbuf1/A 2.18602 $pin
sad_f -max -reference_pin clkbuf1/A 2.21633 $pin
sad_r -min -reference_pin clkbuf1/A 1.62264 $pin
sad_f -min -reference_pin clkbuf1/A 1.64355 $pin
sat_r 0.0625615 -max $pin
sat_f 0.0503023 -max $pin
sat_r 0.055667 -min $pin
sat_f 0.0474351 -min $pin
```

The ilm.pt.gz file contains arrival times and transition times for the nets that have been added to the model because they have a coupling effect on I/O ports. This command file must be applied for validation and when instantiating the block at the top level.

Model Validating for PrimeTime SI ILM

The model validation for a PrimeTime SI ILM uses the same steps as those that are explained in “Model Validating Issues and Resolutions” on page 3-12.

The model validating described in “Modeling Issues” on page 3-12 is also used for PrimeTime SI. However, the ilm.pt.gz command file must be applied together with the ilm_verif.pt.gz file. For comparison, the internal aggressors need to be applied in the same arrival window on input transition. Notice, that the parasitics files contain coupling capacitors; therefore, you must use the `-keep_capacitive_coupling` option with the `read_parasitics` command. You generate timing reports and use the `compare_interface_timing` command in the same way. The flow is as follows:

```
pt_shell> set search_path \
          " . ./db ./verilog ./spef ./libraries"
pt_shell> set link_path " * my_lib.db"
pt_shell> read_verilog A/ilm.v
pt_shell> link_design block
pt_shell> source A/ilm_verif.pt.gz
pt_shell> source A/ilm.pt.gz
pt_shell> read_parasitics -keep_capacitive_coupling A/ilm.spef.gz
pt_shell> update_timing -full
pt_shell> write_interface_timing ilm_tim.rep
```

Using a PrimeTime SI ILM in a Top-Level Netlist

The steps in the previous section for using an ILM at the top-level netlist are identical. You need to add only the ilm.pt.gz file for each instance in their top-level script.

For each ILM, set the current instance to the block and source the script containing assertions and exceptions defined on the block. If any exceptions (those defined relative to clocks on ports) need to be rewritten, do so before sourcing the script. Use the following syntax:

```
pt_shell> current_instance inst1
pt_shell> source A/ilm_inst.pt.gz
pt_shell> source A/ilm.pt.gz
pt_shell> current_instance
```

The scope checking issues described in "Scope Checking Issues" on page 3-12 is also used for PrimeTime SI. You should include the `data_input_arrival and data_input_transitions` options of the `hier_scope_check_defaults` variable when scope checking. For example,

```
pt_shell> set hier_scope_check_defaults \
         clock_arrival clock_transition clock_skew_with_uncertainty \
         data_input_arrival data_input_transition
```

The `check_block_scope` command ensures that the data arrival and transitions of the inputs ports are within the conservative range set when the block was extracted. This ensures that a data arrival and transition change at the top level could not impact timing of the nets that are no longer in the ILM.

4

Extracted Timing Models

You can use extracted timing models in PrimeTime. The `extract_model` command lets you automatically generate a timing model from a gate-level netlist.

Extracted timing models are described in the following sections:

- Model Extraction Overview
- Extraction Requirements
- Extraction Process
- Limitations of Model Extraction
- Extracted Model Types
- Extraction Variables
- Preparing for Extraction
- Model Extraction Options
- Extracted Model Examples
- Extracting Clock-Gating Checks
- Extracting Constant Values
- Extracting Timing Exceptions
- Restricting the Types of Arcs Extracted

- Back-Annotated Delay and Layout Information
- Performing Model Extraction
- PrimeTime User-Defined Attributes in .lib
- Merging Extracted Models

Model Extraction Overview

The `extract_model` command generates a static timing model for the current design from its gate-level netlist. The generated model has the same timing behavior as the original netlist, and can be used in place of the original netlist in a hierarchical timing analysis.

Using an extracted timing model has these advantages:

- The generated model is usually much smaller than the original netlist. When you use extracted models in place of netlists in PrimeTime, you can significantly reduce the time needed to analyze a large design.
- Using a model in place of a netlist prevents a user from seeing the contents of the block, allowing the block to be shared while protecting the intellectual property of the block creator.

Before you can extract a timing model for a design, you must set up the context for extraction. Much of the required information depends on the options you use with the `extract_model` command.

You can control accuracy in model extraction by using the `extract_model` command options and setting the model extraction variables, as explained in "Extraction Variables" on page 4-17.

Extraction Requirements

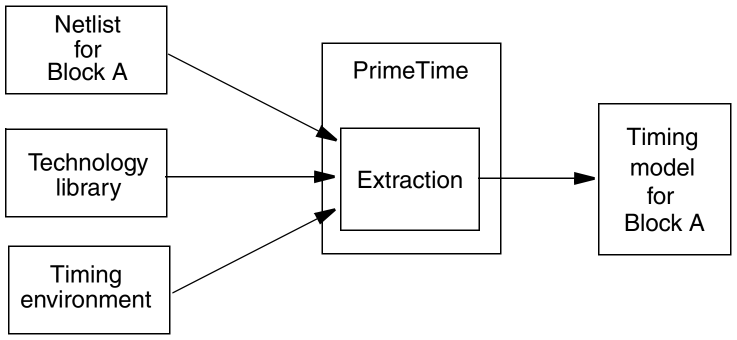
To generate an extracted timing model, you need the following:

- Block netlist
- Technology library
- Timing environment of the block (such as clocks and operating conditions)

  Note:
  Ensure that the netlist contains no timing violations before you generate a model. Use the `check_timing` or `report_constraint` command.

Figure 4-1 shows the extraction process.

Figure 4-1 Extraction Process



Extraction Process

Timing model extraction creates a timing arc for each path in the design from an input port to a register, an input port to an output port, and from a register to an output port.

Figure 4-2 and Figure 4-3 show an example of a gate-level design named “simple” and the model extracted from the netlist.

Figure 4-2 Gate-Level Netlist

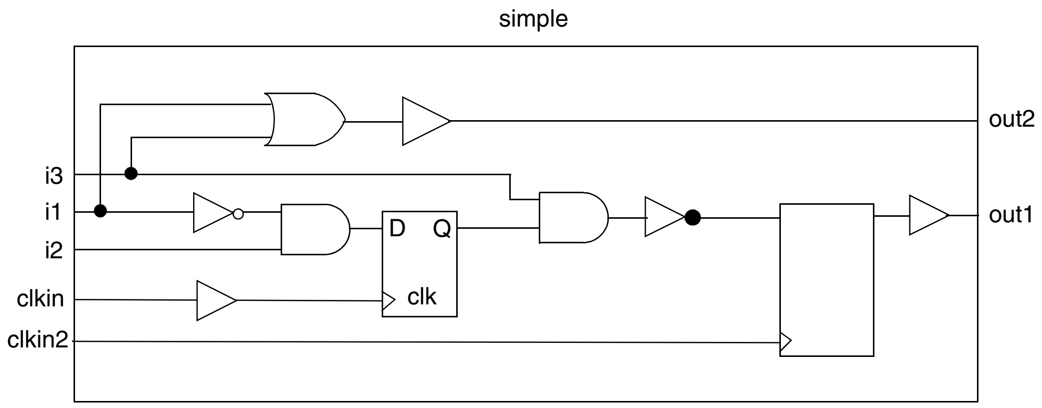
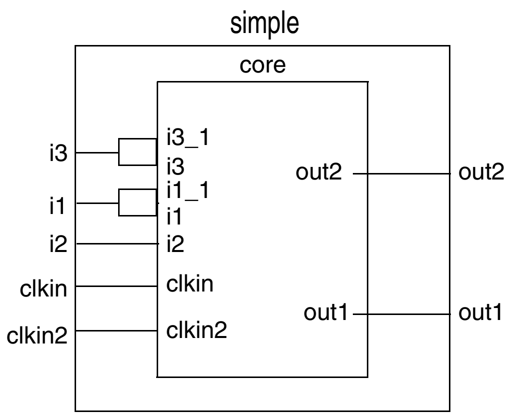
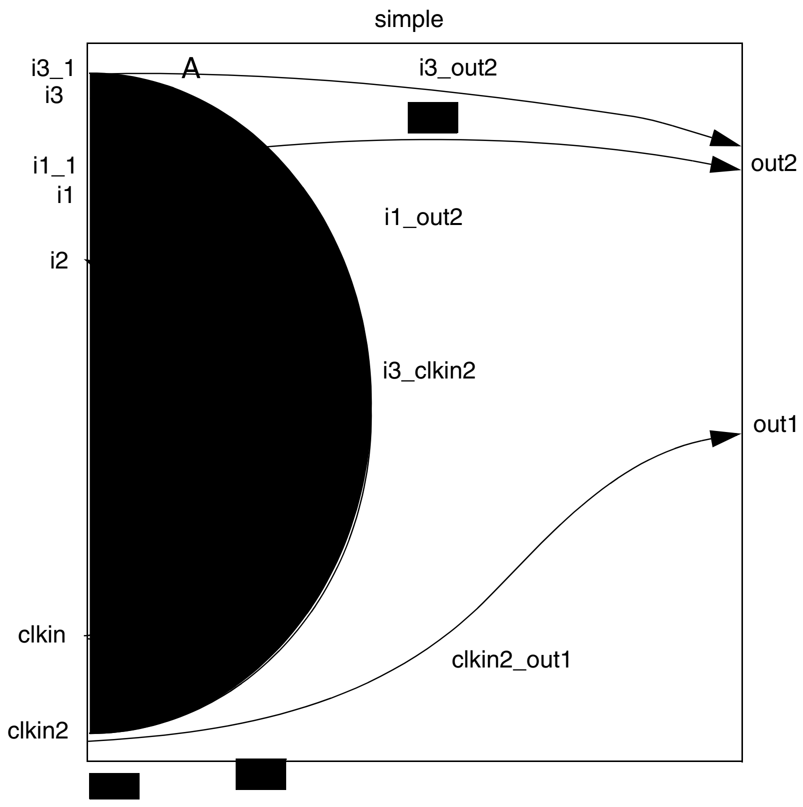


Figure 4-3 Netlist of Extracted Model



The generated timing model is another design containing a single leaf cell, as shown in Figure 4-3. The core cell is connected directly to input and output ports of the model design. This cell contains the pin-to-pin timing arcs of the extracted model. Figure 4-4 shows the timing arcs of the core cell. These arcs are extracted from the timing paths of the original design.

Figure 4-4 Timing Arcs of Core Cell



The delay data in the timing arcs is accurate for a range of operating environments. The extracted delay data does not depend on the specific values from input transition times, output capacitive loads, input arrival times, output required times, and so on. When the model is used in a design, the arc delays vary with the input transition times and output capacitive loads. This is called a “context-independent” model because it works correctly in a variety of contexts.

The characteristics of the extracted model depend on the operating conditions and wire load model in effect at the time of extraction. However, clocking conditions and external constraints do not affect the model extraction process. Commands, such as `create_clock`, `set_clock_latency`, `set_clock_uncertainty`, `set_input_delay`, and `set_output_delay`, do not affect the model extraction process, but the extracted model, when used for timing analysis, is sensitive to those commands.

Extracting Timing Paths

The following sections describe the extraction process for various types of information in the original design: nets, paths, interface latch structures, minimum pulse width and period, false paths, clock-gating checks, and back-annotated delays.

Boundary Nets

The extracted model preserves the boundary nets of the original design, thereby maintaining accuracy for computing net delays after the model design is instantiated in another design. However, if you use the `-library_cell` option with the `extract_model` command, PrimeTime does not preserve the boundary nets, but factors the boundary net delays into the model.

When you do not use the `-library_cell` option, if a boundary net in the original design has annotated parasitics in a standard format such as Detailed Standard Parasitic Format (DSPF) or SPEF, that information is saved in a separate parasitic data file along with the extracted model. For best results, re-apply the lumped boundary annotations on the model. When you use the model, you need to read in the saved parasitic information with the `read_parasitics -path` *`path_name`* command. For information about using a subdesign with annotated parasitics in a larger design, see the information on reading and writing SDF in the *PrimeTime Advanced Timing Analysis User Guide*.

Internal Nets

When performing extraction, if the internal nets are not annotated with detailed parasitics, PrimeTime computes the capacitance and resistance of internal nets. The delay values of the arcs in the model are accurate for the wire load model you set on the design before extraction. To get a model that is accurate for a different set of wire load models, set these models on the design and perform a new extraction. If internal nets are annotated with detailed parasitics (DSPF, Reduced Standard Parasitic Format (RSPF), or SPEF), PrimeTime does not use a wire load model to calculate the net capacitance and resistance. Instead, it takes the values from the detailed parasitics.

If the internal nets are back-annotated with delay or capacitance information, the back-annotated values are used instead of the computed values. For more information, see “Back-Annotated Delays” on page 4-10.

Paths From Inputs to Registers

A path from an input to a register is extracted into an equivalent setup arc and a hold arc between the input pin and the register’s clock. The setup arc captures the delay of the longest path from the particular input to all registers clocked by the same clock, plus the setup time of the register library cell. The hold arc represents the delay of the shortest path from the particular input to all register clocks, including the hold times of the register library cell.

The register’s setup and hold values are incorporated into the arc values. The setup and hold value of each arc is a function of the transition time of the input signal and the transition time of the clock signal. If an input pin fans out to registers clocked by different clocks, separate setup and hold arcs are extracted between the input pin and each clock.

Paths From Inputs to Outputs

A path from an input to an output is extracted into two delay arcs. One of the arcs represents the delay of the longest path between the input pin and the related output pin. The other arc represents the delay of the shortest path between the two pins.

The delay values for these arcs are functions of the input signal transition time and output capacitive load. The extracted arc is context-independent, resulting in different delays when used in different environments.

Paths From Registers to Outputs

A path from a register to an output is extracted into two delay arcs. One arc represents the longest path delay between the register’s clock pin and the output pin, and the other arc represents the shortest path delay between those pins. The clock-to-data output delay is included in the arc value. Different clock edges result in different extracted arcs.

Similar to input-to-output arcs, the delays of register-to-output arcs are functions of input transition times and output load capacitance. The arc delays differ depending on the environment in which they are used.

Paths From Registers to Registers

Paths from registers to registers are not extracted. For timing arc extraction, PrimeTime only traverses the interface logic.

Clock Paths

Delays and transition times of clock networks are reflected in the extracted model. However, clock latency is not included in the model. Therefore, the source latency and network latency must be specified when the timing model is used.

Interface Latch Structures

The `extract_model` command supports extraction of simple latch structures on the interface of the design (latches with fanin from a primary input or fanout to a primary output). The extracted model only preserves the borrowing behavior specified at the time of model extraction, not all possible borrowing behaviors.

Specifying Latch Borrowing

The `extract_model` command has two options that affect the extraction of latch behavior: `-context_borrow` and `-latch_level`. The `-context_borrow` option is the default behavior, which you can override by using the `-latch_level` option.

When you use the `-context_borrow` option, the model extractor identifies latches on the interface that borrow and traces through them and stops each trace at a latch that does not borrow. On the other hand, using the `-latch_level` option specifies that all latch chains at the interface of the design have a specified length and latch borrowing will definitely occur. Using the `-latch_level` option is recommended only when you are certain of the borrowing behavior of latches at the interface of the design and you are able to specify a single latch chain length for the entire design.

The total amount of borrowing should not exceed one clock cycle. If it does, you need to manually set multicycle paths in the extracted model (using the `set_multicycle_path` command) to make the model timing match the netlist timing.

How the Model Extractor Handles Latches

When you use the `extract_model` command, PrimeTime handles the latches in the design as follows:

- It traces through borrowing latches and stops when it encounters a flip-flop, port, or nonborrowing latch.
- It extracts a setup arc when an input port goes through borrowing latches to a flip-flop or nonborrowing latch.

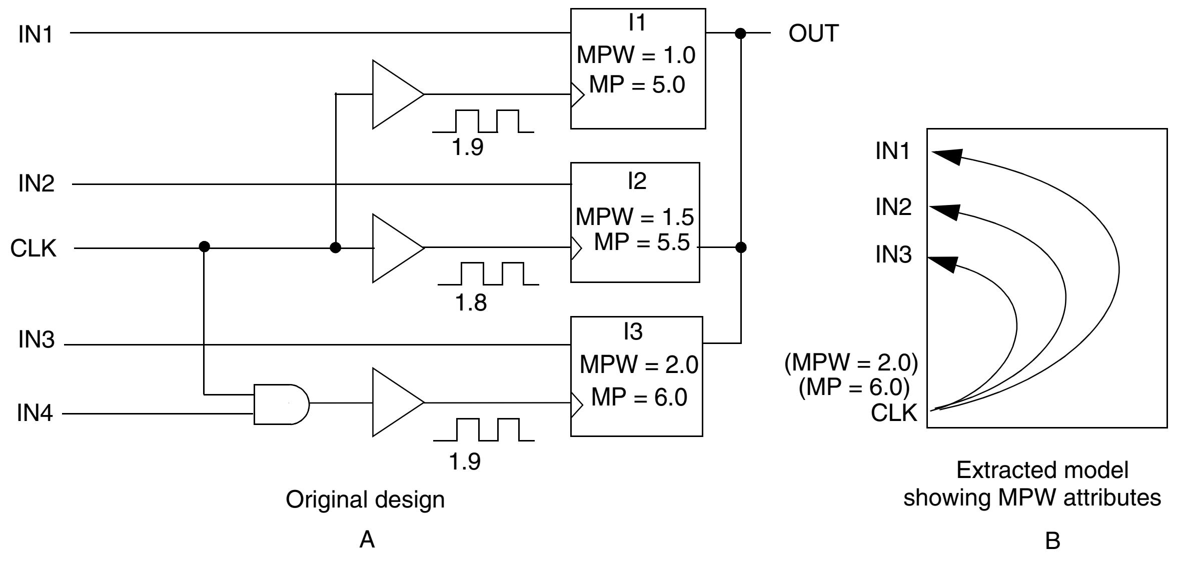
- It extracts a delay arc when an input port, or a clock connected to a flip-flop or nonborrowing latch, goes through borrowing latches to an output port.
- Neither the `-context_borrow` nor `-latch_level` option to `extract_model` causes the actual time borrowed (by a latch) to be factored into the generated setup or delay values.
- The timing of the extracted model matches that of the original netlist as long as the specified borrowing behavior at the time of model generation remains valid for the arrival times defined on input ports and clocks when the model is used. The actual time borrowed on a latch does not need to remain the same. Only the borrowing status (borrowing or not borrowing) must remain the same.

The `write_interface_timing` and `compare_interface_timing` model validation commands are useful for checking the model timing against the netlist timing, especially when there are latches on the interface. The `write_interface_timing` command traverses any borrowing latch in the netlist, which is necessary for matching the numbers reported for the model and for the netlist.

Minimum Pulse Width and Minimum Period

Minimum pulse width and minimum period constraints on cell pins are propagated as minimum pulse width and minimum period attributes on the clock pins that are present in the extracted model, as shown in Figure 4-5.

Figure 4-5 Minimum Pulse Width and Minimum Period Extraction



In the original design A, all latches have minimum pulse width and minimum period attributes on their clock pins. In the extracted model B, the minimum pulse width attributes are translated into attributes on the clock port. Only the maximum of the minimum pulse width and minimum period values are used. CLK has the MPW_HIGH = 2.0 attribute set on it. In the original design A, the input pin IN4 does not have minimum pulse width or minimum period attributes, even though it is in the fanin of the clock pin of latch I3.

The extracted timing model only looks at the minimum pulse width attributes of the library pin, not the user minimum pulse width constraints applied with the `set_min_pulse_width` command. When the extracted timing model is instantiated, you can still include the user-defined minimum pulse width and minimum period constraints in the scripts at a higher level in the hierarchy.

The delay effects (nonsymmetrical rise or fall) and slew propagation along the path are not taken into consideration. Violations can occur when the clock pulse width decreases to less than the required minimum pulse width during the course of its propagation toward the latch clock pin. These violations are not reported for the model.

For example, in Figure 4-5 the clock waveform at the input latch I3 has a width of 1.9, which is less than the required 2.0. Using 2.0 for the clock port attribute without considering the slew effects causes this minimum pulse width violation to be missed.

False Paths

The model extraction algorithm recognizes and correctly handles false paths declared with the `set_false_path` command.

Clock-Gating Checks

If your design has clock-gating logic, the model contains clock-gating checks. Depending on your environment variable settings, these checks are extracted as a pair of setup and hold arcs between the gating signal and the clock, or a pair of no-change arcs between the gating signal and the clock. For more information, see “Extracting Clock-Gating Checks” on page 4-30.

Back-Annotated Delays

If the design to be extracted is back-annotated with delay information, the model reflects these values in the extracted timing arcs. Transition time information for each arc is extracted in the same way as if the design were not back-annotated.

The delays of boundary nets, including any back-annotated delay values, are extracted if the `-library_cell` option is used. If the `-library_cell` option is not used, delays of boundary nets are not extracted, whether or not they are back-annotated. These delays are computed (or can be back-annotated) after the model is instantiated in a design and placed in context.

If the delays of boundary cells (cells connected directly to input or output ports) are back-annotated, the delays of the generated model are no longer context-independent. For example, the delays of paths from inputs are the same for different transition times of input signals. In addition, the delays of paths to outputs are not affected by the input transition time and output load.

Note:
Do not back-annotate the delays of output boundary cells if you want the output delays to change as a function of output load. Similarly, do not back-annotate the delays of input boundary cells if you expect the arc delays to depend on input transition times. To remove already-annotated information, use the `remove_annotated_delay` command.

Block Scope

An extracted timing model is context-independent (valid in different contexts) as long as the external timing conditions are within the ranges specified for block-level analysis. When the timing model is used at a higher level of hierarchy, it is no longer possible to check the internal clock-to-clock timing of the block. To ensure that the block context is valid for the original block-level timing analysis, you can check the block “scope” at the higher level.

When you generate the extracted model, you can also generate a “block scope” file containing information about the ranges of timing conditions used to verify the timing of the block. When you use the timing model in a chip-level analysis, you can have PrimeTime check the actual chip-level conditions for the block instance and verify that these conditions are within the ranges recorded in the scope file. This process ensures the validity of the original block-level analysis.

To generate the block scope file, use the `-block_scope` option of the `extract_model` command. To check the scope of the timing model during chip-level analysis, use the `check_block_scope` command. For more information, see Chapter 6, “Hierarchical Scope Checking.”

Noise Characteristics

PrimeTime SI users can analyze designs for crosstalk noise effects. An extracted timing model supports noise analysis by maintaining the noise immunity characteristics at the inputs of the extracted model, and by maintaining the steady-state resistance or I-V characteristics at the outputs of the extracted model. However, an extracted model does not maintain the noise propagation characteristics of the module. Also, any cross-coupling capacitors between the module and the external circuit are split to ground. Therefore, PrimeTime SI does not analyze any crosstalk effects between the extracted model and the external circuit.

Note:
To create timing models that can handle crosstalk analysis between modules and between different levels of hierarchy, use interface timing models instead of extracted timing models. See "Hierarchical Crosstalk Analysis" on page B-2.

To include noise characteristics in the extracted model, use the `-noise` option of the `extract_model` command. Otherwise, by default, no noise information is included in the extracted model. If you use the `-noise` option, but no noise information is available in the module netlist, then the only noise characteristics included in the extracted model are the steady-state I-V characteristics of the outputs. In that case, the model extractor estimates the output resistance from the slew and timing characteristics.

An extracted timing model can be created in two forms: a wrapper plus core or a library cell. A wrapper-plus-core model preserves the input nets, the noise immunity curves of all first-stage cells at the inputs, the output nets, and all last-stage driver cells at the outputs (including their I-V characteristics). For a library cell, the model extractor must combine parallel first-stage cells at each input to make a single noise immunity curve, and must combine parallel last-stage cells at each output to make a single driver. For this reason, a wrapper-plus-core model provides better accuracy for noise analysis than a library cell model.

An extracted timing model with noise can be created in two forms: wrapper-plus-core or library cell. A wrapper-plus-core model preserves the input nets, the noise immunity curves of all first-stage cells at the inputs, the output nets, the noise I-V curve or steady-state resistance at the outputs, and the accumulated noise at the outputs. For a library cell, the model extractor combines the inputs nets (including coupling) and all leaf-cell noise immunity curves or DC margins into a single, worst-case noise immunity curve. Also, the library cell combines the worst-case resistance of last-stage cells and net resistance at each output to make a single I-V curve or resistance for each noise region. For this reason, a wrapper-plus-core model provides better accuracy for noise analysis than a library cell model.

To create a library cell model, the model extractor considers the noise immunity curves of individual cells connected in parallel within the module netlist. It considers the worst-case (lowest) data points of multiple curves and combines them into a single worst-case curve for the library cell input. For general information about static noise analysis, see the "Static Noise Analysis" chapter of the *PrimeTime SI User Guide*.

Other Extracted Information

In addition to the timing paths, the model extraction algorithm extracts other types of information:

Mode information

Timing modes define specific modes of operation for a block, such as the read and write mode. PrimeTime extracts modes of the original design as modes for the timing arcs in the extracted model.

For a description of how to define mode information for a design, see the *PrimeTime Advanced Timing Analysis User Guide*.

Generated clocks

Generated clocks you specify in the original design become generated clocks in the extracted model. For more information about generated clocks, see the *PrimeTime Advanced Timing Analysis User Guide*.

Capacitance

PrimeTime transfers the capacitance of input and output ports of the original design to the model's extracted ports.

Design rules

PrimeTime extracts the maximum transition at input and output ports. PrimeTime reflects maximum capacitance at output ports within the design in the extracted model.

Operating conditions

The values of timing arcs in the extracted model depend on the operating conditions you set on the design when you perform the extraction.

Design name and date of the extraction

PrimeTime preserves the design name and the date of extraction in the extracted model.

Multicycle paths

PrimeTime extracts multicycle paths and writes that information to a script containing a set of `set_multicycle_path commands` that can be applied to the extracted model.

Three-state arcs

Three-state arcs ending at output ports are included in the extracted model.

Preset and clear delay arcs

Preset and clear arcs are extracted if enabled. For more information, see "Extraction Variables" on page 4-17.

Clock latency arcs

Clock latency arcs are extracted if enabled (see "Extraction Variables" on page 4-17). Clock latency arcs are used by tools, such as Physical Compiler to compensate and balance clock tree skews at chip level. They are also reported by the `report_clock_timing` command in PrimeTime.

Limitations of Model Extraction

This section provides general guidelines and lists the general limitations of model extraction in PrimeTime, the limitations of using extracted models in Design Compiler, and the limitations of extracted models in .lib format.

Observe the following model extraction guidelines:

- Complex latch structures on the design interface are not supported. For more information, see “Interface Latch Structures” on page 4-8.
- Clocks with multiple-source pins or ports are not supported.
- Avoid using the `-remove_internal_arcs` and `-latch_level` options.

The following environment variables affect model extraction:

```
extract_model_enable_report_delay_calculation
extract_model_gating_as_nochange
extract_model_status_level
extract_model_num_capacitance_points
extract_model_num_clock_transition_points
extract_model_num_data_transition_points
extract_model_capacitance_limit
extract_model_clock_transition_limit
extract_model_data_transition_limit
extract_model_with_clock_latency_arcs
timing_clock_gating_propagate_enable
timing_disable_clock_gating_checks
timing_enable_preset_clear_arcs
```

For a description of the extracted model impact, see the specific variable man page.

The following limitations apply to model extraction:

- Minimum and maximum delay constraints

  Paths in the original design that have a minimum or a maximum delay constraint set on them are treated by extraction as follows:

  - If you set the constraint on a timing path, PrimeTime ignores the constraint in the model.
  - If you set the constraint on a pin of a combinational cell along a path, PrimeTime ignores the timing paths that pass through this pin. These paths do not exist in the extracted model.
- User data checks that are applied with the `set_data_check` command are not supported.

For these reasons, it is recommended that you remove all minimum and maximum delay constraints from the design before you perform extraction.

Input and output delays

PrimeTime ignores input and output delays set on design ports when it extracts a model, unless your design contains transparent latches and you specify the `-context_borrow` option of the `extract_model` command.

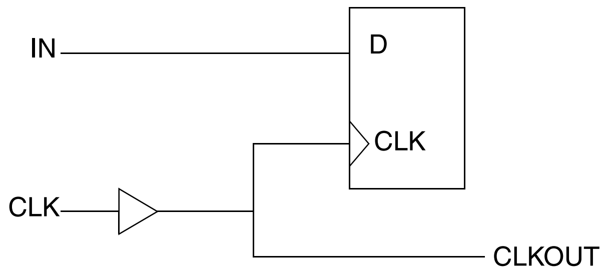
Input or output delay values set on pins of combinational cells along a timing path cause the paths that pass through these pins to be ignored in the extracted model. It is recommended that you remove input and output delay values set on internal paths (or set on objects other than ports) from the design before performing an extraction.

Extraction of load-dependent clock networks

If the delay of a clock to register path in the original design depends on the capacitive load on an output port, the delay of the extracted setup arc to the register in the model is independent of the output load.

For example, consider the design shown in Figure 4-6. In this design, the setup time from the IN input relative to the CLK clock depends on the load on the CLKOUT output. This occurs because the delay of the clock network depends on the load on the CLKOUT output and because the transition time at the register's clock pin depends on the output load.

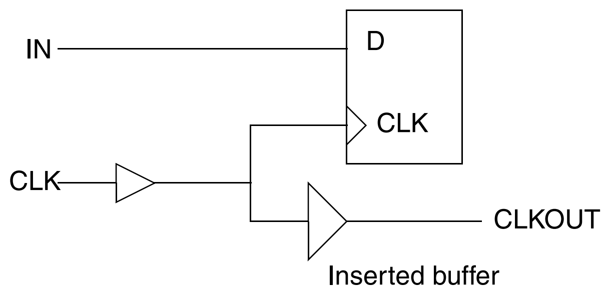
Figure 4-6 Load-Dependent Clock Network



In the extracted model, the setup time from input IN to input CLK is independent of the load on output CLKOUT. However, the extracted CLK delay from input to CLKOUT remains load dependent.

To avoid inaccuracies in the extracted model, isolate the delay of the clock network inside the design to be modeled from the changes in the environment by inserting an output buffer, as shown in Figure 4-7.

Figure 4-7 Isolating the Delay of the Clock Network



Limitations of Extracted Models in Design Compiler

Design Compiler does not support all the features of extracted timing models. Specifically, Design Compiler ignores mode information. All modes are considered enabled.

Limitations of Extracted Models in .lib Format

You can optionally specify .lib format for the extracted model. This format is useful for exporting the timing model to an external tool. The following limitations apply to a model extracted in this format:

- Like extracted models in other formats, the .lib model captures only the timing behavior, not the logic, of the original netlist.

Extracted Model Types

The `extract_model` command can create two different types of timing models: a library cell or a wrapper with a core cell inside. A library cell serves as a port-to-port replacement for the design being modeled, with the boundary net capacitances approximated inside the model. A wrapper-and-core model consists of a central core cell surrounded by a wrapper design that preserves the original boundary nets.

The type of model created by `extract_model` depends on the `-library_cell` switch. If the `-library_cell` switch is present, PrimeTime creates a library cell. The name assigned to the new cell is the same as the design name.

If the `-library_cell` switch is absent, PrimeTime creates a wrapper design with a core cell. The `-format` option controls the output format of the core cell. However, PrimeTime always writes out the wrapper design in .db format, irrespective of the `-format` setting. The name of the core cell is *design_name*_core. The name of the wrapper design is *output*.db, where *output* is the value specified by the `-output` option.

You can optionally create a test design containing an instance of the extracted library cell. To do this, use the `-test_design` switch along with `-library_cell` in the `extract_model` command. The test design is named *design_name*_test and the output file is named *output*_test.db. PrimeTime does not write parasitics to the test design.

Extraction Variables

Several environment variables control the model accuracy and other aspects of timing model extraction.To attain the level of accuracy and complexity you want, consider each variable setting.

Table 4-1 lists the environment variables and summarizes how they affect the `extract_model` command. For more information on any particular variable, see the man page.

Table 4-1 Extraction Environment Variables

Extraction environment variables	Effect on extract_model command
`extract_model_enable_report_delay_calculation`	Specifies whether to allow the final user of the extracted model to get timing information using the `report_delay_calculation` command. Set this variable to `false` if the timing calculations for the model are proprietary.
`extract_model_capacitance_limit`	Specifies the maximum load capacitance on outputs. Model extraction characterizes the timing within this specified limit.
`extract_model_clock_transition_limit`	For clock input ports, specifies the maximum input port transition time. Model extraction characterizes the timing within this specified limit.
`extract_model_data_transition_limit`	For data input ports, specifies the maximum input port transition time. Model extraction characterizes the timing within this specified limit.
`extract_model_num_capacitance_points`	Specifies the number of capacitance data points used in the delay table for each arc.

Table 4-1 Extraction Environment Variables (Continued)

Extraction environment variables	Effect on extract_model command
`extract_model_num_clock_transition_points`	Specifies the number of transition time data points used to make the delay table for each clock arc.
`extract_model_num_data_transition_points`	Specifies the number of transition data points used to make the delay table for each data arc.
`extract_model_with_clock_latency_arcs`	Specifies whether to enable modeling of clock tree insertion delay timing arcs in the extracted model.
`extract_model_status_level`	Specifies the amount of detail provided in model extraction progress messages.
`timing_enable_preset_clear_arcs`	Specifies whether to enable or disable preset and clear timing arcs.
`extract_model_gating_as_nochange`	Extracts clock-gating setup and hold checks as corresponding no-change arcs.
`timing_disable_clock_gating_checks`	When set to true, disables all clock-gating checks and does not extract them to the model.
`timing_clock_gating_propagate_enable`	Determines whether data signals that gate a clock are propagated past the gating logic.

Variable Setting Guidelines

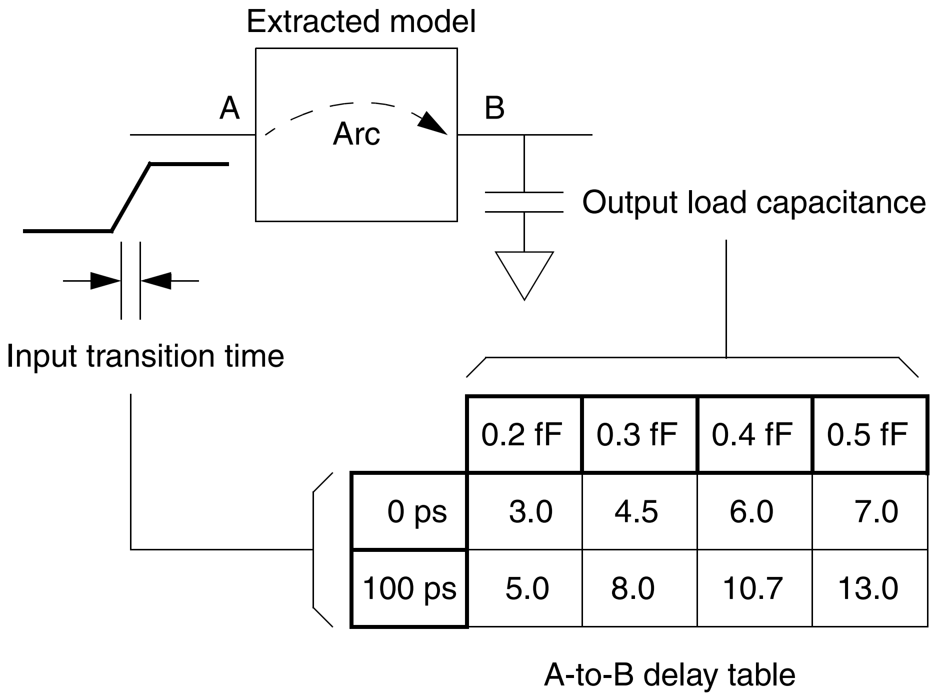
To generate accurate timing models, observe these guidelines:

- If the design contains latches on the interface, run `extract_model` with the `-context_borrow` option.
- Avoid the `-remove_internal_arcs` and `-latch_level` options of the `extract_model` command because they compromise accuracy.

Delay Table Generation

When you use the `extract_model` command, PrimeTime extracts a set of timing arcs and creates a delay table for each arc. A delay table is a matrix that specifies the arc delay value as a function of the input transition time and the output load capacitance. For example, the delay arc shown in Figure 4-8 uses a delay table like the 2-by-4 table shown in the figure. The amount of delay for the arc depends on the input transition time and output load capacitance.

Figure 4-8 Delay Table Example



	0.2 fF	0.3 fF	0.4 fF	0.5 fF
0 ps	3.0	4.5	6.0	7.0
100 ps	5.0	8.0	10.7	13.0

When you use the extracted model in a design, PrimeTime calculates the arc delay according to the context where the model is being used. The delay table makes the model accurate for the range of input transition times and output loads defined in the table.

For transition times or output loads at intermediate values within the table, PrimeTime uses interpolation between the table points to estimate the delay. A table with a larger number of entries (for example, a 10-by-10 matrix) provides better accuracy at the cost of more memory and CPU time for model generation.

For transition times and output loads outside of the ranges defined in the table, PrimeTime uses extrapolation to estimate the delay. A table that spans a larger range of transition time or load capacitance provides better accuracy for a larger range of conditions, but wastes memory and CPU resources if the range is larger than necessary to accommodate actual usage conditions.

You can control the number of data points and the range of conditions specified in delay tables of models generated by model extraction. To do so, set the following variables:

```
extract_model_num_capacitance_points
extract_model_num_clock_transition_points
extract_model_num_data_transition_points

extract_model_capacitance_limit
extract_model_clock_transition_limit
```

`extract_model_data_transition_limit`

The number or `*_points` variables control the number of data points in the generated delay tables. By default, these variables are set to 5, resulting in 5-by-5 delay tables. The `*_limit` variables control the range of capacitance and transition times covered in the generated delay tables. The default settings are 64 pf for the capacitance limit and 5 ns for the transition time limits. For more information, see the man page for each variable.

Preparing for Extraction

To prepare your design for extraction, consider the following steps:

- Set the model extraction environment variables
- Set the operating conditions for the extracted model
- Define the wire load models
- Identify pins being used as clocks in your design
- Define the clocks in the current design
- Prepare for extraction of latch behavior
- Identify false paths in your design
- Remove input or output delays set on objects that are not input or output ports
- Set the design modes, if any, for extraction
- Check the timing
- Prepare for crosstalk analysis

The sections that follow describe how to perform these tasks.

Setting the Extraction Variables

Several environment variables affect the extraction process, as explained in "Extraction Variables" on page 4-17. Set them as needed by using one of these methods:

- Specify variables and their corresponding values individually. For example, enter

```
pt_shell> set extract_model_data_transition_limit 2.0
pt_shell> set extract_model_capacitance_limit 5.0
```

- Specify a variable and its value in a script file, then source the batch file. For example, enter

```
pt_shell> source script_file
```

- Specify the variable and its value in the .synopsys_pt.setup (PrimeTime setup) file.

Setting Operating Conditions

The operating conditions for vendor libraries vary. For example, some conditions might be nominal, worst-case commercial (WCCOM), and best-case commercial (BCCOM).

Model extraction uses the current single operating condition or current minimum and maximum operating conditions. For example, for a single operating condition, enter:

```
pt_shell> set_operating_conditions WCCOM
pt_shell> extract_model -output model
```

To create a single model that has setup and hold arcs using maximum and minimum operating conditions, use commands like the following:

```
pt_shell> set_operating_conditions -min BEST_COND \
          -max WORST_COND -lib your_lib -analysis_type bc_wc
pt_shell> extract_model -output file_name -format db
```

To create a single conservative model using on-chip variation to calculate the timing arcs, set the `-analysis_type` option to `on_chip_variation` instead of `bc_wc`.

You can also extract two different models at two different operating conditions, and then invoke the two models for min-max analysis. For example, you can use a script similar to the following:

```
set link_path "* your_library.db"
read_db BLOCK.db
link BLOCK
source constraint.pt
read_parasitics ...
read_sdf ...
set_operating conditions WORST_COND -lib your_library.db
extract_model -format db -output CORE_max
set_operating conditions BEST_COND -lib your_library.db
extract_model -format db -output CORE_min
remove_design -all
```

Then you can use instances of the core wrapper CORE_max.db in your design, and invoke the two models for min-max analysis using a script similar to the following:

```
set link_path "* CORE_max_lib.db your_library.db"
read_db CHIP.v
set_min_library CORE_max_lib.db -min_version \
  CORE_min_lib.db
link CHIP
```

. . .

Defining Wire Load Models

Defining the wire load models is a prerequisite for extraction unless the delays of all internal nets in the design have been back-annotated. The `extract_model` command uses wire load models to calculate the net capacitance and net delays in the extracted paths.

Use the `set_wire_load_model` and `set_wire_load_mode` commands to define one or more wire load models.

For example, enter the following commands to specify a 20-by-20 wire load model for the top design and a 05-by-05 wire load model for block u4.

```
pt_shell> set_wire_load_mode enclosed
pt_shell> set_wire_load_model -name 20x20 -lib class
pt_shell> set_wire_load_model -name 05x05 -lib class U4
```

The delay values of the arcs in the model are accurate for the wire load model you set on the design before extraction. To get an accurate model for a different set of wire load models, set wire load models on the design and perform a new extraction.

Defining Clocks

Defining all clocks in the current design is a prerequisite for extraction. You identify pins as clocks by using the `create_clock` or `create_generated_clock` command.

The `extract_model` command uses the clock period and input delay information under the following conditions:

- If you use the `-context_borrow` option, the model extractor uses the clock information to determine which transparent latches are borrowing.
- If you set a multicycle path definition at a point that the model extractor cannot easily move to the design boundary, the clock period determines which path is considered the critical path.

Typically, you place the created clocks on input ports of your design. Clocks created with the `create_clock` command on pins inside the design produce internal clocks in the generated model that require special attention. That is, you must specify the clocks on the pins in the model again; however, generated clocks referenced to boundary ports are extracted as expected and require no extra setup when using the extracted model.

The model extractor does not support multiple-source clocks. A multiple-source clock is a clock that is applied to more than one point in the design. For example, model extraction cannot work with a clock defined as follows:

```
pt_shell> create_clock -name CLK -period 10 \
            [get_pins {Clk1 Clk2}]
```

The model extractor also does not support multiple clocks applied to the same point in the design. For example, model extraction cannot work with two clocks defined as follows:

```
pt_shell> create_clock -name CK1 -period 10 [get_pins Clk1]
pt_shell> create_clock -name CK2 -period 12 [get_pins Clk1]
```

Setting Up Latch Extraction

If the design you want to extract contains transparent latches and you want the generated model to exhibit time-borrowing behavior, you can use the `-context_borrow` and `-latch_level` options of the `extract_model` command to specify the method of latch extraction.

Observe the following guidelines when you use the `-context_borrow` and `-latch_level` options:

- If your design has no transparent latches or has transparent latches with no time borrowing, these option settings have no effect on the results.
- With a latch-based design, do not use the `-latch_level` option together with the `-context_borrow` option.
- Setting `-latch_level` option to 0 or 1 is preferred. Using `-context_borrow` or `-latch_level 2` or above can cause some mismatches in model validation. Most of these mismatches are because the total amount of borrowing has exceeded one clock cycle, and it is required to manually set multicycle paths in the model. When performing model validation, you should also use the corresponding `-latch_level` option in the `write_interface_timing` command.
- If the extracted model is to be used in an environment where the clock waveforms or input arrival times (or both) are expected to change drastically, do not use the `extract_model` command. Instead, generate an interface logic model as described in Chapter 3, "Interface Logic Models."

For more information about extraction of latch behavior, see "Interface Latch Structures" on page 4-8.

Identifying False Paths

Before you perform model extraction, define false paths in your design by using the `set_false_path` command. Paths defined as false are not extracted into timing arcs by the `extract_model` command. For example, enter the following command to define the path between input IN4 and output OUT1 as a false path.

```
pt_shell> set_false_path -from IN4 -to OUT1
```

For more information about the `set_false_path` command, see the specifying timing exceptions details in the *PrimeTime Fundamentals User Guide*.

Removing Internal Input and Output Delays

Remove input or output delays set on design objects that are not input or output ports. These objects are usually input or output pins of cells in the design. To remove delays, remove the commands from the original script that set the delays.

Controlling Mode Information in the Extracted Model

A complex design might have several modes of operation with different timing characteristics for each mode. For example, a microcontroller might be in setup mode or in monitor mode. A complex design might also have components that themselves have different modes of operation. A common example is an on-chip RAM, which has a read mode and a write mode.

The `extract_model` command creates a model that reflects the current mode settings of the design. The generated model itself does not have modes, but contains timing arcs for all the paths that `report_timing` detects with the modes at their current settings.

If you extract multiple timing models from a design operating under different modes, different conditions set with case analysis, or different exception settings, you can merge those models into a single model that has operating modes. For more information, see “Merging Extracted Models” on page 4-42.

Performing Timing Checks

To check your design for potential timing problems before you extract, use the `check_timing` command. Because problems reported by the `check_timing` command affect the generated model, fix these problems before you generate the timing model. For more information about the `check_timing` command, see the *PrimeTime Fundamentals User Guide*.

Preparing for Crosstalk Analysis

PrimeTime SI is an optional tool that adds crosstalk analysis capabilities to PrimeTime. If you are a PrimeTime SI user and crosstalk analysis is enabled, you can have the `extract_model` command consider crosstalk effects when it calculates the timing arcs for the extracted model.

The model extractor cannot calculate the crosstalk-induced delays for all combinations of input transitions times for victim and aggressor nets. Instead, it accounts for crosstalk effects in a conservative manner. You specify a range of input delays and slews for the model prior to extraction. When you use the `update_timing` command, PrimeTime SI uses this information to calculate the worst-case changes in delay and slew that can result from crosstalk. The model extractor then adds the calculated changes to the extracted timing arcs.

This is the procedure for including crosstalk effects during model extraction:

1. Use the `set_input_delay` command with both the `-min` and `-max` options to specify the worst-case timing window for each input, given the current clock configuration.
2. Use the `set_input_transition` command with both the `-min` and `-max` options to specify the worst-case slew change for each input.
3. With PrimeTime SI crosstalk analysis enabled (by setting the `si_enable_analyis` variable to `true`) and with the design back-annotated with cross-coupling capacitors, perform a crosstalk analysis with the `update_timing` command. PrimeTime SI calculates the worst-case delay changes and slew changes under the specified conditions using on-chip variation analysis.
4. Run the `extract_model` command in the usual manner. PrimeTime performs model extraction without crosstalk analysis, then adds the fixed delta delay and delta slew values to the resulting timing arc values.

In step 4, PrimeTime only adds crosstalk values that make the model more conservative. For example, it adds a delta delay value to a maximum-delay arc only if the delta delay is positive, or to a minimum-delay arc only if the delta delay is negative.

The transition times you define with the `set_input_transition` command are used to calculate the delta delay values. These transition times are preserved in the extracted model as a set of design rules. When you use the extracted model, if a transition time at an input port of the model is outside of the defined range, you receive a warning message.

Model Extraction Options

The `extract_model` command has a `-format` option, which can be set to any combination of `db`or `lib` to generate models in .db format or .lib format. The .db format can be used directly by most Synopsys tools. Use the .lib format when you want to be able to read and understand the timing arcs contained in the model, or for compatibility with a third-party tool that can read .lib files. The .lib model can be compiled by Library Compiler to get a .db file. The `extract_model` command provides two command options that control the level of detail and accuracy of the extracted model:

- `-library_cell`, which generates the model as a library cell rather than a wrapper and core
- `-remove_internal_arcs`, which generates the model with internal arcs and timing points removed

By default, the generated model is a design containing a single core cell. The model preserves all the boundary nets in the original design and is the more accurate type of model.

The `-library_cell` option causesthe `extract_model` command to generate the model as a library cell instead of a wrapper design and core. In this case, all boundary net delays are added to the arcs in the model.

The library cell model is simpler than the detailed model. However, the boundary net delays are not as accurate. In the case where outputs are shorted, the dependence of output delay on the capacitance of other outputs is lost.

You can use the `-test_design` and `-library_cell` options together to have the model extractor generate a test design that instantiates the model. You can link the model to this design and obtain timing reports using the `report_timing` command. This design is not part of the model. It is provided as a convenience for testing the model after extraction.

If you have a PrimeTime PX license, you can use the `-power` option to generate a power model. By using this option, you can store power data for IPs and large blocks in a model that you can instantiate. This option performs an implicit `update_power`, if necessary; therefore, you must set the `power_enable_analysis` variable to `true` before using the `extract_model -power` option. For more information about this option, see the *PrimeTime PX User Guide* and the `extract_model` command man page.

Extracted Model Examples

Figure 4-9 shows a gate-level netlist for a design called simple.

Figure 4-9 Gate-Level Netlist

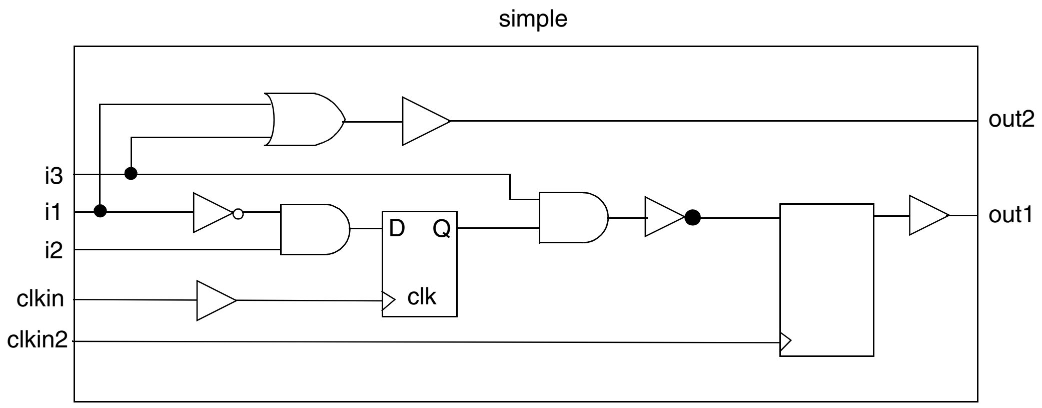


Figure 4-10 shows the extracted model when you do not use the `-library_cell` option. The generated model is a design called simple. This design contains an instance of a single cell called simple_core, which contains all the timing arcs in the model.

Figure 4-10 Extracted Model Without -library_cell

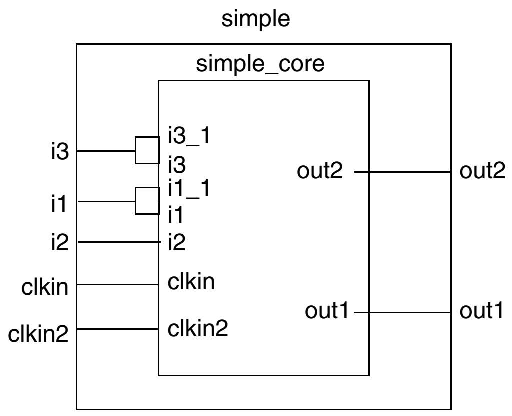


Figure 4-11 shows the core cell and its timing arcs.

Figure 4-11 Core Cell and Timing Arcs

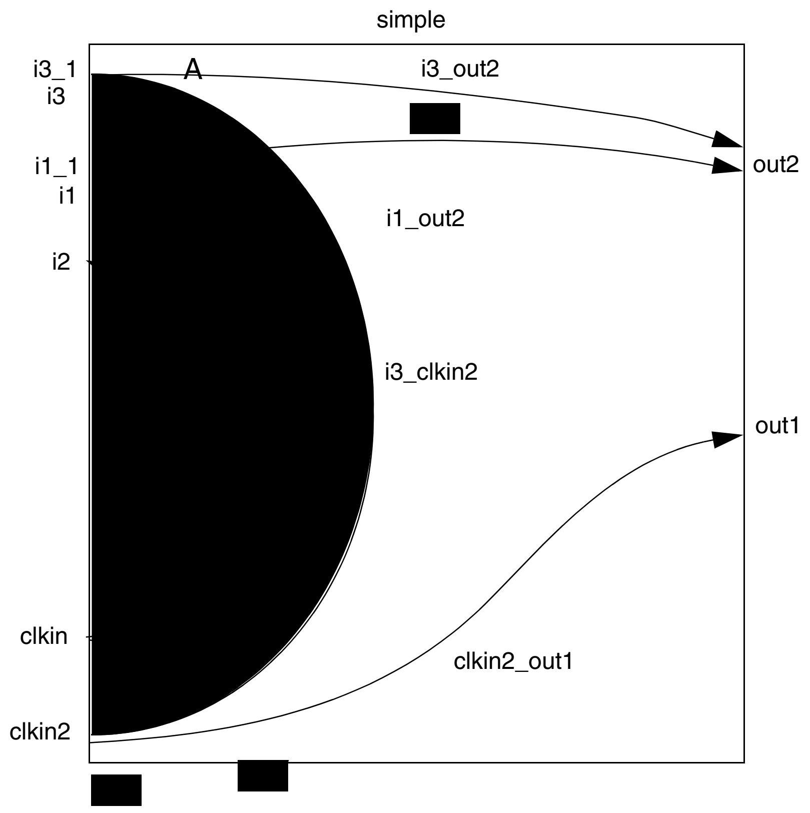


Figure 4-12 shows the model generated when you specify the `-library_cell` option. The model is a library cell called simple. This cell has the same input ports and output ports as the model. It also contains all the timing arcs. This model has no boundary nets. All boundary net delays are lumped into the generated model.

Figure 4-12 Model Extracted With -library_cell

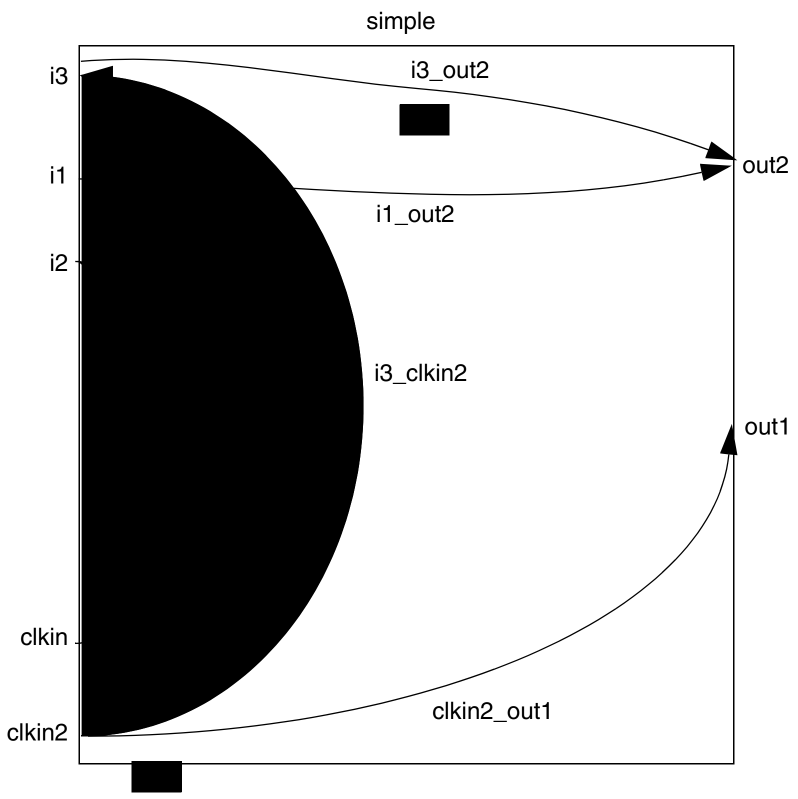
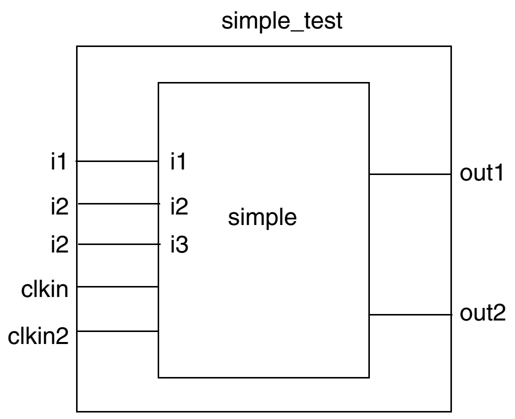


Figure 4-13 shows a test design generated when you specify the `-test_design` option. The design is called simple_test, which instantiates the model. This design is generated for your convenience in obtaining model and timing reports for the model if the model is generated as a library cell.

Figure 4-13 Test Design That Instantiates the Model



Extracting Clock-Gating Checks

The clock-gating setup and hold constraints, which are between the clock-gating pin and the clock, are converted to setup and hold checks between the primary input pin and input clock pin, and then written out to the model.

The following items can affect the way clock-gating checks are extracted:

Clock propagation

You can select whether to enable or disable clock delay propagation by using the `set_propagated_clock` and `remove_propagated_clock` commands. For more information, see the *PrimeTime Advanced Timing Analysis User Guide*.

Clock-gating checks

You can use the `set_clock_gating_check` command to add clock-gating setup or hold checks of any desired value to any design object. See the information about clock-gating setup and hold checks in the *PrimeTime Advanced Timing Analysis User Guide*.

Enabling or disabling gating checks

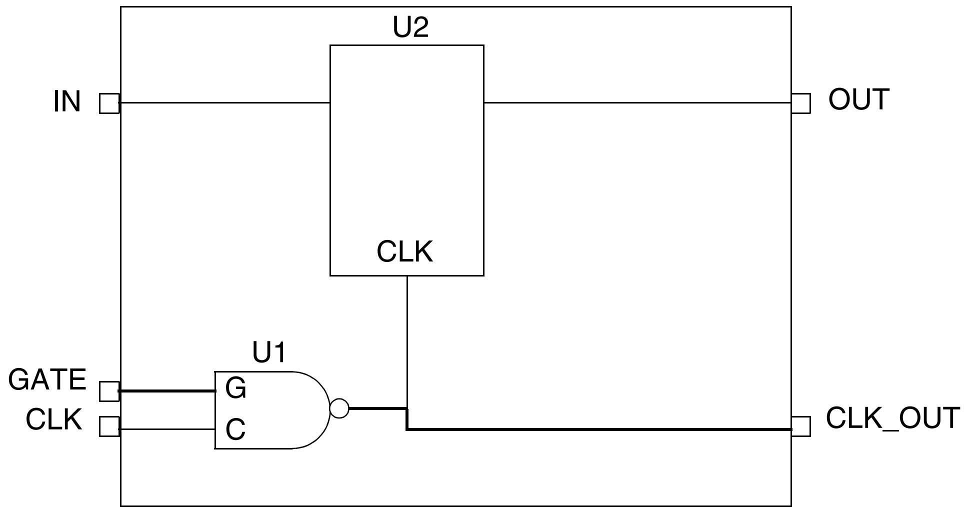
Extraction of clock-gating checks can be enabled or disabled with the `timing_disable_clock_gating_checks` variable prior to extraction.

Extraction of Combinational Paths Through Gating Logic

Combinational paths that start at input or inout ports, go through clock-gating logic, and end at output or input ports are extracted in the model.

In Figure 4-14, the path from GATE through U1/G to CLK_OUT is a combinational path in the clock-gating network and is extracted if the `timing_clock_gating_propagate_enable` variable is set to `true` (the default).

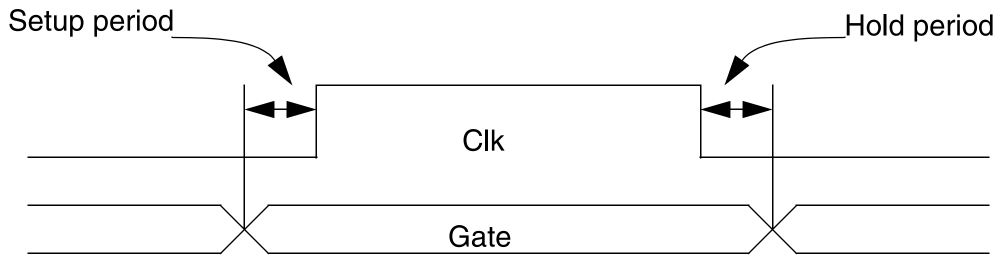
Figure 4-14 Combinational Path Through Clock-Gating Logic



Extraction of Clock-Gating Checks As No-Change Arcs

Clock-gating setup and hold checks can be grouped and merged to form a no-change constraint arc. The no-change arc is a signal check relative to the width of the clock pulse. PrimeTime establishes a setup period before the start of the clock pulse and a hold period after the clock pulse (see Figure 4-15).

Figure 4-15 No-Change Arc



A clock-gating setup check can be combined with its corresponding clock-gating hold check to produce two no-change arcs. One arc is the no-change condition with the clock-gating signal low during the clock pulse, the other is the gating signal high during the clock pulse.

The `extract_model_gating_as_nochange` variable, when set to true, causes the model extractor to convert all clock-gating setup and hold arcs into no-change arcs. When set to `false` (the default), the clock-gating checks are represented as a clock-gating constraint.

For example, if the variable is set to `true`, and if the clock pulse leading edge is rising and the trailing edge is falling, the no-change arcs produced are as follows:

- NOCHANGE_HIGH_HIGH
  Rise table taken from the setup arc rise table. Fall table taken from the hold arc fall table. See Figure 4-16.
- NOCHANGE_LOW_HIGH
  Rise table taken from the hold arc rise table. Fall table taken from the setup arc fall table. See Figure 4-17.

Figure 4-16 NOCHANGE_HIGH_HIGH Arc

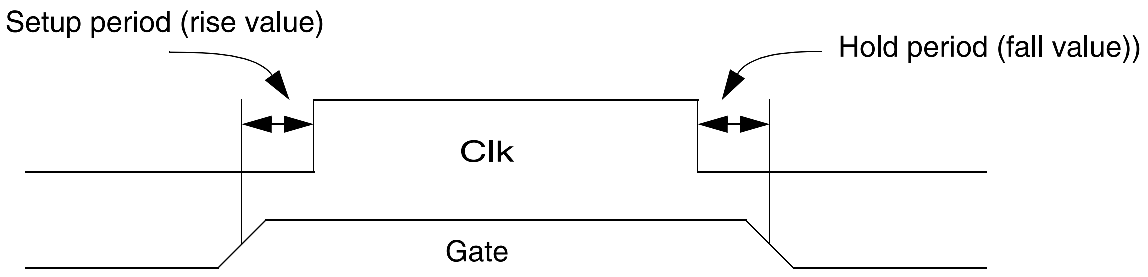
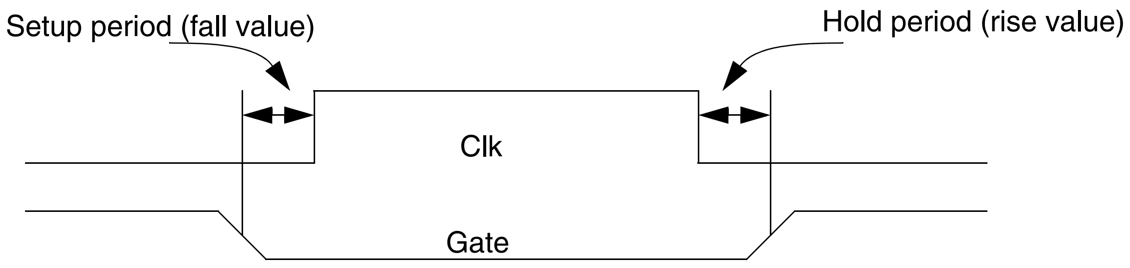


Figure 4-17 NOCHANGE_LOW_HIGH Arc



Extracting Constant Values

If the output ports are driven to a constant value, this value is preserved in the model. If the extracted model is used in a design, the constant value is propagated into its fanout logic.

Extracting Timing Exceptions

When you specify a timing exception for a path, PrimeTime does not treat that path as an ordinary single-cycle path. The most common type of exception is a false path. Other exception types are multicycle, `max_delay`, and `min_delay` paths. You can also use mode analysis (`define_design_mode`, `set_mode`) to control the timing analysis applied to certain defined paths.

The following rules apply to conflicts between timing exceptions and mode analysis:

- If a conflict arises between a specified false path and a moded endpoint, the false path specification prevails.
- If a conflict arises between a specified multicycle path and a moded endpoint, the mode is propagated.

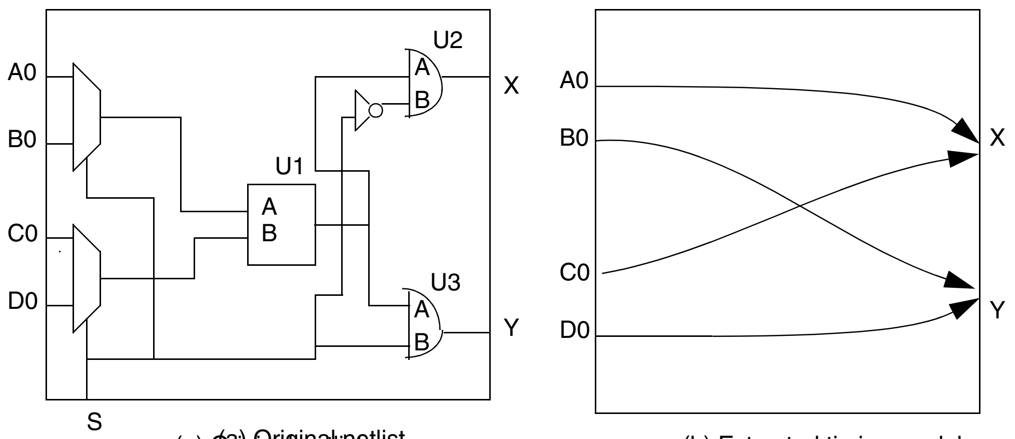
PrimeTime extracts paths with timing exceptions automatically, including exceptions you specify using the `-through` option of the exception-setting commands.

PrimeTime supports the following exceptions:

- False paths
- Multicycle paths
- Moded paths

Figure 4-18 shows an original design and an extracted timing model when false paths are present.

Figure 4-18 Original Design With False Paths and Extracted Model



(a) Original netlist (a) Original netlist

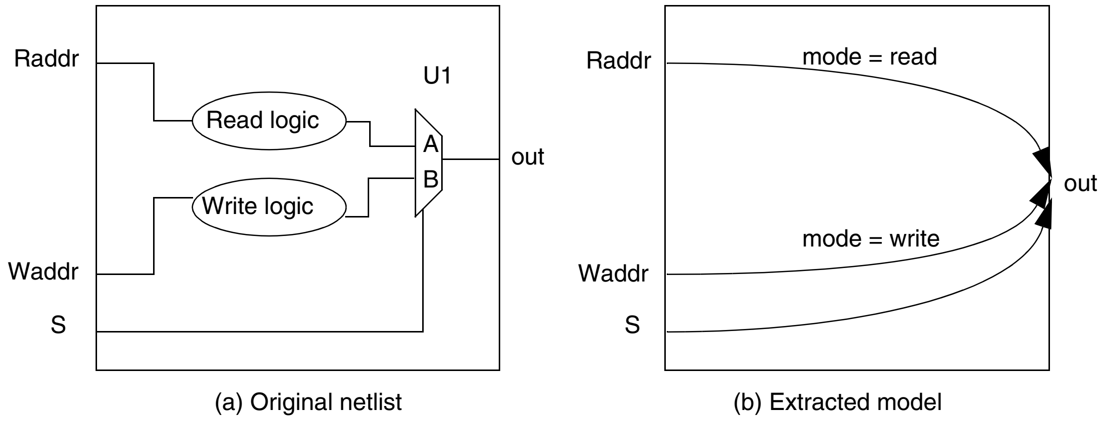
(b) Extracted timing model

If you set the following false paths in the design shown in (a), the extracted model represented by (b) is the result.

```
pt_shell> set_false_path -from {A0 C0} -through U3/A
pt_shell> set_false_path -from {B0 D0} -through U2/A
```

There are timing arcs from A0 and C0 to X, but none from A0 and C0 to Y. Similarly, there are timing arcs from B0 and D0 to Y, but none from B0 and D0 to X. Figure 4-19 shows a design in which modes are defined with the `-through` option, together with the extracted timing model.

Figure 4-19 Extracting Modes With the -through Option



If you define the following modes for the netlist shown in (a), the result is the set of arcs represented in (b).

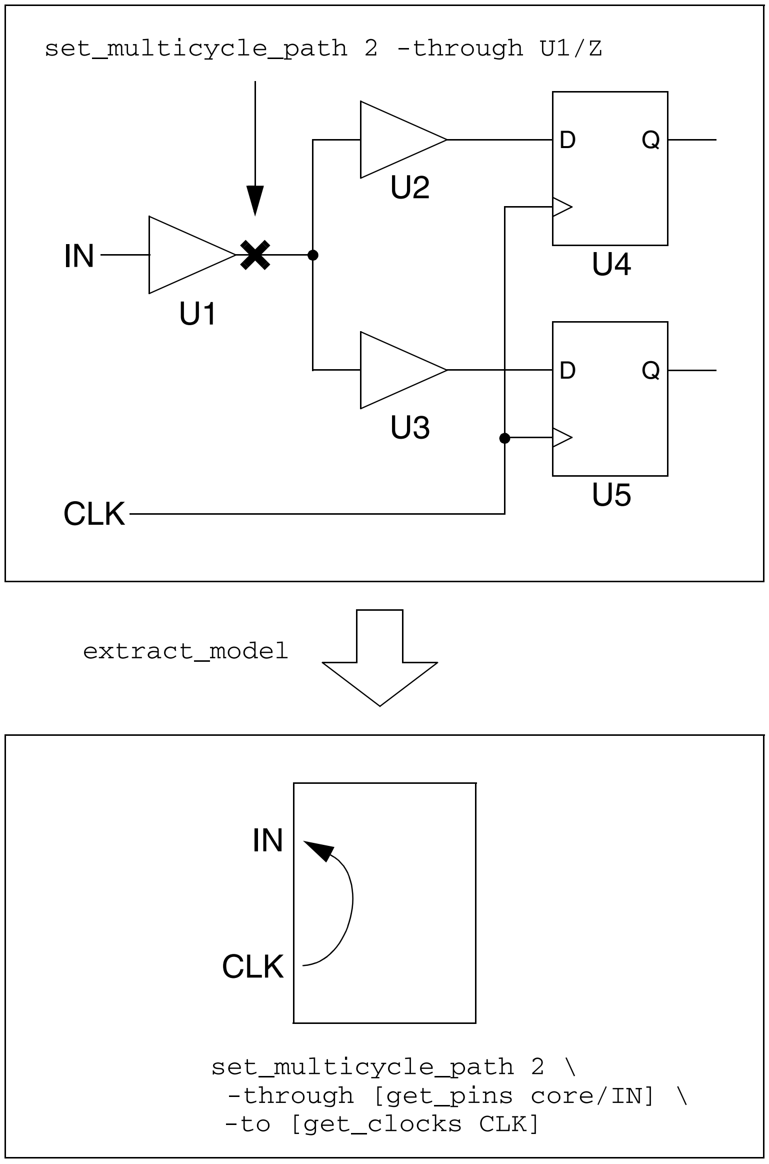
- Raddr to out in read mode
- Waddr to out in write mode
- An unmoded arc from S to out

```
pt_shell> set_mode -from Raddr -through U1/A read
pt_shell> set_mode -from Waddr -through U1/B write
```

If the original design has multicycle paths, the model extractor analyzes the multicycle paths and then writes out a script containing a set of equivalent `set_multicycle_path` commands that can be applied to the extracted model. When necessary, it also adjusts the setup, hold, and delay values for the timing arcs to correctly model the multicycle paths.

The name of the script file containing the multicycle path definitions is *output*_constr.pt, where *output* is the name specified by the `-output` option of the `extract_model` command. You need to apply this script when you use the extracted timing model. For example, Figure 4-20 shows a design in which a multicycle path has been defined at a point just inside the interface logic. The model extractor creates a setup arc from CLK to IN and writes the `set_multicycle_path` command shown in Figure 4-20, which must be applied to the extracted model for accurate results.

Figure 4-20 Multicycle Path Extraction With an output_mcp.pt File



```
set_multicycle_path 2 -through U1/Z
```

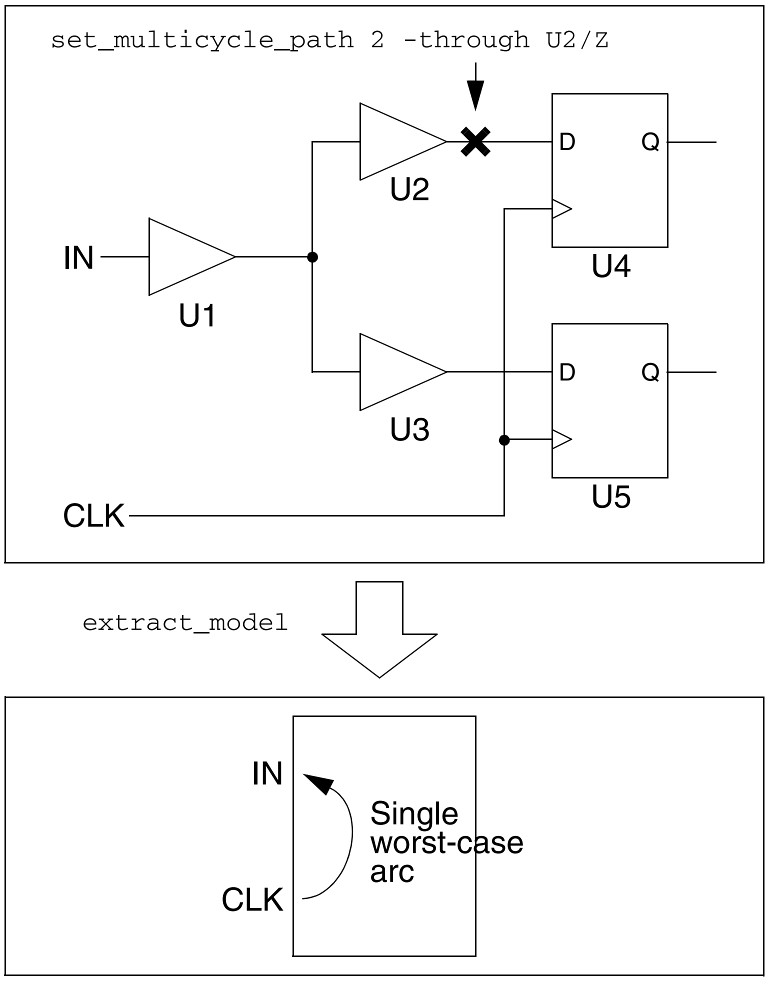
```
extract_model
```

```
set_multicycle_path 2 \
 -through [get_pins core/IN] \
 -to [get_clocks CLK]
```

A timing exception that only applies to clocks (for example, a false path between to clocks) is written to the same constraint file as the multicycle paths, *output_*constr.pt. You need to apply this script when you use the extracted timing model.

If a multicycle path has been defined that cannot be moved to the boundary of the design, the model extractor does not write out a multicycle path definition. Instead, it creates worst-case setup, hold, and delay arcs using the clock period defined at the time of model extraction, as demonstrated in Figure 4-21.

Figure 4-21 Multicycle Path Extraction Without an output_mcp.pt File



Restricting the Types of Arcs Extracted

By default, the `extract_model` command extracts a full set of timing arcs for the model, including the following types: minimum sequential delay, maximum sequential delay, minimum combinational delay, maximum combinational delay, setup, hold, recovery, removal, clock gating, and pulse width arcs.

You can optionally extract only certain arc types for a model. This feature can be useful for debugging purposes when you are only interested in one type of arc, and you want to run multiple extractions under different conditions. Model extraction runs faster with this option because PrimeTime only spends time extracting the requested arcs.

To restrict the types of arcs extracted, use the `-arc_types` option of the `extract_model` command, and specify the types of arcs you want to extract. These are the allowed arc type settings:

- `min_seq_delay`: minimum-delay sequential arcs
- `max_seq_delay`: maximum-delay sequential arcs
- `min_combo_deay`: minimum-delay combinational arcs
- `max_combo_delay`: maximum-delay combinational arcs
- `setup`: setup arcs
- `hold`: hold arcs
- `recovery`: recovery arcs
- `removal`: removal arcs
- `pulse_width`: pulse width arcs

Back-Annotated Delay and Layout Information

You can back-annotate two types of information to a design in PrimeTime:

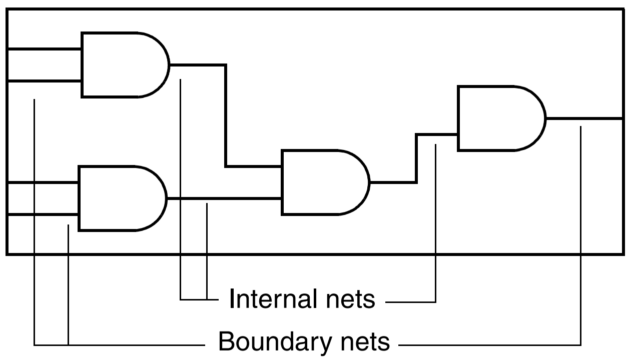
- Layout information, including lumped net capacitance or resistance and detailed RC information using standard parasitic exchange formats.
- Delay information, including net delays, cell delays, or both, using SDF. These delays are calculated for a given transition time.

In the absence of both types of information, PrimeTime calculates arc delays in the model using the cell libraries and calculates net delays using the wire load model set on the design.

Back-Annotated Delay on Nets and Cells

PrimeTime handles back-annotation of delays on internal nets and boundary nets in different ways. Figure 4-22 shows the locations of boundary nets and internal nets in a simple design.

Figure 4-22 Boundary Nets and Internal Nets



Internal net delays are included in the extracted arcs of the model. Boundary net delays cannot be computed until the model is used because the delays depend on the connection of the model to the outside world, unless the `-library_cell` option is used for model extraction, in which case boundary net delays are included in the extracted timing arcs.

Internal Nets

When the internal nets in the design are back-annotated with lumped capacitance, PrimeTime computes the net delays, cell delays, and transition times at cell outputs for model generation using this capacitance instead of the wire load models.

When SDF is back-annotated to internal nets, PrimeTime uses the back-annotated net delays during model extraction. For the transition times at cell outputs to be accurately computed, the net capacitance must also be back-annotated.

The extractor also supports extraction of designs in which internal nets are back-annotated with detailed parasitics in DSPF, RSPF, or SPEF.

Boundary Nets

By default, PrimeTime writes all annotated parasitics of the boundary nets to the extracted timing model, including both detailed and lumped RC information. This preserves the dependence of the net delays on the environment when the model is used. However, using the `ignore_boundary_parasitics` option of the `extract_model` command causes PrimeTime to ignore the boundary parasitics for timing model extraction, including both detailed and lumped RC information.

When the extracted model type is a library cell (created by using the `-library_cell` option of `extract_model`), PrimeTime lumps all of the boundary net capacitors onto the ports of the model. For primary input nets, it adds the wire delay resulting from using the driving cell present on the input port at the time of model extraction. For primary output ports, it calculates the wire delay by using the range of external loads characterized for the driving device.

Cells With Annotated Delays

If the cells in the design are back-annotated with SDF delay information, PrimeTime uses this information when it extracts arc delays in the model. Because the SDF information is computed at a particular transition time, the delays of arcs in the model are accurate for this transition time. The model can be inaccurate for other transition times.

To ensure that the model is context independent, do not annotate SDF to cell delays. If you need to annotate cell delays, the extracted model is accurate for the transition time at which the SDF is generated. To create some dependence on transition time and capacitive load, you can remove the annotations on boundary cells by using the `remove_annotated_delay` and `remove_annotated_check` command before you extract the model.

Guidelines for Back-Annotation

When you perform model extraction, keep in mind the following guidelines for various types of layout and delay data back-annotation:

Internal nets

Use detailed RC data if available. This data generates the most accurate context-independent model.

- Use both SDF and lumped RC data if both are available.
- Use lumped RC data if only this data is available.
- Use SDF data alone if only this data is available. In this case, the extracted model is accurate only for the transition times at which the SDF is generated. However, using SDF data alone is not recommended.

Cell delays

To ensure that the model is context-independent, do not annotate SDF to cell delays. If you need to annotate cell delays, the extracted model is accurate only for the transition times at which the SDF is generated. To create some dependence on transition time, you can remove the annotations on boundary cells by using the `remove_annotated_delay` command and the `remove_annotated_check` command before you extract the model.

Boundary nets

If you have the detailed parasitics for the boundary nets, annotate the information before you extract.

Performing Model Extraction

After you understand the tasks necessary to set up your modeling environment (see "Preparing for Extraction" on page 4-20), you can extract a timing model.

The following sections explain how to do this:

- Loading and Linking the Design
- Preparing to Extract the Model
- Generating the Model

Loading and Linking the Design

Load and link your design into PrimeTime by following these steps:

1. Set the search path of your library. For example, enter

```
pt_shell> set search_path \
            ". /remote/release/v2000.11/libraries/syn"
```

2. Set the link path of your library:

```
pt_shell> set link_path "* class.db"
```

3. Read your design into PrimeTime:

```
pt_shell> read_db design.db
```

4. Link your design:

```
pt_shell> link_design design
```

Preparing to Extract the Model

Preparing a model for extraction includes the following steps:

1. Define wire load models for your design. Enter

```
pt_shell> set_wireload_model -name wire_model_name
```

2. Define the clocks in your model. For example, enter

```
pt_shell> create_clock -period 10 CLK1
pt_shell> create_clock -period 20 CLK2
```

3. Set the false paths in the design:

```
pt_shell> set_false_path -from IN4 -to OUT1
```

4. Check your design for potential timing problems:

```
pt_shell> check_timing
```

5. Verify that the design meets timing requirements:

```
pt_shell> report_timing
```

6. Set the model extraction variables, if applicable. PrimeTime uses the default values for any variables you do not set. For example, enter

```
pt_shell> set extract_model_capacitance_limit 5.0
pt_shell> set extract_model_transition_limit 2.0
```

Generating the Model

Determine the options you want to use for the `extract_model` command and issue the command. For example, enter the following command to extract a timing model for the current operating conditions and create a .db model file and a design in .db format:

```
pt_shell> extract_model -output example_model \
            -format {db}
```

PrimeTime displays a report similar to this:

```
Warning: Environment variable
'extract_model_min_resolution' is
not defined.
Using default value '0.1'. (MEXT-6)
Using default value '0.45'. (MEXT-6)
Wrote model library core to './example_model_lib.db'
Wrote model to './example_model.db'
```

If you have generated a .lib version of the timing model, you may notice it contains user-defined attributes. For a description of these attributes, see “PrimeTime User-Defined Attributes in .lib” on page 4-41.

PrimeTime User-Defined Attributes in .lib

User-defined attributes are used by PrimeTime and can be used by other Synopsys tools. These attributes do not affect timing analysis when using the model.

- `min_delay_flag` - Used to help `merge_models` separate minimum and maximum arcs between a pair of pins with the same sense. This attribute is used to properly merge the arcs and maintain the arc configuration across different modes.

- `original_pin` - Used to help maintain the mapping of extracted time model cell pins to the original pin names that were defined in the block netlist prior to extraction.

Merging Extracted Models

A module can have different operating modes, such as test mode and normal operating mode. The timing requirements for a module can be quite different for different operating modes. In these cases, you need to extract a separate model for each mode. For each extraction, place the module into the applicable mode by setting the instance or design modes, by using case analysis, and by setting the applicable timing exceptions.

To use different models extracted under different operating modes, you can swap each model into the higher-level design by using `swap_cell`. However, instead of keeping and using multiple extracted models, you can optionally merge them into a single, comprehensive model that has different timing arcs enabled for different operating modes. Then you can use this single model and change its behavior by setting it into different modes.

This is the basic procedure for creating and merging timing models for different operating modes:

1. Load and link the module netlist.
2. Back-annotate the SDF data or detailed parasitics.
3. Apply the constraints for the first operating mode.
4. Extract a model in .lib format for the operating mode.

   ```
   pt_shell> extract_model -format lib -output model_1
   ```

5. Remove all constraints on the design.
6. Repeat steps 3, 4, and 5 for each additional operating mode.
7. Merge the generated .lib models

   ```
   pt_shell> merge_models \
             -model_files {model_1.lib model_2.lib ...} \
             -mode_names {mode_1 mode_2 ...} \
             -group_name etm_modes \
             -output my_model \
             -formats {db lib} \
             -tolerance 0.1
   ```

Each `extract_model` command generates a .lib file and a .data file for the extracted model. The `merge_models` command merges the extracted models into a single model that has different operating modes. You specify the operating mode when you use the model.

In the `merge_models` command, you specify the .lib files, the names of the modes corresponding to those models, the name of the new model being generated, the model formats to be generated (.db and .lib), an optional mode group name, and an optional tolerance value. For more information about mode groups, see *PrimeTime Fundamentals User Guide*.

The tolerance value specifies how far apart two timing values can be to merge them into a single timing arc. If the corresponding arc delays in two timing models are within this tolerance value, the two arcs are considered the same and are merged into a single arc that applies to both operating modes. The default tolerance value is 0.04 time units.

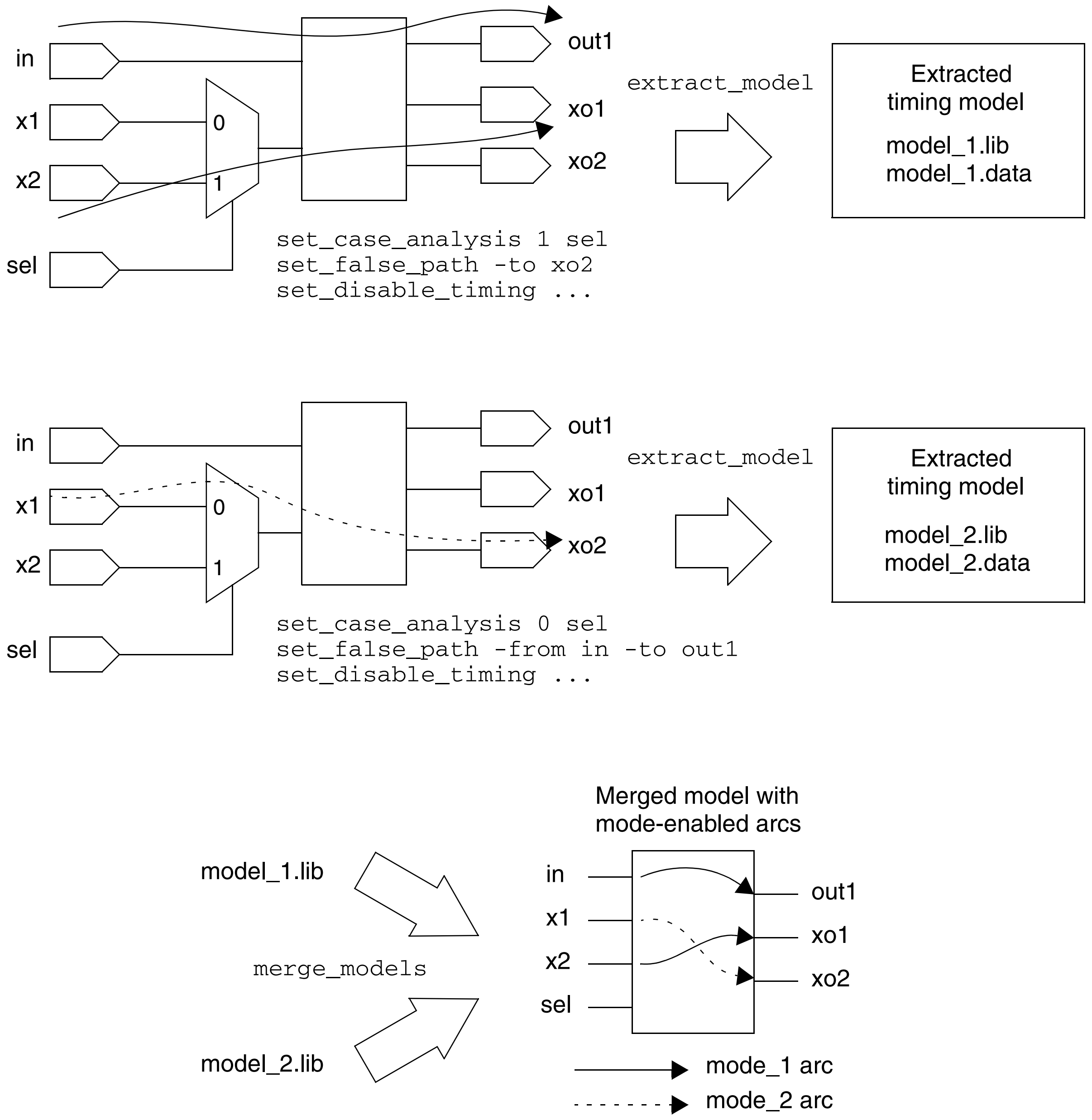
PrimeTime can handle more than one mode per timing arc, but some tools cannot. To generate a merged model that can work with these tools, use the `-single_mode` option in the `merge_models` command. This forces the merged model to have no more than one mode per timing arc, resulting in a larger, less compact timing model.

It might be necessary to disable all arc merging. For instance, if you are performing two independent merges and the resulting merged models must have the same number of arcs. This occurs when one merged model has the maximum and another has the minimum operating condition. To use these two models in the `set_min_library` command, they must have exactly the same number of arcs. To disable all arc merging, you can use the `-keep_all_arcs` option of the `merge_models` command.

The models being merged must be consistent, with the same I/O pins, operating conditions (process/voltage/temperature), and so on. Timing arcs that are different are assigned to the modes specified in the `merge_models` command. Design rule constraints (DRC), such as minimum and maximum capacitance and transition time, are allowed to be different. In those cases, the more restrictive value is retained in the merged model.

Figure 4-23 shows an example of a module from which two timing models are extracted, where the extracted models are merged into a single timing model having two operating modes.

Figure 4-23 Model Extraction and Merging Example



To use a merged timing model with modes:

1. Set the `link_path` variable to include the path to the merged model in .db format.
2. Load and link the top-level design.
3. Apply the constraints and back-annotation on the design.
4. Using the `set_mode` command, set the mode on the module instance to the desired mode.
5. Run the timing analysis.
6. Repeat steps 4 and 5 for each mode that you want to analyze.

Here are some points to consider for model merging:

- To retain in the merged model, case values propagated to the output pins of the models being merged must be the same in all models. If there are mismatching case values, PrimeTime ignores them and issues a warning message.
- Any model that already has modes or moded arcs defined in it cannot be merged.
- Any generated clocks in the models to be merged must be exactly the same.

For more information, see the `merge_models` man page.

5 Model Validation

When you create an extracted timing model or interface logic model, it is recommended that you validate the timing characteristics of the new model against the original gate-level netlist. To accomplish this task, use the `write_interface_timing` and `compare_interface_timing` commands as described in this chapter.

- Model Validation Overview
- write_interface_timing Report
- compare_interface_timing Report
- Comparison Tolerance
- Debugging Timing Arc Comparison Failures
- Debugging Other Comparison Failures

Model Validation Overview

PrimeTime can create two kinds of timing models from a gate-level netlist: an interface logic model or an extracted timing model. An interface logic model is a structural model that maintains the interface logic at the input and output ports of the block, as explained in Chapter 3, “Interface Logic Models. An extracted timing model is a purely behavior model, consisting of a set of timing arcs between the ports of the block, as explained in Chapter 4, “Extracted Timing Models.

Model Validation Commands

You can validate a new interface logic model or extracted timing model against the original gate-level netlist, or compare any two valid timing models in PrimeTime, by using the following commands:

- `write_interface_timing`
- `compare_interface_timing`

The `write_interface_timing` command writes a report on the interface timing of a specified netlist or model. This report includes the worst-case slacks or timing arc values for various types of paths, the transition times at the output ports, the lumped and extracted capacitance on all ports, and the design rules on the ports. It considers the input, output, and combinational paths, but not register-to-register paths. Static noise response characteristics can be included as well, if desired, by using the `-include {timing noise}` option. To generate only noise information in the report, use the `-include {noise}`option. If you have not specified the `-include` option, only timing information is written.

The `compare_interface_timing` command compares two reports generated by the `write_interface_timing` command. It lets you specify the reference file, the comparison file, the types of paths and timing parameters to compare or not compare, and the allowed tolerance levels that trigger comparison failures. If the timing parameter values in the two files are the same or within the specified tolerance, the result is “pass.” Otherwise, the result is “fail.”

For more information about the `write_interface_timing` and `compare_interface_timing` commands, see the man pages.

Model Generation and Validation Procedure

Use the following procedure to generate a new timing model and validate it against the original gate-level netlist.

1. Read and link the original gate-level netlist and apply the applicable timing constraints and environment.
2. Run `check_timing` to make sure all paths are constrained. Otherwise, some paths might not be checked properly because their slack cannot be analyzed.
3. If you are not back-annotating SDF, and if you want to avoid differences between the netlist and timing model resulting from slew propagation differences, set the slew propagation variable as follows:

   ```
   pt_shell> set timing_slew_propagation_mode \
             "worst_arrival"
   ```

4. Write the interface timing data for the netlist design:

   ```
   pt_shell> write_interface_timing net.rpt
   ```

5. Generate the extracted timing model or interface logic model.
6. Remove the netlist design.
7. Read and link the model extracted in step 4, and apply the same environment as in step 1.
8. Write the interface timing data for the model:

   ```
   pt_shell> write_interface_timing model.rpt \
               -timing_type slack
   ```

9. Compare the interface timing reports:

   ```
   pt_shell> compare_interface_timing net.rpt model.rpt \
             -absolute_tolerance 0.1 -output compare.rpt
   ```

   The return code reported by PrimeTime is 0 if the two files match (pass) or 1 if there was any difference (fail). Any other return code or no return code indicates a command error. The `-absolute_tolerance` option sets the tolerance thresholds for comparison failure.

10. Examine the comparison report. If necessary, debug the cause of any comparison failures. Use the `report_etm_arc` command to debug timing differences or the `report_port` command to debug transition time, capacitance, or design rule differences.

Instead of comparing slack values, you can compare timing arc values (setup, hold, and delay) by using the `-timing_type arc` option of the `write_interface_timing` command.

Slew Propagation

The `timing_slew_propagation_mode` variable lets you specify how PrimeTime propagates slew through a circuit. When set to `worst_slew` (the default), it propagates the worst slew selected from the inputs. When set to `worst_arrival`, it selects the slew of the input with the worst arrival time, selecting from multiple inputs propagated from the same clock domain.

The default setting can lead to differences between the extracted model and the original netlist. To avoid these differences, you can set the variable to `worst_arrival` prior to model extraction. Doing this ensures consistent slew propagation behavior for the netlist and the extracted model, but requires more runtime for model validation and debugging.

An alternative method to reduce the differences between the extracted model and the original netlist is to use the `worst_slew` mode with the `extract_model_use_conservative_current_slew` variable is set to `true`.

In that case, in the `worst_slew` (default) mode, the model extractor adjusts the timing arcs of the model to more closely resemble the netlist behavior. The result is a more pessimistic model that is more likely to pass validation when both the netlist and the adjusted model are timed in `worst_slew` mode. This variable setting has no effect in `worst_arrival` mode.

write_interface_timing Report

The `write_interface_timing` command writes a report on the interface timing of a specified netlist or model. The report contains the following major sections:

- Slack or Arc Value – This section reports the worst-case slack or arc value for each path from input port to clock, from clock to output port, and from input port to output port.
- Transition Time – This section reports the actual transition time at each port for the four delay types: min_fall, min_rise, max_fall, and max_rise.
- Capacitance – This section reports the maximum total (lumped) capacitance at each port, and if available, the effective capacitance.
- Design Rules – This section reports all design rules that apply to each port, including maximum capacitance, minimum capacitance, maximum transition time, maximum fanout (for input ports), and fanout load (for output ports).
- Noise Detection – This section reports the noise slack at the inputs.
- Noise Calculation– This section reports the steady-state I-V characteristics of the outputs.

The following example is a typical interface timing report showing arc values.

```
pt-shell> write_interface_timing demo.rpt \
            -timing_type arc
1
pt-shell> sh cat demo.rpt

****************************************
Command: write_interface_timing
         demo.rpt
         -timing_type arc
Design : top
Version: 2001.08-SI2
Date   : Mon Aug 6 14:31:49 2001
****************************************

*********************************************
Section: arc_values
Info   : Worst-case arc values for each port and path group
Design : top
*********************************************
Generated Clock and Source Info:

Attribute:
    L<n> - latch level where <n> is 0 for first level
                 Arc                 Arc
From    To       Type                Value
---------------------------------------------------------
in(r)   out(r)   max_combo_delay     17.35
in(f)   out(r)   max_combo_delay     17.29
in(r)   out(f)   min_combo_delay     17.35
in(f)   out(f)   min_combo_delay     17.29
in(r)   clk(r)   setup               2.63  L1
in(f)   clk(r)   setup               2.29  L2
in(r)   clk(f)   hold                0.17
in(f)   clk(f)   hold                0.51
clk(r)  out(r)   max_seq_delay       17.79
clk(r)  out(f)   max_seq_delay       18.20
clk(f)  out(r)   min_seq_delay       17.79
clk(f)  out(f)   min_seq_delay       18.20

*********************************************
Section: transition_time
Info   : Actual transition time on each port
Design : top
*********************************************

Port     MAX_RISE MAX_FALL MIN_RISE MIN_FALL
-------------------------------------------------------
in         0.00     0.00     0.00     0.00
clk        0.00     0.00     0.00     0.00
out        0.06     0.02     0.06     0.02
*********************************************
Section: capacitance
Info   : The total and effective capacitance on each port
```

```
Design : top
********************************************
                          Max
Port                 Ctot        Ceff
--------------------------------------------------------
in                   1.39         --
clk                  1.39         --
out                  0.39         --

********************************************
Section: design_rules
Info    : Design rules on each port
Design : top
********************************************
Max           Min      Max      Max                  Fanout
Port          Cap      Cap      Trans      Fanout     Load
--------------------------------------------------------
in            --       --       --          --        n/a
clk           --       --       --          --        n/a
out           --       --       --        n/a         0.00
```

Arc Types

In the slack or arc value section of the interface timing report, each reported worst-case value has an associated arc type:

- min_seq_delay (minimum delay arc from clock to output port)
- max_seq_delay (maximum delay arc from clock to output port)
- min_combo_delay (minimum delay arc from input port to output port)
- max_combo_delay (maximum delay arc from input port to output port)
- setup (setup arc from input port to clock)
- hold (hold arc from input port to clock)
- recovery (recovery arc from input port to clock)
- removal (removal arc from input port to clock)
- clock_gating_setup (clock gating setup arc from input port to clock)
- clock_gating_hold (clock gating hold arc from input port to clock)

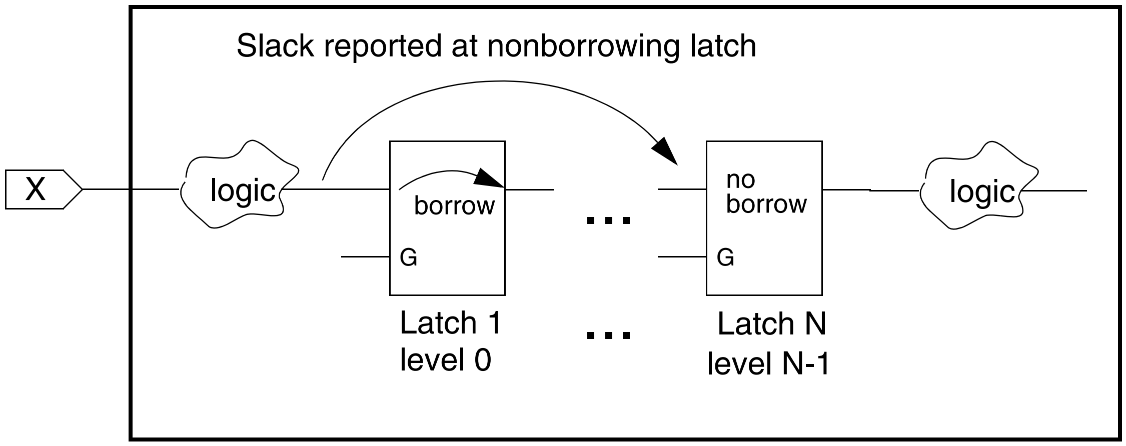
When you use the `compare_interface_timing` command to compare two interface timing reports, you can selectively include only the reports associated with specific arc types. The arc type can also be important when you want to verify a comparison failure with the `report_timing` command. For more information, see "Debugging Other Comparison Failures" on page 5-14.

Sometimes the notation `L`*n* appears after the slack or arc value in the interface timing report. When present, it indicates the level at which borrowing stopped in a chain of transparent latches in the path. The number *n* indicates the latch level number, starting with zero for the first latch. For example, the notation `L2` means borrowing stopped at the third latch encountered in the path.

Time Borrowing by Transparent Latches

The `write_interface_timing` command, like other PrimeTime analysis tools, recognizes that level-sensitive latches can borrow time in a path. The command traces through borrowing latches at the interface of a design until it encounters a flip-flop, port, or nonborrowing latch. In the latter case, the slack is reported at the nonborrowing latch, as illustrated in Figure 5-1.

Figure 5-1 Path Tracing Stopped at a Nonborrowing Latch



You can use the `-latch_level` option of the `write_interface_timing` command to specify the number of levels of latch borrowing that are to occur at the interface of a design. Using this option is recommended only when you are certain of the borrowing behavior of latches at the interface of the design and you are able to specify a single latch chain length for the entire design. The `write_interface_timing` command reports the level of latch borrowing as an attribute, starting with zero for the first latch in the path. If the first latch encountered is nonborrowing, it is reported as level zero. If the design contains latch-borrowing feedback loops, PrimeTime prevents borrowing at the startpoint of the loop.

When PrimeTime creates an extracted timing model or interface logic model, it assumes that the netlist being modeled has already had its internal timing verified. Therefore, the `write_interface_timing` command only considers the borrowing behavior of latches on the interface timing paths. Note that borrowing can only occur on a maximum-delay (not a minimum-delay) path.

compare_interface_timing Report

The `compare_interface_timing` command compares two reports generated by the `write_interface_timing` command. You specify the two reports to compare, the types of paths and timing parameters to compare or not compare, and the allowed tolerance levels that trigger comparison failures. The resulting report shows the comparison results for individual paths, ports, and timing parameters. You can have the report sent to the screen or written to a file.

By default, the `compare_interface_timing` command compares all of the parameters contained in the timing interface reports. The resulting comparison report has the same sections as the interface timing report: slack or arc value, transition time, capacitance, design rules, and noise. You can restrict the scope of the report by using the `-include` option.

The following example is a typical comparison report. For each path, the report shows the path type, the value in the reference file, the value in the comparison file, the difference, and pass/failure status for that individual comparison.

```
pt-shell> compare_interface_timing \
          demo_net.rpt demo_etm.rpt \
          -include arc_values -percent_tolerance {10 5)

****************************************
Command: compare_interface_timing
         demo_net.rpt demo_etm.rpt
                        -include arc
         -percent_tol 10.0 5.0
Design : top
Version: 2002.03-SI1
Date   : Wed Jun 12 11:20:01 2002
****************************************

             Arc                Arc Value
From    To   Type               Ref       Cmp  %Error  Status
-----------------------------------------------------------
in(r)  out(r) max_combo_delay  2.63     2.21     15.97   FAIL
in(f)  out(r) max_combo_delay  2.28     2.32     -1.75   PASS
in(r)  out(f) min_combo_delay  0.17     0.30    -76.47   FAIL
in(f)  out(f) min_combo_delay  0.51     0.51     0.00    PASS
in(r)  clk(r) setup          2.63 (L1) 2.63     0.00    PASS
in(f)  clk(r) setup          2.29 (L2) 2.29     0.00    PASS
in(r)  clk(f) hold           0.1       0.17     0.00    PASS
in(f)  clk(f) hold           0.51(f)   0.51(r)  0.00    PASS
clk(r) out(r) max_seq_delay  17.79     17.79    0.00    PASS
clk(r) out(f) max_seq_delay  18.20     18.20    0.00    PASS
clk(f) out(r) min_seq_delay  2.21      2.21     0.00    PASS
clk(f) out(f) min_seq_delay  1.80      1.80     0.00    PASS
```

```
                             Transition
       Totals  Arc Value Time        Capacitance  Rules
-----------------------------------------------------------
Passed   10      10        0           0
Failed   2       2         0           0           0
Total    12      12        0           0           0
```

If all comparisons in the report are “pass,” the return code for the `compare_interface_timing` command is 0. If one or more comparisons result in “fail,” the return code is 1. Any other return code or no return code indicates a command error.

The L# notation indicates the latch level number. This same notation appears in the right-hand column of a `write_interface_timing` report when a path involves a transparent latch.

Comparison Tolerance

In the `compare_interface_timing` command, you can optionally specify tolerance values for the slack, arc value, transition time, capacitance, and noise comparisons. If the two values being compared are within the specified tolerance, it is considered “pass.” If you do not specify a tolerance, the default is zero, which means that any amount of difference triggers a comparison failure.

There are four tolerance settings, called the absolute, percentage, capacitance, and noise tolerance settings:

- The absolute tolerance setting specifies the absolute amount of difference for slack, arc value, and transition time comparisons, in library time units such as nanoseconds.
- The percentage tolerance setting specifies the percentage amount of difference for arc value, transition time, and capacitance comparisons. A setting of 1.0 means a difference of 1 percent.
- The capacitance setting specifies the absolute amount of difference for capacitance comparisons, in library capacitance units such as picofarads.
- The noise setting specifies the absolute amount of difference for noise slack comparisons, in library voltage units times library time units, such as volt-nanoseconds.

None of these settings apply to design rule comparisons. Design rules must match exactly to pass a comparison test.

Arc value and transition time comparisons can accept both absolute and percentage tolerance settings. In that case, both types of comparisons are done, and both types of comparisons must fail to trigger a comparison failure report. Similarly, capacitance comparisons can accept both percentage and absolute capacitance tolerance settings; both comparisons must fail to trigger a comparison failure report.

An example of a `compare_interface_timing` command that specifies both absolute and percentage tolerances is as follows:

```
pt_shell> compare_interface_timing net.rpt model.rpt \
         -absolute_tolerance {0.1 0.2} \
         -percent_tolerance 1.0 \
         -output compare.rpt
```

The command is a request to compare the two reports net.rpt (the reference file) and model.rpt (the comparison file). The `-output` option causes the comparison report to be written to a file rather than displayed in the transcript.

Absolute Tolerance

In the preceding example, two absolute tolerance values are specified, 0.1 and 0.2 time units. This means that the result of subtracting the comparison value from the reference value must be between –0.1 and +0.2 to pass the comparison test. For example, if the slack in the reference file is 5.0 time units, the slack in the comparison file must be less than 5.1 and more than 4.8 time units for the comparison to pass.

If you specify only one absolute tolerance value in the command, that same value applies to both the positive and negative directions. If no tolerance value is specified, the values must match exactly to pass.

The `-capacitance_tolerance` setting is an absolute tolerance in library capacitance units. It works in the same manner as the `-absolute_tolerance` setting, except that it applies to capacitance comparisons rather than slack, arc value, and transition time comparisons.

Similarly, the `-noise_tolerance` setting is an absolute tolerance in library voltage-time units. It works in the same manner as the `-absolute_tolerance` setting, except that it applies to noise slack comparisons.

Percentage Tolerance

In the preceding example, the percentage tolerance is set to a single number, 1.0, which represents plus or minus 1.0 percent of the arc value, transition time, or capacitance value in the reference file. If the value in the comparison file is outside of this range, the result is a comparison failure. For example, if the arc value in the reference file is 10.0, the arc value in the comparison file must be between 9.9 and 10.1 for the comparison to pass.

The percentage tolerance setting does not apply to slack comparisons. Only the absolute tolerance setting applies to slack.

Debugging Timing Arc Comparison Failures

If the `compare_interface_timing` command reports a comparison failure, the reason for the failure might not be obvious. To find out more about the timing arcs related to the failure, use the `report_etm_arc` command. In this command, you specify the startpoint and endpoint of the arc of interest, as indicated in the comparison report. You also specify the conditions for model extraction, just as you do in the `extract_model` command.

The `report_etm_arc` command generates an extracted timing model (ETM) report. This report shows the details of the data path used by the `extract_model` command to generate the extracted timing arc, including the capacitance, transition time, and delay values calculated at intermediate points along the path. You can optionally get a report on the clock path in addition to the data path.

Another option is to get a report on the critical path between the same two endpoints, as determined by the `report_timing` command operating on the original netlist. This is called a netlist report.

By comparing the ETM report and the netlist report at each point along the path, you can determine the point of divergence and the likely cause of the comparison failure. You might be able to fix such a problem by editing or deleting the arc in the extracted model, or by extracting a new model under different conditions. In some cases, you might decide that the extracted model is accurate enough and may be used without further change.

Validation Flow With Timing Arc Debugging

To generate, compare, and debug an extracted timing model, you can use a procedure similar to the following example.

This example uses two pt_shell sessions running at the same time: one to analyze the netlist and the other to analyze the model. If you have only one PrimeTime license available, you can perform the same tasks by removing and reloading the netlist and model each time for analysis.

1. In the first pt_shell window, read and link the original gate-level netlist and apply the applicable timing constraints and environment. Run `check_timing` to make sure all paths are constrained.
2. Extract the timing model:

```
pt_shell> extract_model -output etm
```

3. Make sure that the slew propagation mode is set properly for model validation and debugging (see “Slew Propagation” on page 5-4).

4. Write the interface timing data for the netlist:

   ```
   pt_shell> write_interface_timing net.rpt
   ```

5. In the second pt_shell window, load the extracted model and apply the applicable timing constraints and environment. Set the slew propagation mode (if applicable), and report the interface timing of the model:

   ```
   pt_shell> write_interface_timing model.rpt
   ```

6. Examine the generated report. For each comparison failure, use the `report_etm_arc` command to generate a report for the failing arc. Use the first pt_shell window, where the netlist is loaded (or reload the netlist first).

   For example, if the arc type shown in the comparison report is max_seq_delay, use a debugging command in the following form:

   ```
   pt_shell> report_etm_arc -from clock -to port  \
             -arc_type max_seq_delay
             -include {clock_path netlist_path}
   ```

   The `report_etm_arc` command generates an ETM report and a netlist report for the specified arc.

In the `report_etm_arc` command, you must use the same options as in the original `extract_model` command, such as `-library_cell` or `-context_borrow`.

The `-from` and `-to` options specify the startpoint and endpoint of the arc to be reported. Depending on the arc type, the startpoint and endpoint each might be either a clock or a port. Instead of `-from`, you can use `-rise_from` or `-fall_from` to restrict the report to only rising edges or only falling edges at the startpoint of the arc. Similarly, instead of `-to`, you can use `-rise_to` or `-fall_to`.

The `-include` option specifies the types of paths you want included in the report. By default, only the netlist path is reported.

Here is another example. Suppose that the failure report looks like this:

```
              Arc
From   To     Type         Ref        Cmp   %Error  Status
-----------------------------------------------------------
in(r)  clk(r) setup        2.63 (L1)  2.53  3.80    FAIL
```

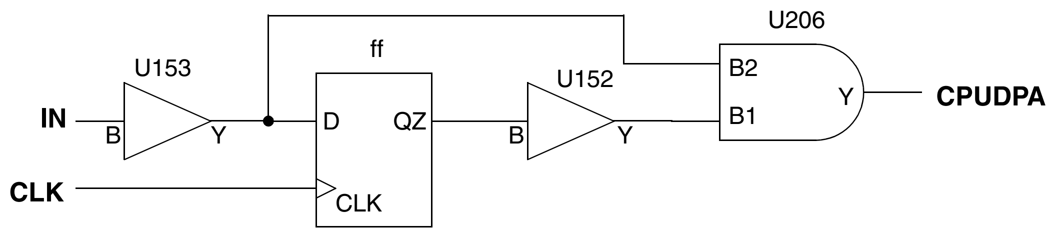
For this type of failure, you could use the following command:

```
pt_shell> report_etm_arc -rise_from in -rise_to clk \
          -arc_type setup \
          -include {clock_path netlist_path}
```

Debugging Report Example

Consider the circuit shown in Figure 5-2.

Figure 5-2 Circuit Used to Generate an Extracted Timing Model



Suppose that you generate an extracted timing model from this design. You generate an interface timing report for the original design and for the extracted model, and then generate a comparison report, which shows a comparison failure for one arc. You use the following commands to define the clock and set the input transition time:

```
pt_shell> create_clock -name CLKr -period 6.0 -waveform \
           {0.0 3.0} [get_ports {CLK}]
pt_shell> set_input_transition 0.700000 CLK
```

You generate a debug report on a min_combo_delay arc from a falling edge on input IN to a falling edge on output CPUDPA:

```
pt_shell> report_etm_arc -fall_from IN -fall_to CPUDPA \
          -arc_type min_combo_delay -library_cell \
          -include {clock_path netlist_path}

ETM report for arc : IN_CPUDPA_min

Point                     Cap      Trans     Incr      Path
---------------------------------------------------------
IN (in)                 3.6344    0.0000    0.0000    0.0000 f
U153/B (NA220)                    0.0000    0.0042    0.0042 f
U153/Y (NA220)          3.8579    0.6339    0.2897    0.2939 r
U206/B2 (BF056)                   0.6339    0.0013    0.2952 r
U206/Y (BF056)        121.6144   11.3933    8.3614    8.6566 f
CPUDPA (out)                     11.3933    0.0000    8.6566 f
----------------------------------------------------------
Netlist path:

Point                     Cap      Trans      Incr      Path
----------------------------------------------------------
IN (in) <-              3.6344    0.0000    0.0000    0.0000 f
U153/B (NA220)                    0.0000    0.0042    0.0042 f
U153/Y (NA220)          3.8579    0.6339    0.2897    0.2939 r
```

```
U206/B2 (BF056)                 0.6339   0.0013   0.2952 r
U206/Y (BF056)       61.6144  11.3933   8.0973   8.3925 f
CPUDPA (out) <-               11.3933   0.0711   8.4636 f
----------------------------------------------------------------
```

The ETM and netlist reports diverge at U206/Y because of different capacitance values defined on the CPUDPA output port. This was the cause of the comparison failure.

Causes of Validation Failure

An extracted model might have slack or delay values different from those reported by the`report_timing` command operating on the original netlist, possibly because of differences in interpreting timing exceptions or differences in handling slew propagation (see "Slew Propagation" on page 5-4).

The model extractor does not always interpret timing exceptions the same as the `report_timing` command. For example, it does not support "rise" or "fall" type exceptions. This can cause the extracted model to use a false path. In that case, `report_etm_arc` command shows the different paths taken in the extracted model and in the netlist. You might be able to fix this condition by changing the constraint (for example, from `set_false_path` to `set_disable_timing`) and running model extraction again.

Debugging Other Comparison Failures

The `compare_interface_timing` command can report comparison failures not related to timing arcs, such as a difference in transition time, output capacitance, or design rule parameter. You might want to manually examine the information in the comparison report or the corresponding lines in the interface timing report to determine the causes of these differences.

Transition Time

To verify the transition times reported in the Transition Time section, use the following command:

```
pt_shell> get_attribute [get_port port_name] attribute
```

where *port_name* is the name of the port and *attribute* is one of the following delay types:

```
actual_rise_transition_min
actual_rise_transition_max
actual_fall_transition_min
actual_fall_transition_max
```

For example,

```
pt_shell> get_attribute [get_port out1] \
          actual_rise_transition_max
```

Capacitance

To verify the total or effective capacitance of a port, use one of the following commands:

```
pt_shell> get_attribute [get_net port_name] \
          total_capacitance_max

pt_shell> get_attribute [get_port port_name]  \
          effective_capacitance_max
```

Design Rules

To verify the design rules that apply to a port, use the following command:

```
pt_shell> report_port -design_rule port_name
```

6

Hierarchical Scope Checking

When you perform hierarchical analysis using interface logic models or extracted timing models, you can have PrimeTime check the “scope” of each timing model to verify that the analysis conditions at the top level are within the ranges used for checking the block at the time of model creation.

Hierarchical scope checking is described in the following sections:

- Scope Checking Overview
- Types of Block Scope Checking
- Block Scope Files
- Checking the Block Scope at the Chip Level

Scope Checking Overview

Interface logic models and extracted timing models are designed to be context-independent within certain ranges of conditions. Before a timing model is created, the internal logic of the block is checked for timing violations, subject to external conditions such as input delay and output delay. Once verified, the internal timing of the model is guaranteed to have no violations as long as the external conditions are within the ranges specified when the internal logic was checked.

For hierarchical crosstalk analysis using PrimeTime SI, the internal logic of the timing model is checked with crosstalk, taking into account certain ranges of conditions on the interface nets. If any interface nets are cross-coupled to internal nets (for an interface logic model) or non-port nets (for an extracted timing model), the external arrival times must be within the specified ranges to guarantee that there are not any timing violations inside the block.

When you use a timing model for chip-level analysis, you can have PrimeTime check the actual chip-level conditions for the block instance against the ‘scope’ of the block checked before model creation, and verify that the actual conditions are within the ranges specified at the time of model validation.

The scope checking process is illustrated in Figure 6-1. After you perform a block-level analysis, you generate a timing model and a block scope file. In the chip-level analysis, PrimeTime verifies that the timing conditions in the context of the top level are within the ranges recorded in the scope file.

The commands used in the scope checking flow are shown in Figure 6-2. After you read in the block-level design, you set the external timing constraints, specifying a range of values, from minimum to maximum, for each constraint. After performing a timing analysis, you generate a timing model using `create_ilm` or `extract_model`, using the `-block_scope` option to also generate a block scope file. The file contains information on the timing parameter ranges used for block timing validation.

Figure 6-1 Hierarchical Scope Checking Summary

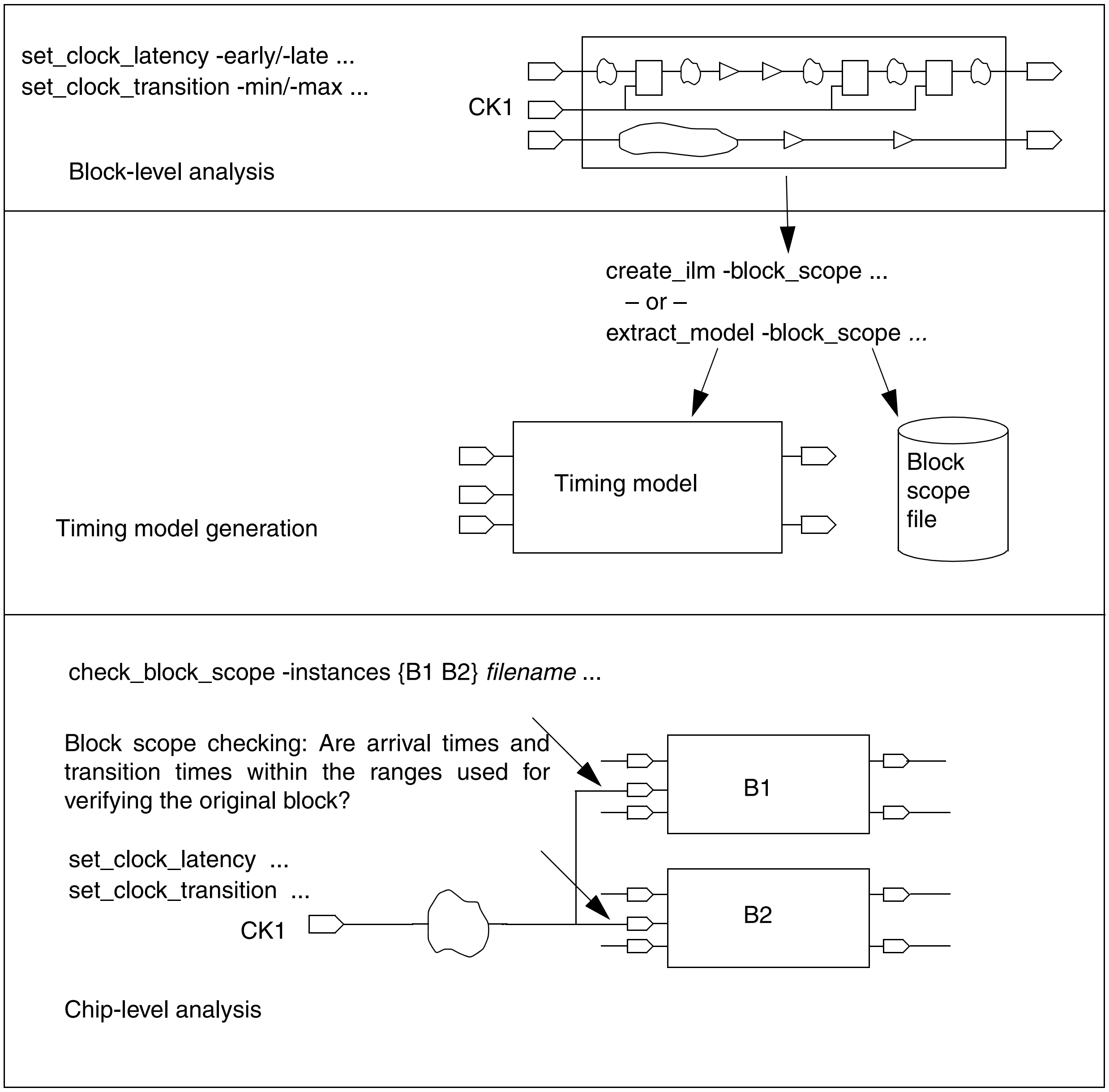
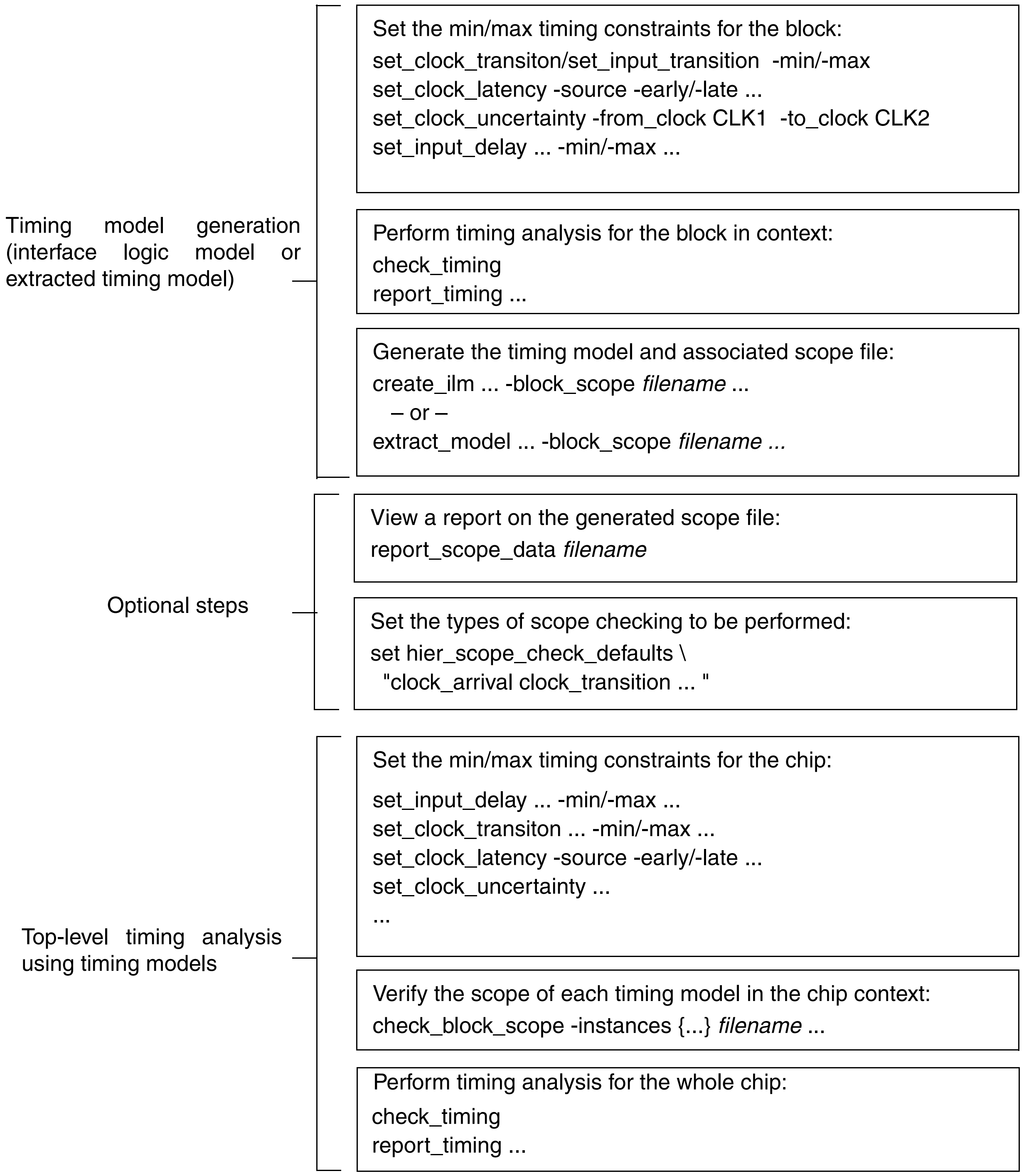

set_clock_latency -early/-late ...
set_clock_transition -min/-max ...
CK1
Block-level analysis
create_ilm -block_scope ...
– or –
extract_model -block_scope ...
Timing model
Block scope file
Timing model generation
check_block_scope -instances {B1 B2} *filename* ...
Block scope checking: Are arrival times and transition times within the ranges used for verifying the original block?
B1
B2
set_clock_latency ...
set_clock_transition ...
CK1
Chip-level analysis

Figure 6-2 Hierarchical Scope Checking Commands


Timing model generation (interface logic model or extracted timing model)
Optional steps
Top-level timing analysis using timing models

Set the min/max timing constraints for the block:

```
set_clock_transiton/set_input_transition  -min/-max
set_clock_latency -source -early/-late ...
set_clock_uncertainty -from_clock CLK1  -to_clock CLK2
set_input_delay ... -min/-max ...
```

Perform timing analysis for the block in context:

```
check_timing
report_timing ...
```

Generate the timing model and associated scope file:

create_ilm ... -block_scope *filename* ...
  – or –
extract_model ... -block_scope *filename ...*

View a report on the generated scope file:

report_scope_data *filename*

Set the types of scope checking to be performed:

```
set hier_scope_check_defaults \
  "clock_arrival clock_transition ... "
```

Set the min/max timing constraints for the chip:

```
set_input_delay ... -min/-max ...
set_clock_transiton ... -min/-max ...
set_clock_latency -source -early/-late ...
set_clock_uncertainty ...
...
```

Verify the scope of each timing model in the chip context:

check_block_scope -instances {...} *filename* ...

Perform timing analysis for the whole chip:

```
check_timing
report_timing ...
```

To get a report on the generated scope file, you can use the `report_scope_data` command. To specify the types of scope checking to perform (clock arrivals, interclock skew, clock uncertainty, clock transition, and so on) set the `hier_scope_check_defaults` variable.

When you use the generated timing model in a chip-level analysis, you can check the timing parameter scope of block instances by using the `check_block_scope` command. This command compares the actual in-context timing parameter values against the contents of the block scope file. If the timing parameters are within the ranges originally defined at the time of model validation, it means that the hierarchical analysis is also valid.

If the timing parameters are outside of the defined ranges, the `check_block_scope` command reports an error and you need to change the constraints, either at the top level or in the lower-level block analysis. Changing the constraints at the top level is preferable because the block level analysis need not be performed again.

Types of Block Scope Checking

The `hier_scope_check_defaults` variable specifies the types of block scope checking performed at the top level. The variable is a string that lists one or more of the following types of checking:

- `clock_arrival`: clock arrival times
- `clock_transition`: clock transition times
- `clock_skew_with_uncertainty`: clock skew and uncertainty together
- `clock_uncertainty`: clock uncertainty alone (without skew)
- `data_input_arrival`: captures the input data arrival times (for crosstalk analysis with PrimeTime SI)
- `data_input_transition`: captures the input data transition times (for crosstalk analysis with PrimeTime SI)

By default, the variable is set to the following string:

```
clock_arrival clock_transition clock_skew_with_uncertainty
```

Therefore, by default, using the `check_block_scope` command checks information on clock arrival times, clock transition times, and clock skew together with uncertainty. In addition, the command always checks the clock waveforms for consistency, irrespective of the variable setting.

To specify a different list of checks, set the variable as in the following example:

```
pt_shell> set hier_scope_check_defaults "clock_arrival \
          clock_transition clock_skew_with_uncertainty \
          data_input_arrival data_input_transition"
```

Clock Arrival, Transition, and Waveform

In a clock arrival check or clock transition time check, the `check_block_scope` command verifies that the clock arrival time or transition time is within the min/max range originally specified for the clock in the timing model. When you generate the timing model, be sure to specify both the minimum and maximum arrival times and transition times for each clock. For example,

```
pt_shell> set_clock_latency -early 3.2 [get_clocks CLK1]
pt_shell> set_clock_latency -late 4.5 [get_clocks CLK1]

pt_shell> set_clock_transition 0.38 -rise [get_clocks CLK1]
pt_shell> set_clock_transition 0.25 -fall [get_clocks CLK1]
```

When you generate the timing model using the `create_ilm` or `extract_model` command with the `-block_scope` option, PrimeTime finds a leaf-level pin in the fanout of each source clock. It records the name of the pin and the min/max arrival times, transition times, and waveform characteristics of the clock at that pin. If more than one pin meets these requirements, PrimeTime chooses just one pin to use for performing the checks.

During chip-level analysis using the timing model, the `check_block_scope` command reads the name of the pin from the block scope file, finds the pin in the design, and performs the types of checks listed in the `hier_scope_check_defaults` variable. For clock arrival clock transition checking, it verifies that the actual clock arrival times and transition times at the pin are within the min/max ranges recorded in the block scope file.

The `check_block_scope` command always verifies that the clock waveform characteristics (clock period and nominal edge times) are the same as in the block scope file.

Interclock Skew and Uncertainty

The `hier_scope_check_defaults` variable can specify either of two different types of clock skew and uncertainty checking: interclock skew and uncertainty together (`clock_skew_with_uncertainty`) or uncertainty alone (`clock_uncertainty`). In an interclock skew with uncertainty check, the `check_block_scope` command verifies that the amount of skew between two clocks is within the range originally specified for the block with the `set_clock_uncertainty` and `set_clock_latency` commands.

When you generate the timing model using the `create_ilm` or `extract_model` command with the `-block_scope` option, PrimeTime stores the clock source latency values and clock uncertainty values separately in the block scope file. It stores the information relative to pins in the fanout of each input port.

During chip-level analysis using the timing model, the `check_block_scope` command checks the clock latency and clock uncertainty values separately using the `clock_arrival` and `clock_uncertainty` type checks.

In addition, the `clock_skew_with_uncertainty` type check measures and verifies the combined effects of interclock uncertainty and clock source skews at block boundary. For example, for timing paths from CLK1 to CLK2 within the block, the combined required maximum effective skews are computed as follows:

```
required_max_setup_skew =

   min_arrival(CLK2) + max_arrival(CLK1) + setup_uncertainty(CLK1 to
CLK2)

required_max_hold_skew =

   max_arrival(CLK2) - min_arrival(CLK1) + hold_uncertainty(CLK1 to CLK2)
```

This same calculation also applies to a single clock used for clocking different paths (intraclock skew). There is separate accounting for rising/falling clock edge combinations when rising and falling edges have different constraints specified on them.

If you specified interclock uncertainty in the timing model, but you only want to check the uncertainty (and not the skew) at the chip level, use the `clock_uncertainty` setting in the `hier_scope_check_defaults` variable. Otherwise, the default setting, `clock_skew_with_uncertainty`, performs both skew and uncertainty checking.

Data Input and Output Conditions for Crosstalk

For crosstalk analysis using PrimeTime SI, you should specify the ranges of arrival times and transition times at the block inputs. For example,

```
pt_shell> set_input_delay -min 3.2 [get_ports IN*]
pt_shell> set_input_delay -max 3.8 [get_ports IN*]

pt_shell> set_input_transition -min 0.4 [get_ports IN*]
pt_shell> set_input_transition -max 0.6 [get_ports IN*]
```

This input signal information affects the arrival windows and slew values of block interface nets that are acting as aggressors. When the block is used at a higher level of hierarchy, the actual arrival times and transition times should be within the specified ranges to guarantee the absence of crosstalk timing violations.

When you generate the timing model using the `create_ilm` or `extract_model` command with the `-block_scope` option, PrimeTime finds records the clock skew and uncertainty characteristics on selected clock pins.

If `data_input_arrival` and `data_input_transition` checking are enabled in the `hier_scope_check_defaults` variable, the `check_block_scope` command verifies that the input arrival times and transition times at the block in the top-level design are within the ranges used for analysis of the original block.

At each output of the block, the size of the load affects the transition characteristics of the output operating as an aggressor net. For a conservative analysis, specify the smallest expected load on the output. If the load on the output is unknown, you can specify a load of zero to ensure a conservative analysis. For example,

```
pt_shell> set_load -min 0.0 [get_ports OUT*]
```

Block Scope Files

A block scope file is a file containing information on the context of a hierarchical timing block. The file is generated by the `create_ilm` or `extract_model` command when the `-block_scope` option is used. When you perform a chip-level analysis using the timing model, the `check_block_scope` command checks the context of the block instance against the context specified at the time of model generation.

Generating a Block Scope File

To generate a block scope file, use the `-block_scope` option of the `create_ilm` or `extract_model` command. When the command generates a model, it also generates the block scope file. For an interface logic model, the name of the generated file is ilm.scope. For an extracted timing model, the name of the file is `output_filename`.scope. The block scope file appears in the same directory as the other model files.

You can rename a block scope file and still use it for block scope checking. You specify the block scope file name in the `check_block_scope` command.

To generate the block scope file without generating the timing model files, use the `-block_scope_only` option rather than the `-block_scope` option. This is useful if you have already generated timing model and you only need to make the block scope file.

In a multiscenario analysis, use the `-scope_scenario` option to specify the scenarios in which to generate the block scope data. Block scope information from multiple scenarios can be stored in the same file. If the scope data file already exists and the specified scenario is not already in the file, PrimeTime appends the new data to the existing file.

When PrimeTime generates a block scope file, it puts all types of block context information into the file, not just those listed in the `hier_scope_check_defaults` variable. The variable setting only affects the type of checking done by the `check_block_scope` command.

Reporting Block Scope Files

Block scope files are in binary format, so you cannot examine them directly. To get a report on the contents of a block scope file, use the `report_scope_data` command. For example,

```
pt_shell> report_scope_data myblock/ilm.scope

****************************************
Report : scope_data
scope_file: /vobs/clt/cltsh/regr/xtalk_2ff/ilm.scope
Version: V-2004.06
Date   : Tue Mar 30 15:11:50 2004
****************************************
   Scope Data Summary
   Block                Scenario            Model Type
   -------------------------------------------------------
   xtalk_2ff            <default>           ILM
1
```

By default, a summary report is generated. To get a detailed report, use the `-verbose` option:

```
pt_shell> report_scope_data myblock/ilm.scope -verbose

****************************************
Report : scope_data
scope_file: /vobs/clt/cltsh/regr/xtalk_2ff/ilm.scope
Version: V-2004.06-Alpha1
Date   : Tue Mar 30 15:12:06 2004
****************************************

Summary
------------------------------------------------
Block name: xtalk_2ff
Scope scenario: <default>
Model type: ILM
------------------------------------------------

  Clock Arrival Time
       Min Condition Arrival Time     Max Condition Arrival Time
-------------------------------------------------------------------------------------
Clock  Early_r   Early_f   Late_r   Late_f  Early_r  Early_f  Late_r   Late_f  Ref_pin
-------------------------------------------------------------------------------------
CLK    1.00      1.00     2.00     2.00   1.00     1.00     2.00     2.00   u1/A

  Clock Transition Time
                                                                  Reference
Clock     Min_rise    Min_fall    Max_rise    Max_fall            Leaf-pin
-----------------------------------------------------------------------------
CLK         0.00        0.00       0.00        0.00                u1/A
```

```
1
```

You must specify at least the name of the block scope file. It is not necessary for any design to be loaded.

The block scope file can contain information from multiple blocks. In that case, you can restrict to the report to a particular block by specifying the block name with the `-block_name` option, or to particular block instances in the current design with the `-instance` option. For example,

```
pt_shell> report_scope_data myblock/ilm.scope -instance B1
...
```

This generates a report on the block scope data for block instance B1 in the current linked design.

By default, the `report_scope_data` command reports the types information listed in the `hier_scope_check_defaults` variable. To override the default list, use the `-check_types` option. To restrict the report to specific ports or clocks, use the `-port_names` or `-clock_names` option, and specify the list of port names or clock names as defined in the original model. For more information, see the `report_scope_data` command man page.

Updating Block Scope Data

To change or update your block scope data files, use the `update_scope_data` command. This command allows you to remove the scope data for specific blocks when that information is no longer needed. This command also allows you to merge two or more block scope files into one file.

The `update_scope_data` command changes only the file being updated. It does not change any other files.

Note:
Any changes made to the file being updated are not reversible, so you may wish to make a backup copy of the current file.

For more information, see the `update_scope_data` command man page.

Checking the Block Scope at the Chip Level

To verify that the context of a timing model is correct in the top-level design, the top-level design using the timing models must be loaded and linked. Then you can use the `check_block_scope` command to perform the scope check. For example,

```
pt_shell> read_verilog mychip.v
pt_shell> link_design
pt_shell> source myconstraints.pt
pt_shell> update_timing
pt_shell> check_block_scope -instances "B1 B2" \
          myblock/ilm.scope

****************************************
Report : scope_check
    -instance blockA
    -scope_scenario <default>
Design : top
Version: V-2004.06-Alpha1
Date   : Tue Mar 30 15:18:36 2004
****************************************
1. Matching top-level and block-level clocks.
   (Information) Found top-level clock 'CLK' matching block-level
clock 'CLK'.
   Clock mappings

       Top-level       Block-level     Ref_pin
   ------------------------------------------------------------
       CLK             CLK             blockA/u1/A         (MET)


2. Checking interclock relationships at block boundary.

   interclock arrival skews adjusted by uncertainty

   From       To        Type      Required  Actual    Slack
   ------------------------------------------------------------
   No interclock uncertainty violations.


3. Checking arrival and transition times at block boundary pins.

   Clock min condition early rise
                             Required  Actual
   Pin           Clock       Arrival   Arrival       Slack
   ------------------------------------------------------------------
   blockA/u1/A   CLK         1.00      0.00         -1.00  (VIOLATED)
   Clock min condition late rise

                             Required  Actual
   Pin           Clock       Arrival   Arrival       Slack
   ------------------------------------------------------------------
   No violations to report.

   Clock min condition early fall

                             Required  Actual
   Pin           Clock       Arrival   Arrival       Slack
   --------------------------------------------------------------------
   blockA/u1/A   CLK         1.00      0.00         -1.00  (VIOLATED)
```

```
Clock min condition late fall

                            Required  Actual
Pin           Clock         Arrival   Arrival       Slack
----------------------------------------------------------------------
No violations to report.

Clock max condition early rise

                            Required  Actual
Pin           Clock         Arrival   Arrival       Slack
----------------------------------------------------------------------
blockA/u1/A   CLK           1.00      0.00          -1.00  (VIOLATED)

Clock max condition late rise
                            Required  Actual
Pin           Clock         Arrival   Arrival       Slack
----------------------------------------------------------------------
No violations to report.

Clock max condition early fall

                             Required  Actual
Pin           Clock          Arrival   Arrival       Slack
----------------------------------------------------------------------
blockA/u1/A   CLK            1.00      0.00          -1.00  (VIOLATED)

Clock max condition late fall

                             Required  Actual
Pin           Clock          Arrival   Arrival      Slack
----------------------------------------------------------------------
No violations to report.

Clock min rise transition

                         Required     Actual
Pin                      Transition   Transition    Slack
----------------------------------------------------------------------
No violations to report.

Clock max rise transition

                         Required     Actual
Pin                      Transition   Transition    Slack
--------------------------------------------------------------
No violations to report.

Clock min fall transition

                         Required     Actual
Pin                      Transition   Transition    Slack
--------------------------------------------------------------
```

```
   No violations to report.

   Clock max fall transition

                           Required     Actual
   Pin                     Transition   Transition    Slack
   ------------------------------------------------------------
   No violations to report.

1
```

To get a detailed report, use the `-verbose` option. For a multi-scenario analysis, use the `-scope_scenario` option to specify the scenario name. For more information, see the `check_block_scope` command man page.

A

Extracting Internal Pins

Internal pins are pins that are not defined as ports of the model. In the .db representation of the extracted model, these pins are defined as internal pins of the model core cell. The types of internal pins created by the `extract_model` command are described in this chapter.

- Clocks on Internal Pins of a Design
- Generated Clocks
- Check Pins for Clock Networks
- Check Pins for Bidirectional Ports

Clocks on Internal Pins of a Design

When a clock is defined on an internal pin of the design (not on a port), the `extract_model` command stores this timing information by creating an internal pin for each source pin of the clock. The name assigned to the internal pin is just the clock name. If the clock name conflicts with a port name, PrimeTime appends _1, _2, or something similar to make the internal pin name.

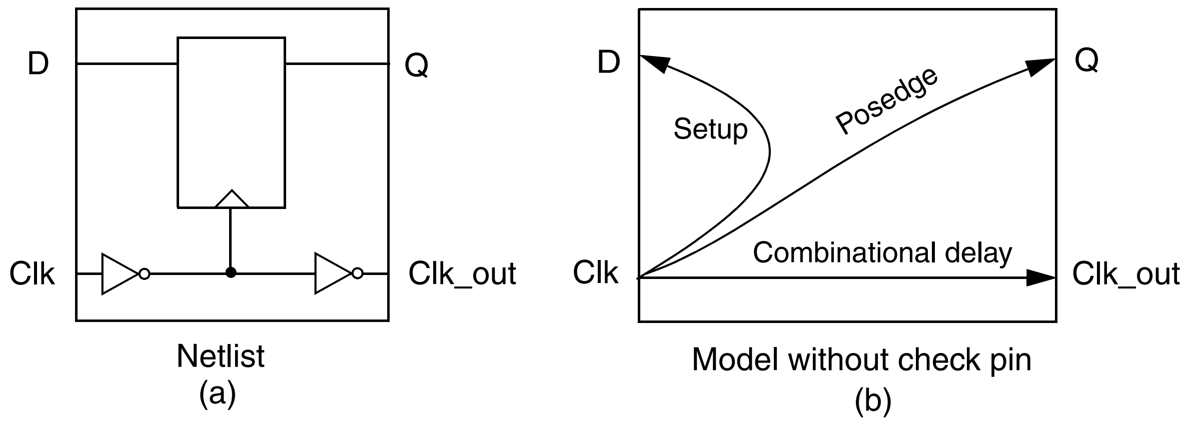
Generated Clocks

When a generated clock is defined with a generated clock source on an internal pin, the `extract_model` command stores this timing information by creating an internal pin for the source pin of the generated clock. The name assigned to the internal pin is just the generated clock name. If the generated clock name conflicts with a port name, PrimeTime appends _1, _2, or something similar to make the internal pin name.

Check Pins for Clock Networks

If the design has combinational paths in the clock network, then the extracted model has a combinational arc from the clock input to the clock output, along with setup and hold arcs from the same clock to latched inputs. Figure A-1 shows what the extracted model would look like in the absence of check pins.

Figure A-1 Original Circuit and Extracted Model Without Check Pins

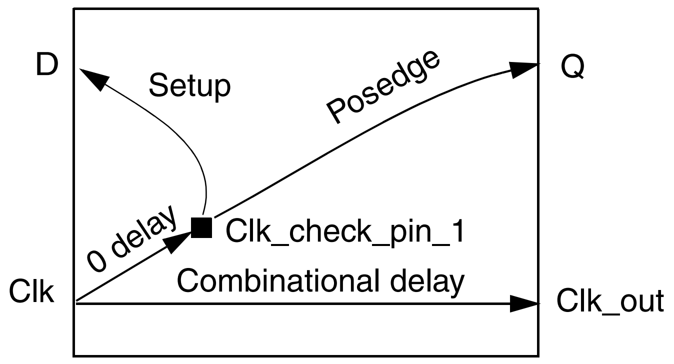


If you were to use this model in PrimeTime, the combinational delay path from the clock would not be traversed because clock tracing stops when PrimeTime detects a setup or hold arc from the clock pin.

If a clock pin or data pin of a setup constraint also has a combinational delay path originating from it, PrimeTime creates a check pin to allow clock tracing to continue to the clock output.

For example, to trace the delay path from the clock pin in Figure A-1, the model extractor creates an extra internal pin in the extracted model as shown in Figure A-2. The setup and hold arc between the clock pin and the input is changed to be between the check pin and the input pin. This causes clock tracing to stop at the check pin, while allowing clock tracing to continue from the real clock input pin to the clock output.

Figure A-2 Model With Check Pin



For the first constraint needing to be moved, PrimeTime names the check pin as follows:

transformed_pin_name`__check_pin_1`

For the second constraint needing to be moved on the same transformed pin, PrimeTime names the check pin as follows:

transformed_pin_name`__check_pin_2`

In this example, the transformed pin name is the clock port name. In other cases, it is the data pin name for the data pin of a setup constraint that also has a combinational delay path from the pin.

The check pins can only be seen in the .db formatby extracting directly to the .db format.

Check Pins for Bidirectional Ports

If a model has one or more combinational arcs to an INOUT (bidirectional) pin and one or more constraint arcs to that same INOUT pin, the .db writer of `extract_model`creates an internal check pin to handle different paths through the bidirectional pin.

The .db writer alters the arcs as follows:

- It creates a zero-delay arc from the INOUT pin to the check pin.
- It moves the constraint arcs that formerly ended at the INOUT pin so that they end at the check pin.

B

Non-Shielded PrimeTime SI ILM Flow

A non-shielded block is a block in which there is coupling from the top-level down into the block. By using the flow outlined in Figure B-2, the block has coupling from the top-level into the block.

This appendix includes the following sections:

- Hierarchical Crosstalk Analysis
- Generating the Block Context
- Context Files
- Top-Level Design Files
- Blocks Below the Top Level
- Block-Level Analysis and Timing Model Generation
- Top-Level Analysis
- Analysis Iterations Using Annotated Arrival and Slew
- Using an ILM

Hierarchical Crosstalk Analysis

You can do hierarchical crosstalk analysis with PrimeTime SI using timing models, taking into account any crosstalk between different hierarchical blocks or between a block and the top level. The analysis flow involves the creating the context or wrapper surrounding each block. The context includes aggressor nets and drivers that are outside the usual scope of the interface logic for the block. This information allows in-context crosstalk analysis at the block level between nets inside and outside of each block.

In Figure B-1 on page B-3, the I1, I2, and I3 blocks are modeled as interface logic models. There is crosstalk between an internal net of block I2 and the top level, and between the internal nets of blocks I2 and I3. Using ordinary interface logic models, these coupling effects would be lost. However, by generating the surrounding context for each block instance, the crosstalk effects can be maintained in the hierarchical analysis flow.

Figure B-1 Top-Level Design With Crosstalk Between Blocks

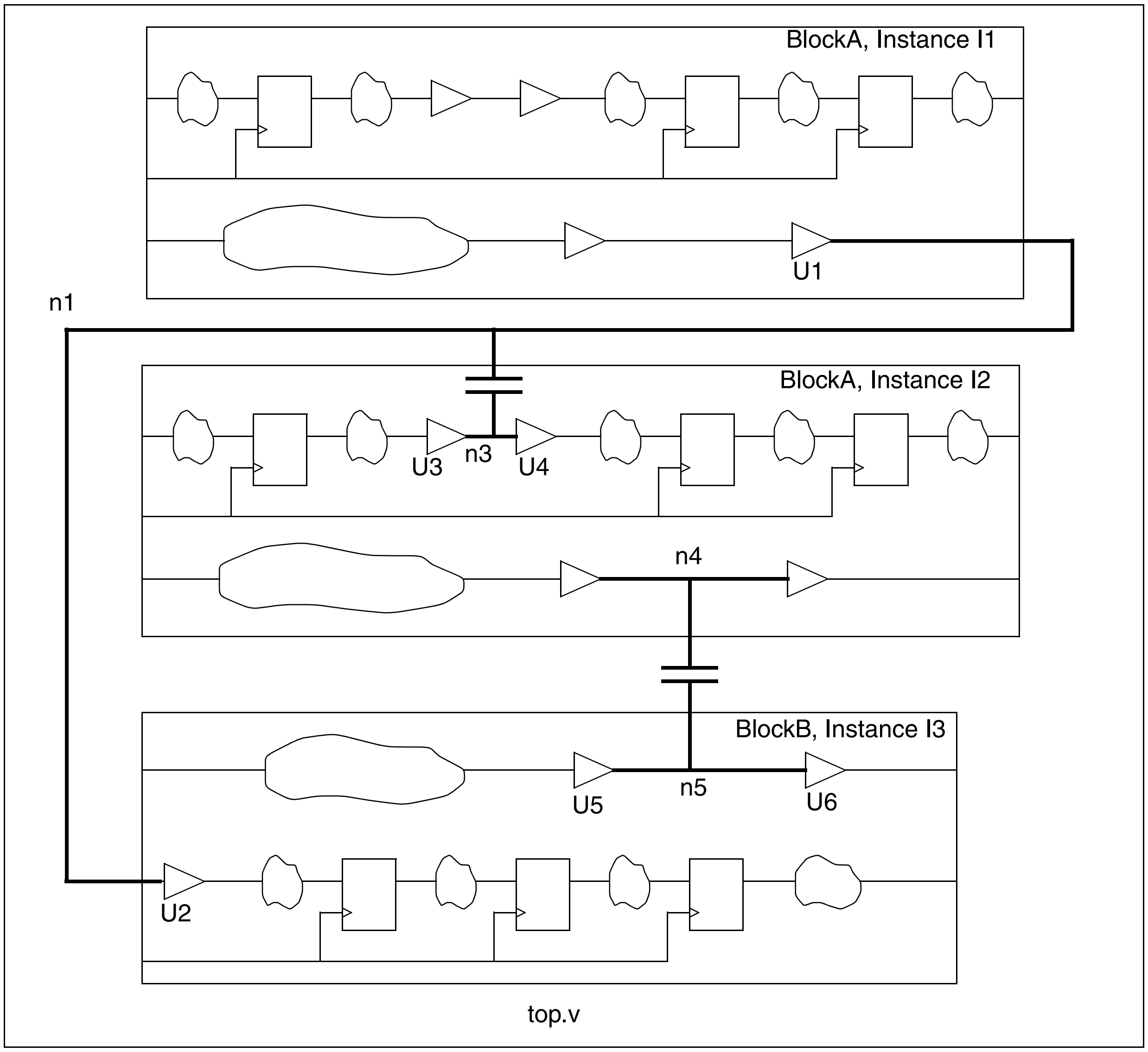
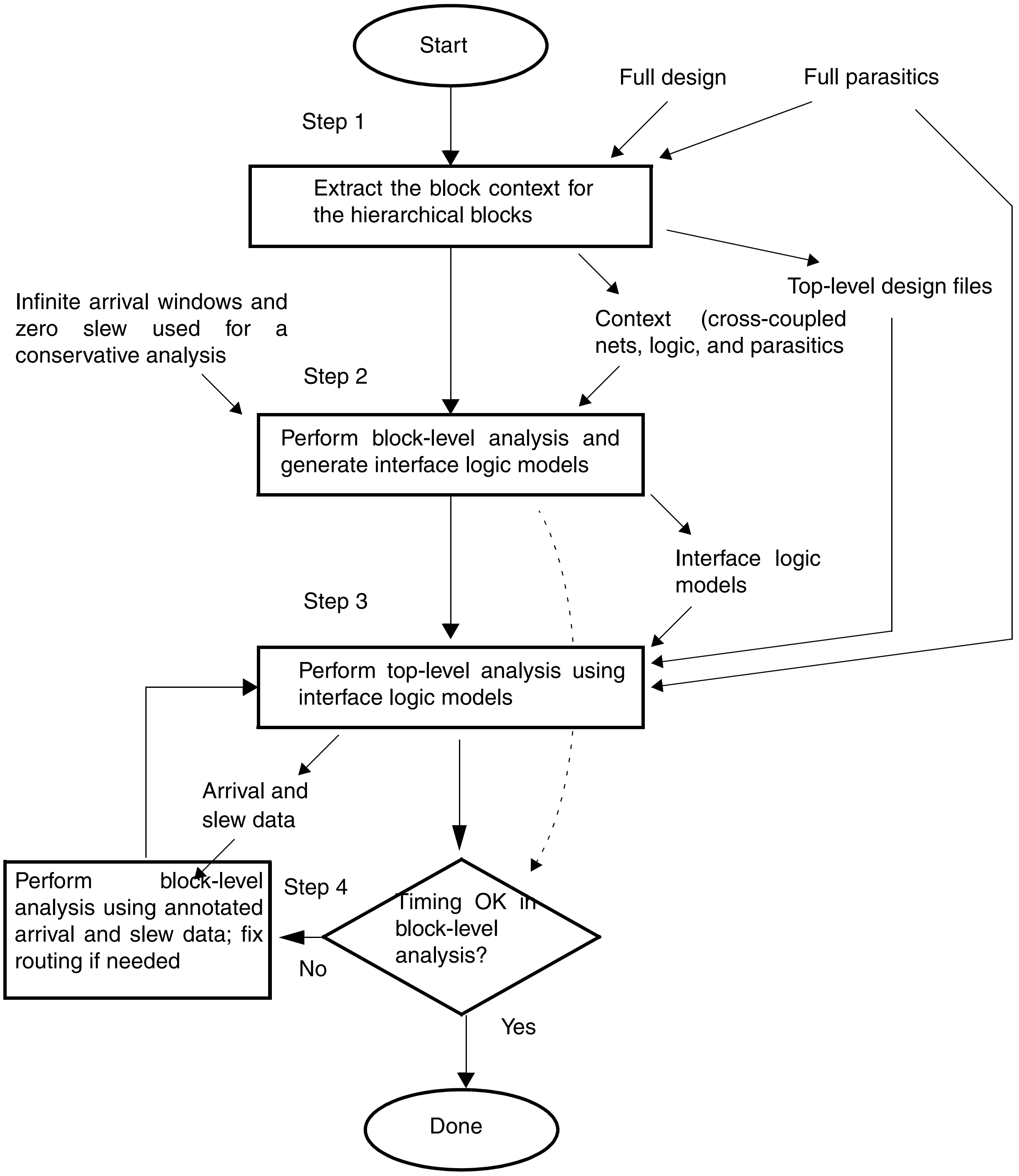


Figure B-2 shows the flow to analyze crosstalk effects between different blocks and different levels of hierarchy.

Figure B-2 Hierarchical Crosstalk Analysis Flow


Start
Full design
Full parasitics
Step 1
Extract the block context for the hierarchical blocks
Top-level design files
Infinite arrival windows and zero slew used for a conservative analysis
Context (cross-coupled nets, logic, and parasitics
Step 2
Perform block-level analysis and generate interface logic models
Interface logic models
Step 3
Perform top-level analysis using interface logic models
Arrival and slew data
Perform block-level analysis using annotated arrival and slew data; fix routing if needed
Step 4
Timing OK in block-level analysis?
No
Yes
Done

The analysis flow consists of the following:

1. For each lower-level block, generate the block context and the parasitics for the block and its context. Generate a top-level design that uses interface logic models for the lower-level blocks. For more information, see "Generating the Block Context" on page B-5.
2. For each lower-level block, perform in-context timing analysis using the block, context, and block/context parasitics. Generate the interface logic models and related data files. For more information, see "Generating the Block Context" on page B-5.
3. Perform top-level analysis using the interface logic models and parasitic data annotated on the nets still remaining in the design. For more information, see "Top-Level Analysis" on page B-14.
4. Ensure that the design met all timing constraints in the previous block-level analysis. If not, it might be due to the conservative analysis used initially for block-level analysis. To find out the issue, generate new arrival/slew information for the block contexts, repeat in-context block-level analysis for each block, and go back to Step 3 to get a more accurate analysis. If necessary, change the routing or design parameters to meet the timing requirements.

This crosstalk analysis flow uses several different files to transfer data from one step to the next. By default, PrimeTime SI writes the files to the current working directory. You can specify a different directory by setting a variable called `pt_ilm_dir`. For example,

```
pt_shell> set pt_ilm_dir "/u/john/ilm"
pt_shell> set search_path "$pt_ilm_dir $search_path"
pt_shell> set sh_source_uses_search_path "true"
```

After executing these commands, the generated files go into the specified directory path and into autocreated subdirectories.

Generating the Block Context

The context of a block is the set of the nets and associated cells outside of a block that are involved in cross-coupling with nets inside the block. To perform in-context timing analysis of a block, you need to generate a file containing the block context. This information lets you analyze the cross-coupling effects of higher-level nets and nets from other blocks acting as aggressors to the internal nets of the block.

To generate the context parasitics, you can use either an external parasitic extraction tool such as Star-RCXT or PrimeTime. To use an external tool, follow the instructions provided with the tool for generating cross-coupling parasitics for each block. To use PrimeTime, you load in the full design, back-annotate the full parasitics, and then generate the context parasitics with the `create_si_context` command.

The top-level design in Figure B-1 on page B-3 is used as an example to describe the analysis flow. The design consists of three design files: `blockA.v`, `blockB.v`, and `top.v`. The parasitics for the full design are available in a file called full_chip.spef.

The three blocks at the top level are I1, I2, and I3. They are all to be modeled as interface logic models. Blocks I1 and I2 are instances of the same block, blockA . A top-level net starts at an output of block I1 and goes to an input of block I3, and has cross-coupling capacitance to net n3 in block I2. In addition, there is cross-coupling capacitance between net n4 in block I2 and net n5 in block I3

The following script generates the context files for the three blocks:

```
read_verilog blockA.v
read_verilog blockB.v
read_verilog top.v
link_design
read_parasitics -keep_capacitive_coupling full_chip.spef
create_si_context -parasitics_options {sbpf_format}
```

The three `read_verilog` commands read in the full design. The read_parasitics command annotates the full-chip parasitics. The `create_si_context` command generates the context files and parasitics for the blocks listed in the command. Unless you specify a list of instances, the command generates context files and parasitics for all blocks at the top level.

To generate contexts for the blocks, it is not necessary to specify constraints (clocks, input delays, and so on) or to run a timing analysis. If you are unable to read in the full design parasitics due to computer memory constraints, you need to generate the context parasitics from Star-RCXT or similar tool.

The `create_si_context` command generates several different files used in the hierarchical crosstalk analysis flow. It generates parasitic data files only if you use the `-parasitics_options` option in the command. You can specify `spbf_format` or `spef_format` as the parasitic data format for the generated files. If you already have parasitics for the blocks instances from Star-RCXT or other external tool, you can omit `-parasitics_options` from the command.

In addition to generating the parasitics for the three block contexts, the command also writes out the full-chip parasitics in Synopsys Binary Parasitic Format (SBPF), as this parasitic data is needed for the top-level analysis. If you already have the parasitic data in SBPF format, you can suppress generation of the data file by using the `-no_design_parasitics` option. The files generated by the `create_si_context` command can be divided into the two categories: context files and top-level design files.

Context Files

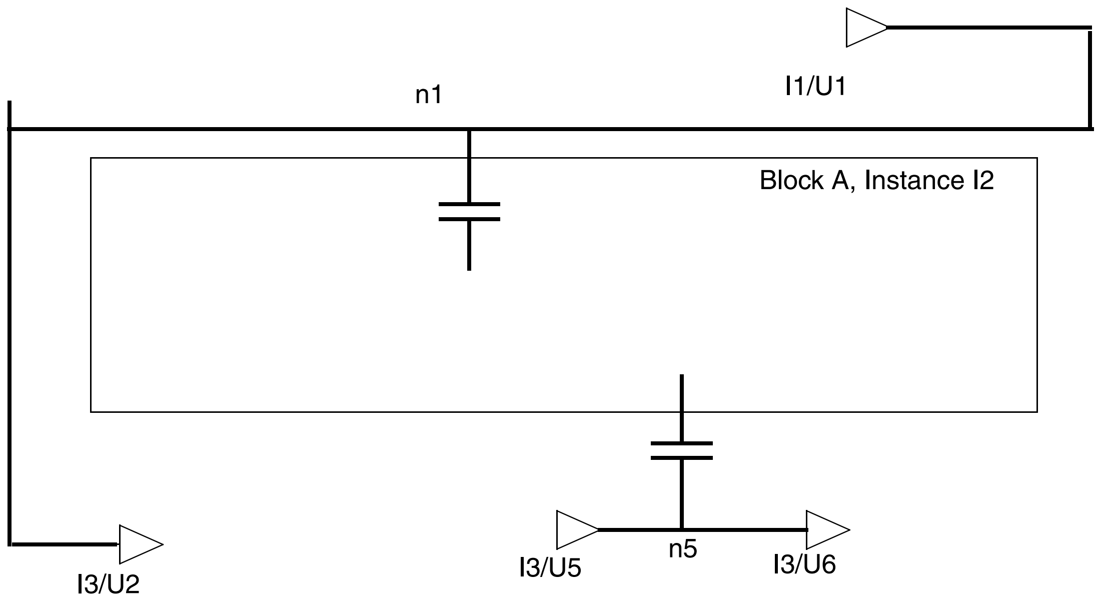
The `create_si_context` command creates the context for each block listed in the command. For the I1 block, it generates the following context files:

- I1/wrapper.v - Verilog design file for context
- I1/wrapper.sbpf - Block and context parasitics
- I1/wrapper.tcl - Tcl script for in-context, block-level analysis
- I1/wrapper.txt - List of aggressor annotation pins
- I1/wrapper.pt.gz - Arrival window annotation script

For the I2 and I3 blocks, it generates similar files and places them into directories named I2 and I3. Note that the context files for the I1 and I2 blocks are different, even though the two blocks are the same internally, because the cross-coupling capacitors to nets outside the blocks are not the same. The I1, I2, and I3 directories are created in the current working directory by default, or in the path specified by the `pt_ilm_dir` variable.

The wrapper.v file is a Verilog design file containing the block instance, the cross-coupled aggressor nets outside of the block, and the drivers and receivers of the aggressor nets outside of the blocks. Figure B-3 shows the design contained in the I2/wrapper.v file. The cross-coupling capacitors shown in light gray are later annotated on the design from the wrapper.sbpf file, along with the parasitics for the inside of the block.

Figure B-3 Contents of Context File I2/wrapper.v



The wrapper.sbpf file is a parasitic data file for the block and its surrounding context. The generated file is in Synopsys Binary Parasitic Format (SBPF). This file is used for in-context, block-level analysis.

The wrapper.tcl file is a Tcl script used to help set up the in-context, block-level analysis. For the I2 block, the I2/wrapper.tcl file contains the following lines:

```
#### You must have read the block design by now
read_verilog I2/wrapper.v
current_design I2_wrapper
link_design
set_operating_conditions -analysis on_chip_variation
read_parasitics -keep_capacitive_coupling -format SBPF I2/
wrapper.sbpf
#### Apply block level constraints now
```

The wrapper.txt file is a text file containing a list of the input pins of the drivers that are driving the aggressor nets in the context. This list can be used later in the flow to annotate arrival times and slew values on the context pins for a more accurate analysis. The I2/wrapper.txt file contains the following text:

```
I1/U1/A
I3/U5/A
```

The wrapper.pt.gz file is a gzip-compressed script file that sets infinite arrival windows on the crosstalk aggressor nets. Later in the flow, if you need to perform a more accurate (less pessimistic) analysis of a block using annotated arrival and slew times, you can overwrite this script file using the `write_arrival_annotations` command.

Top-Level Design Files

The `create_si_context` command generates the following files for the top-level design:

- top.v - Verilog design file with renamed block instances
- top.sbpf - Full-chip parasitics in SBPF binary format
- top.tcl - Tcl script for top-level analysis

The top.v file is the top-level design with the block instances renamed to avoid conflict between blocks, such as the I1 and I2 blocks (multiple instances of the same block with different cross-coupling to outside nets). The blocks are renamed by combining the block name and instance name, giving `blockA_I1`, `blockA_I2`, and `blockB_I3`. You can use this design later to generate different timing models for these blocks.

Note:
If you do not specify the destination directory with the `pt_ilm_dir` variable, the file is written to the current working directory, possibly overwriting an existing file with the same name.

The top.sbpf file contains the detailed parasitics for the whole chip in SBPF format. This file is needed for the top-level analysis. If you already have the full-chip parasitics in SBPF format, you can suppress generation of the file with the `-no_design_parasitics` option.

The top.tcl file is a Tcl script that helps set up the top-level analysis. For the example design, the file contains the following text:

```
read_verilog I1/ilm.v
read_verilog I2/ilm.v
read_verilog I3/ilm.v
read_verilog top.v
current_design top
link_design
set_operating_conditions -analysis on_chip_variation
source I1/ilm_inst.pt.gz
source I1/ilm.pt.gz
source I2/ilm_inst.pt.gz
source I2/ilm.pt.gz
source I3/ilm_inst.pt.gz
source I3/ilm.pt.gz
read_parasitics -format SBPF top.sbpf -ilm_context \
  -keep_capacitive_coupling
##### INSERT CHIP LEVEL CONSTRAINTS NOW
```

For information about how to use this script, see "Top-Level Analysis" on page B-14.

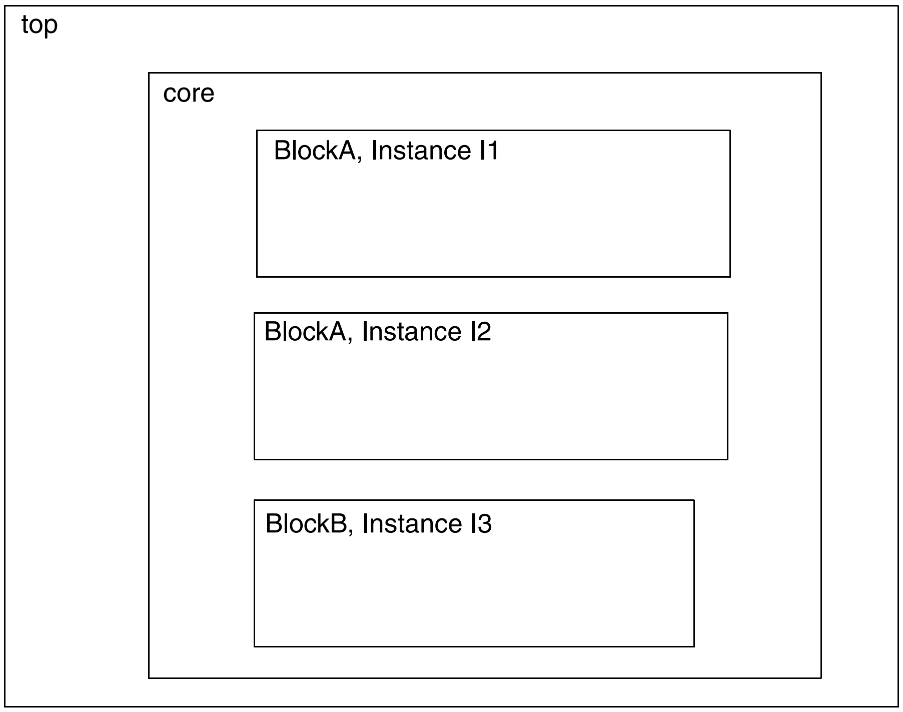
Blocks Below the Top Level

The I1, I2, and I3 blocks are at the top level of the design. If the blocks to be modeled as timing models are not at the top level, you must use the `-top_inst` option in the `create_si_context` command to specify the name of the instance containing the blocks. The blocks must be at the same level of hierarchy and must be within the same higher-level block. For example, suppose that the example design were organized as shown in Figure B-4, with the timing-model blocks one level below the top level, inside a higher-level block called `core`. The command to extract the block context is as follows:

```
pt_shell> create_si_context \
            -instances {core/I1 core/I2 core/I3} \
            -top_inst core -parasitics_options {sbpf_format}
```

This generates a core.v file with the names of I1, I2, and I3 blocks renamed to `blockA_I1`, `blockA_I2`, and `blockB_I3` within the core block.

Figure B-4 Blocks Below the Top Level



Block-Level Analysis and Timing Model Generation

After extracting the block context for each block, the next step is to perform block-level, in-context analysis of each block instance and to generate a context-accurate timing model for each block.

Block-Level Analysis

At this point of the flow, no arrival times or slew values are available for the context, so the analysis uses infinite arrival windows and zero transition times, resulting in a conservative analysis.

For each individual block instance, you can perform timing analysis with commands similar to the following:

```
pt_shell> read_verilog blockA.v
pt_shell> source I2/wrapper.tcl
pt_shell> source your_block_constraints.pt
pt_shell> update_timing
pt_shell> report_timing
pt_shell> ...
```

The `read_verilog` command reads in the original block design, not including the context. Sourcing the Tcl script surrounds the block with the context and applies detailed parasitics to the design, including the block and its context. Then you need to apply the timing constraints to the design such as clocks, input delays, and output delays. After you apply the constraints, you can run the analysis.

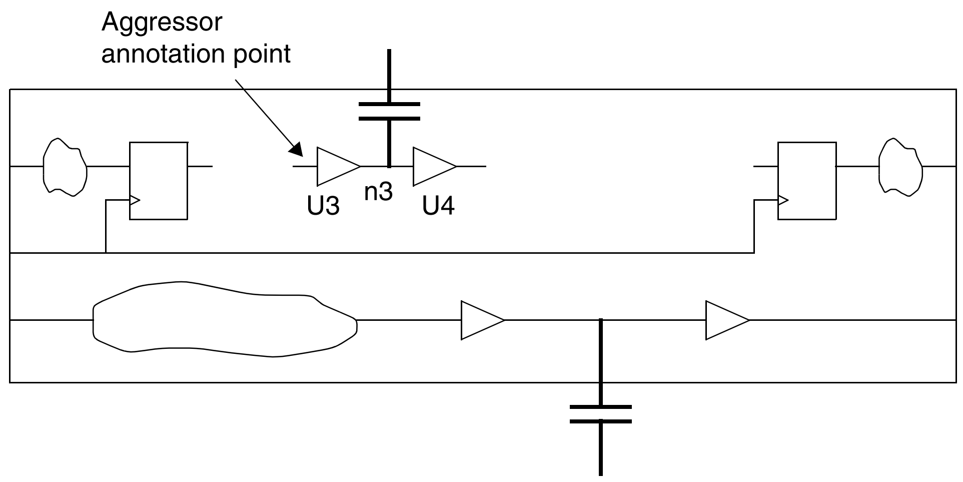
Generating ILMs for a Block Instance

To generate an ILM for a block instance, use the `create_ilm` command. For example,

```
pt_shell> create_ilm -instances {I2} -include \
  {si_delay_pins}
```

This command extracts the timing model from the I2 instance and puts the model files in the current directory by default or in the path specified by the `pt_ilm_dir` variable. Because the `-include {si_delay_pins}` option is used, the command also identifies the interface logic of the block and adds any pins needed for crosstalk analysis. The command generates the Verilog description of the timing model design as shown in Figure B-5.

Figure B-5 Interface Logic Model Created for Block I2



The U3 and U4 cells and n3 net are included in the interface logic because they are involved in cross-coupling with an aggressor net outside of the block. The generated timing models can be different for different instances of the same block, such as the I1 and I2 blocks because of different cross-coupling or different options used in the `create_ilm` commands.

The `create_ilm` command also generates a PrimeTime script to help set up usage of the model and a list of aggressor annotation points. These are the files generated by the command:

- I2/ilm.v - Verilog design file for the interface logic model
- I2/ilm_inst.pt.gz - Script to apply the instance constraints
- I2/ilm.txt - List of aggressor annotation points

Optionally, you can generate a verification script, parasitic data, or SDF data by using the appropriate options in the `create_ilm` command. You can generate the following additional files:

- I2/ilm_verif_pt.gz - Model verification script
- I2/ilm.sbpf or I2/ilm.spef - Parasitic data
- I2/ilm.sdf - Delay data in SDF format

The `create_ilm` command can perform all of the functions of the following ILM commands:

```
identify_interface_logic
write_ilm_netlist
write_ilm_script
write_ilm_parasitics
write_ilm_sdf
```

It can also perform functions that are not available in the ILM commands, such as generating a timing model for just one instance in the design or including specific parts of the netlist that are not part of the interface logic. For more information, see the `create_ilm` command man page.

Annotating Arrival and Slew Information

The list of aggressor annotation points in the I2/ilm.txt file (generated by the `create_ilm` command) can be used to annotate arrival times and slew values at the timing model pins for a more accurate top-level analysis. To annotate this information, you can use the `write_arrival_annotations` command. For example,

```
pt_shell> write_arrival_annotations -instances {I2}
```

For the I2 block, the I2/ilm.txt file contains the name of one pin, the U3/A pin. The `write_arrival_annotations` command writes out the arrival and transition time information as a script file named I2/ilm.pt.gz, which can be used for top-level analysis. The annotations are written with respect to a leaf-level pin in the clock network of the arrival window clock. For example, if leaf_clk/A is the first leaf-level pin found that is connected to the block-level clock source named clk, the `write_arrival_annotations` command creates a script similar to the following:

```
set pin [get_pin u3/A]
set_input_delay -rise -max \
  -reference_pin leaf_clk/A 23.1968 $pin
set_input_delay -fall -max \
  -reference_pin leaf_clk/A 23.1464 $pin
set_input_delay -rise -min \
  -reference_pin leaf_clk/A 14.1356 $pin
set_input_delay -fall -min \
  -reference_pin leaf_clk/A 14.0872 $pin
set_annotated_transition -rise 0.194342 -max $pin
set_annotated_transition -fall 0.137126 -max $pin
set_annotated_transition -rise 0.194429 -min $pin
set_annotated_transition -fall 0.137327 -min $pin
```

At the top level, the actual clock source latency that reaches the pin leaf_clk/A is automatically applied to the arrival window of the u3/A pin.

If multiple clocks are defined at same source or multiple clock sources share the same net, the first level leaf pin has multiple clocks reaching it. In that case, the annotations are written with respect to a virtual clock created by the script. The virtual clock has the same waveform as the block-level clock. You might want to modify the source latency of this virtual clock to fit the conditions of the top-level analysis.

Validating an Interface Logic Model

You can validate a generated ILM against the original netlist for the block with the `write_interface_timing` and `compare_interface_timing` commands. To verify an ILM, use a simple wire load model rather than detailed parasitics because parasitics are not generated in the hierarchical crosstalk analysis flow. When you use the `create_ilm` command, use the `-verification_script` option to generate a script for model validation.

This is the recommended flow for validating ILMs:

1. Read in and link the block-level design.
2. Source the script that applies the block constraints.
3. Write the interface timing report:

   ```
   pt_shell> write_interface_timing net.rpt
   ```

4. Remove the design (or open another pt_shell).
5. Read in and link the interface logic model.

6. Source the script that applies the model verification constraints.
7. Write the interface timing report:

```
pt_shell> write_interface_timing ilm.rpt
```

8. Generate the comparison report:

```
pt_shell> compare_interface_timing net.rpt ilm.rpt \
              tolerances
```

For more information about timing model validation, see Chapter 5, “Model Validation.”

Top-Level Analysis

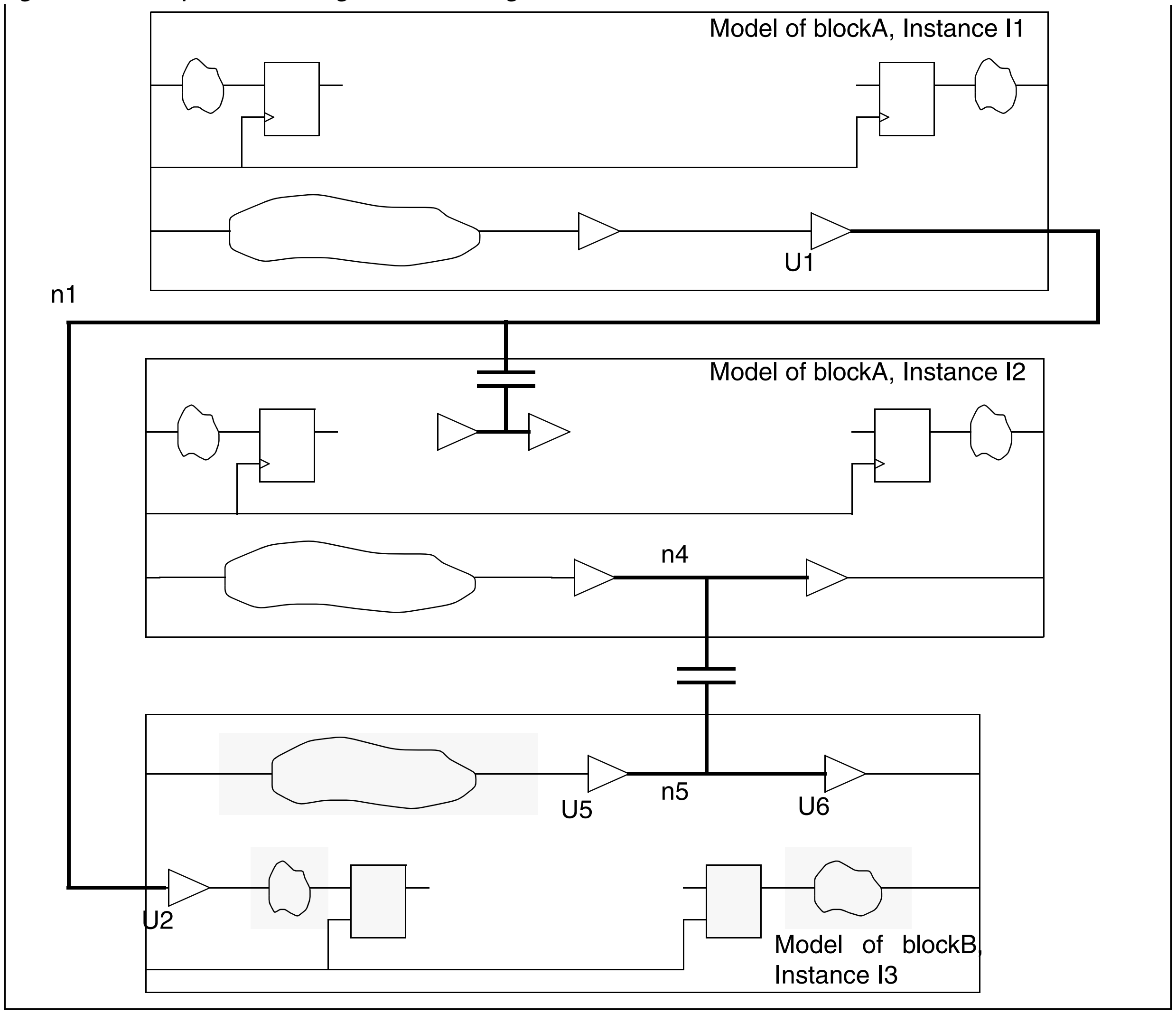
To perform top-level analysis, source the top-level analysis script created by the `create_si_context` command, apply the top-level constraints, and perform the analysis. For example,

```
pt_shell> source top.tcl
pt_shell> source your_top_contraints.tcl
pt_shell> update_timing
pt_shell> report_timing
pt_shell> ...
```

The `top.tcl` script reads in the top-level model and interface logic models, applies the instance-level constraints, applies the arrival window annotation script, and annotates the design with detailed parasitics.

Figure B-6 shows what the top-level design looks like. The register-to-register paths in the timing models have been eliminated, except in places where they are needed for crosstalk analysis. The `read_parasitics` command in the `top.tcl` script annotates the nets in the top-level design and in the interface logic models of each block. The `-ilm_context` option of the `read_parasitics` command suppresses messages about missing objects (nets, cells, pins) that are present in the parasitic data file, but not used in the design.

Figure B-6 Top-Level Design With Timing Models


Model of blockA, Instance I1
U1
n1
Model of blockA, Instance I2
n4
n5
U5
U6
U2
Model of blockB, Instance I3

Analysis Iterations Using Annotated Arrival and Slew

If there are timing violations reported in a block-level analysis (Step 2 in Figure B-2 on page B-4), it might be due to the conservative analysis used initially, with infinite arrival windows for aggressor transitions and zero transition times.

For a more accurate analysis, you can back-annotate the newly calculated arrival times and transition times for the context cells, and then run the block-level analysis again for each lower-level block. It is not necessary to generate ILMs again for this process. For example, this command annotates the timing model pins for the I2 block with new arrival and transition times:

```
pt_shell> write_arrival_annotations -instances {I2} \
              -context
```

The command uses information in the I2/wrapper.txt file and produces a new script file called I2/wrapper.pt.gz, which contains annotations like the ilm.pt.gz file. This is the same process described in "Annotating Arrival and Slew Information" on page B-12. At this point of the flow, however, you might want to annotate all of the contexts of the top-level instances. To do so, use the following command:

```
pt_shell> write_arrival_annotations -context
```

You can then repeat the block-level and top-level analysis with greater accuracy.

Using an ILM

After you have generated and verified the ILM, you can use it at the chip level in place of the full gate-level implementation for timing analysis. To use the generated model,

1. Read in the netlist for each interface model. For example, enter

   ```
   pt_shell> read_verilog block_model.v
   ```

2. Read and link the chip-level design:

   ```
   pt_shell> read_verilog top.v
   pt_shell> link_design top
   ```

3. For each ILM, read in the SDF or parasitics for the block. To specify the hierarchical path name leading to the instance, use the `-path` option with the `read_sdf` and `read_parasitics` commands. Enter

   ```
   pt_shell> read_parasitics -path block -increment \
                 block_model.spef
   ```

   or

   ```
   pt_shell> read_sdf -path block block_model.sdf
   ```

4. For each ILM, set the current instance to the block and source the script containing assertions and exceptions defined on the block. Use the script that was generated using the `write_ilm_script` command with the `-instance` option. If you need to write any exceptions (those defined relative to clocks on ports), do so before sourcing the script. Enter

   ```
   pt_shell> current_instance top/block
   pt_shell> source block_instance.pt
   ```

5. To go back to the top-level netlist, enter

   ```
   pt_shell> current_instance
   ```

6. Perform chip-level timing analysis.

   ```
   pt_shell> report_timing
   ```

Index

A

absolute tolerance 5-10
arc delays
  computing 2-4

B

back-annotated delay extraction 4-10
block context (crosstalk analysis) B-5
block scope checking 6-5
  chip level 6-10
  files 6-8
  generating files 6-8
  reporting 6-9
  updating files 6-10
block scope files 4-11

C

check pins A-2
check_block_scope command 6-5, 6-6, 6-10
check_timing command 4-24
clock-gating extraction 4-30
commands
  check_block_scope 6-5, 6-6, 6-10
  check_timing 4-24
  compare_interface_timing 5-2
  create_ilm 3-5, 6-7, B-11
  create_qtm_delay_arc 2-4, 2-9
  create_qtm_drive_type 2-3, 2-8
  create_qtm_load_type 2-3, 2-8
  create_qtm_model 2-7
  create_qtm_path_type 2-3, 2-7
  create_qtm_port 2-4
  create_si_context B-5
  extract_model 4-1, 4-16, 6-7
  remove_annotated_check 4-39
  remove_annotated_delay 4-39
  report_etm_path 5-11
  report_qtm_model 2-9
  report_scope_data 6-5, 6-9
  save_qtm_model 2-9
  set_clock_latency 6-6
  set_clock_uncertainty 6-6
  set_qtm_global_parameter 2-3, 2-8
  set_qtm_port_drive 2-4
  set_qtm_port_load 2-4
  set_qtm_technology 2-3, 2-7
  update_scope_data 6-10
  write_arrival_annotations B-12, B-15
  write_interface_timing 5-2, 5-7
compare_interface_timing command 5-2
  report 5-8
comparison tolerance 5-9
computing arc delays 2-4

context, block (crosstalk analysis) B-5
create_ilm command 3-5, 6-7, B-11
create_qtm_delay_arc command 2-4, 2-9
create_qtm_drive_type command 2-3, 2-8
create_qtm_load_type command 2-3, 2-8
create_qtm_model command 2-7
create_qtm_path_type command 2-3, 2-7
create_qtm_port command 2-4
create_si_context command B-5
creating
  a model 2-7
  interface logic models 3-5
crosstalk
  block context B-5
  hierarchical analysis B-2
  in extracted models 4-24

D

.db file 2-2
debugging
  comparison failures 5-11
delay tables 4-18

E

extract a timing model for a design 4-40
extract_model
  internal pins A-1
extract_model command 4-1, 4-16, 6-7
  back-annotated delays 4-10
  back-annotated information 4-37
  back-annotation guidelines 4-39
  boundary net back-annotation 4-38
  boundary nets 4-6
  cell back-annotation 4-39
  check_timing command 4-24
  clock definition 4-22
  clock path extraction 4-8
  clock-gating checks 4-30
  constant value extraction 4-32
  crosstalk analysis 4-24
  data path extraction 4-7
  delay tables 4-18
  Design Compiler 4-16
  example schematic and timing arcs 4-26
  false path extraction 4-33
  false paths 4-10, 4-23
  formats 4-25
  internal input/output delays 4-24
  internal net back-annotation 4-38
  internal nets 4-6
  latch extraction 4-8, 4-23
  .lib format limitations 4-16
  -library_cell option 4-16, 4-26, 4-28
  limitations 4-14
  minimum period 4-9
  minimum pulse width 4-9
  miscellaneous extracted information 4-12
  mode information 4-24
  model types 4-16
  multicycle path extraction 4-34
  noise characteristics 4-11
  operating conditions 4-21
  preparation steps 4-20
  process 4-4
  requirements 4-3
  restricting arc types extracted 4-36
  test cell 4-17
  -test_design option 4-26, 4-29
  timing exception extraction 4-33
  usage guidelines 4-14
  usage procedure 4-40
  variable usage guidelines 4-18
  variables considered 4-14, 4-17
  wire load models 4-22
extract_model_gating_as_nochange variable 4-32
extract_model_...limit variables 4-19
extract_model_num_...points variables 4-19
extract_model_use_conservative_current_slew variable 5-4

extracted model types 4-16
extraction procedure 4-40
extraction variables
  setting 4-20

F

false path extraction 4-10

G

global parameters, setting 2-7

H

hier_scope_check_defaults variable 6-5, 6-6
hierarchical crosstalk analysis B-2
hierarchical scope checking
  block scope files 6-8
  chip level 6-10
  clock arrival 6-6
  clock transition 6-6
  for crosstalk 6-7
  interclock skew and uncertainty 6-6
  overview 6-2
  types 6-5
  waveform 6-6

I

interface logic models
  adding pins 3-7
  annotating arrival and slew information B-12
  creating 3-5
interface timing report 5-4
  verifying 5-14
internal pins A-1
  check pins A-2
  clock information A-2
  generated clocks A-2

L

latch extraction 4-8
.lib format, limitations 4-14
looking at a model 2-9

M

model
  creating 2-7
  specifying information 2-3
model extraction 4-1
  overview 1-5
model validation 5-1
  arc types 5-6
  commands 5-2
  interface timing models B-13
  procedure 5-2

N

no-change arc 4-31
noise characteristics in extracted models 4-11

P

percent tolerance 5-10
pt_ilm_dir variable B-5

Q

quick timing model 1-7
  command summary 2-11
  computing arc delays 2-4
  creating 2-2
  design flow 2-2
  global parameters 2-3
  specifying mode information 2-8
  using 2-10

R

remove_annotated_check command 4-39
remove_annotated_delay command 4-39
report_etm_path command 5-11
report_qtm_model command 2-9
report_scope_data command 6-5, 6-9

S

save_qtm_model command 2-9
scope checking
  block scope files 6-8
  chip level 6-10
  clock arrival 6-6
  clock transition 6-6
  for crosstalk 6-7
  interclock skew and uncertainty 6-6
  overview 6-2
  types 6-5
  waveform 6-6
set_clock_latency command 6-6
set_clock_uncertainty command 6-6
set_qtm_global_parameter command 2-3, 2-8
set_qtm_port_drive command 2-4
set_qtm_port_load command 2-4
set_qtm_technology command 2-3, 2-7
setting
  extraction variables 4-20
  global parameters 2-7
specifying
  mode information for quick timing model 2-8
  model information 2-3
STAMP
  converting 2-10

T

technology library 1-6
timing model
  compute arc delays 2-4
  creating quick 2-2
  design flow 2-2
  generating 4-3, 5-2
  quick 1-7
  types 1-2, 1-6
timing model extraction 4-1
tolerance
  absolute 5-10
  comparison 5-9
  percent 5-10
top files (create_si_context) B-8

U

update_scope_data command 6-10
using
  quick timing model in a design 2-10

V

validation of models 5-1
variables
  extract_model_gating_as_nochange 4-32
  extract_model_...limit 4-19
  extract_model_num_...points 4-19
  extract_model_use_conservative_current_slew 5-4
  hier_scope_check_defaults 6-5, 6-6
  pt_ilm_dir B-5

W

wrapper files (extract_block_context) B-7
write_arrival_annotations command B-12, B-15
write_interface_timing command 5-2
  report 5-4
  transparent latches 5-7