NanoTime
User Guide

Version O-2018.06, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information

©2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

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

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

Copyright Notice for the Command-Line Editing Feature

© 1992, 1993 The Regents of the University of California. All rights reserved. This code is derived from software contributed to Berkeley by Christos Zoulas of Cornell University.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright Notice for the Line-Editing Library

© 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

Contents

About This User Guide xxvi

Customer Support. xxviii

1. Overview

NanoTime Features 1-2
Usage Notes 1-5

Static Timing Analysis Overview 1-7
Delay Calculations. 1-7
Constraint Checking 1-9

The NanoTime Analysis Flow 1-12
Netlist Phase 1-14
Clock Propagation and Topology Recognition Phase 1-15
Timing Constraint Specification Phase 1-16
Path Tracing (Timing Analysis) Phase 1-18
Analysis Reporting Phase 1-19

NanoTime Documentation 1-20

NanoTime Session Management 1-22
Setup Files. 1-23
The Command Log File 1-23
Saving and Restoring a NanoTime Session 1-23
License Queuing 1-26

The NanoTime Shell Interface 1-27
Entering Commands Interactively 1-27

Using Command Scripts 1-28
NanoTime Commands 1-29
NanoTime Variables 1-30
NanoTime Error, Warning, and Information Messages 1-31
Adding Context to Messages 1-32
NanoTime Attributes 1-34

Executing Commands in Parallel or in the Background 1-35
The parallel_execute Command 1-35
Examples of the parallel_execute Command 1-37
The redirect -bg Command 1-38
Exiting With Pending Spawned Processes 1-39

Distributed Processing 1-40
Host Files for Distributed Processing 1-41
Testing the Computing Environment 1-43

Parallel NanoTime Runs 1-44
Using Instance Names to Specify Parallel Runs 1-46
Example of Parallel Runs Using Instance Names 1-46
Using File Names to Specify Parallel Runs 1-47
Example of Parallel Runs Using File Names 1-48
Managing Licenses and Computing Resources 1-49

2. Netlist Data and Analysis Setup

Design Data in the NanoTime Analysis Flow 2-2

Path Variables 2-3
The search_path Variable 2-3
The link_path Variable 2-3

Netlist Registration 2-4

Design Linking 2-5
The link_design Command 2-5
Current Design and Current Instance 2-6
Design Units 2-7
Pin and Transistor Naming Conventions 2-9
Defining Ports for SPICE Netlists 2-10
Special Device Handling 2-11
Behavioral Resistors 2-11

Linking Encrypted Device Models 2-11
Retaining Boundary Pins for Discarded SPICE Subcircuits 2-11
Linking Designs With Embedded Parasitics 2-12
Linking Designs With Models That Contain Parasitics 2-13

Linking Timing Models 2-13
Reading Library Models. 2-14
Using Models Preferentially. 2-14
Setting Maximum and Minimum Libraries 2-15
Using CCS Timing Models. 2-17
Effects of Slew Thresholds on Model Use 2-17
Calculating Lower and Upper Slew Thresholds 2-17
Interpreting Libraries With Different Settings 2-18
Precedence Rules for Slew Derate Factors 2-20
Example With Different Slew Derate Factors 2-21

Defining the Voltage Environment 2-22
Specifying Power Supply Nets and Ground Nets 2-23
Setting Voltage Values. 2-24
Analyzing Multivoltage Designs. 2-25

Design Reporting 2-26

Parasitic Data Overview 2-27
Estimating Output Loads 2-28
Capacitance on Ports 2-28
Capacitance on Nets 2-29
Capacitance Attributes 2-29
Generating Parasitic Netlist Files With an Extraction Tool. 2-30
Using Parasitics in the NanoTime Flow 2-31

Working With Parasitics Files 2-34
Using the read_parasitics Command. 2-34
Exporting and Removing Parasitics. 2-36
Checking and Reporting Annotated Parasitics 2-36
Handling Parasitics For Subcircuits. 2-38
Name Mapping for Macro Models. 2-38
Annotating Parasitics to Boundary Pins 2-39
Parasitics Associated With Pins in a Timing Model 2-39
Parasitics Associated With a Discarded Subcircuit. 2-41
Parasitics Annotated to Layout-Only Ports 2-43

Message Control for Parasitics in Black Boxes 2-44
Special Device Handling 2-44
Diodes. 2-44
Resistors. 2-45
Fingered Devices 2-45
Equivalent Nets. 2-46
Completing Partially Annotated Nets 2-47
Preserving Cross-Coupling Capacitors 2-48
Using Rail Net Contact Resistance in Delay Analysis 2-49
Including the Effects of Common Trigger Skew 2-50
Variables for Parasitic Data Analysis 2-51
Variables That Affect Parasitic Netlist Interpretation 2-51
Variables and Commands That Affect Annotation 2-54
Variables That Affect Parasitics Reduction 2-56
Variables and Commands That Affect Timing Analysis 2-56

3. Technology Data

SPICE Transistor Models 3-2
Reading SPICE Models Implicitly 3-2
Reading SPICE Models Explicitly 3-4
SPICE Model File Directives 3-5
Encrypted Device Models 3-7
Custom Modeling Interfaces 3-8
Device Models With Voltage-Controlled Voltage Sources 3-10
Reading Device Parameter Data From Parasitics Files 3-11
Modifying Transistor Drive and Transistor Parameters. 3-11
Reporting Technology Information 3-12
Setting the Technologies for Analysis 3-14
Using Macro Models 3-15
Operations at the check_design Command. 3-16
Reporting the Available Technologies 3-17
SOI Transistor Models 3-18
Partially Depleted SOI Technologies 3-18
SOI Analysis in the NanoTime Flow. 3-19

Body Voltage Range 3-20
Usage Restrictions 3-20
SOI Analysis Commands 3-21
The set_soi_parameters Command 3-21
The set_soi_transistor_type Command 3-23
Reporting SOI Parameter Settings 3-24

4. Topology Operations

Topology Recognition Overview 4-2
Recognizing Channel-Connected Blocks 4-5
Reporting Channel-Connected Blocks. 4-5
Recognizing Storage Nodes 4-9
Reporting Storage Nodes 4-12
Setting and Reporting Transistor Direction 4-13
Debugging the Data Inputs of Sequential Elements 4-17
Manually Marking and Erasing Structures 4-18
Special Purpose Marking Commands 4-19
Reporting Topology Information 4-20
The report_topology Command 4-20
The get_topology Command 4-21
Searching by Name or Pattern Matching 4-23
Marking Topologies With Automatic Pattern Matching 4-26
The Topology Database 4-27
Topology Libraries 4-28
Library Topology Creation (.libt) Files 4-30
Topology Pattern (.sp) Files 4-32
Enabling Selected Libraries and Topologies 4-33
Reporting Library Topologies 4-35

5. Clocks and Clock Networks

Clocking Operations in the NanoTime Flow 5-2
Standard Clocks 5-4
Generated Clocks 5-6

Using Generated Clocks to Propagate Clocks Through Latches 5-9
Pulse Clocks 5-10
Multiple Clocks 5-13
Synchronous Clocks 5-14
Asynchronous Clocks 5-16
Exclusive Clocks 5-16
Timing Checks with Multiple Clock Domains 5-17
Clock Latency 5-18
Propagated and Ideal Clocking 5-19
Source Latency and Network Latency 5-20
Options for Latency 5-20
Clock Uncertainty 5-22
Clock Transition Time 5-24
Clock Networks 5-25
Clock Gates 5-27
Manually Marking Clock Gates 5-28
Clock Gate Analysis 5-29
Reconvergent Clock Gates 5-30
Timing-Based Reconvergent Clock Gate Analysis 5-31
Topology-Based Reconvergent Clock Gate Analysis 5-32
Resolved Pulse Generators and Pulse Shapers 5-32
Unresolved Pulse Generators and Pulse Shapers 5-33
Controlling Reconvergent Clock Gate Analysis 5-34
Reporting Clocks and Clock Networks 5-35
The report_clock Command 5-35
The report_clock_network Command 5-36
The Clock-Gate Topology Report 5-38
The report_clock_arrivals Command 5-38

6. Recognizable Topologies

Supported Topologies 6-2
Inverter Structures 6-3
Transfer Gates 6-4

Feedback Transistors 6-6

Pulldown and Pullup Structures 6-8
Pulldown and Pullup Transistors 6-8
Cross-Coupled Pullup Transistors 6-8
Weak Pullup and Pulldown Transistors 6-9
Turnoff Structures 6-10

Latch Structures 6-12
Marking Latches Manually 6-12
Debugging Latch Data Inputs 6-14
Latch Timing Checks 6-14
Tapped Feedback in a Latch 6-15

Flip-Flop Structures 6-17

Multiplexer Structures 6-20

Domino Precharge Structures 6-21
Domino Precharge Recognition 6-22
Evaluation Clock 6-24
Embedded Latch Structures 6-26
Predischarge Domino Circuits 6-27
Domino Precharge Recognition Examples 6-28
Domino Precharge Circuit Types 6-34

RAM Structures 6-37

Register File Structures 6-38

XOR Structures 6-41

7. Differential Circuits

Differential Circuit Analysis 7-2

Differential Nets, Pins, and Ports 7-3
Differential Circuits With Enable Pins 7-3
Fingered Devices in Differential Circuits 7-4
Attributes for Objects in Differential Circuits 7-5

Differential Clocks 7-5
Creating Boundary Differential Clocks 7-5
Creating Internal Differential Clocks 7-7

Reporting Differential Clocks 7-8

Differential Skew Analysis 7-9
Determining Differential Skew Automatically 7-9
Guidelines for Analysis Setup 7-10
Using Multiple Types of Iterative Analysis 7-10
Setting Differential Skew Manually 7-11

Differential Synchronizers 7-11

Differential Latches and Flip-Flops 7-14

Cascode Voltage Switch Logic 7-16

Level Shifter Circuits 7-19
Topology Marking Commands For Level Shifters 7-19
Topology Markings for Cross-Coupled Domino Circuits 7-23
Level Shifter Marking Examples 7-25
Level Shifter Without Enable or Clamp Transistors 7-25
Level Shifter With Enable Transistor and NAND Clamp 7-26
Level Shifter With Enable and Clamp Transistors 7-27
Level Shifter With Cross-Coupled PMOS and NMOS Transistors 7-29
Level Shifter With Current Follower Output 7-31

8. Timing Constraints

Input and Output Timing 8-2
Input Delay 8-3
Output Delay 8-5
Input Transition Time 8-6
Input Drive Characteristics 8-7

Logic Constraints 8-9

Conditional Analysis 8-10
Case Analysis 8-10
Constant Propagation Through a Library Cell 8-12
Conditional Delay Arcs and Timing Checks 8-13
Reporting Conditional Statements 8-16
Conditional Receiver Capacitance in CCS Delay Models 8-17

Timing Exceptions 8-19
Timing Exception Concepts 8-19
Specifying Timing Exceptions 8-20

False Path Exceptions 8-22
No-Check Exceptions 8-23
Multicycle Path Exceptions 8-26
Same-Cycle and Next-Cycle Checking Exceptions 8-31
Path-Based Settings 8-34
Object-Based Settings 8-34
Default Phasing 8-35
Intersection Transparency 8-35
Exclude Reference Path Exceptions 8-37
Reporting Exceptions 8-39
Restricting Analysis to Specified Paths 8-40
The check_design Command 8-41

9. Timing Checks

Timing Check Concepts 9-2
Timing Check Triggers 9-2
Timing Check Targets 9-4
Library-Based Timing Checks 9-6
Latch Setup and Hold Checks 9-7
Clock-Gating Setup and Hold Checks 9-12
Minimum Pulse Width Checks 9-14
Domino Precharge Timing Checks 9-15
Type D1 Domino Stage Setup and Hold Checks 9-16
Type D2 Domino Stage Setup and Hold Checks 9-18
Domino Precharge Checking Options 9-21
User-Specified Timing Checks 9-21
User-Specified Sequential Checks 9-22
User-Specified Nonsequential Checks 9-22
Modifying Timing Checks 9-24
Reporting and Analyzing Timing Checks 9-25
The report_analysis_coverage Command 9-25
The report_fanin Command 9-28

10. Timing Analysis

Path Tracing Concepts 10-2
Timing Paths 10-2
Simulation Units, Timing Arcs, and Timing Points 10-4
Timing Delay, Transitions, and Adjustments 10-6
Timing Checks 10-6

Effects of Timing Models on Delay Analysis 10-7
Nonlinear Delay Timing Models 10-7
Composite Current Source Timing Models 10-8

Invoking and Controlling Path Tracing 10-9
Increasing the Number of Saved Paths 10-10
Reducing the Number of Saved Paths 10-10

Path Reporting 10-11
The Path Report List Format 10-11
The Path Report Endpoint Format 10-12
The Path Report Detail Format 10-12
The Path Header Section 10-16
Properties That Always Appear in the Path Report 10-16
Optional Properties in the Detail Report 10-17
Paths That Contain Generated Clocks 10-18
Specification of Paths to Report 10-19
Path Selection Using Timing Points 10-20
Latch Error Recovery in Path Reports 10-20
Custom Path Reports 10-21

Multi-Input Switching 10-23
User-Guided Multi-Input Switching 10-23
Guidelines for User-Guided Multi-Input Switching 10-23
Specifying Minimum or Maximum Delay Analysis 10-24
Assumptions for User-Guided Multi-Input Switching 10-24
Maximum Delay Analysis Examples 10-24
Minimum Delay Analysis Examples 10-27
Timing-Based Multi-Input Switching 10-30
Excluding Nets From MIS Analysis 10-31
Multi-Input Skew Capture 10-32
Restrictions on Correlated Regions 10-33
Multi-Input Switching in the Path Report 10-34

Multi-Input Switching With the write_spice Command 10-35

Controlling Accuracy 10-36
Initial Condition Adjustment 10-37
Pessimism Reduction for High-Impedance Nets 10-38
Parasitics Options That Affect Accuracy 10-38
Accuracy Considerations When Using Timing Models 10-40
Using an External Simulator for Selected Timing Arcs 10-40
Nonlinear Waveform Analysis 10-44
Extended Sidebranch Analysis 10-47
Predriver Mix Ratio 10-48
Load and Miller Capacitance Settings 10-49
The sim_miller_use_active_load Variable 10-49
The sim_miller_use_extended_load Variable 10-50
The Direction Check Variable and Option. 10-51
The Fanout Limit Variables 10-51
The Transistor Side Load Variable and Option 10-52
The si_move_miller_caps_into_fets Variable 10-53
Miller Analysis for Circuits With Complementary Signals 10-54

Controlling Runtime 10-54
Identifying Parallel Stacks 10-54
Blocking Path Tracing Through Redundant On-Chains. 10-55
Handling Reconvergent or Duplicated Logic 10-56

11. Correlation with HSPICE Analysis

Overview of the write_spice Command 11-2
Problematic Paths for SPICE Analysis 11-3
Using SPICE Analysis for Debugging 11-3

The write_spice Output File 11-4
Arc0 Subcircuit. 11-5
Arc1 Subcircuit 11-6
Top-Level Netlist 11-7
Control and Measure Statements 11-7

Advanced SPICE Deck Features 11-8
Active and Inactive Transistors 11-9
Voltage-Controlled Voltage Sources 11-11

Including Transistor Models in a SPICE Deck 11-12

The Header File Option 11-12
The Model Cards Option 11-13
The Wrapper Subcircuit Option 11-14

Adding Optional Measure Statements to a SPICE Deck 11-14

Effect of Signal Integrity Analysis on a SPICE Deck. 11-16
Delay Analysis 11-16
Noise Analysis 11-18
Fanout Noise Analysis 11-20

SPICE Decks for Silicon-on-Insulator Transistors 11-23

Using SPICE Decks to Debug Dynamic Clock Simulation Regions 11-23

Simulating SPICE Decks Created With the write_spice Command 11-24

12. Timing Models for Hierarchical Analysis

Overview of Extracted Timing Models 12-2
Timing Model Usage Notes 12-2

Creating a Timing Model. 12-3

Types of Timing Models 12-5

Context Dependency of Models 12-7
Context Dependency of Input Arrivals and Slew Constraints 12-7

Timing Model File Structure 12-9
Timing Arcs in the Model 12-10
Generated Clocks in the Model 12-11

Transparent Timing Models 12-11
MIL and MOL Concepts 12-12
Transparent Model Example 12-12
Timing Arcs in the Example Model. 12-15
Pass-Through Delay Arcs 12-15
Constraint Arcs 12-16
Delay Arcs. 12-17
Multiple MILs and MOLs Per Pin 12-17

Nontransparent Timing Models. 12-20

Differences Between the Path Tracing and Model Extraction Flows 12-20
Transparency in Model Creation 12-22

Data Path Restrictions 12-23
Path Checking for Boundary Transparent Topology 12-24
Transparent Input Latch 12-24
Transparent Output Latch 12-27
Properties of Paths That Contain Generated Clocks 12-29
Paths in Timing Models That Contain Errors 12-29

Specifying the Transition and Load Index Values 12-29

Controlling the Paths Saved in an Extracted Timing Model 12-30
Including Additional Paths in a Timing Model 12-31
Restricting Timing Arc Types 12-31

Debugging a NanoTime Model 12-32
Saving Extra Paths With Timing Violations 12-33

Testing a NanoTime Model 12-34

13. Advanced Timing Model Options

Merging Models 13-2
Using the merge_models Command 13-2
Conditional Statements in Models 13-3
Merging Models With Multiple Power or Ground Supplies 13-3

Boundary Parasitics 13-4
Library Cell Models 13-4
Models With Boundary Parasitics 13-5

Full-Unate and Half-Unate Models 13-7

Composite Current Source Models 13-8
Creating CCS Timing Models 13-10
Driver Waveforms for Models 13-10
Creating CCS Noise Models 13-11
Accuracy Considerations for Creating CCS Models 13-12

HSPICE Recalibration 13-13
Generating Timing Models With HSPICE Simulations 13-14
HSPICE Simulation Directories 13-15
Creating SPICE Decks 13-16
Speeding Up HSPICE Simulations 13-16

Context Characterization 13-17

The characterize_context Command 13-18
The write_context Command 13-19

14. Dynamic Simulation

Overview of Dynamic Simulation 14-2
General Dynamic Simulation Guidelines 14-2

Dynamic Clock Simulation 14-3
Principles of Dynamic Clock Simulation 14-3
Dynamic Clock Simulation Setup Procedure 14-5
Side Input Logic 14-6
Shared Pullup and Pulldown Transistors 14-7
Self-Timed Circuits. 14-7
Signal Integrity Analysis for Dynamic Clock Simulation Regions. 14-8
Reporting Clock Arrivals for Dynamic Clock Simulation Regions 14-9
Dynamic Clock Simulation Variables 14-11

Dynamic Delay Simulation 14-12
Dynamic Delay Simulation Procedure 14-13
Marking Dynamic Simulation Regions 14-14
Reporting Dynamic Simulation Regions 14-14
Dynamic Delay Simulation Vector Files 14-16
Multivoltage Analysis Using Dynamic Simulation 14-18
Dynamic Delay Simulation Commands 14-20

15. Parametric On-Chip Variation Analysis

Parametric On-Chip Variation Analysis Overview 15-2

Using POCV in the NanoTime Flow 15-2

POCV Delay Calculations 15-3
Wire Delay 15-6
Variation Parameters for Specific Timing Arcs 15-6

Variation Parameters for Transistors. 15-8
Matching Transistors to Variation Specifications 15-8
Using Wrapper Parameters to Match Transistors 15-9
Variation Values for Transistors in Series Stacks. 15-10
Variation Values as a Function of Input Slope 15-10

Variation Parameters for Library Cells 15-12
Reading Variation Parameters from Library Cells 15-12
Using Variation Parameters for Library Cells 15-13
Saving Variation Data in Extracted Timing Models 15-13
Variation Parameters for Other Devices 15-14
Recommendations for Variation Parameters 15-15
Reporting Parametric On-Chip Variation 15-16
The report_variation Command 15-16
Listing Unmatched Transistors 15-17
The report_variation_calculation Command 15-18
Variation Analysis in the Path Report 15-18

16. Path-Based Slack Adjustment Analysis

Path-Based Slack Adjustment Analysis Overview 16-2
Using PBSA in the NanoTime Flow 16-2
How Path-Based Slack Adjustment Works 16-3
Adjustments for a Single Path 16-3
Independent Wire Delay and Cell Delay Adjustments 16-6
Path Segments to Consider for Timing Checks 16-6
Setup and Hold Slack Adjustments 16-8
Variables That Control PBSA Analysis 16-9
Overriding Path-Based Slack Adjustments 16-11
Reporting Path-Based Slack Adjustment 16-12
Reporting PBSA Calculation Details 16-14
The PBSA Adjustment Options 16-14
Example PBSA Calculation Report 16-15
Common Point Definition and Pessimism Removal 16-17
Common Point Determination for Differential Circuits 16-18
Common Point Determination for Clock Gates 16-19
Reducing Common Path Signal Integrity Optimism 16-20
Variables That Control Common Point Definition 16-21

17. Signal Integrity Analysis

Signal Integrity Concepts 17-2
Cross-Coupling Models 17-4

Overview of Signal Integrity Analysis 17-5
How Signal Integrity Delay Analysis Works 17-5
How Signal Integrity Noise Analysis Works 17-7
Timing Windows for Signal Integrity Analysis 17-8
Timing Window Overlap. 17-9
Timing Windows For Multiple Aggressors 17-12
Inversion Constraints Between Aggressor Nets 17-13
Timing Windows for Dangling Nets 17-15
Setting Up Signal Integrity Analysis 17-16
Accounting for Capacitance Variability 17-17
Filtering Victim and Aggressor Nets. 17-19
Filtering Victim Nets 17-19
Filtering Aggressor Nets 17-19
Logic Constraints in Signal Integrity Analysis 17-20
Application of Logic Constraints 17-21
Types of Logic Constraints 17-22
Precedence For Aggressor Logic Constraints 17-23
Effect of Logic Constraints on Noise Analysis 17-25
SI Delay Analysis Procedure 17-26
Setting the Analysis Mode 17-26
Selecting or Excluding Nets for SI Delay Analysis. 17-27
Net Reselection 17-28
Excluding Nets From SI Analysis 17-28
Effect of Removing Nets on Capacitance. 17-29
Removing Net Exclusions. 17-30
Controlling SI Delay Iteration Exit Criteria 17-31
Iteration Count 17-33
Number of Reselected Nets 17-33
Delta Delay 17-34
Reporting SI Delay Analysis 17-35
The Convergence Report 17-35
The Crosstalk Delay Sources Report 17-37
The SI Nets Report. 17-38
SI Delays in Path Reports. 17-38
SI Noise Analysis Procedure 17-40
Differences Between the Path Tracing and Model Extraction Flows 17-41
Injecting User-Defined Noise 17-42

Calculating Noise on Nets Driven by Model Pins 17-44
Setting Noise Margins 17-45
Reporting SI Noise Analysis 17-45
The report_noise Command 17-46
The report_noise_violation_sources Command 17-47
The report_noise_coverage Command 17-50

Fanout Noise Analysis Procedure 17-51
Setting Fanout Noise Thresholds 17-52
Setting Fanout Noise Margins 17-53
Reporting Fanout Noise Analysis 17-53

18. Memory Circuit Analysis

Memory Analysis Overview 18-2
Differences From Standard NanoTime Analysis 18-2
License Requirements 18-3
Memory Analysis Modes 18-3

Using NanoTime to Analyze Memories 18-5

Memory Topology Recognition 18-7
Memory Object Creation 18-9
Topology Recognition Procedure 18-10
Bitline Tracking Circuits 18-11
Memory Cell Topologies 18-14
Six-Transistor Single-Port Cell With Differential Read and Write 18-14
Seven-Transistor Dual-Port Cell 18-16
Eight-Transistor Dual-Port Cell With Differential Read and Write 18-17
Ten-Transistor Dual-Port Cell 18-18
Eight-Transistor Dual-Port Cell With Differential Write and Single-Ended Two-NMOS Read 18-19
Peripheral Circuit Topologies 18-20
Precharge Circuits 18-20
Sense Amplifiers 18-21
Read and Write Multiplexers 18-22
Write Drivers 18-25

Clock Propagation for Memories 18-29
Clocking for Pipelined Memories 18-29
Dynamic Clock Simulation for Memory Analysis 18-31
Dynamic Clock Simulation Setup Procedure 18-31

Case Analysis for Memories 18-33
Timing Checks for Memories 18-34
Path Tracing and Timing Analysis 18-40
General Setup 18-40
Automatic HSPICE Simulation 18-41
Model Extraction 18-42
Memory Analysis Reporting 18-42
The report_memory Command 18-43
The report_bitline and report_wordline Commands 18-44
The report_measurement Command 18-45
Black Box Memory Analysis 18-47
Memory Object Creation 18-47
Topology Recognition 18-48
Example Flow 18-49
Timing Checks 18-50
Memory-Specific Syntax 18-51

19. Object Attributes

Using Attributes 19-2
Listing Attribute Names 19-2
Reporting All Attributes for an Object 19-3
Reading Attribute Values 19-3
Creating Custom Reports With Attributes 19-4
Using Paths to Generate Custom Reports 19-4
Using Arcs to Generate Custom Reports 19-6
Attributes of Object Class capture_window_edge 19-7
Attributes of Object Class cell 19-8
Attributes of Object Class clock 19-17
Attributes of Object Class design 19-21
Attributes of Object Class launch_window_edge 19-22
Attributes of Object Class lib 19-22
Attributes of Object Class lib_cell 19-23

Attributes of Object Class lib_pin 19-24
Attributes of Object Class lib_topology 19-25
Attributes of Object Class memory 19-26
Attributes of Object Class model_clock_domain 19-27
Attributes of Object Class model_delay_arc 19-28
Attributes of Object Class net 19-29
Attributes of Object Class parasitic_device 19-42
Attributes of Object Class path_arc 19-45
Attributes of Object Class pin 19-47
Attributes of Object Class port 19-56
Attributes of Object Class simulation 19-68
Attributes of Object Class timing_check 19-69
Attributes of Object Class timing_path 19-72
Attributes of Object Class timing_point 19-78
Attributes of Object Class topology 19-79

Appendix A. Tcl Command Interface

Tcl Syntax and NanoTime Commands A-2
Redirecting and Appending Output A-4
Command Aliases A-5
Tcl Scripts A-5
Command History A-6
Suppressing Messages A-8
Variables A-8
Collections A-10
Creating Collections A-10
Using Wildcard Characters A-12
Filtering Collections A-14
Using Implicit Collections as Arguments A-16
Iterating Over the Elements of a Collection A-16

Removing From and Adding to a Collection A-17
Collection Utility Commands A-18

Lists A-20

Flow Control A-22
Using the if Command A-22
Loops A-23

Procedures A-24
Creating Tcl Procedures A-25
Displaying Tcl Procedures A-26

Appendix B. Custom Compiler User Interface

Starting and Ending a NanoTime Session B-2

Initializing a NanoTime Run Directory. B-4
Creating a New Run Directory B-4
Loading an Existing Run Directory B-5

Generating a Netlist B-6

Editing Constraints B-7

Editing Design Properties B-8

Initiating a NanoTime Run B-9

NanoTime Interactive Dialog Box B-11

Inspecting NanoTime Path Reports B-12

Detailed Path Reports. B-14

Crossprobing Objects B-16

Timing Model Generation B-17

Hierarchical Analysis. B-18

Appendix C. Example latch.lib Model File

Example latch.lib Model File C-2

Glossary

Preface

This preface includes the following sections:

- About This User Guide
- Customer Support

About This User Guide

This manual describes the Synopsys NanoTime transistor-level static timing analysis tool for block-level and full-chip timing verification.

Audience

This user guide is for engineers who perform static timing analysis on custom, semicustom, and gate-level designs.

Related Publications

For additional information about the NanoTime tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys tools:

- PrimeTime®
- StarRC™
- HSPICE ®
- Library Compiler™
- FineSim®

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *NanoTime Release Notes* on the SolvNet site.

To see the *NanoTime Release Notes*,

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

   https://solvnet.synopsys.com/DownloadCenter

2. Select NanoTime, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl 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

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to 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 sign up for an account.

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

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 support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- 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

    https://www.synopsys.com/support/global-support-centers.html
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    https://www.synopsys.com/support/global-support-centers.html

1

Overview

The NanoTime transistor-level static timing analysis tool provides fast and accurate block-level timing verification. This chapter describes the phases of the analysis flow and provides an overview of static timing analysis. In addition, it presents information essential for working with the NanoTime tool.

This chapter contains the following sections:

- NanoTime Features
- Static Timing Analysis Overview
- The NanoTime Analysis Flow
- NanoTime Documentation
- NanoTime Session Management
- The NanoTime Shell Interface
- Executing Commands in Parallel or in the Background
- Distributed Processing
- Parallel NanoTime Runs

NanoTime Features

The NanoTime transistor-level static timing analysis tool performs block-level and full-chip timing verification of custom digital designs. The tool allows you to quickly detect and correct timing violations, thereby reducing verification time and final time-to-market.

Some of the features of the NanoTime tool are as follows:

- Transistor-level delay calculation with near SPICE-level accuracy
- Direct support of many types of transistor models
- Comprehensive checking of sequential circuits (setup, hold, transparency)
- SPICE-level consideration of back-annotated detailed parasitics
- Automatic recognition of complex topologies, including domino precharge logic, latch structures, flip-flops, transfer gates, and gated clocks
- Hierarchical timing model generation and analysis, allowing chip-level analysis with high performance and virtually unlimited capacity
- Ability to create and use timing models that are compatible with other simulation tools
- Powerful and easy-to-use tool command language (Tcl) interface that supports making scripts and generating custom reports
- Support for the Synopsys Design Constraints (SDC) specification format

NanoTime Ultra is an optional add-on product that offers advanced features, including:

- Analysis of differential circuits that have traditionally required dynamic simulation, including differential clocks and clock networks
- Ability to use different supply voltages for different analysis modes, such as minimum delay analysis versus maximum delay analysis or clock path analysis versus data path analysis
- Path-based slack adjustment and parametric on-chip variation analysis, which provide two methods for including the effects of on-chip variations
- Crosstalk delay analysis, which analyzes changes in delay resulting from electrical crosstalk between capacitance-coupled nets
- Crosstalk noise analysis, which analysis noise glitches resulting from crosstalk
- SOI analysis, which accounts for the floating-body effects of SOI technologies
- Integrated HSPICE analysis of complex topographies
- Ability to generate composite current source timing and noise models

- Ability to use composite current source timing models in delay analysis
- Ability to use encrypted SPICE models
- Support for designs with virtual (gated) supplies
- Ability to analyze multiple inputs to a block that switch at the same time (multi-input switching analysis)
- An interface to the Synopsys Galaxy Custom Compiler platform
- Ability to analyze rail contact resistance

Figure 1-1 shows how NanoTime static timing analysis is used in a typical design flow.

Timing analysis requires a transistor-level description of the design in a format such as SPICE or Verilog plus SPICE. NanoTime directly reads transistor models, also known as “direct-read” models. If timing models already exist for submodules in the design, you can read them in the .lib or .db format.

The NanoTime tool traverses the design hierarchy and generates an internal representation of the design that includes the input database, the necessary transistor models, and the parasitics. The tool then examines the topology of the design to identify circuit structures such as latches, multiplexers, domino precharge logic, transfer gates, and register file cells. After you specify the timing constraints and operating conditions, the path tracing operation generates a timing database. From this database, you can produce a variety of timing reports.

NanoTime analysis includes the following checks:

- Setup and hold constraints at latches, flip-flops, gated clocks, and domino circuits
- User-specified timing checks: data-to-data checks, clock-to-data checks, and clock-gating checks
- Minimum pulse width
- Maximum clock skew
- Design rules for minimum and maximum transition time and capacitance
- Nontransparent setup and hold

Figure 1-1 NanoTime in the Design Flow

In advanced process technologies, parasitic RC effects play an important role in timing analysis. If placement and routing are complete, you can annotate the design with detailed parasitics, resulting in a timing analysis with SPICE-level consideration of parasitic effects.

The combination of the NanoTime transistor-level and PrimeTime block-level timing analysis tools provides a comprehensive timing verification solution for a wide variety of designs.

Usage Notes

Effective use of the NanoTime tool requires that you observe all requirements for supported design styles and that you follow all usage recommendations.

Unsupported Design Styles

For topologies other than digital topologies, you assume responsibility for validating the calculated delays. NanoTime analysis does not support the following design styles:

- Analog circuits, except for certain sense amplifier circuits used for embedded SRAM memories
- BiCMOS logic
- Complementary pass transistor logic
- Flash or content-addressable memory

Nonstandard Design Styles

NanoTime supports a wide variety of digital design topologies, as described in this user guide. However, unique designs might encounter unexpected results during NanoTime analysis. You must validate NanoTime delays against HSPICE delays for unusual designs, including the following:

- Topologies with very nonlinear switching characteristics
- Stages with very large transitions at their outputs
- Circuits with multiple channel-connected stages that must be simulated together (such as cross-coupled stages)
- Circuits with complex self-timed loops
- Circuits with multiple inputs that switch simultaneously
- Circuits that require initial conditions within the simulated region
- Circuits that do not have full rail to rail voltage switching
- Circuits that use the source or drain terminals of pass transistors as inputs

Accuracy and Runtime Considerations

For digital designs in new technology nodes, you must perform correlation of NanoTime delays to HSPICE delays at the beginning of a new project. To achieve the required accuracy, you might need to use one or more NanoTime features that improve accuracy at the cost of increased runtime. These features include the following:

- Active Miller simulation
- Nonlinear waveform modeling
- Dynamic simulation
- Signal integrity delay and noise analysis
- Extended sidebranch analysis
- Initial condition adjustment

Error Checking

NanoTime identifies many types of errors in circuit designs and in NanoTime scripts. The tool issues information, warning, and error messages to provide information about unexpected situations. However, your design and your NanoTime scripts might contain errors even if the NanoTime tool does not issue any messages. NanoTime cannot detect all possible design and setup errors.

Static Timing Analysis Overview

Static timing analysis validates the timing performance of a design by checking all possible paths for timing violations. To check for violations, NanoTime breaks the design into a set of timing paths, calculates the signal propagation delay along each path, and checks for violations of timing constraints inside the design and at the inputs and outputs.

Another way to perform timing analysis is to use a circuit simulator such as the HSPICE tool. It it not practical to simulate a whole design with a dynamic simulator because of the size of the design and the large number of test vectors that could be applied.

Delay Calculations

NanoTime analyzes the timing of each path in the design, from startpoint to endpoint. The startpoint is a place in the design where data is launched by a clock edge. The data traverses combinational logic and transparent latches in the path and is captured at the endpoint by another clock edge.

The locations of startpoints and endpoints in the design are determined by topology recognition. The startpoint of a path is a clock pin of a sequential element, or possibly an input port of the design (because the input data can be launched from some external source). The endpoint of a path is a data pin of a sequential element, or possibly an output port of the design (because the output data can be captured by some external sink).

Figure 1-2 shows an example of a simple design and the data paths contained in the design. In this figure, each logic cloud represents a combinational logic network. Each path starts at a data launch point, passes through some combinational logic (and possibly transparent latches), and ends at a data capture point:

- Path 1 starts at an input port and ends at the data pin of a sequential element.
- Path 2 starts at the clock pin of a sequential element and ends at the data pin of a sequential element.
- Path 3 starts at the clock pin of a sequential element and ends at an output port.
- Path 4 starts at an input port and ends at an output port.

Figure 1-2 Timing Paths

A combinational logic stage might contain multiple paths, as illustrated in Figure 1-3. NanoTime uses the longest path to calculate the maximum delay and the shortest path to calculate the minimum delay.

Figure 1-3 Multiple Paths Through Combinational Logic

NanoTime also analyzes the timing of other path types, which are illustrated in Figure 1-4:

- Clock path (a path from a clock input port or cell pin, through one or more buffers or inverters, to the clock pin of a sequential element) for data setup and hold checks
- Clock-gating path (a path from an input port to a clock-gating element) for clock-gating setup and hold checks

Figure 1-4 Path Types

Constraint Checking

After NanoTime determines the timing paths and calculates the path delays, the tool checks for violations of timing constraints such as the following:

- A *setup constraint* specifies how much time is necessary for data to be available at the input of a sequential device before the clock edge that captures the data in the device. This constraint enforces a maximum delay on the data path relative to the clock path.
- A *hold constraint* specifies how much time is necessary for data to be stable at the input of a sequential device after the clock edge that captures the data in the device. This constraint enforces a minimum delay on the data path relative to the clock path.
- A *data-to-data constraint* is a user-specified constraint between any two signals.
- A *minimum pulse width constraint* tests clock signal pulse widths.

The amount of time by which a violation is avoided is the *slack*. For example, if a signal must reach a cell input at 8 ns but is determined to arrive at 5 ns, the slack is 3 ns. A slack of 0 means that the constraint is exactly satisfied. A negative slack indicates a timing violation.

High-performance designs such as microprocessors often incorporate transparent latches. Latch-based designs might use multiple clocks with different phases to control successive registers in a data path.

For example, consider the two-phase latch-based path in Figure 1-5. All three latches are level-sensitive, active when the C input is high. Latches L1 and L3 are controlled by PH1 and latch L2 is controlled by PH2. A rising edge launches data from the latch output, and a falling

edge captures data at the latch input. In this example, consider the latch setup and delay times to be zero.

Figure 1-5 Latch-Based Paths

Figure 1-6 shows how NanoTime performs setup checks between these latches. For the path from L1 to L2, the rising edge of PH1 launches the data. The data must arrive at L2 before the closing edge of PH2 at time = 4. This timing requirement is labeled Setup 1.

Figure 1-6 Setup Checks in a Latch-Based Path

By default, NanoTime assumes that the path from L1 through L2 and arriving at L3 occurs within a single cycle. Therefore, the tool checks the data arrival at the next closing edge at

L3, at time = 2 (Setup 2a in the figure). This is a more restrictive check than the path from L1 to L2.

If L3 is designed to capture data in the next cycle rather than the same cycle, you can specify next-cycle checking behavior. In that case, NanoTime checks for the arrival of data at the closing edge of the next cycle, at time = 6 (Setup 2b in Figure 1-6).

To perform hold checking, NanoTime considers the launch and capture edges relative to the setup check. The tool verifies that data launched at the startpoint arrives late enough not to be captured by the previous capture edge (the capture edge just before the one at which the setup check is performed). This is shown in Figure 1-7.

Figure 1-7 Hold Checks in a Latch-Based Path

The NanoTime Analysis Flow

Figure 1-8 shows the phases in the NanoTime analysis flow, with two ways of referring to them (state or phase), the NanoTime commands that define the boundaries, and typical actions performed within the phases. The analysis phases are described in the following sections:

- Netlist Phase
- Clock Propagation and Topology Recognition Phase
- Timing Constraint Specification Phase
- Path Tracing (Timing Analysis) Phase
- Analysis Reporting Phase

Many commands or operations (such as setting certain variables) should be executed only during specific phases of the analysis flow. The “Command Phasing” section of a command man page describes the phases during which the command can be executed. If you attempt to execute a command during a phase in which the command is not allowed, the tool issues a warning message.

However, some commands can be executed in more than one phase. These overview sections illustrate typical flows, but variations are possible and are discussed in later sections of this user guide.

NanoTime man pages sometimes use a different naming convention to describe a point in the analysis flow. The analysis state is named for the most recently executed phase-completion command. For example, NanoTime enters the “netlist-linked” state after the `link_design` command.

Figure 1-8 NanoTime Execution Phases

Start NanoTime
Execute startup scripts

State:	Phase:	Actions:
Startup-complete	Netlist	Set up environment Read in design Read technology data

link_design

State:	Phase:	Actions:
Netlist-linked	Clock definition Topology recognition	Define clocks Mark topologies Set logic constraints

match_topology

check_topology

State:	Phase:	Actions:
Topology-checked	Constraint setting	Set timing constraints Define exceptions Annotate parasitics

check_design

State:	Phase:	Actions:
Design-checked	Path tracing	Analyze timing Extract timing model

extract_model

trace_paths

characterize_context

State:	Phase:	Actions:
Paths-traced	Reporting	Generate reports

Netlist Phase

The netlist phase is the first phase of an analysis. NanoTime reads in the design and technology data and builds an internal representation of the design for analysis, as illustrated in Figure 1-9.

The `link_design` command completes this phase. The command resolves all references between different modules in the hierarchy and builds an internal representation of the design for timing analysis.

After the design has been successfully linked, you can get specific information about input or output ports, cells, transistors, nets, connections, and so on using reporting commands such as the `report_port`, `report_cell`, and `report_net` commands.

Figure 1-9 Netlist Phase

See Also

- Chapter 2, “Netlist Data and Analysis Setup”

Clock Propagation and Topology Recognition Phase

In the clock propagation and topology recognition phase, which is illustrated in Figure 1-10, you specify the design characteristics used in topology recognition, such as:

- Clock characteristics
- Fixed logic states at inputs
- Logic constraints
- Port direction

Figure 1-10 Clock and Topology Phase

NanoTime needs clocking information to identify latches and other clocked structures.

You must specify case analysis conditions (with the `set_case_analysis` command) and logic constraints (with the `set_logic_constraint` command) in this phase only if they affect topology recognition. Otherwise, you can set these conditions either in this phase or in the timing constraint phase.

After the clocks have been specified, you use the `match_topology` command to perform automated topology recognition. This command propagates the clocks from their defined source locations and recognizes circuit structures based on their topology. For some circuits, you might also need to mark structures manually or perform multiple iterations of clock propagation and topology definition.

After the `match_topology` command, you can use the `report_topology` command and other reporting commands to find out about the structures that have been recognized. You can get detailed reports on the clocks and topology of the design, such as the reason a node was not marked as a clock or the function of a specific transistor.

The `check_topology` command performs a final topology check and completes the topology recognition phase.

See Also

- Chapter 4, “Topology Operations”
- Chapter 5, “Clocks and Clock Networks”
- Chapter 6, “Recognizable Topologies”

Timing Constraint Specification Phase

Figure 1-11 shows the steps typically performed in the timing constraint specification phase. This phase is sometimes referred to as the TAPE phase (Timing Assertions, Parasitics, and Exceptions).

The most common types of constraints are the following:

- Clock characteristics specified with the `create_clock` command
- Input delays and output delays specified with the `set_input_delay` and `set_output_delay` commands
- Input driver conditions specified with the `set_drive` command
- Input transition times specified with the `set_input_transition` command
- Path-level timing exceptions specified with commands such as `set_false_path`, `set_multicycle_path`, and `set_no_same_phase_path`
- Fixed logic conditions on pins or ports specified with the `set_case_analysis` command

- Logic constraints on nets specified with the `set_logic_constraint` command
- Custom timing checks specified with the `create_timing_check`, `set_data_check`, and `create_gated_clock_timing_check` commands
- Output loading conditions specified with the `set_load` command
- Parasitic data back-annotated with the `read_parasitics` command

You conclude this phase with the `check_design` command, which checks the design for proper structure in the context of the timing constraints that have been set. The command reports inconsistencies or structural problems, which you must fix before proceeding.

Figure 1-11 Timing Constraint Specification Phase

See Also

- Parasitic Data Overview
- Chapter 8, “Timing Constraints”
- Chapter 9, “Timing Checks”

Path Tracing (Timing Analysis) Phase

After you specify all constraints and successfully run the `check_design` command, you can proceed with timing analysis. Figure 1-12 shows the steps typically performed in the path tracing phase.

Figure 1-12 Path Tracing Phase

One of the following NanoTime commands is used to complete this phase.

- The `trace_paths` command performs static timing analysis. The most critical timing paths are saved in a database and are available for reports.

  You can control some aspects of path tracing by using the `trace_paths` command options or by setting variables before you execute the command. These options include the types of paths traced (minimum, maximum, or clock), the number and type of paths stored in the path database, latch transparency rules, and the determination of path startpoints and endpoints.

- The `extract_model` command creates a timing model for use at a higher level of hierarchy in the NanoTime or PrimeTime tools. Using timing models is much more efficient than analyzing a single large, flattened design at the top level.

- The `characterize_context` command captures the timing context of a subdesign within a higher-level design. This context information includes clock information, input arrival times, output delay times, timing exceptions, input drives, and capacitive loads. This information can be used to set the constraints for other tools.

See Also

- Chapter 10, “Timing Analysis”

Analysis Reporting Phase

After executing the `trace_paths` command, you can generate design and timing reports. Figure 1-13 shows the steps typically performed upon completion of timing analysis.

The `report_paths` command is commonly used to generate reports after path tracing. You can choose the level of detail and the number of paths reported, as well as the scope of the design to report. A typical report lists the worst-case paths in order of increasing slack. For further analysis, you can write out a SPICE netlist file that represents a selected timing path.

Figure 1-13 Analysis Reporting Phase

See Also

- Path Reporting

NanoTime Documentation

If you need help, information is available from the following resources:

- Command information displayed with the `help` command
- Man pages displayed with the `man` command
- The *NanoTime User Guide* and *NanoTime Tutorial*
- SolvNet articles

The NanoTime Tutorial

A set of instructional files, collectively called the NanoTime tutorial, is available to help you learn to use the tool. The tutorial consists of a design database, script files that demonstrate different aspects of the tool, and a PDF file that describes how to use the scripts. To get the latest copy of the tutorial, search for “NanoTime tutorial” on SolvNet at the following address:

https://solvnet.synopsys.com

Command Help

The `help` command provides concise information about NanoTime commands. You can get a partial or complete list of commands or view the syntax of a specific command.

The `help` command shows a list of NanoTime commands, organized by command group:

```
nt_shell> help
 ...
Default Command Group:
 add_to_collection, all_clocks, all_inputs, all_instances
 ...
```

You can use wildcard characters to restrict the scope of the command list or to find the name of a command that you cannot remember exactly. For example, to find all commands that contain the string “clock,” enter

```
nt_shell> help *clock*
 create_clock                     # Create a clock object
 create_gated_clock_timing_check  # Create gated clock ...
 create_generated_clock           # Create a generated clock object
 erase_clock_gate                 # Remove a clock gate structure ...
 ...
```

For a concise description of a command, enter `help` with the command name:

```
nt_shell> help set_input_delay
 set_input_delay      # Set input delay on ports or pins
```

To see the full command syntax, including options and arguments, use the `-verbose` option:

```
nt_shell> help -verbose set_input_delay
 set_input_delay     # Set input delay on ports or pins
   [-clock clock_name] (Relative clock)
   [-clock_fall]       (Delay is relative to falling edge)
   [-level_sensitive]  (Delay is from closing edge ...
   [-asynchronous]     (Delay is asynchronous)
   [-rise]             (Specifies rising delay)
    ...
```

The `help -verbose` command and the man page provide the most complete lists of options for a command. The user guide describes the most important or most complex options, but it does not describe every option for a command.

An alternate method to display the same information is to enter the command name directly and use the `-help` option:

```
nt_shell> set_input_delay -help
 set_input_delay     # Set input delay on ports or pins
   [-clock clock_name] (Relative clock)
   [-clock_fall]       (Delay is relative to falling edge)
   [-level_sensitive]  (Delay is from closing edge ...
   [-asynchronous]     (Delay is asynchronous)
   [-rise]             (Specifies rising delay)
    ...
```

Man Pages

To get descriptive information about a command, variable, or system message, use the `man` command during a NanoTime session. Type `man` followed by the command, variable, or message code.

Man pages for commands follow a standard format that includes the syntax, a description of each option and argument, a general description of the command and its usage, examples, and a list of related commands and variables.

Man pages for variables show the name, value type (string, list, Boolean, integer, or floating-point number), the default, and a description of the variable and its effects.

Man pages for error, warning, and information messages include the name, a brief description, and some suggestions for followup actions. To view the man page for an error message, use the `man` command with the message code. Type uppercase letters for the error code.

NanoTime Session Management

NanoTime runs under the Linux operating system. Before you can use it, the application must be installed and licensed at your site.

To start an interactive session, enter the `nt_shell` command at the operating system prompt.

NanoTime checks out a license and displays an initial message and the `nt_shell` prompt. Here is an example, but the message you see might be different depending on the version.

```
                        NanoTime
    Version O-2018.06 for linux64 - June 11, 2018
        Copyright (c) 1988-2018 by Synopsys, Inc.

This software and the associated documentation are proprietary to
to Synopsys, Inc. This software may only be used in accordance with
the terms and conditions of a written license agreement with Synopsys,
Inc. All other use, reproduction, or distribution of this software is
strictly prohibited.

nt_shell>
```

If you need assistance, ask your system administrator or consult the installation and licensing documentation.

If the NanoTime shell fails to start, check the following:

- Is NanoTime correctly installed?
- Is the NanoTime installation path included in your path definition?
- Is the Synopsys license server running?
- Is your Synopsys license key file available and current, and does it include a NanoTime license?

To end a NanoTime session, enter the `quit` or `exit` command at the NanoTime prompt:

```
nt_shell> exit
Maximum memory usage for this session: 0.72 MB
CPU usage for this session: 0 seconds
Diagnostics summary: 2 errors

Thank you for using nt_shell!
%
```

Setup Files

When you start a NanoTime session, the tool executes the commands contained in one or more setup files. You can put commands into a setup file to set variables, to specify the design environment, and to select your preferred working options.

The name of a setup file ends with .synopsys_nt.setup. NanoTime looks for setup files in the following directories:

- The Synopsys installation setup directory at admin/setup. For example, if the installation directory is /usr/synopsys, the setup file name is /usr/synopsys/admin/setup/.synopsys_nt.setup
- Your home directory
- The current working directory from which you started NanoTime

If more than one of these directories contains a setup file, the files are executed first in the Synopsys setup directory, then in your home directory, then in the current working directory. Typically, the file in the Synopsys setup directory contains setup information for all users at your site, the file in your home directory sets your personal preferred working configuration, and the file in your current working directory sets the environment for the current project.

To suppress execution of all setup files, use the `-no_init` option when you start NanoTime.

The Command Log File

NanoTime saves the session history in the command log file. This file contains all of the commands executed during the session and serves as a record of your work. You can repeat the session by running the file as a script, using the `source` command.

The log file is named nt_shell_command.log and it resides in the current working directory. A new log file overwrites an existing log file with the same name. Before you start a new session, rename any log files that you want to keep.

You can specify a name for the command log file by setting the `sh_command_log_file` variable in a setup file. You cannot change this variable during a session.

Saving and Restoring a NanoTime Session

You can save a NanoTime session and later restore the saved state into a different NanoTime session by using the `save_session` and `restore_session` commands.

You can save the state of the current session only after path tracing is complete. Save the session after any of the following commands:

- `trace_paths`
- `extract_model`
- `characterize_context`
- `update_noise`
- `continue_trace`

Only certain types of commands, primarily reporting functions, can be used in the restored session. The following command types are available:

- `report_xxx` for standard reporting
- `list_xxx` for standard reporting
- `get_attribute` to access object attribute values
- `get_xxx` to create collections of objects
- `all_xxx` to create collections of objects
- `write_xxx` (limited to commands that would work in a nonrestored session after the `trace_paths` command)

The save and restore feature is platform independent. You can restore the session on any supported platform, but this capability is not meant to be used as a database communication method nor as a design archival method.

The following types of data items are saved:

- NanoTime application variables
- Currently linked design
- Transistor models
- Timing exceptions and other constraints
- Timing checks
- Parasitics
- Analysis data
- Signal integrity data, if available at the time the session is saved

The current session is saved to the named directory. If the directory already exists, an error occurs unless you use the `-replace` option. When this option is present, the existing subdirectories and files are replaced with the data of the current session.

The saved files are binary files, with the exception of two ASCII files. One ASCII file is named README and contains the NanoTime version, the creation date, the current design name, and a list of logic library information. The other ASCII file is named lib_map and contains the full path names for each of the cell libraries that must be loaded, in addition to the files in this directory, to restore the session.

Note:

If the location of any of the cell libraries change, you must edit the lib_map file to provide the correct location. The order and number of cell libraries must not be changed.

When you use the `restore_session` command, the cell library files and design data needed to restore the session are read from the specified directory. The session is successfully restored only if all cell library files exist and have the same library version as when the session was saved.

Note the following guidelines for the save and restore operations:

- Data from one version cannot be restored in a different version, including service pack releases.
- Backward compatibility is not maintained.
- Tcl procedures cannot be saved or restored. You must reload Tcl procedures from your .synopsys_nt.setup file.
- What-if analysis is not supported.
- Commands that alter the timing exceptions or other constraints or require that you run the `trace_paths` command again are not allowed.
- Variables that hold collections are not saved.
- Tcl variables that are not NanoTime application variables are not saved.
- Tcl command history is not saved.

License Queuing

When all NanoTime licenses are in use, you can wait for available licenses by using the license queuing capability. To enable license queuing, set the `SNPSLMD_QUEUE` environment variable to `true`:

```
% setenv SNPSLMD_QUEUE true
```

If license queuing is enabled, the following message appears at the startup of the NanoTime shell:

```
Information: License queuing is enabled.  (LICS-007)
```

You can optionally set the maximum time to wait for an initial NanoTime license by setting the `SNPS_MAX_WAITTIME` environment variable:

```
% setenv SNPS_MAX_WAITTIME time_in_seconds
```

You can set the maximum time for an existing process to wait for a subsequent license (such as a NanoTime Ultra license) by setting the `SNPS_MAX_QUEUETIME` environment variable:

```
% setenv SNPS_MAX_QUEUETIME time_in_seconds
```

The default for the `SNPS_MAX_WAITTIME` variable is 72 hours (259,200 seconds) and the default for the `SNPS_MAX_QUEUETIME` variable is 8 hours (28,800 seconds).

The NanoTime Shell Interface

The NanoTime shell is a command-driven text-only user interface based on the Tcl scripting language. You enter commands at the `nt_shell` prompt. NanoTime carries out the action for each command and reports the results in text format.

The NanoTime command syntax is case-sensitive. Commands, command options, arguments, and variables generally consist of lowercase characters.

Object names in the design are also case-sensitive. For example, the following two commands are not the same because they refer to two different ports named Clk and CLK:

```
nt_shell> create_clock -period 20.0 [get_ports Clk]
nt_shell> create_clock -period 20.0 [get_ports CLK]
```

The nt_shell interface is based on the Tcl scripting language. You can use features of Tcl such as user-defined variables, procedures, conditional execution, lists, and expressions.

The general features of Tcl are beyond the scope of the NanoTime documentation. For this type of information, see a reference book on Tcl.

The prompts are programmable. By default, the primary prompt is `nt_shell>` and the secondary prompt is a question mark (`?`). To change the prompt, set the `tcl_prompt1` or `tcl_prompt2` variable to the name of a procedure that displays the new prompt. The procedure cannot take an argument. For example, to make the primary prompt an asterisk (`*>`), do the following:

```
nt_shell> proc prompt1 {} { echo -n "*> " }
nt_shell> set tcl_prompt1 prompt1
prompt1
*>
```

See Also

- Appendix A, “Tcl Command Interface”

Entering Commands Interactively

You can abbreviate NanoTime command names and options to the shortest unambiguous string. For example, you can abbreviate the `get_attribute` command to `get_attr`, or the command `create_clock -period 5 CLK` to `create_cl -p 5 CLK`.

Using command abbreviations is convenient for interactive sessions. However, avoid using abbreviations in script files because command changes in later NanoTime releases could make the abbreviations ambiguous.

The `sh_command_abbrev_mode` variable determines whether command abbreviation is enabled. The default is `Anywhere`; you can also set the variable to `Command-Line-Only`. To disallow all command abbreviation, set the `sh_command_abbrev_mode` variable to `None`.

If you enter an ambiguous command, NanoTime attempts to help you find the correct command. For example, the `set_min` command as entered here is ambiguous:

```
nt_shell> set_min
Error: ambiguous command 'set_min' matched 2 commands:
       (set_min_library, set_min_pulse_width) (CMD-006)
```

The error message lists up to three possible matches. To list all of the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern. For example,

```
nt_shell> help set_min_*
 set_min_library      # Set/remove min library for a library
 set_min_pulse_width  # Specify min pulse width checks
```

You can split long commands across multiple lines by using the backslash (`\`) continuation character. During entry of a long command using backslashes (or in other incomplete input situations), the tool displays the secondary prompt for each additional line of the command. The default secondary prompt is a question mark. For example,

```
nt_shell> alias my_timing_report {report_paths \
? -max -max_paths 5 -path_type full_clock_expanded}
```

In this user guide, a command that cannot fit on one line is shown on multiple lines with the continuation character. However, the secondary prompt is omitted from the examples.

Using Command Scripts

A command script is a text file containing a sequence of commands. The setup file .synopsys_nt.setup is an example of a script. Also, the log file generated at the end of a NanoTime session can be used as a script. You can create scripts to carry out complex or repetitive tasks.

NanoTime can recognize script files in plain ASCII format, ASCII compressed in gzip format, and ASCII encoded into bytecode format by the TclPro Compiler. To execute a script in any of these forms, use the `source` command:

```
nt_shell> source file_name
```

To execute a script upon startup, use the `-file` option (short form `-f`):

```
% nt_shell -f file_name
```

You can create scripts that use variables, loops, and conditional execution. The flow control commands `if`, `while`, `for`, `foreach`, `break`, `continue`, and `switch` determine the execution order of other commands.

Any line of text in a script file that begins with the pound sign (#) is a comment. Any text from a semicolon and pound sign (;#) to the end of a line is also considered comment text.

You can redirect the output to a file. The following command runs the Tcl script named big_analysis.tcl and redirects all output and error messages to the file result_file.out.

```
% nt_shell -file big_analysis.tcl > result_file.out
```

If your script contains a syntax error, NanoTime stops and waits for input unless the `sh_continue_on_error` variable is set to `true`.

End the script with the `quit` or `exit` command. Otherwise, the `nt_shell` prompt does not appear, and you do not know when the script has finished executing. If your script does not end with the `quit` command, the tool waits for input. Type `quit` or `exit` to end the session.

NanoTime Commands

Commands are statements that cause actions, such as defining values, executing analysis, or displaying reports. Commands return a 1 to indicate success and a 0 to indicate failure in cases where there is no specific resulting output. For example,

```
nt_shell> create_clock -period 6.67 [get_ports clk1]
1
```

Command examples in this user guide do not always show the return value.

For some commands, the result is a collection. For example, the result of the `get_ports` command is a collection of ports. The following command creates a collection of all ports whose names begin with the letters IN. The displayed result is the collection.

```
nt_shell> get_ports IN*
{"IN1", "IN2", "IN3", "IN4"}
```

You can nest commands so that the result of one command is an argument for another. Enclose each nested command in square brackets. For example, the `set_input_delay` command sets a timing constraint on one or more specified input ports. You can gather a collection of input ports with the `get_ports` command and pass the result to the `set_input_delay` command on one command line:

```
nt_shell> set_input_delay 2.3 [get_ports IN*]
```

The `get_ports IN*` command creates a collection, which is used as the second argument of the `set_input_delay` command. The effect is to set the input delay to 2.3 for all ports beginning with the letters IN.

The output of some commands, such as the `report_paths` command, is a report. By default, the display scrolls through the entire report. To pause between screens of text (similar to the `more` command in the operating system), set the `sh_enable_page_mode` variable to `true`.

To view a long report in this mode, press the space bar to view each successive screen. To cancel a long report and return to the `nt_shell` prompt, type the letter *q*.

You can interrupt a command in progress by typing the Ctrl+C key sequence. Computationally intensive commands might take some time to stop. Typing Ctrl+C multiple times terminates the shell and returns to the operating system prompt.

NanoTime Variables

Variables hold data. You can control many NanoTime options by specifying the value of application variables. You can also define user variables for convenience in scripts or at the command line. To specify the value of a variable, use the `set` command:

```
nt_shell> set variable_name value
```

You can use the `set_app_var` command instead of the `set` command when setting the value of an application variable. In this case, if NanoTime does not recognize the variable name, the tool issues a warning and defines a new user variable with the given name:

```
nt_shell> set_app_var abc value
Error: Variable ‘abc’ is not an application variable. Value will still
be set in Tcl. (CMD-104)
Information: Defining new variable ‘abc’. (CMD-041)
```

When you set an application variable, the displayed result is the new setting for the variable:

```
nt_shell> set sh_enable_page_mode true
true
```

If you attempt to set an application variable to an invalid value, NanoTime issues an error message. For example,

```
nt_shell> set sh_enable_page_mode maybe
Error: can't set "sh_enable_page_mode": invalid value:
       use true or false
Use error_info for more info. (CMD-013)
```

To determine the current setting for a variable, use the `printvar` command. For example,

```
nt_shell> printvar sh_enable_page_mode
sh_enable_page_mode  = "false"
```

You can use one or more wildcard characters (*) to view a group of variables. For example, to see a list of variables whose names include the string “clock,” enter

```
nt_shell> printvar *clock*
timing_all_clocks_propagated = "true"
timing_clock_gate_hold_fall_margin = "0"
...
```

To add values to an argument list for a variable, use the `append` command. For example,

```
nt_shell> set link_prefer_model { inv* }
inv*
nt_shell> append link_prefer_model { latch* }
 inv*  latch*
nt_shell> printvar link_prefer_model
link_prefer_model    = " inv*  latch* "
```

NanoTime Error, Warning, and Information Messages

NanoTime issues formal messages when a condition arises that requires user attention. Messages have three severity levels:

- Information: No action required if the condition is acceptable

  For example, the MXTR-006 message states that no parasitics were found on a specific net. This situation is not necessarily wrong and does not prevent timing analysis from proceeding. Check information messages to ensure that you understand their meanings.

- Warning: Serious condition found, likely to be undesirable, but does not stop execution

  For example, the PARS-103 warning message states that a capacitor read in from a parasitic netlist is ignored because the nets that connect to it cannot be found. Only you can determine whether this condition requires correction. Warning messages do not stop the analysis, but you should investigate and understand them.

- Error: Serious condition found that prevents analysis from continuing

  For example, the TECH-001 message states that transistor models cannot be found. Timing analysis is not possible.

Some NanoTime commands allow you to control the number and type of messages generated at a specific step in the flow. The `-message_level` option is available for the `match_topology`, `check_topology`, `check_design`, and `extract_model` commands. The allowed arguments for the `-message_level` option are as follows:

- `silent`: No messages except for warnings and errors
- `normal`: Brief messages (the default)
- `verbose`: Additional details that vary with the specific command used
- `debug`: Further details, beyond the `verbose` mode, possibly including messages that are only seen with this setting; varies with the specific command used

A few NanoTime commands provide a `-quiet` option to suppress all warning and error messages. This is common with the `get_*` commands (such as the `get_cells` or `get_nets` commands) because complicated filtering operations might return many unimportant messages as the filter operates on various objects.

Adding Context to Messages

You can optionally add information to the default error, warning, and information message output. The additional information might include net names, instance names, subcircuit names, and transition direction.

An example of a warning message without the additional information is as follows:

```
Warning: Simulator Warning: The ratio 29.4 of transition time 0.436
at pin X10.X00.MP0.g to the delay 0.015 to net pnode10 exceeds the
user-specified ratio 25.0. (DELC-104)
```

The following example shows the same message when the message context feature is activated. Additional information appears at the end of the message, separated by characters that you define (@&# in this example).

```
Warning: Simulator Warning: The ratio 29.4 of transition time 0.436
at pin X10.X00.MP0.g to the delay 0.015 to net pnode10 exceeds the
user specified ratio 25.0. (DELC-104) @&# [net=X10.X00.A,
inst=X10.X00, subckt=ppullup][net=pnode10]
```

Messages produced by third-party code libraries are not changed from their current form when displayed.

Activate the message context by specifying custom delimiter characters with the `sh_enable_message_context_with_delimiter` variable. Choose unique delimiter characters so that the reported message is not obscured due to similarities to other messages. The following command uses the characters “@&#” as a delimiter.

```
nt_shell> set sh_enable_message_context_with_delimiter @&#
```

The output message syntax when the message context function is activated is as follows:

original_message delimiter_string message_context_information

The syntax for added context information is as follows:

[net=*net_name*, inst=*inst_name*, subckt=*ckt_name*, edge = rising | falling]

When context messaging is activated, the following items of information are appended if they are available:

- The net name – the name of the most significant net that is related to the message
- The instance name – the full instance name of the most related subcircuit
- The subcircuit name – the subcircuit name of the instance when the transistor wrapper is not used as the name of the instance
- The transition direction – the path tracing edge direction of the net if the message is produced during path tracing

Messages that do not have additional information to report are displayed with the delimiter string and empty context as shown in the following example:

```
Without message context
Warning: Specified default supply voltage of 1.300V is outside the range
(1.200V, 1.200V) of supply voltages found in the design. (OPCD-008)

With message context
Warning: Specified default supply voltage of 1.300V is outside the range
(1.200V, 1.200V) of supply voltages found in the design. (OPCD-008) @&#
```

Some messages have more than one information object, as shown in the following warning message. The information objects are `XI1/MM0/G` and `XI2/MM0/G`.

```
Warning: Failed to get delay from pin XI1/MM0/G (net clk) rising to
XI2/MM0/G (net z) falling, unable to trace further. (DELC-003) @&#
[net=clk, inst=XI1, subckt=INV, edge=rising][net=z, inst=XI2, subckt=INV,
edge=falling]
```

In this case, NanoTime displays additional information for each of the two information objects. Square brackets enclose the context information for one object.

NanoTime Attributes

An attribute is a string or value associated with an object in the design that carries some information about that object. For example, the `number_of_pins` attribute attached to a cell indicates the number of pins in the cell. You can write Tcl scripts to get attribute information from the design database and generate custom reports about the design.

Attributes are read-only values that NanoTime assigns during execution. However, many attributes obtain their values from variables or command options that you specify.

To see a list of available attributes, use the `list_attributes` command with the `-application` option:

```
nt_shell> list_attributes -application
```

To generate a report of attributes and their values associated with specific objects in the design, use the `report_attribute` command with the `-application` option, as shown here for a list of nets:

```
nt_shell> report_attribute [get_nets {in}] -application
```

To get a specific attribute of a specific object, use the `get_attribute` command with arguments for the object and the attribute, as follows:

```
nt_shell> get_attribute Sn1 total_capacitance_fall_max
0.039744
```

To find and operate on targeted groups of attributes, use nested commands as shown in the following example. The following example finds the nets with a name prefix of Sn, gets the value of the `maxcap` attribute, and reports the net names and corresponding `maxcap` values.

```
foreach_in_collection snets [get_nets Sn*] {
  set maxcap [get_attribute $snets \
    total_capacitance_fall_max]
  set netname [get_attribute $snets full_name]
  echo "total cap. fall max of net $netname is $maxcap"
  }
```

See Also

- Chapter 19, “Object Attributes”

Executing Commands in Parallel or in the Background

You can improve performance by executing commands in parallel (with the `parallel_execute` command) or in the background (with the `redirect -bg` command) on your local host. These techniques do not require the use of additional NanoTime licenses beyond the licenses required for the parent run. They also do not require you to use or configure distributed processing.

The maximum number of parallel or background processes allowed by the NanoTime tool is 1024. However, your computing environment might place a smaller limit on the number of processes. Contact your system administrator for assistance.

You can use the following commands inside a `parallel_execute` or `redirect -bg` command. Most general Tcl commands are also allowed. You can specify custom Tcl procedures if they contain only allowed commands.

```
alias                    echo                proc_*
all_*                    exit                pwd
analyze_*                filter              query_*
cd                       foreach_*           redirect
*_collection             get_*               rename
*_collections            is_*                report_*
cputime                  list_*              save_*
create_lib_*             lminus              source
create_topology_*        ls                  suppress
current_design           mem                 which
current_instance         parse_*             write_*
define_*                 print
```

See Also

- The parallel_execute Command
- The redirect -bg Command
- Exiting With Pending Spawned Processes

The parallel_execute Command

The `parallel_execute` command specifies a list of actions to be performed in parallel.

Parallel command execution begins immediately. The `nt_shell` prompt reappears so that you can continue working while the commands run in the background. The largest runtime benefit is achieved by specifying the most time-consuming commands first in the list.

If you use the `-wait` option, the `nt_shell` prompt returns only after all parallel commands are complete. This is useful if the parent process needs the results of the spawned processes to continue.

The following rules apply to the `parallel_execute` command:

- The argument of the command is a list of actions to be performed in parallel, where each list item contains a *command string* and a *target file name*, as follows:

  ```
  parallel_execute {
     “command 1” target_1
     “command 2” target_2
     ...    }
  ```

  You can include more than one command in a command string by separating them with semicolons. In the following example, one process executes command 1, command 2, and command 3 sequentially, while another process is launched in parallel to execute command 4.

  ```
  parallel_execute {
     “command 1; command 2; command 3” target_1
     “command 4” target_file_2
     ...    }
  ```

- A command string consists of allowed commands, along with all of the options necessary to form a legal command and to achieve the goal. If the command string contains spaces, you must enclose the command string in quotation marks. For one-word commands, quotation marks are optional.

- If a command requires an argument in standard NanoTime usage, the argument must appear within the quotation marks that enclose the command string.

  In the following example, the `save_session` command requires a directory name as an argument. Therefore, inside a `parallel_execute` command, both the command and the directory name must appear within the quotation marks.

  ```
  "save_session newdir" file1
  ```

  By contrast, in the following example, the `report_paths` command does not require an output file name as an argument, because the default behavior is to write output to the display. However, the command requires that at least one of the `-min`, `-max`, or `-path_id` options (or a list of design objects) be included. Therefore the `-max` option appears inside the quotation marks but the output file name does not.

  ```
  "report_paths -max -rail_voltage" file2
  ```

- You cannot use the > or >> redirection operators in a command string. However, you can use these operators within a custom procedure that is referenced in a command string.

- You must specify a target file name for each command string in the `parallel_execute` command. The target file name must appear outside the quotation marks that enclose the command string. (You can optionally enclose the target file name in quotation marks, however, this is not necessary because the file name cannot include spaces.)

During parallel execution, the NanoTime tool might not write any output from the command to the target file, depending on the nature of the command. However, the tool always writes related formal messages (such as error messages) to the target file. The two cases are as follows:

- The command sends output to the target file.

  Commands such as reporting commands send information to the display in an interactive session. Therefore, an output file name is not part of the required command syntax inside the quotation marks. If you use such a command in a parallel process, the report information is sent to the target file instead of the display.

- The command does not send output to the target file.

  For some commands, the syntax requires a specification for the output. For example, the `save_session` command requires the name of a new directory as part of the syntax to make it a legal command. Therefore, the directory name appears in the command string inside quotation marks. The target file is used only for formal messages.

The `-compress` option of the `parallel_execute` command compresses target files with the gzip utility. This is primarily useful for the reporting commands. Output files that are specified as part of the command string (inside the quotation marks) are not compressed. Compressed files do not receive a *.gz extension automatically. You can either write an output file name that ends in .gz or rename the file after the processes are complete.

If any of the parallel processes fail to spawn successfully due to reaching the NanoTime or system limit, an information message appears in the parent NanoTime process. The specified command is then executed in the parent process.

For each successfully spawned process, the NanoTime tool generates a file whose name is the target file name appended with “.proc;" this file contains information about the process.

Examples of the parallel_execute Command

In the following example, the parallel actions are to save the session, write a set of SPICE decks, and generate two path reports:

```
parallel_execute -compress {
   “save_session newdir” ssfile
   “write_spice -output path_A.sp -simulate \
                [get_timing_paths -from clkA]” wsfile
   “report_paths -min -capacitance -nworst 10” min.rpt.gz
   “report_paths -max -capacitance -nworst 2” max.rpt.gz     }
```

The resulting output is as follows:

- A directory named newdir that contains the saved session

- Multiple SPICE deck files (not compressed), whose names are based on the name path_A.sp supplied in the `write_spice` command string
- HSPICE results files (not compressed), generated because the `-simulate` option of the `write_spice` command initiates simulation of the SPICE decks. (The HSPICE simulations are executed sequentially unless you enable distributed processing.)
- Compressed files min.rpt.gz and max.rpt.gz, which contain the specified path reports along with formal messages related to the `report_paths` commands
- Compressed files ssfile and wsfile, which contain formal messages (such as warning messages) related to the `save_session` and `write_spice` commands
- Process information files ssfile.proc, wsfile.proc, min.rpt.gz.proc, and max.rpt.gz.proc

In the following example, the first process creates a subdirectory named spicedir and changes focus to that directory before executing the `write_spice` command.

```
parallel_execute {
   “mkdir spicedir; cd spicedir; write_spice -output path_A.sp \
                [get_timing_paths -from clkA]” wsfile
   “report_paths -min -capacitance -nworst 10” min.rpt.gz
   ...    }
```

The redirect -bg Command

The `redirect` command generates an output file from one or more commands. The `-bg` option allows you to execute the commands in the background while you continue to work. Use the `redirect` command either inline or in the background, as follows:

- Inline

  In this mode, the `nt_shell` prompt does not return until the command processes are complete. The following example creates a file named paths2.rpt and writes the results of three reporting commands into the file:

  ```
  redirect paths2.rpt {
      report_paths -min -nworst 3
      report_paths -max -nworst 3
      report_noise }
  ```

- Background

  In this mode, the `nt_shell` prompt returns immediately so that you can continue working. The following example creates the same file with the same contents as the previous example, but the execution occurs in the background.

  ```
  redirect -bg -file paths2.rpt {
      report_paths -min -nworst 3
      report_paths -max -nworst 3
      report_noise }
  ```

When you run processes in the background, specifying a file name with the `-bg` option is required, but explicitly including the `-file` option is optional because it is the default. The `-file` option differentiates the output mode from the `-variable` and `-channel` options of the `redirect` command, which are not allowed with the `-bg` option.

You can nest `redirect` commands, as shown in the following example. However, for nested `redirect` commands, the `-bg` option is ignored. The nested commands are executed in the order that they appear.

```
redirect -bg -file max_summary.rpt  {
   report_paths -max -show_path_id
   redirect -bg -file max_detail.rpt {
      report_paths -max -path_type full_clock_expanded -nets -nosplit
   }
}
```

The following rules apply to the `redirect -bg` command:

- Do not use the > or >> redirection operators in the `redirect` command unless they are used inside a custom procedure.
- Do not use the same file name for parent and nested `redirect` commands or for multiple `redirect` commands nested under the same parent command.

Exiting With Pending Spawned Processes

By default, if you issue the `quit`, `exit`, or `suspend_licenses` commands in the parent NanoTime process while spawned processes are active, the tool waits until the spawned processes are complete before executing the command. In this case, the `nt_shell` prompt does not reappear and control returns to the operating system after all of the processes are complete.

You can change this behavior by setting the `pexe_wait_for_exit` variable to `false`. In this case, if you issue the `quit`, `exit`, or `suspend_licenses` commands in the parent NanoTime process, the `nt_shell` prompt returns immediately. The spawned processes continue to execute in the background.

If you terminate the parent process with a Ctrl+C key sequence or a `kill` command, or if the parent process terminates abnormally, associated spawned processes that are currently running are automatically terminated.

Distributed Processing

NanoTime tasks such as extraction recalibration, composite current source noise modeling, and memory column simulation use the HSPICE simulator for analysis. Using distributed processing for HSPICE simulations can greatly reduce the elapsed time for complex analyses. NanoTime distributes simulation tasks to multiple hosts, using Perl scripts to control the load balance between distributed jobs. You can choose the set of hosts, the maximum number of HSPICE licenses to use, and the working directory for Perl scripts.

Using distributed processing, you can also execute complete NanoTime instances in parallel by using the `run_parallel` command.

Distributed processing is available via the Common Distributed Processing Library (CDPL), which is included with every NanoTime installation. Several utility programs are available to help you set up and monitor distributed processes. For more information about CDPL and the utility programs, see the user documents located in the *NanoTime_install_root_directory/cdpl/doc* directory. The documents are the *CDPL Users Manual*, the *DP Manager User Guide* and *DP Manager Frequently Asked Questions*.

To enable and configure distributed HSPICE processing, set the following variables:

- Use the `distributed_processing_host_file` variable to specify the host file.
- Use the `distributed_processing_number_hspice_license` variable to specify the maximum number of HSPICE licenses to check out.
- Use the `distributed_processing_perl_path` variable to specify a path to use for Perl scripts. Perl versions 4.0.1.8 and later are supported.
- Keep the `nt_tmp_dir` variable set to "." (the default) to ensure that all the necessary simulation files are globally accessible.
- Set the `CDPL_FARM_DISABLE_SHELL_INTERPRETER` environment variable if your computing environment does not accept scripts that begin with the #! characters. Set the variable to any value; the important distinction is whether the variable is set or not.

See Also

- Host Files for Distributed Processing
- Testing the Computing Environment
- Parallel NanoTime Runs

Host Files for Distributed Processing

A host file defines the distributed processing environment. In the host file, you specify information about the machines in the compute farm and the way jobs are to be launched. The same NanoTime variable specifies the host file for all HSPICE runs, so you might need to reset the variable within a Tcl script. You must set up the computing environment before starting a NanoTime session.

The following protocols are supported:

- RSH (remote shell)
- SSH (secure shell)
- SGE (Oracle Grid Engine, formerly Sun Grid Engine)
- LSF (Load Sharing Facility from Platform Computing)
- SH (shell process on the local host)
- PBS (Portable Batch System)
- RTDA (Runtime Design Automation Network)

The host file is an ASCII file in which each line follows the same format and provides information about one entity. Comment lines begin with a pound sign (#).

Each line in the host file must observe the following rules:

- The format for each line is: *flag* | *hostname* | *num_slots* | *tmpDir* | *mode* | *command*
- Each field is terminated by a pipe symbol (“|”) except the last field.
- A space must be present for an empty field.
- Each line must be a single unwrapped line.
- The fields contain information as follows:
  - *flag*: 1 to enable; 0 to disable
  - *hostname*: Machine name; empty for some environments
  - *num_slots*: number of worker slots on this host or farm; -1 indicates unlimited
  - *tmpDir*: (optional) temporary work directory
  - *mode*: RSH, SSH, LSF, SGE, SH, PBS, or RTDA
  - *command and options*: string containing the command to connect to worker machines (slave processes), written on a single unwrapped line in the host file

The string in this field can also be the file name (optionally including a path) of a script file that contains the submission command. The contents of the file must be a single unwrapped line of text.

Examples:

- RSH infrastructure with two host machines and allowing a total of 10 worker slots (4 on one machine and 6 on the other)

```
# host file for RSH
1|engr_lab-x9|4|/remote/users/tmp |RSH| rsh
1|engr_lab-x2|6|/remote/users/tmp |RSH| rsh
```

- SSH infrastructure with 2 machines and allowing a total of 12 worker slots

  This environment requires login without passwords.

```
# host file for SSH
1|engr_lab-x15|4|/remote/users/tmp |SSH| ssh
1|engr_lab-x21|8|/remote/users/tmp |SSH| ssh
```

- SGE compute farm

  The hostname field is empty for SGE farms.

```
# host file for SGE
1| |-1| |SGE| qsub -P bnormal -l mem_free=1G
```

- SGE compute farm in which the number of launched jobs is limited to 3 and a directory for temporary files is specified

```
# host file for SGE
1| |3| /remote/users/tmp |SGE| qsub -P bnormal arch=glinux
```

- LSF compute farm allowing an unlimited number of worker slots

  The hostname field is empty for LSF farms.

```
# host file for LSF
1| |-1| |LSF| bsub -R rusage[mem=8000] -R arch=glinux
```

- LSF compute farm allowing an unlimited number of worker slots and using a script file to provide the submit command

  The file named my_bsub contains the following line:

```
/lsf/bin/bsub -R rusage[mem=8000] -R arch=glinux
```

  The host file contains the following:

```
# host file for LSF
1| |-1| |LSF| ./my_bsub
```

- The SH protocol can be used to distribute processing over multiple cores on the same machine when RSH and SSH are not available. The hostname field is not required, but in this example, localhost is entered as a reminder:

  ```
  # host file for SH
  1|localhost |8|/remote/users/tmp |SH| sh
  ```

- PBS infrastructure with one machine and four worker slots

  The hostname field is empty for PBS environments.

  ```
  # host file for PBS
  1| |4|/remote/users/tmp |PBS| qsub -q bnormal -l arch=glinux
  ```

- RTDA infrastructure with one machine and eight worker slots

  The hostname field is empty for RTDA environments.

  ```
  # host file for RTDA
  1| |8|/remote/users/tmp |RTDA| nc run
  ```

Testing the Computing Environment

Before you run a NanoTime script that uses parallel analysis, you must verify that the host file for distributed processing is correct for your computing environment.

Every installation is different, but the general steps are as follows:

1. Set the `CDPL_HOME` environment variable to the cdpl subdirectory of the NanoTime installation root directory. For example,

   ```
   % setenv CDPL_HOME CDPL_home_dir
   ```

   where *`CDPL_home_dir`* is *`NanoTime_install_root_directory/cdpl`*.

2. Update the path variable to include the CDPL utilities, as follows:

   ```
   % set path = ($CDPL_HOME/bin $path)
   ```

3. Create a script file to configure your compute farm environment or obtain a script from your Information Technology support group. The objective is to configure a submit host that submits a distributed processing run to a compute farm.

   Determine the name of the compute farm and check that the machine you are using is a submit host for the farm. Use the `dpvalidate` command and the `dpcheck -m` command with the appropriate argument (`lsf`, `sge`, `pbs`, or `rtda`) to validate the infrastructure. For SSH or RSH protocols, all Linux hosts are submit hosts.

4. Verify that the selected queue has appropriate capabilities (such as memory and duration) for the planned NanoTime runs.

5. Test the farm setup with a simple command such as the `ls` command or the `date` command. For example, if you are using an LSF farm,

   ```
   % bsub -R "rusage[mem=1000]" date
   ```

6. Create and test the host file. For example,

   ```
   % dpcheck -host your_host_file
   ```

7. Use the host file name in a NanoTime script or command line.
8. (Optional) To monitor and control a distributed processing run, use the DP Analyzer feature in the DP Manager utility.
9. (Optional) To monitor and control a distributed processing run, invoke NanoTime with the DP Manager utility by inserting the `dpmanager_nt` command before the `nt_shell` command:

   ```
   % dpmanager_nt nt_shell -file script1.tcl
   ```

   Use options after the `nt_shell` command in the same way that you would without DP Manager. DP Manager allows you to monitor job and task status information, create and edit host files, and stop jobs.

For more information about DP Manager, see the *DP Manager User Guide* and *DP Manager Frequently Asked Questions* documents, located in the *NanoTime_install_root_ directory/cdpl/doc* directory. For more information about CDPL and other utility programs, see the *CDPL Users Guide* located in the same directory.

For assistance with your specific computing environment, contact your Information Technology support group.

Parallel NanoTime Runs

You can execute NanoTime runs in parallel to break up large tasks into smaller tasks, with the following benefits:

- Save turnaround time by using distributed processing to execute tasks in parallel
- Customize the parallel tasks by using case analysis and Tcl scripts

Figure 1-14 illustrates how a NanoTime flow (or script) might branch into three separate tasks. After all of the parallel operations are complete, the script continues. The script controls how the results from the parallel operations are used. For example, you can merge separately created models or create a custom report from multiple analysis runs.

Figure 1-14 Parallel Runs in a NanoTime Script

The `run_parallel` command provides the infrastructure to

- Launch multiple NanoTime instances
- Work with all NanoTime commands and variables
- Support all distributed processing environments
- Support job monitoring and maintain existing license quotas

Use the `run_parallel` command to initiate parallel operation from the `nt_shell` command line or from a script. The `run_parallel` command provides two methods to initiate the parallel runs: by instance names or by script files.

- Instance names (using the `-instance_names` option with a list of names)

  All NanoTime instances are launched from the same script. The script changes the analysis conditions for the different NanoTime instances by using the instance name in the Tcl logic. For example, you could use an `if` statement to set a multiplexer select pin to either 0 or 1 depending on the instance name. The script then executes the two NanoTime timing analysis runs in parallel.

- Script files (using the `-files` option with a list of script files)

  Each NanoTime instance is launched using a different script. In this case, the processing logic or task to be completed can be completely different for each NanoTime instance.

The `run_parallel` command returns a 1 when it succeeds and 0 otherwise. For the command to be a success, all of its parallel instances must complete successfully. The script execution continues beyond the `run_parallel` command only after all the parallel instances are finished. There is no interaction between parallel instances.

Using Instance Names to Specify Parallel Runs

The `-instance_names` option invokes each instance of the NanoTime tool with arguments identical to those of the executing instance, including the same Tcl control file, but using a specific value for the instance name taken from the argument list. All processing logic resides in the same script. The script uses Tcl commands and the instance names to change analysis conditions for each NanoTime instance.

NanoTime launches new instances from the same directory as the executing instance, unless the `-directories` option is specified. NanoTime saves screen output in files named *instance_name.*log. Instance names must be unique and cannot contain special characters. Files from repeated runs overwrite the existing files.

The `-parallel_instance_count` option limits the number of concurrent NanoTime instances launched by the `run_parallel` command. The default is infinity, but because the number of licenses is finite, you should specify this argument to avoid deadlocks. The minimum value is 1. Set the `-parallel_instance_count` option to a value less than or equal to the total number of available NanoTime licenses. However, if each NanoTime analysis requires more than one license, the value should be less than or equal to the total number of available licenses divided by the number of licenses required for each run.

Example of Parallel Runs Using Instance Names

Consider a 2-input multiplexer design named mux_1 with select signals sel0 and sel1. The goal is to characterize the design for each select mode, then merge the resulting models into a single model.

The following command, invoked in the top-level directory, starts the parallel analysis run:

```
nt_shell –x “set distributed_processing_host_file small_hosts; \
run_parallel –files {mux.tcl}” | tee model.log
```

The log outputs for each select mode are placed in the top-level directory; they are named sel0.log and sel1.log. The log from the master instance appears on the screen and also is directed to a file named model.log in the top-level directory.

The script mux.tcl is as follows:

```
if {$distributed_processing_is_master}{
  set distributed_processing_host_file big_hosts
  set success [run_parallel -instance_names {sel0 sel1}]
  if {$success}{
    merge_models -db_files {./timing_model/sel0_lib.db  \
                            ./timing_model/sel1_lib.db) \
         -mode_names {sel0 sel1} -output ./timing_model/mux_merge
  }
  exit
}
set mode $distributed_processing_instance_name

set search_path {. ./designs}
set link_path {*}
read_netlist -format spice {tech.sp mux.sp}
link_design mux_1

#setup commands
...
if {$mode eq “sel1”}{
  set_case_analysis 1 [get_pins -of [get_nets sel] ]
} elseif {$mode eq “sel0”}{
  set_case_analysis 0 [get_pins -of [get_nets sel] ]
}
check_topology
check_design

extract_model -name timing_model/$mode
```

Using File Names to Specify Parallel Runs

In this mode, each NanoTime instance is launched using a different script. NanoTime launches new instances from the directory containing the script file unless the `-directories` option is specified. You must configure the `search_path` and `library_path` variables in the Tcl script so that a NanoTime instance running in the same directory as the Tcl script can access all the files referenced in the script. If the script file name has the format *file_prefix.*tcl, NanoTime saves the screen output in files named *file_prefix.*log; otherwise, the extension .log is appended to the script file name (for example, *file_prefix.*nt generates *file_prefix.nt.*log).

Full-path script names must be unique and cannot contain special characters. Files from repeated runs overwrite files from earlier runs.

You can use the `-files` option to run scripts that themselves contain the `run_parallel` command. A complex hierarchy of instances that launch other instances could result. In general, the sum of all of the values of the `-parallel_instance_count` arguments in

scripts that launch NanoTime runs should be less than or equal to the total number of available NanoTime licenses. Reduce the values of the `-parallel_instance_count` option in cases where each run requires more than one NanoTime license.

Example of Parallel Runs Using File Names

This example analyzes a four-input multiplexer circuit, where each select mode has its own Tcl script, then merges the models.

```
set search_path “.”
set result 0
set distributed_processing_host_file lsf.hosts

if($distributed_processing_is_master){
  set files [list]
  if {![file exists sel0_lib.db]} {
    lappend files “sel0.tcl”
  }
  if {![file exists sel1_lib.db]} {
    lappend files “sel1.tcl”
  }
  if {![file exists sel2_lib.db]} {
    lappend files “sel2.tcl”
  }
  if {![file exists sel3_lib.db]} {
    lappend files “sel3.tcl”
  }
  if {[llength $files]>0} {
    set result [run_parallel -files $files -exclude_master]
  }

  if {$result} {
    set result [merge_models -db_files {\
        sel0_lib.db \
        sel1_lib.db \
        sel2_lib.db \
        sel3_lib.db \
        } \
    -mode_names {sel0 sel1 sel2 sel3} \
    -output 4MUX]
    }
  }
```

Managing Licenses and Computing Resources

When you use the `run_parallel` command, NanoTime licenses are consumed as needed to execute commands. If there are not enough licenses available, instances execute sequentially as licenses become available. You should configure NanoTime licensing to wait for a license to become available instead of exiting when a license is not available. There is no license penalty for the submit host computer. If the submit host is not capable of running NanoTime, the host relinquishes its NanoTime license while waiting for the runs to finish. If the submit host computer can run NanoTime, the host uses a license while it executes one of the instances.

The `-parallel_instance_timeout` option limits the runtime to handle instances that are stalled due to system, design, or user errors. The default is infinity and the minimum value is 30 seconds. In the event of an error, you can rerun the failed instances by checking for the expected output files when you construct the `run_parallel` command.

The `-exclude_master` option prevents the invoking instance from performing any of the parallel tasks. Use this option when the invoking instance is running on a host computer that is not suitable for performing a parallel NanoTime task.

Two read-only Tcl variables are available to create differentiated sections within a script.

- The `distributed_processing_instance_name` variable contains the list of instance names for the `-instance_names` option. You might use this variable to create different logic for memory read and write operation scripts, for example.
- The `distributed_processing_result_list` variable contains a list of the script exit status values from each of the instances. You might use this variable to define logic to handle cases where some of the NanoTime runs result in errors.

Other variables provide information about computing resources.

- The `distributed_processing_is_master` variable is `true` for NanoTime runs that do not have an instance name.
- The `distributed_processing_host_file` variable is used to set information about the computing resources available to the job scheduling system. If no host file is specified, the local host is used. NanoTime worker instances must run on the same architecture as NanoTime master instances.

You can control the number of debugging messages issued during the run by setting the `NT_RUN_PARALLEL_DEBUG_LEVEL` environment variable before beginning the NanoTime run. Valid values are `low`, `medium`, and `high`; the default is `low`.

See Also

- Host Files for Distributed Processing

2

Netlist Data and Analysis Setup

The first step in timing analysis is to load the design data, including the netlist, technology files, library-based timing models, and parasitic data.

This chapter includes the following sections:

- Design Data in the NanoTime Analysis Flow
- Path Variables
- Netlist Registration
- Design Linking
- Linking Timing Models
- Defining the Voltage Environment
- Design Reporting
- Parasitic Data Overview
- Working With Parasitics Files
- Variables for Parasitic Data Analysis

Design Data in the NanoTime Analysis Flow

To perform timing analysis, you must provide the design and technology data and set variables to define analysis options. Some operations must take place during the netlist phase (the part of the NanoTime flow before the `link_design` command), but other data can be provided any time before the `check_design` command, because it does not affect topology recognition. Figure 2-1 shows the steps necessary to define the design.

The topics in this chapter include design linking, voltage setting, and parasitics. Technology data is discussed in Chapter 3, "Technology Data," and the discussion of the topology recognition phase begins with Chapter 4, "Topology Operations."

Figure 2-1 Design Data in the NanoTime Analysis Flow

Path Variables

The `search_path` and `link_path` variables specify where to obtain the data files for reading and linking the design.

The search_path Variable

The `search_path` variable specifies a list of directories that NanoTime searches to find design files, in the order that the directories are listed.

By default, the `source` command, which is used to execute scripts, does not use the `search_path` setting. To search for scripts in the directories specified by the `search_path` variable, set the `sh_source_uses_search_path` variable to `true`.

Set the `search_path` variable with the `set` command. The argument is a list of paths to the directories containing the files needed for analysis. For interactive use, include the period character to search the directory from which a command is issued. Delimit each path with a space and enclose the list in double quotation marks. NanoTime responds to the command by echoing the search path definitions. For example,

```
nt_shell> set search_path ". /u/project/design /u/project/library"
. /u/project/design /u/project/library
```

The link_path Variable

The `link_path` variable specifies a list of libraries, design files, and library files used during linking. The `link_design` command looks at those files and tries to resolve hierarchical references in the order that the files are listed in the variable.

The `link_path` variable can contain three different types of elements: an asterisk character (`*`), a library name, or a file name. By default, the variable is set to the asterisk character. You can leave the variable set to an asterisk or you can set it to an asterisk followed by one or more library names. The asterisk causes the `link_design` command to search all the designs loaded into NanoTime when trying to resolve references.

For elements other than an asterisk, NanoTime searches for a library that has already been loaded. If that search fails, NanoTime searches for a file with the specified name in the directories specified by the `search_path` variable.

Set the `link_path` variable with the `set` command. The argument value should be an asterisk followed by a list of cell library names. Delimit each element with a space and enclose the list in double quotation marks. For example,

```
nt_shell> set link_path "* STDLIB.db"
* STDLIB.db
```

In this example, when the `link_design` command is executed, NanoTime first looks for cells in loaded libraries and then looks in the library called STDLIB.db in the directories specified by the `search_path` variable.

Netlist Registration

Each analysis requires a `register_netlist` command that contains, at a minimum, the top-level design netlist. You can optionally include more than one file in this command.

If you set the `link_path` and `search_path` variables, you only need to register the top design. At design linking, NanoTime automatically finds the other netlist files and libraries if they are located in the specified paths.

When you execute the `register_netlist` command, NanoTime verifies the existence of all referenced files and records the file names in the registry. However, the tool does not read the files or check their contents. The netlist files are not used until the `link_design` command.

The `-format` option specifies the format of the netlist file to be registered, such as `spice` or `verilog`. If you do not use the `-format` option, the command assumes formats based on the file name extension: .sp for SPICE and .v for Verilog. You can register multiple files in different formats and use wildcard characters to register multiple files.

A netlist can be a plain ASCII text file or an ASCII file that has been compressed with the gzip utility. Compressed files must have .gz as the file name extension.

If you specify a file with a relative path, NanoTime searches for the file in the directories specified by the `search_path` variable.

After the files have been registered, use the `list_netlists` command to view the netlist registry. To remove netlist files from the registry, use the `remove_netlist` command.

Design Linking

After the netlist files have been registered, use the `link_design` command to build a representation of the design for analysis. The design linking operation performs a name-based resolution of hierarchical references for the design.

A number of variables affect design linking and therefore must be set before you execute the `link_design` command. The following sections describe many of these variables. For more information about specific variables, see the man pages.

This section contains the following topics:

- The link_design Command
- Current Design and Current Instance
- Design Units
- Pin and Transistor Naming Conventions
- Defining Ports for SPICE Netlists
- Special Device Handling
- Linking Encrypted Device Models
- Retaining Boundary Pins for Discarded SPICE Subcircuits
- Linking Designs With Embedded Parasitics
- Linking Designs With Models That Contain Parasitics

The link_design Command

To successfully run the `link_design` command, you must have a complete, fully functional design connected to all referenced library components and designs. NanoTime finds and links the references to the specified design or current design.

The following example sets the `search_path` and `link_path` variables, registers three Verilog netlist files, and links the top-level design:

```
nt_shell> set search_path "/designs/newcpu/v1.6/netlists /libs/cmos"
/designs/newcpu/v1.6/netlists /libs/cmos
nt_shell> set link_path "* fast_cmos_lib"
* fast_cmos_lib
nt_shell> register_netlist -format verilog { BOX1.v BOX2.v padring.v }
1
nt_shell> link_design newcpu
...
Linking design newcpu...
Design 'newcpu' was successfully linked.
```

By default, the case sensitivity of the link operation is determined by the source of the objects being linked. You can change this behavior by setting the `link_case` variable. The variable default is an empty string, which results in case-sensitive linking without case shifting. Set the variable to `upper` or `lower` to shift all names to uppercase or lowercase. All subsequent references to node and pin names must be made in the specified case.

Linking failures are typically caused by missing libraries or designs, incorrectly specified `link_path` or `search_path` variables, or files that do not have read permission.

In some cases, NanoTime issues many information or warning messages during linking. To limit the number of times a specific message is reported in a session, use the `link_control_options` variable, as in the following example:

```
nt_shell> append link_control_options {set_message_limit:"0x2020107f 10"}
```

Current Design and Current Instance

The `current_design` command sets or reports the current design. If you specify a design as an argument for the command, that design becomes the current design. If you do not specify a design, the tool reports the name of the current design. To display the designs in memory, use the `list_designs` command.

The `current_design` command changes the working design, then sets the current instance to the top level of the new current design. The combination of the current design and current instance defines the context for NanoTime commands.

The `current_instance` command sets the working instance to an instance (cell) in the current level of the design hierarchy. You can view the cells within the current instance by using the `report_cell` command. Setting the current instance enables other commands to be used relative to the specified instance.

The `current_instance` command operates with a variety of arguments:

- If no argument is used, the focus returns to the top level of the current design.
- If the specified instance name is the name of a hierarchical cell at the current level of hierarchy, the current instance is moved down to that level of the design hierarchy.
- If the specified instance name is “ . ”, the current instance is displayed.
- If the specified instance name is “ .. ”, the current instance is moved up one level in the design hierarchy.
- Multiple levels of hierarchy can be traversed by separating multiple cell names with the hierarchy separator character. For example, `current_instance U1.U2` sets the current instance down two levels of hierarchy.

By default, the hierarchy separator is the period. If you want to use a single period (.) or double period (..) to represent the current or higher-level instance, you must set the hierarchy separator to a different character. For example, the following command sets the hierarchy separator to the forward slash:

```
nt_shell> set hierarchy_separator /
```

You can use the `current_instance` command to move up and down the design hierarchy:

```
nt_shell> current_design TOP
{"TOP"}
nt_shell> current_instance U1
U1
nt_shell> set hierarchy_separator /
/
nt_shell> current_instance "."
U1
nt_shell> current_instance U3
U1/U3
nt_shell> current_instance "../U4"
U1/U4
nt_shell> current_instance
Current instance is the top-level of design 'TOP'.
```

Design Units

A set of global variables defines the design units. NanoTime writes the design unit specifications into the built-in library (the in-memory representation of the design) when you execute the `link_design` command. When the tool reads in or writes technology data, data is scaled to match the specified unit sizes. These variables must be set before you execute the `link_design` command.

After the design has been linked, you can report the design unit sizes with the `report_design` command, as in the following example.

```
nt_shell> report_design
 ...
  Design Attribute                Value
- ------------------------------- -------------
Units:
  Time Unit                       1 ns
  Capacitance Unit                1 pf
  Voltage Unit                    1 V
  Resistance Unit                 1 kohm
  Current Unit                    1 mA
  Width/Length Unit               1 um
  Area Unit                       1 um square
```

The units shown in this example are the defaults. The Width/Length and Area units come from the technology data and cannot be set manually. Table 2-1 lists the measurement unit variables and their valid values.

Table 2-1 Global Variables for Measurement Units

Property	Variable	Valid values
Time	`lib_time_unit`	`1ps`, `10ps`, `100ps`, `1ns`
Capacitance	`lib_capacitance_unit`	`1ff`, `10ff`, `100ff`, `1pf`
Voltage	`lib_voltage_unit`	`1mV`, `10mV`, `100mV`, `1V`
Resistance	`lib_resistance_unit`	`1ohm`, `10ohm`, `100ohm`, `1kohm`
Current	`lib_current_unit`	`1uA`, `10uA`, `100uA`, `1mA`, `10mA`, `100mA`, `1A`

Using a large design unit might cause small values to lose precision. For example, a current value of 0.23759 mA would become 237.59 uA if the current unit specified in the `lib_current_unit` variable is set to `1uA`. In this case, no information is lost. However, if the specified current unit is set to `1A`, the value becomes 0.000238 A, losing some digits of precision.

This issue is especially important if you are creating composite current source (CCS) models. You must verify that the `lib_current_unit` variable does not have an adverse effect on the accuracy of current values stored in the model. For best results, use a value of `1uA` for the `lib_current_unit` variable when you create CCS models.

Table 2-2 shows the commands that support user-defined units. Other commands relating to time, capacitance, voltage, resistance, and current support only the default units.

Table 2-2 Command and Argument Pairs That Support User-Defined Units

Command	Argument
`set_delay_coefficients`	-offset *offset_value*
`set_transition_coefficients`	-offset *offset_value*
`set_driving_cell`	-input_transition_rise *rise_slew* -input_transition_fall *fall_slew*
`set_correlated_input`	-skew *skew_value* -slope *slope_value*
`set_max_delay`	*delay_value*
`set_min_delay`	*delay_value*

Pin and Transistor Naming Conventions

Table 2-3 lists variables that define device pin names and transistor names for linking.

Table 2-3 Variables For Device Terminal Names

Device Type	Variable	Default
capacitor	`link_capacitor_term1_pin_name`	term1
	`link_capacitor_term2_pin_name`	term2
	`link_grounded_capacitor_term1_pin_name`	term1
resistor	`link_resistor_term1_pin_name`	term1
	`link_resistor_term2_pin_name`	term2
diode	`link_diode_base_pin_name`	ANODE
	`link_diode_emitter_pin_name`	CATHODE
transistor	`link_transistor_bulk_pin_name`	b
	`link_transistor_drain_pin_name`	d
	`link_transistor_gate_pin_name`	g
	`link_transistor_source_pin_name`	s

These variables accept a list of names as an argument. A parasitics file can refer to any of the specified names. The first name in a list is used for the logical netlist. In other words, the first name is used with commands and reports that refer to pin objects, such as the `get_pins` and `report_cell` commands.

Similarly, the `link_nmos_alias` and `link_pmos_alias` variables specify allowable transistor names. The defaults are shown in Table 2-4.

Table 2-4 Variables For Transistor Names

Device Type	Variable	Default
transistor	`link_nmos_alias`	*nfet* n* *nch*
	`link_pmos_alias`	*pfet* p* *pch*

For example, NanoTime recognizes a transistor as an NMOS transistor if its cell name starts with the letter n or contains nfet or nch. If your netlist uses any other naming convention to identify NMOS transistors, specify that convention in the variable. For example,

```
nt_shell> set link_nmos_alias "nch N n NCH NMOS"
```

Defining Ports for SPICE Netlists

When the design is a lower-level SPICE netlist that has no top-level subcircuit (in other words, a flat netlist), no input or output ports are defined for the design. In that case, you must specify ports with the `create_port` command. The command specifies the port type (input, output, or bidirectional) and a list of nets that are ports of that type.

For example, the following command specifies that nets D0 through D3 are bidirectional ports. The names assigned to the ports are the same as the net names.

```
nt_shell> create_port -inout [get_nets {D0 D1 D2 D3}]
```

The `create_port` command is supported only at the top level of a design. NanoTime issues a warning message if the command is used for nets in any other location, as follows:

```
Warning: Net Xareg.hld11 is not a top-level design net. (CMDS-108)
```

In this case, analysis continues. However, using the `create_port` command on an internal net is not a supported usage and accuracy is adversely affected.

Special Device Handling

Some types of devices require additional consideration during design linking.

Behavioral Resistors

By default, behavioral resistors (voltage-controlled resistors) in the design are converted into constant value parasitic resistors during design linking. You can ignore all behavioral resistors by setting the `rc_enable_behavioral_resistor_conversion` variable to `false`.

You can specify a global voltage to use for converting behavioral resistors into constant value resistors with the `rc_behavioral_resistor_controlling_voltage` variable. The default of this variable is -1.0, in which case the controlling voltage is set to the value of the `oc_global_voltage` variable. If the `oc_global_voltage` variable is not set, the default rail voltage is used.

Behavioral resistors in parasitic netlists are not supported.

Linking Encrypted Device Models

Device model encryption protects the intellectual property in advanced device models. NanoTime can read device model files that are encrypted using the `metaencrypt` utility, which is part of the HSPICE tool suite. All encryption types generated by the `metaencrypt` utility are supported. Encryption by any other utility or tool is not supported.

To enable the use of encrypted models, set the `link_enable_metaencrypted_models` variable to `true` (the default is `false`) before running the `link_design` command.

A NanoTime Ultra license is required to read encrypted device models. Reading encrypted netlists is not supported. NanoTime reporting commands, such as the `report_technology` and `write_spice` commands, do not expose details of encrypted models.

See Also

- Encrypted Device Models

Retaining Boundary Pins for Discarded SPICE Subcircuits

When the `read_parasitics` command is executed, elements in the parasitic netlist might be associated with pins that do not exist in the design netlist, resulting in error messages about unresolved pins. Unresolved pins can occur inside timing models or inside discarded SPICE subcircuits.

In both cases, NanoTime issues error messages at the `read_parasitics` command, discards the RC elements connected to the unresolved pins, and continues execution. Discarding these parasitics might lead to inaccuracy in delay calculations.

To address the unresolved pin errors for a discarded SPICE subcircuit, you must discard the subcircuits properly. The `link_discard_subckt_contents` variable retains the subcircuit boundary pins and connecting nets, making it possible to retain the associated parasitics. Set this variable before executing the `link_design` command. An example is as follows:

```
set link_discard_subckt_contents “abc xyz”
```

By contrast, if you use the `link_control_options` variable to discard subcircuits, the boundary pins of the subcircuit and the nets connecting those boundary pins to the rest of the design are completely discarded, which increases the likelihood of causing unresolved pin error messages during parasitic back-annotation. The following is an example:

```
append link_control_options { discard_subckt_inst: “abc xyz” }
```

See Also

- Handling Parasitics For Subcircuits

Linking Designs With Embedded Parasitics

When the design netlist contains parasitics, you must use one of the following options of the `link_design` command to specify the handling of coupling capacitors:

- If you are performing signal integrity analysis, use the `-keep_capacitive_coupling` option to preserve the parasitic elements.
- Otherwise, use the `-coupling_reduction_factor` *`C_factor`* option to split coupling capacitance between nets into two capacitors to ground, one for each of the coupled nets. The coupling reduction factor *C_factor* determines the value of the new capacitors. Values between 1.0 and 2.0 are most common, depending on the process.

If the netlist contains embedded parasitics, you must resolve any incomplete nets by using the `complete_net_parasitics` command.

See Also

- Completing Partially Annotated Nets

Linking Designs With Models That Contain Parasitics

By default, NanoTime preserves parasitics from macro models that contain parasitics. However, for design flows with corner-specific wrapper subcircuits, the wrapper subcircuit parasitics must be ignored during design linking. The general procedure is as follows:

1. Set the `link_enable_wrapper_subckt_parasitics` variable to `false`.
2. (Optional) Use the `sim_transistor_wrapper_subckts` variable to identify wrapper subcircuits.
3. Execute the `link_design` command.
4. Set up the single-corner flow with the `read_spice_model` and `set_technology` commands.
5. Read parasitics with the `read_parasitics` command without the `-complete_with` option.
6. Execute the `check_design` command with the `-complete_with zero` option.

Linking Timing Models

Timing models (also known as library models) represent the timing behavior of circuit blocks. Using a timing model to represent all or part of a design reduces runtime and protects intellectual property.

NanoTime can read in and use any timing model that the tool creates, with the exception of CCS noise models.

However, models generated by other tools might use features that NanoTime does not support. For example, NanoTime supports two-dimensional lookup tables for timing arc data, but not three-dimensional tables. The tool reads in three-dimensional lookup tables without error, but ignores the data values of the third index. You are responsible for determining if models generated by other tools can be used successfully as input for NanoTime analysis.

Only the signal pins of an extracted timing model can be connected to nets in a netlist. Power and ground pins in the model cannot be connected to power or ground rails in the netlist. The timing information in the model applies only to the specific power supplies used in the design when it was characterized.

See Also

- Effects of Slew Thresholds on Model Use
- Effects of Timing Models on Delay Analysis

Reading Library Models

The `read_library` command and the equivalent `read_db` command are used to read timing model information from Synopsys database library files (.db files), such as those created by the `extract_model` command. A .db file can contain netlist information, but NanoTime does not use this information. To read in netlists, use the `register_netlist` and `link_design` commands.

You can also read library files in Liberty format (.lib files) by using the `read_lib` command. To use the `read_lib` command, you must install the Synopsys Library Compiler tool and link it to the NanoTime tool. For more information, see the *NanoTime Product Installation Notes*.

Each file can contain one library object. To report the library objects after the files are loaded, use the `list_libs` command.

In the following example, the file mycell.db is read in from the /libs/cmos directory, which is included in the `search_path` variable.

```
nt_shell> set search_path "/libs/cmos"
/libs/cmos
nt_shell> set link_path " * mycell_lib.db $link_path"
* mycell_lib.db $link_path
nt_shell> read_library mycell.db
Loading db file '/libs/cmos/mycell.db'
```

Using Models Preferentially

You can specify subcircuit names for which to use timing models instead of netlist definitions by using the `link_prefer_*` variables. The variables must be set before you use the `link_design` command. The models must be in .db form, be in the search path, and must have been read into memory with the `read_library` or `read_db` command.

Two variables are available to specify preferred models, listed in Table 2-5.

Table 2-5 Variables To Specify Preferred Models

Variable	Name type	Port mapping
`link_prefer_model`	subcircuit	position-based
`link_prefer_model_port`	subcircuit	name-based

The variables must be set to a white-space delimited list of names. Wildcard characters are allowed. For example, the following command causes all available timing models to be used in place of subcircuit definitions:

```
nt_shell> set link_prefer_model { * }
```

The substitution rules are as follows:

- Subcircuit names

  Substitution based on subcircuit names means that for each block matching a listed name, NanoTime uses the identically named .db model in place of the netlist. In the following example, NanoTime uses the regblk1.db timing model rather than the regblk1 netlist for each block named regblk1 in the design:

  ```
  nt_shell> set link_prefer_model {regblk1}
  ```

- Position-based port mapping

  NanoTime assumes that the ports are defined in the netlist in the same order that the ports are defined in the timing model, regardless of naming conventions.

- Name-based port mapping

  NanoTime matches the port names in the netlist to the port names in the timing model, regardless of the order in which the ports are defined. If there is a conflict between position-based and name-based variable settings, the tool uses position-based port mapping.

Setting Maximum and Minimum Libraries

You can create a relationship (called a max-min relationship) between two different .db libraries so that NanoTime uses timing models from one library for maximum-delay analysis and identically named timing models from the other library for minimum-delay analysis. The command for establishing this relationship between two libraries is the `set_min_library` command.

When analyzing a minimum delay value for a path containing a library cell, NanoTime looks for a max-min relationship for that library cell. If one exists, the tool uses the model from the minimum library. Otherwise, the tool uses the model from the maximum library. Only the maximum library should be listed in the link path; do not include the minimum library.

The `list_libs` command reports the max-min relationship using the uppercase letter “M” to indicate the maximum library and lowercase “m” to indicate the minimum library. The `report_lib` command shows any minimum library associated with a maximum library.

Example 2-1 shows a typical session using the `set_min_library` command.

Example 2-1 Typical Session Using the set_min_library Command

```
nt_shell> set search_path ". /data"
. /data
nt_shell> set link_path "* typical"
* typical
nt_shell> read_library CL180G/libcell/typical.db
Loading db file '/data/CL180G/libcell/typical.db'
1
nt_shell> read_library CL180G/libcell/slow.db
Loading db file '/data/CL180G/libcell/slow.db'
1
nt_shell> read_library CL180G/libcell/fast.db
Loading db file '/data/CL180G/libcell/fast.db'
1
nt_shell> register_netlist -format verilog rpt_cell.v
1
nt_shell> link_design rpt_cell
Compiling "rpt_cell.v"
Linking design rpt_cell...
Design 'rpt_cell' was successfully linked.
1
nt_shell> set_min_library -min_version fast typical
Created max/min library relationship:
  Max: /data/CL180G/libcell/typical.db:typical
  Min: /data/CL180G/libcell/fast.db:fast
1
```

The `list_libs` and `report_lib` commands display minimum and maximum library information, as follows:

```
nt_shell> list_libs
Library Registry:
    builtin_elements   builtin_elements:None
  m fast     /data/CL180G/libcell/fast.db:fast
    slow     /data/CL180G/libcell/slow.db:slow
  M typical  /data/CL180G/libcell/typical.db:typical
1
nt_shell> report_lib typical
 ...
Operating Conditions:
  Name              Process      Temp   Voltage
  --------------- --------- --------- ---------
  typical            1.0000   25.0000    1.8000
Min Library:
  /data/CL180G/libcell/fast.db:fast
 ...
Lib Cell        Attributes
--------------- ----------
OAI211X0        --
 ...
```

Using CCS Timing Models

The NanoTime tool accepts composite current source (CCS) timing models. Both pin-based and arc-based receiver models are accepted. The timing model might also contain CCS noise and CCS power information, but NanoTime uses only the CCS timing information.

To use CCS delay models, you must set the `ccs_enable_driver_delay_calculation` and `ccs_enable_receiver_delay_calculation` variables to `true` before the `check_design` command. Otherwise, the tool uses only NLDM models. A NanoTime Ultra license is required.

When both variables are set to `true`, NanoTime performs CCS delay calculation on any net with RC parasitics. The delay calculation is consistent with that of the PrimeTime tool, except that analysis is limited to a single operating condition. Multicorner-multimode analysis and scalable CCS models are not supported.

See Also

- Composite Current Source Models

Effects of Slew Thresholds on Model Use

NanoTime recognizes the `slew_derate_from_library` parameter in a .db or .lib file. The `rc_slew_derate_from_library` variable can perform the same function, if needed.

Calculating Lower and Upper Slew Thresholds

The interpretation of library data depends on the library parameters. The following statements might appear in a library (timing model):

```
slew_lower_threshold_pct_fall : 30.0;
slew_upper_threshold_pct_fall : 70.0;
slew_lower_threshold_pct_rise : 30.0;
slew_upper_threshold_pct_rise : 70.0;
input_threshold_pct_fall  : 50.0;
input_threshold_pct_rise  : 50.0;
output_threshold_pct_fall : 50.0;
output_threshold_pct_rise : 50.0;
slew_derate_from_library  : 1.0;
```

In this example, the cells were characterized using thresholds at 30 to 70 percent because `slew_derate_from_library` is set to 1.0.

If the `slew_derate_from_library` parameter is not set to 1.0, NanoTime adjusts the lower and upper thresholds. For example, if the `slew_derate_from_library` parameter value is 0.5, the cells were characterized using thresholds between 30 to 70 percent but are extrapolated to 10 to 90 percent for use in the timing analysis. Formulas from the library

vendor are used to calculate the 10 to 90 percent slew rate from the measured numbers at 30 to 70 percent.

The slew trip points of 10 to 90 percent are calculated as follows:

```
(new range) = (library threshold range)/slew_derate_from_library
(new range) = (70-30)/0.5 = 40 / 0.5 = 80
(new range) = (90-10)
```

In this example, the new range is twice the range found in the library, centered at the same value of 50 percent, which results in a new range of 10 to 90 percent.

If a vendor specifies infeasible values for the `slew_derate_from_library` parameter and the trip points, NanoTime cannot compute the slew correctly.

Interpreting Libraries With Different Settings

If the value of the `slew_derate_from_library` parameter is not 1.0, the interpretation of the library is fundamentally changed. Consider the simple circuit in Figure 2-2 and two timing models for it that are identical except for the `slew_derate_from_library` parameter. In model_1-0.lib, the value is 1.0, while in model_0-5.lib, the value is 0.5.

Both libraries contain the following statements:

```
slew_lower_threshold_pct_fall:  30.0
slew_upper_threshold_pct_fall:  70.0
slew_lower_threshold_pct_rise:  30.0
slew_upper_threshold_pct_rise:  70.0
```

Both libraries contain tables for three capacitance values, one of which is 0.00354 pf; this value is needed for the example in Figure 2-2. The values of the `cell_fall` and `fall_transition` parameters for this capacitance appear in Table 2-6 and Table 2-7. The values are the same for both libraries.

Table 2-6 Values of cell_fall Parameter for One Output Capacitance

cell_rise	0.00030 pf	0.00354 pf	0.0385 pf
0.1 ns	...	0.43463	...
0.2 ns	...	0.54583	...
0.4 ns	...	0.66370	...

Table 2-7 Values of fall_transition Parameter for One Output Capacitance

cell_rise	0.00030 pf	0.00354 pf	0.0385 pf
0.1 ns	...	0.05000	...
0.2 ns	...	0.10000	...
0.4 ns	...	0.40000	...

In the circuit shown in Figure 2-2, an input transition of 0.3 ns at 20 to 80 percent thresholds is applied to the input port in. The capacitance on net out is 0.00354 pF.

Using model_1-0.lib

When model_1-0.lib is used, the 0.3 ns transition at 20 to 80 percent is converted to a 0.2 ns transition at 30 to 70 percent by interpolation. The delay (fall) on net out is determined to be 0.54583 ns and the transition time (fall) is 0.100 ns, based on the values in Table 2-6 and Table 2-7 that correspond to a `cell_rise` value of 0.2 ns. Since the transition time of 0.1 ns is at 30 to 70 percent, the full transition time is 0.25 ns (reported under the column heading Ftrans), determined by extrapolation to the rail voltages.

Figure 2-2 Threshold Applied to Input Port

Part of the resulting path report is as follows:

```
   Path      Incr    Adjust    Trans  FTrans      Cap NT      Point                   Net
-------  -------  -------  ------- ------- ------- --- -- ---------------------- ----
                   0.00000                                  input external delay
0.00000            0.00000  0.30000 0.50000 0.00354  D   f in (in)                 in
0.00000  0.00000           0.30000 0.50000 0.00354  M   f inv_1/A (inv10)         in
0.54583  0.54583           0.10000 0.25000 0.00354  O   r out (out)               out
0.54583                                                    data arrival time
         0.54583  0.00000                                    Total
```

Using model_0-5.lib

With 30 to 70 percent thresholds and the value of the `slew_derate_from_library` parameter set to 0.5, the effective thresholds of library model_0-5.lib are 10 to 90 percent. When model_0-5.lib is used, as shown in Figure 2-3, the 0.3 ns transition at 20 to 80 percent is extrapolated to a 0.4-ns transition at 10 to 90 percent. The delay (fall) on net out is determined to be 0.66370 ns and the transition time (fall) is 0.400 ns, based on the values in Table 2-6 and Table 2-7 that correspond to a `cell_rise` value of 0.4 ns. Since the transition time 0.4 ns is at 10 to 90 percent, the full transition time (FTrans) is 0.500 ns by extrapolation to the rail voltages.

Figure 2-3 Threshold Applied to Input Port

Part of the resulting path report is as follows:

```
   Path      Incr    Adjust    Trans  FTrans     Cap NT     Point                    Net
-------  -------  -------  ------- ------- ------- --- -- ---------------------- ----
                   0.00000                                  input external delay
0.00000            0.00000  0.30000 0.50000 0.00354  D   f in (in)                  in
0.00000  0.00000            0.30000 0.50000 0.00354  M   f inv_1/A (inv10)          in
0.66370  0.66370            0.40000 0.50000 0.00354  O   r out (out)                out
0.66370                                                     data arrival time
         0.66370  0.00000                                   Total
```

Precedence Rules for Slew Derate Factors

Slew derate factors have the following precedence:

1. The `slew_derate_from_library` parameter specified in a .lib or .db library or database
2. The `rc_slew_derate_from_library` NanoTime variable

When a library includes the `slew_derate_from_library` parameter, and you also define the `rc_slew_derate_from_library` NanoTime variable, the library parameter has a higher priority. The `rc_slew_derate_from_library` variable is applied only when the `slew_derate_from_library` parameter is not defined in the library, or its value is 1.0.

During timing analysis, there are three ways to specify lower and upper slew thresholds, listed in order of precedence:

1. Using parameters from a timing model

```
slew_lower_threshold_pct_fall
slew_upper_threshold_pct_fall
slew_lower_threshold_pct_rise
slew_upper_threshold_pct_rise
```

2. Locally, using the `set_measurement_threshold` command

```
set_measurement_threshold -slew_lower_threshold_pct_fall
set_measurement_threshold -slew_upper_threshold_pct_fall
set_measurement_threshold -slew_lower_threshold_pct_rise
set_measurement_threshold -slew_upper_threshold_pct_rise
```

3. Globally, using variables

```
rc_slew_lower_threshold_pct_fall
rc_slew_upper_threshold_pct_fall
rc_slew_lower_threshold_pct_rise
rc_slew_upper_threshold_pct_rise
```

For the delays through timing models, only the thresholds from the .lib or .db files are considered; the global and local settings are ignored. For the delays through the rest of the design, the global and local settings are used.

When there are multiple .lib or .db files, each cell uses the `slew_derate_from_library` parameter value from the library it comes from. Also, the transition time (Trans) and full transition time (FTrans) are reported according to each cell's convention. Therefore, the ratio of transition times to full transition times could be different in a single path.

Example With Different Slew Derate Factors

In the circuit shown in Figure 2-4, inv1 is a timing model that uses data from the model_0-5.lib library.

Figure 2-4 Mixed .lib Cell and Transistor Example

The threshold, slew derate variables, and `set_measurement_threshold` commands are applied as shown:

```
set rc_slew_lower_threshold_pct_fall 10.0
set rc_slew_upper_threshold_pct_fall 90.0
set rc_slew_lower_threshold_pct_rise 10.0
set rc_slew_upper_threshold_pct_rise 90.0
set rc_slew_derate_from_library 0.88
set_measurement_threshold -slew_lower_threshold_pct_fall 40 [get_nets { net1 out }]
set_measurement_threshold -slew_lower_threshold_pct_rise 40 [get_nets { net1 out }]
set_measurement_threshold -slew_upper_threshold_pct_fall 60 [get_nets { net1 out }]
set_measurement_threshold -slew_upper_threshold_pct_rise 60 [get_nets { net1 out }]
```

NanoTime produces the following path report:

Example 2-2 Path Report

```
   Path     Incr   Adjust    Trans  FTrans     Cap NT     Point                  Net
-------  -------  -------  ------- ------- ------- --- -- ---------------------- ----
                   0.00000                                 input external delay
0.00000            0.00000  0.40000  0.50000 0.00354 D   f in (in)                in
0.00000  0.00000            0.40000  0.50000 0.00354 M   f inv_1/A (inv05)        in
0.66372  0.66372            0.40000  0.50000 0.00354     r inv_2/m0/g (inv)       net1
0.68091  0.01720            0.04732  0.23659 0.00500 O   f out (out)              out
0.68091                                                     data arrival time
         0.68091  0.00000                                   Total
```

On net in, the thresholds are 10 to 90 percent, as specified by the rc_slew* variables. The ratio of transition and full transition times is 0.8.

On net1, the thresholds are 10 to 90 percent and the ratio of transition and full transition times is 0.8. The thresholds specified in the model are 30 to 70 percent, but the model also contains the `slew_derate_from_library = 0.5` statement, which causes extrapolation of the transition (see Using model_0-5.lib).

On net out, the thresholds are 40 to 60 percent as specified by the `set_measurement_threshold` commands.

Defining the Voltage Environment

This procedure illustrates typical usage of the commands and variables that define the voltage environment.

1. Add power supply and ground net names to the `link_vdd_alias` and `link_gnd_alias` variables.
2. For accurate transistor capacitances, include in the voltage directive of a SPICE technology file all of the supply voltages that a transistor of that type might encounter.

3. Set the `tech_default_voltage` variable to provide a supply voltage to use when a SPICE technology file does not include a voltage directive.
4. Use the highest supply voltage when defining the gate and drain voltage sweeps in SPICE technology files.
5. Set the `oc_global_voltage` variable for use as the global default voltage (typically either the largest voltage or the most common voltage).
6. Execute the `link_design` command.
7. Use the `set_supply_net` command to identify nets as being supply and ground nets if they were not already defined by the `link_vdd_alias` and `link_gnd_alias` variables.
8. Use the `set_supply_net` command with the `-virtual` option to identify nets as virtual supplies. (This option requires a NanoTime Ultra license.)
9. Use the `set_voltage` command to specify voltages on supply nets that were identified with the `set_supply_net` command or whose voltages were not defined in the netlist.

   Use the `set_voltage` command with the `-min`, `-max`, `-min_clock`, and `-max_clock` options to specify voltages to use for minimum and maximum delay analysis. (These options require a NanoTime Ultra license.)
10. For use in timing analysis, use the `set_input_transition` command with the `-rail_voltage` option to identify input ports that have switching voltages different from the `oc_global_voltage` variable value.
11. For use in timing characterization, use the `set_model_input_transition_indexes` command with the `-rail_voltage` option to identify input ports that have switching voltages different from the `oc_global_voltage` variable value.
12. For use in extracted timing models, set the `model_apply_oc_global_voltage` variable to `true` to use the value of the `oc_global_voltage` variable instead of supply voltages set with the `set_voltage` command.
13. Proceed with clock propagation and topology recognition operations.

Specifying Power Supply Nets and Ground Nets

The `link_vdd_alias` and `link_gnd_alias` variables contain the default power supply and ground net names that NanoTime uses during design linking. The variable defaults are as follows:

```
nt_shell> printvar link_vdd_alias
link_vdd_alias = "vdd vcc !vdd !vcc vdd! vcc! vdd1 vdd2 vdd#"
nt_shell> printvar link_gnd_alias
link_gnd_alias = "gnd vss 0 !gnd !vss gnd! vss! vss1 vss2 vss#"
```

To use other power supply and ground net names, append them to the default lists in the variables before running the `link_design` command. This process is especially important for designs containing back-biased transistors where the bulk connection is driven by a different power supply net. Unresolved power supply nets or ground nets might increase the runtime of the `link_design`, `match_topology`, and `check_topology` commands.

For example, the following command adds vcc#1, vcc#2, and similar names to the default supply voltage list:

```
nt_shell> lappend link_vdd_alias " vcc#*"
vdd vcc !vdd !vcc vdd! vcc! vdd1 vdd2 vdd# vcc#*
```

If power supplies are not defined in the netlist, use the `set_supply_net` command after the `link_design` command to identify power supply nets and ground nets in the design. The following commands identify all nets matching the v* name pattern as power supply nets and all nets matching the gn* name pattern as ground nets:

```
nt_shell> set_supply_net v*
1
nt_shell> set_supply_net -gnd gn*
1
```

To specify virtual supply nets, use the `set_supply_net` command with the `-virtual` option. A virtual supply net is used like an actual supply net. Typically, it is connected to a true supply or ground net through a single transistor (sometimes called a sleeper transistor or power switch transistor) that has a very low resistance. A net cannot be defined as both a true supply and a virtual supply. You must have a NanoTime Ultra license to define a virtual supply net.

The voltage on a virtual supply net might be different from the true supply voltage due to IR drop across the power switch transistor. Use the `set_supply_net` command to specify the actual voltage on this net. If you do not set the voltage on a virtual supply net, NanoTime includes the power switch devices in the simulation for each relevant stage in an attempt to account for the IR drop. However, this is only an approximation because the true effect depends on the dynamic behavior of all stages connected to the power switch transistors.

Use the `remove_supply_net` command to undo a supply net definition.

Setting Voltage Values

To specify the voltage values of power supply nets, use the `set_voltage` command. Setting the voltage on a power supply net overrides any previous voltage values for the net.

The `oc_global_voltage` variable provides the voltage for any net identified as a supply net that does not have a specific voltage setting. NanoTime issues an OPCD-001 warning message at the path tracing step when it detects such nets. The message includes the net name and the default voltage to be used for delay analysis.

NanoTime also uses the value of the `oc_global_voltage` variable as the default rail voltage for generating input signal waveforms. To ensure reasonable generation of input waveforms by default, set the `oc_global_voltage` variable to the main rail voltage early in the design flow. You can override this setting for individual input waveforms by using the `set_input_transition` command with the `-rail_voltage` option.

When creating extracted timing models, NanoTime uses the highest design voltage for setting the nominal voltage value in the model, by default. However, you can change this behavior by setting the `model_apply_oc_global_voltage` variable to `true`. In this case, the model nominal voltage is set to the value in the `oc_global_voltage` variable instead of any voltages defined with the `set_voltage` command. The nominal voltage for consumed timing models is not affected.

Analyzing Multivoltage Designs

NanoTime provides several options for analyzing multivoltage circuits.

Designs with only one supply voltage per stage, such as the examples shown in Figure 2-5, are handled natively. However, for multivoltage designs, you should enable nonlinear waveform analysis to accurately represent the waveforms between voltage or power domains.

Figure 2-5 Multivoltage Circuits

Stage delays for multivoltage designs are measured at 50 percent of the supply voltage, based on the output swing of each individual stage. If a stage has paths to more than one supply voltage, the measurement is based on the supply voltage associated with that path. This is also true for the slope, rise time, and fall time calculations.

Each channel-connected block can have only one supply voltage. If NanoTime detects more than one supply voltage in a single stage, the tool issues an NTL-017 warning message and chooses one voltage for the simulation. For best accuracy, specify all supply voltages explicitly.

For multivoltage designs, you must use the `-rail_voltage` option whenever you use the `set_input_transition` or `set_model_input_transition_indexes` command for all cases where the voltages are different from the design default voltage.

Different supply voltages cause transistors to have different calculated capacitances. To calculate the correct lookup tables, include all supply voltage directives in the SPICE model file. If a SPICE model file does not include a supply voltage directive, the value of the `tech_default_voltage` variable is used.

See Also

- Nonlinear Waveform Analysis

Design Reporting

NanoTime provides the following commands for reporting the characteristics of a loaded design after design linking is complete:

- The `report_cell` command lists all cells in the current instance or displays information about the nets and connected pins of specified cells.
- The `report_design` command reports the attributes of the current design, including unit sizes and operating conditions.
- The `report_hierarchy` command lists the instance names in the hierarchy of the design, block by block, down a specified number of levels.
- The `report_lib` command reports the unit sizes, operating conditions, and cells of the currently loaded libraries.
- The `report_net` command lists the nets and their characteristics such as fanin, fanout, port connections, pin connections, and capacitance.
- The `report_port` command lists the input and output ports and their characteristics such as direction, capacitance, and input or output delay.
- The `report_technology` command reports the details of technology model entries in the logic libraries, including operating conditions, transistor parameters, and transistor lookup table indexes. The report only includes transistor models that are used in the timing analysis. The report does not include models that are read in but never used or models that are replaced due to back-annotation of DSPF files.

Parasitic Data Overview

Timing analysis is more accurate if parasitic information is included. After routing, you use the `read_parasitics` command to back-annotate the design with a parasitic device netlist generated by a parasitic extraction tool.

Before routing, actual parasitics are not available. To provide estimated output load capacitances, use the `set_load` command.

Parasitic extraction tools analyze the layout of a design to determine the location and size of parasitic elements. Examples of parasitic resistances are the resistances of interconnect wires and vias between layers. Examples of parasitic capacitances are the capacitances between wires and the internal capacitances of transistors and other devices.

The input to the extraction step is the output of a layout versus schematic (LVS) tool. You might need to adjust the setup of both the LVS and extraction tools to generate a parasitic netlist that the NanoTime tool can use. The parasitic netlist must contain the locations of the parasitic elements using net names that match the net names in the design netlist.

NanoTime accepts both SPEF (Standard Parasitic Exchange Format) and DSPF (Detailed Standard Parasitic Format) files.

Depending on the extraction commands, parasitic data might be specified in detailed or reduced forms. NanoTime ignores resistors to ground and, by default, splits coupling capacitors into two capacitors to ground. After reading in all of the parasitic information, the tool discards incomplete networks.

NanoTime reduces complex annotated RC parasitics to minimize the amount of parasitic data and the subsequent analysis runtime. The tool combines or removes RC sections from each net to make a simpler representation that has a path delay within a tolerance of the original path delay. The reduction operation combines multiple capacitors between two nodes, but retains multiple resistors as separate elements.

See Also

- Estimating Output Loads
- Generating Parasitic Netlist Files With an Extraction Tool
- Using Parasitics in the NanoTime Flow

Estimating Output Loads

When extracted parasitics are not available, you can provide estimates of load capacitances to use in timing analysis.

Capacitance on Ports

For ports, use the `set_load` command to specify the capacitance (in design units) and the ports to which that capacitance value applies. For example, the following command sets a pin capacitance of 2.5 units on the port named ox1:

```
nt_shell> set_load 2.5 [get_ports ox1]
```

The `report_port` command reports the load settings, as shown in the following example:

```
nt_shell> report_port ox1
 ...
Attributes:
    s  - Same phase checking
    n  - No same phase checking

              Min Rise Min Fall Max Rise Max Fall
Port    Dir        Cap      Cap      Cap      Cap Attrs
------- ----- -------- -------- -------- -------- -----
ox1     inout    2.500    2.500    2.500    2.500
```

The `-add` option adds the specified value to the existing capacitance. Otherwise, the specified value replaces the existing value.

You can use the `-pin_load` and `-wire_load` options to specify whether the load value is the pin capacitance or wire capacitance of the port. If both are used, the command sets both the pin capacitance and the wire capacitance. If neither option is used, NanoTime sets the pin capacitance only (the same as using the `-pin_load` option alone).

You can use the `-min` or `-max` option to specify different load values for minimum or maximum delay analysis. If neither is specified, the same value applies to both. Similarly, you can use the `-rise` or `-fall` option to apply the load value to rising or falling edges only. Otherwise, it applies to both rising and falling edges.

To remove capacitance values that have been set on a port, use the `remove_capacitance` command. To reset all annotated capacitance values in a design, use the `reset_design` command.

Capacitance on Nets

For nets, use the `set_load` command to specify the capacitance (in design units) and the nets to which that capacitance value applies. For example, the following command sets a pin capacitance of 3 units on the net named U1.U2.net3:

```
nt_shell> set_load 3 [get_nets U1.U2.net3]
```

The `-add` option adds the specified value to the existing capacitance. Otherwise, the specified value replaces the existing value.

By default, the `set_load` command does not affect any net that has been back-annotated by the `read_parasitics` command. Attempting to use the `set_load` command in this case results in a warning message, with no change to the net. To allow the `set_load` command to override back-annotated capacitance, set the `parasitics_allow_spf_net_override` variable to `true` before you execute the `set_load` command.

To obtain a report of the capacitance values on a net, use the `report_net` command with the `-connections` option. To view a list of the capacitance attributes of a port or net, use the `report_attribute` command, as shown in the following examples:

```
nt_shell> report_attribute -application -class port port_name
nt_shell> report_attribute -application -class net net_name
```

You can use the `-min` or `-max` option to specify worst-case minimum and maximum load values. If neither is specified, the same value applies to both. Similarly, you can use the `-rise` or `-fall` option to apply the load value to rising or falling edges only. Otherwise, it applies to both rising and falling edges.

To remove capacitances on a net, use the `remove_capacitance` command. To reset all annotated capacitances in a design, use the `reset_design` command.

Capacitance Attributes

Each port or net can have different values for wire capacitance and pin capacitance, for rising and falling transitions, for minimum and maximum delay analysis values, and for data paths and clock paths. Therefore, a port or net can have up to 16 different capacitance values. The total capacitance of a net is the sum of the pin, port, and wire capacitance values associated with that net.

Each of the 16 possible capacitance values for a port or net is assigned to an attribute.

For the `port` object class, the attributes have names in the format `pin_capacitance_*` and `wire_capacitance_*`. There are eight of each type, representing combinations of edge type, minimum or maximum delay analysis, and clock or data path.

For the `net` object class, the attributes also have names in the format `pin_capacitance_*` and `wire_capacitance_*`. In addition, the sums of the pin and wire capacitances for each of the eight conditions are also available in attributes named `total_capacitance_*`.

The following example uses the `get_attribute` command to retrieve the value of the `pin_capacitance_fall_max` attribute on pin X5.Mn0.d, then sets that value on ports in1 and in2.

```
nt_shell> set_load [get_attribute [get_lib_pins X5.Mn0.d] \
                   pin_capacitance_fall_max] [get_ports {in1 in2}]
```

See Also

- Generating Parasitic Netlist Files With an Extraction Tool
- Using Parasitics in the NanoTime Flow

Generating Parasitic Netlist Files With an Extraction Tool

The setup parameters for the LVS tool and extraction tool affect whether NanoTime can use the generated parasitic netlist effectively.

Use the following guidelines when setting up your extraction tool:

- Complete a full layout versus schematic (LVS) check of your design before the extraction step to generate complete cross-references and resolve all errors.
- Use schematic names, not layout names, to allow NanoTime to associate the parasitic devices with the design netlist.
- Do not include coupling capacitors between pins on the same net.
- Check the compatibility of notation conventions such as device terminal names, fingered device delimiter characters, uppercase and lowercase conventions, net name hierarchy separators, and bus notation. Such items can often be specified in both the extraction tool and in the NanoTime tool; therefore, it is critical to ensure that the specifications match. You might also need to check the notation provided by the LVS tool.
- Avoid most types of instance merging because the correspondence with schematic devices might be affected.

Synopsys StarRC Extraction Tool Setup

If you are using the StarRC tool to generate the parasitic netlist, the following commands might be necessary in the StarRC command file. Other StarRC commands and options might also be applicable for your analysis. For more information, see the *StarRC User Guide and Command Reference*.

- Use the `XREF: YES` command to enable schematic cross-referencing.
- Use the `NETLIST_NODENAME_NETNAME: NO` command to preserve the correct naming scheme. You can leave the command out of the command file because the default is `NO`.

- Use the `INTRANET_CAPS: NO` command to remove intranet capacitances. You can leave the command out of the command file because the default is `NO`.
- (Optional) Specify a file in which to save device parameters by using the `NETLIST_IDEAL_SPICE_FILE` command. In the NanoTime tool, read this file by using the `read_device_parameters` command.
- (Optional) If you plan to perform NanoTime signal integrity analysis, use the StarRC `COUPLE_TO_GROUND: NO` command to retain coupling capacitances in the netlist. The command default is `YES`, so you must include this command in the StarRC command file.
- (Optional) If the transistor device models contain transistors implemented in a .subckt block (which is common for advanced process nodes), use the `HN_NETLIST_SPICE_TYPE: model_name X` command to use the letter X as a prefix for the devices.
- (Optional) If the circuit contains designed diode structures such as antenna diodes, use the `CONVERT_DIODE_TO_PARASITIC_CAP` command to provide capacitance conversion factors.
- Use the `NETLIST_MERGE_SHORTED_PORTS: NO` command to avoid merging ports. You can leave the command out of the command file because the default is `NO`.
- Ensure that the NanoTime `parasitics_xref_layout_instance_prefix` variable setting matches the StarRC `XREF_LAYOUT_INSTANCE_PREFIX` command setting.

  Similarly, ensure that the NanoTime `parasitics_xref_layout_net_prefix` variable setting matches the StarRC `XREF_LAYOUT_NET_PREFIX` command setting.
- (Optional) If you use contact resistance analysis by setting the NanoTime `parasitics_enable_rail_net_resistance` variable, consider restricting the number of layers to obtain a runtime benefit.

See Also

- Parasitic Data Overview
- Using Parasitics in the NanoTime Flow

Using Parasitics in the NanoTime Flow

The following procedure describes the general steps required to annotate a parasitics netlist in the NanoTime analysis flow.

The flow allows some flexibility in the placement of many of the commands that read parasitics files and set related variables. This procedure recommends placing these

commands after the `check_topology` command. The commands must be used before the `check_design` command.

1. Obtain a design netlist and one or more parasitics netlists for your design.
2. Specify naming conventions with the `link_*_name` variables to match the contents of the parasitics netlist. For more information, see Pin and Transistor Naming Conventions.

   For example, if you are using the StarRC extraction tool, use the following NanoTime commands:

   ```
   set link_transistor_drain_pin_name DRN
   set link_transistor_gate_pin_name GATE
   set link_transistor_source_pin_name SRC
   set link_transistor_bulk_pin_name BULK
   ```

3. (Optional) If device macro models contain parasitic elements and corner-specific wrapper subcircuits, set the `link_enable_wrapper_subckt_parasitics` variable to `false`. (The default is `true`.) In this case, you must read the SPICE models and set up a single-corner flow. For more information, see Linking Designs With Embedded Parasitics.
4. (Optional) If you discard subcircuits that might be referenced in parasitics files, set the `link_discard_subckt_contents` variable. For more information, see Retaining Boundary Pins for Discarded SPICE Subcircuits.
5. Set other `link_*` variables as needed to enable NanoTime to read and interpret both the design netlist and the parasitics netlist correctly.
6. Link the design with the `link_design` command.

   If the design netlist contains parasitics and you intend to perform signal integrity analysis, use the `-keep_capacitive_coupling` option to preserve the parasitics.

   If the design netlist contains parasitics and you do not plan to perform signal integrity analysis, use the `-coupling_reduction_factor` option to split coupling capacitors to ground with an optional scaling factor.
7. Define topologies and verify successful marking with the `check_topology` command.
8. (Optional) Set the `parasitics_enable_mapping_unresolved_pins` variable to `true` to preserve parasitic elements attached to boundary pins or to nets connected to boundary pins of a timing model or discarded subcircuit. (The default is `false`.) For more information, see Handling Parasitics For Subcircuits.
9. (Optional) For fingered devices, set the `parasitics_enable_drain_source_swap` variable to `true`. (The default is `false`.) For more information, see Special Device Handling.

10. (Optional) To allow terminal swapping for two-terminal devices such as resistors, use the `set_allow_pin_swap` command to specify the terminal names. For more information, see Special Device Handling.

11. (Optional) If you plan to use the `set_load` command later in the flow to specify load capacitance on specific ports or pins, set the `parasitics_allow_spf_net_override` variable to `true`. (The default is `false`.)

12. (Optional) Read device parameter files with the `read_device_parameters` command.

13. (Optional) Use the `report_annotated_parasitics` command to check the annotations. For more information, see Checking and Reporting Annotated Parasitics.

14. (Optional) Use the `complete_net_parasitics` command to complete any partially annotated nets. This command is one of three methods to accomplish this goal. For more information, see Completing Partially Annotated Nets.

15. (Optional) To include rail net parasitics, set the `parasitics_enable_rail_net_resistance` variable to `true`. For more information, see Using Rail Net Contact Resistance in Delay Analysis.

16. (Optional) To include RC delay between connected trigger devices in the analysis, use the `set_enable_input_spf_skew` command. For more information, see Including the Effects of Common Trigger Skew.

17. (Optional) If you plan to create a boundary cell timing model, set the `rc_reduction_exclude_boundary_nets` variable to `true` to prevent the boundary parasitics from being modified by the parasitic reduction operation. (The default is `false`.)

    In addition, you must set the `timing_save_wire_delay` variable to `true`. (The default is `false`.) Setting this variable also enables you to report wire delay values in path reports. For more information, see Boundary Parasitics.

18. (Optional) Set other parasitics-related variables as needed. For more information, see Variables for Parasitic Data Analysis.

19. Use the `read_parasitics` command one or more times to read the parasitics file or files. For more information, see Working With Parasitics Files.

    If you are reading a parasitics file for a cell or block, use the `-increment` option to retain all previously annotated parasitic data on nets that are listed in this file.

    Use the `-complete_with` option to complete any partially annotated nets. This option is one of three methods to accomplish this goal. For more information, see Completing Partially Annotated Nets.

20. Execute the `check_design` command. If you have not already used an option or command to resolve partially annotated nets, use the `-complete_with` option.

See Also

- Parasitic Data Overview
- Generating Parasitic Netlist Files With an Extraction Tool

Working With Parasitics Files

The following sections provide details about reading and using parasitics files:

- Using the read_parasitics Command
- Exporting and Removing Parasitics
- Checking and Reporting Annotated Parasitics
- Handling Parasitics For Subcircuits
- Special Device Handling
- Completing Partially Annotated Nets
- Preserving Cross-Coupling Capacitors
- Using Rail Net Contact Resistance in Delay Analysis
- Including the Effects of Common Trigger Skew

Using the read_parasitics Command

The `read_parasitics` command reads parasitic data from a file and annotates the information on nets in the current design. For example:

```
nt_shell> read_parasitics -format SPEF adder.spef
```

The `read_parasitics` command can read parasitic data files in SPEF and DSPF formats, either in plain ASCII format or compressed with gzip.

Note:
All keywords in parasitic data files must be uppercase. This includes terms such as *|DSPF, *|NET, *|P, and *|I, where * and | are the asterisk and vertical bar characters.

Net and instance pin names in the parasitic data file must match names in the design. Port names in the parasitic data file are sometimes split, depending on the settings of the parasitic extraction or layout versus schematic tools. For example, port abc might appear as abc_0, abc_1, and abc_2 in the parasitics file. To merge split port names, the `read_parasitics` command provides the `merge_spef_split_ports` utility, which allows you to specify the formats to be merged. In the following example, port names that have a

name followed by an underscore and a single integer will be merged into one port with the prefix name.

```
read_parasitics -format SPEF [merge_spef_split_ports -file xyz.spef \
                              -regexp {_[0-9]+$}]
```

The back-annotated parasitic data replaces any existing settings made previously with the `set_load` command. By default, parasitic data back-annotated with the `read_parasitics` command cannot be changed by subsequent instances of the `set_load` command. To allow such changes, set the `parasitics_allow_spf_net_override variable` to `true`.

In some cases, parasitics are embedded in netlist cells, but there is also a top-level parasitics file. In this case, the `read_parasitics` command must include the `-increment` option to preserve the netlist parasitics.

For example, consider a design named Xtop with four blocks, as shown in Figure 2-6. Blocks X1, X2, X3, and X4 are instantiations of subcircuits blk_1, blk_2, blk_3, and blk_4, respectively.

Figure 2-6 Example of Hierarchical Parasitic Annotation

The NanoTime commands to read the parasitics files are as follows:

```
nt_shell> read_parasitics -path X1 -increment blk_1.spef -format spef
nt_shell> read_parasitics -path X2 -increment blk_2.spef -format spef
nt_shell> read_parasitics -path X3 -increment blk_3.spef -format spef
nt_shell> read_parasitics -path X4 -increment blk_4.spef -format spef
nt_shell> read_parasitics top.spef
```

The `-path` options specify the relative paths from the current design to the hierarchical instance names for which the parasitics files have been created. The `-increment` options tell NanoTime to keep all previously annotated parasitic data on nets that are listed in the SPEF file. The `-increment` option is not necessary when reading the top-level parasitics file because it does not name nets included in the other files.

Upon completion of reading and annotating the parasitics, NanoTime generates an annotation report (similar to the `report_annotated_parasitics -check` command):

```
nt_shell> read_parasitics -format SPEF adder.spef
Reading file '/.../adder.spef'
****************************************
Report : read_parasitics /.../adder.spef
 ...
****************************************
 0 error(s)
 Format is SPEF
 Annotated nets           :         7460
 Annotated capacitances   :        86165
 Annotated resistances    :        78705
****************************************
Report : report_annotated_parasitics
 ...
                                                Lumped        Not  Annotation
Net Type            Total RC network  Capacitance  Annotated  Incomplete
------------------ ------ ---------- ------------ ----------  ----------
Internal nets        7252       7250            1          2           0
Boundary/port nets    210        210            0          0           0
-----------------------------------------------------------------------
                     7462       7460            1          2           0
```

You can suppress this report by using the `-quiet` option of the `read_parasitics` command.

Exporting and Removing Parasitics

To write a parasitic data file that can be used later, use the `write_parasitics` command.

To remove all annotated parasitics, use the `remove_annotated_parasitics` command. This command removes all parasitics, whether they are included in the design netlist or read in from a parasitics file with the `read_parasitics` command.

To remove annotated device parameters (DPF data), use the `remove_annotated_device_parameters` command.

The `reset_design` command removes all attributes from the current design, including annotated parasitics.

Checking and Reporting Annotated Parasitics

The following command checks the parasitic data file adder.spef for syntax errors but does not annotate the current design.

```
nt_shell> read_parasitics -syntax_only adder.spef
```

After parasitics have been annotated on the design, NanoTime generates an annotation report. You can generate the same report by using the `report_annotated_parasitics` command. For example,

```
nt_shell> report_annotated_parasitics
 ...

                                          Lumped        Not  Annotation
Net Type             Total RC network  Capacitance  Annotated  Incomplete
------------------- ------ ---------- ------------ ---------- -----------
Internal nets         7250       7249            1          0           0
 - Pin to pin nets    7249       7248            1          0           0
 - Driverless nets       1          1            0          0           0
 - Loadless nets         0          0            0          0           0
-----------------------------------------------------------------------
Boundary/port nets     212        210            0          2           0
 - Pin to pin nets     210        210            0          0           0
 - Driverless nets       2          0            0          2           0
 - Loadless nets         0          0            0          0           0
-----------------------------------------------------------------------
                      7462       7459            1          2           0
```

The report shows the number of nets that have been annotated with detailed parasitics (RC networks) and reduced parasitics (lumped capacitance). The report counts nets without considering hierarchical crossings.

Three options are available for the `report_annotated_parasitics` command to provide control over which nets appear in the report. The options are as follows:

- The `-pin_to_pin_nets` option reports only the nets that have at least one global driver and at least one global load.
- The `-driverless_nets` option reports only the nets that do not have a global driver.
- The `-loadless_nets` option reports only the nets that do not have a global load.

Nets that are missing both a global driver and a global load are counted as driverless nets.

The `-check` option verifies that all annotated networks are complete, with the driver of each net connected through the RC network to the fanouts of the net. The `-internal_nets` and `-boundary_nets` options restricts reporting to internal nets (nets connected only to cell pins) or boundary nets (nets connected to ports). By default, both types of nets are reported.

The `-list_annotated` option lists all nets that have been annotated, while the `-list_not_annotated` option lists the nets that have not been annotated. For example,

```
nt_shell> report_annotated_parasitics -list_not_annotated
 ...

1,  gnd (driver: pin Xls0.Mn8.D)
2,  vdd (driver: pin Xaddsub.Xi0.Mp0.D)
...
```

You can restrict the report to specified nets. For example,

```
nt_shell> report_annotated_parasitics {n5* n61 n62}
```

Handling Parasitics For Subcircuits

The following sections describe methods for correctly handling parasitic back-annotation to subcircuits and library cells (timing models).

Name Mapping for Macro Models

The `parasitics_device_name_mapping_method` variable defines the priority order of methods for mapping device names in parasitics files to the logical netlist. Table 2-8 lists the valid arguments. The default is the following string:

```
direct prepend_x prepend_m prepend_d prepend_r prepend_c   \
    replace_leading_m_with_x replace_leading_r_with_x      \
    replace_leading_c_with_x replace_leading_d_with_x
```

Table 2-8 Name Mapping Variable Arguments

Argument	Description
`direct`	No mapping; the name in the parasitics file is compared directly with the logical name
`prepend_c`	For capacitors, prepends a C to the name in the parasitics file
`prepend_d`	For diodes, prepends a D to the name in the parasitics file
`prepend_m`	For transistors, prepends an M to the name in the parasitics file
`prepend_r`	For resistors, prepends an R to the name in the parasitics file
`prepend_x`	For macro models, prepends an X to the name in the parasitics file
`replace_leading_c_with_x`	Replaces a leading C with an X
`replace_leading_d_with_x`	Replaces a leading D with an X
`replace_leading_m_with_x`	Replaces a leading M with an X
`replace_leading_r_with_x`	Replaces a leading R with an X

NanoTime tries each method, in order of appearance in the variable, until a logical device is found that corresponds to the physical device name. The optimal ordering for a specific design depends on the configuration of the layout versus schematic (LVS) and extraction tools used to generate the parasitics netlist. The default order is successful for most designs.

Annotating Parasitics to Boundary Pins

When you discard a subcircuit or use a timing model (black box), NanoTime has no information about the contents of the discarded subcircuit or the timing model. However, parasitics might be associated with pins inside one of these subcircuits. By default, NanoTime issues an error message when a parasitic netlist references pins inside a subcircuit because the interior pins are not part of the netlist database.

You can preserve parasitic elements attached to black box boundary pins or to nets connected to those pins by setting the `parasitics_enable_mapping_unresolved_pins` variable to `true`. The preserved parasitics are reassigned to the associated boundary pin. All other parasitics inside the model or subcircuit are ignored.

This operation applies to all timing models and to SPICE subcircuits discarded by using the `link_discard_subckt_contents` variable. You should consider the following sources of error that might lead to delay calculation inaccuracy:

- For timing models, the pin capacitance of a boundary pin is preserved, in addition to the extracted parasitics that are reassigned to the boundary pin. Some capacitance might be duplicated, depending on the details of the parasitic extraction. Also, the pin capacitance might not be accurate, depending on how it was defined during model creation.
- For discarded subcircuits, no pin capacitance exists to model the effects of the load inside the subcircuit.

Set the `parasitics_enable_mapping_unresolved_pins` variable before executing the `read_parasitics` command.

Note:
Only DSPF netlists are supported for this capability.

Parasitics Associated With Pins in a Timing Model

Figure 2-7 shows a design that contains a subcircuit named X1. The capacitors and resistors in this figure are parasitic devices determined by an extraction tool.

Figure 2-7 Design With Subcircuit X1

When you substitute a timing model (black box) for subcircuit X1, the netlist database for this circuit is as shown in Figure 2-8. The effective capacitance of the extracted parasitics from point E to point C are represented in the timing model by the pin capacitance of pin E.

Figure 2-8 Replacing Subcircuit X1 With a Timing Model

If you set the `parasitics_enable_mapping_unresolved_pins` variable to `true`, the NanoTime tool reassigns parasitics from inside the subcircuit to the associated interface

pins. The NanoTime tool parses the full hierarchical names of pins in the parasitic netlist and determines if they are part of a timing model. For this example, parasitics that reference pin X1.Mn1.G are attached to interface pin E, as shown in Figure 2-9.

Figure 2-9 Timing Model Parasitics Reassigned to Interface Pin

Note that some parasitic capacitance might be double counted because the timing model pin capacitance, which is retained, might include parasitic capacitance.

Parasitics Associated With a Discarded Subcircuit

If you discard the subcircuit completely by using the `link_control_options` variable, the netlist database contains no trace of the discarded instance, as shown in Figure 2-10.

Figure 2-10 Discard instance flow

However, if you use the `link_discard_subckt_contents` variable to discard the subcircuit, the boundary pins of the subcircuit and the nets that connect the boundary pins to the rest of the design are retained, as shown in Figure 2-11.

Figure 2-11 Black box subcircuit

If you then set the `parasitics_enable_mapping_unresolved_pins` variable to `true`, the NanoTime tool reassigns parasitics from inside the subcircuit to the associated interface pins. For this example, parasitics that reference pin X1.Mn1.G are attached to interface pin E, as shown in Figure 2-12.

Figure 2-12 Subcircuit Parasitics Reassigned to Interface Pin

For this discarded subcircuit, no pin capacitance exists to model the effects of the load inside the subcircuit, which might be a source of timing inaccuracy.

To use the `parasitics_enable_mapping_unresolved_pins` variable to address unresolved pins in a discarded subcircuit, you must discard the subcircuit by using the `link_discard_subckt_contents` variable before linking the design.

Parasitics Annotated to Layout-Only Ports

When you use a timing model to represent a subcircuit in NanoTime analysis, you might choose to exclude the subcircuit from parasitic extraction because the timing model accounts for the parasitics of the subcircuit. The StarRC extraction tool refers to these subcircuits as skipped cells.

Layout versus schematic (LVS) tools, which provide input to extraction tools, can create connections that do not correspond to the schematic netlist, such as “layout-only” ports of the skipped cell. In the parasitic netlist, the StarRC tool assigns names to these ports according to the settings of the StarRC `XREF_LAYOUT_NET_PREFIX` command. The NanoTime tool reads the parasitic netlist and interprets object names based on the setting of the NanoTime `parasitics_xref_layout_net_prefix` variable. Both of these settings default to `ln_` as a prefix.

By default, if a parasitics file references layout-only ports, NanoTime issues a warning message and ignores the associated parasitics. As a result, the wire delay of the stage driving the layout-only net is incorrectly reported as zero.

To handle parasitics on nets that connect to layout-only ports, set the `parasitics_promote_black_box_nodes` variable to `true`. The NanoTime tool treats the layout-only port connection as a subnode of the back-annotated net, as shown in Figure 2-13, which allows back-annotation of the associated parasitics.

The layout-only ports must have names that use the prefix specified by the NanoTime `parasitics_xref_layout_net_prefix` variable.

Figure 2-13 Promotion of Layout-Only Port

Message Control for Parasitics in Black Boxes

The `parasitics_suppress_message_for_objects_in_black_box` variable controls messages about objects in black box circuits (timing models or discarded subcircuits). If you set this variable to `false`, the tool issues a PARS-121 information message when a pin in a parasitics file references a pin inside a black box object. The default is `true`, which ignores the parasitics and suppresses these messages.

Special Device Handling

Special device handling is available for diodes, resistors, and fingered transistors.

Diodes

For timing analysis, NanoTime converts diodes into capacitances and treats them as parasitic elements in an embedded RC flow. However, the diode pins are still available for back-annotation with parasitic data and the diode capacitance value can be overwritten.

Parasitic extraction tools might use different pin names in the parasitics files. You can specify diode pin names explicitly with the `link_diode_base_pin_name` (default `ANODE`) and `link_diode_emitter_pin_name` (default `CATHODE`) variables.

NanoTime does not support corner-specific parasitics. Because diodes are treated as capacitors, corner-specific diode back-annotation using different diode models is not allowed.

Resistors

Design resistors are usually symmetric devices. If the resistor terminals are swapped between the schematic and the layout, design functionality is not affected but parasitic back-annotation to those terminals can fail.

Use the `set_allow_pin_swap` command to specify two pin names that can be swapped for the purpose of parasitics back-annotation. This command requires two options: the `-cell` option to specify the cell or device that contains the pins, and the `-pins` option to specify the two pin names that can be swapped.

The `remove_allow_pin_swap` command removes a specification previously made with the `set_allow_pin_swap` command. The `report_allow_pin_swap` command lists all pin swapping specifications made with the `set_allow_pin_swap` command.

For example, the following commands allow the swapping of two sets of device terminals, then list the occurrences:

```
set_allow_pin_swap -cell rcdel -pins {t1 t2}
set_allow_pin_swap -cell abc -pins {term1 term2}
report_allow_pin_swap
```

If a subcircuit named by the `-cell` option contains only resistors and capacitors, you must also set the `parasitics_enable_annotation_to_embedded_rc` variable to `true`. Alternatively, you can convert the resistors between the swapped terminals to open circuits by using the `link_switch_resistors_as_switch_subckts` variable.

If the equivalent pins are not connected consistently between the schematic and layout views, the NanoTime tool issues a warning message. For example, a warning results if one pin of the device is swapped to the opposite net but the second pin is not.

Fingered Devices

If a parasitic data file contains fingered devices made by splitting transistors in the netlist, you might need to enable recognition of those transistors. Fingered instances are supported only if the fingered device or net names conform to the following syntax:

```
{schematic_device_name}{finger_char}{number}
{schematic_net_name}{finger_char}{number}
```

Examples of valid syntax are mn3@2 (for an instance) or net3@3 (for a net).

By default, the finger character can be either @ or #. You can change this character by setting the `parasitics_fingered_device_chars` variable. You must also set the

`parasitics_enable_drain_source_swap` variable to `true` to annotate parasitics on fingered devices.

For fingered devices, NanoTime automatically creates duplicate transistors and assigns the device data to them.

For fingered nets, NanoTime automatically creates a large resistor that connects from each schematic net to its corresponding fingered net (for example, from net3 to net3@2). As a result, both nets go into the same stage, but the large resistor prevents any inaccuracy that would result from tying the two nets together directly.

For example, Figure 2-14(a) shows the schematic view of a clocked inverter. During layout, the NMOS transistors become fingered devices, with two fingers each, as shown in part (b). Each finger has a unique name for assignment of the extracted parasitics. The NanoTime tool retains the transistor finger names. However, the fingered nets are forced to be part of the same net by connecting them with large-value resistors that do not affect the delay accuracy. This representation is shown in part (c) of the figure.

Figure 2-14 Representation of Fingered Objects

Equivalent Nets

Physical design flows require a layout versus schematic (LVS) tool to prepare the layout data and map the layout nets and devices to the schematic. The LVS tool provides input to a parasitic extraction tool, which reports the parasitic devices in a parasitic netlist. The NanoTime tool back-annotates the parasitics onto the schematic netlist.

During the LVS operation, the connectivity of devices to equivalent nets might be lost, which can result in devices being swapped between stacks. Swapped devices in the parasitics netlist cause back-annotation errors in the NanoTime tool because the device connectivity does not match the design netlist.

Some LVS tools provide a method of identifying nets as equivalent. This is most common for internal nodes of parallel transistor stacks. An example is the Mentor Graphics Calibre tool LVS SHORT EQUIVALENT NODES SPLIT option.

In the NanoTime tool, you can specify nets to be equivalent nets by using the `set_lvs_equivalent_nets` command. This command requires either the `-nets` option with a list of hierarchical net names (relative to the current instance) or the `-net_combined_report` option with the name of a reference file generated by the LVS tool.

When you use the `set_lvs_equivalent_nets` command, NanoTime inserts a 1-Gohm resistor between the equivalent nets. The tool treats the equivalent nets as a combined network, as shown in Figure 2-15, which allows successful parasitics annotation.

Coupling capacitance between equivalent nets is ignored. However, equivalent nets are expected to be equipotential and therefore the coupling capacitance has no effect.

Figure 2-15 LVS Equivalent Nets

If NanoTime determines that nets specified to be equivalent are not at the same potential, the tool issues an error message and ignores the specification. You can force the nets to be equivalent by using the `-force` option of the `set_lvs_equivalent_nets` command, in which case the tool issues a warning message instead.

The `report_lvs_equivalent_nets` command lists the equivalent nets.

Completing Partially Annotated Nets

The `-complete_with` option of the `read_parasitics`, `complete_net_parasitics`, and `check_design` commands completes the connection from a hierarchical pin to any additional pins at a lower level when the parasitic data is not completely specified for the net. This option allows you to use an incomplete parasitic data file.

During the completion process, NanoTime evaluates the nets connected to each hierarchical pin. If all the pins at the next-lower hierarchical level are leaf-level transistor pins, NanoTime completes the partial net parasitics by inserting capacitors and resistors between the hierarchical pin and the leaf-level transistor pins.

Valid arguments of the `-complete_with` option are `zero` or `rc`. To complete a partially annotated net, the `zero` setting inserts capacitors and resistors that have near-zero values. The `rc` setting inserts capacitors and resistors whose values are defined by the `parasitics_completion_capacitance` and `parasitics_completion_resistance` variables.

Hierarchical back-annotation flows might require NanoTime to read in multiple parasitics files. To complete the nets properly, use one (only one) of the following strategies:

- Execute all but one of the `read_parasitics` commands without any `-complete_with` options, then use the `-complete_with` option with the last `read_parasitics` command.
- Execute all of the `read_parasitics` commands without any `-complete_with` options, then use the `complete_net_parasitics` command with the `-complete_with` option.
- Execute all of the `read_parasitics` commands without any `-complete_with` options, then use the `check_design` command with the `-complete_with` option.

Preserving Cross-Coupling Capacitors

To perform analysis of crosstalk between nets with NanoTime signal integrity analysis, you must retain the cross-coupling capacitors in the parasitic data file. To do this, use the `read_parasitics` command with the `-keep_capacitive_coupling` option. Otherwise, by default, NanoTime converts cross-coupling capacitors to grounded capacitors for delay calculation purposes.

To convert cross-coupling capacitors to grounded capacitors, NanoTime splits each such capacitor to ground as shown in Figure 2-16. By default, the two new grounded capacitors each have the same value as the original cross-coupling capacitor.

Figure 2-16 Splitting a Cross-Coupling Capacitor to Ground

You can optionally define a reduction factor that is applied to each new capacitor, by using the `-coupling_reduction_factor` option of the `read_parasitics` command. For example, if you specify a reduction factor of 0.5, each new grounded capacitor has half the capacitance of the original cross-coupling capacitor. The default factor is 1.0.

Using Rail Net Contact Resistance in Delay Analysis

Contact resistance occurs between interconnect layers and transistor source or drain regions. For advanced technologies, contact resistance can be several hundred ohms, which might significantly affect path delays.

By default, NanoTime ignores back-annotated resistance and capacitance on power rails. Coupling capacitances between a signal net and a rail net are converted to capacitances between the signal net and ground.

You can take rail net parasitic resistance into account and analyze arbitrary resistor networks on rail nets. This feature requires a NanoTime Ultra license.

Set the `parasitics_enable_rail_net_resistance` variable to `true` to enable rail net analysis. The following configurations are supported:

- Mesh resistor networks, which are common in parasitics extracted from FinFET trench contacts
- Resistance annotation on virtual supplies
- Parasitics on the full rail network, including the metal interconnect layers

Analyzing parasitics from many layers might cause runtime to increase without a significant change in accuracy. For the greatest benefit at a reasonable cost, include the rail net parasitics only for the device layers.

To generate a report on the rail net resistances, use the `report_rail_net_resistance` command.

The following usage notes apply:

- Path delays might increase or decrease when contact resistance analysis is enabled.
- Power nets have many nodes. You might need to increase the values of the `parasitics_rejection_net_size` and `parasitics_warning_net_size` variables to allow parasitic annotation to all of the rail nodes. If the limit is reached, some nodes are annotated with lumped capacitances instead of detailed parasitics and the tool issues PARA-003, PARA-004, or PARA-006 warning messages.
- Rail contact resistance analysis is supported for rail nets that do not have a port statement (a *|P statement) in the parasitics file. You must set the `parasitics_accept_node_name_net_name` variable to `true` to enable this capability.
- NanoTime does not analyze IR drop. The tool considers the rail resistance only to calculate accurate single-stage delays.

Including the Effects of Common Trigger Skew

By default, NanoTime analysis ignores skew caused by parasitic RC delay between connected trigger devices. Figure 2-17 shows a circuit with three connected PMOS gates. The parasitic RC delay between these gates can be large enough to affect the accuracy of the timing analysis.

The `set_enable_input_spf_skew` command allows you to set a threshold in time units for evaluating the magnitude of this RC delay. If the value exceeds the threshold, the timing analysis takes the input skew into consideration.

Circuits with large RC skew might also require using nonlinear waveform analysis with a sufficient number of samples to represent the waveform accurately.

Figure 2-17 Input Skew

Variables for Parasitic Data Analysis

This section provides brief descriptions of some of the variables that affect the use of parasitics in NanoTime analysis. Not all variables related to parasitics are listed here.

For more information about specific variables, see the man pages.

The variables are described in the following sections:

- Variables That Affect Parasitic Netlist Interpretation
- Variables and Commands That Affect Annotation
- Variables That Affect Parasitics Reduction
- Variables and Commands That Affect Timing Analysis

Variables That Affect Parasitic Netlist Interpretation

Parasitic extraction tools, such as the Synopsys StarRC tool, provide options that can affect the format and content of parasitic netlists. You must verify that the extraction tool is set up correctly for your needs.

The following variables affect how NanoTime interprets parasitic netlist data:

- The `parasitics_accept_node_name_net_name` variable

  This variable enables recognition of RC subnodes that do not have subnode numbers. Some extraction tools write DSPF or SPEF files in which the RC node names do not have the ":#" extension. If this variable is set to `true`, NanoTime accepts the node name without an extension; otherwise, the tool issues an error message.

- The `parasitics_fingered_device_chars` variable

  This variable allows you to change the special character that denotes fingered devices.

- The `parasitics_enable_drain_source_swap` variable

  This variable enables recognition of swapped drain and source terminals to support recognition of fingered devices.

- The `parasitics_allow_mos_gate_delta_resistance` variable

  This variable allows NanoTime to read and use MOS gate parasitic resistances generated using a delta network model in the extraction tool. Delta network models sometimes produce nonphysical resistor representations such as negative resistances. By default, this variable is `true`; set it to `false` if you do not want to allow delta model resistance values.

Note:
The use of delta model resistors is not supported for dynamic clock simulation analysis.

- The `parasitics_xref_layout_net_prefix` variable

  This variable sets a prefix for parasitic net names that are not cross-referenced to a schematic net, thereby enabling recognition of these nets.

  For example, the StarRC tool `XREF_LAYOUT_NET_PREFIX` command specifies the prefix to use for layout instances that do not match schematic instances; its default is `ln_`. The NanoTime `parasitics_xref_layout_net_prefix` variable also defaults to `ln_`.

  NanoTime supports layout-only nets (sometimes called floating parasitic nets) in flat and back-annotated designs. Because these nets do not exist in the design netlist, NanoTime does not keep parasitics associated with them. However, all coupling capacitors to layout-only nets are grounded on the design nets.

  For example, the following lines might appear in a DSPF netlist generated by the StarRC tool using the default prefixes:

  ```
  *|NET ln_002 10FF
  *|I (Xld_XP1:G Xld_XP1 G|5.96e-18 0.964 15.576)
  Cg1 Xld_XP1:G 0 2.25309e-16

  *|NET ln_003 10FF
  *|I (Xld_XN1:G Xld_XN1 G|5.96e-18 0.964 15.576)
  Cg1 Xld_XN1:G 0 2.25309e-16
  ```

- The `parasitics_xref_layout_instance_prefix` variable

  This variable sets a prefix for parasitic instance names that are not cross-referenced to a schematic instance, thereby enabling recognition of these instances.

  For example, the StarRC `XREF_LAYOUT_INST_PREFIX` command specifies the prefix to use for layout instances that do not match schematic instances; its default is `ld_`. The NanoTime `parasitics_xref_layout_instance_prefix` variable defaults to an empty string; therefore, if you are using the StarRC extraction tool, you must set the NanoTime variable to match.

  For example, a DSPF or DPF file generated by the StarRC tool using the default prefixes might include lines such as the following:

  ```
  Xld_XP1 Xld_XP1:S Xld_XP1:G VDD pdev_a L=0.04U W=0.32U AD=...
  Xld_XN1 Xld_XN1:S Xld_XN1:G VSS ndev_a L=0.04U W=0.20U AD=...
  ```

- The `parasitics_port_delimiter` variable

  This variable specifies the delimiter character in DSPF parasitics files that separates fields in the names of pins expanded from a single port name.

Parasitics files might contain references to multiple pins connected to a single port. For example, if you use the StarRC extraction tool, the `SHORT_PINS: NO` command specifies that each pin has a separate netlist entry, as shown in the following DSPF file:

```
...
*|NET vdd 0PF
*|P (vdd_1 X 0 17 50)
*|P (vdd_2 X 0 31 79)
...
R1 vdd_1 vdd_2 0.01
R2 vdd_1 vdd 0.01
...
```

To read this file in a NanoTime script, set the `parasitics_port_delimiter` variable before executing the `read_parasitics` command, as follows:

```
set parasitics_port_delimiter "_"
```

In this example, port vdd has two pin definitions, vdd_1 and vdd_2. When the NanoTime tool parses the file, the tool uses the information about the delimiter character to match the pins to the original port vdd. The tool annotates parasitics on the pins to the corresponding port.

The delimiter must be a single character. Setting the variable to `" "` (with a space) or `""` (without a space) disables port and pin matching. The naming convention in the parasitics file must be the port name followed by the delimiter followed by an integer, as shown in this example:

```
port_nameDELIMITERinteger
```

An example is vdd_1 (port name vdd + delimiter _ + pin number 1).

Variables and Commands That Affect Annotation

The following variables and commands affect how NanoTime annotates parasitics onto the design:

- The `parasitics_suppress_dpf_inheritance` variable

  This variable specifies pre-layout netlist parameters to be ignored during macro-model device parameter annotation. For example, the multiplication factor should not be inherited from a netlist transistor to post-layout fingered devices. The default is an empty string.

- The `parasitics_ground_incomplete_coupling_cap` variable

  This variable enables recognition of incomplete coupling capacitances in DSPF files.

  By default, a coupling capacitance is defined between two nets whose parasitics are also defined in the DSPF file. If the file contains the definition of only one of the two nets, NanoTime ignores coupling capacitances between them. If you set this variable to `true`, NanoTime recognizes an incomplete coupling capacitance and converts it to a ground capacitance on the defined net.

- The `set_allow_pin_swap` command

  This command specifies two pin names in a specific cell that can be swapped for the purpose of facilitating back-annotation to those pins. For more information, see Special Device Handling.

- The `parasitics_enable_annotation_to_embedded_rc` variable

  This variable enables back-annotation of parasitics to netlist resistors and capacitors. Back-annotation does not replace the netlist RC values; it adds RC values to the existing RC pins. Back-annotation to both primitive devices (such as R1 A B 12.8) and subcircuit devices (such as XR1 A B rpoly l=3.57e-06 w=2e-06) is supported. The default is `false`.

  You must set the following RC terminal variables correctly for accurate back-annotation: `link_resistor_term1_pin_name`, `link_resistor_term2_pin_name`, `link_capacitor_term1_pin_name`, and `link_capacitor_term2_pin_name`.

- The `parasitics_rc_count_per_net_warning_threshold` variable

  This variable specifies a threshold for the total number of resistor and grounded capacitor elements per net, beyond which the tool issues a warning. The default is 5000.

- The `parasitics_warning_net_size` variable

  This variable specifies the number of annotated nodes on a net above which a warning message is issued, to guard against excessively long runtime. If the number of annotated nodes is larger than this value but less than the value of the `parasitics_rejection_net_size` variable, a PARA-003 warning message appears. NanoTime accepts and uses all of the parasitics. The default is 10000.

- The `parasitics_rejection_net_size` variable

  This variable specifies the maximum number of nodes that can be annotated on a net. If the value is exceeded, a PARA-004 warning message appears and the parasitic network is replaced by a lumped capacitance on that net. The default is 100000.

- The `parasitics_cap_warning_threshold` variable

  The tool issues a warning message if it reads a capacitance greater than or equal to this value from a parasitics file. The default is 0.0, which means that checking is disabled.

- The `parasitics_res_warning_threshold` variable

  The tool issues a warning message if it reads a resistance greater than or equal to this value from a parasitics file. The default is 0.0, which means that checking is disabled.

- The `parasitics_completion_r_count_per_net_warning_threshold` variable

  The tool issues a PARS-025 warning message if the number of resistor elements on a single net exceeds this value. The default is 50000. Nets with a vary large number of resistors might cause stack overflow during parasitics completion in some computing environments, as well as long runtime.

- The `parasitics_enable_mapping_unresolved_pins` variable

  This variable applies both to timing models and to SPICE subcircuits discarded with the `link_discard_subckt_contents` variable. During the parasitics reading operation, RC elements attached to interior pins that are connected to boundary pins are reassigned to the boundary pins.

- The `parasitics_suppress_message_for_objects_in_black_box` variable

  This variable controls messages about annotating parasitics to objects in black box circuits (timing models or discarded subcircuits). If you set this variable to `false`, the tool issues a PARS-121 information message when a pin in a parasitics file references a pin inside a black box object. The default is `true`, which ignores the parasitics and suppresses these messages.

- The `parasitics_device_name_mapping_method` variable

  This variable defines the priority order of methods for mapping device names in parasitics files to the logical netlist. For more information, see Name Mapping for Macro Models.

- The `parasitics_promote_black_box_nodes` variable

  This variable allows back-annotation on nets that connect to layout-only ports by treating the ports as subnodes of the back-annotated nets.

Variables That Affect Parasitics Reduction

The following variables affect parasitics reduction:

- The `parasitics_reduction_forced_min_delta_delay` variable

  If NanoTime cannot achieve parasitics reduction that meets the goal set by the `rc_reduction_max_net_delta_delay` variable, this variable sets the minimum allowed value for the change in delay that results from reducing (simplifying) the annotated parasitics on a net. The default is 0.000001 ns (0.001 ps).

- The `parasitics_reduction_no_touch` variable

  This variable disables all parasitics reduction. The default is `false`. Setting the variable to `true` disables all reduction. However, the increase in runtime might be significant.

- The `rc_reduction_min_net_delta_delay` variable

  This variable sets the minimum allowed value for the change in delay that results from reducing (simplifying) the annotated parasitics on a net. The default is 0.001 ns (1 ps). If the value is greater than the value of the `rc_reduction_max_net_delta_delay` variable, the `rc_reduction_min_net_delta_delay` variable is ignored.

- The `rc_reduction_max_net_delta_delay` variable

  This variable sets the maximum allowed value for the change in delay that results from reducing the annotated parasitics on a net. The default is 0.01 ns (10 ps). A very large value provides maximum reduction, possibly removing all resistors. A value of 0 prevents any reduction. However, the increase in runtime might be significant. For high-accuracy analysis, a setting of 0.005 ns is typically sufficient.

- The `rc_reduction_exclude_boundary_nets` variable

  This variable is `false` by default. Set it to `true` if you plan to create a timing model that preserves boundary parasitics by using the `-extract_boundary_parasitics` option of the `extract_model` command.

Variables and Commands That Affect Timing Analysis

The following variables and commands affect how parasitics are used in timing analysis:

- The `parasitics_enable_rail_net_resistance` variable

  This variable enables the use of rail net contact resistance. For more information see Using Rail Net Contact Resistance in Delay Analysis.

- The `set_enable_input_spf_skew` command

  This command includes parasitic elements between connected trigger devices. For more information, see Including the Effects of Common Trigger Skew.

3

Technology Data

Compact transistor models represent complex electrical behavior in a simplified way that enables fast and accurate timing analysis. Most device types, including multigate transistors or FinFETs, are supported without any changes to the NanoTime analysis flow. Some device types, such as SOI transistors, require additional settings to enable their use.

This chapter includes the following sections:

- SPICE Transistor Models
- Encrypted Device Models
- Custom Modeling Interfaces
- Device Models With Voltage-Controlled Voltage Sources
- Reading Device Parameter Data From Parasitics Files
- Modifying Transistor Drive and Transistor Parameters
- Reporting Technology Information
- Setting the Technologies for Analysis
- SOI Transistor Models

SPICE Transistor Models

SPICE transistor models provide a convenient and accurate way to specify technology data for a given design. NanoTime creates technology data lookup tables for each supported combination of model name, transistor netlist parameters (such as width and length), and selected operating conditions. Because there is an exact match between the generated lookup tables and the actual transistor device parameters used in the design, the tables can be used for timing analysis without extrapolation or interpolation.

You can read in a SPICE transistor model in either of the following ways:

- Implicit, by including a .lib statement in the SPICE netlist file
- Explicit, by using the `read_spice_model` command (preferred for 45 nm or smaller technologies)

In both cases, NanoTime first reads the SPICE model template into memory, storing the model information in a technology. Then, from the SPICE model template, the tool generates a set of lookup tables to match the transistors used in the design and stores the generated models in another technology. The reading and lookup table generation processes create two in-memory technologies. The table generation process occurs at the `check_design` command.

If the SPICE technology file contains macro transistor modeling or a wrapper, you must use the `read_spice_model` command to assign the technology SPICE file in addition to using the `register_netlist` command. After the `link_design` command, you must also use the `set_technology` command to associate the technology name with the transistors. An example of the flow is as follows:

```
read_spice_model -name SS tech_SS.sp
register_netlist -format spice tech_SS.sp
...
link_design
...
set_technology SS
```

Reading SPICE Models Implicitly

To read in SPICE transistor models implicitly, the SPICE netlist files (specified with the `register_netlist` command) must contain a .lib statement to direct NanoTime to the SPICE model file name, and the `link_enable_netlist_spice_model_linking` variable must be set to `true` (the default). Under these conditions, when you link the design with the `link_design` command, NanoTime reads the SPICE model template into an in-memory technology (an internal version of the design). At the `check_design` command, the tool also generates the transistor lookup tables and places them in another in-memory technology.

The following example flow reads a SPICE model implicitly:

```
nt_shell> set search_path {. ./../designs}
. ./../designs
nt_shell> set link_path {*}
*
nt_shell> register_netlist -format spice {alu.sp tech.sp}
1
nt_shell> link_design ALU
Compiling "/ ... /designs/alu.sp"
Compiling "/ ... /designs/tech.sp"
Compiling "../libs/tn90lp_v1.0.lib"
 ...
Linking design ALU...
Design 'ALU' was successfully linked...

nt_shell> check_design

Linking transistor models...
Information: 14232 out of 14232 transistors have
transistor models linked to them.  (TECH-003)
1
nt_shell> list_technology
 ...
Attributes (T):
    t - Tech data from techfile
    m - Tech data from SPICE model
    s - SPICE model

Name     Voltage    Temp Nmos    Pmos    T Source file
-------- ------- ------- ------- ------- - ---------------
typ        1.000  85.000 nch     pch     m <netlist>
typ        1.000  85.000 nch     pch     s <netlist>
1
```

In this example, the tech.sp file has the following lines to specify the process technology and the SPICE model library file used for analysis:

```
*nanosim tech="voltage 1.00"
*nanosim tech="body_bias 0 1.02 0.02"
*nanosim tech="vds 0 1.02 0.02"
*nanosim tech="vgs 0 1.02 0.02"
*nanosim tech="delta_vt -.35 .35"
* TEMP = 85
.TEMP 85
* PROC = SS
.lib './../libs/tn90lp_v1.0.lib' SS
```

The `check_design` command implicitly reads the SPICE transistor models into a technology, generates the transistor data lookup tables for the transistors used in the design, and puts that information into another technology.

The following part of the `check_design` message indicates that the lookup tables have been generated for all the transistors used in the design:

```
Information: 11032 out of 11032 transistors have all
transistor models linked to them.  (TECH-003)
```

The name of the technology created by implicit reading of SPICE models is determined by the `tech_netlist_spice_model_name` variable. The default for this variable causes the technology to be named “typ.”

The `list_technology` command lists all the technologies that are available to use. In the resulting report, the letter in the T column indicates the type of technology. The letter s indicates a SPICE model template file read directly from a SPICE file, and the letter m indicates a set of lookup table models generated from the template file.

Reading SPICE Models Explicitly

Instead of allowing NanoTime to generate the transistor lookup tables automatically, you can explicitly read in different SPICE model files (and possibly technology files as well) and specify which ones to use for a specific analysis run.

To read a SPICE model file explicitly, use the `read_spice_model` command. Specify the name of the SPICE model file and the associated technology name. The SPICE model file must be in plain ASCII format. NanoTime reads the file into a type “s” technology. The tool generates the transistor lookup tables when the `check_design` command is executed.

The following example shows a flow using explicit reading of a SPICE model:

```
nt_shell> set search_path {. ./../designs}
. ./../designs
nt_shell> set link_path {*}
*
nt_shell> register_netlist -format spice {alu.sp}
1
nt_shell> link_design ALU
Compiling "/ ... /designs/alu.sp"

Linking design ALU...
Design 'ALU' was successfully linked.
1
nt_shell> read_spice_model -name mytech tech.sp
Compiling "/ ... /designs/tech.sp"
Compiling "../libs/tn90lp_v1.0.lib"
 ...
nt_shell> check_design

Linking transistor models...
Information: 11032 out of 11032 transistors have all transistor
models linked to them.  (TECH-003)
1
```

```
nt_shell> list_technology
Technology Registry
 ...
Name         Voltage    Temp Nmos    Pmos    T Source file
------------ ------- ------- ------- ------- - ------------
mytech         1.000  85.000 nch     pch     m tech.sp
mytech         1.000  85.000 nch     pch     s tech.sp
```

The `read_spice_model` command lets you assign a technology name to the SPICE model file you are reading. The `set_technology` command lets you specify by name which technologies to use for analysis. You can specify up to four different technologies, for minimum data tracing, maximum data tracing, minimum clock tracing, and maximum clock tracing. NanoTime generates the transistor lookup models at the `check_design` command, filling the type m technology with the models needed to analyze the design.

The `check_design` command checks for missing or improperly configured transistor models. You can fix errors by reading additional models with the `read_spice_model` command followed by the `set_technology` command.

Use the `list_technology` and `report_technology` commands to get information about the transistor models and sizes that have been used and modeled. To remove SPICE models previously read in or to remove technology data lookup tables created from these SPICE models, use the `remove_technology` command.

See Also

- Setting the Technologies for Analysis

SPICE Model File Directives

The SPICE model file can contain directives to specify how NanoTime expands the model to generate the lookup tables. Directives in the SPICE file must follow one of these forms:

```
*nanosim tech= "delta_vt delta_vtn delta_vtp"
*nanosim tech= "model_alias name1 name2"
*nanosim tech= "voltage voltage1 voltage2 ..."
*nanosim tech= "vds 0 vds_end vds_step"
*nanosim tech= "vgs 0 vgs_end vgs_step"
*nanosim tech= "body_bias 0 vbs_end vbs_step"
```

For backward compatibility, the following forms of syntax in the SPICE model file are accepted by NanoTime as equivalent directives:

```
*nanosim tech= "directive"
*epic tech= "directive"
```

The directives are as follows:

- Delta Vt

  The `delta_vt` directive specifies how far into the subthreshold region to generate the lookup tables for NMOS and PMOS transistors. Use values closer to zero for faster simulation or farther from zero for more accurate simulation of subthreshold currents. For example, if the NMOS threshold voltage is 0.7 volts and `delta_vt` is -0.5 volts, the generated lookup tables start at 0.2 V. The tables produce a current of zero for gate voltages below 0.2 volts and accurate subthreshold currents between 0.2 and 0.7 V.

  The defaults are -0.5 V for NMOS and +0.5 V for PMOS, equivalent to the following:

  ```
  *nanosim tech= “delta_vt -0.5 0.5”
  ```

- Model alias

  The `model_alias` directive defines equivalent names for transistor models. This directive is useful when you have multiple technologies from different vendors that define their models with names that are different from those in your netlist. For example, to define the model names nch and nmos to be equivalent, use the following statement:

  ```
  *nanosim tech= “model_alias nch nmos”
  ```

- Design voltage

  The `voltage` directive specifies a list of supply voltages used in the design. NanoTime uses these voltages only to calculate transistor capacitance values for the creation of transistor model tables, not as supply voltages for timing analysis. If the `voltage` directive is not present, the tool uses the `tech_default_voltage` variable.

  For example, to define a single voltage supply of 1.2 volts, use the following statement:

  ```
  *nanosim tech= “voltage 1.2”
  ```

- Voltage sweeps

  The `vds`, `vgs`, and `body_bias` directives specify the end voltage and the voltage step size for the sweeps used to generate the lookup tables for the drain-to-source, gate-to-source, and body-to-source voltage tables. You can specify a finer resolution to minimize interpolation error, or a larger end voltage to accommodate a larger voltage swing, at the cost of more simulation time. For example, to set the VDS end voltage to 5.0 and the step size to 0.05 volts, use the following statement:

  ```
  *nanosim tech= “vds 0 5.0 0.05”
  ```

  In the absence of these directives, the end voltage is the supply voltage and the step size is 1/50 of the supply voltage.

Encrypted Device Models

NanoTime can read encrypted transistor models consisting of HSPICE wrapper models and model cards. However, the tool cannot read encrypted netlists or back-annotation files. A NanoTime Ultra license is required to read encrypted files.

NanoTime can read files that are encrypted using the `metaencrypt` utility, which is part of the HSPICE tool suite. All encryption types generated by the `metaencrypt` utility are supported; encryption by any other utility or tool is not supported.

To enable the use of encrypted models, set the `link_enable_metaencrypted_models` variable to `true` (the default is `false`) before running the `link_design` command.

In the presence of encrypted models, the following NanoTime commands are modified to prevent disclosing any encrypted data:

- `write_spice`
- `extract_model -hspice_timing`
- `extract_model -library_elements {nldm ccs_timing ccs_noise}`
- `report_technology`
- `report_attribute` (when applied to transistor parameters)
- `list_technology`

It might be difficult to debug an analysis containing an encrypted device model due to the modified output of these NanoTime commands. A good practice is to validate a model before using it in encrypted form.

The following requirements apply to the use of encrypted models:

- Encrypted resistors, capacitors, and diodes are not supported.
- A protected model file cannot contain unprotected model files. If any part of the wrapper models or model cards is encrypted, all models are considered to be encrypted.
- Encrypted model cards or wrapper models cannot appear in the same file as a pseudo-top cell (an unnamed .subckt block used in place of a top cell to contain the root portions of the design when a top cell does not exist).
- Multicorner flows are not supported.
- Elaborations of encrypted parameters are not protected (for example, using encrypted .param statements with unprotected .model cards).

Custom Modeling Interfaces

CMI models

CMI models support TSMC extensions for standard compact models for process technologies of 45 nm and below. To use CMI models, make the following changes in the netlist or model file:

- Add the `cmiflag` global option to load the dynamically linked custom CMI library (`libCMImodel`).
- Set `.option cmipath =”$your_cmi_lib_path”` for the path of the CMI .so library.

TMI models

TMI models support TSMC extensions for standard compact model for process technologies of 45 nm and below. TSMC provides a single compiled TMI library (.so) to which simulation tools can link during runtime. The TMI setup options are included in SPICE model libraries. To use TMI models, make the following changes in the netlist or model file:

- Set `.option tmiflag = 1` (or set `.option tmiflag` in the model file).
- Set `.option tmipath =”$your_tmi_lib_path”` for the path of the TMI .so library.
- Set `.option macmod= <1|2|3>` for the macro-model design for backward compatibility.

Table 3-1 lists the options for use with TMI models.

Table 3-1 TMI Model Options

Mapping	Option line	Description
M to X mapping	`.option macmod = 1`	Netlist instance name starts with M, but the TMI model of the same device name refers to a macro model, not a transistor model.
X to M mapping	`.option macmod = 2`	Netlist instance name starts with X, but the TMI model of the same device name refers to an actual transistor model.
X to M and M to X mapping	`.option macmod = 3`	Netlist has a mixed case of values 1 and 2.

Automatic Platform Selection

Automatic platform selection can help you when you might not have information about the platform that NanoTime is running on. With this feature, an automatic custom library platform can be selected and linked to NanoTime at each NanoTime run. The tool automatically completes the path name according to the platform you are using.

Modify the model file as follows:

- For TMI models, add the following statement to your model file. If multiple statements are specified, the last one is applied:

  `.option tmipath='lib_path'`

- For CMI models, add the following to your model file. If multiple statements are specified, the last one is applied:

  `.option cmipath='lib_path'`

Set the path variable to point to the TMI (or CMI) .so library path. NanoTime searches for the `libTMImodel` shared library (or `libCMImodel` library) in the `tmipath/platform` directories (or `cmipath/platform` directories). Use the following guidelines:

- The specified *lib_path* can have only one path.
- If *lib_path* is an absolute path, NanoTime links to the library from *lib_path/platform.*
- If *lib_path* is a relative path, NanoTime links to the library from *current_file_path/lib_path/platform*, where *current_file_path* is the path to the model file containing `.option tmipath='lib_path'` (or `.option cmipath='lib_path'`).

For TMI models, follow these steps to enable automatic platform selection:

- Set `.option tmipath` to point to the model file. For example,

  ```
  .option tmipath= '/abc/nt/tmi/lib'
  ```

- Place all libTMImodel.so files according to the platform. For example:

  ```
  /abc/nt/tmi/lib/SUN/libTMImodel.so
  /abc/nt/tmi/lib/SUN_64/libTMImodel.so
  /abc/nt/tmi/lib/RH/libTMImodel.so
  /abc/nt/tmi/lib/RH_64/libTMImodel.so
  ```

For CMI models, follow the same steps, but use the CMI path and model names.

Flexible Instance Parameter Naming Capability

NanoTime supports flexible instance parameter naming for transistors. In addition to the standard transistor parameters, the tool allows up to 50 flexible instance parameters, which are usually defined by the foundry. For example:

```
Mnxx d g s b <mname>
+ W=<> L=<> NF=<> AS=<> AD=<> PS=<> PD=<> RDC=<> RSC=<>
(Flexible instance parameters)
```

To enable flexible instance parameter support, make the following change in the netlist or model files:

```
.option process_dev_params=1
```

Dynamic Bin Selection

The model bin selection can be adjusted by the functions defined in shared object libraries. To enable the dynamic bin selection feature for TMI models, make the following change in the netlist or model file:

```
.option tmiusrflag=1
```

For CMI models, use the following statement instead:

```
.option cmiusrflag=1
```

Device Models With Voltage-Controlled Voltage Sources

Device libraries might include a voltage-controlled voltage source at a transistor gate. As shown in Figure 3-1, a voltage source can be positioned at the transistor gate to model the drain-induced barrier lowering variation.

Figure 3-1 Transistor Macro Model With Voltage-Controlled Voltage Source

The voltage-controlled voltage source element is represented by the formula:

$$V_{GG'} = \mu |V_{DS}|$$

A transistor attribute, the `transistor_bvs_gain` attribute, represents the linear voltage source gain factor μ of the transistor. You can query the value of the gain factor of a transistor cell instance (M1 for example) by using the following command:

```
get_attribute -class cell [get_cells M1] transistor_bvs_gain
```

To change the gain value, use the `-bvs_gain` option with the `set_transistor_parameter` command. Gain values are typically much less than 1.0.

Designs that use voltage-controlled voltage sources cannot use dynamic clock simulation or dynamic delay simulation.

Reading Device Parameter Data From Parasitics Files

You might have device parameter data in a .dpf or DSPF file from an external tool such as the Synopsys StarRC tool. The `read_device_parameters` command reads device parameter data from files in this format and annotates the information on transistors in the current design, thereby allowing more accurate modeling of transistor behavior. For example, to read in and apply the device parameter data in the file adder.dpf, use the following command:

```
nt_shell> read_device_parameters adder.dpf
```

To report the parameters, use the `report_annotated_device_parameters` command. To remove them, use the `remove_annotated_device_parameters` command.

Modifying Transistor Drive and Transistor Parameters

You can modify the drive strength and transistor parameters of selected transistors without creating any new models by using the `set_transistor_drive_factor` and `set_transistor_parameter` commands.

The `set_transistor_drive_factor` command changes the drive strength of transistors by a specified factor. For example, a factor of 2.0 doubles the drive strength of the specified transistors. You can use the `-min`, `-max`, `-rise`, and `-fall` options to restrict the scope of the change.

Similarly, the `set_transistor_parameter` command sets one or more parameters for specified transistors in the design, overriding the parameters in the technology. You can modify a number of model parameters; for a complete list of the options, see the man page for the command.

You can also control the usage of length of diffusion and well proximity effect information by setting the `mos_enable_lod` and `mos_enable_wpe` variables.

Changing the drive strength or transistor parameters changes the design, which takes you back to the analysis stage just before the `check_design` command. You must execute the `check_design` and `trace_paths` commands to get analysis results, even if those commands have been run before.

Attributes of the `cell` and `parasitic_device` object classes contain transistor parameters. To view the parameter values for a transistor, use the `report_attribute` or `get_attribute` command. For example,

```
nt_shell> report_attribute -class cell \
          -application [get_cells Xreg51.X3.Mp0]
```

Reporting Technology Information

For information about the technology data loaded into memory, including both SPICE transistor models and models from technology files, use the `list_technology` and `report_technology` commands.

The `list_technology` command lists the technology source files that have been loaded and are available for linking in NanoTime. Here is an example of a technology listing:

```
nt_shell> list_technology
Technology Registry

Attributes (T):
    t - Tech data from techfile
    m - Tech data from SPICE model
    s - SPICE model

Name         Voltage    Temp Nmos    Pmos    T Source file
------------ ------- ------- ------- ------- - ------------
typ            1.000  85.000 nch     pch     m <netlist>
typ            1.000  85.000 nch     pch     s <netlist>
```

For each technology, the report shows the technology name, supply voltage, temperature, NMOS transistor model name, PMOS transistor model name, technology attributes, and model source file. The “Attributes” column uses the letter code `t`, `m`, or `s` to indicate the type of technology:

- The letter `t` indicates a set of lookup models taken from a technology file (read with the `read_techfile` command). This file type is now obsolete.
- The letter `m` indicates a set of lookup models generated by NanoTime from a SPICE model template. The models are generated with the `check_design` command.
- The letter `s` indicates a SPICE model template read explicitly by the `read_spice_model` command or implicitly by the `check_design` command.

The “Source file” column indicates the file from which the technology was obtained, which could be a technology file or a SPICE model file. The notation `<netlist>` indicates that the models were read in implicitly during linking by the `link_design` command.

The `report_technology` command reports the details of model entries in the technologies. Here is an example of a technology report:

```
nt_shell> report_technology
Model Tech.
Name  Name  Volt  Temp   Width   Length   Delvto  Mulu0   Total
----- ----- ----- -----  ------  -------  ------  ------  -----
nch   typ   1.20  85.00  1.444u  0.130um  0.000   1.000   2
pch   typ   1.20  85.00  1.444u  0.130um  0.000   1.000   4
```

For each technology, the report shows the following information: the transistor model name, technology name, voltage, temperature, channel width, channel length, delta threshold voltage offset (an adjustment to the voltage threshold), electron mobility adjustment multiplier, and the total number of transistor models.

The columns Width, Length, Delvto, and Mulu0 show the number of index values supported by the technology, or the value itself if there is only a single value in the technology. The total number of transistor models is the number of supported combinations of Width, Length, Delvto, and Mulu0.

By default, the `report_technology` command reports all transistor models. To restrict the scope of the report to a specific model name, technology name, voltage, or temperature, use the `-model`, `-name`, `-voltage`, or `-temperature` option. To show the range of index values (minimum and maximum) for Width, Length, Delvto, and Mulu0, use the `-ranges` option. Use the `-index` option to list all index values.

The following command reports the technology model called pch and shows the width and length index values:

```
nt_shell> report_technology -model pch -name typ -index

      Tech.
Name  Name      Volt  Temp   Width Length Delvto  Mulu0  Total
----- --------- ----- ------ ----- ------ ------- ------ ------
pch   typ       1.70  85.00  21    2      0.000   1.000  42

Length index list:
     0.225um    0.150um
Width index list:
     300.000um  200.000um  100.000um  50.000um  40.000um
     30.000um   20.000um   15.000um   10.000um  9.000um
     8.000um    7.000um    6.000um    5.000um   4.000um
     3.000um    2.000um    1.000um    0.750um   0.500um
     0.315um
```

To remove technology data that has been read in but is no longer needed, use the `remove_technology` command.

Setting the Technologies for Analysis

A NanoTime technology contains transistor models that describe the electrical parameters of transistors under specified operating conditions. You create a technology by reading a SPICE model directly with the `read_spice_model` command or implicitly with a .lib statement in a SPICE netlist file. You can report the names and characteristics of the technologies that are currently available by using the `list_technology` and `report_technology` commands.

The operating conditions (voltage, temperature, and process parameters) are determined by the SPICE models or technology files specified by the `set_technology` command.

You can specify up to four separate technologies (corners), as follows:

- `min` – for minimum-delay (shortest) data paths
- `max` – for maximum-delay (longest) data paths
- `min_clock` – for minimum-delay (shortest) clock paths
- `max_clock` – for maximum-delay (longest) clock paths

You must specify the technologies in one of the following three combinations:

- A single technology. NanoTime uses the same technology for all analysis. For example,

  ```
  nt_shell> set_technology typ
  ```

- Two technologies, using the `-min` and `-max` options. NanoTime uses the worst-case technology for each path segment in a timing check. This is the most conservative kind of checking. For example:

  ```
  nt_shell> set_technology -min MINp1 -max MAXp1
  ```

  In this example, the two technologies are assigned to the four path types as follows:

  ```
  min technology = MINp1
  max technology = MAXp1
  min_clock technology = MINp1
  max_clock technology = MAXp1
  ```

- Four technologies, one for each of the `-min`, `-max`, `-min_clock`, and `-max_clock` options. For example:

  ```
  nt_shell> set_technology -min MINp1 -max MAXp1 \
            -min_clock MAXp1 -max_clock MINp1
  ```

In this example, the resulting settings are as follows:

```
min technology = MINp1
max technology = MAXp1
min_clock technology = MAXp1
max_clock technology = MINp1
```

The `set_technology` command must be executed before reading in a device parameter file (with the `read_parasitics` or `read_device_parameters` commands) because the file evaluation that occurs at the `check_design` command requires the model libraries.

The following example illustrates the general DSPF file flow:

```
# read in one or more technology corners
read_spice_model -name corner_name corner_spice_file
...
link_design top_level_name
...
# map the technology to the design
set_technology ...
...
# read parasitics from the DSPF file
read_parasitics ...
...
check_design
```

The following example illustrates the general SPEF file flow:

```
# read in one or more technology corners
read_spice_model -name corner_name corner_spice_file
...
link_design top_level_name
...
# map the technology to the design
set_technology ...
...
# read device parasitics from the DPF file
read_device_parameters ...
# read parasitics from the SPEF file
read_parasitics ...
...
check_design
```

Using Macro Models

Macro models contain a wrapper subcircuit in the library. These models require special handling, which is enabled when the `link_enable_corner_specific_wrapper_subckts` variable is set to `true`. NanoTime usually detects the absence of macro models and sets the variable to `false`. If you have added a wrapper to a non-macro model, it might be necessary to manually set the variable to `false`.

When using macro models with multicorner analysis, in addition to specifying the SPICE models with the `read_spice_model` command, you must specify a model as part of the netlist as follows:

```
read_spice_model -name tt tech_tt.sp
read_spice_model -name tt1 tech_tt1_new.sp
register_netlist -format spice tech_tt.sp
```

The model specified in the `register_netlist` command does not need to be one of the corners used in the analysis. The specified model makes it possible to find the transistor cells. If NanoTime cannot correctly locate the transistors in the netlist, it might be necessary to identify the models with the `sim_transistor_wrapper_subckts` variable.

This variable should be set to all possible macro model names in your model files. For example:

```
set sim_transistor_wrapper_subckts "pmos nmos pmos_hv nmos_h"
```

Operations at the check_design Command

When you execute the `check_design` command, NanoTime links each transistor in the design to a transistor model and reports the number of linked transistors, as shown here:

```
Linking transistor models...
Information: 11032 out of 11032 transistors have all
transistor models linked to them.  (TECH-003)
```

During the linking process, NanoTime generates lookup models from SPICE model templates as needed to fill the type “m” technologies reported by the `list_technology` command.

If not all the transistors in the design are linked to models, you see a warning message similar to the following:

```
Linking transistor models...
Warning: 478 out of 11032 transistors do not have all
transistor models linked to them.  (TECH-002)
1
```

This warning might occur when you are using transistor models derived from technology files and no models are available to match some of the transistors in the design.

To increase the usage of models derived from technology files, increase the allowable difference between transistor parameter values in the design and the parameter values of the models. To do this, set the `tech_match_length_pct`, `tech_match_width_pct`, and `tech_match_param_pct` variables to nonzero values.

Before you can analyze the design, each transistor must have a linked model. The `check_design` command verifies that this requirement is satisfied.

Reporting the Available Technologies

To report the technologies that have been set for a design, use the `report_design` command. An example of the report is as follows:

```
nt_shell> report_design
 ...
  Design Attribute                Value
- ------------------------------- -------------
Units:
  Time Unit                       1 ns
  Capacitance Unit                1 pF
  Voltage Unit                    1 V
  Resistance Unit                 1 kohm
  Current Unit                    1 mA
  Width/Length Unit               1 um
  Area Unit                       1 um square

Technology:
  max_technology                  typ
  min_technology                  typ
  max_clock_technology            typ
  min_clock_technology            typ
...
```

SOI Transistor Models

NanoTime offers the option to analyze the floating-body effects of partially depleted silicon-on-insulator (SOI) technologies. You must have a NanoTime Ultra license to analyze SOI designs.

This chapter contains the following sections:

- Partially Depleted SOI Technologies
- SOI Analysis in the NanoTime Flow
- SOI Analysis Commands

Partially Depleted SOI Technologies

In a silicon-on-insulator (SOI) fabrication technology, transistors are built in a silicon layer resting on an insulating layer of silicon dioxide. The insulating layer can be created by implanting oxygen into a plain silicon wafer at a fixed depth below the surface, then heating the wafer to oxidize the silicon, thereby creating a uniform buried layer of oxide. The insulating layer increases device performance by reducing junction capacitance and by allowing lower threshold voltages to be used in the technology.

Figure 3-2 shows a cross section view of an N-channel MOS transistor made by conventional means and the same type of transistor fabricated with a silicon-on-insulator process technology.

Figure 3-2 NMOS Transistor in Conventional and SOI Technologies

In a conventional process technology, the P-type body of the NMOS transistor is held at the ground voltage by means of a metal contact to the substrate. A PMOS transistor in this technology (not shown in the diagram) is fabricated in an N-well, with the transistor body held at the Vdd supply voltage by means of a metal contact to the N-well.

In a silicon-on-insulator process technology, the source, body, and drain regions of transistors are insulated from the bulk substrate. The parasitic junction capacitance is reduced because source and drain are insulated from the substrate. The transistor area is also reduced because there is no need for metal contacts to N-wells used for making PMOS transistors. The body of each transistor is typically left unconnected, but it can be tied to the rail voltage or to the source node of the transistor to eliminate the floating-body effect. Body contacts are used only where needed because they increase the layout area and decrease performance.

In an NMOS transistor, applying a positive voltage to the gate depletes the body of P-type carriers and induces an N-type inversion channel on the surface of the body. If the insulated layer of silicon is made very thin, the inversion layer fills the full depth of the body, and the whole body behaves as an N-type conducting channel. A technology designed to operate this way is called a “fully depleted” SOI technology. The thin body avoids a floating voltage, but it has lower conductivity and longer switching times.

On the other hand, if the insulated layer of silicon is made thicker, the inversion region does not extend the full depth of the body. A technology designed to operate this way is called a “partially depleted” SOI technology. The undepleted portion of the body is not connected to anything, so its voltage level is unpredictable. The exact voltage depends on the history of source, gate, and drain voltages leading up to the current time (the “history effect”). However, the voltage can be expected to fall within a known range.

The body voltage affects the conduction of the channel and therefore the switching speed and parasitic capacitance of the circuit. In an NMOS transistor, a lower initial body voltage results in a thinner inversion layer, lower conductivity, and slower switching. Conversely, a higher initial body voltage results in faster switching. In a PMOS transistor, the opposite is true; a lower initial body voltage results in faster switching.

SOI Analysis in the NanoTime Flow

In a partially depleted SOI technology, NanoTime takes into account the possible range of body voltages by considering the worst condition for a given timing constraint. For example, for an NMOS transistor in a data path, the tool uses the lowest body voltage (slowest switching) to calculate the setup delay, or the highest body voltage (fastest switching) to calculate the hold delay.

You must have a NanoTime Ultra license to use the SOI analysis feature. To enable SOI analysis, set the `soi_enable_analysis` variable to `true`. To specify the range of body voltages resulting from the history effect, use the `set_soi_parameters` command for each transistor model. To specify the presence of any body contacts used in the design, use the `set_soi_transistor_type` command.

Body Voltage Range

NanoTime uses the partially depleted SOI transistor model provided in the SPICE model to calculate delays. The tool considers the voltage on the body node of the transistor. For a given transition, it selects either the minimum or maximum of the body voltage range as the initial body voltage, depending on which is worse for the timing constraint.

You can specify the body voltage ranges for each transistor model registered in the logic library, for each valid combination of gate, source, and drain states for the model. The voltage range settings apply to all instances of the same transistor model. There are separate settings for NMOS and PMOS type models.

For easy initial setup, NanoTime offers a built-in default mode to automatically select conservative voltage bounds. Using this method, you do not need to set any specific voltage values to get reasonable results. The “conservative” method gives worst-case results that are pessimistic but sure to detect timing violations. You can also automatically select “average” or typical bounds that are reasonable but possibly optimistic, or “scaled” bounds that are scaled by a specified factor between the “conservative” and “average” bounds.

For the best accuracy with the least pessimism, characterize the transistors using external tools, and then explicitly specify the body voltage ranges for each transistor model and each gate-source-drain state.

Usage Restrictions

The following restrictions apply to SOI analysis:

- Body voltage bounds can be defined only on a per-model (not per-instance) basis. If different instances require different voltage boundaries, it is necessary to create a separate SPICE model for each set of boundary values.
- The design can contain only SOI transistors. Conventional bulk-CMOS and SOI and transistors cannot be mixed within a design.
- Information about the transistor bulk pin voltages is not available from the design attributes, and you cannot back-annotate attributes on the bulk pins. You can, however, get information about the body voltage range settings with the `report_technology` command. You can also get detailed modeling information for a circuit segment by using the `write_spice` command.
- Only SPICE models provided directly to NanoTime can be used for SOI analysis.

SOI Analysis Commands

The `set_soi_parameters` command specifies the range of body voltages that are can result from the memory effect for each transistor model in the technology. For transistors in the design that have the body connected to the transistor source pin or to the rail voltage, the `set_soi_transistor_type` command specifies the type of body contact.

The set_soi_parameters Command

For each SOI transistor switching event, NanoTime takes into account the body voltage of each transistor, considering the worst possible starting voltage of the body for a given timing constraint. The starting voltage has a possible range due to the unknown switching history of the transistor. The `set_soi_parameters` command specifies the upper and lower bounds on the body voltage.

You can let the tool estimate the body voltage ranges by using the `-conservative`, `-average`, or `-scaled` options, or you can specify the possible ranges explicitly by using the `-min`, `-max`, and `-rail_reference` options.

The following example invokes the conservative method for estimating body voltages for the “nch” transistor model, for gate-source-drain states 0r0, 000, and 010.

```
nt_shell> set_soi_parameters -model nch -conservative \
          -states {0r0 000 010}
```

The following example specifies that the initial body voltage bounds as 0.2 to 0.8 volts for the “nch” transistor model, for gate-source-drain states 0r0, 000, and 010. The rail reference voltage is 1.8 volts.

```
nt_shell> set_soi_parameters -model nch \
          -min 0.2 -max 0.8 -rail_reference 1.8 \
          -states {0r0 000 010}
```

In the three-character gate-source-drain codes, 0 means the “off” voltage, “1” means the “on” voltage, and “r” means the rail-tied voltage. For an NMOS transistor, the “on” voltage is high and the “off” voltage is low.

Model Name

The `-model` *`name`* option specifies the name of the SPICE transistor model affected by the `-min`, `-max`, and `-rail_reference` values. If this option is not used, the same settings apply to all transistor models.

NanoTime supports BSIM4SOI and BSIM3SOI models.

Automatic Body Voltage Ranges

The `-conservative`, `-average`, and `-scaled` options invoke automatic estimation of the body voltage bounds, making it possible to get reasonable results without specifying body voltage values. The chosen method applies to all transistor models.

The `-conservative` option uses worst-case estimates for the body voltage ranges for each gate-source-drain state. The timing results are likely to be pessimistic, but detection of timing violations due to the floating body effect is ensured.

The `-average` option uses a typical estimated body voltage for each gate-source-drain state, using the same value for both the minimum and maximum. The timing results are likely to be optimistic.

The `-scaled` option uses body voltage values that are scaled between the conservative and average methods by a specified factor between 0 and 100 percent. A scaling factor of 50 results in values halfway between the conservative and average methods. A scaling factor of 0 is the same as the conservative method and a scaling factor of 100 is the same as the average method.

A transistor that changes state infrequently is likely to have a more extreme body voltage, so using the conservative method or a scaling factor closer to 0 percent is appropriate in this case. A transistor that changes state frequently is likely to have a moderate, intermediate body voltage, so using an intermediate scaling factor such as 50 to 70 percent is appropriate in this case.

To get a report on the estimated body voltage bounds, use the `report_technology -soi_parameters` command.

Specified Body Voltage Ranges

The `-min` and `-max` options specify the minimum and maximum expected body voltages, respectively, for the specified transistor model and gate-source-drain states. You can use an external tool such as HSPICE to determine the expected ranges for a technology.

If you use the `-min` and `-max` options, you must also use the `-rail_reference` option to specify the nominal rail voltage for the transistor. If multiple supply voltages exist in the design, the minimum and maximum boundary values are adjusted according to the difference between the applied voltage and the specified rail reference voltage.

To get a report on the body voltage bounds that have been set, use the `report_technology -soi_parameters` command.

Gate-Source-Drain States

The `-states` option specifies the initial gate, source, and drain states. Specify a set of three characters for the gate, source, and drain, respectively: 0 for off, 1 for on, or r to indicate tied to the rail voltage. If the `-states` option is not used, the command applies to all states.

For an NMOS transistor, 0 means a low voltage, 1 means a high voltage, and r means the source tied to ground. For a PMOS transistor, 0 means a high voltage, 1 means a low voltage, and r means the source tied to Vdd.

Table 3-2 lists the gate-source-drain states that can be specified and the corresponding voltages on NMOS and PMOS transistor pins. The states 101 and 110 are not allowed because they represent the transistor in an unstable state, with the transistor turned on and the source and drain at opposite voltages.

Table 3-2 Gate-Source-Drain States

Transistor state	NMOS gate-source-drain	PMOS gate-source-drain
000	low-low-low	high-high-high
001	low-low-high	high-high-low
010	low-high-low	high-low-high
011	low-high-high	high-low-low
100	high-low-low	low-high-high
101 - not used	high-low-high (unstable)	low-high-low (unstable)
110 - not used	high-high-low (unstable)	low-low-high (unstable)
111	high-high-high	low-low-low
0r0	low-gnd-low	high-Vdd-high
0r1	low-gnd-high	high-Vdd-low
1r0	high-gnd-low	low-Vdd-high

The tied-to-rail state is different from 0 or 1 because the source terminal is held strongly at the rail voltage, causing less deviation of the body voltage from the rail voltage. This results in a tighter maximum-conduction bound than using 0 or 1.

The set_soi_transistor_type Command

Some transistors in the design might have the body connected to a constant voltage or to the source terminal of the transistor. In these cases, the body is not floating and its voltage does not span the full possible range of floating-body transistors. To get a more accurate

(less pessimistic) analysis of these transistors, you should specify their body connection type with the `set_soi_transistor_type` command, as in the following example:

```
nt_shell> set_soi_transistor_type -body_contact 1.2 x00.mp1
```

The argument is a list of transistor instances affected by the command. If the body has an electrical contact to the constant voltage, use the `-body_contact` option and specify the voltage. If the body is connected to the source terminal of the transistor, use the `-body2source` option.

Reporting SOI Parameter Settings

To get a report on SOI parameter settings, use the `report_technology` command with the `-soi_parameters` option. For example,

```
nt_shell> report_technology -soi_parameters

ModelName State UpperBound LowerBound RailReference ParameterSource
-----------------------------------------------------------------
 tpssfh   0r0        2.8        2.1           2.5    conservative
 tpssfh   0r1        2.5        2.1           2.5    conservative
 tpssfh   1r0        2.5        1.9           2.5    conservative
 tpssfh   000        2.8        1.6           2.5    conservative
 ...
```

The command reports the body voltage range and rail reference voltage for each state of each transistor model. The parameter source is reported as `conservative`, `average`, `scaled`, or `user`, depending on how the SOI parameters were set.

To get detailed modeling information for a circuit segment, use the `write_spice` command with the `-soi_comments` option and examine the generated SPICE file.

4

Topology Operations

The topology recognition phase includes the definition of clocks, clock networks, and other topology structures. NanoTime provides many options for defining and working with topology objects.

Topology operations are described in the following sections:

- Topology Recognition Overview
- Recognizing Channel-Connected Blocks
- Recognizing Storage Nodes
- Setting and Reporting Transistor Direction
- Debugging the Data Inputs of Sequential Elements
- Manually Marking and Erasing Structures
- Special Purpose Marking Commands
- Reporting Topology Information
- Searching by Name or Pattern Matching
- Marking Topologies With Automatic Pattern Matching
- The Topology Database

Topology Recognition Overview

The topology recognition phase takes place between the `link_design` command and the `check_topology` command. NanoTime needs information about topology to perform the appropriate timing checks at the correct places in the design. After the `check_topology` command, the design cannot be modified.

Achieving complete topology recognition often requires several steps. For example, clock propagation must occur before sequential structures such as latches can be recognized.

NanoTime can recognize the following structure types:

```
clock_gate            flip_flop             register
cross_coupled         mux                   tgate
cross_coupled_pmos    precharge             turnoff
feedback              pulldown              weak_pullup
inverter              pullup                xor
latch                 ram                   differential_synchronizer
```

Use the `-structure_types` option of the `match_topology` command to specify a list of topologies that NanoTime looks for and marks in the design. For example, the following command looks for and marks only the feedback and multiplexer structures in the design:

```
nt_shell> match_topology -structure_types {feedback mux}
```

If the `-structure_types` option is not used, NanoTime looks for the structures listed in the `topo_auto_search_class` variable. By default, the variable is set to the following string:

```
mux clock_gate turnoff xor cross_coupled nand nor latch precharge
ram feedback weak_pullup differential_synchronizer
```

In some cases, NanoTime marks additional structures that are needed to identify the specified structures. For example, if you use the `-structure_types precharge` option, structures such as `clock_gate`, `feedback`, and `latch` are also marked because they are needed to identify precharge structures.

The Topology Recognition Flow

For a complex design, you might need to use multiple steps to modify the clocks or clock networks, erase or define topologies, and set logic constraints to achieve accurate topology recognition. Figure 4-1 illustrates a topology recognition flow that allows complex topology definition strategies.

Figure 4-1 Topology Recognition Flow

You can manually mark topologies either before or after the `match_topology` command. For structures such as latches that require a clock signal, you must mark those structures after the `match_topology` command.

By default, NanoTime propagates the clock network and recognizes topologies one time, at the first occurrence of either the `match_topology` or `check_topology` command.

If you want to propagate the clock network and perform automatic topology recognition more than one time, you must perform these steps:

1. Set the `topo_match_topology_reset_clock_and_topology` variable to `true`.
2. Specify additional topology marking or erasing commands.
3. Execute the `match_topology` command with the `-force` option.

The `report_topology`, `get_topology`, `report_transistor_direction` commands and other reporting commands provide information about structures in the design. In addition, NanoTime sets the value of many attributes associated with the design objects. The reports and attributes can help you to understand the topologies.

See Also

- Chapter 5, “Clocks and Clock Networks”
- Chapter 6, “Recognizable Topologies”
- Chapter 7, “Differential Circuits”

Recognizing Channel-Connected Blocks

The topology recognition process begins with subdividing the circuit into channel-connected blocks (CCBs), also known as channel-connected regions (CCRs).

A channel-connected block is a small circuit with a single trigger device and fixed side inputs to facilitate path tracing form the trigger input to the circuit output. The CCB is a group of devices that can possibly be connected by current flow, even if all of the transistors are not turned on at the same time. The source and drain of a transistor are always part of the same channel-connected block. In contrast, current cannot flow through a transistor gate.

Figure 4-2 shows a simple example with two inverters separated by a pass gate. The first inverter and the pass gate constitute a channel-connected block because the possible current paths include all of their transistors. The second inverter is not part of the channel-connected block because current flow is blocked at the gates. The dashed lines indicate the boundaries of the channel-connected regions.

Figure 4-2 Channel-Connected Blocks

Reporting Channel-Connected Blocks

The `report_channel_connected_block` command reports characteristics of the channel-connected blocks in a net or collection of nets. When no options are specified, the report includes the number of devices, nets, inputs and outputs, and some attributes, such as whether the channel-connected block includes the timing model or port. NanoTime assigns a name for each channel-connected block by taking the name of one of the nets in the block.

The circuit in Figure 4-3 is used to illustrate a channel-connected block report.

Figure 4-3 Circuit Example for Reporting Channel-Connected Blocks

The summary report of the channel-connected blocks for nets lm, db, om, and os is produced by the following command:

```
nt_shell> report_channel_connected_block "lm db om os"

Report : report_channel_connected_block
...
CCB ID : A net in the CCB which is the representative of the CCB
CCB Inputs : The nets which drive the gate pins of the CCB devices
CCB Outputs : The nets in the CCB which drive the inputs of other CCBs

            #CCB    #CCB     #CCB      #CCB
Net      devices    nets    Input   outputs     Attrs   CCB ID
------- --------- ------- ------- ---------- --------- -------
db             8       3        5          1        cl      db
lm             8       3        5          1        cl      db
om             8       3        5          2        cl      ls
os             2       1        1          1                os
Total #CCB : 3
```

Nets db and lm are part of the same channel-connected block, therefore the first and second rows of the report are identical. The “CCB ID” column shows the name assigned to the block (db). The report covers four nets, so there are four lines in the report and four unique entries in the “Net” column. However, there are only three unique channel-connected blocks and three unique entries in the “CCB ID” column. The number of unique CCBs appears at the end of the report.

The `-verbose` option generates a listing of all devices and nets of the channel-connected block in the report, as shown in this example:

```
nt_shell> report_channel_connected_block "lm" -verbose
...
Report : report_channel_connected_block
Attributes:
c  - Clocked device
b  - Feedback device
t  - Timing model
l  - Latch net
p  - Precharge net
r  - Register file net
g  - Gated clock net
f  - Forced clock net
s  - Stopped clock net

CCB ID : A net in the CCB which is the representative of the CCB
Net : lm
CCB ID: db
Nets         Attrs
---------- ---------
lm                l
db
X2.int

Devices      Attrs
---------- ---------
P1               c
N1               c
X1.p
X1.n
X2.p1            b
X2.p2
X2.n1            b
X2.n2            b
```

To report the channel-connected block inputs or outputs, use the `-input_connection` or `-output_connection` options. If a channel-connected block output connects to an extracted timing model input, the model outputs which have a delay arc from the model input are reported with the `report_channel_connected_block -output_connection` command. If a channel-connected block input connects to an ETM output, the ETM output pin is reported with the `report_channel_connected_block -input_connection` command.

An example of the report is as follows:

```
nt_shell> report_channel_connected_block "ls" -input_connection
-output_connection
...
Report : report_channel_connected_block

Attributes:
c  - Clocked device
b  - Feedback device
l  - Latch net
p  - Precharge net
r  - Register file net
g  - Gated clock net
f  - Forced clock net
s  - Stopped clock net
P  - Port

CCB ID :  A net in the CCB which is the representative of the CCB
CCB Inputs : The nets which drive the gate pins of the CCB devices
CCB Outputs : The nets in the CCB which drive the inputs of other CCBs
CCB Fan-ins : The IDs of other CCBs which drive the reported CCB's inputs
CCB Fan-outs : The IDs of other CCBs which are driven by the reported
CCB's outputs

Net:      om
CCB ID:   ls

   CCB                     CCB
Inputs       Attrs     Fan-ins
---------- ------- -------------
ck               c          ck
ckb              c         ckb
lm               l          db
rs               P      reset2
os               l          os

    CCB                    CCB
Outputs      Attrs    Fan-outs
---------- ------- -------------
ls               l          os
ls               l          qN
om                          lm
```

Recognizing Storage Nodes

A latch node or storage node is a net that intentionally maintains a constant logic state. A loop consisting of two channel-connected blocks is widely used to build latches. If such a loop is not marked as any type of sequential element, the automatic recognition of the sequential elements might be incomplete. NanoTime provides an optional method for finding and reporting storage nodes in the design.

During the topology matching step, NanoTime finds storage nodes by detecting double channel-connected block loops. NanoTime recognizes a storage node when it is the output of a channel-connected block that is part of a double channel-connected block loop, as shown in Figure 4-4. The tool only recognizes loops with two channel-connected blocks.

Figure 4-4 Storage Nodes in a Multiple Channel-Connected Block Loop

NanoTime ignores large channel-connected blocks in a double channel-connected block loop if the number of devices in the channel-connected block exceeds the threshold defined by the `topo_big_ccb_transistors` variable. The default is 100. A warning message appears whenever a large channel-connected block is ignored.

Storage nodes that are not part of a marked topology or that are not clocked are considered to be error conditions. To understand the situation, examine the storage node report. To resolve the issues, either mark a sequential topology on the storage node, resolve clock propagation issues, or remove the storage node designation.

In the case of a register file, such as the one shown in Figure 4-5, one pair of channel-connected blocks might have multiple storage nodes between the blocks.

Figure 4-5 Register File With Multiple Storage Nodes Between Channel-Connected Blocks

In the case of the flip-flop shown in Figure 4-6, the channel-connected block in the center belongs to loops with both the master latch and the slave latch.

Figure 4-6 Flip-Flop with Three Channel-Connected Blocks

The strict sequential topology setup flow, shown in Figure 4-7, is available to help you set up sequential topologies correctly. Set the `topo_check_storage_node` variable to `true` before executing the `match_topology` command to turn on the flow. When you use this flow, the `check_topology` command issues an error message if there are any storage node error conditions. Otherwise, NanoTime does not flag the storage node issues and you assume responsibility for the circuit design.

Figure 4-7 Strict Sequential Topology Setup Flow

Reporting Storage Nodes

The `report_storage_node` command produces a summary of all storage nodes in the design, including the counts of valid sequential topologies, invalid sequential topologies, and storage nodes without topology marking. The invalid sequential topologies are user-defined or automatically recognized latches, precharge structures, and register files that are not driven by clocks. Invalid sequential topologies and storage nodes without topology markings are reported as errors, which must be fixed.

An example of a storage node report is as follows:

```
nt_shell> report_storage_node

****************************************
Report : storage_nodes
...
****************************************
Type                                   Count
------------------------------------ ----------
Valid Sequential Marking                    2
Invalid Sequential Marking                  0
No marking                                  6
------------------------------------ ----------
Total                                       8
```

The `-verbose` option causes the report to list the subcircuit containing the storage node, other storage nodes in the same multiple channel-connected block loop, and attributes of the storage node. If the storage node is marked or already recognized as a latch, precharge, or register file net, the report shows the attribute l, p, or r, respectively. The attribute c indicates that a clock propagates to the channel-connected block loop of the storage node.

An example of a storage node report with the verbose option is as follows:

```
nt_shell> report_storage_node -verbose
...
Attributes:
c  - Clocked
l  - Latch
p  - Precharge
r  - Register file

Storage Node     Related Storage Nodes  Sub Circuit     Attributes
---------------- --------------------- -------------- ------------
fb_out              {ff_out}             inv_latch          lc
ff_out              {fb_out}             inv_latch          lc
X1.fb_out           {X1.ff_out}          unknown_loop        c
X1.ff_out           {X1.fb_out}          unknown_loop        c
X2.fb_out           {X2.ff_out}          unknown_loop        c
X2.ff_out           {X2.fb_out}          unknown_loop        c
cross_nand2_out     {cross_nand1_out}    cross_coupled_nand
cross_nand1_out     {cross_nand2_out}    cross_coupled_nand
```

The `-errors` option causes the report to include the number of storage nodes with error conditions. The argument for the `-errors` option specifies which type of error to report:

- `not_marked` — Storage nodes that are not marked as a latch net, precharge net, or register file net, and have no feedback device driving the storage node
- `not_clocked` — Storage nodes for which a clock does not propagate to the channel-connected block loop containing the storage node
- `all` — Both types of error

You can generate a report for one specific storage node or a collection of storage nodes. You can obtain a report of the other storage nodes in the same channel-connected block loop as the specified node by using the `report_channel_connected_block` command.

You can remove the storage node classification from a net or a collection of nets by using the `remove_storage_node` command.

The storage node report shows the subcircuit to which a storage node belongs. To waive multiple storage nodes in instances of the same subcircuit, you can review one of them and remove the whole group with subcircuit matching.

Setting and Reporting Transistor Direction

NanoTime checks transistor direction at the `check_topology` command. By default, transistors with unresolved directions are considered to be errors. This is known as the strict transistor direction flow.

If you prefer not to check transistor direction, set the `topo_check_transistor_direction` variable to `false` before executing the `check_topology` command.

For some designs, the transistor direction cannot be fully resolved without simulation or circuit analysis, as shown in the example in Figure 4-8. Using your knowledge of the design to set transistor directions can improve performance and reduce pessimism during path tracing.

Figure 4-8 Unresolved Transistor Direction

After you use the `match_topology` command, check the direction of all transistors by using the `report_transistor_direction` command. If the report shows a large number of bidirectional transistors that are not truly bidirectional, use the `set_transistor_direction` command to specify the actual direction. An excessive number of bidirectional transistors might adversely affect topology recognition, especially for multiplexers, and might cause longer path searching times. User-specified bidirectional transistors are allowed.

The following example shows a transistor direction report:

```
**********************************
Report : transistor direction
...
**********************************
Type                                   Count
----------------------------------- --------
Bidirectional                              3
Unidirectional (Automatically Set)        26
Unidirectional (User Defined)              0
-------------------------------------------
Total                                     29
```

Use the `-transistor` option of the `set_transistor_direction` command to specify the direction. Valid arguments are `s2d`, `d2s`, `bidi`, or `none`, which mean source-to-drain, drain-to-source, bidirectional, or nonconducting, respectively. Alternatively, you can use the `-to_net` or `-from_net` options of the `set_transistor_direction` command to set the signal flow direction toward or away from specified nets, for all transistors listed in the command or for all transistors connected to the specified nets.

NanoTime automatically recognizes many nondirectional (nonconducting) transistors. For example, in Figure 4-9, if input A is tied to ground, transistor mnA is off. Transistors mnB and

mnC are connected only to a rail (ground) and an off transistor. Therefore the direction of transistor mnB is inconsequential and NanoTime waives the direction checking operation.

Figure 4-9 Transistor Direction Checking Example

Some transistor directions are not resolved when case analysis or logic constraints turn off the signal flow through the transistor. NanoTime reports these nondirectional transistors as errors. However, you can waive direction checking on nondirectional transistors by setting the `topo_waive_nondirectional_transistor` variable to `true` before running the `check_topology` command.

The following example shows a transistor direction report, with additional information about bidirectional transistors generated by using the `-bidirectional` option.

```
****************************************
Report : transistor direction
...
****************************************
f --- floating output related bi-directional transistors
u –-- user defined direction

Bidirectional Transistor  Attribute   Sub Circuit
------------------------ ----------- -----------
Test1.inst1.mn1                          c1
Test1.inst2.mn1                 f        c1
Test1.inst3.mn3                 u        c2
```

The following example shows a report that provides the SPICE netlist for an unresolved transistor as a result of using the `-transistor` and `-channel_connected_block` options of the `report_transistor_direction` command:

```
****************************************
Report : transistor direction
...
****************************************
f --- floating output related bi-directional transistors
u –-- user defined direction

Transistor                        Direction  Attribute   Sub Circuit
------------------------------ ---------- ----------  -------------
Test1.inst1.mn1                       bidi       u                c1
.SUBCKT test1.inst1.mn1 s1 g1 d1
Test1.inst1.mn1 s1 g1 d1 vss nch
Test1.inst1.mn2 s2 g2 d1 vss nch
Test1.inv1.mn vss g3 s1 vss nch
Test1.inv1.mp vdd g3 s1 vdd pch
Test1.inv2.mn vss g4 s2 vss nch
Test1.inv2.mp vdd g4 s2 vdd pch
.ENDS
```

Debugging the Data Inputs of Sequential Elements

Sequential topologies such as latches have clock input pins and data input pins. By default, the clock input pins are expected to be driven by periodic clock signals and the data input pins are expected to be driven by data signals.

NanoTime can determine when the data input pin of a latch is driven by a clock signal. This might be acceptable for some circuits, such as a clock divider circuit. In other cases, this situation might indicate problems with the design or with topology recognition.

To evaluate the types of input signals that arrive at latch data input pins, set the `topo_find_clock_driven_data_inputs` variable to `true`. NanoTime issues a warning message at the `check_topology` command if either of the following conditions is true:

- The data input is driven by a clock.
- The data input is connected to a rail.

Analysis does not stop for a warning. If you want to stop the run when a clock signal drives a data input, set the `topo_sequential_structure_strict_input_matching` variable to `true`. This variable takes effect only if the `topo_find_clock_driven_data_inputs` variable is set to `true`. The default of both variables is `false`.

Note:

In clock divider circuits, the data input is usually a clock signal. NanoTime automatically recognizes simple clock divider topologies and does not apply the strict input matching test. However, for complex clock divider circuits, you might need to set the `topo_sequential_structure_strict_input_matching` variable to `false` to allow the analysis to proceed.

For manually marked latches, the data input pin is explicitly specified. For automatically recognized structures, the tool must identify the data input pin. To report the pins identified as data input pins for sequential elements, use the `-message_level debug` option with the `check_topology` command.

Manually Marking and Erasing Structures

NanoTime automatically recognizes structures based on the circuit topology. In addition, the `mark_*` commands such as the `mark_latch` and `mark_mux` commands allow you to specify structures that NanoTime does not recognize automatically.

The `erase_*` commands, such as the `erase_latch` and `erase_mux` commands, remove topology markings. Before the first execution of a `match_topology` command, the only structures available for erasing are manually marked structures. After a `match_topology` command performs automatic recognition, many topologies are marked. You can erase any existing topology structure.

Important:

Executing an `erase_*` command does not prevent similar topology structures from being recognized later. If the `topo_match_topology_reset_clock_and_topology` variable is set to `true`, NanoTime repropagates the clock network and rerecognizes topologies at every subsequent `match_topology -force` command (note the use of the `-force` option). In this case, the tool is likely to rerecognize any automatically recognized structures that you erased. You can avoid this occurrence by performing topology erasing steps after the last `match_topology` command in the flow.

To automatically mark or erase multiple occurrences of a circuit pattern or cell, use the `foreach_match` command.

An alternative to using the `erase_*` commands to remove recognition of structures is to use the `erase_topology` command with the `get_topology` command. The `get_topology` command creates a collection of structures and the `erase_topology` command removes recognition of all the structures in the collection.

Using the `get_topology` and `erase_topology` commands together might be easier than using the `erase_*` commands because of the flexibility of the `get_topology` command. You can create a collection of all instances of a structure and erase them immediately. For example, the following command removes recognition of all latch structures in the design:

```
nt_shell> erase_topology [get_topology -all -structure_type latch]
```

Special Purpose Marking Commands

Three topology marking commands provide special instructions for objects in a design:

- The `mark_pin` command applies to a list of pins.

  This command has only one option: the `-kill_path_constraints` option. This option specifies that logic constraints previously set on the specified pin or pins should be ignored. Use this option to suppress timing checks on paths that include the specified pins.

- The `mark_net` command applies to a list of nets.

  The options of the `mark_net` command address a variety of purposes, including (but not limited to) the following:

  - Marking partially differential circuits
  - Specifying the load model (transistor or capacitor) to use for nonswitching transistors attached to the specified nets
  - Specifying how or whether to check nets for Miller analysis switching direction or clock gate resolution

- The `mark_instance` command applies to a list of transistor elements.

  The options of the `mark_instance` command generally specify that the transistors in the list should not be included in some set of operations, such as path tracing or dynamic simulation.

  Excluding a transistor from a path tracing operation also prevents path tracing and noise analysis from occurring on all paths downstream from that transistor.

Reporting Topology Information

After running the `match_topology` command, you can use the `report_topology` and `get_topology` commands to generate reports on recognized structures in the design. The `report_topology` command lists the number and locations of specified structure types. The `get_topology` command creates a collection of structures in the design based on the structure type and structure attributes.

The report_topology Command

The `report_topology` command reports the topology structures in the design, whether the structures were recognized automatically by the `match_topology` command or marked manually with the `mark_*` commands. By default, the `report_topology` command lists the structure types and reports the number of occurrences of each type of structure in the design, as shown in the following example.

```
nt_shell> report_topology
 ...
                                        User   Global
                     Auto Manually  Library  Library
Structure Type     Marked   Marked   Marked   Marked    Total
--------------- -------- -------- -------- -------- --------
clock_gate           192        0        0        0      192
cross_coupled          0        0        0        0        0
cross_coupled_pmos     0        1        0        0        1
feedback             224        0        0        0      224
inverter            3171        0        0        0     3171
latch                192      128        0        0      320
flip_flop       disabled        0        0        0        0
mux                  256        0        0        0      256
precharge            224        0        0        0      224
pulldown        disabled        0        0        0        0
pullup          disabled        0        0        0        0
ram                    0        0        0        0        0
register        disabled        0        0        0        0
tgate                576        0        0        0      576
turnoff                0        0        0        0        0
weak_pullup     disabled        0        0        0        0
xor                    0        0        0        0        0
```

For information about each structure, use the `-verbose` option. To restrict the report to just one type of structure, use the `-structure_type` option and specify one or more of the structure types. For example,

```
nt_shell> report_topology -structure_type feedback -verbose
 ...
Attributes:
    u - User Defined
    C - Common Topology Library Defined
    U - User Topology Library Defined
    G - Global Topology Library Defined

Feedback                            Attrs
---------------------------------- -----
Xaddsub.Xadder.Xla13.Xlgen.XC1.Mp1
Xaddsub.Xadder.Xla4.XG3.X0.Mp1
Xaddsub.Xadder.Xla13.Xlgen.XC4.Mp1
 ...
```

To restrict the scope of the search to just part of the design rather than the whole design, specify a topology collection with the `get_topology` command. For example,

```
nt_shell> report_topology -structure_type latch \
          -verbose [get_topology -of_objects \
          [get_cells Xaddsub]]
```

The get_topology Command

The `get_topology` command creates a collection of topologies in the current design. Topologies are created by automatic recognition with the `match_topology` command and by user directives such as the `mark_latch` command.

The collection created by the `get_topology` command can be reported with the `report_topology` command, erased with the `erase_topology` command, or traversed with the `foreach_in_collection` command. During traversal, you can report each topology object's attributes with the `get_attribute` command.

For example, the following command removes recognition of all latch structures in the design:

```
nt_shell> erase_topology [get_topology -all \
          -structure_type latch]
```

The following command creates a collection of inverter structures. The `echo` command reports the number of inverters currently recognized or marked in the design.

```
nt_shell> echo [sizeof_collection [get_topology -all \
          -structure_type inverter]]
25
```

The `get_topology` command must use either the `-all` or `-of_objects` option. The `-all` option causes the command to search through the whole current design. The `-of_objects` option causes the command to search for topologies associated with a specified collection of nets or leaf-level cells.

Each structure type is associated with a net or cell, as indicated in the following list. For example, to get a collection of inverter structures using the `-of_objects` option, you would search through a collection of nets, not cells.

```
clock_gate    net       cross_coupled_pmos    net
flip-flop     net       inverter              net
latch         net       mux                   net
precharge     net       ram                   net
register      net       turnoff               net
xor           net
feedback      cell      pulldown              cell
pullup        cell      tgate                 cell
weak_pullup   cell
```

You can use the `-filter` option of the `get_topology` command to restrict the collection to structures for which a logical expression is true. For example, the following command generates a list of all latches having the structure name mylatch1. Structure names can be assigned with the `-name` option of the `mark_latch` command, other `mark_*` commands, and the `match_topology` command.

```
nt_shell> get_topology -all -structure_type latch \
          -filter "structure_name == mylatch1"
```

Each topology has a set of attributes that you can report with the `foreach_in_collection` and `get_attribute` commands. The following script defines a process called `print_latch_nets` that gets a collection of all latches and prints the latch net of each latch.

```
proc print_latch_nets {} {
  set latches [get_topology -all -structure_type latch]
  foreach_in_collection latch $latches {
    set net [get_attribute $latch latch_net]
    echo [get_attribute $net full_name]
  }
}

nt_shell> print_latch_nets
xl4.qn
xl3.qn
xl2.qn
xl1.qn
```

See Also

- Chapter 19, “Object Attributes”

Searching by Name or Pattern Matching

The `foreach_match` command identifies all instances of a pattern or named subcircuit and applies a command to each matching instance. The applied command can specify a structure (latch, multiplexer, inverter, register, and so on) or other features, such as transistor direction or clock network characteristics. The matching is pattern-based if you use the `-pattern` option, or name-based otherwise.

For example, the following command performs a name-based search. It applies the `set_transistor_direction` command to all instances of the subcircuit named MUX and specifies the signal direction as source-to-drain for transistor MN1 in MUX.

```
nt_shell> foreach_match MUX -command \
          { set_transistor_direction -transistor s2d MN1 }
1
1
1
...
```

The command returns a 1 for each successful execution of the embedded command, followed by the total number of matches upon completion of the `foreach_match` command.

To use the `-pattern` option, you must first read the pattern into NanoTime with the `read_pattern` command. To display the subcircuits that have been read in as patterns, use the `list_patterns` command.

The following example reads the pattern file named pattern.sp and lists the subcircuits that have been read in as patterns. Then, for each matching occurrence of the pattern in the current design, it applies the `erase_turnoff` command at output net z.

```
nt_shell> read_pattern -format spice pattern.sp
Compiling "/node1/user1/designs/pattern.sp"
1

nt_shell> list_patterns
Pattern Registry

Pattern Name
-------------------------------------------
pre_an2k
1
nt_shell> foreach_match -pattern pre_an2k \
          -command {erase_turnoff -output_net z}
1
...
```

The `read_pattern` command looks for the specified pattern file in the directories specified by the `library_path` variable. The pattern file must be read after the `link_design` command and before the `foreach_match` command. SPICE format is the most common.

The following is an example of a pattern file in the form of a SPICE subcircuit, corresponding to the schematic in Figure 4-10:

```
*********************************************
* Block: pre_an2k                           *
*********************************************
.subckt pre_an2k clk d1 d2 z
mp1 vdd clk enode vdd p
mp2 vdd z enode vdd p
mp3 vdd clk n1 vdd p
mp4 vdd enode z vdd p
mn1 enode d1 n1 gnd n
mn2 n1 d2 n2 gnd n
mn3 n2 clk gnd gnd n
mn4 z enode gnd gnd n
.ends pre_an2k
```

Figure 4-10 Pattern File Circuit

The pattern file can contain multiple pattern subcircuits, although each subcircuit must be a flat netlist. Hierarchical netlists are not supported for patterns.

A SPICE pattern file might contain device parameter information such as transistor widths and lengths, but that information is not used for pattern matching. NanoTime checks for an identical configuration of interconnected NMOS and PMOS transistors, ignoring transistor parameters other than the NMOS or PMOS type.

A SPICE pattern file must follow all rules for SPICE format. For example, the first line must be a comment. If the pattern is for an existing SPICE subcircuit, the SPICE file can be used directly. For example,

```
*****  Pattern Files *****
.subckt mynand A B Z
Mn0 Z A n1 gnd nch W=dwn L=0.130U
+  NRD=0.157 NRS=0.091
Mn1 n1 B gnd gnd nch W=dwn L=0.130U
+  NRD=0.157 NRS=0.091
Mp0 vdd A Z vdd pch W='dwp * 2' L=0.130U
+  NRD=0.045 NRS=0.069
Mp1 vdd B Z vdd pch W='dwp * 2' L=0.130U
+  NRD=0.045 NRS=0.069
.ends
```

The transistor sizes and other parameters are ignored for pattern matching. You can omit the unused information and use “n” and “p” for the transistor types:

```
***** Pattern Files *****
.subckt mynand A B Z
Mn0 Z A n1 n
Mn1 n1 B gnd  n
Mp0 vdd A Z  p
Mp1 vdd B Z p
.ends
```

All input and output nets of the pattern should be specified as pins. Unused outputs do not cause the pattern matching to fail, but if an output is not specified as a pin and that output is used elsewhere in the circuit, the pattern does not match.

Power and ground need not be specified as pins. If specified, they are ignored for pattern matching. If the only pins specified for a pattern are power and ground, NanoTime does not try to match the pattern, as it has no connection to the rest of the circuit.

The `foreach_match -pattern` command returns the number of matches. You can also add commands to print out the details of the pattern match. For example,

```
foreach_match -pattern abc -command {
  set_logic_constraint -one_hot { A B }
  echo [get_attribute [get_nets "n1"] full_name]
}
X5.X6.n1
X3.n1
2
```

The final `2` is the number of matches of pattern abc. Node `n1` in the pattern matches `X5.X6.n1` and `X3.n1`.

For best pattern matching results, keep patterns small in size, to match just enough of the netlist to apply the required commands. Avoid putting unused outputs in the pin or port list to minimize the pattern matching time.

Use transistor names and net names in the pattern that have meaning and that allow you to execute pattern-matching Tcl commands easily. For example,

```
foreach_match -pattern LAT_PAT -command {
   mark_feedback -transistors [get_cells -nocase Mfdbk*] }
```

To remove a pattern that you no longer need, use the `remove_pattern` command.

If the `pattern_merge_parallel_transistors` variable is `true` (the default), multiple transistors that share common source, drain, and gate connections are considered to be a single transistor for pattern matching, including swapped source and drain transistors.

However, this does not include parallel stacks. For example, if two NAND gates are connected in parallel with common inputs and output, the two stacked NMOS transistors are not considered to be in parallel because the intermediate connection between the two stacked transistors is not connected in parallel between the two gates. As a result, there would be two matches reported, not one.

See Also

- Marking Topologies With Automatic Pattern Matching
- The Topology Database

Marking Topologies With Automatic Pattern Matching

The `-match_pattern` option is available for the `mark_latch`, `mark_precharge`, and `mark_register_file` commands. This option enables NanoTime to build a pattern with the nets and devices specified in the command, perform pattern matching, and apply the command on the matched instances.

This option requires that you specify additional portions of the topology so that NanoTime can build an accurate pattern for matching. For example, the `mark_register_file` command requires only a pair of register file nets to mark the topology; the transistors in the register file are optional. However, when the `-match_pattern` option is used, you must specify the transistors in the register file in order for pattern matching to work.

For commands that have the `-match_pattern` option, the `topo_create_lib_topology` variable saves the patterns and commands as a user topology library. By default, the variable is `false`. Set the `topo_create_lib_topology` variable to `true` to create a topology library. NanoTime creates a subdirectory named _user_lib in the current run directory to hold the generated patterns and library topologies. The subdirectory contains a user library definition file named _user_lib.ntlib. To store the library in a different directory, specify the directory with the `topo_create_lib_topology_directory` variable.

The pattern and the topology library are named topology_$*n*.sp and topology_$*n*.lib, respectively, where *n* is a unique integer for each pattern.

Using the circuit in Figure 4-3, the following example shows how to mark the storage nodes within the unknown_loop subcircuit as latches:

```
nt_shell> mark_latch -latch_net X1.fb_out -output X1.ff_out \
   -feedback "X1.fb_tx1 X1.fb_tx2" -feedforward "X1.ff_tx1 X1.ff_tx2" \
   -clock "X1. ck_p.g X1.ck_n.g" -match_pattern

nt_shell> report_storage_node
...
Type                                   Count
------------------------------------ ----------
Valid Sequential Marking                  6
Invalid Sequential Marking                0
No marking                                2
---------------------------------------------
Total                                     8
```

See Also

- Searching by Name or Pattern Matching
- The Topology Database

The Topology Database

The NanoTime installation includes a built-in set of topology patterns that the tool can recognize. These patterns are stored within a topology database. You can add your own custom patterns to the local topology database to ensure that NanoTime recognizes and marks those patterns in addition to the standard patterns.

These terms describe the topology database:

- Library topology or lib_topology: A file or pair of files that describe the pattern or subcircuit name and the list of actions taken when the pattern or subcircuit is found.
- Topology library: A directory containing a set of lib_topology definitions.
- Topology database: The set of topology libraries that NanoTime can use for topology recognition.

See Also

- Enabling Selected Libraries and Topologies
- Reporting Library Topologies

Topology Libraries

The topology database consists of topology libraries, which include a common library, a global library, and optional user libraries. Each topology library is a directory that contains one control file plus the files that describe the individual topologies. Figure 4-11 shows the structure of a topology database containing the global and common libraries and two user libraries.

Figure 4-11 Topology Database Structure

The common library contains a set of the most commonly used topologies, such as simple logic gates, latches, and precharge circuits. The global library contains another set of topologies that are used less often. A user library contains user-defined topologies such as design-specific or proprietary circuits.

The common and global topology libraries are built into the NanoTime installation and should not be modified. They can be found in the installation path as follows:

```
nt_install_dir/auxx/nt/topolibs/common
nt_install_dir/auxx/nt/topolibs/global
```

A user library can be installed in any directory.

Each library must contain a control file named *dir_name*.ntlib. For example, the global library contains a file named global.ntlib and the common library contains a file named common.ntlib. A control file contains only the `create_topology_library` command. This command cannot be used at the `nt_shell` prompt.

For each library, including the global and common libraries, NanoTime executes the `create_topology_library` command from the *dir_name*.ntlib file. This action creates the topology library and assigns a user-specified version string and description string to the library. The two strings are used for reference and do not affect topology recognition.

NanoTime automatically reads in the global and common libraries, which allows searching through these libraries during topology recognition.

To include user libraries, use the `read_topology_library` command to read them in before executing any topology recognition commands. In addition, you must include the name of the user libraries in the `topo_library_order` variable.

Topology objects within the libraries must be individually enabled for searching before they can be used. By default, the objects within the common library are enabled, but the objects within the global library and user libraries are not enabled.

Only the libraries listed in the `topo_library_order` variable are available in the current NanoTime session. NanoTime searches libraries in the order in which they are listed in the variable. The variable includes the global and common libraries by default.

For example, to enable searching in a library named user1, use the following commands:

```
nt_shell> printvar topo_library_order
topo_library_order   = "global common"
nt_shell> set topo_library_order "global common user1"
global common user1
```

To remove a topology library that has been read in, use the `remove_topology_library` command. Do not remove the global or common libraries.

See Also

- Enabling Selected Libraries and Topologies
- Reporting Library Topologies

Library Topology Creation (.libt) Files

For each lib_topology object contained in a library, there must be a file named *topo_name*.libt. Within this file, the `create_lib_topology` command creates the lib_topology object. This command can be used only in a .libt file in a topology library, not at the NanoTime shell prompt.

The `-version` and `-description` options of the `create_lib_topology` command specify two strings used to reference the lib_topology object. The strings do not otherwise affect topology recognition.

The `-enable_search` option causes searching for the topology to be enabled by default when its library is also enabled. If this option is not present, you can enable the topology for searching by using the `set_search_enabled` command at the NanoTime shell prompt or in a script. By default, the objects within the common library are enabled, but the objects within the global library are not enabled.

The `create_lib_topology` command provides a set of options that specify actions (sets of commands) for NanoTime to take when the topology object is matched. At least one action should be defined that marks one or more topology objects in the matching netlist pattern. There are seven points in the analysis flow where the actions can occur. NanoTime tracks all of the matched patterns and executes the corresponding action commands at the designated times in the analysis flow.

There are two methods for searching and identifying structures in the design: subcircuit matching and pattern matching.

- In the subcircuit matching method, NanoTime searches for SPICE subcircuits that match the name provided with the `-subckt_name` option. The name can contain wildcard characters, for example, `lat_s*`.
- In the pattern matching method, NanoTime searches for transistor structures in the netlist that match a SPICE pattern file in the topology library. The pattern file must be present when the library is read in with the `read_topology_library` command. The pattern file must have the same base name as the lib_topology file, with the extension .sp rather than .libt. The file must contain one or more subcircuit definitions using SPICE netlist syntax. The subcircuit names of the search patterns apply only to the local lib_topology definition.

Multiple pattern definitions can be used to represent variations of the basic netlist structure of the lib_topology object. NanoTime searches for each type of pattern in the order that it appears in the pattern file and applies the action commands to each successful match. The net and device names used in multiple subcircuit definitions must all be consistent with the action commands.

To apply action commands on the current design, use the `-current_design` option. If this option is not specified, you must supply subcircuit names for subcircuit name matching or a search pattern in a .sp file.

You can separately specify the actions taken at any of the following phase points:

- Before or after the `match_topology` command
- After clock propagation
- Before or after the `check_topology` command
- Before or after the `check_design` command

For example, the following file, muxflop.libt, performs a latch-marking operation after clock propagation for each occurrence of the “muxflop” subcircuit:

```
create_lib_topology \
  -description "muxflop latch" -version ‘V20070215’ \
  -subckt_name muxflop \
  -enable_search \
  -action_post_clock_propagation { \
     mark_latch -latch_net lat1n -output lat1 \
     -feedforward { X5.Mn0 X5.Mp0 } \
     -feedback { Mp6 Mp7 Mn6 Mn7 } \
     -clock { Mn0.G Mp0.G } -name muxflop \
  }
```

Table 4-1 lists the phase points during which actions can be carried out and the typical actions that occur during these phases. Actions are executed in the same order that the topologies are searched. The “pre” phase actions occur before any built-in actions. The “post” phase actions occur after any built-in actions have been processed.

Table 4-1 Topology Action Phase Points and Precedence

Phase point	Typical actions performed
`-action_pre_match_topology`	Execute clock tree propagation adjustments needed for correct marking of clocked topologies. Library topologies take precedence over automatic recognition.
`-action_post_clock_propagation`	Execute topology marking after clock propagation completion. User action command takes precedence. Action scripts can remove any automatically recognized topology.
`-action_post_match_topology`	Execute topology marking after built-in recognition. Automatic recognition takes precedence. Action script can remove any automatically recognized topology.

Table 4-1 Topology Action Phase Points and Precedence (Continued)

Phase point	Typical actions performed
`-action_pre_check_topology`	Execute clock tree propagation adjustments needed for correct marking of clocked topologies. Library topologies take precedence over automatic recognition.
`-action_post_check_topology`	Execute topology marking after built-in recognition. Automatic recognition takes precedence. Action script can remove any auto recognized topology.
`-action_pre_check_design`	Execute commands to adjust settings of marked topologies. Actions are taken before `check_design` activities are launched.
`-action_post_check_design`	Execute commands to adjust timing constraints generated from marked topologies. The action scripts in this phase can verify that marked topologies generated the expected timing constraints.

Topology Pattern (.sp) Files

If there is no `-subckt_name` option with the `create_lib_topology` command, a *topo_name*.sp file must be present to specify the circuit search pattern in SPICE format. For example, the following .libt and .sp files define a topology pattern called latch_otri_istack, which is marked as a latch when found.

The latch_otri_istack.sp file is as follows:

```
*********************************************************
* block: latch_otri_istack
*********************************************************
.subckt latch_otri_istack clk clkb en enb latchnet latchout
* feedforward
mffp1 latchout latchnet ffp1 pch
mffp2 ffp1 enb __vdd__ pch
mffn1 latchout latchnet ffn1 nch
mffn2 ffn1 en __gnd__ nch
*feedback
mfbp1 latchnet latchout fbp1 pch
mfbcp2 fbp1 clkb __vdd__ pch
mfbn1 latchnet latchout fbn1 nch
mfbcn2 fbn1 clk __gnd__ nch
.ends
```

The latch_otri_istack.libt file is as follows:

```
create_lib_topology\
  -description "latch_otri_istack latch"\
  -version "V20070215"\
  -action_post_clock_propagation {
      set clock_txs [get_cells -nocase "mfbc*"]
      if { [ topodb_latch_check_clock_collection $clock_txs ] } {
        set clock_pins \
            [topodb_get_attribute_collection \
             $clock_txs transistor_gate_pin]
        mark_latch -latch_net [get_nets -nocase latchnet] \
            -output [get_nets -nocase latchout] \
            -clock $clock_pins \
            -feedback [get_cells -nocase "mfb*"] \
            -feedforward [get_cells -nocase "mff*"] \
            -name "latch_otri_istack"
      }
  }
```

Enabling Selected Libraries and Topologies

To enable a user topology library, you must read the topology library into NanoTime with the `read_topology_library` command and also add the name to the `topo_library_order` variable. For example,

```
nt_shell> read_topology_library ../topolibs/user1
Loading library 'user1' with ntlib file ...
Compiling ...
...
nt_shell> set topo_library_order "global common user1"
```

By default, when you enable a topology library, you enable searching for all topologies that were defined in that library with the `-enable_search` option of the `create_lib_topology` command. You can selectively enable or disable searching for individual lib_topology objects by using the `set_search_enabled` and `remove_search_enabled` commands.

To search for a topology without taking any action when it is found, use the `-only_used` option:

```
nt_shell> set_search_enabled -only_used
```

NanoTime searches for all loaded lib_topology objects, whether or not they were previously enabled for searching, and marks each lib_topology object's `search_enabled` attribute to `true` if the topology exists in the design, or to `false` otherwise. No actions are carried out on the topologies found. After running this command, searching is enabled for all topologies found to exist in the design, and disabled for ones that do not exist in the design.

To specify explicitly which lib_topology objects to enable or disable, use the `set_search_enabled` or `remove_search_enabled` command on a lib_topology collection created with the `get_lib_topology` command, as in the following examples:

```
nt_shell> set_search_enabled [get_lib_topology library_name.pattern]
```

```
nt_shell> remove_search_enabled [get_lib_topology library_name.pattern]
```

For example, to get a collection of all the lib_topology objects in the global library:

```
nt_shell> get_lib_topology global.*
{"global.latch_itri_istack", global.latch_itri ... }
```

Instead of using the `get_lib_topology` command directly, you can specify the lib_topology names in a separate file. For example,

```
nt_shell> set_search_enabled -from_file mylist
```

File mylist specifies each lib_topology object by library name and lib_topology name, separated by a space, as follows:

```
common nando_istack2
common noro_istack2
global latch_itri_ostack
user_topo_lib latch_itri_ostack
...
```

You can filter the collection based on the `lib_topology` object attributes. For example, the following command gets a collection of all the global lib_topology objects that exist in the design (those that have a `matched_count` attribute greater than zero):

```
nt_shell> get_lib_topology -filter "matched_count>0" global.*
{"global.latch_itri_ostack"}
```

You can enable or disable a lib_topology object from a script, as in the following example:

```
 if { [file exists  alu5.topodb.list] } {
         set_search_enabled -from_file alu5.topodb.list
 }  else {
         set_search_enabled -only_used
         list_lib_topology -only_used -output alu5.topodb.list
 }
```

The following command gets a collection of global lib_topology objects for which searching is currently enabled:

```
nt_shell> get_lib_topology -filter “search_enabled==true” global.*
{"global.latch_itri_ostack ... "}
```

If a lib_topology object is already enabled by the `-enable_search` option of the `create_lib_topology` command in the .libt file in the topology library, then that lib_topology is enabled by default, so an explicit `set_search_enabled` command is not necessary to enable searching for that topology.

See Also

- The Topology Database
- Reporting Library Topologies

Reporting Library Topologies

To see a list of topology libraries that have been read in, use the `list_topology_library` command:

```
nt_shell> list_topology_library
Topology Library Registry

Name         Description                       Version
------------ --------------------------------- ----------
global       Global pattern library            V20070126
common       Builtin common pattern library    V20061218
user1        User pattern library              V20070126
```

To list lib_topology objects that have been read in, use the `list_lib_topology` command:

```
nt_shell> list_lib_topology -library user1
Topology Library Registry

Name         Description                       Version
------------ --------------------------------- ----------
user1        muxflop latch                     V20070126
...
```

The `list_lib_topology` command lists the lib_topology objects in all the loaded libraries, including the common and global libraries. The `list_lib_topology -only_used` command lists only the topologies that exist in the design. This option works only after topology matching has been performed or the `set_search_enabled -only_used` command has been executed.

To list the lib_topology objects that are present in the design, use the `report_topology_library` command. Using the command without options lists all the lib_topology objects in the loaded user libraries. The report shows the lib_topology name, the search type (either “pattern” or the subcircuit name), the number of occurrences matched in the design, the number of those occurrences marked, the enable status (either

enabled or disabled for search), and types of action taken (before the `match_topology` command, after clock propagation, and so on).

An example of the report is as follows:

```
nt_shell> report_topology_library
 ...
Attributes:
    + - Enabled for search
    - - Disabled for search
    m - Pre match topology action (unclocked)
    c - Post clock propagation action
    M - Post match topology action (clocked)
    t - Pre check topology action
    T - Post check topology action
    d - Pre check design action
    D - Post check design action

Library name: user_topo_lib2
Total number of lib_topologies in library: 1
                                                          Topology

Name             Search type <pattern>   Matched    Marked     Attribute
---------------- ----------------------- ---------- ---------- ---------
muxflop          muxflop                 128        128        +c
bignand_latch    pattern                 0          0          -c
 ...
```

Use the `-enabled`, `-only_used`, or *`lib_topology_list`* options to restrict the scope of the report.

See Also

- The Topology Database
- Enabling Selected Libraries and Topologies

5

Clocks and Clock Networks

An essential part of topology recognition is the accurate specification of clocks, clock networks, and clock properties such as latency and uncertainty.

This chapter contains the following sections:

- Clocking Operations in the NanoTime Flow
- Standard Clocks
- Generated Clocks
- Pulse Clocks
- Multiple Clocks
- Clock Latency
- Clock Uncertainty
- Clock Transition Time
- Clock Networks
- Clock Gates
- Reconvergent Clock Gates
- Reporting Clocks and Clock Networks

Clocking Operations in the NanoTime Flow

A clock is a periodic signal that is used to launch and capture data at sequential devices in the design. NanoTime supports three types of clocks:

- Standard clocks
- Generated clocks, including pulse clocks
- Differential clocks

  Analysis of differential clocks requires a NanoTime Ultra license. For more information about differential circuits, see Chapter 7, “Differential Circuits”.

You must specify each clock and its properties, including the period, waveform, latency, uncertainty, and transition time. You might also need to provide guidance to propagate the clock network correctly. The clock network information is necessary for topology recognition of sequential circuit elements.

The `create_clock` command specifies a clock in the design and can also define the period and waveform. However, NanoTime does not use the period or waveform information until later in the analysis flow, when you use the `check_design` command. Therefore, you can change the properties of a clock at any time before the `check_design` command by running a new `create_clock` command to modify the period or waveform. You can also set the latency, uncertainty, and transition time at any point before the `check_design` command.

Some commands require the name of a clock as an argument. To ensure that you are specifying a clock object (not a port, pin, or net), use the `get_clocks` command. For example, to set the uncertainty of all clocks beginning with the letters “PH,” use the following command:

```
nt_shell> set_clock_uncertainty 0.1 [get_clocks "PH*"]
```

You can set the collection to a variable and then operate on that variable. For example, the following example first creates a collection of all clocks that have a period of 4.0 time units, then sets the uncertainty of those clocks.

```
nt_shell> set clock_list [get_clocks * -filter “period==4.0”]
{"MCLK", "clk1", "clk2", "clk3"}
nt_shell> set_clock_uncertainty 0.2 $clock_list
1
```

The following general procedure is an overview of clock-related operations in the NanoTime flow. This chapter covers Steps 1 to 6.

1. Create one or more clocks. Clock types are described later in this chapter.
2. (Optional) Set clock properties such as latency, uncertainty, and transition time. This step can be deferred until later (but must occur before the `check_design` command).
3. If needed, mark clock gate topologies or set the `is_requires_clock` attribute.
4. If needed, mark points in the design at which to force or stop clock networks.
5. Execute the `match_topology` or `check_topology` command to propagate the clock network and recognize topologies.
6. Use the `report_clock` and `report_clock_network` commands to examine the clock sources and networks. Fix any errors before proceeding.
7. Set or modify the clock properties.
8. Set input and output delays, transition times, and loads.
9. If needed, define false paths, timing exceptions, and case analysis.
10. Set timing checks and minimum pulse width checks.
11. (Optional) Load parasitic device data.
12. Execute the `check_design` command.
13. Set gated clock timing check attributes.
14. Execute the `trace_paths` command to perform path tracing.
15. Use the `report_clock_arrivals` command to check clock waveforms.
16. (Optional) Reset the design, then execute the `extract_model` command to create a timing model for the design.
17. (Optional) Examine the model clock domains.

Standard Clocks

Use the `create_clock` command to define single-ended clocks that do not depend on any other circuits. This command creates a clock object and applies that clock to a specified source object, which must be a port or a boundary pin. If you want to define a clock on an internal pin, use the `create_generated_clock` command instead of the `create_clock` command.

You ca define only one clock on any single port or pin. NanoTime traces the clock network so that the clock reaches all sequential elements in the transitive fanout of the source object.

If you do not specify a clock source object, NanoTime creates a virtual clock that can be used to represent a clock outside of the design for specifying input and output delays.

You must specify the period of the clock with the `-period` parameter of the `create_clock` command. Unless you specify the clock edges explicitly, the clock has a single pulse per period and a duty cycle of 50 percent, with a rising edge at the beginning of the period and falling edge in the middle, as shown in Figure 5-1.

Figure 5-1 Default Clock Waveform

To specify an asymmetrical or inverted waveform, use the `-rise` and `-fall` options to specify the times at which the rising and falling edges occur within the clock period, or use the `-waveform` option to specify the times at which successive clock edges occur.

For example, the following command creates a clock at the port named PH1. The clock has a period of 10.0, a rising edge at time 5.0, and a falling edge at time 9.5, as shown in Figure 5-2. The clock name is the same as the port name, PH1.

```
nt_shell> create_clock -period 10.0 -rise 5.0 -fall 9.5 PH1
1
```

Figure 5-2 Clock Waveform Example

The following command creates the same clock as the previous example, using the `-waveform` option instead of the `-rise` and `-fall` options:

```
nt_shell> create_clock -period 10.0 -waveform {5.0 9.5} PHI1
```

In the next example, the `create_clock` command creates a clock named PH2 on an input port named ck2, with a period of 10.0, a rising edge at 6.0, and a falling edge at 9.5. The clock name is different from the port name.

```
nt_shell create_clock -name PH2 -period 10.0 \
         -waveform { 6.0 9.5 } [get_ports ck2]
```

The following command creates a virtual clock named PH2 with a period of 10.0, a rising edge at 0.0, and a falling edge at 5.0. A virtual clock is not associated with any port or pin and can be used as a basis for specifying input delays and output delays at data ports.

```
nt_shell> create_clock -name PH2 -period 10.0 waveform {0.0 5.0}
```

To remove a clock, use the `remove_clock` command.

See Also

- Generated Clocks
- Pulse Clocks
- Multiple Clocks

Generated Clocks

A generated clock is a clock whose timing characteristics (period, waveform, and latency) depend on another clock in the design, called the master clock. If the master clock characteristics change, the generated clock characteristics change accordingly. For example, a divide-by-2 clock circuit produces a generated clock signal with half the frequency of the master clock. The latency of the generated clock is equal to the latency of the master clock at the clock divider plus the propagation delay of the clock generator circuit.

A generated clock does not inherit model transition index values from its master clock. When you create a generated clock, you should set the index values explicitly by using the `set_model_input_transition_indexes` command. If you do not set the indexes explicitly, the values come from the `model_default_input_transition_indexes` variable.

You can specify a generated clock on a boundary pin by using the `create_clock` command, just like any other clock. However, if you use the `create_generated_clock` command instead, NanoTime automatically handles any change in the master clock characteristics. In addition, the `create_generated_clock` command can be used on internal pins, but the `create_clock` command is restricted to boundary pins and ports.

For example, you can specify a divide-by-2 generated clock as follows:

```
nt_shell> create_generated_clock -name CLK2 \
          -source [get_ports SCLK] \
          -divide_by 2 [get_pins -of_objects FF1/Q \
          -filter “lib_pin_name==$link_transistor_gate_pin_name”]
```

Figure 5-3 shows the clock created by this command.

Figure 5-3 Divide-by-2 Clock Generator

In this example, the new clock is named CLK2. The master clock (the source of the generated clock) is port SCLK. FF1/Q is the output pin where the divide-by-2 clock is

generated and is the generated clock source object. The frequency of the generated clock is half that of the master clock, as specified by the `-divide_by 2` option; the period is therefore twice as long.

You specify two source objects: the generated source object (where the generated clock is defined) and the master source object (where the master clock is defined). The generated source object can be an input port, pin, or net. If a net is specified as a net, the driving pin of the net is considered to be the clock source object.

The master clock source object is typically specified with the `-source` option. However, if the `-source` option is not used, the master clock is a virtual clock, a clock with no source object. In that case, you must specify the name of the virtual clock by using the `-master_clock` option with the `create_generated_clock` command.

The following command creates a generated clock from a virtual master clock. Specifying the source pin is optional if the master clock is a virtual clock. The generated clock inherits the clock latency of the master clock.

```
nt_shell> create_generated_clock -master_clock ref_clk \
          -edges {1 2 3} [get_ports clk]
```

You define the timing relationship between the master clock and generated clock by using one of the following options:

- `-divide_by` (frequency divider)
- `-multiply_by` (frequency multiplier)
- `-edges` (edge-derived generated clock)

The `-divide_by` option defines a generated clock by dividing the frequency of the master clock by an integer (for example, a divide-by-2 clock divider). If the specified divide-by factor is a power of 2 (2, 4, 8, 16, ...), NanoTime uses only the rising edges of the master clock to determine the edges of the generated clock. For all other factors, NanoTime scales all the edge times of the master clock to determine the edge times of the generated clock.

The `-multiply_by` option defines a generated clock by multiplying the frequency of the master clock by an integer (for example, a multiply-by-2 phase-locked loop clock generator).

You can invert a frequency-divided or frequency-multiplied clock with the `-invert` option, as shown in the following example. This command creates a clock that is inverted with respect to clock CLK. Note that the `divide_by` option is required with the `-invert` option, even though divide by 1 means there is no change in the clock frequency.

```
nt_shell> create_generated_clock -divide_by 1 -invert \
          -source [get_pins CLK] [get_pins CLKINV]
```

The `-edges` option lists three edge numbers of the master clock that are used to trigger the edges of the generated clock. For example, consider the clock divider circuit shown in Figure 5-4.

Figure 5-4 Divide-by-3 Clock Generator

```
create_clock -period 2 -name SCLK [get_ports SCLK]
```

The generated clock signal DIV3A has a period three times longer than the master clock, with an asymmetric waveform.

To specify this waveform, enter the following command:

```
nt_shell> create_generated_clock -edges {1 5 7} \
          -name DIV3A -source [get_ports SCLK] \
           [get_pins U2/Q \
           “lib_pin_name==$link_transistor_gate_pin_name]
```

The `-edges {1 5 7}` option specifies that the rising, falling, and rising edges of the generated clock are triggered by the first, fifth, and seventh edges of the master clock, respectively. Both rising and falling edges of the master clock are counted, beginning with a rising edge as edge number 1.

If you are using the `-edges` option to generate a normal-edge clock (not a pulse clock), you can shift each edge of the generated clock by a specified amount of time by using the `-edge_shift` option. You should use the `-edge_shift` option only to represent the intentional operation of the clock generator circuit, not to represent clock latency.

The `set_clock_latency`, `set_clock_uncertainty`, `set_propagated_clock`, and `set_clock_transition` commands apply to generated clocks as well as standard clocks.

For generated clocks, NanoTime automatically computes the clock source latency if the master clock of the generated clock has propagated latency and no user-specified value for generated clock source latency exists. If the master clock is ideal and has source latency, and there is no user-specified value for the generated clock's source latency, then zero source latency is assumed.

To display information about both regular and generated clocks, use the `report_clock` command. To undo the effects of the `create_generated_clock` command, use the `remove_generated_clock` command.

Using Generated Clocks to Propagate Clocks Through Latches

By default, the output of a latch or flip-flop is a data signal. If the output of a latch or flip-flop is a clock signal driving the clock input of another sequential element, you must define the output as a clock using the `create_generated_clock` command.

The latch node of the latch (or slave latch node of a master-slave flip-flop) must also be defined as a clock signal to allow clock propagation from the clock pin to the output of the latch or flip-flop. To accomplish this, use the `mark_clock_network -force_propagation` command.

When the latch node becomes a clock signal, the structure no longer behaves as a latch topology. If you mark the topology with the `mark_latch` or `mark_flip_flop` command, logic conflicts block the clock propagation through the structure. Use the `erase_latch` or `erase_flip_flop` command to remove the topology markings, or avoid marking the structures in the first place.

See Also

- Standard Clocks
- Pulse Clocks
- Multiple Clocks

Pulse Clocks

A generated clock that consists of a sequence of short pulses is called a pulse clock. You can specify this type of clock by using the `-edges` option with a repeated digit in the list of edges. The repeated edge of the master clock triggers both the leading and trailing edges of the generated clock pulse.

For example, using the `-edges` option with an argument of `{1 1 3}` means that the rising and falling edges of the initial pulse of the generated clock are both triggered by the first edge of the master clock, and the next rising edge of the generated clock is triggered by the third edge of the master clock. This effectively means that the first pulse occurs at master clock edge 1 and the second pulse occurs at master clock edge 3, which effectively defines the period of the generated clock.

Using the same master clock edge twice in the option argument implies a pulse width of zero. In reality, the clock generator circuit is usually designed so that a circuit delay defines the pulse width. NanoTime does not analyze the circuit delay. Instead, you must use the `set_clock_latency` command to define the pulse width.

For example, consider the pulse clock circuit shown in Figure 5-5. Each rising edge of the master clock SCLK generates an active-high pulse on CLKP. Master clock edge number 1 triggers both the rising and falling edges of the first pulse, while edge number 3 triggers the second pulse and sets the pulse clock period. The pulse width is determined by the delay of the inverter chain, but you must specify the rise and fall latencies to represent this delay.

Specify the generated pulse clock CLKP as follows:

```
nt_shell> create_generated_clock -name CLKP -source [get_ports SCLK] \
          -edges {1 1 3} [get_pins -leaf -of U1/Z \
          -filter “lib_pin_name==$link_transistor_gate_pin_name”]
```

Set the rise and fall latencies to produce a pulse width of 0.2:

```
nt_shell> set_clock_latency -source -rise 0.0 [get_clocks CLKP]
nt_shell> set_clock_latency -source -fall 0.2 [get_clocks CLKP]
```

The clock pulses must have a positive pulse width. If you create a clock with active-low pulses, you must specify the clock latency to have a fall time which occurs before the rise time.

Figure 5-5 Pulse Clock Example

To specify a similar clock with active-low pulses instead of active-high pulses, modify the commands as follows:

```
nt_shell> create_generated_clock -name CLKP -source SCLK \
          -edges {1 3 3} [get_pins U1/Z]
nt_shell> set_clock_latency -source -rise 0.2 [get_clocks CLKP]
nt_shell> set_clock_latency -source -fall 0.0 [get_clocks CLKP]
```

You could create a pulse clock by defining a standard clock with closely spaced rise and fall times. However, specifying the pulse clock as a generated clock ensures the correct checking of delays between the master clock and generated clock domains.

In summary, to define a pulse clock, you need to specify its period, whether it is active-high or active-low, and whether it triggers from a master clock rising edge or falling edge. The `-edges` option allows you to specify those three things in a concise way:

- If the repeated digits in the option argument appear at the beginning of the list, then the pulse clock is active-high. If the repeated digits are at the end of the list, then the pulse clock is active-low.
- If the repeated digits are odd numbers, then the pulse trigger is a master clock rising edge. If the repeated digits are even, then the trigger is a falling edge.
- The pulse clock period is equal to the time difference between the master clock edges listed as the first and last values in the argument list.

For example, the `{1 1 3}` argument results in active-high pulses that are triggered on a master clock rising edge. The period is equal to the time difference between master clock edge 1 and master clock edge 3.

During path tracing, NanoTime sets fixed logic states on nets for simulation. However, it is not possible to sensitize a pulse clock net to a specific static logic state. To allow simulation of the circuit, you need to disable the logic checks on those nets with the `set_disable_logic_check` command. This action sets the value of the net attribute `logic_check_is_disabled` to `true`.

For example, suppose that a pulse generator has a net pgen1 that cannot be sensitized to a static logic 1, causing an error during path tracing. The following command disables logic checking of that net, allowing analysis of the design to proceed:

```
nt_shell> set_disable_logic_check [get_nets pgen1]
```

To cancel the effects of this command, use the `remove_disable_logic_check` command.

See Also

- Standard Clocks
- Generated Clocks
- Multiple Clocks

Multiple Clocks

When multiple clocks are defined for a design, the relationships between the clock domains depend on how the clocks are generated and how they are used in the design. A clock domain is a specific edge of a specific clock; one clock can therefore be the source of more than one clock domain.

The relationship between two clocks can be synchronous, asynchronous, or exclusive, as shown by the examples in Figure 5-6.

Figure 5-6 Synchronous, Asynchronous, and Exclusive Clocks

To analyze paths between different clock domains, you might need to use case analysis to exclude one or more clocks from consideration, or to specify only the clocks that are active in the current operating mode.

The following conditions are not supported:

- The simultaneous existence of multiple clocks on the same port or pin
- Multiple clocks feeding into a single generated clock

Synchronous Clocks

Two clocks are synchronous with each other if they share a common source and have a fixed phase relationship. Unless you block timing paths between clocks by using the `set_false_path` command, NanoTime assumes that two clocks are synchronous if there is a path with data launched by one clock and captured by the other clock. The clock waveforms are synchronized at time zero, as defined by the `create_clock` command.

For example, consider the following `create_clock` commands:

```
nt_shell> create_clock -period 4 -name CK1 -waveform {0 2}
nt_shell> create_clock -period 4 -name CK2 -waveform {1 3}
nt_shell> create_clock -period 6 -name CK3 -waveform {2 3}
```

NanoTime creates the clocks as specified in the commands, with the waveforms synchronized as shown in Figure 5-7. NanoTime adjusts the timing relationships to account for any specified or calculated latency or uncertainty.

In a design that uses these three clocks, there might be paths launched by one clock and captured by another clock. When such paths exist, to test all possible timing relationships between different clock edges, NanoTime internally expands the clocks to the least common multiple of all synchronous clock periods, thus creating longer-period clocks with multiple rising and falling edges.

For example, the three clocks in the example have periods of 4, 4, and 6. The least common multiple of these periods, known as the base period, is 12. To analyze the paths that cross the clock domains, NanoTime internally expands the clocks by repeating them over the base period. The resulting clock waveforms are shown in Figure 5-8. Each expanded clock has a period of 12.

NanoTime checks timing paths between all edges in the expanded clocks. For example, the most restrictive setup check between a falling edge of CK2 and a rising edge of CK3 is from time=7 to time=8, as shown by the dashed arrow in Figure 5-8.

Figure 5-7 Original Synchronous Clock Waveforms

Figure 5-8 Expanded Clock Waveforms

Asynchronous Clocks

Two clocks are asynchronous if they do not have a timing relationship in the design. For example, a free-running, on-chip oscillator is asynchronous with a system clock signal coming into the chip from the outside. Clock edges in the two clock domains can occur at any time with respect to each other.

You can declare that all paths between the two clocks are false paths to prevent path tracing on those paths. For example,

```
nt_shell> set_false_path -from [get_clocks CK1] -to [get_clocks CK2]
nt_shell> set_false_path -from [get_clocks CK2] -to [get_clocks CK1]
```

Exclusive Clocks

Two clocks are exclusive if they do not interact with each other. For example, a circuit might multiplex two different clock signals onto a clock line, one a fast clock for normal operation and the other a slow clock for low-power operation. Only one of the two clocks is enabled at any given time, so there is no interaction between the two clocks.

To prevent NanoTime from spending time checking the interaction between exclusive clocks, you need to analyze the circuit behavior one clock at a time by using case analysis, or by creating only the clocks that are active in a given operating mode.

For example, consider a clock selection circuit like the one shown in Figure 5-9. The input signal SEL enables either clock CK1 or CK2.

Figure 5-9 Clock Selection Circuit

To analyze one clock at a time using case analysis, set one clock to an inactive state while allowing the other to be analyzed. For example, to analyze the case where SEL=0 and CK1 is enabled, use the following command:

```
nt_shell> set_case_analysis 0 [get_ports SEL]
```

An alternative method is to use the `create_clock` command to define only the clock that is active for the current operating mode, leaving the other clock undefined.

Timing Checks with Multiple Clock Domains

NanoTime evaluates timing checks using all clock domains on a reference pin. Some of the timing checks might never occur. Approaches to removing timing checks are listed here in order from the least specific to the most specific.

- Specifying false paths

  The `set_false_path` command prevents all path tracing and timing check evaluation on specified paths. Using this command is likely to remove many more timing checks throughout the circuit than just the ones at the cell with multiple clock domains.

- Eliminating timing checks for entire paths

  The `set_no_timing_check_path` command blocks timing check evaluation while allowing path tracing beyond the timing check. For example, in the case of a circuit with two clock domains that consist of the rising edge of clock clk1 and the falling edge of clock clk1, the following commands remove the timing checks between the domains:

  ```
  set_no_timing_check_path -rise_from clk1 -rise_to clk1
  set_no_timing_check_path -fall_from clk1 -fall_to clk1
  ```

  This approach eliminates all timing checks between the two domains. However, it might block some needed timing checks in addition to unwanted ones.

- Eliminating specific timing checks

  You can use all of the path selection options of the `set_no_timing_check_path` command to be very specific about the timing checks to eliminate.

  NanoTime first finds paths that match the `-from`, `-through`, and `-to` options in the command. The tool evaluates all timing checks up to the point where the options are matched. Timing checks for the next latch in the path (or other topology that includes timing checks) are not evaluated. More than one timing check might be eliminated, depending on the specific topology. Path tracing continues beyond the affected topology and all timing checks in the rest of the path are evaluated.

After all of the invalid cross-clock timing checks are removed, NanoTime checks transparency for each remaining reference clock domain. If any reference clock domain is not transparent, the tool considers all domains to be nontransparent. Therefore, it is essential to remove all invalid cross-clock timing checks so that transparency for each path arriving at a timing check is evaluated against a minimal set of reference clock domains.

Clock Latency

Latency is the amount of time that a clock signal takes to get from its ideal-waveform origin point to a sequential element inside the design. Clock latency consists of two parts, source latency and network latency.

Source latency is the amount of time the clock signal takes to get from its ideal-waveform origin point to the clock definition point in the design, often called the clock source object. The clock source object is the port or pin specified in the `create_clock` command. Network latency is the amount of time the clock signal takes to get from the clock source object to sequential elements in the design.

For each sequential element in the design, NanoTime uses either propagated or ideal clocking to determine the latency. For propagated clocking, NanoTime calculates the cumulative delays along the clock network, including the effects of wire parasitics. For ideal clocking, NanoTime uses latency values set on objects with the `set_clock_latency` command, or zero latency where no latency value has been set. Compare Figure 5-10 and Figure 5-11.

Figure 5-10 Propagated Latency

Figure 5-11 User-Specified Network Latency

Propagated and Ideal Clocking

By default, NanoTime assumes that all clocks are propagated. This approach is different from the PrimeTime tool, which uses ideal clocking by default. NanoTime is primarily a transistor-level timing tool that calculates delays with SPICE-like accuracy from basic circuit elements, whereas PrimeTime is primarily a gate-level analysis tool that uses library-defined gate delays.

You might want to use ideal clocking in NanoTime, for example, to annotate delays from a delay data file in SDF format. To create an ideal clock in NanoTime, set the `timing_all_clocks_propagated` variable to `false` before you use the `create_clock` command to create the clock. To use ideal clocking for specified clocks, ports, or pins, use the `set_clock_latency` command and set the latency values. To set ideal clocking on clocks, ports, or pins without applying a latency value, use the `remove_propagated_clock` command. This results in zero latency if no latency values were set before. To restore propagated clocking behavior, use the `set_propagated_clock` command.

Source Latency and Network Latency

The `set_clock_latency` command can be used to specify either the amount of source latency for a clock or the amount of network latency for sequential elements in the design.

Using the `-source` option causes the command to specify source latency for a clock. In that case, the command must specify the name of a clock or a clock source object in the design, and the source latency delay value to be applied.

If you do not use the `-source` option, the command sets the network latency for sequential elements in the design. The command must specify a latency delay value and a list of clocks, ports, or pins. Specifying a clock makes the clock ideal and sets the network latency of all sequential elements clocked by that clock. Specifying a port or pin sets the network latency of all objects in the transitive fanout of the port or pin, and makes those objects use ideal clocking.

You can use source latency to model off-chip clock latency when the clock generation circuit is not part of the current design. For a generated clock, you can use source latency to model the delay from the master clock to the generated clock.

For a generated clock, NanoTime automatically computes the clock source latency if the master clock has propagated latency and there is no user-specified value for the generated clock source latency. If the master clock is ideal and has source latency, and if there is no user-specified source latency value for the generated clock, then zero source latency is assumed. For more information about generated clocks, see Generated Clocks.

To undo the effects of the `set_clock_latency` command, use the `remove_clock_latency` command.

To report network or source latency set on a clock, use the `report_clock` command with the `-skew` option. To report network or source latency set on a port or pin, use the `get_attribute` command. For example,

```
nt_shell> get_attribute [get_ports clk3] clock_latency_fall_max
```

Options for Latency

By using the `-rise` or `-fall` option with the `set_clock_latency` command, you can specify the latency value for just rising edges or just falling edges of the clock signal. Without these options, the latency value applies to both rising and falling edges. For source latency, the edge type applies to the clock signal at the clock source port or pin. For network latency, the edge type applies to the signal arriving at the sequential element, which can be different from the transition at the clock source because of logical inversions along the clock path.

For example, the following commands specify a rising-edge latency of 1.2 and a falling-edge latency of 0.9 for the clock named CLK1:

```
nt_shell> set_clock_latency 1.2 -rise [get_clocks CLK1]
nt_shell> set_clock_latency 0.9 -fall [get_clocks CLK1]
```

Because the `-source` option is not used, each command sets the network latency of clock CLK1, which applies to all sequential elements clocked by CLK1. The edge type (rising or falling) refers to the signal transition arriving at the sequential element, including any inversions along the clock path.

Similarly, you can optionally use the `-min` and `-max` options to set different latency values for minimum and maximum operating conditions.

To specify a worst-case range of source latency values, use the `-early` or `-late` option together with the `-source` option. NanoTime uses the early and late latency values to determine the earliest and latest possible clock arrival times, respectively. For example, for a setup check, it uses the late value for data launch and the early value for data capture.

For example, consider source latency that varies from 1.5 to 2.5 ns, as shown in Figure 5-12.

Figure 5-12 External Source Latency

To specify this type of source latency, use commands such as the following:

```
nt_shell> create_clock -period 10 [get_ports CLK]
nt_shell> set_clock_latency 1.5 -source -early [get_clocks CLK]
nt_shell> set_clock_latency 2.5 -source -late [get_clocks CLK]
```

NanoTime uses the more conservative source latency value (either early or late) for each startpoint and endpoint clocked by that clock. For setup analysis, it uses the late value for each startpoint and the early value for each endpoint. For hold analysis, it uses the early value for each startpoint and the late value for each endpoint.

Figure 5-13 shows the early and late timing waveforms and the clock edges used for setup and hold analysis in the case where the startpoint and endpoint are both clocked by CLK.

Figure 5-13 Early and Late Source Latency Waveforms

See Also

- Clock Uncertainty
- Clock Transition Time

Clock Uncertainty

Uncertainty is the amount of variation in the arrival times of a clock edge. An ideal clock has an uncertainty of zero. A real clock has some variation in the arrival time of each edge. NanoTime uses the worst possible variation in clock arrival times to perform setup and hold checks.

Use the `set_clock_uncertainty` command to specify either simple uncertainty (between different edges of the same clock) or interclock uncertainty (skew between two different clocks). Interclock uncertainty is important when the clock used to launch data at the path startpoint is different from the clock used to capture data at the path endpoint.

- For simple uncertainty, specify a list of clocks, ports, or pins. If you specify a clock, the uncertainty value applies to all sequential elements clocked by that clock. If you specify a port or pin, the uncertainty value applies to all sequential elements in the fanout of the port or pin.
- For interclock uncertainty, use the `-from`, `-rise_from`, or `-fall_from` options to specify the source clock and the `-to`, `-rise_to`, or `-fall_to` options to specify the

destination clock. The uncertainty applies to the paths that start from the source clock domain and end in the destination clock domain.

For interclock uncertainty, you must specify all possible interactions of clock domains. For example, for paths between CLKA and CLKB, you must specify the uncertainty for both directions, even if the uncertainty values are the same.

If no interclock uncertainty is specified for a path, the value for simple uncertainty is used. A specification for interclock uncertainty has higher precedence than a conflicting command that specifies simple uncertainty.

To view the current clock uncertainty settings, use the `report_clock -skew` command. To remove uncertainty values, use the `remove_clock_uncertainty` command.

The following commands specify that all paths ending at elements clocked by CLK have setup uncertainty of 0.65 and hold uncertainty of 0.45:

```
nt_shell> set_clock_uncertainty -setup 0.65 [get_clocks CLK]
nt_shell> set_clock_uncertainty -hold 0.45 [get_clocks CLK]
```

The following commands specify interclock uncertainty within and between clock domains PHI1 and PHI2. Both directions must be specified separately.

```
nt_shell> set_clock_uncertainty 0.4 -from PHI1 -to PHI1
nt_shell> set_clock_uncertainty 0.4 -from PHI2 -to PHI2
nt_shell> set_clock_uncertainty 1.1 -from PHI1 -to PHI2
nt_shell> set_clock_uncertainty 1.1 -from PHI2 -to PHI1
```

The following example contains two conflicting `set_clock_uncertainty` commands, one for simple uncertainty and one for interclock uncertainty. The interclock uncertainty value of 2.0 takes precedence for paths from CLKB to CLKA:

```
nt_shell> set_clock_uncertainty 5.0 [get_clocks CLKA]
nt_shell> set_clock_uncertainty 2.0 \
          -from [get_clocks CLKB] -to [get_clocks CLKA]
```

In the following example, the first command specifies simple uncertainty for paths from CLKA. The second command specifies interclock uncertainty of 2.0 for paths from CLKB to CLKA. If there are paths between CLKA and other clocks for which interclock uncertainty has not been specified, the simple uncertainty for CLKA is used (in this case, 5.0).

```
nt_shell> set_clock_uncertainty 5.0 [get_clocks CLKA]
nt_shell> set_clock_uncertainty 2.0 \
          -from [get_clocks CLKB] -to [get_clocks CLKA]
```

See Also

- Clock Latency
- Clock Transition Time

Clock Transition Time

The transition time (slew) of a signal is the amount of time it takes the signal to change from one logic state to the other.

If you are analyzing a design for which the clock tree has not yet been implemented, the clock transition times calculated from driver and load considerations are not accurate. For example, if a single device is driving 100,000 latches without clock tree buffers, the large capacitive load on the driver causes an extremely long transition time.

In this case, you can perform an analysis by making the clock ideal and setting the estimated clock transition time with the `set_clock_transition` command. For example,

```
nt_shell> remove_propagated_clock [get_clocks PH2]
nt_shell> set_clock_latency 0.8 [get_clocks PH2]
nt_shell> set_clock_transition 0.2 [get_clocks PH2]
```

The `remove_propagated_clock` command makes clock PH2 an ideal clock rather than a propagated clock, in which case the clock delays and transition times are no longer calculated from driver and load considerations. The `set_clock_latency` command sets the network latency (the delay from the clock source to sequential elements), while the `set_clock_transition` command specifies a transition time of 0.2 time units.

If the clock tree has been implemented, NanoTime can calculate the clock network delays and transition times from the drivers and loads. Use the `set_input_transition` command to specify the transition time explicitly or the `set_drive` command to specify the drive resistance of the external device that is driving the clock port. If you use both the `set_drive` and `set_input_transition` commands, NanoTime models the external driver as a voltage ramp driving the resistor and connected to the port.

For example, to set the transition time at the input of clock port CK1 to 0.2 time units, use the following command:

```
nt_shell> set_input_transition 0.2 [get_ports PH2]
```

To set the drive resistance of the external device that is driving clock port PH2 to 20 resistance units, use the following command:

```
nt_shell> set_drive 20 [get_ports PH2]
```

If no transition time is set for a port (including a clock port), NanoTime uses the transition times determined by the `sim_transition_max_fall`, `sim_transition_max_rise`, `sim_transition_min_fall`, and `sim_transition_min_rise` variables.

See Also

- Clock Latency
- Clock Uncertainty

Clock Networks

An accurate clock network is a requirement for recognition of other sequential topology structures. At the `match_topology` command, NanoTime propagates the clock network from the clock sources through inverters, transistors that are turned on, clock-gating structures, and enabled combinational logic gates.

Valid clock endpoints are as follows:

- Latch clock pin input
- Register file clock pin input
- Precharge clock pin input
- Reference pin of a user-defined timing check
- Output port of the design
- Input pin of subsequent generated clock
- Clock pin of an extracted timing model
- Channel-connected block with no fanout
- Port, pin, or net identified with the `mark_clock_network -stop_propagation` command

NanoTime automatically stops clock propagation for some conditions. A pin is defined as an invalid endpoint when any of the following conditions exists:

- The pin resides within the transitive fanout of a clock or net having the `is_clock` attribute.
- The pin drives a channel-connected block without a clock net output.
- The pin does not meet any of the criteria for a valid endpoint.

The mark_clock_network Command

The NanoTime tool automatically recognizes many clock topologies. However, you might need to use your knowledge of the design to ensure correct clock network behavior. The `mark_clock_network` command marks objects of a design with clock-related properties.

The `mark_clock_network` command applies the actions specified by the command options to design objects in the argument list. The object names can represent ports, pins, or nets.

The `erase_clock_network` command cancels the effect of the `mark_clock_network` command and restores the default behavior.

Forcing or Stopping Clock Propagation

Use the `-force_propagation` option with the `mark_clock_network` command to help the NanoTime tool to propagate the clock network. For example, this option is needed when a clock network contains latches along the control or reset paths that must be toggled to create the correct waveform. The latches must be marked as clocks to be included in the clock network.

Use the `-stop_propagation` option to stop clock propagation at an object. Any clock paths beyond that point are not considered part of the clock network, which means that no timing checks can be performed for sequential devices beyond that point.

Using Attributes to Analyze Clock Nets

To analyze the clock network, you can use the `set_requires_clock` command to set the `is_requires_clock` attribute to `true` for nets that you know to be part of the clock network. During clock propagation, NanoTime sets a related net attribute called `is_clock` to `true` for all nets that the tool determines are part of the clock network. When you execute the `report_clock_network` command, the tool checks the two attributes. Nets with the `is_requires_clock` attribute set to `true` and the `is_clock` attribute set to `false` are reported as errors.

At the `report_clock_network` command, Invalid clock network endpoints are also analyzed further to determine if a net with the `is_requires_clock` attribute exists in the transitive fanout of the clock endpoint; if such a net exists, the net is reported as a “possible clock gate.”

Merging Clock Nets

For a high-fanout net such as a clock net, the clock tree might be implemented with multiple parallel drivers. To merge the behavior of multiple nets in the clock tree for timing analysis, list the net names in the `set_merged_nets` command, as in this example:

```
nt_shell> set_merged_nets {c1 c2 c3 c4}
```

For a path starting from any one of these nets, NanoTime eliminates all but one of the parallel timing arcs for delay analysis. For a path ending at one of these nets, merging has no effect. Also, merging does not affect pattern matching, structure recognition, or the netlist.

To report nets that have been merged, use the `report_merged_nets` command. To cancel the effects of the `set_merged_nets` command and restore independent behavior for one or more of the merged nets, use the `remove_merged_nets` command.

Clock Gates

A clock gate is a topology structure whose inputs include one or more clock signals and whose output is also a clock signal. Recognizing the output as a clock signal is important for the analysis of downstream sequential topologies.

NanoTime automatically recognizes clock-gating structures when a clock signal enters a logic gate, as shown in Figure 5-14. During timing analysis, the tool performs setup and hold timing checks to ensure that the enable signals arrive before the applicable clock edges, resulting in a gated-clock output signal that is free of glitches and clipped pulses.

Timing checks are attached only to the enable pins. No timing checks are performed between clock inputs.

Figure 5-14 Clock Gate Example

Simple clock gates use enable signals to control whether an input signal propagates to the output. Reconvergent clock gates such as pulse generators and pulse shapers use two or more inputs from a single clock source to generate complex clock signals at the output.

NanoTime provides the following methods for analyzing reconvergent clock gates:

- Timing-based analysis
- Topology-based analysis

The following conditions must be met to recognize clock-gating topologies automatically:

- For topology-based analysis, a clock gate must consist of a channel-connected block with at least one PMOS transistor and one NMOS transistor driven by the same enable input net. Either the NMOS or PMOS transistors must form a stack in a NAND or NOR configuration. Constant value clock inputs and enable signals are ignored.

  For timing-based analysis, an enable signal is not required. Consequently, a larger number of clock gates might be recognized automatically.

- There must be at least one clock input, which must be in the clock network.
- Two or more clock inputs are allowed if the clock inputs originate from the same clock source definition and belong to the same clock domain, making the gate a pulse generator or pulse shaper.
- If any feedback loop devices drive the clock-gate output net, you must force clock-gate recognition by using the following commands:

  ```
  mark_clock_network -force_propagation clock_gate_output
  mark_feedback -transistors feedback_loop_device
  ```

- The propagation must not be stopped and the output must not be forced to a specific logic state, as can happen with the following commands:

  ```
  mark_clock_network -stop_propagation clock_gate_output
  set_case_analysis [1|0] [get_pins -leaf -of_objects clock_gate_output]
  ```

See Also

- Reconvergent Clock Gates
- Reporting Clocks and Clock Networks

Manually Marking Clock Gates

If NanoTime does not automatically recognize a clock gate, use the `mark_clock_gate` command to manually specify the structure.

Figure 5-15 shows an example of a clock-gating circuit that NanoTime recognizes automatically. The command to specify this clock gate manually is as follows:

```
nt_shell> mark_clock_gate -clock {N1.G P1.G} -output CLKG \
                          -positive_enable {N2.G P2.G]
```

The options that specify the input clock pin and the output (gated-clock) net are required.

To cancel the `mark_clock_gate` command or to override recognition of an automatically detected clock gate, use the `erase_clock_gate` command.

Figure 5-15 Clock-Gating Circuit

By default, NanoTime does not consider logic values set by case analysis during clock propagation. Set the `topo_clock_propagation_strict_logic_check` variable to `true` to change the default behavior. For example, if clock and data values are multiplexed, you can define a case that selects the clock signal to propagate to the multiplexer output.

Clock Gate Analysis

All clock gates, whether manually marked or automatically recognized, undergo analysis to determine which paths to trace. After analysis, clock gates are classified as either resolved or unresolved, as follows:

- Resolved

  If NanoTime selects specific paths through the clock gate to use for worst-case timing analysis, the clock gate is said to be resolved. All other clock paths are ignored for timing analysis.

- Unresolved

  If the tool cannot determine the best paths to use for timing analysis, the clock gate is said to be unresolved. All clock paths are propagated through the clock gate, which requires more runtime and might lead to pessimistic results.

For complex clock gates, the NanoTime tool might have difficulty identifying the controlling pins (the input clock pins that define the critical paths through the clock gate). You can identify the controlling pins manually by using the `mark_clock_gate` command with the

`-max_rise_controlling_pin`, `-min_rise_controlling_pin`, `-max_fall_controlling_pin`, and `-min_fall_controlling_pin` options.

These options provide a method to circumvent the clock gate analysis process for a specific clock gate when you know how signals propagate through the structure. You assume responsibility for the correctness of the definitions.

If you use any of these options, you must use all of them. These options dictate how paths are traced through the clock gate. Therefore, the clock gate is resolved by definition. Because you are defining the paths to be traced, variable settings that affect global clock gate analysis are ignored for the specified clock gate.

Reconvergent Clock Gates

NanoTime automatically recognizes circuits that have more than one input clock signal as reconvergent clock-gating structures known as pulse generators or pulse shapers. Figure 5-16 shows a simple example.

Figure 5-16 Simple Reconvergent Clock-Gating Circuit

In this example, the clock signals reconverge at the NAND gate inputs blk and clk. For the circuit to be automatically recognized as a clock-gating structure, the reconvergent signal must not intersect any other clock signal and must belong to the same clock domain as the original clock signal.

The `topo_clock_gate_allow_reconvergent_clocks` variable affects this recognition; its default is `true`. When NanoTime recognizes a clock gate, logic checking on the clock input of the clock-gating structure is disabled. If you want finer control over clock-gating structures, you can optionally set the `topo_clock_gate_allow_reconvergent_clocks` variable to `false` and manually mark the clock-gating circuits. In this case, you must manually disable logic checking on the clock signals.

NanoTime can analyze reconvergent clock gates in two different ways:

- Timing-based analysis

  The tool analyzes the timing of multiple clock signals arriving at a clock gate to determine which paths controls the switching of the output signal. Depending on the logic of the clock gate, the controlling path for maximum-delay analysis might not be the latest-arriving path and the controlling path for minimum-delay analysis might not be the earliest-arriving path.

- Topology-based analysis

  The tool chooses the paths most likely to be the worst-case maximum-delay and minimum-delay paths based on the number of topology features in each path, under the assumption that paths with more levels (more channel-connected blocks) have longer delays.

See Also

- Clock Gates
- Reporting Clocks and Clock Networks

Timing-Based Reconvergent Clock Gate Analysis

To enable timing-based analysis, set the `topo_clock_gate_timing_resolution` variable to `true`. The setting is global; you cannot select the analysis type for specific clock gates. This variable overrides the `topo_clock_gate_allow_reconvergent_clocks` variable.

When timing-based reconvergent clock-gate analysis is enabled, NanoTime determines which NMOS and PMOS devices control the switching of the clock gate output. The tool propagates only the paths through the controlling pins.

Timing-based analysis provides less pessimistic results than topology-based analysis for clock shaper circuits that are designed with standard series-parallel transistor configurations. In addition, a larger number of clock gates might be automatically recognized because timing-based analysis allows clock gates that do not have enable signals.

Timing-based analysis is not attempted for pulse generators or for manually marked clock gates. Even if timing-based analysis is enabled, NanoTime uses topology-based analysis in those cases. For designs with unusual arrangements such as bridge circuits, topology-based analysis might provide better results than timing-based analysis.

If NanoTime can determine the controlling paths for a specific clock-gating shaper circuit, the clock gate is resolved and the `clock_gate_type` attribute is set to `timing_resolved`. Otherwise the attribute is set to `timing_unresolved`. Timing-based analysis might be beneficial even for unresolved clock gates due to the availability of accurate delays for at least some of the path conditions.

Some conditions that cause clock gates to be unresolved are as follows:

- The clock gate contains bidirectional devices or bridge circuits.
- Different enable signals control different clock net inputs.
- Clock gates or library cells appear in the clock input paths and the `topo_clock_gate_strict_checking` variable is set to `true`.
- Arrival times are missing at one or more clock inputs.

If a clock gate is resolved, the names of the controlling pins are stored in four attributes associated with the clock-gate topology object: the `controlling_nmos_pin_max`, `controlling_nmos_pin_min`, `controlling_pmos_pin_max`, and `controlling_pmos_pin_min` attributes.

If the clock gate is unresolved, all clock pins remain active and some of these attributes might not be set.

Topology-Based Reconvergent Clock Gate Analysis

In topology-based clock gate analysis, NanoTime classifies clock gates into four categories:

- Resolved pulse generators
- Resolved pulse shapers
- Unresolved pulse generators
- Unresolved pulse shapers

The `clock_gate_type` attribute is set to one of the following values: `pulse_generator`, `pulse_shaper`, `pulse_shaper_unresolved`, or `pulse_generator_unresolved`.

Resolved Pulse Generators and Pulse Shapers

For NanoTime to automatically recognize a reconvergent clock-gating structure as a resolved pulse generator or pulse shaper, the following conditions must be met:

- Only two unique clock nets control the clock gate; they must originate from the same upstream clock net.
- The two clock nets belong to the same clock domain.
- The transistors of the clock gate are part of the same channel-connected block.
- The two clock nets control a stack of NMOS transistors or PMOS transistors.

When an upstream logic gate or library cell blocks a clock input to a reconvergent clock gate, NanoTime classifies the clock gate as an unresolved pulse shaper or unresolved pulse generator. Performing this strict checking guarantees pessimism. However, you can relax the requirement by setting the `topo_clock_gate_strict_checking` variable to `false` (the default is `true`). When you do this, NanoTime performs more aggressive clock gate recognition and classifies the clock gate as a resolved pulse shaper or resolved pulse generator.

NanoTime analyzes the number of simulation levels (channel-connected blocks) from the common clock net to the clock gate as follows:

- If the number of levels is an odd value for one clock input and an even value for the other clock input, the structure is a pulse generator.
- If the number of levels is an even value for both clock inputs or an odd value for both, but the two values are different, the structure is a pulse shaper.

Unresolved Pulse Generators and Pulse Shapers

Other reconvergent clock structures can be automatically recognized as clock-gating structures when some of the conditions are relaxed. These structures are unresolved pulse generators and unresolved pulse shapers.

Unresolved pulse generators have the following properties:

- The clock gate has three or more inputs originating from the same clock source definition.
- The number of simulation levels from the common clock net to each of the clock-gate inputs is a mix of even and odd values.
- The transistors of the clock gate are part of the same channel-connected block.
- The clock nets control a stack of NMOS transistors or PMOS transistors.

Unresolved pulse shapers have the following properties:

- The clock gate has two or more inputs originating from the same clock source definition.
- The number of simulation levels from the common clock net to each of the clock-gate clock inputs is the same (it can be either an even or odd number).
- The transistors of the clock gate are part of the same channel-connected block.
- The two or more clock nets control a stack of NMOS transistors or PMOS transistors.

For both types of unresolved clock-gating structures, NanoTime propagates all clock paths through the clock gate, producing a pessimistic output waveform.

You should attempt to reduce the pessimism of the clock-gate output waveform by guiding the critical clock paths through the clock gate, using commands such as the following:

```
mark_instance -dont_search_thru_gate \
               clock_cells_driven_by_noncritical_paths
set trace_disable_switching_net_logic_check true
set_false_path -through clock_gate_input_pin \
               -through clock_gate_output_pin
```

Controlling Reconvergent Clock Gate Analysis

By default, NanoTime identifies the possibility that an upstream logic gate can block a clock input to a downstream reconvergent clock gate. When this happens, the tool labels the downstream reconvergent clock gate as an unresolved pulse shaper or unresolved pulse generator to guarantee pessimism.

To disable this feature, set the `topo_clock_gate_strict_checking` variable to `false` (the default is `true`). In this case, NanoTime performs more aggressive clock gate recognition and marks any such downstream clock gate to be a resolved pulse shaper or resolved pulse generator.

Both variable settings might be correct for some clock gates and incorrect for others, especially for clock gates that have noninverting logic or a timing model in one or more of the reconvergent paths. You can refine the analysis of individual clock gates as follows:

- When the `topo_clock_gate_strict_checking` variable is `true`, some clock gates might be incorrectly labeled as unresolved. To remove strict checking on an individual clock gate and allow it to be resolved, use the `mark_net` command with the `-clock_gate_checking non_strict` option as follows, where *clock_out* is the clock gate output net:

  ```
  mark_net clock_out -clock_gate_checking non_strict
  ```

- When the `topo_clock_gate_strict_checking` variable is `false`, some clock gates might be incorrectly labeled as resolved. To force strict checking on an individual clock gate, which might cause it to become unresolved, use the `mark_net` command with the `-clock_gate_checking force_strict` option as follows:

  ```
  mark_net clock_out -clock_gate_checking force_strict
  ```

To remove a net-specific designation, use the `-clock_gate_checking` option with the `erase_net` command.

Reporting Clocks and Clock Networks

Because clocks are essential to most phases of timing analysis, many reports within NanoTime contain information about clock objects, networks, and delays. However, several reports focus specifically on clocks. To report the clocks that have been created and their characteristics, use the `report_clock` command. To review or debug the clock network, use the `report_clock_network` command. To get arrival time information at sequential pins in the design or along the clock network, use the `report_clock_arrivals` command.

See Also

- Clock Gates
- Reconvergent Clock Gates

The report_clock Command

By default, the `report_clock` command shows the clock name, period, edge times, clock attributes, and clock source for each clock in the design. For example,

```
nt_shell> report_clock
...
Attributes:
    p - Propagated clock
    G - Generated clock
    H - Pulse clock, high
    L - Pulse clock, low
    D - Differential clock

Clock      Period Waveform          Attrs Sources
-------- -------- ----------------- ----- ------------
MCLK        4.000 {0.000 2.000}     p
clk1        4.000 {0.000 2.000}     p     clk1
clk2        4.000 {1.000 3.000}     p     clk2
clk3        4.000 {2.000 0.000}     p     clk3
```

The numbers in the Waveform column indicate the times at which the rising and falling edges of the clock occur.

The letters in the Attrs column indicate the attributes of the clock (propagated, generated, pulse clock high, or pulse clock low). A propagated clock uses calculated rather than ideal delays. A pulse clock is a clock whose leading and trailing edges are triggered by the same event, but with different delays from the common triggering event.

The Sources column indicates the source object specified at the time of clock creation.

To get a report on clock latency and uncertainty for an ideal (not propagated) clock, use the `-skew` option of the `report_clock` command. For example,

```
nt_shell> report_clock -skew
...
              Min Rise Min Fall   ...          Hold       Setup
Object         Latency  Latency   ...   Uncertainty Uncertainty
------------ -------- --------   ---   ----------- -----------
MCLK               --       --   ...         0.200       0.200
clk1               --       --   ...         0.200       0.200
clk2               --       --   ...         0.200       0.200
clk3               --       --   ...         0.200       0.200
```

To restrict the scope of the report to specific clocks, list the clocks in the `report_clock` command. For example,

```
nt_shell> report_clock MCLK
...
Clock      Period Waveform          Attrs Sources
-------- -------- ---------------- ----- ------------
MCLK        4.000 {0.000 2.000}     p
```

The report_clock_network Command

The `report_clock_network` command can help you to identify and configure a clock network. It can be used only after the `match_topology` command.

To analyze the clock network, you can set the `is_requires_clock` attribute on nets to `true` by using the `set_requires_clock` command. During clock propagation, NanoTime sets a related net attribute called `is_clock` to `true` for all nets that it determines are part of the clock network. When you execute the `report_clock_network` command, NanoTime checks the two attributes. Nets with the `is_requires_clock` attribute set to `true` and the `is_clock` attribute set to `false` are reported as errors.

By default, the command reports a summary for each clock in the design. The summary includes the number of clock nets, clock gates, valid endpoints, and invalid endpoints.

The `-from` option can be used to restrict the report to a single clock or a subsection of a single clock network. If a net is specified, then the report considers only that net and the transitive fanout from that net. If a clock is specified, then the transitive fanout of all source pins of that clock is used.

The `-errors` option reports the details of all errors found in every clock network. By default, a summary of errors is listed. The summary includes the number of invalid endpoints in every clock network. Also, there might be some nets that are not part of the clock network but should be, perhaps because they have the `is_requires_clock` attribute; these nets are reported as “non-clock specific errors.” You can use the `-errors` option in conjunction with the `-from` option to investigate errors from a single clock port or net.

The `-verbose` option reports detailed information about the errors. For invalid endpoints, the channel-connected block driven by the endpoint is further analyzed. If the driven channel-connected block contains a net with an attribute indicating a potential storage node, then channel-connected block is listed on the report as a “possible missing sequential topology.” If the channel-connected block has a net with the `is_requires_clock` attribute in its transitive fanout, then it is reported as a “possible missing clock gate.” For a nonclock specific error, the reasons why the net should be a clock are listed.

An example of a clock network report is as follows:

```
****************************************
Report : clock network
Design : ALU
Version: G-2012.06
Date   : Tue Aug 14 12:07:11 2012
****************************************

Clock network:                MCLK
Number of invalid endpoints :    0

Clock network:                clk1
Number of invalid endpoints :    0

Clock network:                clk2
Number of invalid endpoints :   32

Invalid end point pins
---------------------------------------------------------------------
Xaddsub.Xadder.Xla1.Xlgen.XC1.Mp0.G   missing sequential topology
Xaddsub.Xadder.Xla1.Xlgen.XC2.Mp0.G   missing sequential topology
Xaddsub.Xadder.Xla1.Xlgen.XC3.Mp0.G   missing sequential topology
...
```

To investigate a specific net within the clock network, specify a single net for the *`nets`* argument. If the net is not part of the current clock network, then the transitive fanin of the net is searched, and a list of clock network endpoints or primary inputs with fanout to the current net is displayed. The points found are sorted by the fewest number of traversed channel-connected blocks. The `-from` option cannot be used to generate a report for multiple nets.

If the net is part of the clock network, then a back propagation through channel-connected blocks of the clock network occurs and the driving clock ports are displayed.

If the `-verbose` option is used, then the full channel-connected block paths are also displayed when showing transitive fanin points of the report.

The Clock-Gate Topology Report

If NanoTime recognizes clock-gating structures, the topology report generated with the `report_topology -structure_type clock_gate -verbose` command includes the clock gate classification.

The following codes indicate the type of clock gate:

- p – resolved pulse generator
- s – resolved pulse shaper
- np – unresolved pulse generator
- ns – unresolved pulse shaper
- ts – timing resolved shaper
- tn – timing unresolved shaper

The report_clock_arrivals Command

The `report_clock_arrivals` command reports the arrival times of clock signals at sequential device pins in the design or along intermediate points of a full clock tree. Any time after the `check_design` command, you can use the `report_clock_arrivals` command to view a list of the clock nets. However, to see the actual arrival times in the report, you must run the `trace_paths` command first.

If a net sees multiple clock domains, it is reported on multiple lines, one for each clock.

The report lists the arrival times for clock nets in the design, including intermediate nets on clock paths. The report includes the “Min Rise,” “Min Fall,” “Max Rise,” and “Max Fall” arrival times and the clock source (with attributes, if applicable) as shown here:

```
...
Attributes:
   H - Pulse clock with high pulse
   L - Pulse clock with low pulse
   D - Differential clock

Net          Min Rise  Min Fall  Max Rise  Max Fall Period Attrs Clock
----------- --------- --------- --------- --------- ------ ----- ------
Xsreg.ck10      3.549     0.322     3.628     0.423  6.000       clk2n
Xbreg.ck10      4.052     0.830     4.134     0.937  6.000       clk2n
Xareg.ck10      4.160     1.013     4.245     1.095  6.000       clk2n
 ...
```

Some clock arrivals might be missing from the report because NanoTime does not calculate unneeded delays. For example, if a clock only drives a P-channel pullup transistor, the tool

analyzes only the falling edge for that pin. You can force the report to include all clock arrival times by adding pulse width checks on all clocks, as shown in the following example.

```
set_min_pulse_width 200 [get_pins -leaf -of \
       [get_nets -hier * -filter "is_clock==true"] \
       -filter "lib_pin_name==$link_transistor_gate_pin_name"]
```

Use the `-tree` option with the `report_clock_arrivals` command to get a report on a full clock tree and all associated clock pins. The clock tree report looks like this:

```
...
Max trees for clock 'clk1'
0.000 r clk1 (Clock source pin)
|- 0.000 r Xreg15.X1.Mn0.g(nand2) (via net)
|  |- 0.048 f Xreg15.X2.Mp0.g(inv2) (via clock_gate)
|  |  |- 0.078 r Xreg15.Mn4.g(dff) (via inverter) (checked pin)
|  |  `- 0.078 r Xreg15.Mp0.g(dff) (via inverter) (checked pin)
 ...
```

This report shows the full clock network using a tree representation. Each node in the tree, represented by one line in the report, is a timing point in the clock tree. Timing points propagated from another timing point are shown indented below the higher-level point.

At each timing point, the report shows the following information:

- Arrival time (delay from clock source)
- Edge direction at clock source (`r` = rising, `f` = falling)
- Timing point name (pin or net name)
- Method of propagation from the previous point:

  `via net` = connection from an input port

  `via inverter` = propagation through an inverter

  `via clock_gate` = propagation across a clock gate

  `via force_propagation = mark_clock_network` setting

  `via unknown gate type` = other situation
- If a clock endpoint, the reason propagation stopped:

  `checked pin` = sequential device clock pin

  `generated clock pin = create_generated_clock` pin

  `stop_propagation = mark_clock_network` setting

  `end of clock network` = no fanout, output port

  `false path, logic blocking, other` = other situation

When you use the `-tree` option, you can use the `-min` or `-max` and `-rise` or `-fall` options to restrict the scope of the report. You can also specify which clocks are to be reported. Otherwise, the command reports all conditions and all clocks. Use the `-nets` option if you want the report to show net names rather than pin names.

6

Recognizable Topologies

Automatic topology recognition is available for a wide range of circuit structures such as transfer gates, latches, registers, inverters, domino structures, and multiplexers.

This chapter contains the following sections:

- Supported Topologies
- Inverter Structures
- Transfer Gates
- Feedback Transistors
- Pulldown and Pullup Structures
- Latch Structures
- Flip-Flop Structures
- Multiplexer Structures
- Domino Precharge Structures
- RAM Structures
- Register File Structures
- XOR Structures

Supported Topologies

For topologies other than digital topologies, you assume the responsibility for validating the calculated delays. The NanoTime tool does not support the following design styles:

- Analog circuits, except certain sense-amp circuits used in an SRAM context
- BiCMOS logic
- Complementary pass transistor logic (CPL)
- Flash or content-addressable memory

NanoTime supports a wide variety of digital design topologies, as described in this user guide. However, unique designs might encounter unexpected results during NanoTime analysis. You must validate NanoTime delays against HSPICE for unusual designs, including the following:

- Topologies with very nonlinear switching characteristics
- Stages with very large transitions at their outputs
- Circuits with multiple channel-connected stages that must be simulated together (such as cross-coupled stages)
- Circuits with complex self-timed loops
- Circuits with multiple inputs that switch simultaneously
- Circuits that require initial conditions within the simulated region
- Circuits that do not have full rail to rail voltage switching
- Circuits that use the source or drain terminals of pass transistors as inputs

Inverter Structures

NanoTime automatically recognizes standard CMOS inverter structures, including long-channel inverters consisting of all transistors in one channel-connected block, as shown in Figure 6-1. NanoTime uses inverter information to propagate clocks and to trace data signals through the design.

Figure 6-1 CMOS Inverter Structures

The `mark_inverter` command lets you specify the location of an inverter that NanoTime does not recognize automatically. In the command, you specify the NMOS pulldown transistor and the PMOS pullup transistor for the inverter. For long-channel inverters, you specify the transistors in the NMOS pulldown stack and in the PMOS pullup stack, between the output of the inverter to the power rails.

To remove recognition of recognized inverters, use the `erase_inverter` command.

You can mark only one inverter at a time with the `mark_inverter` command. However, you can remove recognition of more than one inverter with the `erase_inverter` command.

Transfer Gates

NanoTime automatically recognizes a transfer gate consisting of an NMOS and a PMOS transistor with their sources and drains connected. Figure 6-2 shows some types of transfer gates that are automatically recognized. NanoTime uses information about transmission gates to correctly generate critical paths and to improve delay calculation accuracy.

Figure 6-2 CMOS Transfer Gates

By default, NanoTime recognizes NMOS-PMOS transistor pairs as transfer gate structures, even when there is no inverter between the gates of the transistors. To cause recognition to occur only when an inverter is present, set the `topo_tgate_mark_all_pairs` variable to `false`.

For the transfer gate shown on the lower left of Figure 6-2, to calculate the delay from the input x, NanoTime includes the inverter as part of the stage. For all other types of transfer

gates such as the one shown on the right, NanoTime assumes that the complementary inputs are symmetrical. In other words, it calculates the slope of the signal at the gate of the NMOS or PMOS transistor and assumes that the other gate signal switches in the opposite direction with the same slope.

The `mark_tgate` command lets you specify the location of a transmission gate that NanoTime does not recognize automatically. In the command, you specify the NMOS and PMOS transistors in the design that operate as pass transistors. For example,

```
nt_shell> mark_tgate -n_transistor mn21 -p_transistor mp18
```

To cancel the effects of the `mark_tgate` command, or to override recognition of an automatically detected transmission gate, use the `erase_tgate` command.

Feedback Transistors

NanoTime recognizes feedback transistors connected between the input and output of an inverter. It removes them from path searches to prevent looping through the circuit. However, it considers them to be active devices during delay calculation. Figure 6-3 shows examples of PMOS and NMOS feedback transistors.

Figure 6-3 Feedback Transistors

An NMOS transistor (or PMOS transistor) is recognized as a feedback transistor if it meets both of the following requirements:

- Connectivity – The gate is connected to the output of the inverter, the drain (or source) is connected to the input of the inverter, and the source (or drain) is connected to ground.
- Resistance – The resistance of the feedback transistor must be greater than the resistance of the NMOS transistor of the forward inverter.

If the circuit is designed with an NMOS feedback transistor that is stronger (having less resistance) than the NMOS transistor of the forward inverter, NanoTime does not recognize the feedback transistor by default. To ensure proper recognition, you can set variables that determine the relative resistance threshold for recognizing feedback transistors.

NanoTime checks the following relationship:

$$R_{tran} > R_{forward} \times ratio$$

where R_{tran} is the resistance of the transistor in question, $R_{forward}$ is the resistance of the transistor in the forward inverter, and ratio is the value of one of the ratio variables. The `topo_feedback_n_res_ratio` variable is used if the transistor in question is an NMOS transistor, and the `topo_feedback_p_res_ratio` variable is used for a PMOS transistor.

If the relationship is true, the transistor is recognized as a feedback transistor. A feedback transistor is eliminated from path tracing but is considered an active device during delay calculation. If the relationship is false, the transistor is not recognized as a feedback transistor and is treated as a parasitic element.

The ratio defaults to 1.0, causing a direct comparison between the two transistor resistance values. A smaller ratio, such as 0.8, facilitates the recognition of a transistor as a feedback transistor. A ratio of 0.0 causes all transistors meeting the connectivity requirements to be recognized as feedback transistors, regardless of the resistance values.

A buskeeper circuit is a feedback structure designed to retain the last value on a tristate bus and to prevent a floating value on the bus line. NanoTime automatically recognizes structures as buskeeper circuits if they have back-to-back inverters but are connected to other circuitry at the output of only one of the inverters, as shown in Figure 6-4. The transistors in the inverter driving the bus line are considered to be feedback transistors regardless of their sizes.

Figure 6-4 Buskeeper Circuit

If the feedback transistors are not recognized automatically, you can manually mark the transistor instances with the `mark_feedback` command. The `mark_feedback` command disables path tracing through both the gate and channel of the specified transistors. To disable path tracing in a more targeted manner, use the `mark_instance` command with the `-dont_search_thru_gate` or `-dont_search_thru_channel` option.

To cancel the effects of the `mark_feedback` command or erase automatic recognition of an automatically recognized feedback transistor, use the `erase_feedback` command.

Pulldown and Pullup Structures

NanoTime recognizes several types of pulldown or pullup structures: simple transistors, cross-coupled structures, and turnoff topologies.

Pulldown and Pullup Transistors

The `mark_pulldown` command specifies the location of a pulldown transistor in the design, where a rising-edge transition cannot be propagated due to the lack of a pullup transistor.

For example, the following command identifies transistors mn3 and mn8 as pulldown transistors that cannot propagate a rising-edge transition:

```
nt_shell> mark_pulldown -transistors {mn3 mn8}
```

Similarly, the `mark_pullup` command specifies the location of a pullup transistor in the design, where a falling-edge transition cannot be propagated due to the lack of a pulldown transistor.

Figure 6-5 shows pulldown and pullup transistors.

Figure 6-5 Pulldown and Pullup Circuits

To cancel the effects of a `mark_pulldown` or `mark_pullup` command, use the `erase_pulldown` or `erase_pullup` command.

Cross-Coupled Pullup Transistors

The `mark_cross_coupled_pmos` command identifies PMOS transistors that typically pull up two output nets. The devices must be PMOS devices and cross-coupled as shown in Figure 6-6. Typically this structure is used in low power differential outputs. NanoTime does

not support the automatic recognition of this structure and issues an error message if the structure does not conform to a conventional cross-coupled PMOS topology. The following command marks transistors MP1 and MP2 as a cross-coupled PMOS structure:

```
nt_shell> mark_cross_coupled_pmos -transistors {MP1 MP2}
```

To cancel the effect of the `mark_cross_coupled_pmos` command, use the `erase_cross_coupled_pmos` command.

Figure 6-6 Cross-Coupled PMOS Devices

Weak Pullup and Pulldown Transistors

NanoTime assumes that any transistor that is turned on and connected to Vdd is a weak pullup that pulls the net high, assuming that nothing is pulling the net down, as in Figure 6-7. Similarly, weak pulldown transistors are turned on and are connected to ground.

Figure 6-7 Weak Pullup Transistors

The `mark_weak_pullup` command lets you specify the location of a pullup or pulldown transistor that NanoTime does not recognize automatically. For example, the following command identifies transistor m1 as a weak pullup transistor:

```
nt_shell> mark_weak_pullup m1
```

To cancel the effects of the `mark_weak_pullup` command, or to override recognition of an automatically detected weak pullup or pulldown transistor, use the `erase_weak_pullup` command.

Turnoff Structures

NanoTime automatically recognizes three-state turnoff structures like the one shown in Figure 6-8. The structure consists of two PMOS pullup transistors and two NMOS pulldown transistors connected in series, with complementary enable signals at the respective PMOS and NMOS transistor gates.

Figure 6-8 Recognized Turnoff Structure

The `mark_turnoff` command lets you specify the location of turnoff structures that NanoTime does not recognize automatically, such as the one shown in Figure 6-9.

Figure 6-9 Marked Turnoff Structures

For example, the following command identifies a three-state turnoff structure with an output net Out and enable and disable input pins Mn1.g and Mp1.g:

```
nt_shell> mark_turnoff -enable_pins {Mn1.g Mp1.g Mn2.g Mp2.g} \
          -output_net Out
```

In the command, you must specify the output net of the turnoff structure. You can optionally specify the enable and disable pins of the structure, where NanoTime performs turn-off and turn-on timing checks.

To cancel the effects of the `mark_turnoff` command, or to override recognition of an automatically detected turnoff structure, use the `erase_turnoff` command.

Latch Structures

NanoTime automatically recognizes latch structures based on the meeting of clock and data signals where inverter, NAND, NOR, or three-state elements are arranged in a feedback configuration. Figure 6-10 shows examples of recognizable latch structures.

Figure 6-10 Recognized Latch Structures

Marking Latches Manually

The `mark_latch` command lets you specify the location of a latch that NanoTime does not recognize automatically.

You must specify the latch node. If the `topo_auto_find_latch_clock` variable is set to `false` (the default), you must also specify the clock pin. If any of the specified clock pins blocks the input or inputs from setting the latch net during the transparency window, the tool issues a warning message.

If you set the `topo_auto_find_latch_clock` variable to `true` and do not specify a clock pin, NanoTime attempts to find a clock pin for the marked latch.

You can optionally specify the input, output, feedback elements, feedforward elements, and timing endpoints in the latch. For example, consider the latch circuit shown in Figure 6-11.

Figure 6-11 Latch Circuit Example

The latch circuit shown in Figure 6-11 is automatically recognized. If it could not be recognized automatically, you would specify it manually with the following command:

```
nt_shell> mark_latch -latch_net lat_node \
          -inputs lat_in -output lat_out -clock m1.g \
          -feedforward {m7 m8} -feedback {m2 m3 m4 m6}
```

The feedforward elements go from the latch node to the input of the feedback devices (not necessarily to the output of the latch).

If you specify feedback elements, they must be driven by the output net of the latch. NanoTime stops path tracing at each feedback element to prevent looping through the circuit. If you specify feedforward elements, NanoTime simulates the feedforward elements together to get more accurate delay results.

By default, when NanoTime encounters a latch with an inverter loop, the tool recognizes the inverter with smaller resistance (stronger drive) as the forward inverter and assumes that the weaker inverter is the feedback inverter. The input of the forward inverter is the latch input

node and its output is the latch output node. To change this behavior, set the `topo_feedback_inv_ratio` variable to a value smaller than the default of 1.0.

By default, the NanoTime tool does not recognize a latch or perform latch timing checks if a clock signal is forced through a latch, for example with the following commands:

```
mark_clock_network -force_propagation latch_net
create_generated_clock latch_output_net
```

Set the `topo_allow_latch_on_clock_nets` variable to `true` to retain latch topology marking and latch timing checks even if the latch net is a clock net. In this case, you might see more latches in the topology report and more timing checks in the path report. Latch topology has higher priority than clock gate topology. In other words, if a structure meets the requirements to be interpreted as both structure types during topology recognition, the tool retains the latch designation and discards the clock gate designation.

To cancel the effects of the `mark_latch` command or to override recognition of an automatically detected latch, use the `erase_latch` command.

Debugging Latch Data Inputs

To evaluate the type of input signals that arrive at latch data input pins, set the `topo_find_clock_driven_data_inputs` variable to `true`. NanoTime issues a warning message at the `check_topology` command if either of the following conditions is true:

- The data input is driven by a clock.
- The data input is connected to a rail.

For more information, see Debugging the Data Inputs of Sequential Elements.

Latch Timing Checks

The `-setup_to` option specifies where NanoTime performs setup checking in the latch circuit. It can be set to `latch_net` (the default), `input`, or `output`. Specifying `output` causes NanoTime to check for the arrival of data at the latch output, resulting in a more conservative check than using the latch net. Specifying `input` results in a less restrictive check. If the `-setup_to` option is not present, the setup checking point is determined by the `topo_latch_setup_to` variable, which is set to `latch_net` by default.

Similarly, the `-hold_to` option specifies where NanoTime performs hold checking in the latch circuit. It can be set to either `latch_net` (the default) or `input`. Setting it to `input` causes NanoTime to check for the arrival of data at the latch input, resulting in a more conservative check than using the latch net. In the absence of this option, the hold checking point is determined by the `topo_latch_hold_to` variable, which is set to `latch_net` by default.

You can change the places where NanoTime performs setup and hold checking by using the `set_timing_check_attachment` command. For details, see the man page for the command.

To prevent the setup check from being changed from the default location for nontransparent latches, set the `timing_enable_non_transparent_setup_delta_delay` variable to `false`.

Tapped Feedback in a Latch

A latch can have only one feedforward stage. All other devices or stages required to close the loop back to the latch net must be identified or marked by the `mark_latch` command as feedback devices. Path tracing is not performed through devices identified or marked as a feedback device.

However, if valid paths are tapped off from those feedback nets or stages, path tracing must be performed through those devices. Examples include the master latch to the slave latch of the same flip-flop or the slave latch of one flip-flop to the master latch of another flip-flop. The feedback devices that need to propagate paths are called tapped feedback devices.

Figure 6-12 shows a diagram of a tapped feedback device. Typically, tapped feedback latches occur when there is a transmission gate in the feedback path and the latch output is tapped off of the net connected to the channel of the transmission gate. In this circuit, no path tracing would ordinarily occur from QB through cell X3 to Q if cell X3 is marked only as a feedback device and not as a tapped feedback device.

However, the `topo_latch_find_tapped_feedback` variable allows NanoTime to find and mark tapped feedback devices in a latch topology. This variable defaults to `true`.

If the variable is set to `false`, you can manually mark the tapped feedback devices. The command to mark the structure in Figure 6-12 is as follows:

```
nt_shell> mark_latch -latch_net LAT -inputs D -feedforward {X2/*} \
          -feedback {X3/* X4/*} -tapped_feedback {X3/*} -output Q
```

Figure 6-12 Tapped Feedback Device

X2 is a feed-forward device for latch node LAT
X3 and X4 are feedback devices

There are two ways to perform path tracing through tapped-feedback devices:

- Allow NanoTime to find and mark tapped-feedback devices. By default, the `report_topology -verbose` command reports such devices with a *b*+ annotation; regular feedback devices are marked with a *b* annotation. Feedback device marking is controlled by the `topo_latch_find_tapped_feedback` variable. The variable is set to `true` by default, which allows NanoTime to automatically mark tapped-feedback devices and continue path tracing through such devices.
- Explicitly mark tapped feedback devices with the `-tapped_feedback object_list` option of the `mark_latch` command. NanoTime continues path tracing through these marked devices. Any device identified as a tapped feedback in the `-tapped_feedback` object list must also belong to the feedback device list.

Flip-Flop Structures

A flip-flop is a structure where two latches are connected together sequentially with both latches clocked by the same clock, but with an inverted phase relationship. The first latch is the master, which must have a transparent path to the second latch. The second latch is the slave, which is evaluated as a nontransparent device. The clock that drives the slave must be an inverted version of the clock that drives the master. See Figure 6-13 for an example.

Figure 6-13 Flip-Flop Example

NanoTime can detect flip-flop structures where the master latch output (master latch feed-forward transistor) and slave latch net are in the same channel-connected block. In the example in Figure 6-14, the master latch output is gated through a transfer gate, which drives a slave latch net. Because both sides of a transfer gate are in the same channel-connected block, the flip-flop is recognized.

Figure 6-14 Flip-flop with Contiguous Channel-Connected Blocks

To enable recognition of simple flip-flops constructed from two consecutive latches, add the `flip_flop` argument to the `topo_auto_search_class` variable. You should also enable automatic recognition of topologies in the global database. Use the following commands before you execute the `match_topology` command:

```
set_search_enabled global$(hierarchy_separator)*
set topo_auto_search_class “$topo_auto_search_class flip_flop”
...
match_topology
...
```

You can set this condition as the default in the NanoTime flow.

NanoTime can also recognize flip-flops when the slave latch net is not in the same channel-connected block as the master feed-forward device, as in the case when an inverter exists between the master latch output and the transfer gate, as shown in Figure 6-15. To enable recognition of these flip-flops, set the `topo_flip_flop_strict_slave_check` variable to `false`. The default for this variable is `true`.

Figure 6-15 Flip-flop with Separated Channel-Connected Blocks

NanoTime cannot automatically recognize flip-flops with AOI or OAI structures in the feedforward or feedback legs of either latch. They must be marked explicitly.

If a flip-flop is not automatically recognized, use the `mark_flip_flop` command to mark it manually. The `mark_flip_flop` command should be used after the `match_topology` command, which finds the automatically recognized latches. For user-defined flip-flops, the `mark_flip_flop` command should be used after the `mark_latch` command, but it can occur before the `match_topology` command.

You must specify the master object and slave object of the flip-flop. For each object, you can specify either the name of the latch net or the name of a latch structure previously recognized by the `match_topology command` or marked manually with the `mark_latch` command.

By default, NanoTime uses the latch net as the path endpoint for setup and hold checks. You can change the places where NanoTime performs these checks by using the `set_timing_check_attachment` command.

To cancel the effects of the `mark_flip_flop` command, or to override recognition of an automatically detected flip-flop, use the `erase_flip_flop` command.

Multiplexer Structures

NanoTime automatically recognizes multiplexer structures based on the presence of parallel pass gates or transmission gates that are connected together at the source or drain. See Figure 6-16 for some examples. NanoTime uses multiplexer information to prevent the tracing of false paths.

Figure 6-16 Multiplexer Circuit Examples

To be recognized as a multiplexer, the pass gates or transfer gates must share a common drain or source. In addition, by default, the resistance of the pass gates or transfer gates must be exactly the same. To enable recognition of multiplexer circuits with pass gates or transfer gates with slightly different strengths, set the `topo_mux_drive_res_ratio` variable to specify the allowable difference in strengths.

The `mark_mux` command lets you specify the location of a multiplexer that NanoTime does not recognize automatically. In the command, you must specify the names of the output net and the multiplexer selection pins. For example:

```
nt_shell> mark_mux -output_net N1 \
          -select_pins [get_pins -leaf -of_objects [get_nets {S0 S1}] \
          -filter "lib_pin_name == G"]
```

To cancel the effects of the `mark_mux` command, or to override recognition of an automatically detected multiplexer, use the `erase_mux` command.

Domino Precharge Structures

CMOS domino logic is based on cascades of stages separated by static inverters. Figure 6-17 shows a typical domino stage. Each stage operates under the control of a clock.

Figure 6-17 Domino Logic Circuit

When the clock goes low, the precharge transistor turns on and the evaluate transistor turns off. This charges the precharge node high and forces output Q low. When the clock goes high, it turns off the precharge transistor and turns on the evaluate transistor. Depending on the input values A, B, and C, the circuit either does or does not discharge the precharge node, causing the output Q to either change from low to high or stay low.

The NMOS logic network can contain any combination of NMOS transistors arranged in an AND-OR (series-parallel) configuration. This NMOS network is the pulldown stack. If the stack can be guaranteed to be always off during the precharge phase, the NMOS evaluate transistor is redundant and can be omitted from the circuit. A domino stage without an evaluate transistor is called a footless circuit.

A weak PMOS feedback transistor between the output Q and the precharge node holds the precharge node steady during the evaluate stage. When the evaluation stack is off, the feedback transistor prevents the precharge node from unwanted discharge due to leakage, cross-coupled noise, or charge sharing with nodes in the pulldown stack.

In a cascade of domino stages, multiple stages in sequence can be evaluated within a single clock cycle. The precharge phase sets up the dominoes, and the evaluate phase conditionally topples them, in sequence, until blocked by a logical condition at any one stage. After a domino falls, it cannot rise again until it is set up by another precharge phase.

Domino Precharge Recognition

NanoTime automatically recognizes a variety of domino precharge structures and performs precharge timing checks to ensure that the setup and hold time requirements are met.

The `mark_precharge` command lets you specify the location of a domino precharge structure that NanoTime does not recognize automatically.

To cancel the effects of the `mark_precharge` command, or to override recognition of an automatically detected domino precharge structure, use the `erase_precharge` command.

NanoTime recognizes domino circuit configurations as follows:

- Main precharge block

  NanoTime recognizes either circuit configuration shown in Figure 6-18 as the main precharge block of a domino stage. The block has a PMOS precharge transistor connected to the supply voltage, a static inverter, and a weak PMOS feedback transistor connected to the output or to its own static inverter.

Figure 6-18 Main Precharge Blocks

- Internal precharge element

  Internal precharge elements are normally used to avoid potential charge-sharing problems at internal nodes of the evaluation block. See Figure 6-19 for two examples.

NanoTime does not mark a node with internal precharge elements as a domino node unless the node is driving a gate of another transistor. NanoTime does not trace paths from the gate of an internal precharge element.

Figure 6-19 Internal Precharge Element

Automatic recognition of a precharge node in a domino circuit depends on not only the topology of the circuit, but also the widths and lengths of the feedback transistor and forward inverter. NanoTime recognizes the PMOS transistor as a feedback transistor if its resistance is greater than that of the NMOS transistor in the forward inverter. The resistance comparison is affected by the `topo_feedback_p_res_ratio` variable. To have feedback transistors recognized by connectivity only, set the variable to 0.

Evaluation Clock

NanoTime can handle multiple evaluation circuits connected to a single evaluation clock, such as the configuration shown in Figure 6-20.

Figure 6-20 Single Evaluation Clock

Within a single evaluation stack, the evaluation clock device need not be at the bottom of the stack as shown in Figure 6-21.

Figure 6-21 Evaluation Clock Location

For a single domino, there can be multiple evaluation stacks controlled by different clocks as shown in Figure 6-22.

Figure 6-22 Different Evaluation Clocks for Same Domino Nodes

The precharge clock and the evaluation clock do not have to be the same clock node. The evaluation stacks without evaluation clocks are recognized as footless domino logic.

Embedded Latch Structures

If the domino precharge circuit has a latch function, you can specify the latch type with the `-latch_type` option of the `mark_precharge` command, and NanoTime performs the appropriate timing checks. The valid values of the `-latch_type` option are:

- `none`: No keeper.
- `half_latch`: Half-latch keeper.
- `latch`: Full keeper, different phase from data.
- `retain`: Full keeper, same phase as data.
- `flop`: Full keeper, same phase as data, flip-flop.

The circuits represented by these settings are illustrated in Figure 6-23.

Figure 6-23 Embedded Latch Structures

A half-latch keeper is a pullup feedback transistor connected between the domino output and the evaluate net. A full keeper has both pullup and pulldown feedback transistors, thus making a set of two inverters that feed back into each other.

The phrase “same phase as data” means that the domino gate and the latch or domino gate that precedes it in the path are controlled by the same clock and are transparent on the same phase of that clock.

The `flop`, `latch`, and `retain` latch types all have the same full-keeper feedback structure. Because of the pulldown transistor on the evaluate net, a data input that is high does not

need to remain high throughout the evaluate phase. This relaxes the falling-clock to falling-input hold constraint. The other timing checks done for the `latch` and `retain` cases are the same as those done for the `half_latch` case. A precharge gate acts like a latch because its output can be in a different clock cycle from its input.

For `latch`, NanoTime assumes that the domino gate should be on a clock phase boundary. That is, the clock for the domino gate should be inverted with respect to any domino gate or latch that drives its input. If this is not the case, NanoTime issues a warning during path tracing.

For `retain`, NanoTime assumes that the domino gate should not be on a clock phase boundary. If this is not the case, it issues a warning during path tracing.

For `flop`, NanoTime treats the gate as nontransparent, does not trace paths through the gate, and does not issue any warnings about the phase of the data. It allows a full cycle for setup and hold checks if the data comes from a latch or domino gate controlled by the same clock and clock phase. (For the other latch types, NanoTime expects the data to arrive at the input of the domino gate in the same clock cycle that it leaves the previous latch or domino gate.) Otherwise, the checks are the same as for the `latch` or `retain` case.

Predischarge Domino Circuits

The `-type` option of the `mark_precharge` command specifies the type of transistor used in the evaluate stack, either `NMOS` for a precharge circuit or `PMOS` for a predischarge circuit.

For a PMOS type circuit, the clock is high during the predischarge phase and low during the evaluate phase. The `-precharge_clock` device is a clock-controlled NMOS transistor with a path to ground that discharges the evaluate net in the predischarge phase. The `-evaluate_clock` device, if present, is a PMOS transistor that is turned off during the predischarge phase and prevents the evaluate stack from conducting. Then, in the evaluate phase, the `-evaluate_clock` device is turned on, and the `-evaluate_data` devices (PMOS transistors) in the evaluate stack conditionally charge up the evaluate node.

An NMOS type (precharge) circuit is also known as a N-domino type circuit, and a PMOS type (predischarge) circuit is also known as a P-domino type circuit.

Domino Precharge Recognition Examples

The following examples demonstrate how to specify domino precharge structures with the `mark_precharge` command.

The first three examples can be recognized automatically, so the `mark_precharge` command is not necessary for them. The command is shown to demonstrate the command syntax.

The last three examples are not recognized automatically and must be specified with the `mark_precharge` command to have setup and hold timing checks performed on them.

The example shown in Figure 6-24 is a simple precharge gate having one precharge clock transistor, a half-latch keeper, and a footless stack with two evaluate data transistors. NanoTime recognizes this structure automatically and it is not necessary to use the `mark_precharge` command.

Figure 6-24 Footless Precharge Gate With Half-Latch Keeper

The following command would identify this structure:

```
nt_shell> mark_precharge -type NMOS \
            -latch_type half_latch \
            -evaluate_net enode \
            -precharge_clock MP1 \
            -keeper_data MP2 \
            -evaluate_data { MN1 MN2 }
```

The example shown in Figure 6-25 is a predischarge gate having one predischarge clock transistor, a stack with two evaluate data transistors, and an evaluate clock transistor. NanoTime recognizes this structure automatically and it is not necessary to use the `mark_precharge` command.

Figure 6-25 Predischarge Gate Without Keeper

The following command would identify this structure:

```
nt_shell> mark_precharge -type PMOS \
            -latch_type none \
            -evaluate_net enode \
            -precharge_clock MN1 \
            -evaluate_data { MP1 MP2 } \
            -evaluate_clock MP3
```

The example shown in Figure 6-26 is a set of two precharge gates that share a common evaluate clock transistor. NanoTime recognizes this structure automatically and it is not necessary to use the `mark_precharge` command.

Figure 6-26 Multiple Evaluate Stacks With a Shared Evaluate Transistor

The following command would identify this structure:

```
nt_shell> mark_precharge -type NMOS \
            -latch_type half_latch \
            -evaluate_net {e1 e2} \
            -precharge_clock {MP11 MP21} \
            -keeper_data {MP12 MP22} \
            -evaluate_data {MN11 MN12 MN21 MN22} \
            -evaluate_clock MN13
```

The example shown in Figure 6-27 is a precharge gate that has a half-latch keeper and a secondary precharge transistor to precharge an intermediate node in the evaluate stack. NanoTime does not recognize this structure automatically.

Figure 6-27 Precharge Gate With Secondary Precharge Transistor

The following command would identify this structure:

```
nt_shell> mark_precharge -type NMOS \
            -latch_type half_latch \
            -evaluate_net enode \
            -precharge_clock MP1 \
            -precharge_other MP3 \
            -keeper_data MP2 \
            -evaluate_data { MN1 MN2 } \
            -evaluate_clock MN3
```

The example shown in Figure 6-28 is a precharge gate that has a full keeper and two precharge data transistors to precharge an intermediate node in the evaluate stack. NanoTime does not recognize this structure automatically.

Figure 6-28 Precharge Gate With Precharge Data Transistors

The following command would identify this structure:

```
nt_shell> mark_precharge -type NMOS \
            -latch_type latch \
            -evaluate_net enode \
            -precharge_clock MP1 \
            -precharge_data MP3 MP4 \
            -keeper_data {MP2 MN3} \
            -evaluate_data { MN1 MN2 }
```

The example shown in Figure 6-29 is a precharge gate that has a half-latch keeper on the evaluate net, a secondary precharge transistor to precharge an intermediate node in the evaluate stack, and a full keeper on the intermediate node in the stack. NanoTime does not recognize this structure automatically.

Figure 6-29 Precharge Gate With Keeper on Data Node

The following command would identify this structure:

```
nt_shell> mark_precharge -type NMOS \
            -latch_type half_latch \
            -evaluate_net enode \
            -precharge_clock MP1 \
            -precharge_other MP4 \
            -keeper_data {MP2 MP3 MN3} \
            -evaluate_data { MN1 MN2 }
```

Domino Precharge Circuit Types

There are two types of evaluate circuits, called type D1 and type D2. A type D1 domino structure has a controlling clock on the evaluate stack, whereas a type D2 domino structure does not. There are also two types of charging circuits, called N-domino (precharge) and P-domino (predischarge).

An N-domino circuit has an NMOS evaluate stack. This type of circuit charges a precharge node to a high voltage during a precharge phase, when the clock is low; and then conditionally discharges that node during the evaluate phase, when the clock is high.

A P-domino circuit has a PMOS evaluate stack. This type of circuit discharges a predischarge node to a low voltage during a predischarge phase, when the clock is high; and then conditionally charges that node during the evaluate phase, when the clock is low.

Figure 6-30 through Figure 6-33 show some examples of domino circuits and the names of nets in those circuits that are referenced in path timing reports.

Figure 6-30 Type D1 N-Domino (Basic) Example

Figure 6-31 Type D2 N-Domino (Basic) Example

Figure 6-32 Type D1 N-Domino Retain Example

Figure 6-33 Type D2 P-Domino Retain Example

Each domino stage falls into one of the following categories:

- Domino (basic): A half-keeper domino gate that is clocked with a clock that has the same phase as the clock that triggers the input data, and that has not been marked as a flip-flop by the `mark_flip_flop` command.
- Domino retain: A full-keeper domino gate that is clocked with a clock that has the same phase as the clock that triggers the input data, and that has not been marked as a flip-flop by the `mark_flip_flop` command. This case is similar to the basic domino stage, except that it has a full keeper rather than a half PMOS keeper.
- Domino latch: A full-keeper domino gate that is clocked with a clock having a different phase from the clock that triggers the input data. The stage can act as a transparent latch. NanoTime uses the new-phase clock for timing checks.
- Domino flip-flop: A full-keeper domino gate that is clocked with a clock having the same phase as the clock that triggers the input data, and has been marked as a flip-flop by the `mark_flip_flop` command. These gates are treated as flip-flops. The timing margin calculations account for the cycle change. Instead of using the same cycle for the basic typical domino checks, NanoTime uses the next clock cycle for the checks.

RAM Structures

NanoTime automatically recognizes a random-access memory (RAM) structure when two equal-strength inverters feed back into each other. Figure 6-34 shows a typical RAM structure.

Figure 6-34 RAM Structure

By default, the strengths of the two inverters must be the same for automatic recognition of the RAM structure. To enable recognition of inverters of slightly different strengths, set the `topo_ram_drive_res_ratio` variable to specify the allowed amount of difference in strengths.

By default, NanoTime continues path tracing through RAM structures. To have NanoTime stop path tracing when it reaches a RAM structure, set the `topo_ram_search_thru_cell` variable to `false`.

The `mark_ram` command lets you specify the location of a RAM structure that NanoTime does not recognize automatically. The command must specify the names of the two input nets of the RAM structure.

For example, the following command identifies a RAM structure having input nodes at nets n47 and n48:

```
nt_shell> mark_ram -node1 n47 -node2 n48
```

Back-to-back inverters can also be used for differential synchronizers. If the two nets on each end of the structure have been marked as a differential circuit pair, the circuit in Figure 6-34 is recognized as a differential synchronizer and not as a RAM cell.

To cancel the effects of the `mark_ram` command, or to override recognition of an automatically detected RAM structure, use the `erase_ram` command.

Register File Structures

NanoTime automatically recognizes a register file structure when two equal-strength inverters feed back into each other (as in the RAM topology) and are controlled by two clocks. A register file memory is different from static RAM (SRAM) memory because the read circuitry samples data at the internal inverters. In SRAM memory, the read and write circuitry share the bit column and read and write data have to flow through pass gates from the bit column to the internal inverters.

The `topo_regfile_search_all_clock_pins` variable controls the method of searching for register file clock pins. When the variable is set to `false` (the default), NanoTime starts the clock pin search from the register file net and stops the search upon finding the first clocked device. When the variable is set to `true`, the tool finds the clock pins by searching all clocked devices that are channel-connected to the register file nets.

Figure 6-35 shows a typical register file structure. The strengths of the paired inverters must be the same for automatic recognition of the internal RAM.

Figure 6-35 Typical Register Structure

By default, timing paths and timing checks for register file structures do not include the path segment across the register file (the path between nets reg1 and reg2 in Figure 6-35). The only path from IN to OUT goes through net n2, and the associated timing checks are defined with respect to the clock signal on transistor m2.

If you want NanoTime path tracing to include the path segment across the register file, set the `timing_register_file_add_cross_delay_constraint_arcs` variable to `true`. In this case, two paths are traced from IN to OUT, the original path through net n2 and a new path through nets reg1 and reg2. New timing checks are also automatically created. In this example, the timing checks reference the clock signal on transistor m1.

Figure 6-36, Figure 6-37, and Figure 6-38 show additional register file structures that NanoTime recognizes automatically. The tool can also recognize the structure in

Figure 6-36 if the initial NAND gate is replaced with a NOR gate, and the structures in Figure 6-37 and Figure 6-38 if the initial inverter is replaced with a NAND gate or NOR gate.

Figure 6-36 Pulldown Register File Topology

Figure 6-37 Pullup Register File Topology

Figure 6-38 Inverter Register File Topology

The `mark_register_file` command lets you specify the location of a register file structure that is not recognized automatically. The command must specify the names of the nets that serve as the register data nodes and clock pins.

For example, the following command identifies a register structure having data nodes at nets n47 and n48 and controlling clocks at pins m51.g and m52.g:

```
nt_shell> mark_register_file -net1 n47 -net2 n48 \
          -clock1 m51.g -clock2 m52.g
```

The circuit is shown in Figure 6-39.

Figure 6-39 Marked Register Example

XOR Structures

NanoTime automatically recognizes a 6-transistor implementation of XOR (exclusive OR) and XNOR (exclusive NOR) logic gates. The recognized XOR implementation is shown in Figure 6-40. The recognized XNOR structure is similar. NanoTime uses its knowledge of XOR and XNOR structures to eliminate false paths for some logic states.

Figure 6-40 Recognized XOR Structure

The `mark_xor` command lets you specify the location of an XOR structure that NanoTime does not recognize automatically. The command must specify the input pins and output net of the XOR structure. For example, the following command identifies an XOR structure with input pins Mn6.g and Mn7.g and output net out1:

```
nt_shell> mark_xor -output_net out1 -input_pins { Mn6.g Mn7.g }
```

To cancel the effects of the `mark_xor` command, or to override recognition of an automatically detected XOR or XNOR structure, use the `erase_xor` command.

7

Differential Circuits

NanoTime supports static timing analysis of differential circuits, allowing you to define differential clocks, propagate clock signals, and recognize differential topologies.

This chapter contains the following sections:

- Differential Circuit Analysis
- Differential Nets, Pins, and Ports
- Differential Clocks
- Differential Skew Analysis
- Differential Synchronizers
- Differential Latches and Flip-Flops
- Cascode Voltage Switch Logic
- Level Shifter Circuits

Differential Circuit Analysis

Differential full-swing clocking is a design technique for high-performance architectures that provides better noise immunity and faster performance than single-ended clock distribution schemes. A differential signal is composed of a pair of related signals that switch full rail in opposite directions with minimal skew between them. One of the signals is called the reference and the other is called the complement.

Figure 7-1 shows a simple differential circuit. Signals clkn and clkp are differential clock signals feeding into a differential clock network. The differential synchronizer Xsync maintains a fixed skew between nets clkn and clkp. A differential latch accepts differential input signals Dn and Dp and generates differential output signals Qn and Qp.

Figure 7-1 Differential Clock Environment

Analysis of differential clocks and circuits requires a NanoTime Ultra license.

The following conditions apply:

- Sequential topologies can have only one differential clock pair.
- You must use the strict transistor direction flow for differential circuits.
- You cannot use dynamic simulation or multi-input switching to analyze differential circuits. NanoTime issues a CMDS-110 error message at the `mark_simulation` and `check_design` commands if a differential net pair is detected in the dynamic simulation region. Use automatic skew analysis to obtain accurate skews and delays for differential circuits.

Differential Nets, Pins, and Ports

Use the `set_differential` command to specify that two objects of the same type should be a differential pair. The first object is the reference object; the second is the complement.

The usage depends on the object type, as follows:

- Ports

  Specifying differential ports is primarily for the purpose of creating differential clocks at the boundary of the design. The reference object must be a clock port.

- Nets

  Specifying differential nets is the most common usage of the `set_differential` command. In general, NanoTime does not infer the presence of differential nets based on circuitry. You should mark known differential nets for best results. However, if you specify two pins or two ports as a differential pair, NanoTime sets the two nets associated with the pins or ports to be a differential net pair.

- Pins

  Specifying differential nets does not cause NanoTime to mark any pins on those nets as differential pins. It is not necessary to specify all differential pin pairs. However, explicit marking of differential pin pairs is beneficial to help guide the analysis. The reference pin must be a leaf pin of a clock source.

To undo a differential pair association, use the `remove_differential` command with either one of the member objects as an argument. If objects clkp and clkn constitute a differential pair, you can use either of the following commands:

```
nt_shell> remove_differential clkp
nt_shell> remove_differential clkn
```

Differential Circuits With Enable Pins

You can use the `set_differential` command even if a pair of objects exhibits differential behavior only part of the time. For example, an enable signal might cause a pair of differential nets to have opposite logic states when enabled but to have the same state when disabled.

The `mark_net` command provides targeted guidance for the analysis of specific nets. The following options are available for marking the enable pins of differential circuits. In both

cases, the override pins must be in the same channel-connected block as the net to which they are assigned.

- The `-differential_override_clear` option specifies the gate pins of transistors that set the corresponding output net to a logic low state.
- The `-differential_override_preset` option specifies the gate pins of transistors that set the corresponding output net to a logic high state.

The pins set by these options are stored in the `differential_override_clear` and `differential_override_preset` net attributes.

To remove a net-specific designation, use the `-differential_override` option with the `erase_net` command.

Fingered Devices in Differential Circuits

When you define differential circuits that include fingered devices, specify only one representative pin of the fingered device. You can determine which pin is the reference pin of a fingered device by examining the `is_parallel_reference` attribute (a pin attribute). Use the `is_parallel_non_reference` attribute to filter the other pins.

For example, the following commands ensure that nonreference pins of fingered devices are filtered from the lists of source and target objects of a generated clock:

```
nt_shell> set source_pins [get_pins -leaf -of {n2clk1n n2clk1p}  \
                           -filter “!is_parallel_non_reference”]
{“xnd1/MN1/G”,”Xcn4/md1/D”,”Xcn4/mdn1/D”,”xnd1/MP1/G”,”Xcp4/mup1/D”,
“Xcp4/mdn1/D”}

nt_shell> set target_pins [get_pins -leaf -of gclkn -filter \
          “lib_pin_name==$link_transistor_gate_pin_name &&  \
          !is_parallel_non_reference”]
{“Xgi/mdn1/G”,”Xgi/mup1/G”,”Xgcn0/mdn1/G”,”Xgcn0/mup1/G”}

nt_shell> create_generated_clock -divide_by 2 -name gclkn \
          $target_pins -source $source_pins
```

Attributes for Objects in Differential Circuits

NanoTime sets the value of several attributes when you mark an object as part of a differential pair. The `is_differential_synchronizer` attribute for the cell object class is set to `true` for cells recognized as synchronizers. In addition, the following attributes are available for the net, port, and pin object classes:

- The `is_differential` attribute is `true` if the object is part of a differential pair.
- The `is_differential_reference` attribute is `true` if the object is the reference of a differential pair.
- The `differential_complement` attribute contains the name of the other half of the pair.
- The `differential_skew_max` and `differential_skew_min` attributes contain the user-defined skew values set with the `-skew_max` and `-skew_min` options of the `set_differential` command. These attributes are not updated during iterative skew analysis.

Differential Clocks

Use the `set_differential` command on ports or pins with the `create_clock` and `create_generated_clock` commands to create differential clocks.

Creating Boundary Differential Clocks

Figure 7-2 shows a simple circuit with parallel elements.

Figure 7-2 Simple Differential Clock

The following procedure creates a differential clock pair at ports clkp and clkn in Figure 7-2:

1. Define a standard clock on port clkp with the `create_clock` command.
2. Assign ports clkp and clkn to be a differential pair with the `set_differential` command. Because clkp is the clock object, it is the reference object of the differential pair that includes ports clkp and clkn. Therefore, it must appear first in the argument list.

The resulting commands are as follows:

```
create_clock -name clkn_clkp [get_ports clkp] \
             -waveform {0 1.0} -period 2.0
set_differential [get_ports {clkp clkn}]
```

You can then specify clock properties such as latency and uncertainty on the reference port or pin of the differential pair. You can set clock properties any time before the `check_design` command.

Alternatively, you can first create a reference clock to use as a basis for creating generated clocks. The reference clock enables you to define properties such as input delay and latency more easily when there are multiple generated clocks.

To create the differential clock in Figure 7-2 with this method, use the following procedure:

1. Define a reference clock with the `create_clock` command.
2. Define a generated clock at port clkp based on the reference clock, using the `create_generated_clock` command. Using the `-edges` option with the `create_generated_clock` command is required when you are creating differential clocks for use in extracted timing models.
3. Assign ports clkp and clkn to be a differential pair with the `set_differential` command.

The resulting commands are as follows:

```
create_clock -name refclk -waveform {0 1.0} -period 2.0
create_generated_clock -name clkn_clkp -master_clock \
          [get_clocks refclk]  [get_ports clkp] -edges {1 2 3}
set_differential [get_ports {clkp clkn}]
```

You can then specify clock properties such as latency and uncertainty on the reference clock. Generated clocks inherit these properties of their reference clocks. You can set clock properties any time before executing the `check_design` command. However, the input transition indexes for creating timing models are not inherited from the master clock and must be set explicitly with the `set_model_input_transition_indexes` command or the `model_default_input_transition_indexes` variable.

By default, NanoTime copies certain clock properties from a clock net to its differential complement net, as follows:

- Marking with the `mark_clock_network -force_propagation` command
- Marking with the `mark_clock_network -stop_propagation` command
- Marking with the `mark_clock_network -no_pulse` command

To prevent copying these clock properties from one net to its differential complement, set the `topo_copy_clock_property_from_differential_net` variable to `false`.

Creating Internal Differential Clocks

To modify clock properties on internal nets, you must create a new differential clock from an existing differential clock. For example, in Figure 7-3 the differential clock signals clkp and clkn are propagated to the latch. However, to change the clock frequency at outputs Qn and Qp, you must use the `create_generated_clock` command to specify a new differential clock.

The commands to create a differential clock at cells X7 and X8 in Figure 7-3 are as follows:

```
create_clock -name refclk -waveform {0 1.0} -period 2.0 [get_ports clkp]
set_differential {clkp clkn}
set_differential {bclkp bclkn}
set_differential {Dn Dp}
set_differential {Ln Lp}
create_generated_clock -name div2_Ln_Lp -edges {2 4 6} \
               -source [get_pins {X6/mp*/g X5/mp*/g}] [get_pins X7/*/g]
set_differential {X5/mp/g X6/mn/g}
set_differential {X6/mp/g X5/mn/g}
set_differential {X7/mp/g X8/mp/g}
set_differential {X7/mn/g X8/mn/g}
```

Figure 7-3 Differential Clock Example

The following procedure describes the steps to create the script:

1. Mark nets that are known to be parallel with the `set_differential` command.
2. Use the `create_generated_clock` command to define a generated clock named div2_Ln_Lp at the gate pins of cell X7, with a frequency based on that of clkp and clkn. The clock source pins are the PMOS pins of cells X5 and X6. Use the `-edges` option with

the `create_generated_clock` command instead of either the `-divide_by` or `-multiply_by` options when defining a differential generated clock.

3. Define differential pairs between the PMOS and NMOS pins of each transmission gate with the `set_differential` command. The PMOS pins appear first in the argument list because they are the reference clock objects.

   Executing a `set_differential` command on a net does not cause the pins of that net to be marked as differential. You must mark both the pins and the nets of parallel structures in differential clock networks.

4. Define one differential pair between the PMOS pins of cells X7 and X8 and another between the NMOS pins, using the `set_differential` command.

Reporting Differential Clocks

Several clock-related reports indicate differential clocks in the attributes column.

The `report_clock` command shows the clock name, period, edge times, clock attributes, and clock source for each clock in the design. The attribute D indicates a differential clock. The Sources column indicates the source object specified at the time of clock creation. For differential clocks, both the reference and complement objects of the differential pair are shown. The reference object appears first in the list, as shown in the following example:

```
nt_shell> report_clock
...
Attributes:
    p - Propagated clock
    G - Generated clock
    H - Pulse clock, high
    L - Pulse clock, low
    D - Differential clock

Clock      Period Waveform          Attrs Sources
-------- -------- ----------------- ----- --------------
MCLK        4.000 {0.000 2.000}     p
clk1        4.000 {0.000 2.000}     p     clk1
clk2        4.000 {1.000 3.000}     p     clk2
clk3n       1.400 {0.000 0.700}     pD    {clk3n clk1p}
```

The `report_clock_arrivals` command reports the arrival times of clock signals at sequential device pins or along intermediate points of a full clock tree. After using the `check_design` command, you can use the `report_clock_arrivals` command to view a list of the clock nets. However, to see the actual arrival times in the report, you must run the `trace_paths` command first.

The report lists the sequential device clock input nets, the “Min Rise,” “Min Fall,” “Max Rise,” and “Max Fall” arrival times at each pin, and the clock source (with attributes, if applicable) as shown here:

```
nt_shell> report_clock_arrivals
...
Attributes:
   H - Pulse clock with high pulse
   L - Pulse clock with low pulse
   D - Differential clock

Net          Min Rise  Min Fall  Max Rise  Max Fall Period Attrs Clock
----------- --------- --------- --------- --------- ------ ----- --------
out3n           0.014     1.014     0.014     1.014  2.000   D   vip_vin
out3p           1.014     0.014     1.014     0.014  2.000   D   vip_vin
out9n           1.021     0.021     1.021     0.021  2.000   D   vip_vin
```

Differential Skew Analysis

Differential synchronizers reduce, but do not eliminate, skew between two nets in a differential pair. NanoTime can accurately analyze differential skew using an iterative approach. Automatic skew analysis applies skew between differential pins.

Alternatively, you can define a constant skew value to be applied between differential nets. This is unlikely to be accurate, but is available for investigations and for backward compatibility.

You cannot use dynamic clock simulation or dynamic delay simulation to analyze differential circuits. NanoTime issues an error message at the `mark_simulation` and `check_design` commands if a differential net pair is detected in a dynamic simulation region. Use automatic skew analysis to obtain accurate skews and delays for differential circuits.

Determining Differential Skew Automatically

To enable automatic skew analysis, set the `timing_differential_iteration_count` variable to an integer value greater than 1 before initiating path tracing. A value of 1 (the default) disables the feature. You can also specify the minimum number of iterations with the `timing_min_iteration_count` variable, which applies to all forms of iterative path tracing analysis.

If there are no differential structures in the design, the variables have no effect.

Automatic differential skew applies to differential pins, not differential nets. The NanoTime tool first identifies the trigger pin, then selects a pin on the differential complement net to be the differential complement pin of the trigger pin. If you have not explicitly marked any pins as differential pairs, NanoTime might not select the correct pin to be the complement. In this

case, the calculated skew between the trigger pin and its complement might not be accurate. The inaccuracy is likely to be small if the parasitics associated with all of the pins on the complement net are similar. However, for best results, you should specify known differential pins explicitly.

During each iteration of skew analysis, NanoTime computes a new skew and transition slope for every pair of differential pins in the design, using the values from the previous iteration as a starting point. The largest changes typically occur within two to four iterations.

During analysis, the tool issues a PATH-023 message for each differential skew iteration and one or more PATH-024 messages to list the resulting skew and slope values for each differential pin.

Guidelines for Analysis Setup

If iterative differential skew analysis is enabled, any differential net skew values set using the `-skew_max` option of the `set_differential` command are used as starting points for the first iteration. Poor starting values might cause the analysis to take longer to converge. For best results, do not use the `-skew_max` option if you are using iterative differential skew analysis.

Some designs contain a large number of differential synchronizers in series along a path. This number is the differential synchronizer depth. If a large amount of skew is introduced early in the path, you must specify enough skew iterations to propagate the skew along the entire path. For best results, set the `timing_differential_iteration_count` variable to a value equal to or greater than the maximum differential synchronizer depth that exists on any path.

Using Multiple Types of Iterative Analysis

You can use differential skew analysis at the same time as signal integrity analysis and timing-based multi-input switching analysis, which are also iterative calculations. If one or more of these features is enabled, NanoTime performs path tracing iterations. Analysis results are updated during each iteration for every enabled feature. Iterations stop when all exit criteria for the enabled features are satisfied.

For example, if you specify two iterations for differential skew analysis and four iterations for signal integrity analysis, NanoTime executes four iterations and updates both sets of results during each iteration. In this case, the tool issues a PATH-025 information message to let you know that additional differential skew iterations are being performed beyond those required by the `timing_differential_iteration_count` variable.

Note:

In rare cases, oscillations between iterations might result, especially when strong aggressors are logically constrained to switch in a direction that removes pessimism. You should always use the worst-case results for your analysis and choose iteration exit criteria that produce the worst-case results.

Setting Differential Skew Manually

If the `timing_differential_iteration_count` variable is set to 1 (the default), skew iteration is disabled. In this case, you can manually specify a constant value for the skew between two differential nets with the `-skew_max` option of the `set_differential` command. The skew value must be positive or zero (the default). If skew iteration is enabled, the `-skew_max` option has no effect. The `-skew_min` option is used only for path-based slack adjustment analysis.

The skew options of the `set_differential` command do not apply to ports or pins. Use the `set_clock_latency` command for ports and pins.

Differential net skew is nondirectional and edge-agnostic. The complement net always switches later than the trigger net. For example, consider two nets data1n and data1p. Set a skew value of 5 ps for maximum delay analysis with the following command:

```
nt_shell> set_differential {data1n data1p} -skew_max 5
```

If data1n switches during maximum delay analysis, data1p switches 5 ps later. If data1p switches first, data1n switches 5 ps later.

You can experiment with different values of differential skew. First, use the `reset_design -paths` command to reset the analysis database. Then, use the `set_differential -skew_max` command to change the manually applied skew. Run path tracing again to observe the results.

Differential Synchronizers

Differential synchronizers reduce the skew between two nets in a differential pair. NanoTime automatically recognizes the synchronizer circuit types shown in Figure 7-4 and Figure 7-5.

Figure 7-4 Cross-Coupled NMOS and PMOS Differential Synchronizers

Figure 7-5 Cross-Coupled Inverter Differential Synchronizers

Several conditions are necessary for a structure to be recognized as a differential synchronizer:

- The synchronizer must consist of a loop containing exactly two channel-connected blocks.
- Each channel-connected block must form one side of the synchronizer, where a side is a connection between the two enclosing nets (the two members of the differential net pair whose signals are to be synchronized).

Figure 7-6 shows examples of acceptable and unacceptable synchronizer designs. In each side of circuit (a), an inverter feeds into a pass gate.

However, although circuit (b) also contains two channel-connected blocks, each channel-connected block does not connect the enclosing nets bclkn and bclkp. NanoTime cannot handle circuit (b) as a differential synchronizer.

Figure 7-6 Differential Synchronizer Designs

To enable automatic recognition of synchronizer circuits, you must mark the enclosing nets (the differential nets associated with the synchronizer) as a differential pair before executing the `match_topology` command:

```
nt_shell> set_differential {bclkn bclkp}
...
nt_shell> match_topology
```

You can also manually mark circuits with the `mark_differential_synchronizer` command. If any additional devices (such as inverters) exist between the nets on the two sides of a synchronizer, you must manually mark the synchronizer to force NanoTime to recognize the synchronizer as being distinct from the other devices. This allows the tool to control the delays through the synchronizer based on the specified skew values while simulating paths through the other circuitry.

The transistors listed in the `mark_differential_synchronizer` command must not be part of another synchronizer. Also, the nets associated with the synchronizer must already be marked as a differential pair.

You must manually mark nonstandard synchronizer designs. For example, the following commands are necessary to recognize the structure in Figure 7-7 as a differential synchronizer:

```
nt_shell> set_differential {bclkn bclkp}
nt_shell> mark_differential_synchronizer \
          -transistors {mp1 mn1 mp2 mn2 X3/* X4/*}
```

Figure 7-7 Nonstandard Differential Synchronizer

Timing analysis does not include paths through differential synchronizers. To perform timing analysis through paths that traverse any part of a differential synchronizer, you must mark that synchronizer manually to indicate that some or all of the transistors in the synchronizer are tapped transistors by using the `-tapped_transistors` option of the `mark_differential_synchronizer` command.

For example, in Figure 7-7, nets set, rst, out and outx are internal to the synchronizer. If you want to perform timing checks related to these internal nets, use the following commands to mark the transistors of cells X3 and X4 as tapped transistors:

```
nt_shell> set_differential {bclkn bclkp}
nt_shell> set_differential {out outx}
nt_shell> set_differential {clk clkn}
nt_shell> mark_differential_synchronizer \
          -transistors {mp1 mn1 mp2 mn2 X3/* X4/*} \
          -tapped_transistors {X3/* X4/*}
```

If you also want to perform timing checks between the nets bclkn and bclkp, you must additionally mark the transistors of the pass gates as tapped transistors:

```
nt_shell> set_differential {bclkn bclkp}
nt_shell> set_differential {out outx}
nt_shell> set_differential {clk clkn}
nt_shell> mark_differential_synchronizer \
          -transistors {mp1 mn1 mp2 mn2 X3/* X4/*} \
          -tapped_transistors {mp1 mn1 mp2 mn2 X3/* X4/*}
```

Differential Latches and Flip-Flops

Figure 7-8 shows a differential latch. If the synchronizer (cells X1 and X2) is recognized or marked as a synchronizer, NanoTime automatically recognizes the latch. However, you might need to mark some latches manually if they are not recognized automatically.

Figure 7-8 Differential Latch

To manually mark the latch shown in Figure 7-8, use the following procedure:

1. Mark known differential nets with the `set_differential` command, including the latch nets (Lp and Ln), the clocks (bclkp and bclkn), and the inputs (Dp and Dn).
2. Execute the `match_topology` command to propagate the clocks.
3. Mark the latch with the `mark_latch` command, but do not include the differential synchronizer cells X1 and X2 in the latch definitions.
4. Review the topology report (obtained with the `report_topology` command) to verify that the topologies are correct.

The resulting commands are as follows:

```
set_differential {Ln Lp}
set_differential {bclkn bclkp}
set_differential {Dn Dp}
match_topology
mark_latch -latch_net {Ln} -inputs {Dn} -clock {X6/*}
mark_latch -latch_net {Lp} -inputs {Dp} -clock {X5/*}
```

Figure 7-9 shows an example of a differential flip-flop. Differential flip-flops can be automatically recognized if these conditions are met:

- The `topo_auto_search_class` variable includes structure type `flip_flop`. This is not true by default.
- The underlying latches have been either automatically recognized or manually marked.
- The master latch net is an input of the channel-connected block that contains the slave latch net, excluding the differential synchronizers.

To manually mark a differential flip-flop, use the following procedure:

1. Mark nets that are known to be parallel with the `set_differential` command.
2. Execute the `match_topology` command to propagate the clocks.
3. Mark the latches with the `mark_latch` command.
4. Mark the flip-flops with the `mark_flip_flop` command.
5. Execute the `check_topology` command.

Figure 7-9 Differential Flip-Flop

The resulting commands for the example in Figure 7-9 are as follows:

```
set_differential {nclkp nclkn}
set_differential {Dn Dp}
set_differential {Mn Mp}
set_differential {Q1p Q1n}
set_differential {Sn Sp}
set_differential {Qn Qp}
...
match_topology
...
mark_latch -latch_net {Mn} -inputs {Dn} -clock {X6/*}
mark_latch -latch_net {Mp} -inputs {Dp} -clock {X5/*}
mark_latch -latch_net {Sn} -inputs {Mn} -clock {X16/*}
mark_latch -latch_net {Sp} -inputs {Mp} -clock {X15/*}
...
mark_flip_flop -master_latch {Mn} -slave_latch {Sn}
mark_flip_flop -master_latch {Mp} -slave_latch {Sp}
```

Cascode Voltage Switch Logic

Differential cascode voltage switch logic (known as DCVS or CVSL) is a differential logic design style that provides fast switching, low input capacitance, low static power consumption, and complementary functions. DCVS gates contain a pair of cross-coupled PMOS pullup transistors and a set of NMOS transistor stacks that perform the logic. Differential input signals are required, and differential output signals are generated. NanoTime supports analysis of DCVS circuits.

Figure 7-10 shows an example of a DCVS inverter. The circuit performs as both an inverter and a buffer because output signal Z and its complement Zn are both available.

Figure 7-10 DCVS Inverter and Buffer

Figure 7-11 shows two simple DCVS logic gates. The first is a differential 2-input AND and NAND gate with respect to output Z. Similarly, the second gate provides both OR and NOR logic functions. The transistor layout is identical but the assignment of inputs and outputs is different.

Figure 7-11 DCVS Logic Gates

To analyze these circuits, you must mark the differential net pairs for the input and output nets and mark the differential synchronizers. For example, to mark the differential inverter shown in Figure 7-10, use the following commands:

```
foreach_match cvsl_inv -command {
   set_differential { A An }
   set_differential { Z Zn }
   mark_differential_synchronizer -transistors { Mp0 Mp1 }  \
                                  -tapped_transistors { Mp0 Mp1 }
   mark_weak_pullup -transistors { Mp0 Mp1}
}
```

Marking the cross-coupled PMOS transistors as tapped transistors (with the `-tapped_transistors` option) includes both of the pulldown structures and the cross-coupled PMOS transistors in the same stage, allowing analysis of the delays from any of the inputs to any of the outputs.

Mark the differential logic structures for the gates shown in Figure 7-11 as follows. The statements are identical for both structures, with the exception of the structure name.

```
foreach_match cvsl_and2 -command {
   set_differential { A An }
   set_differential { B Bn }
   set_differential { Z Zn }
   mark_differential_synchronizer -transistors { Mp0 Mp1 }  \
                                  -tapped_transistors { Mp0 Mp1 }
   mark_weak_pullup -transistors { Mp0 Mp1}
}
```

Level Shifter Circuits

Level shifter circuits that are based on a differential design can be analyzed using differential circuit analysis. Using differential circuit analysis instead of dynamic simulation enables the use of high-accuracy options such as signal integrity analysis and active Miller capacitance analysis.

To use this approach, the level shifter must meet the following requirements:

- The level shifter must have two output nets that are differential when the circuit is enabled.
- All transistors connected to a level shifter output net must be unidirectional.
- The signal flow must be set toward the output net for all devices in the level shifter circuit and away from the output net for all fanout devices. Using the strict transistor direction checking flow is strongly recommended.

NanoTime supports level shifters that drive channel-connected fanout devices, including transmission gates, transmission gate latches, differential transmission gate latches, multiplexers, and reconvergent drivers.

See Also

- Setting and Reporting Transistor Direction
- Topology Marking Commands For Level Shifters
- Topology Markings for Cross-Coupled Domino Circuits
- Level Shifter Marking Examples

Topology Marking Commands For Level Shifters

The following procedures provide a general approach to writing the topology marking commands for the output nets of a level shifter.

Note:

If one of the cross-coupled nets interfaces with downstream logic through a driver circuit, the outermost net is considered to be the output net for that side of the circuit. The outermost net must be part of the same channel-connected block (CCB) as the cross-coupled net. See Figure 7-17 for an example.

Level Shifters Without Enable Signals

The following procedure applies to level shifters that do not have an enable signal. In this case, the level shifter output nets are truly differential and their state is always a function of the input nets.

1. Mark the two output nets as differential by using a `set_differential` command.
2. Use the `mark_net` command with the `-differential_cross_coupled` option, as shown in Table 7-1. Write two `mark_net` commands, one for each of the output nets.

Table 7-1 Level Shifter Marking Commands for Always-Enabled Output Net

Description	Option for the mark_net command	Argument of the option
No clamped state (level shifter output is always enabled)	-differential_cross_coupled	none

Level Shifters With Enable Signals

The following procedure applies to level shifters that have an enable signal.

1. Mark the two output nets as differential by using them as the two arguments in the `set_differential` command.
2. Determine whether the enable signal (also known as an override signal) is high (rising) or low (falling) when the level shifter is disabled.

   The transistor or transistors that force an output net to be clamped high or low when the level shifter is disabled are said to *override* the differential nature of the output nets. When the level shifter is disabled, the output nets might or might not be differential.
3. Follow steps 4 through 6 for the two output nets separately. The goal is to write two `mark_net` commands, one for each of the output nets.
4. If the output net is floating when the level shifter is disabled, use the `mark_net` command with the `-differential_cross_coupled` option, as shown in Table 7-2.

Table 7-2 Level Shifter Marking Commands for Floating Output Net

Description	Option for the mark_net command	Argument of the option
No clamped state (output floating)	-differential_cross_coupled	none

5. If the output net is clamped low when the level shifter is disabled, use the `mark_net` command with the options listed in Table 7-3. You must use all three options.

Table 7-3 Level Shifter Marking Commands for Output Net With Low Clamped State

Description	Option for the mark_net command	Argument of the option
Indication that the clamped state is low	-differential_cross_coupled_clear	none
Pin(s) that clamp the output low	-differential_override_clear	MOS gate(s)
Pin(s) that enable level shifter functionality (release the output from the clamped state)	-differential_override_preset	MOS gate(s) of opposite type

6. If the output net is clamped high when the level shifter is disabled, use the `mark_net` command with the options listed in Table 7-4. You must use all three options.

Table 7-4 Level Shifter Marking Commands for Output Net With High Clamped State

Description	Option for the mark_net command	Argument of the option
Indication that the clamped state is high	-differential_cross_coupled_preset	none
Pin(s) that clamp the output high	-differential_override_preset	MOS gate(s)
Pin(s) that enable level shifter functionality (release the output from the clamped state)	-differential_override_clear	MOS gate(s) of opposite type

7. Repeat steps 4 through 6 for the second output net.
8. Observe the following requirements for the override nets:
   - The gates of all enable (override) transistors for both output nets must be connected to the same net, which must be found within two channel-connected blocks of the output nets.
   - The pins specified as arguments of the `-differential_override_clear` and `-differential_override_preset` options must be of opposite types (NMOS or PMOS).

     For example, if the enable signal rises and turns on an NMOS transistor that pulls the output net low, thus disabling the level shifter operation, the argument of the `-differential_override_clear` option is the gate pin of that NMOS transistor. The argument of the corresponding `-differential_override_preset` option must then

contain PMOS gate pins, which are the transistors that must be turned on to release the clamped (low) state and allow normal level shifter operation.

If you specify more than one pin as the argument for the `-differential_override_clear` or `-differential_override_preset` options, timing arcs are created from each of those pins to the corresponding output net. If the delays from the pins are likely to be similar, specifying more than one pin in the argument might increase runtime without providing much accuracy benefit.

Reducing Pessimism for Complex Input Nets

In some designs, an input signal (which might be one of the logic inputs or the enable input) might affect only one of the output signals.

To reduce pessimism of the maximum delay paths through the input signal that affects both output signals, you can apply a false path logic constraint. This is necessary only if there is at least one input signal that affects only one of the output signals.

For example, consider a design in which input net in1 affects both output nets out1 and out2. Input net in2 affects only net out2 (and not net out1) through some combinational logic that also has net in1 as an input. The combinational logic interfaces with the CCB of net out2 through net comb_out. Specify the following false path constraint to reduce pessimism:

```
set_false_path -through in1 -through comb_out -through out2
```

See Also

- Level Shifter Circuits
- Topology Markings for Cross-Coupled Domino Circuits
- Level Shifter Marking Examples

Topology Markings for Cross-Coupled Domino Circuits

For some designs, NanoTime might automatically recognize parts of a level shifter as a domino precharge or flip-flop topology.

As described in the previous section, you can manually mark a cross-coupled circuit in a level shifter by using the `mark_net` command with options specific to level shifters. Alternatively, you can allow the topology to be automatically recognized or you can use the `mark_precharge` command to manually mark the topology.

However, the NanoTime tool honors only one set of topology markings for a circuit.

The following guidelines apply to potential topology marking conflicts:

- User-specified marking commands take precedence over automatic topology recognition.
- If the `mark_net` and `mark_precharge` commands are both specified for the same structure, NanoTime retains the level shifter markings from the `mark_net` command and removes the markings from the `mark_precharge` command.
- If NanoTime removes topology markings from a `mark_precharge` command or from automatic topology recognition, the tool also removes any associated timing checks. If the timing checks are needed, you must create user-specified timing checks to replace them.

For example, the level shifter in Figure 7-12 can be marked in the following ways:

- Using domino precharge marking commands, which automatically creates timing checks:

```
mark_precharge -type nmos -evaluate_net {z0 z1} -evaluate_clock Mn4 \
               -precharge_clock {Mp1 Mp1} -evaluate_data {Mn2 Mn3} \
               -keeper_data {Mp2 Mp3} -keeper_other {Mn0 Mn1} \
               -latch_type flop
```

- Using level shifter marking commands:

```
set_differential {z0 z1}
mark_net z1 -differential_cross_coupled_preset  \
            -differential_override_preset Mp0/G \
            -differential_override_clear Mn4/G
mark_net z0 -differential_cross_coupled_preset  \
            -differential_override_preset Mp1/G \
            -differential_override_clear Mn4/G
```

  If you want to have timing checks for the precharge circuit, you must create them with the `create_timing_check` command, as shown in the following example:

```
create_timing_check 0.0 -setup -rise_to Mn2/G -from Mn4/G
create_timing_check 0.0 -setup -rise_to Mn3/G -from Mn4/G
```

Figure 7-12 Level Shifter With Domino Precharge Circuit

See Also

- Level Shifter Circuits
- Topology Marking Commands For Level Shifters
- Level Shifter Marking Examples

Level Shifter Marking Examples

These examples demonstrate how to mark level shifters with different topologies.

In these tables, the X character in the column for a specific output net (out or out_b) indicates that the option on the same line must be used for that net. For options that require an argument, the argument appears in place of the X character. If the column is blank, the option should not be used for that net.

Level Shifter Without Enable or Clamp Transistors

Table 7-5 applies to the level shifter in Figure 7-13, which has no enable transistor and no clamped output state.

Table 7-5 Level Shifter Marking Commands for Output Nets

Clamp state	Description	Option for the mark_net Command	Net out	Net out_b
None	No clamped state (output is floating)	-differential_cross_coupled	X	X

Figure 7-13 Level Shifter Without Enable or Clamp Transistors

Mark the level shifter in Figure 7-13 as follows:

```
set_differential {out out_b}
mark_net out -differential_cross_coupled
mark_net out_b -differential_cross_coupled
```

NanoTime automatically detects the inverter between the in and in_b nets and includes it in the analysis. Therefore, do not include a `set_differential` command for the input nets in and in_b.

Level Shifter With Enable Transistor and NAND Clamp

Table 7-6 applies to the level shifter in Figure 7-14.

Table 7-6 Level Shifter Marking Commands for Output Nets

Clamp state	Description	Option for the mark_net command	Net out	Net out_b
None	No clamped state (output is floating)	-differential_cross_coupled	X	
0	Indication that clamped state is low Pin(s) to clamp the output low Pin(s) to release the low state (which enables level shifter function)	-differential_cross_coupled_clear -differential_override_clear -differential_override_preset		X Xen/Mn*/g Mp5/g

Figure 7-14 Level Shifter With Enable Transistor and NAND Clamp

Mark the level shifter in Figure 7-14 as follows:

```
set_differential {out out_b}
mark_net out   -differential_cross_coupled
mark_net out_b -differential_cross_coupled_clear       \
               -differential_override_preset Mp5/g     \
               -differential_override_clear  Xen/Mn*/g
```

Level Shifter With Enable and Clamp Transistors

Table 7-7 applies to the level shifter in Figure 7-15.

Table 7-7 Level Shifter Marking Commands for Output Nets

Clamp state	Description	Option for the mark_net command	Net out	Net out_b
0	Indication that clamped state is low	-differential_cross_coupled_clear	X	X
	Pin(s) to clamp the output low	-differential_override_clear	Mn3/g	Mn4/g
	Pin(s) to release the low state (which enables level shifter function)	-differential_override_preset	Mp5/g	Mp5/g

Figure 7-15 Level Shifter With Shared Enable Transistor

Mark the level shifter in Figure 7-15 as follows:

```
set_differential {out out_b}
mark_net out_b -differential_cross_coupled_clear    \
               -differential_override_preset Mp5/g  \
               -differential_override_clear  Mn4/g
mark_net out   -differential_cross_coupled_clear    \
               -differential_override_preset Mp5/g  \
               -differential_override_clear  Mn3/g
```

Level Shifter With Cross-Coupled PMOS and NMOS Transistors

Table 7-8 applies to the level shifter in Figure 7-16. NanoTime automatically detects the cross-coupled NMOS and PMOS structures.

Table 7-8 Level Shifter Marking Commands for Output Nets

Clamp state	Description	Option for the mark_net command	Net out	Net out_b
0	Indication that clamped state is low	-differential_cross_coupled_clear	X	X
	Pin(s) to clamp the output low	-differential_override_clear	Mn2/g	Mn1/g
	Pin(s) to release the low state (which enables level shifter function)	-differential_override_preset	Mp5/g	Mp5/g

Figure 7-16 Level Shifter With Cross-Coupled PMOS and NMOS Transistors

Mark the level shifter in Figure 7-16 as follows:

```
set_differential {out out_b}
mark_net out    -differential_cross_coupled_clear    \
                -differential_override_preset Mp5/g  \
                -differential_override_clear  Mn2/g
mark_net out_b  -differential_cross_coupled_clear    \
                -differential_override_preset Mp5/g  \
                -differential_override_clear  Mn1/g
```

Level Shifter With Current Follower Output

Table 7-9 applies to the level shifter in Figure 7-17. The net that drives downstream logic is net out_x and not net out_b. Therefore the `mark_net` and `set_differential` commands must be applied to net out_x.

Table 7-9 Level Shifter Marking Commands for Output Nets

Clamp state	Description	Option for the mark_net command	Net out	Net out_b
None	No clamped state (output is floating)	-differential_cross_coupled	X	
0	Indication that clamped state is low Pin(s) to clamp the output low Pin(s) to release the low state (which enables level shifter function)	-differential_cross_coupled_clear -differential_override_clear -differential_override_preset		X Mn4/g Mp5/g

Figure 7-17 Level Shifter With Current Follower Output

Mark the level shifter in Figure 7-17 as follows:

```
set_differential {out out_x}
mark_net out   -differential_cross_coupled
mark_net out_x -differential_cross_coupled_clear    \
               -differential_override_preset Mp5/g  \
               -differential_override_clear  Mn4/g
```

See Also

- Level Shifter Circuits
- Topology Marking Commands For Level Shifters
- Topology Markings for Cross-Coupled Domino Circuits

8

Timing Constraints

Constraints such as clock characteristics, input and output delays, timing exceptions, and logic constraints set the conditions for timing analysis.

Timing constraints are described in the following sections:

- Input and Output Timing
- Logic Constraints
- Conditional Analysis
- Timing Exceptions
- Restricting Analysis to Specified Paths
- The check_design Command

Input and Output Timing

To perform a timing analysis, NanoTime needs information about the timing conditions at the inputs and outputs. You must specify these timing constraints because they cannot be determined from the netlist. You specify the input and output timing constraints with the `set_input_delay` and `set_output_delay` commands, together with the clock information set with the `create_clock` command, as illustrated in Figure 8-1.

Figure 8-1 Input and Output Timing Constraints

The `set_input_delay` command specifies the minimum and maximum amount of delay from a clock edge to the arrival of a signal at a specified input port. NanoTime uses this information to check for timing violations in the transitive fanout from that input port.

Similarly, the `set_output_delay` command specifies the minimum and maximum amount of delay between the output port and the external sequential device that captures data from that output port. This setting establishes the times at which signals must be available at the output port to meet the setup and hold requirements of the external element.

At the inputs of the design, the transition times depend on the external drivers. You can specify the external driver characteristics with the `set_drive` command, or explicitly specify the input transition times with the `set_input_transition` command, as illustrated in Figure 8-2.

Figure 8-2 Input Transition Time, Input Drive, and Output Load

See Also

- Input Delay
- Output Delay
- Input Transition Time
- Input Drive Characteristics

Input Delay

The `set_input_delay` command specifies the amount of delay from a clock edge that launches data to the arrival of the data at the input of the design, for a specified list of inputs.

For example, the following command sets an input delay of 1.2 time units relative to the rising edge of CLK1 for all input ports in the design:

```
nt_shell> set_input_delay -clock CLK1 1.2 [all_inputs]
```

The `-clock` option specifies the clock from which the delay applies. Use the `-clock_fall` option if the delay is from the falling edge of the clock. A clock does not need to be specified

for a purely combinational design. If the clock is a differential clock, you must use the reference side of the differential pair with the `-clock` option.

If the external launch element is a domino precharge gate, use the `-level_sensitive` option. In that case, when NanoTime performs setup and hold checking, it assumes that the specified input delay applies to the launch of data on the closing edge of the clock, rather than the opening edge.

The `-asynchronous` option specifies that the input is not tied to a clock. This option is required to activate recovery or removal timing checks, and consequently disqualifies synchronous timing checks from being evaluated (which includes any latch, flip-flop, precharge, and gated clock setup and hold checks whose constraint pin is driven by a path from this asynchronous input). The `-clock` option cannot be used with this option.

The `-rise` or `-fall` option specifies that the input delay value applies only to the specified edge type arriving at the input.

The `-min` or `-max` option specifies that the input delay value is the worst-case minimum (early) or maximum (late) arrival time. In the absence of these options, the same value is used for both the minimum and maximum.

The `-add_delay` option adds the delay information to the existing input delay information, instead of overwriting it. Use this option in multiple `set_input_delay` commands to capture information about multiple paths leading to an input port that are relative to different clocks or clock edges.

The `-network_latency_included` option causes the clock network latency to be included in the input delay value when ideal clocking is being used. If the option is not used, the network latency is added to the input delay value. The `-source_latency_included` option causes the clock source latency to be included in the input delay value. In the absence of this option, the clock source latency of the related clock is added to the input delay value.

The following example sets the input delay for the bidirectional port INOUT1. The input signal arrives at INOUT1 2.5 units after the falling edge of CLK1. The `set_output_delay` command specifies the output delay for the port.

```
nt_shell> set_input_delay 2.5 -clock CLK1 -clock_fall [get_ports INOUT1]
nt_shell> set_output_delay 1.4 -clock CLK2 [get_ports INOUT1]
```

In the following example, there are three paths to input port IN1. The first path is relative to the rising edge of CLK1. The second path is relative to the falling edge of CLK1. The third path is relative to the falling edge of CLK2. The `-add_delay` option adds the new input delay settings without overwriting the old ones.

```
nt_shell> set_input_delay 2.2 -max -clock CLK1 -add_delay {IN1}
nt_shell> set_input_delay 1.7 -max -clock CLK1 \
          -clock_fall -add_delay {IN1}
nt_shell> set_input_delay 4.3 -max -clock CLK2 \
          -clock_fall -add_delay {IN1}
```

To get a report on input delays set on ports, use the `report_port` command with the `-input_delay` option. To remove input delay values, use the `remove_input_delay` command.

See Also

- Output Delay
- Input Transition Time
- Input Drive Characteristics

Output Delay

The `set_output_delay` command specifies the amount of time it takes for a signal to travel from an output port to the external sequential element that captures the data on a clock edge.

For example, the following command sets an output delay of 1.7 relative to the rising edge of CLK1 for all output ports in the design:

```
nt_shell> set_output_delay -clock CLK1 1.7 [all_outputs]
```

The `-clock` option specifies the clock to which the delay applies. Use the `-clock_fall` option if the delay is to the falling edge of the clock. A clock does not need to be specified for a purely combinational design. If the clock is a differential clock, you must use the reference side of the differential pair with the `-clock` option.

If the external capture element is a level-sensitive latch, use the `-level_sensitive` option. In that case, when NanoTime performs setup and hold checking, it assumes that the specified output delay applies to the capture of data on the closing edge of the clock, rather than the opening edge.

The `-rise` or `-fall` option specifies that the output delay value applies only to the specified edge type at the data output.

The `-min` or `-max` option specifies that the output delay value is the worst-case minimum (shortest path) or maximum (longest path) delay to the external latch. You can set different minimum and maximum times; NanoTime uses the worst-case value for each timing check. If neither option is present, the same value is used for both the minimum and maximum.

The `-add_delay` option adds the delay setting to the existing output delay, instead of overwriting it. Use this option to capture information about multiple paths leading from an output port that are relative to different clocks or clock edges.

The `-network_latency_included` option causes the clock network latency to be included in the output delay value when ideal clocking is being used. Propagated clocking is more commonly used than ideal clocking.

The `-source_latency_included` option causes the clock source latency to be included in the output delay value. In the absence of this option, the clock source latency of the related clock is added to the input delay value.

In the following example, there are three paths from output port OUT1. One of the paths is relative to the rising edge of CLK1. Another path is relative to the falling edge of CLK1. The third path is relative to the falling edge of CLK2. The `-add_delay` option adds the new output delay settings without overwriting the old ones.

```
nt_shell> set_output_delay 2.2 -max -clock CLK1 -add_delay {OUT1}
nt_shell> set_output_delay 1.7 -max \
          -clock CLK1 -clock_fall -add_delay {OUT1}
nt_shell> set_output_delay 4.3 -max \
          -clock CLK2 -clock_fall -add_delay {OUT1}
```

To get a report on output delays set on ports, use the `report_port` command with the `-output_delay` option. To remove output delay values, use the `remove_output_delay` command.

See Also

- Input Delay
- Input Transition Time
- Input Drive Characteristics

Input Transition Time

The transition time of a signal (also known as slew) is the amount of time it takes for the signal to change from one logic state to the other. The `set_input_transition` command specifies a fixed transition time for a list of input ports or bidirectional ports. This transition time affects the calculation of delays for nets and cells in the transitive fanout of the port.

For example, the following command specifies that ports matching the pattern DATA_IN* have a transition time of 0.75 time units:

```
nt_shell> set_input_transition 0.75 [get_ports DATA_IN*]
```

The `-rise` or `-fall` option specifies that the transition time applies only to the specified edge type arriving at the input.

The `-min` or `-max` option specifies that the transition time is the worst-case minimum or maximum transition time. You can set different minimum and maximum times and NanoTime uses the worst-case value for each timing check.

You can specify the transition time relative to a clock by using the `-clock` option. This option makes the transition time apply only to external paths driven by the specified clock. For a falling-edge clock, use the `-clock_fall` option together with the `-clock` option.

The `-rail_voltage` option specifies the rail voltage used to generate the input transition waveform. If you do not use this option, NanoTime generates the input waveform based on the global rail voltage defined by the `oc_global_voltage` variable. By default, this variable is set to –1.0. To ensure reasonable generation of input waveforms in the absence of the `-rail_voltage` option, it is a good idea to set the `oc_global_voltage` variable to the main rail voltage early in the design flow, for example, before the `set_technology` or `check_topology` commands.

To view the transition times that have been set on ports, use the `report_port -drive` command.

If no transition time is set for a port, NanoTime uses the transition times determined by the values of the `sim_transition_*_**` variables, where * represents `max` or `min` and ** represents `fall` or `rise`. These variables default to 0.05 nanoseconds.

The `sim_transition_min_limit` and `sim_transition_max_limit` variables specify the minimum and maximum allowable transition times used for simulation, in nanoseconds.The default minimum is 0.003 ns and the default maximum is 10 ns. Any value outside of this range, whether calculated by NanoTime or specified with the `set_input_transition` command, is forced to the allowed minimum or maximum value for analysis.

See Also

- Input Delay
- Output Delay
- Input Drive Characteristics

Input Drive Characteristics

The `set_drive` command sets the resistance of the driver that is driving one or more input ports or bidirectional ports in the current design. You specify the resistance value (in the resistance units of the technology) and the input ports or bidirectional ports in the design to which that resistance value applies. You can optionally restrict the setting to rising or falling edges. Also, you can optionally specify different minimum and maximum worst-case values.

The following example sets the drive resistance to 0.25 on all input ports in the current design, then displays the ports and their drive resistance values.

```
nt_shell> set_drive 0.25 [all_inputs]
1
```

```
nt_shell> report_port -drive
 ...
Resistance

Input Port   Min Rise Min Fall Max Rise Max Fall
------------ -------- -------- -------- --------
BUS[0]          0.250    0.250    0.250    0.250
BUS[1]          0.250    0.250    0.250    0.250
selA            0.250    0.250    0.250    0.250
 ...
```

NanoTime models the external driver as the voltage supply connected to a resistor for a rising transition, or ground connected to a resistor for a falling transition. It uses the resistance value and the capacitive load of the port to calculate the wire delay of the port. A higher drive resistance means less drive capability and a longer delay at the input port. Conversely, a lower drive resistance results in a smaller external delay. The specified resistance must be greater than zero.

If a transition time has also been set on the port with the `set_input_transition` command, NanoTime models the external driver as a voltage ramp driving the resistor and connected to the port.

To remove the drive resistance set on a port, use the `remove_drive_resistance` command.

See Also

- Input Delay
- Output Delay
- Input Transition Time

Logic Constraints

The NanoTime tool infers some types of logic constraints from the design topology. You can specify additional logic constraints on pins or nets to eliminate analysis of logic conditions that never occur in the actual circuit. This might save runtime and produce more accurate results. Use the `set_logic_constraint` command to specify logic constraints.

For example, the following command defines the set of control signals ct0 through ct3 to be one-hot. This means that at any given time, exactly one signal is at logic level 1 and the others are at logic level 0.

```
nt_shell> set_logic_constraint -one_hot [get_nets {ct0 ct1 ct2 ct3}]
1
```

You can set the following logic constraints on a list of pins or nets:

- `-one_hot:` At any time, exactly one of the pins or nets is at logic 1.
- `-one_off:` At any time, exactly one of the pins or nets is at logic 0.
- `-at_most_one_hot:` At any time, no more than one of the pins or nets is at logic 1.
- `-at_most_one_off:` At any time, no more than one of the pins or nets is at logic 0.
- `-invert:` The first pin or net in the argument list is the input of an inverter; the second is the output of the inverter. Exactly two objects must be specified.
- `-equal:` At any time, all of the pins or nets have the same value, either logic 0 or logic 1.
- `-nand:` The last pin or net in the argument list is the output of a NAND gate; the other pins or nets are the inputs of the NAND gate.
- `-nor:` The last pin or net in the argument list is the output of a NOR gate; the other pins or nets are the inputs of the NOR gate.

Setting or removing logic constraints changes the design and invalidates any existing analysis results. You must run the `check_design` and `trace_paths` commands again to perform a new timing analysis.

Specific nets might be involved in multiple logic constraints, in which case conflicts could arise. The order of precedence for considering overlapping logic constraints is as follows, from highest to lowest priority:

1. User-specified inversion constraints
2. User-specified or topology-induced mutual exclusion constraints
3. Topology-induced inversion constraints

For example, if a net is included in both a user-specified inversion constraint and a mutual exclusion constraint, the user-specified inversion constraint is honored for that net and the mutual exclusion constraint is ignored for that net. However, the mutual exclusion constraint still applies to any other nets in its definition.

You can elevate the priority of a user-specified logic constraint during signal integrity analysis by using the `-crosstalk_force` option of the `set_logic_constraint` command in addition to the option that defines the constraint.

To report logic constraints that have been set with the `set_logic_constraint` command, use the `report_logic_constraint` command. The generated report indicates whether a constraint is a forced constraint.

To remove logic constraints that have been set with the `set_logic_constraint` command, use the `remove_logic_constraint` command.

See Also

- Precedence For Aggressor Logic Constraints

Conditional Analysis

You can set conditions on the timing analysis in several ways.

This section contains the following topics:

- Case Analysis
- Conditional Delay Arcs and Timing Checks
- Conditional Receiver Capacitance in CCS Delay Models

Case Analysis

To apply a logic state to the analysis, use the `set_case_analysis` command to specify the logic state. For example, if you are analyzing the test mode of a circuit , you can assert the test mode input to a logic 1. You can also restrict logic conditions by using the `set_logic_constraint` command. For example, if only one of a set of control signals is active at any given time, you can define those signals to be a one-hot set. See Figure 8-3 for an example. Specifying logic restrictions can reduce runtime and improve accuracy.

Figure 8-3 Case Analysis and Logic Constraints

To restrict the timing analysis to a specific logic condition, use the `set_case_analysis` command. The command set a constant logic value (either 0 or 1) on a port or pin in the design.

For example, you can set logic 1 on an enable input and logic 0 on a test-mode input to analyze the chip in the enabled mode and to eliminate consideration of the test mode circuitry, as follows:

```
nt_shell> set_case_analysis 1 [get_ports EN]
1
nt_shell> set_case_analysis 0 [get_ports TEST]
1
```

NanoTime propagates this constant value forward through the design if the value controls the value of the downstream logic, as shown in the example in Figure 8-4. Propagation of the logic constant works in the forward direction, and also backward through inverters.

Figure 8-4 Propagation of a Logic Constant

When the latch input is in a fixed logic state, you can direct NanoTime to propagate logic constraints to latch nets by setting the `topo_latch_enable_logic_propagation` variable to `true`. NanoTime adds an `equal` logic constraint from the input net to the latch net when they are connected by pass transistors.

Setting or removing a logic constant changes the design and invalidates the current timing results. You must run the `check_design` and `trace_paths` commands again to generate new analysis results.

You can analyze a design for different cases, then merge the resulting timing models into a single model. The operating modes in the final model are derived from the logic settings specified by the `set_case_analysis` command. The default behavior of the model is to assume that cases are mutually exclusive. If this is not true, use the `set_case_analysis` command with the `-exclude_from_when` option to define those logic conditions.

To report case analysis values that have been set, use the `report_case_analysis` command. For example,

```
nt_shell> report_case_analysis
 ...
Case  Net
----- -------------------------------------------
one   EN
zero  TEST
```

The report shows the values directly set on ports and pins. It does not include information about the propagation of the logic settings into the design.

To get a list of nets that have been set to a constant, use the following command:

```
nt_shell> report_logic_state -valid_only \
          [get_nets -hierarchical        \
          -top_net_of_hierarchical_group]
```

To remove a case analysis setting, use the `remove_case_analysis` command.

The `set_case_analysis` command in the Synopsys Design Constraints (SDC) format supports the setting of a rising or falling transition on a port or pin. This capability is not supported in NanoTime. However, you can remove a specified transition from consideration in NanoTime by using the `set_false_path` command.

Constant Propagation Through a Library Cell

A cell in a Liberty format timing model can contain a function construct, which specifies the state of an output pin as a function of the states of the input pins. NanoTime cannot write function statements into extracted timing models. However, when timing models are read in as part of a design, the tool can read and use function statements to propagate a constant through combinational library cells.

NanoTime does not handle sequential cell logic defined with ff, latch, or state table groups in Liberty format models. In addition, `contention_condition` attributes are ignored. For combinational cells, constant propagation occurs only in the forward direction; the tool does not infer input logic states from output logic states.

Use the `set_case_analysis` command to define a constant logic value (0 or 1) for a specific port or pin. A constant logic value can be assigned to a specific pin or port or to a group of pins or ports. Valid arguments for this command are `0`, `1`, `zero`, and `one`. In the following example, a constant of 1 is assigned to the IN0 port:

```
nt_shell> set_case_analysis 1 IN0
```

Figure 8-5 shows a simple circuit in which the inverter and AND gates are library cells. A value of 0 is set on input IN1, which makes the path from IN0 irrelevant. Therefore the path from IN0 to output OUT is a false path. NanoTime cannot detect the false path without considering the AND gate function. The tool propagates the 0 to the AND gate output and does not trace paths through the gate. As a result, the false path is detected and pruned.

Figure 8-5 Inverter and AND Gate Library Cells

The constant propagation feature handles logic conflicts that might happen when NanoTime propagates logic values for `set_case_analysis` or `set_logic_constraint` commands. When a conflict is detected, NanoTime issues an error message for the failed command:

```
nt_shell> set_case_analysis 1 in0
1
nt_shell> set_case_analysis 0 in0
Error: Constraint conflicts with previous constraints: unable to set net
in0 to logic state 0 (LOGC-005)
0
```

NanoTime does not disable timing arcs across a tristate cell using the associated tristate logic function.

Conditional Delay Arcs and Timing Checks

The NanoTime tool supports the `when` attribute in a library model, which specifies the condition that activates a conditional delay arc or a conditional timing check. The value is a Boolean expression.

Important:
The tool does not support analysis of models with modes defined by the `mode_definition` and `mode_value` attributes.

During the `check_design` phase, NanoTime constructs representations for library cell delay arcs and uses the `when` statement to decide whether to disable the arcs. Disabled delay arcs are not considered during path tracing. If a cell has an input set to a logic constant, NanoTime evaluates the `when` expression of the underlying library cell arcs using the logic constant values. A delay arc is enabled or disabled based on the following rules:

- If the `when` expression evaluates to `true`, NanoTime enables the library cell arc. Any parallel default arcs (arcs that do not have conditions) of the same timing sense are disabled.
- If the `when` expression evaluates to `false`, NanoTime disables the library cell arc.
- If the `when` expression evaluates to X (the don't care state) or to an unknown value, NanoTime enables the library cell arc.

NanoTime skips the creation of conditional timing checks based on the same rules. Skipped timing checks are not considered during path tracing.

Example 8-1 is a multiplexer named MUX2 with two inputs, `I0` and `I1`. From the select signal `S` to the output `Q`, there are three timing arcs. Assume that `set_case_analysis 0` is applied to input `I0` and `set_case_analysis 1` is applied to input `I1`. The `negative_unate` arc is disabled because the `when` expression `"I0&!I1"` is `false`. The `positive_unate` arc remains enabled because the `when` expression `"!I0&I1"` is `true`. The default arc is disabled because the `when` expression of the `positive_unate` arc evaluates to `true`.

Example 8-1 Multiplexer With Conditional Cell Arcs

```
cell (MUX2) {
...
     pin (I0) {
          direction : input;
          ...
     }
     pin (I1) {
          direction : input;
          ...
     }
     pin (S) {
          direction : input;
          ...
     }
     pin(Q) {
          direction : output;
          function : "((S I1) + (!S I0))";
          timing () {
               related_pin : "S";
               when : "!I0&I1";
               timing_sense : positive_unate;
               ...
          }
          timing () {
               related_pin : "S";
               when : "I0&!I1";
               timing_sense : negative_unate;
               ...
          }
          timing () {   /* This is the default arc */
               related_pin : "S";
               timing_sense : non_unate;
               ...
          }
     ...
     }
     ...
}
```

Reporting Conditional Statements

Use the following commands to understand the usage of `when` statements:

- The `report_disable_timing` command lists timing arcs that have been disabled because of a `when` statement.

  For the multiplexer in Example 8-1, the disable timing report is generated by the following command:

```
nt_shell> report_disable_timing "Xmux2" -nosplit

****************************************
Report : disable_timing
Design : test
****************************************
Flags: C Conditional arc
       d default arc

Cell or Port  From  To  Sense           Flag  Reason
------------  ----  --  --------------- ----  ----------
Xmux2          S    Z   negative_unate    C   I0&!I1 = 0
Xmux2          S    Z   non_unate         d
```

- The `report_lib` command reports the `when` statement of a library timing arc.

  For the multiplexer in Example 8-1, the library cell timing arcs are reported by the following command:

```
nt_shell> report_lib -timing_arcs

****************************************
Report : library
Design : Test
...
****************************************
Time Unit                  : 1 ns
Capacitance Unit           : 1 pF
Voltage Unit               : 1 V
Resistance Unit            : 1 kohm
Current Unit               : 1 mA

Operating Conditions:
  Name              Process      Temp    Voltage
  --------------- --------- --------- ----------
  DEFAULT             1.000    50.000      0.000
```

```
Library Cells:
  Attributes:
      b - black box (function unknown)
      d - dont_touch
      s - state table
      u - dont_use
      A - abstracted timing model
      E - extracted timing model
      I - Interface timing spec model (ITS)
      S - Stamp timing model
      Q - Quick timing model (QTM)

                     Arc
Lib Cell  Attributes  #  Sense/Type     Arc From   Arc To   When
-------- ---------- --- --------------- ---------- -------- -----
MUX2          --      0  positive_unate   S         Q       !I0&I1
MUX2          --      1  negative_unate   S         Q       I0&!I1
MUX2          --      2  non_unate        S         Q
```

Conditional Receiver Capacitance in CCS Delay Models

The `when` attribute is provided in the pin-based `receiver_capacitance` group to support conditional data modeling in composite current source (CCS) delay models.

NanoTime evaluates the `when` statement for CCS receiver models according to the following rules:

- If a pin has multiple conditional models, any of the models with `when` statements that evaluate to `true` are enabled; models with `when` statements that evaluate to `false` are disabled. If any conditional model is enabled, the default model is disabled. In this case, the default model does not have a `when` statement defined.
- If a pin has only one conditional model, which evaluates to true or X, the model is marked as enabled; otherwise, it is marked as disabled.
- If a pin only has one nonconditional model, the model is marked as enabled.

The CCS receiver model selected has impact when calculating delays using CCS. All disabled models are not considered during calculation, which speeds up the calculation time.

An example of a conditional CCS receiver model is shown in Example 8-2. If `set_case_analysis 1` is applied to pin `I`, then the `when` statement is true, and the receiver capacitance for condition 1 is enabled.

Example 8-2 Pin-Based Conditional CCS Receiver Model

```
cell(my_cell) {
     ...
     pin(A) { /* pin-based receiver model defined for pin 'A' */
          direction : input;
          /* receiver capacitance for condition 1 */
          receiver_capacitance() {
               when : "I";
               receiver_capacitance1_rise(LTT1) {
                    values("1, 2, 3, 4");
               }
               receiver_capacitance1_fall(LTT1) {
                    values("1, 2, 3, 4");
               }
               receiver_capacitance2_rise(LTT1) {
                    values("1, 2, 3, 4");
               }
               receiver_capacitance2_fall(LTT1) {
                    values("1, 2, 3, 4");
               }
          }
          /* receiver capacitance for condition 2 */
          receiver_capacitance() {
               when : "!I";
               receiver_capacitance1_rise(LTT1) {
                    values("11, 12, 13, 14");
               }
               receiver_capacitance1_fall(LTT1) {
                    values("11, 12, 13, 14");
               }
               receiver_capacitance2_rise(LTT1) {
                    values("11, 12, 13, 14");
               }
               receiver_capacitance2_fall(LTT1) {
                    values("11, 12, 13, 14");
               }
          }
     }
     ...
} /* end cell */
```

Timing Exceptions

During path tracing, NanoTime analyzes the delays of timing paths throughout the design and evaluates timing checks. A timing check is a comparison of the arrival times of signals at two different locations.

NanoTime defines many default timing checks following rules based on the recognized topologies. To modify the default behavior for specific circumstances, you can use timing exception commands to change something about either the path tracing operation or the way that timing checks are evaluated.

Paths excluded from timing analysis because of timing exceptions are still considered for signal integrity analysis.

This section contains the following topics:

- Timing Exception Concepts
- False Path Exceptions
- No-Check Exceptions
- Multicycle Path Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions
- Exclude Reference Path Exceptions
- Reporting Exceptions

Timing Exception Concepts

The types of path-based timing exceptions that you can specify, in order from highest to lowest priority, are as follows:

- False path – A path that can never propagate data due to the logic of the design. NanoTime does not trace these paths.
- No-timing-check path – A path for which one or more timing checks is disabled.
- Multicycle path – A path that takes multiple clock cycles for the data to propagate from the startpoint to the endpoint, rather than one cycle. NanoTime needs this information to properly evaluate the timing checks.
- Same-phase path or no-same-phase path – A path that captures data in the same phase or the next phase as the data launch, possibly contrary to the global behavior setting or settings made on the objects in the path.

- Common required capture path – A launch path that requires the capture path to pass through common nets or pins.

A timing exception command can apply to a single path or to a group of related paths, such as all paths from one clock domain to another, or all paths that end at a specified point. You can report the exceptions that have been set with the `report_exceptions` command.

The timing exception commands are typically set before the `check_design` command because they affect the way NanoTime analyzes the design database to prepare for path tracing. You can use a timing exception command after the `check_design` command only if you use the `-use_existing_timing_points` option, which enables the command to be applied to the existing database but does not allow changes to the database.

In cases of conflicting exceptions for a path, the timing exception priority determines the behavior. Multiple exception settings that are not in conflict with each other can apply to the same path. The following cases are examples of multiple exceptions:

- There is no conflict between the `set_multicycle_path` and `set_same_phase_path` commands, so both can apply to the same path. NanoTime applies the same-phase setting first, followed by the multicycle path setting to move the capture edge a number of cycles away from the position determined by the same-phase setting.
- There is no conflict between the `-setup` and `-hold` options for any specific exception.
- There is no conflict between the `set_exclude_reference_path` command and the other exception commands.

Specifying Timing Exceptions

The timing exception commands offer a common set of options that specify the paths for which to apply the exceptions: the `-from`, `-to`, `-through`, `-rise_from`, `-rise_to`, `-rise_through`, `-fall_from`, `-fall_to`, and `-fall_through` options. You can use these options alone or in combination.

The simplest approach is to explicitly specify the startpoint using the `-from` option and the endpoint using the `-to` option. These commands affect all paths that start at the specified startpoint and end at the specified endpoint. For example,

```
nt_shell> set_false_path -from PORTA -to PORTZ
nt_shell> set_false_path -from FFB1/CP -to FFB2D
nt_shell> set_multicycle_path -setup 2 -from FF4 -to FF5
```

You can also specify only a startpoint or only an endpoint, as in the following examples, which might result in a large number of paths that meet the criteria.

```
nt_shell> set_multicycle_path -setup 2 -from [get_clocks CLK1]
nt_shell> set_multicycle_path -setup 2 -to   [get_pins mod1.m5.ck]
```

A startpoint can be any of the following objects:

- Clock (all registers and primary inputs are used as startpoints)
- Primary input port
- Sequential cell (all input pins and bidirectional pins are used as startpoints)
- Clock input pin of a sequential cell
- Data pin of a level-sensitive latch
- Pin that has an input delay

An endpoint can be any of the following objects:

- Clock (all registers and primary outputs are used as endpoints)
- Primary input port
- Sequential cell (all paths ending at pins of the cell are used as endpoints)
- Clock input pin of a sequential cell
- Data pin of a level-sensitive latch
- Pin that has an output delay

You can use the `-through` option to specify all paths that pass through one or more specific pins, ports, or cells. However, this method is more computationally intensive, especially in larger designs.

You can use multiple `-through`, `-rise_through`, and `-fall_through` options in a single command to specify a group of paths. For example, the following command specifies any path that starts at A1, passes through B1 and C1 in that order, and ends at D1:

```
nt_shell> set_false_path -from A1 -through B1 -through C1 -to D1
```

The following example specifies any path that starts at A1, passes through either B1 or B2, then passes through either C1 or C2, and ends at D:

```
nt_shell> set_false_path -from A1 -through {B1 B2} -through {C1 C2} -to D
```

The `-consecutive` option means that the specified “from,” “through,” and “to” points must be consecutive, without other points in between. For example,

```
nt_shell> set_false_path -from A1 -through B1 -to C1 -consecutive
```

This causes only the path that starts at A1, passes through B1 on the next arc, and ends at C1 on the next arc to be a false path. Without the `-consecutive` option, paths with other intermediate points would also be set to false paths, such as a path that starts at A1, passes through X1, B1, Y1, and Z1, and ends at C1.

You can provide an additional restriction on the selected timing paths by using the `-rise_from`, `-rise_to`, `-rise_through`, `-fall_from`, `-fall_to`, or `-fall_through` options. Their effects are as follows:

- When the referenced object is not a clock, these options consider only those paths that exhibit a rising or falling transition at the specified object.
- When the referenced object is a clock, these options specify paths based on the launch of data at startpoints or the capture of data at endpoints for a specific edge of the source clock. The transition direction at the path endpoint does not matter.

The `-rise_from`, `-rise_to`, `-fall_from`, and `-fall_to` options take transparency windows into account, as follows:

- The `-rise_to` and `-fall_to` options always refer to the closing edge of the transparency window.
- The `-rise_from` and `-fall_from` options always refer to the opening edge of the transparency window.

To cancel an exception, use the `remove_*` command that corresponds to the original exception-setting command: the `remove_false_path`, `remove_no_timing_check_path`, `remove_multicycle_path`, `remove_no_same_phase_path`, and `remove_same_phase_path` commands.

See Also

- False Path Exceptions
- No-Check Exceptions
- Multicycle Path Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions
- Exclude Reference Path Exceptions

False Path Exceptions

The `set_false_path` command specifies paths in the design for which to disable path tracing. The `set_false_path` command prevents path tracing and timing analysis for the specified paths, and those paths are not saved in the path tracing database. However, NanoTime takes capacitive loading effects from false paths into account during analysis.

To remove false path exceptions set by the `set_false_path` command, use the `remove_false_path` command.

In the following example, the test mode input port named *tmode* should not be analyzed. The following command declares all paths from that input to be false paths.

```
nt_shell> set_false_path -from [get_ports tmode]
```

In this example, a design has two cells, ff12 and ff34, that are never used at the same time. The following commands declare the timing paths between the cells to be false paths.

```
nt_shell> set_false_path -from [get_cells ff12] -to [get_cells ff34]
nt_shell> set_false_path -from [get_cells ff34] -to [get_cells ff12]
```

You can restrict the false path declaration to maximum delay path tracing by using the `-setup` option. Similarly, you can specify minimum delay path tracing by using the `-hold` option.

Important:

The `-setup` and `-hold` options for this command do not refer to setup and hold timing checks.

The following command disables minimum delay checking for endpoints clocked by CLK1:

```
nt_shell> set_false_path -hold -to [get_clocks CLK1]
```

The following command declares as false paths all paths that start from ff1.CP, rise through one or more of U1.Z and U2.Z, fall through one or more of U3.Z and U4.C, and end at ff2/D:

```
nt_shell> set_false_path -from ff1.CP -rise_through {U1.Z U2.Z}\
          -fall_through {U3.Z U4.C} -to ff2/D
```

See Also

- Timing Exception Concepts
- No-Check Exceptions
- Multicycle Path Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions
- Exclude Reference Path Exceptions

No-Check Exceptions

The `set_no_timing_check_path` command disables timing checks at specified locations in the design. This command is different from the `set_false_path` command, which causes the specified paths to be ignored completely and stops all path tracing through them. The `set_no_timing_check_path` command does not stop path tracing; it merely prevents timing checks from being performed.

To remove no-check exceptions set by the `set_no_timing_check_path` command, use the `remove_no_timing_check_path` command.

The following guidelines and restrictions apply:

- The `-through` type options cannot specify a clock object.
- The `-setup` and `-hold` options for this command do not refer to setup and hold timing checks. The `-setup` option restricts the command to maximum delay paths only. The `-hold` option restricts the command to minimum delay paths only.

A timing exception set by the `set_no_timing_check_path` command works as follows:

- NanoTime finds a path segment that matches all of the `-from`, `-through`, and `-to` type options.
- The tool evaluates all timing checks that occur in the path before the location at which the `-from`, `-through`, and `-to` type options are all matched.
- After the options are all matched, timing checks for the next latch (or other topology that includes timing checks) in the path are not evaluated. This might include more than one timing check; the tool disables all of the timing checks that would have been evaluated at that pin if the timing exception were not in place.
- After the latch (or other topology) boundary, the timing exception command has been fulfilled and its application is suspended. Path tracing continues and all timing checks in the rest of the path are evaluated as usual.

If the specified endpoint is a latch, paths that end at that latch are not saved into the path database. Path tracing continues through the latch as if it were transparent, but without any cycle accounting adjustments. In effect, the action of choosing not to evaluate a timing check at a nontransparent latch converts it to a transparent latch.

As an example of a simple no-check exception, a domino precharge circuit might use two different evaluate clocks, ck1 and ck2. By default, NanoTime performs a timing check between data arriving at the precharge node launched by one clock and captured by the other, but this situation cannot occur in the circuit. To prevent all checks of this type from being evaluated, use the following commands:

```
set_no_timing_check_path -from [get_clocks ck1] -to [get_clocks ck2]
set_no_timing_check_path -from [get_clocks ck2] -to [get_clocks ck1]
```

As another example, Figure 8-6 shows a path through four latches. NanoTime automatically performs timing checks for each of the latches, but you might not want to evaluate all of those timing checks.

Figure 8-6 Path With Four Latches

The following command defines the entire path as a false path. In this case, no path tracing is performed and the tool does not evaluate timing checks for any of the latches. Specifying the `-to b` option instead of the `-to e` option would have the same result, because the path tracer would not reach the downstream portions of the path after net b.

```
set_false_path -from a -to e
```

The following command allows path tracing but disables timing checks for latch3, because latch3 is the first structure encountered after all of the path specification options are matched. The timing checks for latch1, latch2, and latch4 are still evaluated.

```
set_no_timing_check_path -from a -through b -to c
```

The following example uses clk1 as the `-to` object. In this case, timing checks for latch1 and latch3 (and any other latches reached by clk1) are not evaluated. However, you could disable timing checks only for one specific latch (say, latch3) by adding the `-through` option to be very specific about the affected path.

```
set_no_timing_check_path -to clk1
```

See Also

- Timing Exception Concepts
- False Path Exceptions
- Multicycle Path Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions
- Exclude Reference Path Exceptions

Multicycle Path Exceptions

A setup check verifies that a signal launched at a path startpoint by a clock edge arrives at the path endpoint soon enough before the capture clock edge. A hold check verifies that the signal arrives late enough not to be captured by the previous capture clock edge.

The clock edges that are considered to be the launch and capture clock edges depend on several conditions, including the relative timing of the launch and capture clocks, the types of sequential elements (transparent latches or flip-flops) in the path, the same-cycle or next-cycle settings for the sequential elements and the path, and the value of the `timing_intersection_transparency` variable.

For example, Figure 8-7 shows a path using transparent latches clocked by the same clock edge. By default, NanoTime assumes same-cycle checking and verifies the timing at the edges shown in the timing diagram in Figure 8-7.

Figure 8-7 Same-Cycle Checking, Single Clock

If the circuit is designed to use next-cycle checking, you should specify that type of checking by one of the methods described in the section Same-Cycle and Next-Cycle Checking Exceptions. In that case, NanoTime performs the setup and hold checking at the edges shown in Figure 8-8.

Figure 8-8 Next-Cycle Checking, Single Clock

The setup and hold timing relationships can be specified by choosing same-cycle or next-cycle checking and by using the `set_multicycle_path` command. If both methods are used, the effects are added together.

The command must specify the paths affected using one or more “from,” “through,” and “to” type options, and a path multiplier.

By default, NanoTime performs each setup (maximum delay) check at the first capture edge that occurs after the launch edge. To modify this check, use the `-setup` option and specify the number of capture edges after launch where the setup check is to be done. The number you specify is the path multiplier.

For example, `-setup 1` has no effect because it causes the setup check to occur at the first capture edge after launch, the same as the default. Use `-setup 2` to perform the check at the second capture edge, one clock cycle later than the default, as shown in Figure 8-9. Use `-setup 3` to perform the setup check two clock cycles later than the default, as shown in Figure 8-10, and so on. The larger the number, the later the setup check is performed and the easier it is for the setup constraint to be satisfied.

Figure 8-9 Multicycle Path Setup = 2, Same-Cycle Checking

Figure 8-10 Multicycle Path Setup = 3, Same-Cycle Checking

Moving the setup check with the `-setup` option does not affect the timing of the hold (minimum delay) check. To modify a hold check, use either the `-hold` or `-hold_cycle` option.

The `-hold` option moves the hold timing check backward in time from the position at one clock cycle before the current setup check. For example, `-hold 0` causes the hold check to occur one clock cycle before the setup check, including a setup check that has been moved with the `-setup` option of another `set_multicycle_path` command. Using `-hold 1` causes the hold check to occur two clock cycles before the setup check, and so on. See Figure 8-11 for an example. The larger the `-hold` number, the earlier the hold check is performed and the easier it is for the hold constraint to be satisfied.

Figure 8-11 Multicycle Path Setup = 3, Hold = 1, Same-Cycle Checking

The `-hold_cycle` option moves the hold timing check forward in time from its default position, independent of any movement of the setup check. Using `-hold_cycle 0` causes the hold check to occur one clock cycle earlier than the default time. Using `-hold_cycle 1` causes the hold check to occur at the default time. Using `-hold_cycle 2` causes the hold check to occur one clock cycle later than the default time, and so on. See Figure 8-12 for an example. The larger the `-hold_cycle` number, the later the hold check is performed and the more difficult it is for the hold timing constraint to be satisfied.

Figure 8-12 Multicycle Path Setup = 2, Hold Cycle = 2, Same-Cycle Checking

When you use the `-hold` option, NanoTime sets the hold check time relative to where the setup check is being performed. However, during path tracing, NanoTime converts the `-hold` setting to the equivalent `-hold_cycle` setting. (You can see this change if you use

the `report_exceptions` command.) As a result, if you change the setup time after you run the `trace_paths` command, the hold check stays the same and is not based on the setup time.

If you omit the `-setup`, `-hold`, and `-hold_cycle` options from the command, NanoTime applies the path multiplier to both the `-setup` and `-hold_cycle` options.

A `set_multicycle_path` command shifts the edges at which checking is done away from whatever positions they would otherwise have, including any same-cycle or next-cycle setting. For example, Figure 8-13 shows a multicycle path setting (setup = 2) applied to a next-cycle check. Compare the shifted checks in Figure 8-13 with the unshifted checks in Figure 8-8.

Figure 8-13 Multicycle Path Setup = 2, Next-Cycle Checking

When different clocks are used for launch and capture, multiple setup relations can exist between the two clocks. The most restrictive relationship (the smallest time difference from a setup launch edge to a setup capture edge) determines the maximum delay requirement for the setup check.

To perform the hold check, NanoTime considers the data paths relative to the setup check. It verifies that the same data signal launched in the setup check arrives late enough not to be captured by the previous capture edge. The most restrictive hold timing relationship determines the hold requirement for the path.

If the launch (start) clock and capture (end) clock are two different clocks with different frequencies, you should consider which clock is being used to move a setup or hold check. By default, NanoTime moves a setup check by a multiplier of the end clock period, but it moves a hold check by a multiplier of the start clock period. If this behavior is not what you want, use the `-start` or `-end` option to explicitly specify which clock period to use for adjusting the setup or hold check.

You can optionally use the `-rise` or `-fall` option to restrict the scope of the command to just rising edges or just falling edges at the datapath endpoint. Otherwise, the command affects both rising and falling edges.

See Also

- Timing Exception Concepts
- False Path Exceptions
- No-Check Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions
- Exclude Reference Path Exceptions

Same-Cycle and Next-Cycle Checking Exceptions

If the output of a latch has a combinational path to the input of another latch, and if the opening edge of the clock at the upstream (launch) latch occurs at the same time as the opening edge at the downstream (capture) latch, NanoTime assumes by default that the two latches operate in the same phase and that the capture latch is transparent. The default checking behavior under these conditions is called same-cycle checking.

The launch and capture opening edges are considered to occur at the same time when the edge times of the reference clocks (as defined by the `create_clock` command) occur at the same time, regardless of differences due to uncertainty, latency, or clock network delay.

Under same-cycle checking conditions, for a setup check, data launched at the upstream latch by the opening edge of the launch clock must arrive at the downstream latch before the closing edge of the capture clock *in the same clock cycle*. For a hold check, the same data launched at the upstream latch must arrive at the downstream latch no earlier than the closing edge of the capture clock *in the previous clock cycle*.

Figure 8-14 shows same-cycle setup and hold checking for the case of a single clock used at both the launch and capture latches. A launch event occurs at L1 on the rising edge of the clock. A capture event occurs at L2 and L3 on the falling edge of the clock in the same clock cycle. The setup check verifies that the data arrives soon enough before the setup capture edge. The hold check verifies that the same data arrives late enough not to be captured by the previous capture edge. By default, all latches are assumed to be transparent, so the data passes through latch L2 as if it were a combinational delay element.

Figure 8-14 Same-Cycle Checking

If a downstream latch is designed to capture data in the next clock cycle rather than the same clock cycle, you should specify next-cycle checking so that the timing checks are performed at the correct times. In next-cycle checking, for a setup check, data launched at the upstream latch by the opening edge of the launch clock must arrive at the downstream latch before the closing edge of the capture clock *in the next clock cycle*. For a hold check, the same data launched at the upstream latch must arrive at the downstream latch no earlier than the closing edge of the capture clock *in the same clock cycle*.

Figure 8-15 shows next-cycle setup and hold checking for the same circuit as the one shown in Figure 8-14. A launch event occurs at L1 on the rising edge of the clock at time 0. A capture event occurs at L2 on the falling edge of the clock in the next clock cycle. Data launched from L2 in the next clock cycle must arrive before the capture edge in the next clock cycle after that.

Figure 8-15 Next-Cycle Checking

The possible ways to specify same-cycle or next-cycle checking, in order from highest to lowest priority, are as follows:

- The `set_same_phase_path` and `set_no_same_phase_path` commands (path-based timing exceptions)
- The `set_timing_check_attributes` command with the `-force_same_phase` and `-force_no_same_phase` options (object-based timing exceptions)
- The `set_phasing` command (object-based setting)
- The `timing_default_phasing` variable (default global behavior)

Path-based settings have priority over object-based settings.

The choice between same-cycle and next-cycle checking applies only to cases where the opening edges at the launch and capture latches occur at the same time. By default, if the opening edges at the launch and capture latches occur at different times, next-cycle setting is used, regardless of the same-cycle or next-cycle setting. However, this behavior can be changed by the `timing_intersection_transparency` variable.

If a multicycle path exception is applied to a path with a same-cycle or next-cycle setting, it shifts the capture edge by a specified number of cycles away from the same-cycle or next-cycle position. In other words, the capture edge depends on both the same-cycle or next-cycle setting and the multicycle path setting.

Path-Based Settings

The `set_same_phase_path` and `set_no_same_phase_path` commands affect the checking of specified timing paths in the design.

These commands have options for specifying the affected paths, such as the `-from, -to` and similar options. These paths use same-cycle or next-cycle checking throughout the path, regardless of the settings on objects in the paths. In other words, path-specific settings have priority over object-based settings.

The `set_same_phase_path` and `set_no_same_phase_path` commands are point-to-point timing exception commands, like the `set_false_path` and `set_multicycle_path` commands. They are reported by the `report_exceptions` command.

Object-Based Settings

The `set_phasing` command selects same-cycle or next-cycle checking for specified instances of objects in the design where sequential timing checks are performed, for example, latches, flip-flops, precharge gates, and output ports. This setting overrides the default behavior set with the `timing_default_phasing` variable.

The `set_phasing` command has lower priority than path-based settings made with the `set_same_phase_path` or `set_no_same_phase_path` command.

Using the `set_phasing` command sets the `phasing` attribute on the object to the string `same_cycle` or `next_cycle`.

To undo the effects of the `set_phasing` command, use the `remove_phasing` command.

Use the `set_timing_check_attributes` command to apply same-cycle and next-cycle checking to timing checks instead of objects. A timing check is considered to be an object-based phasing setting, due to its association with objects at the startpoint and endpoint.

Default Phasing

If you want to use next-cycle checking throughout the design or for most of the design, set the `timing_default_phasing` variable to `next_cycle`. NanoTime then uses next-cycle checking throughout the design, except where you have explicitly set it to same-cycle checking.

If you want to use same-cycle checking throughout the design or for most of the design, leave the `timing_default_phasing` variable set to its default of `same_cycle`. NanoTime then uses same-cycle checking throughout the design, except where you have explicitly set it to next-cycle checking.

The `timing_default_phasing` variable can be changed at any time before you run the `trace_paths` command. To change this variable after running the `trace_paths` command, reset the path database by using the `reset_design -paths` command, then set the variable and run the `trace_paths` command again.

Intersection Transparency

By default, NanoTime performs same-cycle checking only when the opening edges at the launch and capture latches occur at the same time. If the two edges occur at different times, NanoTime performs checking in the next succeeding cycle, ignoring the same-cycle or next-cycle settings for the path or objects in the path.

However, if the `timing_intersection_transparency` variable is set to `true`, the opening edges for launch and capture do not have to coincide to perform same-cycle checking. In that case, NanoTime uses same-cycle checking when there is any amount of overlap between the transparency windows at the launch and capture latches. An overlap occurs when the opening edge of the launch clock occurs before the closing edge of the capture clock. The amount of overlap must be greater than zero to cause same-cycle checking. Figure 8-16 shows an example.

Figure 8-16 Intersection Transparency

```
nt_shell> set timing_intersection_transparency true
```

See Also

- Timing Exception Concepts
- False Path Exceptions
- No-Check Exceptions
- Multicycle Path Exceptions
- Exclude Reference Path Exceptions

Exclude Reference Path Exceptions

You can use the `set_exclude_reference_path` command to exclude specific capture paths (reference paths) that are associated with a specific launch path from being checked during path searching.

By default, NanoTime evaluates the worst-case path to the checked point (the launch path) and compares it to the worst-case path to the reference point (the capture path). The two paths are traced independently, which might not be realistic.

For example, Figure 8-17 shows a design in which multiplexers select between clock signals clka and clkb. The launch and capture paths are both referenced to either clka or clkb, and never to a combination of clka and clkb. However, under some conditions, a timing check might use a launch path through clka and a capture path through clkb.

Figure 8-17 Common Capture Path Example

To specify the launch path of interest, the `set_exclude_reference_path` command accepts the same path selection options as other timing exception commands, such as the `-to` option, the `-rise_to` option, and so on.

To specify the capture paths to be excluded, use the following options:

- The `-exclude_reference_through`, `-exclude_reference_rise_through`, and `-exclude_reference_fall_through` options

  These options take an argument list containing either nets or pins. Capture paths that pass through these nets or pins are excluded from path searching. You can restrict analysis to rising-edge or falling-edge signals at the specified locations by using the `-exclude_reference_rise_through` and `-exclude_reference_fall_through` options.

You can use only one instance of any of the `-exclude_reference_through` options in a single `set_exclude_reference_path` command. However, the argument of the option can be a list of nets and pins, which are combined with a logical OR operation.

Clock and port objects are not supported as arguments for these options.

- The `-min` and `-max` options

  By default, the exclude reference exception applies to both minimum and maximum delay launch paths. To restrict the scope, use either the `-min` or `-max` option.

Consider the example in Figure 8-17. The following command specifies that all capture paths that pass through net clkb are excluded from any minimum-delay launch paths that pass through net clka and net xyz. Because of the `-min` option, this requirement does not apply to maximum-delay launch paths, in which case the tool follows the default behavior and finds the worst-case arrival time over all possible capture paths.

```
set_exclude_reference_path -min -through clka \
                           -exclude_reference_through clkb -through xyz
```

An example usage is for a circuit with two clocks (clk1 and clk2) that are applied in a mutually exclusive manner to downstream sequential elements. The following commands exclude timing analysis when a launch path goes through one of the clock nets while the capture path goes through the other clock net:

```
set_exclude_reference_path -through clk1 -exclude_reference_through clk2
set_exclude_reference_path -through clk2 -exclude_reference_through clk1
```

The following usage notes apply:

- If a `set_false_path` command matches all or part of the specified paths, the false path takes precedence because path tracing is prevented.
- If a `set_no_timing_check_path` command matches the location of a launch path checked point, no timing checks occur at that point.
- Dynamic clock simulation (DCS) is not supported. If you use DCS, stop the clock network before the point specified by any of the `-exclude_reference_through` options. NanoTime issues a warning if this exception is specified within the DCS region.
- If all capture paths for a timing check are excluded due to the use of exclude reference exceptions, the tool issues a warning, does not perform the timing check, and reports the timing check as untested with the reason code capture_paths_excluded (CPE).
- When multiple exclude reference exceptions occur in a single path trace, the tool issues a warning and ignores all of the exclude reference point exceptions.

To remove exceptions set by the `set_exclude_reference_path` command, use the `remove_exclude_reference_path` command.

See Also

- Timing Exception Concepts
- False Path Exceptions
- No-Check Exceptions
- Multicycle Path Exceptions
- Same-Cycle and Next-Cycle Checking Exceptions

Reporting Exceptions

The `report_exceptions` command reports the timing exceptions previously defined by the exception-setting commands.

For each exception, the report shows the objects specified as the “from,” “through,” and “to” points; the type of exception applied for setup and hold checks; and the number of matching occurrences of that exception encountered during path tracing.

To report all exceptions, use the `report_exceptions` command with no options. To restrict the scope of the report, use the same “from,” “through,” and “to” options that were used in the original exception-setting commands.

A general path specification reports a matching general exception or a more specific exception if there is a matching object in the path specification of the original exception-setting command. For example, the `report_exceptions -from selA` command reports all exceptions originally specified with the `-from selA`, `-rise_from selA`, and `-fall_from selA` options.

The following is an example of the timing exception report:

```
nt_shell> report_exceptions
 ...
From     Through   To         Setup          Hold       Matches
-------- --------- ---------- -------------- ---------- -------
selA     *         BUS[8]     FALSE          FALSE            4
selA(r)  *         {and add}  FALSE          FALSE            4
*        {m1 m2}   *          cycle=3        --              28
cyc1     *         *          no_same_phase  no_same_phase    8
cyc0     *         *          same_phase     same_phase      16
test     *         *          no_check       no_check       168
*        {m1}
         {m2(e)}   *          --             exclude_ref      2
```

The From, Through, and To columns show the objects specified in the original exception-setting commands. The notation “(r)” in the From column indicates that the object selA was specified with the `-rise_from` option rather than the `-from` option.

The Setup and Hold columns indicate the type of exceptions applied for setup and hold checking, using the following labels: false path, no_same_phase, same_phase, no_check, multicycle, or exclude_ref.

Multicycle checks are displayed with the formats “cycle=3” and “abs_cycle=3.” In the Setup column, entries are always in the format “cycle=3” and they result from `set_multicycle_path -setup` commands. In the Hold column, entries can be in either format before path tracing. The format “cycle=3” results from `set_multicycle_path -hold` commands and the format “abs_cycle=3” results from `set_multicycle_path -hold_cycle` commands. After path tracing, all entries in the Hold column have the format “abs_cycle=3” because NanoTime converts all settings to a consistent format during path tracing.

The Matches column indicates the number of times that the exception occurred during path tracing. Two dashes (--) indicate either that path tracing has not been performed or that no matches occurred during path tracing. The number of matches reported for an exception-setting command such as the `set_multicycle_path` command can be affected by conflicting higher-priority exceptions, such as false path exceptions.

Restricting Analysis to Specified Paths

In some situations you might want to analyze only specific paths rather than the whole design. Restricting the analysis to a smaller number of paths reduces the scope of the timing reports and can significantly reduce the runtime and memory usage.

Use the `set_find_path` command to specify the paths in the design that you want to analyze. The command specifies a list of allowable startpoints, throughpoints, endpoints, and edge types.

The `-through`, `-rise_through`, and `-fall_through` options cannot be used when clock objects are specified with the `-from` or `-to` options.

The following command selects all paths that start with a rising edge on port rst:

```
nt_shell> set_find_path -rise_from [get_ports rst]
```

The following command selects paths that start from port IN1 and end on port OUT1 or port OUT2:

```
nt_shell> set_find_path -from [get_ports IN1] -to [get_ports {OUT1 OUT2}]
```

Using the `set_find_path` command is the only way to force the analysis of specific paths that are defined by a “through” point. Standard path tracing saves only the single path with the worst delay to an endpoint. Even if you choose to save a large number of the worst paths to every endpoint during path tracing, paths through your “through” point of interest might not be included.

Use the `set_find_path` command after the `link_design` command, but before the `check_design` and `trace_paths` commands. After you specify the paths, the `check_design` command checks only the specified paths and the `trace_paths` command traces only those paths.

You can use the `set_find_path` command multiple times to specify multiple sets of paths to be traced. NanoTime processes the sets of paths sequentially.

You can report the path restrictions defined by the `set_find_path` command with the `report_find_path` command. To cancel the effects of the `set_find_path` command, use the `remove_find_path` command.

The check_design Command

The `check_design` command checks the current design for proper structure and prepares the design for path tracing. In a NanoTime analysis session, it must be run after the `check_topology` command and after all timing constraints have been set, but before the `trace_paths` command.

The `check_design` command divides the design into units called channel-connected blocks (CCBs). Each channel-connected block is a set of transistors whose sources and drains are connected together in a network, in series or in parallel, or both. NanoTime checks the channel-connected blocks and reports any problems it has in interpreting their configuration. You can correct such problems by using commands such as `mark_*` and `erase_*`, possibly in conjunction with the `foreach_match` command.

This is a typical response to the `check_design` command:

```
nt_shell> check_design
Pre-processing structural constraints...
Done pre-processing structural constraints
Creating new timing graph...
Analyzing transistor structures for timing graph...
Building timing graph...
Checking timing graph for consistency...
1
Linking transistor models...
Information: 2237 out of 2237 transistors have transistor models linked
to them.  (TECH-003)
Annotated capacitances           :        73993
Reduced capacitances             :        12261
Annotated resistances            :        85303
Reduced resistances              :        15627
```

The `-message_level` option of the `check_design` command specifies the amount of information reported in the terminal window. These are the allowed settings:

- `silent`: no messages except for warnings and errors
- `normal` (the default): brief messages
- `verbose`: statistics about channel-connected blocks
- `debug`: information about individual channel-connected blocks

The `-message_level verbose` option of the `check_design` command provides statistical information about channel-connected blocks in the design:

```
nt_shell> check_design -message_level verbose
Pre-processing structural constraints...
 ...
Histogram of number CCBs by number of inputs nets
Maximum number of inputs nets for any CCB is 18
Total number of CCBs is 2245

Inputs  CCBs     %  Net in a CCB with this number of inputs nets
------  ---- -----  ---------------------------------------------
     1  1387  61.8  Xlc23.or
     2   193   8.6  Xareg.Xreg0.X2.A
     4   512  22.8  Xareg.Xreg0.X3.A
   ...
```

9

Timing Checks

Timing checks test whether the design meets the timing goals. Many timing checks are derived from the circuit topology. You can also define custom timing checks.

The specification of timing checks is described in the following sections:

- Timing Check Concepts
- Latch Setup and Hold Checks
- Clock-Gating Setup and Hold Checks
- Minimum Pulse Width Checks
- Domino Precharge Timing Checks
- User-Specified Timing Checks
- Modifying Timing Checks
- Reporting and Analyzing Timing Checks

Timing Check Concepts

A timing check or constraint is a timing requirement against which NanoTime tests the actual circuit performance. The tool automatically defines many timing checks based on the recognized topologies in the design. You can also define additional timing checks.

Timing Check Triggers

By definition, a path tracing arc (delay arc) begins at a pin known as a trigger and ends at a pin known as a target. A timing check (constraint arc) begins at a reference pin, sometimes called a “from” pin (because it is the argument of `-from` options in various commands). The timing check is evaluated at a checked pin, also called a “to” pin. The “to” pin of a timing check and the target pin of a delay arc are often the same.

As an example, consider the latch in Figure 9-1. The arrows show two delay arcs through the circuit:

- Delay arc 1 starts at pin Mp0.g and ends at pin Mn1.g; this arc is traversed when a falling transition at D turns Mp0 and Mn1 on.
- Delay arc 2 starts at pin Mn0.g and ends at pin Mp1.g; this arc is traversed when a rising transition at D turns Mn0 and Mp1 on.

The figure also shows two timing checks (constraint arcs) for this latch. By default, setup and hold timing checks for latches begin at a pin controlled by a clock signal and are evaluated at the latch net (net b). The setup timing checks for this example are from pin Mn3.g to Mn1.g and from Mn3.g to Mp1.g.

Figure 9-1 Latch Circuit With Default Timing Checks

A timing check is associated with the delay arc endpoint. This means that in Figure 9-1, if path tracing arrives at pin Mn1.g, the timing check labeled “constraint arc 1” is evaluated. In some designs, more than one timing check might end at pin Mn1.g, and more than one delay arc might end there as well. All of the timing checks that end at Mn1.g are evaluated whenever path tracing reaches that pin—no matter which delay arc is traversed to get there.

To explicitly control when or if a timing check should be evaluated, you can specify the trigger for the timing check. For example, you can specify that timing check 1 in Figure 9-1 should be performed only after a transition at input D by using the `-trigger` option of the `create_timing_check` command. The “from” and “to” nets of the timing check remain the same (clk to b), but the timing check is evaluated only if path tracing traverses the delay arc from transistor Mp0 to transistor Mn1. The command to accomplish this is as follows:

```
nt_shell> create_timing_check -from Mn3.g -to Mn1.g -trigger Mp0.g
```

For further control, you can use the `-rise_trigger` or `-fall_trigger` option to require a rising or falling transition at the trigger device. The trigger options are also available for the `set_data_check` command for creating nonsequential timing checks. You must define new timing checks before executing the `check_design` command to ensure that the tool designates all needed pins as timing points.

Timing Check Targets

For certain topologies, NanoTime provides methods to move the “to” points of timing checks. The affected topologies are latches, precharge structures, and flip-flops. For the latch example, you might want to verify the timing at the latch output net c instead of the latch net b for a more conservative check.

For example, to move a setup check for a latch, use one of these two strategies:

- Use the `-setup_to output` option with the `mark_latch` command when marking the latch.
- Use the `-setup_to output` option with the `set_timing_check_attachment` command after topology checking is complete.

Moving the timing check endpoint causes NanoTime to consider the entire latch structure to be one simulation unit, even though it includes more than one channel-connected block. However, a delay arc can only traverse one channel-connected block. Therefore, if you move a timing check by using a `-setup_to` or `-hold_to` option, the “to” point of the timing check and the target of the delay arc are now different pins, as shown in Figure 9-2. For this example, the trigger, target, “from” point, and “to” point are four different pins.

Because the target and “to” pin are different pins, the timing check is now associated with the delay arc instead of the timing check “to” pin. In other words, the path from the trigger to the target must be traced for the timing check to be evaluated. This is a consequence of moving the timing check evaluation point.

Figure 9-2 Latch Circuit With Modified Timing Check

After the `check_design` command, you can modify some aspects of a timing check by using the `set_timing_check_attributes` command. For example, you can change the check value or specify whether to continue path tracing if there is an error.

Since the timing check must already exist, the `-from`, `-to`, `-target`, and related options of the `set_timing_check_attributes` command are filters that select the timing check to be modified by other command options. The `-target` option is most commonly used to search for timing checks with relocated "to" points in latch, precharge, and flip-flop topologies.

For example, the following `set_timing_check_attributes` command applies to Figure 9-2. The intention is to find a specific timing check and to apply the `-continue_with_error_adjustment` option to that timing check. The `-trigger`, `-target`, `-from`, and `-to` options specify which timing path is to be modified.

```
nt_shell> set_timing_check_attributes -from Mn3.g -to Mn2.g \
          -trigger Mp0.g -target Mn1.g -continue_with_error_adjustment
```

A timing check is normally associated with path tracing arrival at the timing check “to” point. If you want the target and “to” points to be the same, but you want to specify that a timing check is associated with the path tracing traversal of a delay arc instead, use the `-to_not_target` option of the `set_timing_check_attributes` command to change that property of the selected timing check.

For the example in Figure 9-3, the following command specifies that the timing check from net clk to net b occurs only if path tracing traverses the path from net D to net b:

```
nt_shell> set_timing_check_attributes -from Mn3.g -to Mn1.g \
          -trigger Mp0.g -target Mn1.g -to_not_target
```

Figure 9-3 Latch Circuit

Library-Based Timing Checks

NanoTime performs data checking for any cell that has nonsequential timing constraints defined in the library cell, unless the signal at the related pin is defined to be a clock in NanoTime. In that case, the tool ignores library-defined nonsequential timing constraints.

In the Library Compiler tool, you can define nonsequential constraints on a cell by specifying a related pin and assigning the following `timing_type` attributes to the constrained pin:

```
non_seq_setup_rising
non_seq_setup_falling
non_seq_hold_rising
non_seq_hold_falling
```

For more information, see the Library Compiler documentation.

Defining nonsequential constraints in the library cell results in a more accurate analysis than using the `create_timing_check` command because the setup and hold times can be made sensitive to the slew of the constrained pin and the related pin. The `create_timing_check` command is not sensitive to slew.

The `remove_timing_check` command does not remove data checks defined in library cells.

Latch Setup and Hold Checks

NanoTime performs a setup check to verify that a signal launched at a path startpoint by a clock edge arrives at the path endpoint soon enough before the capture clock edge. It performs a hold check to verify that the same signal arrives late enough not to be captured by the previous capture clock edge.

The clock edges that are considered the launch and capture clock edges depend on several conditions, including the relative timing of the launch and capture clocks, the types of sequential elements (transparent latches or flip-flops), the same-cycle or next-cycle settings for the sequential elements or the path, and the value of the `timing_intersection_transparency` variable.

The default rules for determining the clock edges for setup and hold checks typically reflect the intended operation of the circuit. However, for paths that do not operate according to these rules, you can specify a multicycle path timing exception as described in Multicycle Path Exceptions, or same-cycle or next-cycle checking as described in Same-Cycle and Next-Cycle Checking Exceptions.

By default, NanoTime assumes that latches are transparent. Latches that are not designed to operate transparently should not allow path tracing to proceed through them. To specify this behavior for selected topologies or clocks in the design, use the `set_non_transparent` command. For example,

```
nt_shell> set_non_transparent [get_topology \
          -of_object n1 -structure_type latch]
```

This command sets nontransparent checking on the latch that has its latch node at net n1. When nontransparent checking is set on a latch or a precharge net, path tracing cannot continue past that object. When it is set on a clock, NanoTime uses nontransparent checking behavior on all latches clocked by that clock.

To cancel the effects of the `set_non_transparent` command, use the `remove_non_transparent` command.

Latch-based designs can use multiple clocks with different phases to control successive registers in a data path. For example, consider the two-phase, latch-based path shown in Figure 9-4. All three latches are level-sensitive, with the gate active when the G input is high.

L1 and L3 are controlled by PH1, and L2 is controlled by PH2. A rising edge launches data from the latch output and a falling edge captures data at the latch input.

Figure 9-4 Latch-Based Paths

Figure 9-5 shows how NanoTime performs setup checks between these latches. For the path from L1 to L2, the rising edge of PH1 launches the data. The data must arrive at L2 before the next closing edge of PH2, at time = 4.

Figure 9-5 Setup Checks Using Two-Phase Clocks

If the data launched by L1 arrives at L2 before the opening edge of clock PH2, the data enters the latch on the opening edge at L2, at time = 2, and the data is captured on the closing edge, at time = 4. This situation is marked A in the figure. In this case, NanoTime assumes that L2 is the intended endpoint of the data path and stops path tracing at L2.

Latch L2 is transparent when clock PH2 is high. If the data launched by L1 arrives at L2 during this period of transparency, NanoTime performs a setup check for the arrival of data at L2 before the closing edge at time = 4. This situation is marked B in Figure 9-5.

Because L2 is transparent in situation B" NanoTime continues path tracing from L2 and checks for the arrival of data at the next latch downstream, at L3. It performs the same kind of setup check there. If the data arrives during the period of transparency at L3, NanoTime continues path tracing forward, and so on. Path tracing stops at a nontransparent sequential device or an output port.

If the data launched by L1 arrives at L2 after the closing edge of clock PH2, the result is a timing violation, as shown by example C in Figure 9-5. To test for possible violations downstream from this point, NanoTime adjusts the data arrival time back to the zero-slack value and continues tracing the path forward, as indicated by the letter D in the figure. The `report_paths` command reports this condition as a "latch error recovery" adjustment.

To cause NanoTime to stop path tracing when a setup error occurs, set the `trace_latch_error_recovery` variable to `false`.

To perform hold checking, NanoTime considers the capture edges relative to the setup check. It verifies that data launched at the startpoint arrives late enough not to be captured by the previous edge, which is the capture edge just before the one at which the setup check is performed. This behavior is shown in Figure 9-6.

Figure 9-6 Hold Checks Using Two-Phase Clocks

The default timing margin at the path endpoint is zero. You can specify a nonzero setup or hold timing margin by setting the `timing_latch_*_**_margin` variables (where * represents `setup` or `hold` and ** represents `rise` or `fall`).

Similarly, for timing paths that end at RAM cells, you can specify a nonzero setup or hold timing margin by setting the `timing_ram_*_**_margin` variables.

By default, the margin is set to zero. Specify a positive margin for a more conservative and restrictive check, or a negative margin for a more permissive check. The variable must be set before you use the `check_design` command. After the `check_design` command, you can adjust timing margins at individual locations in the design by using the `set_timing_check_attributes` command.

Setup and hold checking at flip-flops is similar to the checking done at transparent latches. However, there is no transparency through a flip-flop, and launch and capture both occur on the same clock edge, either rising or falling, depending on the type of flip-flop. Use the `timing_flip_flop_*_**_margin` variables to set the timing margins for setup and hold checks at flip-flops.

NanoTime determines a setup relationship by finding the first capture edge at the path endpoint that comes after the launch edge at the path startpoint. Then it determines the hold relationship by checking the arrival of the data for the setup relationship relative to the previous capture edge, and also the arrival of the next data launch relative to the setup capture edge.

For example, in Figure 9-7, the setup relationships are different because of the different clocks used for setup and capture. The more restrictive setup relationship is Setup 1, so that relationship establishes the setup requirement for the path. For the hold requirement, the most restrictive check is Hold 2b, so that relationship establishes the hold requirement for the path.

Figure 9-7 Setup Constraints For Flip-Flops With Different Clocks

```
nt_shell> create_clock -period 10 -waveform {0 5} CLK1
nt_shell> create_clock -period 25 -waveform {5 12.5} CLK2
```

Clock-Gating Setup and Hold Checks

A gated clock signal occurs when the clock network contains logic other than inverters or buffers. For example, if a clock signal acts as an input to a logical AND function and a control signal acts as the other input, the output is a gated clock.

NanoTime automatically checks for setup and hold violations on gating inputs to ensure that the clock signal is not interrupted or clipped by the gate. This checking is done only for combinational gates where one signal is a clock that can be propagated through the gate, and the gating signal is not a clock.

The clock-gating setup check ensures that the controlling data signal enables the gate before the clock becomes active. The arrival time of the leading edge of the clock signal is checked against both edges of any data signal that feeds the data pins to prevent a glitch at the leading edge of the clock pulse or a clipped clock pulse.

Figure 9-8 shows examples of clock-gating setup violations.

Figure 9-8 Clock-Gating Setup Violations

The clock-gating hold check ensures that the controlling data signal remains stable while the clock is active. The arrival time of the trailing edge of the clock pin is checked against both

edges of any data signal that feeds the data pins. A clock-gating hold violation causes either a glitch at the trailing edge of the clock pulse or a clipped clock pulse.

Figure 9-9 shows examples of clock-gating hold violations.

Figure 9-9 Clock-Gating Hold Violations

NanoTime checks setup and hold times for gated clocks that it recognizes. The default timing margin for a setup or hold check is zero, unless the library cell for the gate has gating setup or hold timing arcs. You can specify a nonzero setup or hold timing margin for specified paths in the design by using the `set_gated_clock_timing_check_attributes` command, or you can specify them globally for all gated clock timing checks with the `timing_clock_gate_*_**_margin` variables (where * represents `setup` or `hold` and ** represents `rise` or `fall`).

To create a gated-clock timing check where none exists by default, use the `create_gated_clock_timing_check` command. To remove a gated clock timing check, use the `remove_gated_clock_check` command. For more information, see the man pages for the commands.

Minimum Pulse Width Checks

You can optionally direct NanoTime to check clock or data signals for minimum pulse width violations. For example, a clock pulse width violation can occur if a clock signal is negated too soon after it is asserted. Clock pulse width violations might prevent sequential devices from capturing data properly.

To specify a minimum pulse width check, use the `set_min_pulse_width` command. The command specifies the minimum allowed pulse width and the clocks, cells, or pins where the constraint is to be applied. For example, the following command sets a minimum pulse width requirement of 2.0 for all signals in the clock tree of clock CK1:

```
nt_shell> set_min_pulse_width 2.0 [get_clocks CK1]
```

In this example, a pulse width of less than 2.0 time units anywhere in the CK1 clock tree triggers a timing violation during path tracing.

You can use the `-low` or `-high` option to restrict the check to high or low pulses only. Otherwise, the constraint applies to both high and low pulses.

Specifying a cell applies the minimum pulse width check to all pins of the cell. If the cell also has a minimum pulse width constraint defined in the logic library, NanoTime uses the more restrictive (smaller) value.

Note:

Setting pulse width checks globally on data nets is not recommended, because the resulting large number of checks is likely to generate a large number of false violations. Use the `set_data_check` command to define timing checks on a specific set of data nets instead.

To report the existing pulse width checks, use the `report_constraint -min_pulse_width` command or the `report_min_pulse_width` command. You can use these commands only after the `trace_paths` command has been run.

To remove minimum pulse width checks, use the `remove_min_pulse_width` command.

Domino Precharge Timing Checks

A domino precharge circuit has two phases of operation, precharge and evaluate. These phases are defined by the state of the clock signal that controls the circuit. The data signal must arrive at the correct time with respect to both the rising and falling edges of the clock.

The types of setup and hold checks performed by NanoTime depend on the construction of the domino precharge circuit. A domino circuit that has a controlling clock on the evaluate stack is called a type D1 (footed domino) stage. A domino circuit that does not have a controlling clock on the evaluate stack is called a type D2 (footless domino) stage.

The types of timing performed also depend on whether the inputs of the precharge circuit are driven by precharge logic. If you specify that an input net is driven by precharge logic, NanoTime can avoid the timing checks for combinational logic. To specify the nets that are driven only by precharge logic, use the `set_precharge_input_net` command. To cancel the effects of this command, use the `remove_precharge_input_net` command.

Table 9-1 lists the domino checks performed by NanoTime and the codes used to refer to the checks in timing reports.

Table 9-1 Domino Precharge Timing Check Notation in Path Reports

Code	Description
S1	Setup check at a falling precharge net, footed or footless, full or half keeper
S2a	Setup check at a falling precharge input, footless, driven by a domino device, full or half keeper
S2b	Setup check at a falling precharge input, footed or footless, driven by a non-domino device, full or half keeper
H3a1	Hold check at a falling precharge net, footed, driven by a domino device, half-keeper, with min_latch_transparent enabled
H3a2	Hold check at a falling precharge net, footless, driven by another domino device
H3b1	Hold check at a falling precharge net, footed, driven by a non-domino device, half-keeper, with min_latch_transparent enabled
H3b2	Hold check at a falling precharge net, footless, driven by a non-domino device
H3x1	Hold check at a falling precharge net, footed, full keeper; or half-keeper with min_latch_transparent disabled; a nontransparent check
H4	Hold check at a falling precharge input, footed or footless, full or half keeper

Type D1 Domino Stage Setup and Hold Checks

A type D1 domino stage has a controlling clock on the evaluate stack. This construction prevents charge from leaking away from the precharge node during the precharge phase for all logic states of the data input signals. See Figure 9-10 for an example.

Figure 9-10 Type D1 Domino Stage

For a logic-high input signal, NanoTime verifies that the rising edge of the input signal arrives soon enough to discharge the precharge node before the end of the evaluate phase. This is called the S1 setup check. NanoTime also verifies that the falling edge of the input signal occurs no sooner than the end of the evaluate phase, so that the precharge node is not left floating. This is called the H4 hold check. See Figure 9-11.

Figure 9-11 Type D1 Checking For a Logic-High Input

For a logic-low input signal, NanoTime verifies that the falling edge of the input signal arrives soon enough before the start of the evaluate phase to prevent unwanted discharge of the precharge node. This is called the S2b setup check. NanoTime also verifies that the rising edge of the input signal occurs no sooner than the end of the evaluate phase, to prevent unwanted discharge of the precharge node. This is called the H3 hold check. See Figure 9-12.

Figure 9-12 Type D1 Checking For a Logic-Low Input

Type D2 Domino Stage Setup and Hold Checks

A type D2 domino stage does not have a controlling clock on the evaluate stack, so during the precharge phase, at least one of the data input signals must be low to prevent charge from leaking away from the precharge node. See Figure 9-13 for an example of a type D2 domino circuit.

Figure 9-13 Type D2 Domino Example

For a logic-high input signal, NanoTime verifies that the rising edge arrives soon enough to discharge the precharge node before the end of the evaluate phase. This is called the S1 setup check. NanoTime also verifies that the falling edge of the input signal occurs no sooner than the end of the evaluate phase, so that the precharge node is not left floating. This is called the H4 hold check, as shown in Figure 9-14 for a logic-high input signal and in Figure 9-15 on a D2 for logic-high input signal.

Figure 9-14 Type D2 Checking For a Logic-High Input

Figure 9-15 Type D2 Checking For a Logic-Low Input

The `timing_precharge_contention_check` variable specifies whether to perform contention checking at D2 type domino stages during the precharge phase. If the variable is set to `true` (the default), NanoTime performs type S2a setup checking and type H3a hold checking, which check input signals arrivals with respect to the precharge phase. If the variable is set to `false`, NanoTime performs type S2b setup checking and type H3b hold checking, which check input signal arrivals with respect to the evaluate phase.

When the variable is set to `true`, for a falling input signal, NanoTime verifies that the falling edge occurs before the start of the precharge phase. This check is the S2a setup check. NanoTime also verifies that the signal remains low at least until the end of the precharge phase, to prevent discharge of the precharge node. This check is the H3a hold check.

When the variable is set to `false`, NanoTime verifies that the falling edge of the input signal arrives soon enough before the start of the evaluate phase to prevent unwanted discharge during the evaluate phase. This check is the S2b setup check. NanoTime also verifies that the signal remains low at least until the end of the evaluate phase, to prevent incorrect discharge during that phase. This check is the H3b hold check.

In Figure 9-15, the input from another domino stage must remain high until the end of the previous evaluate phase, so that the precharge node is not left floating during that phase.

This is the H4 hold check. In order for this edge to meet both the S2a setup check and the H4 hold check, some skew must exist between the precharge and evaluate clocks.

If a domino circuit has a full-cycle keeper, the H4 hold check is performed in a different manner. Without a full-cycle keeper, the high-to-low data input transition must occur after the end of the previous evaluate phase. With a full-cycle keeper to hold the precharge node low, the high-to-low data input transition can occur any time after the beginning of the previous evaluate phase. See Figure 9-16.

Figure 9-16 Full-Cycle Keeper Circuits and H4 Hold Check

A domino stage with a full-cycle keeper falls into one of these categories:

- Domino retain: A full-keeper domino gate with a clock that has the same phase as the clock that triggers the input data, and that has not been marked as a flip-flop by the `mark_flip_flop` command. This case is similar to the basic domino stage, except that it has a full keeper rather than a half PMOS keeper.
- Domino latch: A full-keeper domino gate with a clock having a different phase from the clock that triggers the input data. The stage can act as a transparent latch. NanoTime uses the new-phase clock for timing checks.
- Domino flop: A full-keeper domino gate with a clock having the same phase as the clock that triggers the input data, and that has been marked with the `mark_flip_flop` command. These gates are treated as flip-flops. The timing margin calculations account for the cycle change. NanoTime uses the next clock cycle for the timing checks.

Domino Precharge Checking Options

By default, NanoTime performs hold checking in precharge circuits by finding the arrival time at the evaluate net. For a more conservative check, you can check for the arrival time at the precharge input. To do this, use the `set_timing_check_attachment` command with the `-setup_to input` or `-hold_to input` options to move checks from the evaluate net to the `input`. Use the `get_topology` command as a filter to specify which precharge topologies with use this adjusted check.

The default timing margin at each path endpoint is zero. You can specify a nonzero setup or hold timing margin at evaluate nets by setting the `timing_precharge_*_**_margin` variables (where * represents `setup` or `hold` and ** represents `rise` or `fall`).

Similarly, you can specify a nonzero setup or hold timing margin at the inputs of precharge circuits by setting the `timing_precharge_in_*_**_margin` variables.

By default, the margins are set to zero. Specify a positive margin for a more conservative and restrictive check, or a negative margin for a more permissive check. These variables must be set before you use the `check_design` command.

Other `timing_precharge_*` variables control different aspects of timing checks in precharge circuits. For more information, see the man pages.

User-Specified Timing Checks

NanoTime performs setup and hold checking at the sequential structures that it recognizes, such as latches, registers, and precharge gates. In addition to these timing checks, you can specify other timing checks with the `create_timing_check` command for sequential checks and the `set_data_check` command for nonsequential checks. These commands appear to be similar, but they function differently and they serve different purposes. Table 9-2 compares the essential features of the two types of checks.

Table 9-2 Comparison of User-Specified Timing Checks

Sequential check (create_timing_check)	Nonsequential check (set_data_check)
is defined from a clock object to a data object	is defined between two clock objects or two data objects
is evaluated during path tracing	is evaluated after path tracing
might affect subsequent path tracing	cannot affect path tracing
is associated with a pin	is associated with a path segment

In both cases, NanoTime performs these checks in addition to any other checks already being performed between the “from” and “to” points, such as those automatically created by NanoTime based on the topology. The new timing checks do not replace existing checks.

User-Specified Sequential Checks

Specify additional sequential timing checks from a clock object to a data object with the `create_timing_check` command. NanoTime associates these timing checks with a pin (the “to” pin). In other words, if path tracing arrives at that pin through any path (not necessarily from the “from” pin), NanoTime evaluates the timing check immediately and might modify the actions of further path tracing based on the results.

If a setup violation occurs, NanoTime adjusts the data arrival time backward to the passing arrival time and continues path tracing forward. The `-continue` option forces NanoTime to continue propagation after it evaluates the timing check; you should use the `-continue` option for almost all timing checks.

You must define timing checks before the `check_design` command. However, you can modify them later in the flow with the `set_timing_check_attributes` command. To remove a sequential timing check, use the `remove_timing_check` command.

User-Specified Nonsequential Checks

Specify a setup or hold check between any two data ports or pins or between any two clock ports or pins by using the `set_data_check` command. NanoTime associates these timing checks with a path segment. In other words, if path tracing traverses the specified segment, NanoTime evaluates the specified timing check after all path tracing is complete.

The `set_data_check` command lets NanoTime know before path tracing that certain delays are of special interest, to ensure that the tool calculates them and saves them in the path tracing database.

Nonsequential timing checks (also called data checks) can be useful for analyzing the following types of timing constraints. You should use data checks only in situations such as these. You should not use data checks to replace ordinary sequential checking.

- Constraints on handshaking interface logic
- Constraints on asynchronous or self-timed circuit interfaces
- Constraints on signals with unusual clock waveforms that cannot be easily specified with the `create_clock` command
- Constraints on skew between bus lines
- Recovery and removal constraints between asynchronous preset and clear input pins

You must specify data checks before the `check_design` command. You can remove a data check with the `remove_data_check` command. Data checks are not evaluated at the `extract_model` command unless you include the `-non_sequential_arcs` option.

Figure 9-17 shows a simple example of a cell that has a nonsequential constraint. The cell has two data inputs, D1 and D2. The rising edge of D2 is the active edge that might be used to latch data at D1. Pin D1 (the “to” pin) is sometimes called the checked pin and pin D2 (the “from” pin) is sometimes called the reference pin.

Figure 9-17 Simple Data Check Example

In this example, the signal at D1 must be stable for a setup time before the active edge. It must also be stable for a hold time after the active edge. To define this check, use commands similar to the following:

```
nt_shell> set_data_check -rise_from D2 -to D1 -setup 3.5
nt_shell> set_data_check -rise_from D2 -to D1 -hold 6.0
```

The “from” pin is the related pin and the “to” pin is the constrained pin. If the data checks apply to both rising and falling edges on the related pin, use the `-from` option instead of the `-rise_from` or `-fall_from` option, as shown in the following example:

```
nt_shell> set_data_check -from D2 -to D1 -setup 3.5
nt_shell> set_data_check -from D2 -to D1 -hold 6.0
```

Modifying Timing Checks

Use the following general procedure to modify timing checks.

1. Create timing checks with any of the following methods:
   - Define topologies, allowing NanoTime to create automatic timing checks.
   - Use the `create_timing_check` command to create new sequential timing checks.
   - Use the `set_data_check` command to create new nonsequential timing checks.
2. If needed, move timing check endpoints for latch, precharge, or flip-flop topologies with the following methods:
   - When marking a latch or precharge structure, use the `-setup_to` or `-hold_to` options with the `mark_latch` or `mark_precharge` commands.
   - Use the `-setup_to` or `-hold_to` options with the `set_timing_check_attachment` command to move the “to” point of latch, precharge, or flip-flop structures.
3. Use the `set_timing_check_attributes` command to search for specific timing checks and apply changes to those timing checks.

Aspects of a timing check that you can modify include the following:

- Path tracing options at the endpoint of the timing check: stop, continue, or continue with error adjustments
- The setup or hold time requirements: specific values, margin adjustments, or complex requirements based on signal transition times
- How to apply path-based slack adjustment
- Whether to use same-cycle checking

Reporting and Analyzing Timing Checks

After path tracing, you can use the `report_analysis_coverage` and `report_fanin` commands to determine whether the timing checks have been evaluated and the timing constraints have been met.

The analysis coverage report summarizes how many timing checks are met, violated, untested, or removed. To collect the information necessary to generate the report, you must set the `timing_analysis_coverage` variable to `true` before performing path tracing. If you want to report details about untested and removed timing checks, you must set the `timing_analysis_coverage_fanin` variable to `true` before performing path tracing.

Note:
Collecting information for the detailed level of timing check reporting might cause increased runtime and memory usage.

To enable and report timing check coverage, perform the following steps:

1. Set the `timing_analysis_coverage` variable to `true` (the default is `false`).
2. (Optional) If you want to report details about the reasons for untested timing checks, set the `timing_analysis_coverage_fanin` variable to `true` (the default is `false`).
3. Execute the `trace_paths` or `extract_model` command.
4. Execute the `report_analysis_coverage` command.
5. (Optional) Use the `report_fanin` command to investigate untested timing checks by reporting sources (pins or instances) of possible untested reasons in the fanin cone. This command requires the `timing_analysis_coverage_fanin` variable to be `true`.

The report_analysis_coverage Command

An example of the default timing check coverage report is as follows:

```
nt_shell> report_analysis_coverage
...
Type of Check    Total        Met          Violated     Untested
---------------------------------------------------------------------
setup            10           6 (60%)      0 ( 0%)      4 (40%)
hold             10           6 (60%)      0 ( 0%)      4 (40%)
---------------------------------------------------------------------
Total            20           12 (60%)     0 ( 0%)      8 (40%)
```

To report only a subset of timing checks, use the `-status_details` option with an argument of `untested`, `violated`, `removed`, or `met`. To specify the type of check in the report, use the `-check_type` option with an argument of `hold` or `setup`. To sort the report, use the `-sort_by` option with an argument of `slack`, `check_type`, or `name`.

For example, the following command reports the coverage analysis of untested timing checks:

```
nt_shell> report_analysis_coverage -status_details untested
```

The resulting report is similar to the following:

```
****************************************
Report : timing check coverage report
Design : ALU
...
****************************************

Untested reasons:
    NP    - no_paths
    NEC   - no_endpoint_clock
    CPE   - capture_paths_excluded
    CD    - constant_disabled
    UNK   - unknown
    FP    - false_path
    NTC   - no_timing_check_path
    DCD   - data_constant_disabled
    CCD   - clock_constant_disabled
    DST   - dont_search_thru
    PC    - previous_check


Type of Check    Total     Met          Violated     Untested
---------------- --------- ------------ ------------ ------------
setup            3996      1942 (49%)   774 (19%)    1280 (32%)
hold             3996      2411 (60%)   305 ( 8%)    1280 (32%)
-----------------------------------------------------------------
Total            7992      4353 (54%)   1079 (14%)   2560 (32%)


type   label                  slack reason    Constrained Pin   dir Related Pin    dir
------ ------------------ --------- --------- ----------------- --- -------------- ---
hold   gated clock hold    untested NP        Xreg0.X1.Mn1.G     r  Xreg0.X1.Mp0.G  f
setup  gated clock setup   untested NP        Xreg0.X1.Mn1.G     r  Xreg0.X1.Mn0.G  r
hold   gated clock hold    untested NP        Xreg0.X1.Mp1.G     f  Xreg0.X1.Mp0.G  f
setup  gated clock setup   untested NP        Xreg0.X1.Mp1.G     f  Xreg0.X1.Mn0.G  r
hold   latch hold (trans)  untested FP        Xreg0.X5.Mn0.G     r  Xreg0.Mp0.G     r
...
```

Table 9-3 describes the reason codes for untested timing checks. The first four reason codes can appear in all analysis coverage reports. The remaining reason codes are reported only if the `timing_analysis_coverage_fanin` variable is set to `true`. If the

variable is set to `false`, NanoTime reports those conditions as either no_paths or constant_disabled.

Table 9-3 Reason Codes For Untested Timing Checks

Reason code in default reports	Reason code in detailed reports	Description
unknown	unknown	The reason could not be determined.
no_paths	no_paths	No valid data path reached the constrained pin.
no_endpoint_clock	no_endpoint_clock	No valid clock path reached the related pin.
capture_paths_excluded	capture_paths_excluded	All clock paths that reach the related (reference) pin were excluded when the constrained (launch) path matched an exception specified with the `set_exclude_reference_path` command.
constant_disabled	constant_disabled	The net at the constrained pin is in a fixed logic state due to a `set_case_analysis` command or other means of setting a fixed signal value.
constant_disabled	data_constant_disabled	Fixed logic propagates to the constrained pin.
constant_disabled	clock_constant_disabled	Fixed logic propagates to the reference pin of a constraint.
no_paths	false_path	A `set_false_path` command was successfully executed somewhere in the fanin cone of the constrained pin.
no_paths	no_timing_check_path	A `set_no_timing_check_path` command was successfully executed somewhere in the fanin cone of the constrained pin.
no_paths	dont_search_thru	A `mark_instance -dont_search_thru_gate` or `mark_instance -dont_search_thru_channel` command was successfully executed somewhere in the fanin cone of the constrained pin.
no_paths	previous_check	A timing check was created or modified somewhere in the fanin cone of the constrained pin without specifying the `-continue` option, causing path tracing to stop.

A report of the removed timing checks generated with the `report_analysis_coverage -status_details removed` command uses the reason codes described in Table 9-4.

Table 9-4 Reason Codes For Removed Timing Checks

Reason code	Description
removed_check	A default timing check was removed with the `remove_timing_check` or `remove_gated_clock_check` command.
removed_user_check	A user-created timing check was removed with the `remove_timing_check` or `remove_gated_clock_check` command.

The report_fanin Command

To investigate untested timing checks, use the `report_fanin` command. To enable this capability, the `timing_analysis_coverage_fanin` variable must be set to `true`.

You must supply a target pin as the command argument and one or more of the following options: `-false_path`, `-no_timing_check_path`, `-constant_origin`, `-dont_search_thru`, `-previous_check`, and `-all`. The timing checks corresponding to each reason are presented in separate reports.

For example, the following command generates two fanin reports:

```
nt_shell> report_fanin -false_path -no_timing_check_path XI4.XI2.*.g

*****************************************************
Report:  fanin coverage report for option false_path
Design:
...
*****************************************************
target-pin                pins in transitive fanin cone
------------------------- --------------------------------
XI4.XI2.MN1.g             XI1.XI2.MN1.g
XI4.XI2.MP1.g             XI1.XI2.MP2.g

*****************************************************
Report:  fanin coverage report for option no_timing_check_path
Design:
...
*****************************************************
target-pin                pins in transitive fanin cone
------------------------- --------------------------------
XI4.XI2.MN1.g             XI5.MP1.g
XI4.XI2.MP1.g             XI5.MN1.g
```

10

Timing Analysis

Path tracing is the heart of timing analysis. During path tracing, NanoTime exhaustively calculates delays through the design. The tool performs a variety of timing checks, such as latch setup and hold checks, and creates a database that provides input for many types of reports.

This chapter contains the following sections:

- Path Tracing Concepts
- Effects of Timing Models on Delay Analysis
- Invoking and Controlling Path Tracing
- Path Reporting
- Multi-Input Switching
- Controlling Accuracy
- Controlling Runtime

Path Tracing Concepts

Static timing analysis validates the timing performance of a design by checking all possible paths for timing violations. To check for violations, NanoTime breaks the design into a set of timing paths, calculates the signal propagation delay along each path, and checks for violations of timing constraints inside the design and at the inputs and outputs.

Path tracing begins when you execute the `trace_paths` command. Before you can perform path tracing, you must prepare and check the design data by successfully executing the `link_design`, `check_topology`, and `check_design` commands.

This section contains the following topics:

- Timing Paths
- Simulation Units, Timing Arcs, and Timing Points
- Timing Delay, Transitions, and Adjustments
- Timing Checks

Timing Paths

A timing path is an analysis unit defined both by a set of connected devices and the signals on those devices. A physical path with a rising transition and the same path with a falling transition are considered to be different timing paths.

The startpoint of a timing path is a place in the design where data is launched by a clock edge. The data proceeds through combinational logic and transparent latches in the path and is captured at the endpoint by another clock edge. Properties of path startpoints and endpoints are as follows:

- A startpoint can be a clock, a primary input port, a sequential cell, a clock input pin of a sequential cell, a data pin of a level-sensitive latch, or a pin that has an input delay specified. If you specify a clock, all registers and primary inputs related to that clock are used as path startpoints. If you specify a cell, the command applies to all input pins and bidirectional pins of that cell.
- An endpoint can be a clock, a primary output port, a sequential cell, a data input pin of a sequential cell, or a pin that has an output delay specified. If you specify a clock, all registers and primary outputs related to that clock are used as path endpoints. If you specify a cell, the command applies to one path endpoint on that cell.

Timing paths fall into four categories of data paths and two categories of clock paths. Figure 10-1 shows the four types of data paths:

Figure 10-1 Timing Paths

In this figure, “Logic” refers to a combinational logic network. Each path starts at a data launch point, passes through some combinational logic (and possibly transparent latches), and ends at a data capture point:

- Path 1 starts at an input port and ends at the data pin of a sequential element.
- Path 2 starts at the clock pin of a sequential element and ends at the data pin of a sequential element.
- Path 3 starts at the clock pin of a sequential element and ends at an output port.
- Path 4 starts at an input port and ends at an output port.

Combinational logic might contain multiple paths, as illustrated in Figure 10-2. NanoTime uses the longest path to calculate the maximum delay and the shortest path to calculate the minimum delay.

Figure 10-2 Multiple Paths Through Combinational Logic

In addition to data paths, NanoTime also analyzes clock and clock-gating paths, as shown in Figure 10-3:

- A clock path is a path from a clock input port or cell pin, through one or more buffers or inverters, to the clock pin of a sequential element.
- A clock-gating path is a path from an input port to a clock-gating element.

Figure 10-3 Path Types

Simulation Units, Timing Arcs, and Timing Points

A timing path is made up of one or more timing arcs separated by timing points.

Timing arcs (also known as delay arcs) are the atomic analysis units of timing paths. A timing arc traverses exactly one channel-connected block. The arc begins at the gate of a transistor where the signal transition originates (the trigger) and ends at the gate of the transistor that turns on when the transition is complete. These transistor gates are timing points (also known as timing vertexes). Input and output ports can also be timing points.

For example, Figure 10-4 shows a circuit with a channel-connected block, consisting of an inverter and a pass gate, that subsequently drives another inverter. When signal IN switches low to high, transistors mn1 and subsequently mp3 turn on. The associated timing arc (arc 1 in this example) begins at the gate of mn1 and ends at the gate of mp3. Conversely, for a falling transition at IN, the timing arc (arc 2) begins at the gate of mp1 and ends at the gate of mn3.

The gates of transistors mp1, mn1, mp3, and mn3 automatically become timing points because they are the arc startpoints and endpoints.

Figure 10-4 Timing Arcs With Respect to Channel-Connected Blocks

Transistors mp2 and mn2 form a pass gate, and signals A and B are side inputs (inputs whose states might affect the timing). Under most circumstances, NanoTime does not automatically define timing arcs that begin or end at side inputs. However, if you manually create a timing check between A and OUT, the gate of transistor mp2 becomes a timing point and timing arcs can begin or end at that point.

The trigger of a timing arc is the location (port or pin) of the active transition at the startpoint of the arc. The trigger of arc 1 is the gate pin of transistor mn1; the trigger of arc 2 is the gate pin of transistor mp1.

A NanoTime simulation unit is frequently the same as a channel-connected block. However, it can be either smaller than or larger than a channel-connected block, as follows:

- A simulation unit includes only the transistors that are turned on. For example, only one of several parallel transistors in a NAND or NOR gate might be conducting. That leg of the circuit is included in the simulation unit and the nonconducting devices are not.
- A simulation region for dynamic simulation can include more than one channel-connected block.

See Also

- Chapter 14, “Dynamic Simulation”

Timing Delay, Transitions, and Adjustments

Timing delay has two components: cell delay (also known as gate delay) and wire delay.

- Cell delay is the delay between the gate pin of the switching (driver) transistor and its source or drain pin.
- Wire delay is the delay between the driver source or drain pin and the gate pin of the transistor at the endpoint of the arc. Wire delays are present only if you read in a parasitics file.

  If present, wire delay is always used for analysis. However, to save the wire delay information for reporting, you must set the `timing_save_wire_delay` variable to `true` before path tracing.

The cell delay of a timing path is the sum of the individual cell delays of the timing arcs that make up the path. The wire delay of a path is the sum of wire delays for the constituent arcs.

Several NanoTime features calculate an adjustment to the basic delay of a path. The values of some adjustments are available in reports and attributes. For example, crosstalk delta is the delay adjustment resulting from signal integrity analysis. Adjustments might be positive or negative.

Transition time is the time required for the signal at the endpoint to switch between 10 percent of the rail voltage and 90 percent of the rail voltage. Full transition time is an approximation of the time required to switch between the full rail voltages; it is calculated by extrapolating a straight line drawn through the 10 percent and 90 percent points. .

Timing Checks

Timing checks (also known as constraints) test whether signals arrive at specific locations fast enough to ensure correct operation of the circuit. NanoTime automatically performs several types of timing checks during path tracing. You can also define additional timing checks.

A setup check (constraint) specifies how much time is necessary for data to be available at the input of a sequential device before the clock edge that captures the data in the device. This constraint enforces a maximum delay on the data path relative to the clock path.

A hold check (constraint) specifies how much time is necessary for data to be stable at the input of a sequential device after the clock edge that captures the data in the device. This constraint enforces a minimum delay on the data path relative to the clock path.

See Also

- Chapter 9, “Timing Checks”

Effects of Timing Models on Delay Analysis

A NanoTime timing path might include a timing model (library cell). The path through the model constitutes one timing arc of a complete path. This section describes how the tool calculates the delay through a model.

To use timing models in your analysis, you must read in and link the models at the time of design setup. NanoTime can create and use two types of timing models: nonlinear delay models and composite current source models.

See Also

- Linking Timing Models

Nonlinear Delay Timing Models

A nonlinear delay model (NLDM) uses delay tables to represent the delay through the model as a function of input transition time and output load capacitance, as illustrated in Figure 10-5. When you use NLDM timing models in a design, NanoTime calculates the arc delays according to the context where the models are used. The model is accurate for the range of input transition times and output loads defined in the table.

Figure 10-5 Delay Table Example

For transition times or output loads at intermediate values within the table, NanoTime uses linear interpolation between the data points to estimate the delay. For transition times and output loads outside of the ranges defined in the table, NanoTime 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.

Composite Current Source Timing Models

Composite current source (CCS) timing models are more accurate than nonlinear delay models for advanced process technologies. A CCS driver model uses a time- and voltage-dependent current source, as shown in Figure 10-6. The advantage of this driver model is its ability to handle high impedance nets and other nonmonotonic behavior.

The CCS driver lookup table is indexed by the input transition time and the output load capacitance, similar to a NLDM model. However, the values in the CCS model lookup table represent the driver output current, which is used for a more accurate timing analysis than the simple delay table.

In addition, the CCS model can include the time dimension, allowing a different current-versus-time model to be used for each combination of input transition time and output load capacitance.

Figure 10-6 CCS Timing Driver and Receiver Models

The CCS timing receiver model uses two different capacitor values rather than a single lumped capacitance. The first capacitance is used as the load up to the input delay

threshold. When the input waveform reaches this threshold, the load is dynamically adjusted to the second capacitance value.

A timing arc through a logic library is modeled as a driving cell, an RC network at the output of the cell, and a capacitive load, as shown in Figure 10-6. NanoTime computes the response at the driver output and at the load pins, given the input slew or waveform at the driver input.

- The driver model is intended to reproduce the response of the driving cell's underlying transistor circuitry when connected to an arbitrary RC network, given a specific input slew.
- The reduced-order network model is a simplified representation of the full annotated network that has nearly the same response characteristics as the original network.
- The receiver model represents the complex input capacitance characteristics of a stage input pin, including the effects of the transition direction, the slew at the pin, the receiver output load, and the voltage and temperature conditions.

Note:
A direct comparison to delays from the PrimeTime or SPICE tools cannot be made for nets with transistor drivers and CCS receiver models (either pin-based or arc-based) because the PrimeTime tool does not support transistor drivers and SPICE does not support CCS receiver models.

Invoking and Controlling Path Tracing

The `trace_paths` command initiates path tracing. The goal of path tracing is to find the longest paths, shortest paths, and worst-case paths (those with the worst slack) and to save information about those paths. Options of the `trace_paths` command affect the amount of information saved to the database and therefore the information that you can later report.

If you make changes to the design or to the analysis conditions, the worst-case path might change. For example, annotating parasitics to a design might change the calculated delays substantially. If you want to monitor specific paths before and after the changes, you might need to increase the number of paths saved in the database.

Path tracing might take a long time, depending on the size and complexity of the design and the path tracing options. To get periodic progress reports during path tracing, set the `trace_show_stack_period` variable to a nonzero integer value. The variable specifies the number of minutes of elapsed time between periodic reports.

Increasing the Number of Saved Paths

As NanoTime traverses paths in the design, the tool prunes paths that are not the worst-case paths. By default, the tool saves information about only two paths per startpoint-endpoint pair: the worst maximum-delay path and the worst minimum-delay path. You can save more paths in the database at the expense of runtime and memory. To keep more paths,

- Set the `-keep_paths_within` option to a time value larger than 0.0 (in design time units) to keep paths whose delay is within this value of the worst path.
- Set the `-npaths` option to an integer larger than 1 to specify the maximum number of paths to save for each startpoint-endpoint pair.

For example, the following command keeps all paths that have a slack within 0.2 time units of the worst path, up to a maximum of three paths per startpoint-endpoint pair:

```
nt_shell> trace_paths -keep_paths_within 0.2 -npaths 3
```

The `-full_path_enumeration` option is useful for investigating or debugging the path tracing process. It causes the `trace_paths` command to exhaustively trace all possible paths, without pruning. Using this option is the same as setting the `-keep_paths_within` option to a large time value. The `-full_path_enumeration` and `-keep_paths_within` options are mutually exclusive. To keep the number of reported paths to a reasonable number, use the `-npaths` option in addition to either the `-full_path_enumeration` or `-keep_paths_within` option.

Because of the large runtime, the `-full_path_enumeration` option is practical only with very small designs and when used with the `set_find_path` command to limit the scope of the design being considered in the analysis.

Reducing the Number of Saved Paths

Alternatively, you might want to save fewer paths in the database. These options reduce runtime and memory usage. To save fewer paths,

- Use the `-worst_fanin_path` option to keep only the information about the worst path in the fanin to each endpoint.
- Restrict the scope of the analysis with the `set_find_path` command.
- Use the `-clock_only`, `-min_only` or `-max_only` options to analyze only the clock paths, the minimum delay paths, or the maximum delay paths.

To reduce memory usage, you can set an overall limit on the number of paths saved for the whole design. The `timing_*_**_paths_saved` variables set the limits for different types of

timing paths (where * represents `max` or `min` and ** represents `delay` or `slack`). The variables all default to a value of 100 million.

For example, to limit the total number of maximum-delay paths saved in the database to 500 paths for the whole design, use the following command:

```
nt_shell> set timing_max_delay_paths_saved 500
```

Path Reporting

The `report_paths` command generates a report on timing paths in the design. You must first execute path tracing with the `trace_paths` command, which saves detailed information about the worst-case timing paths into a path database. If you would like to see more paths in the path reports, you must use options during the path tracing operation to ensure that the additional paths are traced and saved in the path database.

There are three report formats: list (the default), endpoint, and detail.

All report types allow you to add columns for path ID and path index (using the `-show_path_id` and `-show_path_index` options) and order the report by delay or by slack (using the `-by` option).

The Path Report List Format

The list report (specified with the `-path_type list` option) displays one line per path and includes the slack, the path delay, the path type, the transition direction at the startpoint, the startpoint, the transition direction at the endpoint, and the endpoint.

You can optionally add columns for the start net and the end net.

An example of the list report is as follows:

```
****************************************
Report :   paths
     -path_type list
     -max
Design :   ALU
...
****************************************

          Path  Path
   Slack    Delay  Type    Startpoint         Endpoint
-------- -------- ----- -- --------------- -- ---------------------
   0.513    4.171   C-L r  clk2            f  Xareg.Xreg63.X5.Mp0.G
   0.515    4.169   C-L r  clk2            f  Xareg.Xreg62.X5.Mp0.G
   0.530    4.171   C-L r  clk2            f  Xbreg.Xreg63.X5.Mp0.G
   0.533    4.169   C-L r  clk2            f  Xbreg.Xreg62.X5.Mp0.G
   ...
```

The Path Report Endpoint Format

The endpoint report (generated with the `-path_type end` option) displays one line per path and includes the endpoint, the total delay, the transition direction (rising or falling), the required time, and the slack.

An example of the endpoint report is as follows:

```
****************************************
Report :   paths
     -path_type end
     -max
Design :   ALU
...
****************************************

                                     Path        Path
Endpoint                            Delay    Required    Slack
------------------------------- -------- -- -------- --------
Xbreg.Xreg50.X5.Mp0.G              9.435 f     6.784   -2.651
Xbreg.Xreg50.X5.Mp0.G              7.935 f     6.784   -2.651
Xbreg.Xreg50.X5.Mp0.G              6.435 f     6.784   -2.651
...
Xbreg.Xreg51.X5.Mp0.G              9.294 f     6.753   -2.542
Xbreg.Xreg51.X5.Mp0.G              7.794 f     6.753   -2.542
...
```

The Path Report Detail Format

The detail report (generated with the `-path_type` option with arguments `short`, `summary`, `full`, `full_clock`, or `full_clock_expanded`) contains a header, a section covering the data path, and a section covering the clock path. Each section contains one line per path segment, but the number and size of the path segments varies according to the specified path reporting option.

An example of the detail report is as follows:

```
****************************************
Report :   paths
     -path_type full
     -max
Design :   ALU
...
****************************************

Net Types (NT):
    D     - Data input
    O     - Data output
    SZ    - Data turnoff output
...  rest of Net Type legend

Startpoint:         clk1 (in port)
Endpoint:           Xbreg.Xreg50.X5.Mp0.G
Path Type:          max
Constraint:         latch setup (transparent)
PBSA Common Net:    clk1

 Path   Incr    Adjust   NT      Point
-----   -----   ------ -----  -- -------------------------------
 0.000          0.000    C&    r clk1 (in)
 0.794  0.794            C&    r Xareg.Xck1.Mn0.G (inv2)
 1.013  0.219            C&    f Xareg.Xck11.Mp0.G (inv2)
...   data path intermediate points
 4.608  0.285            n2&   r Xaddsub.Xadder.X1a2.X1gen.Mn445.G (carry_gen)
 4.959  0.909  -0.558    N2&   f Xaddsub.Xadder.X1a2.X1gen.XG4.X0.Mp0.G (inv2)
 9.435  0.296            L&    f Xbreg.Xreg50.X5.Mp0.G (inv2)
 9.435                           data arrival time
       18.094  -8.658            Total

 6.000          6.000    C&    r clk1 (in)
 6.261  0.261            C&    r Xbreg.Xck4.Mn0.G (inv2)
...   clock path intermediate points
 6.784  0.149                    Xbreg.Xreg50.Mn0.G (muxflop)
        0.784   6.000            Total
 6.784  0.000                    setup time
 6.784  0.000                    clock uncertainty
 6.784                           data required time
---------------------------------------------------------------------
 6.784                           data required time
-9.435                           data arrival time
---------------------------------------------------------------------
-2.651                           slack (VIOLATED)
```

A detail report breaks up paths into different levels of granularity depending on the `-path_type` and `-clock_only` options. Table 10-1 shows the effect of the `-path_type` options, with and without the `-clock_only` option.

Table 10-1 Detail in Path Report Types

	Expanded full clock	**Full clock**	**Full**	**Summary**	**Short**	**Clock only, except short**	**Clock only, with short**
Data path startpoint	X	X	X	X	X		
Data path intermediate points	X	X	X	X			
Data path endpoint, total delays, and adjustments	X	X	X	X	X		
Clock path startpoint	X	X				X	X
Clock path intermediate points	X	X				X	
Chained generated clock intermediate points	X					X	
Clock path endpoint and arrival time	X	X	X		X	X	X
Clock path total delays and adjustments	X	X				X	X
Slack calculation	X	X	X		X		

Net types appear under the column heading NT. The codes are defined in Table 10-2. An abbreviated form of this table appears in the heading of every detail report.

Timing checks for domino circuits depend on the construction of the domino gate. The table refers to type D1 and type D2 domino precharge structures. A type D1 domino structure has a controlling clock on the evaluate stack, whereas a type D2 domino structure does not.

The table also refers to N-domino and P-domino circuits. An N-domino circuit has an NMOS evaluate stack and charges a precharge node to a high voltage during a precharge phase, when the clock is low. A P-domino circuit has a PMOS evaluate stack and discharges a predischarge node to a low voltage during a predischarge phase, when the clock is high. For

more information about the construction of domino precharge circuits, see Domino Precharge Circuit Types.

Table 10-2 Net Types in the NT column in Path Reports

Code	Net type	Code	Net type
D	Data input port	N1	Precharge node of D1 N-domino
O	Data output port	N2	Precharge node of D2 N-domino
SZ	Data turnoff output	N3	Precharge node of D1 N-domino retain
Z	Turnoff	N4	Precharge node of D2 N-domino retain
E	Turnoff enable	N5	Precharge node of D1 N-domino latch
C	Clock	N6	Precharge node of D2 N-domino latch
CX	Clock, dynamic simulation	N7	Precharge node of D1 N-domino flip-flop
L	Latch	N8	Precharge node of D2 N-domino flip-flop
R	Register file latch	n1	Input of D1 N-domino
A	Adjusted latch	n2	Input of D2 N-domino
RA	Adjusted register file latch	n3	Input of D1 N-domino retain
G	Gated clock	n4	Input of D2 N-domino retain
T	Transparent gated clock	n5	Input of D1 N-domino latch
SC	Stopped clock	n6	Input of D2 N-domino latch
U	User-defined constraint	n7	Input of D1 N-domino flip-flop
DU	User-defined data-to-data constraint	n8	Input of D2 N-domino flip-flop
M	Timing model	P1	Predischarge node of D1 P-domino
LP	Latch to latch loop	P2	Predischarge node of D2 P-domino
PC	Precharge clock	P3	Predischarge node of D1 P-domino retain
EC	Eval clock	P4	Predischarge node of D2 P-domino retain
PP	Precharge PC, predischarge EC	P5	Predischarge node of D1 P-domino latch
PN	Precharge EC, predischarge PC	P6	Predischarge node of D2 P-domino latch
GC	Generated clock	P7	Predischarge node of D1 P-domino flip-flop
Mxx	Memory-specific timing checks	P8	Predischarge node of D2 P-domino flip-flop
$	Simultaneous switching inputs	p1	Input of D1 P-domino
&	Net has back-annotated parasitics	p2	Input of D2 P-domino
		p3	Input of D1 P-domino retain
		p4	Input of D2 P-domino retain
		p5	Input of D1 P-domino latch
		p6	Input of D2 P-domino latch
		p7	Input of D1 P-domino flip-flop
		p8	Input of D2 P-domino flip-flop

The Path Header Section

A short header section introduces every path in a detail report. The header contains the following information:

- Startpoint
- Endpoint
- Path type (min or max)
- Constraint label

  The label for a constraint that is automatically derived from the topologies in the path is a general description generated by the tool. An example of an automatic constraint label is “latch setup.”

  If the type of an automatic timing check is modified by a user action, the constraint label displayed in the path report reflects that change. The new constraint is followed by parentheses enclosing the words “switchover from” and the old constraint label. For example, changing the transparency of a latch affects timing checks for that latch. The constraint label in the path report might say “non-transparent setup (switchover from latch setup (transparent)).”

  The label for a user-defined constraint is specified in the constraint generation command, such as the `create_timing_check` or `set_data_check` command, and can be any string.

- Analysis-dependent properties

  Additional information that might appear, depending on the analysis options, includes the PBSA Common Net and PBSA Complement.

Properties That Always Appear in the Path Report

The properties that are always reported for each path segment (each line) in the report are as follows:

- Path

  The total delay from the startpoint to the location in the Point column.

- Incr

  The change in delay from the previous line; the delay between the location in the previous line and the location in the current line.

- Adjust

  Any change to the intrinsic delay calculation, or a starting value for partial paths. Adjustments are already included in the total delay in the Path column.

- NT

  The net type of the net in the Point column.

- Transition direction (unlabeled)

  The transition direction at the location in the Point column; r for rising and f for falling.

- Point

  The endpoint of the path segment reported on the line. You can optionally show the subcircuit name associated with the point.

Optional Properties in the Detail Report

For any of the detail reports, you can display additional properties by using one or more options with the `report_paths` command. Each property appears in a separate column in the report. The available properties are as follows:

- Effect of delay coefficients (two options)

  The `-coefficient_adjustment` option reports the portion of the stage delay caused by the `set_delay_coefficients` command. The `-coefficient_ratio` option reports the ratio of the total delay (the sum of the unadjusted delay and the delay adjustment) to the unadjusted delay.

- Wire delay (using the `-wire_delay` option)

  The measured delay from the driver diffusion to the gate in the Point column. You must set the `timing_save_wire_delay` variable to `true` before path tracing to save the wire delay information for reporting (it is always used for analysis).

  The calculated delay depends on several factors, including the parasitic RC values and the signal slew rate. Using signal integrity analysis affects wire delay because it changes the effective capacitance on a net.

- Crosstalk delta (using the `-crosstalk_delta` option)

  The portion of the stage delay due to crosstalk. You must enable signal integrity analysis to analyze crosstalk. Crosstalk delta might be positive or negative.

  Crosstalk delta values for dynamic clock simulation (DCS) regions in path reports are reported as “n/a” instead of zero. The stage delay includes the dynamically simulated delay for the DCS region. However, the specific portion of the delay due to crosstalk is not separated.

  If any stage of a path has a crosstalk delay value of “n/a,” the crosstalk delta value for the complete path is also reported as “n/a.”

- Transition time (using the `-transition_time` option)

  The transition time at the location in the Point column.

- Full transition time (using the `-full_transition_time` option)

  The extrapolated transition time for a full-rail swing.

- Rail voltage (using the `-rail_voltage` option)

  The rail voltage at the location in the Point column.

- Final voltage (using the `-final_voltage` option)

  The final voltage at the location in the Point column.

- Capacitance (using the `-capacitance` option)

  The total capacitance of the path segment between the endpoint in the previous line and the endpoint in the current line.

- Stage variation (using the `-variation` option)

  The variation of the path segment between the endpoint in the previous line and the endpoint in the current line. Values are nonzero only if you run the `trace_paths` command with the `-pocv` option.

Paths That Contain Generated Clocks

Some differences apply to paths that contain generated clocks. A generated clock represents a change in clock properties, which affects the behavior of downstream circuits and the definition of downstream timing checks. NanoTime stops path tracing at a generated clock, saves the paths that end at that point, and begins tracing new paths that start at that point.

For a path that includes a generated clock, the standard path report displays the paths that begin at the generated clock. To include the path from the clock source net to the generated clock, use the `-path_type full_clock_expanded` option. However, the path from the clock source net to the generated clock is an unconstrained path. You cannot report this path using the `-from`, `-rise_from`, or `-fall_from` options.

See Also

- Specification of Paths to Report
- Latch Error Recovery in Path Reports
- Custom Path Reports

Specification of Paths to Report

You must use one of the `-min, -max,` or `-path_id` options with the `report_paths` command. They are mutually exclusive and specify maximum delay paths, minimum delay paths, or explicit path IDs. Path IDs are arbitrary numbers that NanoTime assigns during path tracing. You can see path IDs by using the `-show_path_id` option with any report. Alternatively, you can provide a collection of paths created by the `get_timing_paths` command.

By default, the `report_paths` command lists the paths in order of increasing slack, starting with the path having the least (most negative) slack. To list paths in order of delay instead, starting with the path having the worst delay, use the `-by delay` option. The `-by delay` option reports unconstrained paths (for example, a path to an output that has no output delay set), as well as constrained paths.

Path reports might be very long. Options for the `report_paths` command restrict the amount of information in the report.

To report only the 10 worst-case maximum-delay paths, use the following command:

```
nt_shell> report_paths -max -max_paths 10
```

To limit the report to paths having a slack worse than a specified value, use the `-slack_less_than` option. For example, to report only the maximum-delay paths that have a negative slack (those that violate timing), use the following command:

```
nt_shell> report_paths -max -slack_less_than 0
```

Many different paths can share a common endpoint. By default, all of these paths are reported. To specify the number of paths reported per endpoint, use the `-nworst` option. For example, to report only the single worst path per endpoint, use the following command:

```
nt_shell> report_paths -max -nworst 1
```

To reduce the number of paths in the report when there are many similar paths (such as in designs with buses), use the `-suppress_similar_paths` option in conjunction with the `create_net_group` command. When two or more paths are similar, only the first path appears in the report, along with a notation showing the number of similar paths that are not being reported. This results in a much shorter report.

A path is considered similar to another path if both of the following conditions are true:

- The path has at least one net in the same group as the corresponding net in the other path. Define net groups with the `create_net_group` command.
- The total path delays, slacks, and stage delays of the two paths are similar.

  You can define how paths should be considered to be similar in terms of relative or absolute delay or slack by using the `report_paths_suppress_similar_paths_*` variables.

Path Selection Using Timing Points

You can specify the paths to report based on the path startpoints, throughpoints, and endpoints. To do so, use one or more of the `-from`, `-through`, `-to` and related options with the `report_paths` command. For example, the following command reports only the maximum-delay paths that end at port BUS[0]:

```
nt_shell> report_paths -max -to [get_ports BUS[0]]
```

The `-through` option might return unexpected results because of the way NanoTime saves paths during path tracing. When executing the `trace_paths` command, you can choose to save the worst N paths to an endpoint, or all of the paths within a certain delay of the worst path to that endpoint, but you cannot choose to save paths based on startpoints or throughpoints. When you use the `report_paths -through` option, the database might not contain any paths that pass through the point specified in the `-through` option. Even if some paths are reported by using the `report_paths -through` command, there is no guarantee that those paths represent the worst delays to the specified endpoint.

To force NanoTime to perform path tracing through a specific point, use one or more `set_find_path` commands with the `-through` option before the `trace_paths` command. You can then use the `report_paths` command to examine the delays on these paths. In this case, path tracing is performed only on the specified paths. Delays in the rest of the design are not saved into the path database and are therefore not available for reporting. The paths defined with the `set_find_path` command do not appear in the report generated by the `report_exceptions` command; to list them, use the `report_find_path` command.

See Also

- Path Reporting
- Latch Error Recovery in Path Reports
- Custom Path Reports

Latch Error Recovery in Path Reports

By default, NanoTime uses latch error recovery during path tracing. Error recovery is controlled by the `trace_latch_error_recovery` variable. Its default is `true`. During path tracing, there is no indication whether latch error recovery is being used, regardless of the variable setting. However, the path report includes latch error recovery details.

For example, consider the case of a data signal arriving late at a structure for which a transparent check is being done during maximum delay path tracing. With latch error recovery enabled, instead of stopping the path tracing, NanoTime adjusts the time to the closing edge of the clock and continues path tracing.

A portion of a path report with latch error recovery is shown in the following example:

```
Path   Incr   Adjust Trans  Cap    NT    Point                  Net
----   -----  ------ -----  -----  --    --------------------   ------
              0.100                       clock clk1 (rise)
              4.000                       input external delay
4.100         0.000  0.100  0.020  D&  r data1 (in)             data1
4.100  0.000         0.100  0.020  &   r X1,Mn0.G (inv2)        data1
4.132  0.032         0.039  0.020  &   f X2.Mp1.G (latch2)      data1n
4.153  0.113 -0.092  0.185  0.045  A&  r X2.X6.Mn0.G (inv2)     X2.lat1n
  (cycle=2 clock=clk1 low period=4.000 open=1.960 f close=4.153 r pin=X2.Mp0.G)
4.185  0.032         0.047  0.028      f  X11.Mp0.G (inv2)      q0
4.216  0.031         0.041  0.019      r  X4.Mn1.G (latch1)     q0n
4.288  0.062         0.086  0.045  L&  f  X4.X6.Mp0.G (inv2)    X4.lat1n
4.288                       data arrival time
       0.271  4.008              Total
```

The input, data1, arrives at latch node X2.lat1n at 4.245 ns (the initial 4.1 ns value plus the values in the increment columns to this point, 0.032 ns and 0.113 ns). The latch clock turns off at 4.153 ns according to the cycle line of the report, which is 0.092 ns before the data1 input arrives. Using latch error recovery, NanoTime adjusts the time by -0.092 ns so that the data arrives at the latch just as the clock turns off. Path tracing continues to the X4.lat1n latch node, where the path stops because it has arrived before the opening edge of the clock for that latch. Latch error recovery allows you to find additional timing errors after the first setup error.

See Also

- Path Reporting
- Specification of Paths to Report
- Custom Path Reports

Custom Path Reports

The `get_timing_paths` command creates a collection of paths according to specified criteria and is useful for generating customized path reports. Use the `foreach_in_collection` command to iterate through the paths in the collection, then use the `get_attribute` command to obtain information about the paths.

To see a list of attributes associated with a timing path, use the `list_attributes` or `report_attribute` command. For example,

```
nt_shell> trace_paths
nt_shell> set path1 [get_timing_paths -max -max_paths 1]
nt_shell> report_attribute -application $path1
 ...
Design    Object         Type      Attribute Name   Value
-----------------------------------------------------------------
ALU       timing_path    string    object_class     timing_path
ALU       timing_path    int       path_id          10317
ALU       timing_path    float     arrival          4.382556
ALU       timing_path    float     required         4.000000
ALU       timing_path    float     delay            3.382556
...
```

The following procedure displays the startpoint name, endpoint name, and the slack of the worst long path to each unique endpoint:

```
proc custom_report_worst_path_per_endpoint {} {
    echo [format "%-20s %-20s %7s" "From" "To" "Slack"]
    echo "-------------------------------------------------"
    foreach_in_collection path [get_timing_paths -max -nworst 1] {
        set slack [get_attribute $path slack]
        set startpoint [get_attribute $path startpoint]
        set sp_obj [get_attribute $startpoint object]
        set endpoint [get_attribute $path endpoint]
        set ep_obj [get_attribute $endpoint object]
        echo [format "%-20s %-20s %s" \
                [get_attribute $sp_obj full_name] \
                [get_attribute $ep_obj full_name]  $slack]
    }
}

nt_shell> custom_report_worst_path_per_endpoint
From                 To                     Slack
-------------------------------------------------
clk1                 out2                -3.000000
clk1                 out1                -3.000000
clk1                 xl4.xi2.mn1.g       -1.000000
clk1                 xl3.xi2.mp1.g       -1.000000
clk1                 xl3.xi2.mn1.g       -1.000000
in0                  xl1.xi2.mn1.g        6.000000
```

See Also

- Path Reporting
- Specification of Paths to Report
- Latch Error Recovery in Path Reports

Multi-Input Switching

By default, for a circuit with multiple inputs, NanoTime considers the delay that results from switching one input at a time. The tool selects one of the inputs as the switching input and sets the nonswitching inputs to a constant logic state.

However, in actual usage, the delay might be different when two or more inputs switch simultaneously. NanoTime offers two types of multi-input switching analysis: user-guided and timing-based. Both types of multi-input switching analysis require a NanoTime Ultra license.

See Also

- User-Guided Multi-Input Switching
- Timing-Based Multi-Input Switching
- Multi-Input Skew Capture
- Multi-Input Switching in the Path Report
- Multi-Input Switching With the write_spice Command

User-Guided Multi-Input Switching

You can achieve a more conservative analysis by considering the timing with multiple inputs switching simultaneously at the cost of more runtime. To do so, use the `set_conservative_max_delay` and `set_conservative_min_delay` commands.

The `set_conservative_max_delay` command directs NanoTime to conduct maximum delay analysis by using the worst-case, user-guided, multi-input switching of inputs connected to the channel-connected region of the specified list of nets. Using this command might result in longer reported maximum delays and shorter maximum-delay slacks.

Similarly, the `set_conservative_min_delay` command affects the calculation of minimum delays resulting from user-guided multi-input switching of inputs connected to the channel-connected region of the specified list of nets.

The `write_spice` command, when applied to the applicable path, shows the worst-case simultaneous-switching sensitization used for analysis of the timing path.

Guidelines for User-Guided Multi-Input Switching

In a simple NAND or NOR gate, when minimum delay analysis is specified, a shorter delay occurs if all inputs switch simultaneously. When maximum delay analysis is specified, a longer delay can occur.

You should use this feature when, in a given stage, there is a possibility of multi-input switching that can result in one of the following situations:

- A maximum delay longer than the delay resulting from conventional single-input switching scenarios
- A minimum delay shorter than the delay resulting from conventional single-input switching scenarios

Specifying Minimum or Maximum Delay Analysis

To enable nets for maximum delay analysis in a channel-connected region, use the `set_conservative_max_delay` command, which takes a list of nets as an argument. You can specify any net in the channel-connected region.

To cancel the effect of a previously executed `set_conservative_max_delay` command, use the `remove_conservative_max_delay` command.

Similarly, for minimum-delay analysis, use the `set_conservative_min_delay` and `remove_conservative_min_delay` commands.

The side inputs configured for conservative minimum or maximum delay are not checked for logical consistency. For this reason, using minimum or maximum delay switching analysis globally is not recommended.

Assumptions for User-Guided Multi-Input Switching

The following assumptions apply:

- Side inputs configured as switching are assumed to switch simultaneously with the trigger (in other words, with no skew).
- The slew rates and voltage swing on the inputs that switch with the trigger are all the same.

Maximum Delay Analysis Examples

Figure 10-7 through Figure 10-9 are maximum delay analysis examples illustrating the function of user-guided multi-input switching.

Figure 10-7 Three-Input NAND With Maximum Delay

In the circuit shown in Figure 10-7,

- The trigger switching high is the NMOS device connected to input A. The nets that are configured as switching (high) are B and C, the gate terminals of the other NMOS devices in this circuit.
- For this circuit, when a PMOS device is a trigger (for example, A is switching low), there are no other switching inputs. This means inputs B and C are tied high.
- The case for a three-input NOR device is similar; all PMOS inputs are simultaneously switched high.

Figure 10-8 Parallel Stack With Maximum Delay

In the circuit shown in Figure 10-8,

- The switching stack has all devices switching.
- The side branches are configured for maximum loading.
- The trigger for this circuit is the NMOS device connected to input A rising on the first NMOS stack. The NMOS devices connected to B and C (switching stack) are configured as switching. The side inputs P, Q, and R are tied high for side input loading.

Figure 10-9 AOI (Complementary Side) With Maximum Delay

In the circuit shown in Figure 10-9,

- The NMOS device connected to net A is the trigger.
- The non-trigger switching NMOS devices are those connected to nets B and C. The PMOS device connected to D is tied low (logic 0) in this case.

Minimum Delay Analysis Examples

Figure 10-10 through Figure 10-15 are minimum delay analysis examples that illustrate the function of the simultaneous switching feature.

Figure 10-10 Three-Input NAND With Minimum Delay

In the three-input NAND shown in Figure 10-10,

- The trigger switching low is the PMOS device connected to input A. The nets which are configured as switching (low) are B and C, which are the gate terminals of the other PMOS devices in this circuit.
- A three-input NOR device is similar. (All NMOS inputs are simultaneously switched high.)

Figure 10-11 Parallel Stack 1 With Minimum Delay

In the circuit shown in Figure 10-11,

- Each stack not controlled by the trigger has one switching device.
- The non-trigger switching device is the one closest to the output.
- The trigger for this circuit is the NMOS device connected to input A rising on the first NMOS stack. The non-trigger switching NMOS devices are those connected to nets P, Q, and R respectively. All other gate terminals are tied high (logic 1) in this case.

Figure 10-12 Parallel Stack 2 With Minimum Delay

In the circuit shown in Figure 10-12,

- Each stack not controlled by the trigger has one switching device.
- The non-trigger switching device is the one closest to the output.
- The trigger for this circuit is the NMOS device connected to the middle input A in the first column of the NMOS stacks. The non-trigger switching NMOS devices are those connected to nets P, Q, and R respectively. All other gate terminals are tied high (logic 1) in this case.

Figure 10-13 Parallel Stack 3 With Minimum Delay

In the circuit shown in Figure 10-13,

- Each stack that is not controlled by the trigger has one switching device.
- The non-trigger switching device is the one closest to the output.
- The trigger for this circuit is the NMOS device connected to A on the first column of the NMOS stacks. The non-trigger switching NMOS devices are those connected to nets P, Q, and R respectively. All other gate terminals are tied high (logic 1) in this case.

Figure 10-14 Evaluation Clock With Minimum Delay

In the circuit shown in Figure 10-14, the trigger is the NMOS device connected to input CLK rising. Since all stacks are connected to this evaluation device, other inputs are tied high (logic 1).

Figure 10-15 AOI (Complementary Side) With Minimum Delay

In the circuit shown in Figure 10-15,

- The PMOS device connected to A is the trigger.
- The non-trigger switching PMOS devices are those connected to B and C (which is closest to output). The PMOS device connected to D is tied low (logic 0) in this case.

See Also

- Timing-Based Multi-Input Switching
- Multi-Input Skew Capture
- Multi-Input Switching in the Path Report
- Multi-Input Switching With the write_spice Command

Timing-Based Multi-Input Switching

In timing-based multi-input switching analysis, the NanoTime tool automatically detects channel-connected regions that should be analyzed together based on timing window information. This feature is useful when multiple inputs to a stage switch simultaneously. Set the `timing_enable_multi_input_switching` variable to `true` to enable it.

Timing-based multi-input switching is an iterative analysis. It can be used at the same time as signal integrity analysis and differential skew analysis, which are also iterative. If more than one type of iterative analysis is enabled, NanoTime updates the results for all enabled features during each path tracing iteration until all exit criteria are satisfied.

Note:

In rare cases, oscillations between iterations might result, especially when strong aggressors are logically constrained to switch in a direction that removes pessimism. You should always use the worst-case results for your analysis and choose iteration exit criteria that produce the worst-case results.

The variables in Table 10-3 affect the behavior of the iterations.

Table 10-3 User Controls for Iterations

Variable	Definition
`timing_min_iteration_count`	Minimum number of iterations. Applies to all iterative analysis types. The default is 3.
`timing_multi_input_max_iteration`	Maximum number of iterations. The default is 3.
`timing_multi_input_exit_reevaluated_nets`	Number of reevaluated nets below which analysis stops. The default is 0.
`timing_multi_input_exit_reevaluated_nets_pct`	Percentage of reevaluated nets below which analysis stops. The default is 0.
`timing_multi_input_overlap_tolerance`	Tolerance that determines if two or more input windows overlap. The default is 0.003 ns. Large values might cause iteration instability.

To report the nets that are reselected or excluded from reselection during timing-based multi-input switching iterations, use the `report_multi_input_switching` command.

The two forms of multi-input switching analysis (user-guided and timing-based) are mutually exclusive. In other words, you cannot use the `timing_enable_multi_input_switching` variable with the `set_conservative_max_delay` or `set_conservative_min_delay` commands.

The following assumptions apply to timing-based multi-input switching:

- Side inputs configured as switching are assumed to switch simultaneously with the trigger (no skew).
- The slew rates and voltage swing on the inputs that switch with the trigger are the same.

Excluding Nets From MIS Analysis

Multi-input switching might not be appropriate for all circuits. For example, some scan clock nets, clock nets, and reset lines might not be suitable for multi-input switching analysis. In addition, side inputs configured for multi-input switching are not checked for logical consistency. For this reason, you should exclude latch nodes where complementary signals might switch simultaneously.

The `exclude_multi_input_switching` command takes a list of nets as an argument. This command excludes the specified nets and their channel-connected regions from consideration during timing-based multi-input analysis.

You can refine exclusions as follows:

- Use the `-min` option to exclude minimum delay analysis. For example, this includes situations where multiple inputs that switch together cause faster delays when parallel transistors (or stacks) turn on at the same time.
- Use the `-max` option to exclude maximum delay analysis. For example, this includes situations where multiple inputs that switch together cause slower delays when parallel transistors (or stacks) turn off at the same time that series transistors are turning on.
- Use the `-rise` and `-fall` options to exclude analysis on rising or falling transitions at the specified nets.

Note that the command *excludes* nets from analysis. Therefore, to perform only maximum-delay analysis and only on rising transitions, you must use the `exclude_multi_input_switching` command with the `-min` and `-fall` options.

Remove a previously specified exclusion by using the `-remove` option of the `exclude_multi_input_switching` command.

See Also

- User-Guided Multi-Input Switching
- Multi-Input Skew Capture
- Multi-Input Switching in the Path Report
- Multi-Input Switching With the write_spice Command

Multi-Input Skew Capture

For designs with multi-input switching, NanoTime can identify correlated input signals and capture their relative skew, which can then be used for subsequent delay calculations.

A multiplexer is an example of a circuit with correlated input signals. In standard path tracing, the disabling or inactive select signals are tied to constant voltage sources to facilitate path tracing from the active or enabling select signal. However, in reality skew exists between the multiplexer select signals, such as that introduced by a decoder as shown in Figure 10-16. Signals A0 and A1 determine which one of the four signals D0 to D3 switches to a logic high state. With respect to the A0 and A1 inputs, signal D3 switches sooner than signals D0, D1, and D2 because there is no stage delay from the inverters.

Figure 10-16 Decoder and Multiplexer

If a disabling select signal arrives at the multiplexer after an enabling select signal, contention occurs because the multiplexer output cannot fully switch until the effect of the disabling select signal is gone. This introduces an additional delay through the multiplexer that must be simulated to avoid overly optimistic results.

Use the `mark_correlated_region` command to help NanoTime analyze multiple switching inputs during a delay calculation. List the inputs (ports, pins, or nets) in the argument of the `-inputs` option and the outputs (ports or nets) in the argument of the `-outputs` option.

Every output object must be in the transitive fanout of one or more of the inputs. For the decoder in Figure 10-16, the command is as follows:

```
nt_shell> mark_correlated_region -inputs {A0 A1} -outputs {D0 D1 D2 D3}
```

This command must be executed between the `link_design` and `check_design` commands. During path tracing, NanoTime computes the best and worst paths to each output from a transition on an input. NanoTime then uses the delay differences from these paths to the current path to compute the skew. The slopes at the correlated path outputs are also captured and used in the simulation of the following stage.

If you are using multi-input switching analysis, you should exclude the correlated outputs from simultaneous switching to calculate accurate skew.

If you want to define the skew between two input pins manually instead of directing NanoTime to calculate it, use the `set_correlated_input` command.

A pin attribute named `is_correlated_output` is set to `true` for correlated outputs. You can obtain a list of the correlated outputs for a pin by checking the contents of the `correlated_output_pins` attribute.

Restrictions on Correlated Regions

A correlated region has the following restrictions:

- A pin can be a correlated output of only one correlated region.
- The correlated region cannot include any clocked or register devices, such as latches, flip-flops, or RAM cells.
- The correlated region should not include an overly large section of the design. Otherwise, runtime could be adversely affected.
- The correlated region outputs must correspond directly with the inputs to the following stage. The inputs are usually nets driving select pins of multiplexer topologies.
- The correlated outputs should be specified on a channel-connected block boundary.

The `write_spice` command does not include the skews or offsets for multi-input switching stages captured by the `mark_correlated_region` command.

See Also

- User-Guided Multi-Input Switching
- Timing-Based Multi-Input Switching
- Multi-Input Switching in the Path Report
- Multi-Input Switching With the write_spice Command

Multi-Input Switching in the Path Report

NanoTime reports timing-based multi-input switching nets in the detailed path report if you specify the `-path_type full` option. Nets analyzed for timing-based multi-input switching have a “$” symbol tagged under the node type field. The legend for the report is also updated.

The following example shows a path report in which the delay for node out2n has been calculated using timing-based multi-input switching.

```
Startpoint:     ck1 (in port)
Endpoint:       out2 (out port)
Path Type:      max
Constraint:     set_output_delay

Path     Incr     Adjust    NT              Point              Net
-------- -------- -------- ---- -- ------------------------------
2.000     2.000              C    f ck1 (in)              ck1
2.000     0.000              C    f X4.X1.Mp0.G (inv2)    ck1
2.053     0.053              C    r X4.X2.Mn0.G(inv2)     X4.clkn
2.076     0.023              C    f X4.Mp0.G(latch1)      X4.clk
2.172     0.096              L    r X4.X6.Mn0.G(inv2)     X4.lat1n
2.210     0.038                   f X6.Mp1.G (nand2)      q1
2.248               0.038    $    r X7.Mn0.G (inv2)       out2n
2.263               0.015    O    f out2 (out)            out2
2.263                           data arrival time
             0.263  2.000                 Total
```

See Also

- User-Guided Multi-Input Switching
- Timing-Based Multi-Input Switching
- Multi-Input Skew Capture
- Multi-Input Switching With the write_spice Command

Multi-Input Switching With the write_spice Command

If you use the `write_spice` command to create a SPICE deck for a circuit that uses multi-input switching, the inputs that switch at the same time as the trigger net are represented by a voltage-controlled voltage source definition.

In the following example, node q0 is a voltage source that switches with q1:

```
.SUBCKT Path22_Arc5 q1 out2n
* q1 input conn_to_trigger
* out2n output conn_to_trigger
MNX6_Mn0 out2n q0 X6_n1 Path22_Arc5_gnd nch W=1.5u L=0.13u AS=0.405p
 AD=0.405p PS=3.54u PD=3.54u NRS=0.091 NRD=0.157
...
.IC V(X6_n1)=0
V1 Path22_Arc5_gnd 0 DC=0
.IC V(out2n)=0
E2 q0 0 q1 0 1.0
V3 Path22_Arc5_vdd 0 DC=1.2
.ENDS Path22_Arc5
```

See Also

- User-Guided Multi-Input Switching
- Timing-Based Multi-Input Switching
- Multi-Input Skew Capture
- Multi-Input Switching in the Path Report

Controlling Accuracy

The accuracy of NanoTime timing analysis is comparable to that of HSPICE simulations. You can analyze much larger designs with the NanoTime tool than with the HSPICE dynamic simulator, but this capability requires simplifications and settings that might affect the results. As with most analysis techniques, greater accuracy often requires longer runtime. NanoTime provides many options for customizing this tradeoff.

Note:

A different aspect of accuracy is whether the calculated delays match measurements on actual devices. Within NanoTime, if your design setup has issues such as incorrect transistor models, NanoTime results will be incorrect compared to finished devices—but they will match a similarly set up (similarly incorrect) HSPICE run.

This section describes the following features that strongly affect accuracy:

- Initial Condition Adjustment
- Parasitics Options That Affect Accuracy
- Accuracy Considerations When Using Timing Models
- Using an External Simulator for Selected Timing Arcs
- Nonlinear Waveform Analysis
- Extended Sidebranch Analysis
- Predriver Mix Ratio
- Load and Miller Capacitance Settings

Several NanoTime features that are very important for high-accuracy analysis are described in other parts of this user guide, as follows:

- Chapter 7, “Differential Circuits”
- Chapter 16, “Path-Based Slack Adjustment Analysis”
- Chapter 14, “Dynamic Simulation”
- Chapter 17, “Signal Integrity Analysis”

See Also

- Controlling Runtime
- Chapter 11, “Correlation with HSPICE Analysis”

Initial Condition Adjustment

When analyzing a timing path, the NanoTime tool initializes side inputs and internal nodes to appropriate equilibrium values. For example, consider the three-input NAND gate in Figure 10-17. The path of interest is from input B to output OUT and the stimulus (trigger) is a rising transition at B. A rising transition at B has an effect on the output only if the other two NMOS transistors are on; therefore side inputs A and C are set to logic 1 (VDD). Node x2 is initially pulled high through PMOS transistor mpB and NMOS transistor mnA. For transistor mnA to be on, its gate voltage must be higher than node x2 by at least the threshold voltage. Therefore the initial condition for node x2 is VDD minus the NMOS threshold voltage.

Figure 10-17 Initial Condition Adjustment

You can control the initial conditions as follows:

- The `set_initial_condition_adjustment` command operates on the channel-connected blocks of specified nets. The command scales the threshold voltage before adding it to the ground voltage (for PMOS) or subtracting it from the supply voltage (for NMOS). You can use different scaling factors for NMOS and PMOS transistors.
- The `weak_vdd_ic_multiplier_limit` global variable specifies the lowest initial condition voltage allowed in an NMOS transistor stack.
- The `weak_gnd_ic_multiplier_limit` global variable specifies the highest initial condition voltage allowed in a PMOS transistor stack.

For example, the following command causes the internal node in Figure 10-17 to be set to VDD instead of the default of VDD-VTHn by specifying a scaling factor of 0:

```
set_initial_condition_adjustment -nfactor 0 [get_nets x2]
```

For level 72 and level 101 BSIM transistor models, the NanoTime tool uses current flow to determine the transistor threshold voltages, which in turn are used to set accurate initial voltage conditions.

Pessimism Reduction for High-Impedance Nets

To reduce delay pessimism, the following features provide more control over the initial conditions of the intermediate nodes of high-impedance nets:

- The `timing_high_impedance_pessimism_reduction` variable, when set to `true`, sets the initial condition of intermediate nodes of high-impedance nets to the final voltage that occurs after switching, which provides a less pessimistic delay calculation.

  By default, the variable is `false` and NanoTime assigns a pessimistic initial condition to the intermediate nodes. For example, when a long-channel inverter is driven by a rising input signal, the intermediate node in the NMOS stack is initially in a high impedance state. If you set the variable to `true`, the intermediate node in the NMOS stack is set to logic 0 (ground) to match the expected state after switching.

- The `high_impedance` net attribute is set to `true` as a result of setting the `timing_high_impedance_pessimism_reduction` variable to `true`.

- The `-ignore_high_impedance_pessimism_reduction` option of the `mark_net` command allows you to name specific nets whose initial conditions should be set by the default method, even if the `timing_high_impedance_pessimism_reduction` variable has caused them to be marked by the `high_impedance` attribute.

- The `-ignore_high_impedance_pessimism_reduction` option of the `erase_net` command allows you to remove a net marking previously made with the same option of the `mark_net` command.

Parasitics Options That Affect Accuracy

Many aspects of parasitic netlist creation and annotation can affect the timing results. Pay attention to the following items for high-accuracy analysis:

- Parasitics pruning

  NanoTime prunes low-impact resistors and capacitors to improve runtime. The tool combines or removes RC sections from each net to make a simpler representation that has a path delay within a tolerance of the original delay of the RC network. The tolerance must be equal to or larger than the `rc_reduction_min_net_delta_delay` variable and less than or equal to the `rc_reduction_max_net_delta_delay` variable. The defaults are 1 and 10 ps, respectively.

  Setting the variables to 0 turns off pruning completely. However, the accuracy improvement is usually small while the increase in runtime might be significant. Setting

the `rc_reduction_max_net_delta_delay` variable to a value of 0.005 ns (5 ps) is usually sufficiently small.

- Input skew

  By default, NanoTime analysis ignores skew caused by parasitic RC delay between connected trigger devices. Figure 10-18 shows a circuit with three connected PMOS gates. The parasitic RC delay between these gates might be large enough to affect the accuracy of the timing analysis. The `set_enable_input_spf_skew` command allows you to set a threshold in time units for evaluating the magnitude of this RC delay. If the value exceeds the threshold, the timing analysis takes the input skew into consideration.

  Common-trigger skew might affect fingered transistors, especially if the number of fingers is larger than 4.

  Large RC skew might result in very nonlinear waveforms. In this case, use nonlinear waveform analysis with a sufficient number of samples to represent the waveform accurately.

Figure 10-18 Input Skew

- Minimum capacitance

  The default minimum capacitance used in NanoTime analysis is 1e-6 pF. When reading in parasitic capacitance values, the tool automatically converts any values smaller than this to the minimum value. You can modify this conversion by setting the `parasitics_min_capacitance` variable to a smaller value such as `1e-7`.

- Transistor parameter inheritance

  Missing parameters in device parameter files (SPF or DSPF files) are inherited from corresponding netlist transistors. The `parasitics_suppress_dpf_inheritance` variable specifies parameters that should be ignored during macro-model device parameter annotation. For example, the multiplication factor should not be inherited from a netlist transistor to post-layout fingered devices. In this case, use the following command to ignore the multiplication factor:

  ```
  set parasitics_suppress_dpf_inheritance "multi nf"
  ```

- Contact resistance

  Enable rail contact resistance analysis by setting the `parasitics_enable_rail_net_resistance` variable to `true`.

See Also

- Working With Parasitics Files
- Variables for Parasitic Data Analysis

Accuracy Considerations When Using Timing Models

When you use designs that include timing models, ensure that there are no DELC-109 and DELC-110 warnings. Although these warning messages do not halt the analysis, they highlight issues with the timing models that might affect the simulation accuracy.

DELC-109 warnings indicate that the slew and load when the timing model is used are not within the characterization range of the input slope or the output load for the model. You can use variables to control how the tool interprets slews from library cells, but for the best accuracy, you should generate a new timing model with the correct operating conditions.

DELC-110 warnings indicate that the library cell nominal voltage is different from the analysis voltage. You can control the size of the allowable voltage difference. However, NanoTime analysis does not perform any scaling based on the voltage difference. For best accuracy, you should generate a new timing model with the correct operating conditions.

See Also

- Effects of Slew Thresholds on Model Use

Using an External Simulator for Selected Timing Arcs

Some analysis conditions pose a challenge for the NanoTime internal simulator, such as stages with very slow input transitions, multivoltage supplies, or small RC time constants.

To obtain more accurate results for specific nets, you can use the `set_simulator` command to specify an external simulator instead of the default internal simulator, at the cost of additional runtime. The simulator choices are as follows:

- The HSPICE simulator (using the `-hspice` option; the default)
- The FineSim tool (using the `-finesim_embedded` option)

This feature requires a NanoTime Ultra license.

The number of paths and therefore the runtime might increase greatly if you select a large number of nets for external simulation.

Table 10-4 compares the features of the HSPICE and FineSim dynamic simulators when used within the NanoTime tool.

Table 10-4 External Simulator Comparison

Simulator	Simulator license required	Parallel runs supported	Variable to specify launch command	SI noise analysis supported
HSPICE	Yes	Yes	Yes	No
FineSim	No	No	No	Yes

The use of an external simulator interacts with other choices in the NanoTime flow as follows:

- Timing models created with the `-hspice_timing` option of the `extract_model` command use the HSPICE tool to simulate the model paths. The `set_simulator` command has no effect on these models.
- Timing models created without the `-hspice_timing` option of the `extract_model` command have improved accuracy if you use the `set_simulator` command to simulate the paths saved in the model.
- Dynamic clock simulation takes precedence over the `set_simulator` command.
- Dynamic delay simulation for regions marked with the `mark_simulation` command takes precedence over the `set_simulator` command.
- Signal integrity (SI) delay analysis is supported. The external simulation does not change any aspect of aggressor timing window determination.
- Multi-input switching (MIS), path-based slack adjustment (PBSA), and parametric on-chip variation (POCV) analysis are supported.
- SI noise analysis is supported with the FineSim tool, but not with the HSPICE tool.

Controlling HSPICE Simulations

HSPICE simulations are launched using the command specified in the `sim_hspice_cmd` variable. By default, the variable is set to the following string:

```
hspice -i <input_deck> -o <listing_file>
```

Note:

You can change the command that launches the simulation tool in your computing environment. However, the `<input_deck>` and `<listing_file>` tags are syntax terms that the NanoTime tool uses as variables to represent the correct input and output files. You must use these tags exactly as written, including the angle brackets and the wording inside the brackets.

Analyzing All of Part of a Channel-Connected Block

Figure 10-19 illustrates the effect of the `-ccb_containing` option, with the original circuit on the left and the reduced circuit on the right.

Figure 10-19 Example Circuit

The and-or-invert gate includes ten transistors that are all part of the same channel-connected block. There are many possible paths through this CCB. However, only a few of them might be of interest. For example, case analysis commands might set input signal A to logic 0 and input signals B, C, and D to logic 1. The reduced circuit contains the essential devices to analyze the effect of a transition of input signal E, therefore simulating the entire CCB is unnecessary.

The following command uses the external simulator to analyze only paths through net z:

```
set_simulator z
```

The following command uses the external simulator to analyze all paths through the CCB:

```
set_simulator -ccb_containing z
```

Many more paths are analyzed in this case. The same paths are analyzed no matter which net within the CCB (for example x, y, w, or z) is used as the command argument.

Output

A detailed path report includes a comment on the line after a timing arc whose values are derived from external simulation, as shown in the following example for an HSPICE simulation:

```
Path   Incr   Adjust Trans  Cap    NT    Point                   Net
----   -----  ------ -----  -----  --    --------------------    ------
              0.100                       clock clk1 (rise)
              4.000                       input external delay
4.100         0.000  0.100  0.020  D&  r data1 (in)              data1
4.100  0.000         0.100  0.020  &   r X1,Mn0.G (inv2)         data1
4.132  0.032         0.039  0.020  &   f X2.Mp1.G (latch2)       data1n
4.153  0.113 -0.092  0.185  0.045  A&  r X2.X6.Mn0.G (inv2)      X2.lat1n
(HSPICE simulation)
4.185  0.032         0.047  0.028      f  X11.Mp0.G (inv2)       q0
4.216  0.031         0.041  0.019      r  X4.Mn1.G (latch1)      q0n
4.288  0.062         0.086  0.045  L&  f  X4.X6.Mp0.G (inv2)     X4.lat1n
...
```

If the external simulator is the embedded FineSim tool, the comment line says (FSE simulation).

See Also

- Distributed Processing

Nonlinear Waveform Analysis

For transition waveforms that have significantly nonlinear behavior, you can increase the analysis accuracy with nonlinear waveform modeling using the `set_nonlinear_waveform` command.

During path tracing, NanoTime breaks up the design into discrete units and simulates those units to characterize their timing. By default, NanoTime models an input transition as a simple linear slope from the lower to the upper threshold voltage for a rising transition, or from the upper to the lower threshold for falling transition, extrapolated to the rail voltages. The transition time is the amount of time it takes for the voltage to change from one threshold to the other.

By default, the lower thresholds are set to 10 and the upper thresholds are set to 90, so NanoTime models a rising transition as a straight line from 10 percent to 90 percent of the rail voltage. When transitions are significantly nonlinear, you can sometimes get better accuracy by using threshold voltages that are farther from the rail voltages. See Figure 10-20 for an example.

Figure 10-20 Linear Approximation of Transition Waveform

The thresholds used for analysis can be set for specific objects by using the `set_measurement_threshold` command, or for the whole design by setting the `rc_slew_*_threshold_pct_**` variables, where * represents `lower` or `upper` and ** represents `fall` or `rise`.

A linear slope approximation typically provides good tradeoff between accuracy and runtime. However, where input transition waveforms differ significantly from the linear model, you can get better accuracy, at the cost of memory and runtime, by using a nonlinear waveform for simulation. A nonlinear waveform uses a number of voltage-time data points rather than just two data points at the two threshold voltages, as shown in Figure 10-21.

Figure 10-21 Nonlinear Modeling of Transition Waveform

NanoTime can determine the smoothness of each calculated waveform. If the waveform is relatively smooth, it converts the waveform to an equivalent linear transition for simulating the next stage. On the other hand, if a waveform is relatively nonlinear or bumpy, it can optionally use the piecewise linear (nonlinear) model for better accuracy.

NanoTime checks the smoothness by comparing the slopes of adjacent piecewise linear segments of the waveform, based on the delay measurement point and the slew measurement points. If the adjacent slopes are nearly the same, the waveform is considered smooth. If the slopes are sufficiently different, the waveform is considered nonlinear.

To perform this test, NanoTime considers the slopes of adjacent segments, A and B. It looks at the larger slope ratio, either A/B or its inverse B/A, and compares this ratio to the threshold value (5.0 by default). A ratio smaller than this threshold indicates a smooth waveform; a larger ratio indicates a nonlinear waveform.

A glitch in the waveform can cause the ratio A/B to have a negative value. In that case, the waveform is considered nonlinear.

The `set_nonlinear_waveform` command specifies the type of transition waveform modeling to use at specified nets in the design.

The `-mode` option specifies one of four waveform handling modes: `fast`, `detect_only`, `efficient`, or `accurate` (the default):

- The `fast` mode uses the simple linear model unconditionally. This mode is the fastest, but it can be inaccurate where the actual transition waveforms are significantly nonlinear.
- The `detect_only` mode also uses the simple linear model. However, NanoTime smoothes the waveform and marks the nets where the slope ratio exceeds the threshold. The nets that have been marked can be found by using the `report_net` command or by checking the `is_nonlinear` net attribute.
- The `efficient` mode smoothes the waveform and retains the nonlinear model where the slope ratio threshold is exceeded, and uses the linear waveform otherwise. This mode is recommended for initial analysis of new designs.
- The `accurate` mode saves the original transition waveform at the net, ensuring the highest possible accuracy, at the cost of more runtime. The `-threshold` option has no effect for this mode.

The comparison threshold can be set with the `-threshold` option to any value from 1.0 to 10000. The lower the threshold, the more likely it is that a waveform is considered nonlinear. A value of 1.0 effectively forces all waveforms to be considered nonlinear. A large value like 1000 causes most waveforms to be considered smooth. In the absence of this option, the threshold is determined by the `nonlinear_waveform_detection_ratio` variable, which is set to 5.0 by default.

When NanoTime saves the nonlinear transition waveform, it uses the number of data points specified by the `-samples` option of the `set_nonlinear_waveform` command. If the `-samples` option is not used in the command, the number of samples is determined by the `nonlinear_waveform_samples` variable, which is set to 10 by default. More samples give better accuracy at the cost of memory and runtime.

The transition simulation mode information is stored in three attributes associated with the net: `nonlinear_mode`, `has_waveform`, and `is_nonlinear`. The `nonlinear_mode` attribute is a string indicating the transition simulation mode for the net: `fast`, `detect_only`, `efficient`, or `accurate`. The `has_waveform` attribute is `true` when the nonlinear waveform data points are saved for the net, which happens during path tracing when the specified conditions are met. The `is_nonlinear` attribute is set to `true` when NanoTime determines that the difference between the original and linear waveforms exceeds the threshold. This also happens during path tracing.

You can get information about nets by checking the attributes with the `get_attribute` command, or you can use the `report_net` command. In a report generated by the `report_net` command, the letter *w* in the Attrs column indicates that the `has_waveform` attribute is `true`, and the letter *n* indicates that the `is_nonlinear` attribute is `true`.

Extended Sidebranch Analysis

By default, NanoTime prunes simulation units with a large number of inputs to retain only those devices that form a chain of switching devices to the supply or ground rails. This reduces runtime but might lead to inaccuracy for some types of input stages such as multiple-input multiplexers, pass-gate carry chains, and domino precharge structures with several parallel pulldown stacks and multiple inputs in each stack.

You can specify the effort level (amount of analysis) for side branch exploration. The `timing_extended_sidebranch_analysis_level` variable sets the default effort level for all structures, while the `set_extended_sidebranch_level` command sets the effort level on specific nets. Net-specific settings override the global setting.

A level of 0 is the default for both the variable and the command. Larger effort levels increase the scope of sidebranch exploration. You cannot control or predict which side inputs are chosen for the analysis.

An effort level of 3 allows unlimited sidebranch exploration, which might lead to a significant increase in runtime and memory.

For example, Figure 10-22 shows a hypothetical circuit in which all of the transistors are part of the same channel-connected block. When input A is the trigger, the seven inputs B through H are side inputs. The default effort level of 0 is sufficient to take all of these side inputs into account.

Figure 10-22 Circuit With Multiple Side Inputs

However, some types of circuits can have a larger number of side inputs, such as the carry chain circuit in Figure 10-23. If input A is the trigger, inputs B through T are side inputs—19 of them in the same channel-connected block. A sidebranch exploration effort level of 1 is necessary to analyze the circuit as shown. In fact, the core unit of the carry chain might be

replicated many more times, resulting in an even larger number of side inputs for this circuit. You must consider whether analyzing all of the side inputs offers enough benefit to outweigh the cost of additional runtime.

Figure 10-23 Carry Chain Circuit

After path tracing is complete, you can use the `write_spice` command to write a SPICE input file for a specific timing path to see which transistors are included in the simulation.

Predriver Mix Ratio

The `sim_snps_predriver_mix_ratio` variable defines the input waveforms of a predriver. A value of 0 produces an exponential waveform, while a value of 1 produces a linear ramp. The default of 0.5 generates an intermediate shape. The mix ratio applies to both rising and falling edges.

To use this predriver, set the `sim_use_snps_predriver` variable to `true` (the default is `false`).

The predriver is a fast and simple model that provides a way of modifying the waveform propagated through a channel-connected block to more accurately reflect the waveform shape. The mix ratio depends on the driver circuit outside the block, the length of the timing path, the edge rate, the voltage at the input, and other connected devices. You must determine an appropriate mix ratio for your design by comparing the waveform generated by a NanoTime SPICE deck to an actual waveform from the netlist SPICE simulation. A mix ratio of 0.5 is suitable for most conditions.

Load and Miller Capacitance Settings

Miller capacitance arises from the capacitive coupling of a transistor gate to its own drain, which can have a large effect on delay calculations. For example, in the inverter in Figure 10-24(a), the Miller capacitance C couples a portion of the output transient back to the input, which affects the input waveform. By default, NanoTime models Miller capacitance as grounded capacitances on the input and output, as shown in Figure 10-24(b).

Figure 10-24 Miller Effect for an Inverter

You can disable Miller analysis by setting the `sim_cfg_miller_effect` variable to `false` (the default is `true`). However, for best accuracy, keep this variable set to the default.

If Miller capacitance analysis is enabled, a SPICE deck generated with the `write_spice` command includes the additional active devices needed to replicate the NanoTime analysis.

You can control the nature and extent of Miller capacitance analysis by using the following variables and options. Due to the effect on runtime, you might want to use these settings only for signoff runs.

The sim_miller_use_active_load Variable

By default, NanoTime models side load or fanout transistors as constant capacitive loads. To represent the nonlinear nature of the transistor capacitances accurately, set the `sim_miller_use_active_load` variable to `true` (the default is `false`) to model a side load circuit as an inverter with a capacitive load Cm, as shown in Figure 10-25. NanoTime calculates capacitance Cm individually for each side load circuit.

Figure 10-25 Miller Effect for Side Loads

The active load output is set to switch in the opposite direction of the trigger net to achieve a worst-case condition.

The `sim_miller_use_active_load` variable applies only to inverting fanout loads because they have the largest effect on maximum delay paths. To enable Miller effect analysis on noninverting fanout loads, set the `sim_miller_use_active_load_min` variable to `true` (the default is `false`).

The sim_miller_use_extended_load Variable

By default, in a series configuration of transistors, only the switching device is modeled using active Miller analysis. This is true for both the path being traced and for side load circuits. For example, in Figure 10-26(a), only transistor Mn1 is modeled using active Miller analysis.

If you set the `sim_miller_use_extended_load` variable to `true`, NanoTime expands active Miller analysis by modeling one additional transistor in the on-chain with a transistor model. Model replacement occurs only for the transistor adjacent to the active-load (switching) transistor in the direction of signal flow, for example, transistor Mn2 in Figure 10-26(a).

Additional transistor modeling does not occur if either of the following is true:

- Setting the additional transistor to an on state results in a conflict with other connections to that transistor.
- There are no adjacent transistors on the downstream side (in the direction of signal flow) of the active-load transistor. In Figure 10-26(b), the switching device is transistor Mn3, with transistors Mn1 and Mn2 in a fixed state. In this case, setting the

`sim_miller_use_extended_load` variable to `true` does not add any transistor models to the simulation.

Figure 10-26 Effect of the sim_miller_use_extended_load Variable Set To true

The Direction Check Variable and Option

Applying active Miller loads might be too pessimistic for some situations. The default assumption is that the output signal switches in the opposite direction as the trigger input, which is a worst-case condition. However, in designs such as a multiplexer fanout, the output signal might switch in the same direction as the trigger net.

To check the direction of switching on the next downstream net to determine if it is the same as or different from the trigger net, set the `sim_miller_direction_check` variable to `true`. The tool then sets the initial conditions appropriately. This variable applies only to nets that drive transmission gates that are part of multiplexers. For other transmission gates, use the `-check_miller_direction` option with the `mark_net` command.

The Fanout Limit Variables

When the `sim_miller_use_active_load` variable is set to `true`, all fanout transistors are modeled as active load devices. To limit the modeling effort and improve runtime, set the `sim_miller_active_load_enable_fanout_limit` variable to `true` (the default is `false`).

If fanout limiting is enabled, the maximum number of active loads is the number specified by the `sim_miller_active_load_fanout_limit` variable. The default is 10.

The Transistor Side Load Variable and Option

By default, the NanoTime tool models nonswitching transistors as linear capacitances. You can improve accuracy at the cost of runtime by modeling these loads as transistors, which includes all of the nonlinear capacitance behavior inherent in transistor models. You can specify the model globally or locally, as follows:

- Global specification

  Set the `sim_side_transistor_pin_load_model` variable to `transistor` to consider all load transistors in the design. To save runtime, NanoTime selectively applies the full-transistor model only to those devices in the design that exhibit the greatest delay accuracy benefit. However, setting the variable to `transistor` might significantly increase runtime.

- Local specification

  Use the `mark_net` command with the `-transistor_pin_load_model` option to specify the load model for listed nets. Valid values are `transistor` and `capacitor` (the default).

Figure 10-27 shows a NAND gate in which input signal B is rising and controlling the switching behavior. Input signal A is fixed at logic high; therefore, the PMOS transistor at input A is turned off. By default, the `sim_side_transistor_pin_load_model` variable is set to `capacitor` and NanoTime models the side load as a linear capacitance.

If you set the `sim_side_transistor_pin_load_model` variable to `transistor`, NanoTime models the load as a transistor.

Figure 10-27 Effect of the sim_side_transistor_pin_load_model Variable

The si_move_miller_caps_into_fets Variable

The `si_move_miller_caps_into_fets` variable changes the way that transistor capacitances are analyzed during signal integrity analysis. For backward compatibility, the default is `false`. However, for best accuracy, set the variable to `true`.

This variable has an effect only if signal integrity analysis is enabled by setting the `si_enable_analysis` variable to `true`. For the best accuracy, you must also set the `sim_miller_use_active_load` variable to `true`.

By default, the NanoTime tool accounts for the timing effect of coupling capacitors by treating them as signal integrity aggressors. This results in some amount of pessimism because signal integrity analysis is a pessimistic bounding analysis.

When a coupling capacitor connects a pair of logical nets that connect to pins of the same transistor (such as the gate and drain pins), you can move the effect of the capacitor into the simulation of the transistor by setting the `si_move_miller_caps_into_fets` variable to `true`. In this case, the capacitance effect is incorporated into every simulation that involves the transistor, which improves the accuracy of the waveforms at the transistor pins.

Figure 10-28 illustrates the effect. The dashed lines indicate the extent of the transistor model, which includes a set of charge-voltage relationships that are used whenever the device appears in the circuit.

Figure 10-28 Effect of the si_move_miller_caps_into_fets Variable

Miller Analysis for Circuits With Complementary Signals

When Miller loads are applied to circuits that use complementary signals, rail-to-rail shorts might occur. To evaluate the possibility of short circuits among load transistors, set the `sim_miller_rail_short_detection` variable to `true`. The default is `false` to avoid the additional processing for low-risk circuits. However, for differential circuits or any other design that uses complementary signals, set this variable to `true`.

Controlling Runtime

The path tracing operation is the most time-consuming step in the NanoTime flow. During path tracing, the tool breaks the design in discrete units, analyzes the timing behavior of the individual units, and combines the units to determine the delays along specific paths. The paths with the most critical timing are saved into the results database.

Many factors affect the overall runtime for a specific design. The usage of NanoTime commands and options has a direct effect on runtime. In addition, certain design styles can require longer runtime.

Runtime and accuracy often present tradeoff decisions. You must consider your requirements for both runtime and accuracy when setting up the analysis.

This section discusses techniques that can help control runtime:

- Identifying Parallel Stacks
- Blocking Path Tracing Through Redundant On-Chains
- Handling Reconvergent or Duplicated Logic

See Also

- Controlling Accuracy

Identifying Parallel Stacks

By default, NanoTime identifies parallel on-chain transistor stacks at the `match_topology` command and performs path tracing through only one of the parallel structures to save runtime. Transistor stacks are considered to be parallel if each stack contains the same number, type, and arrangement of transistors as well as the same boundary nets.

Figure 10-29 shows a circuit with three parallel NMOS transistor stacks. By default, NanoTime performs path tracing through only one of the stacks. However, if you set the `topo_find_parallel_stack` variable to `false` (the default is `true`), the tool performs path tracing through all three stacks, which might increase the runtime.

Figure 10-29 Parallel Stacks

Blocking Path Tracing Through Redundant On-Chains

Figure 10-30 shows a simple structure that contains duplicated structures, in this case a stack of two NMOS transistors. All of the devices, including the PMOS and NMOS transistors, are part of the same channel-connected block.

Figure 10-30 Circuit With Always-On Devices

During path tracing, NanoTime searches for paths through channel-connected blocks by finding on-chains (devices in series that are connected to ground or Vdd) that drive an output signal low or high. By default, the tool traces paths through every transistor in an on-chain and through every on-chain at a CCB output.

Searching through duplicate on-chains adds runtime without providing much benefit because parallel structures typically have similar delays.

You can use the `-dont_search_thru_channel` option of the `mark_instance` command to block path searching through redundant on-chain arcs. This strategy is useful when you know that certain on-chains are not relevant to the overall circuit timing. For example, the logic state of an on-chain might be fixed if one or more transistor inputs are controlled by `set_case_analysis` commands.

Figure 10-31 shows the result of using the following command on the example circuit:

```
mark_instance -dont_search_thru_channel X1
```

As a result, the timing arcs marked in green in Figure 10-31 are not included in path searching, reducing runtime.

Figure 10-31 Manually Blocking Path Tracing

Handling Reconvergent or Duplicated Logic

NanoTime performs depth-first static timing analysis. This means that as the tool traces a path, it compares the delay to a pin to delays previously found on other paths through that pin.

- If the delay on the current path is worse than the delays on previously found paths, the new delay is stored for reference and path searching continues forward. A worse delay is a longer delay for maximum delay analysis and a shorter delay for minimum delay analysis.

- If the delay on the current path is better than the delays on previously found paths, path tracing is not continued forward. In other words, those paths are “pruned” from the search, thereby saving runtime.

By default, NanoTime saves only the worst-case path from each startpoint to each endpoint. You can choose to save more paths at a cost of increased runtime and memory usage. The `-keep_paths_within` option of the `trace_paths` command allows you to specify a time value (relative to the worst path) within which all paths are kept.

For example, if you use the following command, NanoTime keeps all paths that have a slack within 0.2 time units of the worst path for each startpoint-endpoint pair:

```
trace_paths -keep_paths_within 0.2
```

Using the `-keep_paths_within` option with either the `trace_paths` or `extract_model` commands might result in extremely long runtime. Use one or both of the following strategies to avoid excessively long runtime:

- Use very small time values with the `-keep_paths_within` option of the `trace_paths` and `extract_model` commands. The default is 0. A time value equal to a clock cycle is the equivalent of performing no path pruning at all, which is practical only for very small designs.
- Mark nets with the `mark_net -ignore_keep_within` command. This option applies the standard path pruning operation to the marked nets regardless of the options used with the `trace_paths` and `extract_model` commands.

Knowledge of your design is very important. For example, Figure 10-32 shows a reconvergent circuit that has many possible paths from the inputs to the output. Standard path pruning keeps only the most critical path through the circuit. If you use the `-keep_paths_within` option, the tool keeps more paths. In most cases, the delays through the alternate paths tend to be different enough from each other that the increase in runtime is manageable, depending on the time value and the nature of the design.

Figure 10-32 Reconvergent Logic

However, Figure 10-33 shows a type of delay line circuit that poses a bigger problem. This chain of inverters includes parallel pullup and pulldown devices in every stage. The delay through a single stage is controlled by the number of pullup or pulldown devices that are turned on (larger on-current leads to faster transition time).

By default, NanoTime creates a timing arc through every one of the parallel transistors, resulting in many possible combinations for a timing path from input to output. The additional runtime required to analyze these paths provides little or no accuracy benefit, because the delays through the parallel paths are likely to be very nearly the same.

Figure 10-33 Duplicated Logic

It is impossible to find a time value for the `-keep_paths_within` option that keeps the total number of paths to a reasonable number. Use the `-ignore_keep_within` option of the `mark_net` command to mark all of the output nets of the inverter stages. In this case, NanoTime analyzes the most critical path through the chain and prunes all other paths regardless of the options used with the `trace_paths` and `extract_model` commands.

11

Correlation with HSPICE Analysis

NanoTime can create input files for use with the HSPICE dynamic simulator as a benchmark for NanoTime delays.

The `write_spice` command is described in the following sections:

- Overview of the write_spice Command
- The write_spice Output File
- Advanced SPICE Deck Features
- Including Transistor Models in a SPICE Deck
- Adding Optional Measure Statements to a SPICE Deck
- Effect of Signal Integrity Analysis on a SPICE Deck
- SPICE Decks for Silicon-on-Insulator Transistors
- Using SPICE Decks to Debug Dynamic Clock Simulation Regions
- Simulating SPICE Decks Created With the write_spice Command

Overview of the write_spice Command

The `write_spice` command creates a SPICE netlist, also called a SPICE deck, for a path or network of interest. You can generate different types of SPICE decks for the following purposes:

- To run dynamic simulation on a timing path or arc to obtain timing delays and transition times for comparison to the delays computed by the NanoTime tool
- To run dynamic simulation on a collection of nets for noise analysis
- To assist in debugging a dynamic clock simulation region

Using the `write_spice` command is the most accurate method of creating input files for dynamic simulation because it includes all of the relevant side inputs, initial conditions, and parasitics.

When creating an input deck for dynamic simulation tools, you must execute the `write_spice` command after path tracing is complete (in other words, after executing the `trace_paths`, `extract_model`, or `characterize_context` commands). For debugging dynamic clock simulation regions, you can use the `write_spice` command any time after the `check_design` command.

The following is a simple example of the `write_spice` command:

```
nt_shell> write_spice -output out.sp [get_timing_paths -max -max_paths 1]
```

This command generates the SPICE deck for the most critical maximum path to the file out.sp. If multiple paths result from the `get_timing_paths` command, a number is appended to the output file name. For example, if the command is specified with the `-max_paths 3` option, three output SPICE decks are created: out.sp, out_00001.sp, and out_00002.sp.

You can embed the `get_timing_paths` command in the `write_spice` command argument to specify paths using the path ID:

```
nt_shell> write_spice -output out.sp [get_timing_paths -path_id 18394]
```

Use the `-rise`, `-fall`, `-from`, `-through`, and `-to` options of the `get_timing_paths` command to be more specific about paths of interest. For example,

```
nt_shell> write_spice -output out.sp [get_timing_paths -max \
          rise_from clk1 -fall_through X1.lat -to out1]
```

Problematic Paths for SPICE Analysis

There are paths for which the `write_spice` command cannot return a usable SPICE deck. These are paths that run through .lib models, through clocks when dynamic clock simulation is used, and through some dynamic delay simulation blocks.

- If the specified path includes a .lib model, the resulting SPICE deck includes a black box in the path. The HSPICE simulation stops at the input to the model, and the measure statement from input to output does not return a value. You can edit the SPICE deck to perform an HSPICE simulation from the model output to the path output. You can also add additional measurements to obtain the individual path segments.
- If dynamic clock simulation (DCS) is used, the NanoTime tool has no knowledge of the path traversed from the input pin to a clock endpoint. In a SPICE deck for a path that includes a DCS region, the clock path begins at the clock endpoint, using the DCS delay analyzed by the NanoTime tool.

  The `-dcs` option of the `write_spice` command creates a SPICE deck for debugging the topology of a DCS region. However, the SPICE deck is not meant for simulating delays through the DCS region.

Dynamic simulation within the NanoTime tool correctly simulates the interdependencies of signal arrivals. However, the HSPICE tool cannot handle negative timing offsets in a SPICE deck. Consequently, offsets that are negative in reality must be set to zero in the SPICE deck. In this case, the NanoTime tool issues a CMDS-121 warning message to inform you that negative offset vector delays for a DDS stage are either converted to zero or modified to post-switching values. To improve accuracy, you can manually edit the VCVS statements in the SPICE deck to modify the waveform.

Using SPICE Analysis for Debugging

The `write_spice` command can help you to debug setup problems. For example, if a `one_hot` definition is missing from a multiplier stage, the SPICE deck would show that all of the multiplier select signals are on, which causes all of the other multiplier inputs to be included as side loads.

If you have an accuracy issue on a stage, check these items:

- Are the side input voltages correct?
- Are the initial conditions as expected?
- Are enough transistors included in the stage to give the correct simulation?

Possible actions include fixing topology markings, adding logic constraints, or specifying a dynamic delay simulation region to combine multiple stages.

The write_spice Output File

The circuit in Figure 11-1 is used to illustrate a SPICE deck created by the `write_spice` command. The file is ready to run, except for the transistor models.

Figure 11-1 Three-Inverter Circuit

The SPICE deck begins as follows:

```
* Version M-2016.12-SP2 -- linux64
* Date   : Fri Feb 24 13:46:50 2017
.option lis_new=1 statfl=1 post=0 runlvl=6 autostop
.save level=none
* Begin user SPICE header.
*
...
* End user SPICE header.
```

The first two lines of the SPICE deck show the NanoTime version and date.

The `.option` and `.save` statements improve HSPICE simulation time. Modify these lines as needed for your computing environment. You can override the HSPICE options in the user-defined header file, with the exception of the `autostop` option.

If you include a header file by using the `-header` option of the `write_spice` command, the lines of the header file are written in the SPICE deck as comment lines.

The next section of the SPICE deck is a series of comment lines that contain the results of an associated reporting command. The report comes from the `report_paths`, `report_noise`, or `report_fanout_noise` command, as appropriate for the analysis.

The SPICE deck for the delay analysis of the circuit in Figure 11-1 begins with the path report, as follows:

```
* PATH 2
*
*
* Startpoint:   IN (in port)
* Endpoint:     OUT (out port)
* Path Type:    max

* Constraint:   set_output_delay check
*
*  Path  Incr Adjust  Trans   Cap NT  Point                Net
* ---------------------------------------------------------------
*             0.0000             clock CLK (rise)
*             0.3000             input external delay
* 0.3000      0.0000 0.0800 0.0236 D& f IN (in)            IN
* 0.3039 0.0039      0.0804 0.0236 &  f X1.Mp0.g (inv2)    IN
* 0.3599 0.0560      0.0859 0.0446 &  r X2.Mn0.g (inv2)    INB
* 0.4092 0.0493      0.0807 0.0478 &  f X3.Mp0.g (inv2)    OUTB
* 0.4376 0.0284      0.0350 0.0136 O& r OUT (out)          OUT
* 0.4376                       data arrival time name latch
...
```

Arc0 Subcircuit

The SPICE circuit for this path is shown as individual subcircuits for each stage of the path, corresponding to the stages shown in the path report: a subcircuit from IN to IN, a subcircuit from IN to INB, and so on. Sometimes two stages are combined into a single stage in the SPICE deck; this can occur with feedback or other situations that require multiple stages to be simulated together.

The netlist of the Arc0 subcircuit, which consists of the first stage from port IN to the transistor pins, appears as follows:

```
.SUBCKT Path2_Arc0 IN@n1 IN@n3 IN@n2
* IN@n1 input conn_to_trigger
* IN@n3 output
* IN@n2 output conn_to_trigger
* Port IN IN@n1 PIN_IN
CIN@c1 IN@n1 0 0.286097FF
CIN@c2 IN@n2 0 4.37358FF
CIN@c3 IN@n3 0 2.64932FF
RIN@r1 IN@n1 IN@n2 161.021
RIN@r2 IN@n2 IN@n3 404.325
.IC V(IN@n1)=1.2
.IC V(IN@n2)=1.2
.ENDS Path2_Arc0
```

This stage is composed solely of parasitics, connecting the port IN@n1 to outputs IN@n2 and IN@n3. (NanoTime names nets within a parasitic group with netname@nn. The port name being used is IN@n1 because of RC reduction.) The comments indicate that IN@n1 is the trigger input, and that the IN@n3 output is the trigger to the next stage. As the path from IN falls, net IN@n3 is the output that drives the PMOS transistor in the next stage. Initial condition statements are listed at the end of the subcircuit statement to set the output nets to 1.2 V (VDD) as the input rises from high to low.

The parasitics included in this netlist are the result of the NanoTime RC reduction operation. To preserve all of the original parasitics and include them in the SPICE deck, disable RC reduction by setting the value of the `rc_reduction_max_net_delta_delay` variable to 0.

Arc1 Subcircuit

The second subcircuit in this path, Arc1, exists between IN and INB and includes the parasitics on INB. The netlist is as follows:

```
.SUBCKT Path2_Arc1 IN@n3 IN@n2 INB@n1 INB@n2
* IN@n3 input
* IN@n2 input conn_to_trigger
* INB@n1 output
* INB@n2 output conn_to_trigger
MNX1_Mn0 INB@n3 IN@n3 Path2_Arc1_gnd Path2_Arc1_gnd nch W=4u L=0.13u
AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
MPX1_Mp0 Path2_Arc1_vdd IN@n2 INB@n4 Path2_Arc1_vdd pch W=6u L=0.13u
AS=1.62p
AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
CINB@c1 INB@n1 0 4.57716FF
CINB@c2 INB@n2 0 8.79222FF
CINB@c3 INB@n3 0 3.0413FF
CINB@c4 INB@n4 0 0.502325FF
RINB@r1 INB@n1 INB@n2 226.034
RINB@r2 INB@n2 INB@n3 395.338
RINB@r3 INB@n3 INB@n4 113.455
.IC V(IN@n2)=1.2
.IC V(INB@n2)=0
.IC V(INB@n3)=0
.IC V(INB@n4)=0
V1 Path2_Arc1_gnd 0 DC=0
V2 Path2_Arc1_vdd 0 DC=1.2
.ENDS Path2_Arc1
```

This subcircuit includes MNX1 and MPX1, two transistors that form the inverter, along with the parasitics and initial conditions. Notice that, in addition to the trigger input IN@n2, IN@n3 is also included as a subcircuit pin for the complete simulation of the inverter. The

last section in this subcircuit netlist contains the voltage settings for VDD and GND. They are defined separately in each stage of the output as follows:

```
V1 Path2_Arc1_gnd 0 DC=0
V2 Path2_Arc1_vdd 0 DC=1.2
```

Top-Level Netlist

The top-level netlist, shown in the following lines, instantiates the subcircuits:

```
X0 IN@n1 IN@n3 IN@n2 Path2_Arc0
X1 IN@n3 IN@n2 INB@n1 INB@n2 Path2_Arc1
X2 INB@n1 INB@n2 OUTB@n4 OUTB@n3 Path2_Arc2
X3 OUT@n1 OUTB@n4 OUTB@n3 Path2_Arc3
```

Control and Measure Statements

The final portion of the netlist contains control and measure statements, as follows:

```
.TEMP 85
* input slope is 0.08
VIN IN@n1 0 PWL(0ns 1.2 0.1ns 0)
* output slope is 0.0350145
* Transient analysis
.TRAN 0.1PS 0.325212NS
* Look at the input and output
.PRINT TRAN V(IN@n1) V(OUT@n1)
* Path measurements.
.MEASURE TRAN path_delay TRIG v(IN@n1) VAL=0.6 CROSS=1 TARG v(OUT@n1)
VAL=0.6 CROSS=1
.MEASURE TRAN path_input_transition TRIG v(IN@n1) VAL=1.08 CROSS=1 TARG
v(IN@n1) VAL=0.12 CROSS=1
.MEASURE TRAN path_output_transition TRIG v(OUT@n1) VAL=0.12 CROSS=1 TARG
v(OUT@n1) VAL=1.08 CROSS=1
.END
```

The components of this section are as follows:

- The input stimulus follows the `.TEMP` statement. The input slope is based on the upper and lower RC slew thresholds and the fall and rise values. The `PWL` statement indicates a piecewise linear input that uses the slope extrapolated to a 100 percent transition. In this case, the 0.08-ns slope is specified with the default 10 to 90 percent slew thresholds, which extrapolates to a 0 to 100 percent slew of 1.2 ns.
- The `.TRAN` statement provides enough margin before the end time of the simulation to allow the IN to OUT delay to complete its transition.
- The `.PRINT` and `.MEASURE` statements reflect the three measurements that are made by default: the path delay between the midpoints of the input and output, the transition time

of the input signal, and the transition time of the output signal. For the path delay measurement, the VAL settings are one half of the VDD value, or 0.6 V. For the transition measurements, the VAL settings are the upper and lower RC slew thresholds and the fall and rise values used in the run. In this example, the values are 10 percent to 90 percent, or 0.12 V to 1.08 V.

See Also

- Including Transistor Models in a SPICE Deck
- Advanced SPICE Deck Features

Advanced SPICE Deck Features

The circuit in Figure 11-2 is used to demonstrate additional options of the `write_spice` command.

Figure 11-2 Circuit Example

The following command generates the output SPICE deck:

```
nt_shell> write_spice -output comp.sp [get_timing_paths -max \
          -rise_from A -fall_through selb -to OUT]
```

Active and Inactive Transistors

The following portion of the SPICE deck describes the Path1_Arc1 subcircuit, which consists of the path from A to int0 across NAND gate X1:

```
.SUBCKT Path1_Arc1 A int0
* A input conn_to_trigger
* int0 output conn_to_trigger

MNX1_Mn0 X1_n1 A Path1_Arc1_gnd Path1_Arc1_gnd nch W=4u L=0.13u AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.091 NRD=0.157
MNX1_Mn1 X1_n2 IN1 X1_n1 Path1_Arc1_gnd nch W=4u L=0.13u AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.091 NRD=0.157
MNX1_Mn2 int0 IN2 X1_n2 Path1_Arc1_gnd nch W=4u L=0.13u AS=1.08p AD=1.08p
PS=8.54u PD=8.54u NRS=0.091 NRD=0.157
MPX1_Mp0 Path1_Arc1_vdd A int0 Path1_Arc1_vdd pch W=6u L=0.13u AS=1.62p
AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045

* Off transistors
MPX1_Mp1 Path1_Arc1_vdd Path1_Arc1_vdd int0 Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
MPX1_Mp2 Path1_Arc1_vdd Path1_Arc1_vdd int0 Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
MNX5_Mn0 0 int0 0 Path1_Arc1_gnd nch W=2u L=0.13u AS=0.54p AD=0.54p
PS=4.54u PD=4.54u NRS=0.091 NRD=0.157
CX5_Mn0_miller_load int0 0 0.366441FF
MPX5_Mp0 Path1_Arc1_vdd int0 Path1_Arc1_vdd Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
CX5_Mp0_miller_load int0 0 0.796727FF

.IC V(A)=0
V1 IN1 0 DC=1.2
V2 IN2 0 DC=1.2
.IC V(X1_n1)=1.08929
.IC V(X1_n2)=1.08929
V3 Path1_Arc1_gnd 0 DC=0
.IC V(int0)=1.2
V4 Path1_Arc1_vdd 0 DC=1.2
.ENDS Path1_Arc1
```

Unlike the simple three-inverter circuit in Figure 11-1 where all transistors were included, this subcircuit includes only active transistors and any side transistors whose load affects the path delay. For this three-input NAND, all NMOS transistors are included, with the other inputs IN1 and IN2 set to the VDD of 1.2 to allow the rising signal A to cause the output to transition to zero.

Only the PMOS transistor driven by input A is included as an active device; the other two PMOS transistors are inactive and listed in the “Off transistors” section of the netlist:

```
* Off transistors
MPX1_Mp1 Path1_Arc1_vdd Path1_Arc1_vdd int0 Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
MPX1_Mp2 Path1_Arc1_vdd Path1_Arc1_vdd int0 Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
```

The gates of these inactive transistors are not driven by the actual inputs but by the VDD supply, as they do not switch or provide current in the simulation of this arc.

Two inactive transistors that are driven by int0 in NOR gate X5 are also included in this subcircuit:

```
MNX5_Mn0 0 int0 0 Path1_Arc1_gnd nch W=2u L=0.13u AS=0.54p AD=0.54p
PS=4.54u PD=4.54u NRS=0.091 NRD=0.157
CX5_Mn0_miller_load int0 0 0.366441FF

MPX5_Mp0 Path1_Arc1_vdd int0 Path1_Arc1_vdd Path1_Arc1_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
CX5_Mp0_miller_load int0 0 0.796727FF
```

These transistors are included as simple load devices with the drain and source of each connected to VDD or GND. An extra capacitor is included with each transistor; NanoTime adds this extra capacitor to account for the Miller effect on these transistors. The Miller effect is the effective change in capacitance between transistor terminals that occurs with a change in voltage across those terminals.

If the timing analysis is run using the active Miller mode, which is enabled by setting the `sim_miller_use_active_load` variable to `true`, these devices are included as active transistors with the following statements:

```
* element X5_Mn0 is load
MNX5_Mn0 X5_Zn int0 Path1_Arc1_gnd Path1_Arc1_gnd nch W=2u L=0.13u
AS=0.54p AD=0.54p PS=4.54u PD=4.54u NRS=0.091 NRD=0.157

* element X5_Mp0 is load
MPX5_Mp0 Path1_Arc1_vdd int0 X5_n1 Path1_Arc1_vdd pch W=6u L=0.13u
AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
```

These active devices do not have Miller capacitors added to their outputs; instead, the active transistors have the following loads:

```
.IC V(X5_Zn)=0
CX5_Zn X5_Zn 0 6.4FF
.IC V(X5_n1)=1.2
CX5_n1 X5_n1 0 6.4FF
```

Finally, in the voltage section of this arc, there are the initial condition settings for the internal nodes of the NAND gate:

```
.IC V(X1_n1)=1.08929
.IC V(X1_n2)=1.08929
```

These voltages are set to a logical 1, but cannot reach the full value of VDD because of the threshold drop of the NMOS transistors. You can change these calculated values to increase or decrease pessimism by using the `set_initial_condition_adjustment` command.

Voltage-Controlled Voltage Sources

The netlist of subcircuit Path1_Arc4 details the path from the selb net through the MUX2 multiplier to the X9_outn output, as shown in the following section of the SPICE deck:

```
.SUBCKT Path1_Arc4 selb X9_outn
* selb input conn_to_trigger
* X9_outn output conn_to_trigger
MPX7_Mp0 Path1_Arc4_vdd IN5 in5b Path1_Arc4_vdd pch W=6u L=0.13u AS=1.62p
AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
MNX8_Mn0 in6b IN6 Path1_Arc4_gnd Path1_Arc4_gnd nch W=4u L=0.13u AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
MNX9_Mn0 in5b sel X9_outn Path1_Arc4_gnd nch W=4u L=0.13u AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.091 NRD=0.157
MNX9_Mn1 in6b selb X9_outn Path1_Arc4_gnd nch W=4u L=0.13u AS=1.08p
AD=1.08p PS=8.54u PD=8.54u NRS=0.091 NRD=0.157
MPX9_Mp0 in5b selb X9_outn Path1_Arc4_vdd pch W=6u L=0.13u AS=1.62p
AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
MPX9_Mp1 in6b sel X9_outn Path1_Arc4_vdd pch W=6u L=0.13u AS=1.62p
AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
* Off transistors
MNX7_Mn0 in5b 0 0 Path1_Arc4_gnd nch W=4u L=0.13u AS=1.08p AD=1.08p
PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
MPX8_Mp0 Path1_Arc4_vdd Path1_Arc4_vdd in6b Path1_Arc4_vdd pch W=6u
L=0.13u AS=1.62p AD=1.62p PS=12.54u PD=12.54u NRS=0.069 NRD=0.045
V1 IN5 0 DC=0
V2 IN6 0 DC=1.2
.IC V(X9_outn)=0
V3 Path1_Arc4_gnd 0 DC=0
.IC V(in5b)=1.2
.IC V(in6b)=0
E4 sel 0 selb Path1_Arc4_vdd -1.0
.IC V(selb)=1.2
V5 Path1_Arc4_vdd 0 DC=1.2
.ENDS Path1_Arc4
```

The transistors are the four transistors in MUX2 and the two driving inverters from inputs IN5 and IN6. When the selb signal switches, the transmission gate requires the sel signal to

switch in the opposite direction. The SPICE deck accomplishes this by including a voltage-controlled voltage source:

```
E4 sel 0 selb Path1_Arc4_vdd -1.0
```

If the transmission gate inputs are separated by a simple inverter, the inverter is included in the SPICE deck and the voltage-controlled voltage source is not necessary. However, since the inversion is formed with more complex logic in this case, the voltage-controlled voltage source is needed. The transmission gate is recognized by NanoTime because the `topo_tgate_mark_all_pairs` variable is set to `true` by default.

See Also

- The write_spice Output File

Including Transistor Models in a SPICE Deck

The following methods are available for including transistor models in the SPICE deck:

- The Header File Option
- The Model Cards Option
- The Wrapper Subcircuit Option

The Header File Option

The `-header` option of the `write_spice` command specifies a file to be included at the head of the SPICE output deck. This file can contain the SPICE model that was used in the static timing analysis run that produced the SPICE deck.

The following example file, header.sp, contains a single line:

```
.lib './model.013' SS
```

The header file is specified with the `write_spice` command as follows:

```
nt_shell> write_spice -output invs.sp \
          -header header.sp [get_timing_paths -max -max_paths 1]
```

The resulting SPICE deck is named invs.sp and includes the following lines:

```
...
* Begin user SPICE header.
.lib './model.013' SS
* End user SPICE header.
* PATH 2
...
```

The Model Cards Option

Small geometry transistor models often enclose the transistor definition within a wrapper. In the netlist, these macro models appear as instances whose names begin with the letter X instead of as models denoted by the letter M. For example,

```
.subckt inv2 A Z
xMn0 Z A gnd gnd nch_mac W='dwn * 2' L=dl
xMp0 vdd A Z vdd pch_mac W='dwp * 2' L=dl
.ends
```

Each transistor can be annotated through a parasitic device parameter file (DPF) file with parameters such as SA, SA1, and SB. NanoTime creates a transistor model for each unique set of parameters it finds on a transistor. To generate a SPICE deck that includes these transistor model cards, use the `write_spice` command with the `-insert_model_cards` option:

```
nt_shell> write_spice -output invs.sp \
          -insert_model_cards [get_timing_paths -max -max_paths 1]
```

The resulting SPICE deck contains this model statement:

```
.MODEL nch_0__dmax.14 NMOS level=54 lmin=3.6e-08 lmax=4.3848e-08
+ wmin=5.4576e-07 wmax=9.0576e-07 version=4.5 binunit=2 paramchk=1
+ mobmod=0 capmod=2 igcmod=1 igbmod=1 diomod=1 rdsmod=0 rbodymod=0
```

The transistor itself is specified as follows:

```
MNxi1_xmn0_main n1 a[0] Path2_Arc1_gnd Path2_Arc1_gnd nch_0__dmax W=0.8u
L=0.04u AS=0.0675781p AD=0.0675781p PS=1.76942u PD=1.76942u NRS=0.053
NRD=0.053 SA=0.29u SB=0.29u SW=0.0844444u SD=0.0844444u
```

A very large SPICE deck is generated when the number of model cards is large. However, some cases require the use of the `-insert_model_cards` option instead of the `-use_wrapper_subckt` option:

- If parasitics are included in the macro model, the `-insert_model_cards` option must be used. In this case, the wrapper subcircuit parasitics flow should be enabled by setting the `link_enable_wrapper_subckt_parasitics` variable to `true`.
- If multicorner analysis is performed with separate minimum or maximum clock or data technology files, the `-insert_model_cards` option must be used to obtain the correct model data for each transistor in the path.

The Wrapper Subcircuit Option

If the transistor models are binary or encrypted, you must use the `-use_wrapper_subckt` option.

With this option, the transistors are listed as instances instead of transistors, so the normal call to the .lib file can be specified with the `-header` option, as shown in the following example:

```
nt_shell> write_spice -output invs.sp \
          -header header.sp -use_wrapper_subckt \
          [get_timing_paths -max -max_paths 1]
```

This command produces a SPICE deck with instances, which uses the basic wrapper model instead of transistors. A portion of this SPICE deck is as follows:

```
.SUBCKT Path2_Arc1 a[0] n1
* a[0] input conn_to_trigger
* n1 output conn_to_trigger
Xxi1_xmn0_main n1 a[0] Path2_Arc1_gnd Path2_Arc1_gnd nch_mac w=8e-07
l=4e-08
Xxi1_xmp0_main Path2_Arc1_vdd a[0] n1 Path2_Arc1_vdd pch_mac w=1.2e-06
l=4e-08
.IC V(a[0])=1.2
V1 Path2_Arc1_gnd 0 DC=0
.IC V(n1)=0
V2 Path2_Arc1_vdd 0 DC=1.2
.ENDS Path2_Arc1
```

Adding Optional Measure Statements to a SPICE Deck

NanoTime automatically includes measure statements for the input-to-output delay of the path and the transition time of the input and output signals. You can optionally include measure statements for the delays and transitions of the individual arcs of the path.

Use the `-measure_subckt_delays` option to add measure statements for delays between the input and output of each arc in the SPICE deck. For example, consider the following `write_spice` command:

```
nt_shell> write_spice -output invs.sp \
          -measure_subckt_delays [get_timing_paths -max -max_paths 1]
```

The resulting SPICE deck contains the following measure statements:

```
* Path measurements.
.MEASURE TRAN path_delay TRIG v(IN@n1) VAL=0.6 CROSS=1 TARG v(OUT@n1)
VAL=0.6
CROSS=1
.MEASURE TRAN path_input_transition TRIG v(IN@n1) VAL=1.08 CROSS=1 TARG
v(IN@n1) VAL=0.12 CROSS=1
.MEASURE TRAN path_output_transition TRIG v(OUT@n1) VAL=0.12 CROSS=1 TARG
v(OUT@n1) VAL=1.08 CROSS=1

* Subckt measurements.
.MEASURE TRAN Path2_Arc0_delay TRIG v(IN@n1) VAL=0.6 CROSS=1 TARG
v(IN@n2)
VAL=0.6 CROSS=1
.MEASURE TRAN Path2_Arc1_delay TRIG v(IN@n2) VAL=0.6 CROSS=1 TARG
v(INB@n2)
VAL=0.6 CROSS=1
.MEASURE TRAN Path2_Arc2_delay TRIG v(INB@n2) VAL=0.6 CROSS=1 TARG
v(OUTB@n3)
VAL=0.6 CROSS=1
.MEASURE TRAN Path2_Arc3_delay TRIG v(OUTB@n3) VAL=0.6 CROSS=1 TARG
v(OUT@n1)
VAL=0.6 CROSS=1
```

The individual subcircuit measurements include Path2_Arc0 from IN@n1 to IN@n2 and Path2_Arc1 from IN@n2 to INB@n2.

For measurements of the transition times of the individual arcs, use the following command:

```
nt_shell> write_spice -output invs.sp -measure_subckt_transitions \
          [get_timing_paths -max -max_paths 1]
```

The resulting SPICE deck contains the measure statements, as follows:

```
* Subckt measurements.
.MEASURE TRAN Path2_Arc0_transition TRIG v(IN@n2) VAL=1.08 CROSS=1 TARG
v(IN@n2) VAL=0.12 CROSS=1
.MEASURE TRAN Path2_Arc1_transition TRIG v(INB@n2) VAL=0.12 CROSS=1 TARG
v(INB@n2) VAL=1.08 CROSS=1
.MEASURE TRAN Path2_Arc2_transition TRIG v(OUTB@n3) VAL=1.08 CROSS=1 TARG
v(OUTB@n3) VAL=0.12 CROSS=1
.MEASURE TRAN Path2_Arc3_transition TRIG v(OUT@n1) VAL=0.12 CROSS=1 TARG
v(OUT@n1) VAL=1.08 CROSS=1
```

The transitions are measured on the arc outputs, IN@n2, INB@n2, OUTB@n3, and OUT@n1.

Effect of Signal Integrity Analysis on a SPICE Deck

This section covers the `write_spice` command options for use with signal integrity analysis for delay, noise, or fanout noise analysis.

- Delay Analysis
- Noise Analysis
- Fanout Noise Analysis

Delay Analysis

The circuit in Figure 11-3 is used as an example to create a SPICE deck for a NanoTime run with signal integrity delay analysis.

Figure 11-3 Signal Integrity Delay Circuit

This circuit has three sets of inverter pairs, with 5.1-fF capacitors coupled between data1n and data2n and between data1n and data3n. In the path from data1 to out1, the data1n net is a victim net while the data2n and data3n nets are aggressor nets. The relative input timing has been set such that only the data2n net is a valid aggressor.

If signal integrity analysis is enabled, the `write_spice` command generates a SPICE deck that includes the active aggressor inputs and capacitors by default. You can create a SPICE deck without the aggressor information by using the `-no_si_aggressors` option of the `write_spice` command.

The following command creates a SPICE deck that includes aggressor information for one of the paths in Figure 11-3:

```
nt_shell> write_spice -output si_delay \
          [get_timing_paths -max -from data1 -to out1]
```

The resulting SPICE deck lists the timing of each aggressor signal in two parameters, timing and slew, in the “Alignment for SUBCKT” section of the SPICE deck, as follows:

```
*Alignment for SUBCKT: Path2_Arc1
*victim net data1n aggressor net data2n
.PARAM alignment_Path2_Arc1_data1n_data2n_in_ns = 0.0757731ns
.PARAM slew_Path2_Arc1_data1n_data2n_in_ns = 0.0499405ns
```

The next section of the SPICE deck contains the netlist for the Path2_Arc1 subcircuit from data1 to data1n, as follows:

```
.SUBCKT Path2_Arc1 data1 data1n@n1
* data1 input conn_to_trigger
* data1n@n1 output conn_to_trigger
MNX1_Mn0 data1n@n1 data1 Path2_Arc1_gnd Path2_Arc1_gnd nch W=3u L=0.13u
AS=0.81p AD=0.81p PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
MPX1_Mp0 Path2_Arc1_vdd data1 data1n@n1 Path2_Arc1_vdd pch W=4u L=0.13u
AS=1.08p AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
* victim net data1n
Cdata1n@c1 data1n@n1 0 1.007FF
* victim net data1n aggressor net data2n
Cdata1n_data2n@c1 data1n_data2n@n1 0 1.007FF
* victim net data1n aggressor net data2n
Cdata1n_data2n@cc1 data1n@n1 data1n_data2n@n1 5.1FF
* victim net data1n aggressor net data2n
Vdata1n_data2n@v1 data1n_data2n@n1 0 PWL(0ns 1.2
slew_Path2_Arc1_data1n_data2n_in_ns 0 TD =
alignment_Path2_Arc1_data1n_data2n_in_ns )
* victim net data1n aggressor net data3n
Cdata1n_data3n@c1 data1n@n1 0 5.1FF
.IC V(data1)=1.2
.IC V(data1n@n1)=0
V1 Path2_Arc1_gnd 0 DC=0
V2 Path2_Arc1_vdd 0 DC=1.2
.ENDS Path2_Arc1
```

The capacitors are included with comment cards. Cdata1n@c1 and Cdata1n_data2n@c1 are the parasitic loading capacitors on the victim and aggressor nets. The coupling capacitor is Cdata1n_data2n@cc1. The aggressor input is defined by Vdata1n_data2n@v1 and uses the timing and slew parameters given in the “Alignment for SUBCKT” section.

Since the timing window of data3n does not overlap that of data1n, data3n is not an effective aggressor of data1n, and the coupling capacitor Cdata1n_data3n@c1 is included as a load capacitor to GND for data1n.

When using this SPICE deck to compare timing with HSPICE, you might need to adjust the alignment numbers. The HSPICE delay for the victim net input might be different from the NanoTime calculated delay, and the effect of the aggressor might be reduced unless the timing is adjusted to align with the victim.

Noise Analysis

If you perform signal integrity noise analysis, you can use the `-si_noise` option of the `write_spice` command to create SPICE decks for noise analysis. You can specify one or more of the following noise types in the argument list: `above_high`, `below_high`, `below_low`, or `above_low`.

Consider the following command for the circuit in Figure 11-3:

```
nt_shell> write_spice -output -si_noise above_high data1n
```

The output SPICE deck is as follows:

```
.option scale=1.0
.SUBCKT Noise_net_data1n_above_low_Arc0 data1
* data1 input conn_to_trigger
MNX1_Mn0 data1n@n1 data1 Noise_net_data1n_above_low_Arc0_gnd
Noise_net_data1n_above_low_Arc0_gnd nch W=3u L=0.13u AS=0.81p AD=0.81p
PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
MPX1_Mp0 Noise_net_data1n_above_low_Arc0_vdd data1 data1n@n1
Noise_net_data1n_above_low_Arc0_vdd pch W=4u L=0.13u AS=1.08p AD=1.08p
PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
* victim net data1n
Cdata1n@c1 data1n@n1 0 1.007FF
* victim net data1n aggressor net data2n
Vdata1n_data2n data1n_data2n 0 PWL(0ns 0 0.0499405n 1.2 TD = 0.00499405n)
* victim net data1n aggressor net data2n
Cdata1n_data2n@cc1 data1n@n1 data1n_data2n 5.1FF
* victim net data1n aggressor net data3n
Vdata1n_data3n data1n_data3n 0 PWL(0ns 0 0.0499405n 1.2 TD = 0.00499405n)
* victim net data1n aggressor net data3n
Cdata1n_data3n@cc1 data1n@n1 data1n_data3n 5.1FF
* Off transistors
MNX11_Mn0 0 data1n@n1 0 Noise_net_data1n_above_low_Arc0_gnd nch W=3u L=0.13u
AS=0.81p AD=0.81p PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
CX11_Mn0_miller_load data1n@n1 0 0.553879FF
MPX11_Mp0 Noise_net_data1n_above_low_Arc0_vdd data1n@n1
Noise_net_data1n_above_low_Arc0_vdd Noise_net_data1n_above_low_Arc0_vdd
pch W=4u L=0.13u AS=1.08p AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
CX11_Mp0_miller_load data1n@n1 0 0.530841FF
V1 data1 0 DC=1.2
V2 Noise_net_data1n_above_low_Arc0_gnd 0 DC=0
V3 Noise_net_data1n_above_low_Arc0_vdd 0 DC=1.2
.ENDS Noise_net_data1n_above_low_Arc0
X0 data1 Noise_net_data1n_above_low_Arc0
.TEMP 85
* Transient analysis
.TRAN 0.1PS 0.149822NS
* Noise measurements.
.MEAS TRAN X0.data1n@n1_max MAX V(X0.data1n@n1) FROM 0.0ns TO 0.149822ns
.MEAS TRAN X0.data1n@n1_noise_above_low PARAM='X0.data1n@n1_max'
.END
```

This SPICE deck contains a single subcircuit with the transistors connected to net data1n. The capacitors are labeled as comments: victim net, victim net, and aggressor net. Voltage waveforms are provided for the aggressor signals, for example the data2n aggressor:

```
* victim net data1n aggressor net data2n
Vdata1n_data2n data1n_data2n 0 PWL(0ns 0 0.0499405n 1.2 TD = 0.00499405n)
```

Because the input is held at a steady state, the piecewise linear timing points in the PWL argument do not need adjustment.

Two measure statements are included:

```
* Noise measurements.
.MEAS TRAN X0.data1n@n1_max MAX V(X0.data1n@n1) FROM 0.0ns TO 0.149822ns
.MEAS TRAN X0.data1n@n1_noise_above_low PARAM='X0.data1n@n1_max'
```

These measure statements specify the measurement of the maximum voltage height of the noise bump. The measurement is divided into two measure statements for the case when the noise bump is from a nonzero supply. If the `write_spice` command had included the `-si_noise below_high` option, the following measure statements would be included:

```
* Noise measurements.
.MEAS TRAN X0.data1n@n1_min MIN V(X0.data1n@n1) FROM 0.0ns TO 0.149822ns
.MEAS TRAN X0.data1n@n1_noise_below_high PARAM='1.2 - X0.data1n@n1_min'
```

The first measure statement finds the minimum value of the voltage on net data1n, and the second subtracts that from the 1.2-V VDD supply to get the height of the noise bump.

Use the `-si_noise_shape` option of the `write_spice` command to include measure statements for a transient simulation with a time span sufficiently long to capture the entire width of the simulated noise bump. These statements obtain values for both the width and time-to-peak parameters of the noise bump.

The SPICE deck for injected noise has a single stage. The stimuli switch the aggressors that are coupled to the victim net, and the trigger pin of the stage is quiet. This SPICE deck includes only the calculated injected noise and no user-defined injected noise.

Fanout Noise Analysis

The circuit shown in Figure 11-4 is used as an example for creating a SPICE deck for fanout noise analysis. In this circuit, the noise victim net is data1n.

Figure 11-4 Signal Integrity Fanout Circuit

When you perform signal integrity fanout noise analysis in addition to signal integrity noise analysis, you can specify the inclusion of the fanout net in the SPICE deck. To do this, use the `-si_noise` option and specify the original noise bump net in the `write_spice` command.

For example, to analyze the worst case noise bump on the noise victim net data1n, use the following command:

```
nt_shell> write_spice -output si_fanout -si_noise above_low \
                      -fanout data1n
```

The resulting SPICE deck has two subcircuits. The first subcircuit is for the data1n net as described previously in the initial signal integrity noise section. The second subcircuit is for the fanout net, out1n, which was selected instead of net out4n because it has a larger noise bump.

The SPICE deck is as follows:

```
.SUBCKT Noise_net_data1n_above_low_Arc1 data1n@n1
* data1n@n1 input conn_to_trigger
MNX11_Mn0 out1n data1n@n1 Noise_net_data1n_above_low_Arc1_gnd
Noise_net_data1n_above_low_Arc1_gnd nch W=3u L=0.13u AS=0.81p AD=0.81p
PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
MPX11_Mp0 Noise_net_data1n_above_low_Arc1_vdd data1n@n1 out1n
Noise_net_data1n_above_low_Arc1_vdd pch W=4u L=0.13u AS=1.08p AD=1.08p
PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
* Off transistors
MNX13_Mn0 0 out1n 0 Noise_net_data1n_above_low_Arc1_gnd nch W=3u L=0.13u
AS=0.81p AD=0.81p PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
CX13_Mn0_miller_load out1n 0 0.553879FF
MPX13_Mp0 Noise_net_data1n_above_low_Arc1_vdd out1n
Noise_net_data1n_above_low_Arc1_vdd Noise_net_data1n_above_low_Arc1_vdd
pch W=4u L=0.13u AS=1.08p AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
CX13_Mp0_miller_load out1n 0 0.530841FF
.IC V(data1n@n1)=0
V1 Noise_net_data1n_above_low_Arc1_gnd 0 DC=0
.IC V(out1n)=1.2
V2 Noise_net_data1n_above_low_Arc1_vdd 0 DC=1.2
.ENDS Noise_net_data1n_above_low_Arc1
```

The input to this subcircuit is data1n@n1, the net specified by the `write_spice` command. This subcircuit includes the two transistors of the inverter from data1n to out1n as well as the load transistors on out1n, included as inactive transistors along with their Miller capacitors.

The SPICE fanout noise deck includes the following measure statements:

```
* Noise measurements.
.MEAS TRAN X0.data1n@n1_max MAX V(X0.data1n@n1) FROM 0.0ns TO 0.191585ns
.MEAS TRAN X0.data1n@n1_noise_above_low PARAM='X0.data1n@n1_max'
.MEAS TRAN X1.out1n_min MIN V(X1.out1n) FROM 0.0ns TO 0.191585ns
.MEAS TRAN X1.out1n_noise_below_high PARAM='1.2 - X1.out1n_min'
```

In addition to the two measure statements for net data1n, there are also two similar measure statements for the fanout net out1n. Notice that because net out1n is inverted from net data1n, the type of noise analysis is `below_high` instead of `above_low`.

To get a SPICE fanout noise deck of a net other than the worst case net, you can add the `-fanout_net` option to the `write_spice` command as follows:

```
nt_shell> write_spice -output si_fanout -si_noise above_low -fanout \
          -fanout_net data1n
```

This command includes the NOR gate that is connected to out4n in the SPICE deck instead of the inverter that is connected to out1n. The SPICE deck is as follows:

```
.SUBCKT Noise_net_data1n_above_low_Arc1 data1n@n1
* data1n@n1 input conn_to_trigger
MNX12_Mn0 out4n data1n@n1 Noise_net_data1n_above_low_Arc1_gnd
Noise_net_data1n_above_low_Arc1_gnd nch W=1.5u L=0.13u AS=0.405p
AD=0.405p
PS=3.54u PD=3.54u NRS=0.091 NRD=0.157
MPX12_Mp0 Noise_net_data1n_above_low_Arc1_vdd data1n@n1 X12_n1
Noise_net_data1n_above_low_Arc1_vdd pch W=4u L=0.13u AS=1.08p AD=1.08p
PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
MPX12_Mp1 X12_n1 data4 out4n Noise_net_data1n_above_low_Arc1_vdd pch W=4u
L=0.13u AS=1.08p AD=1.08p PS=8.54u PD=8.54u NRS=0.069 NRD=0.045
* Off transistors
MNX12_Mn1 out4n 0 0 Noise_net_data1n_above_low_Arc1_gnd nch W=1.5u
L=0.13u
AS=0.405p AD=0.405p PS=3.54u PD=3.54u NRS=0.091 NRD=0.157
MNX14_Mn0 0 out4n 0 Noise_net_data1n_above_low_Arc1_gnd nch W=3u L=0.13u
AS=0.81p AD=0.81p PS=6.54u PD=6.54u NRS=0.069 NRD=0.045
CX14_Mn0_miller_load out4n 0 0.553879FF
MPX14_Mp0 Noise_net_data1n_above_low_Arc1_vdd out4n
Noise_net_data1n_above_low_Arc1_vdd Noise_net_data1n_above_low_Arc1_vdd
pch W=4u L=0.13u AS=1.08p AD=1.08p PS=8.54u
PD=8.54u NRS=0.069 NRD=0.045
CX14_Mp0_miller_load out4n 0 0.530841FF
.IC V(X12_n1)=1.2
.IC V(data1n@n1)=0
V1 data4 0 DC=0
V2 Noise_net_data1n_above_low_Arc1_gnd 0 DC=0
.IC V(out4n)=1.2
V3 Noise_net_data1n_above_low_Arc1_vdd 0 DC=1.2
.ENDS Noise_net_data1n_above_low_Arc1
```

Note:

The SPICE deck for fanout noise has two stages. The first stage is the injected noise stage and the second stage is the fanout noise stage. The trigger pin of the fanout noise stage has an injected noise bump from the previous stage. Regardless of whether the `-si_noise_shape` option is specified, the SPICE deck for fanout noise uses an independent voltage source for its trigger when user-defined injected noise is present at the trigger pin. The independent voltage source represents the user-defined injected noise. If the `-add_noise` option of the `set_input_noise` command has been specified at the trigger pin, a dependent voltage source (representing the simulated injected noise bump) is also added in series with the independent voltage source. The dependent voltage source replicates the simulated noise bump from the injected noise portion of the SPICE deck.

SPICE Decks for Silicon-on-Insulator Transistors

Using the `-soi_comments` option adds information about the state of each silicon-on-insulator (SOI) transistor in the output deck. An example is as follows:

```
nt_shell> write_spice -output maxpath.spc -soi_comments \
          [get_timing_paths  max  max_paths 1]
```

The resulting SPICE deck includes comments for each active transistor along with the initial condition of the bulk supply. Comments are also included for each inactive transistor:

```
MNXaddsub_Xi2_Mn0 Xaddsub_subi2@n9 sub@n10 Path28216_Arc1_gnd
Path28216_Arc1_gnd bulk_Xaddsub_Xi2_Mn0 nch W=4u L=0.13u AS=0p AD=0p
PS=4u PD=4u NRS=0.069 NRD=0.045
* soi transistor: state=1r0 body_voltage_bound=upper rail_reference=2.5
supply=1.2 is_user_specified_parameter=false
device_type=floating_rail_tied
.IC V(bulk_Xaddsub_Xi2_Mn0)=0.384

* Off transistors
MNXaddsub_Xib10_Mn0 0 Xaddsub_subi2@n7 0 Path28216_Arc1_gnd
bulk_Xaddsub_Xib10_Mn0 nch W=4u L=0.13u AS=0p AD=0p PS=4u PD=4u NRS=0.069
NRD=0.045
* soi off transistor: state=0r0 body_voltage_bound=average
rail_reference=2.5 supply=1.2 is_user_defined_parameter=false
device_type=floating_rail_tied
.IC V(bulk_Xaddsub_Xib10_Mn0)=0.048
CXaddsub_Xib10_Mn0_miller_load Xaddsub_subi2@n7 0 0.631953FF
```

Using SPICE Decks to Debug Dynamic Clock Simulation Regions

You can create a SPICE deck to debug dynamic clock simulation (DCS) regions by using the `-dcs` option of the `write_spice` command. This SPICE deck does not contain measurement statements and is not intended to generate any delay correlation data. Instead, its purpose is to provide a netlist of the DCS region.

The `-dcs` option cannot be used with the *`objects`* argument or the `-header`, `-soi_comments`, `-measure_subckt_delays`, `-measure_subckt_transitions`, or `-si_noise` options.

The `-dcs` option is for debugging purposes only. The initialization might take multiple cycles, especially if dividers or multipliers are included in the clock network.

Because this type of SPICE deck does not require path tracing results, you can use the `write_spice` command with the `-dcs` option any time after executing the `check_design` command.

Simulating SPICE Decks Created With the write_spice Command

The `-simulate` option of the `write_spice` command automatically launches simulations of SPICE decks generated by the `write_spice` command. You can perform custom preprocessing and postprocessing operations on the simulation runs and also take advantage of distributed processing.

The preprocessing and postprocessing scripts are part of the simulation run. The complete analysis of a single SPICE deck might include a preprocessing step, a simulation run, and a postprocessing step. Multiple instances of this three-step analysis are launched in parallel if distributed processing is enabled, as illustrated in Figure 11-5.

Figure 11-5 Parallel SPICE Simulations

The following variables control the simulations:

- The `write_spice_sim_cmd` variable

  This variable specifies the command that launches simulations in your computing environment. The default is `hspice -i <input_deck> -o <listing_file>`, which runs the HSPICE simulator. You can set the variable to an empty string to execute preprocessing or postprocessing scripts on the SPICE files without running any simulations.

  The `<input_deck>` tag represents a SPICE input deck. At execution time, the tag is replaced with the absolute path of the file. SPICE deck names are derived from the

argument of the `-output` option of the `write_spice` command. If you use the `-simulate` option, you must use the `-output` option.

The `<listing_file>` tag represents the root name of the SPICE input deck (the name without the file extension) and is used to generate output file names.

Note:

The `<input_deck>` and `<listing_file>` tags are syntax terms. You can change the `write_spice_sim_cmd` variable to use other simulation tools. To specify the input and output files, you must use these tags exactly as written, including the angle brackets and the wording inside the brackets.

- The `write_spice_sim_pre_processing` and `write_spice_sim_post_processing` variables

  Set the `write_spice_sim_pre_processing` variable to provide a script file name or an operating system command to be executed on each SPICE deck before simulation. Set the `write_spice_sim_post_processing` variable to provide a script or command to be executed immediately after a simulation run is completed.

  For example, you might want to search for and rename specific elements in a SPICE deck before simulation. After simulation, you might want to extract values from the results files to create a customized report.

  If you set either or both of these variables, the scripts (or commands) are executed even if you disable simulation by setting the `write_spice_sim_cmd` variable to an empty string. However, if all three variables are set to empty strings, the tool issues a DISP-008 warning message and the `-simulate` option has no effect.

  You can use the `<input_deck>` and `<listing_file>` tags in the command line or script.

- The `write_spice_sim_max` variable

  This variable specifies the maximum number of simulations to perform. NanoTime issues a warning message if a larger number of SPICE decks was generated. The default is 250,000.

- The `write_spice_sim_max_decks_per_task` variable

  This variable specifies the number of HSPICE decks that constitute one task. A task is a group of simulations submitted to one worker slot. The default is 20.

Example

In this example, assume that the distributed processing host file specifies a total of 10 worker slots. In other words, in this computing environment, at most 10 HSPICE simulations can be executed in parallel.

Assume that the control variables are set as follows:

```
set write_spice_sim_max 500
set write_spice_sim_max_decks_per_task 20
```

```
set write_spice_sim_pre_processing “before.sh”
set write_spice_sim_post_processing “after.tcl”
```

The NanoTime script contains the following command:

```
write_spice -output test.sp -insert_model_cards -measure_subckt_delays \
            -simulate [get_timing_paths -max -max_paths 1000]
```

The following actions occur:

- The tool creates one SPICE input deck for each of the 1000 timing paths with the longest delays. The file names are test.sp, test_00001.sp, test_00002.sp, and so on up to test_00999.sp.
- NanoTime packages the files into tasks, which are groups of 20 SPICE decks. The tool launches 10 parallel tasks, one on each of the 10 worker slots. Each worker runs the 20 simulations in its assigned task serially. When a worker completes a task, another task (package of 20 simulations) is assigned to that worker.
- Every simulation also includes executing one script before the simulation (before.sh) and another script after the simulation (after.tcl).
- Execution stops after 500 simulations, even though 1000 SPICE decks are available, because the `write_spice_sim_max` variable is set to 500.

See Also

- Distributed Processing

12

Timing Models for Hierarchical Analysis

A hierarchical analysis uses timing models to represent the behavior of lower-level blocks. This chapter describes the concepts and structure of NanoTime timing models and provides instructions for creating basic models.

This chapter includes the following sections:

- Overview of Extracted Timing Models
- Creating a Timing Model
- Types of Timing Models
- Context Dependency of Models
- Timing Model File Structure
- Transparent Timing Models
- Nontransparent Timing Models
- Differences Between the Path Tracing and Model Extraction Flows
- Specifying the Transition and Load Index Values
- Controlling the Paths Saved in an Extracted Timing Model
- Debugging a NanoTime Model
- Testing a NanoTime Model

Overview of Extracted Timing Models

A timing model represents the input and output timing of a design block. You can analyze a large chip design by using timing models to represent lower-level blocks in the hierarchy. Compared to analyzing a large, flat design, using timing models can reduce the total analysis time by breaking down a large task into smaller units, especially when lower-level blocks are used multiple times. Timing models are also useful for protecting intellectual property.

An extracted timing model (ETM) is a logic library file in Liberty format, an open-source format supported by many circuit design and analysis tools. The model consists of a set of input and output ports and timing arcs between those ports. Tables in the file model the delays and constraints of the critical paths in the design block.

NanoTime creates extracted timing models that are optimized for use with the PrimeTime tool. NanoTime can also read in and use timing models that were created by NanoTime (with the exception of CCS noise models).

Timing Model Usage Notes

All timing models are imperfect representations of the actual behavior of a circuit block. In most cases, the advantages of using a timing model outweigh the disadvantages. However, you should be aware of the limitations of using timing models.

General limitations are as follows:

- Critical path collapsing is always enabled.
- Due to critical path collapsing, nonboundary clock domains are forced into boundary clock domains and the top-level clock uncertainty might not be correct.
- All timing models are dependent on the top-level clock properties. If the clock definition changes, the model must be regenerated.
- There is no built-in checking to ensure that a model is used in an environment appropriate for the selections made when the model was created.
- The effects of complex analysis options such as path-based slack adjustment, dynamic simulation, and signal integrity analysis are directly included in the model delays. The model does not include information about whether such options were used or how they were specified.
- The effects of timing exceptions such as false path and multicycle path exceptions are directly included in the model delays. The model does not include information about whether timing exceptions were used or how they were specified.

Creating a Timing Model

The default extracted timing model is a context-independent transparent model that uses nonlinear delay modeling and contains delays calculated by the NanoTime internal simulator. This model is suited for most general-purpose applications.

The high-level procedure for extracting a timing model is as follows:

1. Read in and link the design information.
2. Apply worst-case clocking conditions, power supply voltages, transistor technologies, and parasitics.
3. Specify input delays, input transition times, output delays, and output loads that represent the environment in which the model will be used.
4. Run timing analysis with the `trace_paths` command.
5. Fix all timing violations.
6. Reset the paths database with the `reset_design` command.

   Alternatively, you can start a new run and repeat the design setup from the beginning. In this case, do not use the `trace_paths` command.
7. Create a model with the `extract_model` command. (Optional) Use command options to modify the model characteristics or to create debugging files.
8. (Optional) Debug the model by examining the optional ASCII version of the model file and the model paths file.
9. Test the model in a simple environment.
10. Use the model in hierarchical timing analysis in the NanoTime or PrimeTime tool.

For special purposes, you can modify the model to customize it for the environment in which it will be used. Table 12-1 lists some of the options available and provides links to further information in this user guide.

This chapter and the next chapter cover many timing model topics. For additional information about the options of the `extract_model` command, see the command man page.

Table 12-1 Model Creation Choices

Parameter	Choices (X = default)	For more information
Type of model	Transparent (X) Combinational Nontransparent Custom	Transparent Timing Models Nontransparent Timing Models
Usage context	Context-independent (X) Context-dependent	Context Dependency of Models
Timing representation	Nonlinear delay model (X) Composite current source models	Nonlinear Delay Timing Models Composite Current Source Timing Models Creating CCS Timing Models
Timing source	Internal simulator (X) HSPICE	HSPICE Recalibration
Boundary parasitics	Built-in (X) Extracted	Boundary Parasitics
Arc segmentation	Full-unate (X) Half-unate	Full-Unate and Half-Unate Models

Types of Timing Models

NanoTime can create the following types of extracted timing models:

- Transparent model (the default)

  A transparent timing model provides the most accurate representation of transparent paths through the design. The transparent timing model provides the best performance for latch-based designs. This is the only model type that can model any circuit that NanoTime can analyze.

  This model type uses transparent latches called the model input latch (MIL) and model output latch (MOL) to mimic the timing transparency windows. The model file tends to be somewhat larger and more complex than other model types.

- Combinational model

  The transparent path from in to out is represented as a combinational arc. Converting sequential paths into combinational arcs is inherently inaccurate. However, creating a combinational model is sometimes helpful for debugging modeling issues.

  This model type is suitable only for designs with a single clock domain.

- Nontransparent model

  The last latch (or other sequential element in the path) is forced to be nontransparent for a nonclock input. The path is represented by the setup time to the opening edge of the clock of this latch and by a path from this clock to the output.

  This model type is most suited for designs based on flip-flops. Even if you do not select this model type explicitly, NanoTime automatically recognizes when transparency between input and output does not exist and creates a nontransparent model.

- Custom model

  A custom model is a timing model specified in your own format. You use Tcl programming to get timing results after path tracing and write the results into a file. You are responsible for ensuring the compatibility of the custom model with subsequent analysis tools.

Table 12-2 lists some of the benefits and limitations of different types of timing models.

Table 12-2 Differences Between Extracted Timing Model Types

Model type	Benefits	Issues
Transparent (default)	Captures the worst setup constraint Captures the worst clock to out delay Can be used in different top-level phasing environments Preserves boundary clock domains The least pessimistic model Provides the best performance when used in timing analysis tools	Larger model file size Uses MIL and MOL structures, which can be difficult to understand and which some third-party tools cannot interpret
Combinational	Smaller model file size (no MIL and MOL structures) Captures the worst clock to out delay Captures the worst setup constraint	Pessimistic Always presents transparent paths even during nontransparent arrivals Might have false violations Not suitable for multiple clock domains
Nontransparent	Smaller model file size (no MIL and MOL structures) Captures the worst clock to out delay	Might not capture the worst setup constraint Might miss critical paths (especially if any transparent paths are critical paths) Might have false violations

Context Dependency of Models

Context refers to the design environment in which the model is to be used. If you use a timing model in an inappropriate context, the results are not likely to be accurate.

Most context dependencies result from your design and from the analysis choices you make when setting up the NanoTime run. The only context dependency that you can affect at the time of model creation is how to consider the input arrival times and slew constraints.

You cannot change the following context dependencies:

- Context-independent conditions (the model remains accurate even when these conditions change)
  - Data output required time
  - Data output load
- Context-dependent conditions (the model is accurate only under the conditions used in the analysis):
  - Clock frequency and duty cycle
  - Rail voltages
  - Logic libraries (technology libraries)

Context Dependency of Input Arrivals and Slew Constraints

The default NanoTime timing model is context-independent for input arrival times and slew constraints because, in most cases, you do not know exactly when the higher-level hierarchy will provide input data to the model.

To create a context-dependent model that requires the specific input arrival times and slew constraints provided in the NanoTime analysis setup, use the `-context_dependent` option with the `extract_model` command.

Limitations of context-independent models are as follows:

- The model is pessimistic for input arrivals that are not the worst-case arrivals.
- The model is pessimistic when signal integrity analysis is enabled.
- The model cannot account for edge-specific uncertainty. Noninput boundary latch checks are phase- and cycle-shifted back into the boundary latch clock domain.

Limitations of context-dependent models are as follows:

- There is no checking for a valid arrival context when the model is used.
- If the model will be used in multiple places in a design, it can be difficult to choose an appropriate context for saving the model.

When signal integrity analysis is enabled for a context-independent model, infinite timing windows are used from the input ports to the input boundary latches. For a context-dependent model, the SI timing windows use the specified input arrival times.

NanoTime achieves input arrival independence by ignoring the `set_input_delay` command on the data inputs and using instead a delay of 1000 ns for maximum delay paths and 0 ns for minimum delay paths. Figure 12-1 illustrates the effect for a maximum delay path through a transparent latch.

Figure 12-1 Effect of the extract_model -context_dependent Command

Part (a) shows normal latch input timing. The signal at pin D switches later than the input signal IN by the amount of the user-specified input delay. When clock CLK switches, the signal propagates from D to Q. During path tracing, the latch error recovery feature allows NanoTime to continue path tracing even in the presence of a setup violation by adjusting the data arrival time to the zero-slack value.

Part (b) of Figure 12-1 illustrates how the latch error recovery feature is used during model creation to maximize transparency and create a model that is independent of the input arrival time. For a maximum delay path, the tool inserts an input delay of 1000 ns. The arrival time at the latch is adjusted to the zero-slack value. This represents the worst possible input arrival time. The path search continues until the path endpoint or a nontransparent timing element is encountered. The model paths report shows the amount of timing adjustment.

Timing Model File Structure

A timing model is a logic library file in Liberty format, an open-source format supported by many circuit design and analysis tools. Logic library files can contain other types of information besides circuit timing. NanoTime only reads from or writes to the sections that are related to circuit timing.

The binary (compiled) version of an extracted timing model is a file with a *.db extension; binary files protect intellectual property and are faster to read. NanoTime creates *.db files by default, but offers the option to create ASCII *.lib files for model debugging.

Figure 12-2 shows the timing sections of a model file. Information is grouped by cell, then by pin. A pin section contains multiple timing sections if the pin is associated with multiple timing arcs; a timing section contains information about one timing arc.

Figure 12-2 Structure of a NanoTime Timing Model

See Also

- Appendix C, "Example latch.lib Model File"
- *Library Compiler User Guide*

Timing Arcs in the Model

The content of a timing section varies, depending on factors such as the type of timing arc and the timing representation (NLDM or CCS). Common types of timing arcs are as follows:

- Setup and hold constraint arcs
- Clock to output delay arcs
- Recovery and removal arcs
- Combinational input and output arcs
- Retain arcs

Every timing section contains a header, an opening comment, the timing information, and a closing comment. The comments provide information to help you understand the model.

- The opening comment contains a brief description of the timing arc and the path IDs associated with the arc. The IDs correspond to path IDs in an optional debug path report.

  NanoTime generates a timing model even if timing violations exist or other problems arise during path tracing. If a problem is found, the path ID is marked with an asterisk (*).

- The closing comment contains the name of the arc.

  The name of a timing arc is formed from the startpoint pin name followed by the endpoint pin name and a descriptive suffix. The default separator is the underscore character; use the `model_name_separator_char` variable to specify a different separator character. Table 12-3 lists some commonly encountered model name suffixes. A suffix might also include a number if multiple arcs would otherwise have the same name.

Table 12-3 Typical Suffixes Used in Timing Model Arc Names

Arc type	Edge type	Maximum delay arc	Minimum delay arc
Setup constraint	rising	stupr	stupr_min
Setup constraint	falling	stupf	stupf_min
Hold constraint	rising	hldr	hldr_min
Hold constraint	falling	hldf	hldr_min
Full-unate delay	n/a	una	una_min
Half-unate delay	falling	fedg	fedg_min
Half-unate delay	rising	redg	redg_min

Generated Clocks in the Model

A generated clock is represented in the model by a generated clock section that includes the reference clock (called the master pin) and the pin or pins on which the clock is defined. The master pin must be a clock pin, as indicated by the `clock : true` statement in the pin block.

For example:

```
generated_clock (div_x128) {
   clock_pin : “xi6/xiff/xmp1/main/g xi6/xiff/xmn1/main/g” ;
   master_pin : cin ;
   divided_by : 128 ;
}
/* End of generated clocks */

pin(“cin”) {
   direction : input ;
   clock : true ;
   max_transition “ 0.640000 ;
   capacitance : 1.141963 ;
   rise_capacitance_range (1.141963, 1.141963) ;
   fall_capacitance_range (1.141963, 1.141963) ;
} /* End of pin cin */
```

Transparent Timing Models

NanoTime creates transparent timing models by default. A transparent timing model provides the most accurate representation of transparent paths through a design. However, if the design has no transparent paths, the tool automatically creates a nontransparent model instead.

The transparent timing model can represent the widest variety of circuit designs and operating conditions. The model uses a set of transparent latches called model input latches (MILs) to represent all of the design inputs and a second set of transparent latches, called model output latches (MOLs), to represent the outputs. Using MIL and MOL structures enables efficient representation of all of the possible paths between multiple inputs and multiple outputs.

Transparent modeling concepts are discussed in the following sections:

- MIL and MOL Concepts
- Transparent Model Example
- Timing Arcs in the Example Model
- Multiple MILs and MOLs Per Pin

MIL and MOL Concepts

NanoTime models transparent paths using the Liberty tlatch structure, which is a transparent latch with C, D, and Q pins. Data passes from the D pin to the Q pin when the clock is in the logic state that enables the tlatch. The following is the beginning of a tlatch section in a model file:

```
pin("mil_d_IN_CLK_F_CLK") {
     direction : internal ;
     capacitance : 0.000000 ;
     tlatch("mil_c_IN_CLK_F_CLK") {
          edge_type : rising;
          tdisable : false;
     }
```

The tlatch element appears in the model file under its D input pin. The `edge_type : rising` statement indicates that the latch is active high, that is, it opens with a rising clock edge. The `tdisable` flag is always set to `false` to enable evaluation of the tlatch.

One MIL is created for every unique combination of input port, clock port, and capture clock domain. The clock domains are the internal phases derived from the input clock. A MIL is used to represent the following:

- The setup and hold timing of the data input relative to each associated clock domain
- The phasing of the first capturing device in the path from the data input

One MOL is created for every unique combination of output port, launch clock domain, and clock port. A MOL is used to represent the following:

- The minimum or maximum delays from each applicable clock input and edge
- The transparency window of the transparent path
- The phasing of the last capturing device in the path to the data output

Transparent Model Example

The simple latch circuit in Figure 12-3 is used to illustrate the components of a transparent model. The model file is provided in Appendix C, “Example latch.lib Model File.”

Note:

This example uses nonlinear delay model (NLDM) timing. A composite current source (CCS) model would contain the same structure and timing arcs as shown in this example, but the actual delay tables would be different.

Figure 12-3 Simple Latch Circuit

At a high level of hierarchy (in other words, to a circuit that uses the model), a timing model for this latch circuit must contain the following timing arcs:

- A delay arc from CLK rising to OUT
- Setup and hold constraint arcs from CLK falling to IN
- A combinational arc from IN to OUT, dependent on the state of CLK

The timing model does not contain these arcs directly. Instead, the model contains building block timing arcs, based on the tlatch element, that can be combined to yield the complete timing paths. For example, Figure 12-4 shows the model input latch (MIL), the model output latch (MOL), and the timing arcs in the transparent model for the circuit in Figure 12-3.

Figure 12-4 Arcs in the latch.lib Example File

The model contains three types of arcs:

- Constraint arcs: 1A, 1B, 11A, and 11B
- Delay arcs: 3 and 10
- Combinational arcs: 2, 4, 5, 6, 8, and 9

For hierarchical analysis, the CLK to OUT timing path is composed of model arcs 6 and 10. The IN to OUT path is composed of model arcs 5, 2, 8, and 9.

Table 12-4 shows the names of the arcs in the model, which are based on the pins at the arc startpoint and endpoint. The pins of a MIL or MOL do not correspond directly to a pin in the design. Instead, NanoTime uses a standardized naming convention for these pins.

The name of a MIL pin begins with mil_c, mil_d, or mil_q (always lowercase). The name is extended by adding the name of the input port connected to the MIL, the name and edge of the clock used for timing checks, and the name of the clock port from which the clock originates.The D pin of the MIL in Figure 12-4 is therefore named mil_d_IN_CLK_F_CLK (if the relevant edge of the clock is the falling edge).

Names of MOL pins are similar. They start with mol_c_mol_d, or mol_q and use the name of the output port connected to the MOL.

Table 12-4 Arc Names in the latch.lib Model

Arc	From pin	To pin	Arc name
1A, B	mil_c	mil_d	mil_c_IN_CLK_F_CLK_mil_d_IN_CLK_F_CLK_stupf mil_c_IN_CLK_F_CLK_mil_d_IN_CLK_F_CLK_hldf
11A, B	mol_c	mol_d	mol_c_OUT_CLK_R_CLK_mol_d_OUT_CLK_R_CLK_stupf mol_c_OUT_CLK_R_CLK_mol_d_OUT_CLK_R_CLK_hldf_min
2	mil_d	mil_q	mil_d_IN_CLK_F_CLK_mil_q_IN_CLK_F_CLK_una
3	mil_c	mil_q	mil_c_IN_CLK_F_CLK_mil_q_IN_CLK_F_CLK_una
4	CLK	mil_c	CLK_mil_c_IN_CLK_F_CLK_una
5	IN	mil_d	IN_mil_d_IN_CLK_F_CLK_una
6	CLK	mol_c	CLK_mol_c_OUT_CLK_R_CLK_una_min
8	mil_q	mol_d	mil_q_IN_CLK_F_CLK_mol_d_OUT_CLK_R_CLK_una
9	mol_d	OUT	mol_d_OUT_CLK_R_CLK_OUT_una
10	mol_c	OUT	mol_c_OUT_CLK_R_CLK_OUT_redg mol_c_OUT_CLK_R_CLK_OUT_redg_min

The default separator is the underscore character; use the `model_name_separator_char` variable to specify a different separator character.

Timing Arcs in the Example Model

This section describes several of the most common types of timing arcs that appear in a NanoTime extracted timing model.

Pass-Through Delay Arcs

A pass-through delay arc (also known as a zero delay arc) has no delay, but it preserves the transition time of the input signal by using a unit transition delay table. The zero delay arcs in Figure 12-4 are arcs 2, 3, and 5.

The following is part of the definition of arc 5 in the model, which is the path from IN to the D pin of the MIL (the path name is mil_d_IN_CLK_F_CLK). The tlatch appears in the section that defines the MIL D pin and is named the same as its clock pin name (mil_c_IN_CLK_F_CLK).

```
pin("mil_d_IN_CLK_F_CLK") {
        direction : internal ;
        capacitance : 0.000000 ;
        tlatch("mil_c_IN_CLK_F_CLK") {
                edge_type : rising;
                tdisable : false;
        }
        timing () {
                related_pin : "IN" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : input->mil d; */
                cell_rise( scalar ){
                        values ( "0.000000");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                  "1.000000, 1.000000");
                ...
} /* end of arc IN_mil_d_IN_CLK_F_CLK_una*/
```

To specify zero delay, the `cell_rise` and `cell_fall` values are set to a scalar value of zero. To specify that the transition time at the arc startpoint is preserved at the arc endpoint, the `rise_transition` and `fall_transition` tables are used. The tables vary the `index_1` value (the transition time at the arc startpoint, or IN) from 0 to 1 and set the entries in the `values` table (the transition time at the arc endpoint, the D pin) to vary in exactly the same way. This linear relationship is valid for any transition time because tools that use the timing model employ interpolation and extrapolation outside the given range. `Index_2` is the output load, which has no effect on the values in this case.

Constraint Arcs

Setup and hold constraints are represented in the model by constraint arcs between the C and D pins of a MIL or MOL. In the constraint arc tables (`rise_constraint` and `fall_constraint`), `index_1` is the constrained pin input slope and `index_2` is the related pin input slope.

The following example shows the beginning part of the setup constraint arc for the MIL. The timing section for this arc appears in the section that defines the MIL D pin (the constrained pin). The related pin is the MIL C pin.

```
timing ()   {
     related_pin : "mil_c_IN_CLK_F_CLK" ;
     timing_type : setup_falling ;
     /* comment : reference path 8, checked path 13,
                  reference path 7, checked path 15; */
     rise_constraint( f_dtrans_ctrans ){
             index_1 ( "0.080000, 0.160000, 0.320000");
             index_2 ( "0.080000, 0.160000, 0.320000");
             values ("0.002835, -0.015773, -0.040040",\
                     "0.017094, -0.001515, -0.025782",\
                     "0.035973, 0.017364, -0.006903");

            rise_constraint( f_dtrans_ctrans ){
            ...
} /* end of arc mil_c_IN_CLK_F_CLK_mil_d_IN_CLK_F_CLK_stupf*/
```

The following example shows the beginning part of the setup constraint for the MOL. A timing check is not necessary for modeling the MOL, but some simulation tools require that the model contain a timing check at each latch. The MOL setup and hold arcs are set to a scalar value such that the MIL D pin setup and hold arc falls before the MOL D pin arc.

```
pin(“mol_d_OUT_CLK_R_CLK”) {
...
     timing () {
          related_pin : "mol_c_OUT_CLK_R_CLK" ;
          timing_type : setup_falling ;
          /* comment : mol c->d transparency end setup,
                       rise path 14, fall path 16; */
          rise_constraint( scalar ){
               values ( "- 0.378071");
          }
          fall_constraint( scalar ){
               values ( "- 0.412058");
          }

} /* end of arc mol_c_OUT_CLK_R_CLK_mol_d_OUT_CLK_R_CLK_stupf*/
```

Delay Arcs

A delay arc includes four two-dimensional tables, representing the rise delay (labeled `cell_rise`), the rise transition time (`rise_transition`), the fall delay (`cell_fall`), and the fall transition time (`fall_transition`). The indexes of the four tables are identical; `index_1` is the transition time of the input net and `index_2` is the load (capacitance) of the output net.

For example, the model for the timing arc from the MOL D pin to the OUT port (arc 9 in Figure 12-4) begins as follows:

```
pin(“OUT”) {
...
     timing () {
          related_pin : "mol_d_OUT_CLK_R_CLK" ;
          timing_type : combinational ;
          timing_sense : positive_unate ;
          /* comment : mol d->out max launch arc, path 14 (delta),
                       path 16 (delta); */
          cell_rise( f_itrans_ocap ){
              index_1 ( "0.080000, 0.160000, 0.320000");
              index_2 ( "0.061393, 0.111393, 0.211393");
              values ( "-0.006496, 0.021207, 0.076998",\
                        "0.007764, 0.035466, 0.091257",\
                        "0.026644, 0.054346, 0.110137");
          }
          rise_transition( f_itrans_ocap ){
              index_1 ( "0.080000, 0.160000, 0.320000");
              index_2 ( "0.061393, 0.111393, 0.211393");
              values (  "0.074208, 0.134301, 0.254921",\
                        "0.074208, 0.134301, 0.254921",\
                        "0.074208, 0.134301, 0.254921");
          }
      ...
      } /* end of arc mol_d_OUT_CLK_R_CLK_OUT_una*/
```

Multiple MILs and MOLs Per Pin

There is no direct correspondence between the number of input and output ports and the number of MILs or MOLs in the model.

A unique MIL is required on an input port IN:

- For every unique clock that drives a boundary latch from IN
- For every unique closing clock edge that drives a boundary latch from IN
- For any transparent path (at least one) from IN to any output port that goes through a boundary latch driven by a unique clock or a unique closing edge

Similarly, a unique MOL is required on an output port OUT:

- For every unique clock that drives a boundary latch to OUT
- For every unique opening clock edge that drives a boundary latch to OUT
- For any transparent path (at least one) from any input port to OUT that goes through a boundary latch driven by a unique clock or a unique closing edge

Figure 12-5 shows a circuit with multiple MILs on the input. A similar circuit could be drawn with multiple MOLs driving the output.

Figure 12-5 Multiple Input Latch Circuit

In Figure 12-5, pin IN has three transparent paths to OUT going through latches L1, L2, and L3. Latches L1 and L2 are clocked by the same edge of CLK, so they are represented by a single MIL with the worst case timing. Because L3 is clocked on the other edge of CLK, it requires a separate MIL. The OUT pin requires only a single MOL.

Figure 12-6 shows the arcs in the transparent ETM for the circuit shown in Figure 12-5. As before, zero delay arcs are shown as a dashed line.

Figure 12-6 Arc Diagram

The arcs for the two tlatch objects (mil_c_IN_CLK_F_CLK and mol_c_OUT_CLK_R_CLK) are the same as those for the single latch case even though the paths (through L1, L2, and L4 in Figure 12-5) go through two latches instead of one. The additional arcs correspond to the additional tlatch, mil_c_IN_CLK_R_CLK. These arcs provide a setup constraint arc from the other edge of CLK to IN with its C to D arc and a portion of a second transparent path from IN to OUT with its Q to the MOL D arc.

Nontransparent Timing Models

By default, the extracted model preserves the timing behavior of the worst-case paths, including any number of successive transparent latches per path.

For special purposes, you can create nontransparent models by using the following options with the `extract_model` command:

- The `-combinational_input_to_output_transparency` option generates a transparent model that does not use the `tlatch` syntax (and therefore does not use MIL and MOL structures). Some third-party tools cannot use `tlatch` structures. Using combinational logic to represent the circuit results in a pessimistic model with transparent paths that are always enabled.
- The `-non_transparent` option sets the last latch in a transparent path through the model to nontransparent to prevent the path. In this case, the setup time is calculated relative to the opening edge of the last latch in the path.
- The `-ignore_input_to_output_transparency` option ignores the transparent path. In this case, the setup time is calculated relative to the closing edge of the last latch in the path.

Using the `-non_transparent` and `-ignore_input_to_output_transparency` options causes a change in the model only if there were transparent paths through the model; the changes only affect those input or output pins. For either option, the original transparent paths are split between setup and clock to output arcs, but their total delay might be more than twice the delay of the original transparent path.

If you know the maximum number of successive transparent latches in any given path within the design, you can use the `-latch_level` option of the `extract_model` command to set a limit. Doing so prevents the model generator from spending time trying to trace through more transparent latches than necessary. In addition, it helps ensure accurate identification of longer paths where the circuit structure might resemble a transparent latch.

Differences Between the Path Tracing and Model Extraction Flows

The `trace_paths` command and `extract_model` command both initiate path tracing in the NanoTime tool, but some aspects of the path tracing operation are different in these two modes.

When you use the `trace_paths` command, you are most likely working on the analysis setup or investigating circuit behavior. You might check the timing repeatedly as you make changes to the design or to the analysis setup. The goal of path tracing in this phase is to provide a large amount of information during the debugging process. Therefore, the `trace_paths` command performs path tracing using the specific operating conditions that

are in place at the time. In addition, the tool can save and report many types of information about intermediate nodes.

By contrast, when you use the `extract_model` command, you have already optimized the setup and fixed most (if not all) timing violations. The goal of path tracing in this phase is to create an accurate boundary timing model for use in hierarchical analysis in the NanoTime or PrimeTime tool. Typically, the `extract_model` command must enable the timing model to be used in design environments with varying operating conditions.

Path tracing for model extraction differs from standard path tracing in the following ways:

- The default NanoTime model maximizes transparency by ignoring the `set_input_delay` command specifications on the data inputs and using instead a delay of 1000 ns. This creates a context-independent timing model that is valid for any data arrival time.
- The tool discards internal latch-to-latch paths. Path searching invoked by the `extract_model` command is concerned only with data paths that begin or end at input or output ports.
- Paths through transparent topologies are checked against the clock for the boundary transparent timing topology (either the first transparent timing topology encountered when the tool searches from an input or the last transparent timing topology encountered when the tool searches to an output). The timing of the path is normalized to report the check to the boundary latch clock such that if this check is met, the timing to the internal latch is also met.

Transparency in Model Creation

Consider the case of a data signal arriving late at a topology for which a transparent check is being done during maximum delay path tracing. Using register error recovery, NanoTime adjusts the time to the closing edge of the clock and continues path tracing.

A portion of a path report generated by the `extract_model` command is as follows:

```
Path     Incr  Adjust      Trans Cap   NT   Point                Net
----     ----- --------    ----- ----- --   -------------------- --------
                  0.100                      clock clk1 (rise)
                  4.000                      input external delay
1000.000          0.000    0.100 0.020 D& f data1 (in)           data1
1000.000 0.000             0.100 0.020 &  f X1,Mp0.G (inv2)      data1
1000.046 0.045             0.054 0.020 &  r X2.Mn1.G (latch2)    data1n
   4.138 0.067 -995.974    0.091 0.045 A& f X2.X6.Mp0.G (inv2)   X2.lat1n
  (cycle=2 clock=clk1 low period=4.000 open=1.953 f close=4.138 r pin=X2.Mn0.G)
   4.174 0.036             0.044 0.028    r X11.Mn0.G (inv2)     q0
   4.196 0.022             0.026 0.019    f X4.Mp1.G (latch1)    q0n
   4.303 0.106             0.179 0.045 L& r X4.X6.Mn0.G (inv2)   X4.lat1n
   4.303                                      data arrival time
         0.277 -991.874                       Total
```

This is similar to a path report generated by the `trace_paths` command. However, the input delay time is set to 1000 ns to force maximum transparency. If the delay had been set such that it arrived at X2.lat1n before the clock transparency window, the trace paths report would not have shown this path. Because of the 1000 ns specification, the extracted model would still have reported the path. Also, the slack in the extracted model.paths file does not matter; it can be positive or negative. The setup and hold arc delay calculations use the actual path delays, modified by any user-specified margins. In this example, the total incremental delay of 0.277 ns is the data delay that is used with the incremental clock delay to the latch to determine the setup time.

Although register error recovery is indicated in path tracing reports, there might be references to latch error adjustment in specific instances. For example, you might see a reference to a latch error recovery as shown in the following warning message:

```
MXTR-030 (information) Offsetting LER adjustment ...
```

Even though a register error recovery search is performed, the delay adjustment uses latch error recovery because a transparent setup error occurred.

Data Path Restrictions

Only data paths that begin or end at input or output ports are captured during timing model extraction. Internal latch-to-latch paths are not retained.

For example, consider the circuit with two flip-flops shown in Figure 12-7, where ML stands for master latch and SL stands for slave latch. The partial path report in Example 12-1 shows the path between the flip-flops (highlighted in red in the figure).

Figure 12-7 Circuit With Two Flip-Flops

Example 12-1 Partial Trace Paths Report

```
Startpoint:       CLK (inport)
Endpoint:         XFF2/X1/Mp0/g
Path Type:        max
Constraint:       latch setup (transparent)
PBSA Common Net:  CLKFF

Path     Incr    Adjust  Trans  Cap    NT   Point                     Net
-------- ------- ------  -----  -----  --   ------------------------  ------------
 0.000           0.000   0.040  0.020  C  r CLK (in)                  CLK
 0.000   0.000           0.040  0.020  C  r XBuf2/X1/mdn1/g (inv)    CLK
 0.107   0.107           0.203  0.024  C  f XBuf2/X2/mup1/g (inv)    XBuf2/outb
 0.426   0.320           0.597  0.070  C  r XFF1/Xt3/mdn1/g (TGATE)  CLKFF
 0.680   0.254           0.251  0.017  L  r XFF1/X3/Mn0/g (nand)     XFF1/SL
 0.808   0.128           0.191  0.049     f XBuf4/X1/mup1/g (inv)    Q1
 0.962   0.154           0.249  0.024     r XBuf4/X2/mdn1/g (inv)    XBuf4/outb
 1.169   0.206           0.267  0.017  L  f XFF2/X1/Mp0/g (nand)     XFF2/ML
 1.169                                      data arrival time
         1.169   0.000                      Total
```

```
10.000          10.000   0.040  0.020  C  r CLK (in)                 CLK
10.000   0.000           0.040  0.020  C  r XBuf2/X1/mdn1/g (inv)    CLK
10.107   0.107           0.203  0.024  C  f XBuf2/X2/mup1/g (inv)    XBuf2/outb
10.426   0.320           0.597  0.070  C  r XFF2/Xt1/mup1/g (TGATE)  CLKFF
         0.426  10.000                      Total
10.426   0.000                              setup time
10.426   0.000                              clock uncertainty
10.426                                      data required time
-------------------------------------------------------------------------------
10.426                                      data required time
-1.169                                      data arrival time
-------------------------------------------------------------------------------
 9.258                                      slack (MET)
```

However, when you run the `extract_model` command, this internal path (highlighted in Figure 12-7) is not captured in the extracted timing model. Only the setup and hold arcs (CLK to IN1) and the clock output arcs (CLK to OUT) are captured in the model.

These paths can be verified using the `trace_paths` command. The model does not need to check this type of path because the input and output conditions do not change the timing result. The models created are only valid for the specified clock period. Changing the clock period can cause this type of path to fail because a given model is valid only for the specified clock waveforms.

Path Checking for Boundary Transparent Topology

All paths through the transparent timing topologies are checked against the clock for boundary transparent timing topology, which is either

- The *first* transparent timing topology encountered when the tool searches *from an input*
- The *last* transparent timing topology encountered when the tool searches *to an output*

The timing of the path is normalized to report the check to the boundary latch clock such that if this check is met, the timing to the internal latch is also met.

If the check were to be made against the clock for the internal latch, the setup check might pass while the check to the first transparent latch fails. Also, for correct phasing, any timing tool using this model must be able to determine the boundary clock domain.

Transparent Input Latch

For example, consider the circuit shown in Figure 12-8, which consists of a transparent latch clocked by CLK1 and a nontransparent latch clocked by CLK2. The following commands specify nonoverlapping clocks. As a result, the setup check for IN2 to CLK1 appears in the .lib model for this circuit.

```
create_clock -period 2.0 -rise 0.1 -fall 0.9 CLK1
create_clock -period 2.0 -rise 1.1 -fall 1.9 CLK2
```

Figure 12-8 Circuit With One Transparent Latch and One Nontransparent Latch

The relevant portion of the timing model is as follows:

```
pin("IN2") {
     direction : input ;
     max_transition : 0.320000 ;
     capacitance : 0.016292 ;
     rise_capacitance_range (0.016292, 0.016292);
     fall_capacitance_range (0.016292, 0.016292);
     timing () {
          related_pin : "CLK1" ;
          timing_type : setup_falling ;
          /* comment : reference path 13 (11), checked path 26 (25),
             reference path 14 (12), checked path 28 (27); */
          rise_constraint( f_dtrans_ctrans ){
               index_1 ( "0.080000, 0.160000, 0.320000");
               index_2 ( "0.080000, 0.120000, 0.160000,0.200000");
               values ( "0.120592, 0.112625, 0.105625, 0.101517",\
                        "0.138193, 0.130226, 0.123225, 0.119117",\
                        "0.161422, 0.153454, 0.146454, 0.142346");
```

The model file contains a pin section for every boundary pin of the design. The first few statements in the section are properties of the pin. Every pin section contains at least one timing section, which can describe timing arcs that start at the pin in question or constraint arcs for timing checks that are evaluated at the pin.

This example shows a setup constraint arc that describes the setup time from pin CLK1 to pin IN2. The comment in the timing section lists the paths involved in the setup check.

The setup delay is presented as a two-dimensional lookup table whose indexes are the constrained pin transition time (index_1) and the related pin transition time (index_2). In this example, IN2 is the constrained pin and CLK1 is the related pin.

As shown in the following example, the checked path for the rising constraint is path 26 in the .paths file, which shows the timing check to CLK2. Note the comment within the constraint for this pin.

```
Path ID:       26
Startpoint:    IN2 (in port)
Endpoint:      X10.X6.Mn0.g
Path Type:     max
Constraint:    non-transparent setup (switch over from latch setup (transparent))

Path      Incr    Adjust      Trans   Cap     NT   Point                     Net
--------  ------- --------    -----   -----   --   -----------------------   ----------
                     0.0000                          input external delay
1000.0000            0.0000   0.1600  0.0163  D  r IN2 (in)                  IN2
1000.0000 0.0000              0.1600  0.0163     r X2.Mn1.g (nand2)          IN2
1000.0564 0.0564              0.0738  0.0343     f X5.Mp1.g (latch)          DIN1
   1.0109 0.1082  -999.1537   0.1560  0.0553  A  r X5.X6.Mn0.g (inv4)       X5.lat1n
     (cycle=1 clock=CLK1 high period=2.0000 open=0.1975 r close=1.0109 f pin=X5.Mp0.g)
   1.0418 0.0310              0.0425  0.0390  &  f X6.Mp0.g (inv2)          Q1
   1.1254 0.0836              0.1540  0.1277  &  r X7.Mn0.g (inv2)          Q1N
   1.1640 0.0386              0.0498  0.0277     f X10.Mp1.g (latch)        DIN2
   1.2670 0.1029              0.1554  0.0553  L  r X10.X6.Mn0.g (inv4)      X10.lat1n
   1.2670                                            data arrival time
          0.4207  -999.1537                          Total

   1.1000            1.1000   0.1600  0.0163  C& r CLK2 (in)               CLK2
   1.1000 0.0000              0.1600  0.0163  C& r X8.Mn0.g (inv2)         CLK2
   1.1392 0.0392              0.0509  0.0277  C& f X9.Mp0.g (inv2)         CLK2N
   1.1708 0.0317              0.0407  0.0277  C& r X10.X1.Mn0.g (inv2)     CLKL2
   1.1975 0.0267              0.0325  0.0385  C& f X10.Mp0.g (latch)       X10.clkn
          0.0975     1.1000                          Total
   1.1975 0.0000                                     setup time
   1.1975 0.0000                                     clock uncertainty
   1.1975                                            data required time
-------------------------------------------------------------------------------------
   1.1975                                            data required time
  -1.2670                                            data arrival time
-------------------------------------------------------------------------------------
  -0.0695                                            slack (VIOLATION)
```

NanoTime derives the setup to CLK1 by adjusting the CLK2 setup. The tool calculates the setup time to the CLK2 rising edge by using the total incremental delay of 0.4207 ns for the input delay minus the incremental CLK2 delay of 0.0975 ns, which results in a setup time of 0.3232 ns to the CLK2 rising edge. Next, the tool subtracts the time between the falling edge of CLK1 and the rising edge of CLK2 (0.16 ns) to end up with a setup time of 0.1232 ns to the CLK1 falling edge. You can see this value in the timing model example for pin IN2, in the third entry in the second row of values in the rise_constraint table (corresponding to a slew of 0.16 for both input and data).

If the setup to CLK2 had been used in the ETM and the data arrived only 50 ns before the rising edge of CLK2, the model would not have given an error, but the setup to CLK1 on latch1 would have failed. Using the adjusted setup time assures that the correct timing is

checked when the model is used. Note that the model is dependent on the specified relationship of the clocks in the NanoTime run.

This example shows a path with negative slack at latch2. If the setup for latch2 had been positive, the model would have saved the path and the setup in the model would have been only to latch1. This occurs because meeting the latch1 setup time would have guaranteed that the setup time to latch2 was met.

Transparent Output Latch

Now consider the case where, in Figure 12-8, latch1 is nontransparent and latch2 is transparent. Examine the model delay to OUT1. In this case, if a path exists from CLK1 through latch1 and then through latch2 to OUT1, the path is modeled as a delay arc from CLK2.

Here is the initial section of the .lib model for pin OUT1:

```
pin("OUT1") {
     direction : output ;
     max_capacitance : 0.200000 ;
     min_capacitance : 0.000001
     capacitance : 0.011393 ;
     rise_capacitance_range (0.011393, 0.011393);
     fall_capacitance_range (0.011393, 0.011393);
     timing () {
          related_pin : "CLK2" ;
          timing_type : rising_edge ;
          /* comment : clk->out for OUT1:CLK2:R:CLK2, path 17, path 18;*/
          cell_rise( f_itrans_ocap ){
               index_1 ( "0.080000, 0.120000, 0.160000, 0.200000");
               index_2 ( "0.011394, 0.061393, 0.111393, 0.211393");
               values ( "0.285852, 0.312389, 0.340068, 0.395869",\
                        "0.293819, 0.320357, 0.348035, 0.403836",\
                        "0.304927, 0.331465, 0.359143, 0.414944");
```

The following example shows a portion of the path tracing report for path 17 from the model paths file.

```
Item:               5
Path ID:            17
Startpoint:         CLK1 (in port)
Endpoint:           OUT1 (out port)
Path Type:          max
Constraint:         set_output_delay check

    Path     Incr   Adjust   Trans    Cap     NT    Point                    Net
-------- -------- -------- - -------- ----    --    ----------------------   -------
  0.1000            0.1000   0.1600   0.0163  C&  r CLK1 (in)                CLK1
  0.1000   0.0000            0.1600   0.0163  C&  r X3.Mn0.g (inv2)         CLK1
  0.1392   0.0392            0.0509   0.0277  C&  f X4.Mp0.g (inv2)         CLK1N
  0.1708   0.0317            0.0407   0.0277  C&  r X5.X1.Mn0.g (inv2)      CLKL1
  0.1975   0.0267            0.0325   0.0385  C&  f X5.X2.Mp0.g (inv2)      X5.clkn
  0.2199   0.0224            0.0297   0.0209  C   r X5.Mn0.g (latch)        X5.clk
  0.2692   0.0493            0.0826   0.0553  L   f X5.X6.Mp0.g (inv4)      X5.lat1n
  0.5815   0.3122            0.6231   1.0390  &   r X6.Mn0.g (inv2)         Q1
  1.1379   0.5564            0.7808   1.0277  &   f X7.Mp0.g (inv2)         Q1N
  1.2454   0.1075            0.1441   0.0277      r X10.Mn1.g (latch)        DIN2
  1.3272   0.0818            0.0944   0.0553  L   f X10.X6.Mp0.g (inv4)     X10.lat1n
   (cycle=1 clock=CLK2 high period=2.0000 open=1.2199 r close=2.0318 f pin=X10.Mn0.g)
  1.3621   0.0349            0.0435   0.0390  &   r X11.Mn0.g (inv2)        Q2
  1.3852   0.0231            0.0269   0.0277  &   f X12.Mp0.g (inv2)        Q2B
  1.4008   0.0156            0.0184   0.0114  O   r OUT1 (out)              OUT1
  1.4008                                              data arrival time
           1.3008   0.1000                             Total


  0.0000   0.0000                                      output external delay
  0.0000   0.0000                                      clock uncertainty
  0.0000                                               data required time
-------------------------------------------------------------------------------
  0.0000                                               data required time
 -1.4008                                               data arrival time
-------------------------------------------------------------------------------
 -1.4008                                               slack (VIOLATED)
```

The model output paths are checked to the last transparent topology in the path. Even though the path is from CLK1, the modeled arc is specified from CLK2. The incremental delay of the path is shown as 1.3008 ns, but NanoTime subtracts the difference between the two clock opening edges, which is 1.0 ns. Therefore, the modeled delay time from CLK2 is 0.3008 ns.

Note that for this path, the modeled arc must start at CLK2 to ensure proper timing of downstream circuitry. However, the path characterization (the transition and load indexes) is taken from CLK1 because the actual physical path originates from CLK1.

Properties of Paths That Contain Generated Clocks

A generated clock does not inherit model transition indexes from its master clock. When you create a generated clock, you should set the indexes explicitly by using the `set_model_input_transition_indexes` command.

If you do not set the indexes explicitly, NanoTime obtains the index values from the `model_default_input_transition_indexes` variable.

Paths in Timing Models That Contain Errors

NanoTime generates a timing model even if timing violations exist or other problems arise during path tracing. If a problem is found, the tool marks the path ID with an asterisk (*) in the comment line. For example, if the analysis of path 26 exceeds the allowed analysis time, the analysis is truncated and the reported setup time might not be accurate. In this case, the comment line appears as follows:

```
/* comment : reference path 13 (11), checked path 26* (25),
   reference path 14 (12), checked path 28 (27); */
```

Specifying the Transition and Load Index Values

The delay through a timing model varies according to the input transition times and output loads that occur where the model is used. During model creation, NanoTime analyzes the circuit behavior under a range of operating conditions and generates lookup tables to represent how the delay changes under those conditions. The specific transition time and load values used in the lookup tables are called index values.

When NanoTime uses a timing model, the tool uses interpolation for conditions between the index values and extrapolation for conditions beyond the index values to calculate the delay through the model.

You can specify the range and spacing of lookup table entries in the extracted model by using the following commands before you extract the model:

- `set_model_input_transition_indexes`
- `set_model_load_indexes`

The `set_model_input_transition_indexes` command sets the values that determine the range and spacing of transition times in the model lookup tables for specified inputs. It can also specify the nominal value used in the initial collection of paths for the model.

For example, the following command sets the input transition values to 0.0, 0.5, and 1.0 in the maximum-slew model lookup tables and sets the nominal value to 0.8, for all inputs in the design:

```
nt_shell> set_model_input_transition_indexes \
          -max -nominal 0.8 {0.1 0.5 1.0} [all_inputs]
```

Similarly, the `set_model_load_indexes` command in the following example sets the output load values to 0.2, 0.4, and 0.6 in the maximum-load model lookup tables for all outputs in the design:

```
nt_shell> set_model_load_indexes -max {0.2 0.4 0.6} [all_outputs]
```

The specified ranges of values should reflect the actual expected ranges of transition times and output loads where the model is used. At least three index values must be specified. If you do not set the index values explicitly, the model generator uses the index values from the `model_default_input_transition_indexes` and `model_default_load_indexes` variables.

Note:

If signal integrity analysis is enabled, the output load index range should begin from a value smaller than the minimum coupling capacitance and end at a value that is 3 times the coupling capacitance expected for maximum loading.

Under some conditions, NanoTime uses the load index values of the existing CCS driver model. If the `-library_elements` option is set to `{nldm ccs_timing}` in the `extract_model` command, the design for which you are generating an extracted model contains library cells, and a CCS-based library cell drives an output port of the design, NanoTime ignores the load index values specified by the `set_model_load_indexes` command and the `model_default_load_indexes` variable.

Library cells do not affect the input transition index values. NanoTime always uses the values specified by the `set_model_input_transition_indexes` command or the `model_default_input_transition_indexes` variable.

Controlling the Paths Saved in an Extracted Timing Model

An extracted model contains only the essential timing paths that represent the timing behavior of the circuit. However, several options allow you to affect the paths saved in a NanoTime model.

If you make changes to the design or to the analysis conditions, the worst-case path might change. For example, annotating parasitics to a design might change the calculated delays substantially. If you want to monitor specific paths before and after the changes, you might need to increase the number of paths saved in the model.

Including Additional Paths in a Timing Model

By default, the extracted model preserves the timing behavior only of the single worst-case path from each input to the single most constraining sequential element in the fanout of the input port. You can include additional paths in the model in several ways. Using these options results in a larger model that provides increased visibility into the behavior of the original design. However, it does not increase the timing accuracy of the model for subcritical paths.

- The `-extend_inputs` and `-extend_outputs` options increase visibility into the behavior of the original design with respect to input and output ports.
  - The `-extend_inputs` option

    Considering all of the sequential elements in the fanout of an input port increases the model's visibility into the design for that input port. For example, if you change the external conditions to correct the worst violation, a model with extended input visibility can still detect other possible violations, such as in the path to the second-worst or third-worst sequential element. Use the `-extend_inputs` option to specify the names of the input ports that require this increased visibility.
  - The `-extend_outputs` option

    Each output port can have both clock-to-output and input-to-output timing arcs. By default, the model includes only input-to-output timing arcs of transparent paths where the arrival time is later than the worst clock-to-output arrival. Use the `-extend_outputs` option to specify the names of the output ports that require consideration of all input-to-output timing arcs.
- The `-keep_paths_within` option keeps timing paths in the path database that have a path delay within the specified time value of the worst path delay of each startpoint-endpoint pair. By default, the value is set to 0.0, which keeps only the single worst path per startpoint-endpoint pair. To keep more paths, specify a time value larger than 0.0.
- The `-npaths` option of the `extract_model` command saves more paths per startpoint-endpoint pair.

Restricting Timing Arc Types

To restrict the arc types produced in the extracted model, use the `-arc_types` option and specify the types of arcs you want to extract, either `min` or `max`. If you specify `min`, only the timing arcs from the minimum-delay (shortest) paths are extracted. If you specify `max`, only the timing arcs from the maximum-delay (longest) paths are extracted. In the absence of the `-arc_types` option, all arc types are extracted.

By default, the `extract_model` command cannot be used after `set_find_path` commands have been issued because model generation uses its own complete set of paths. However, if you have a specific path in mind to use for creating a constraint arc or delay arc (or both) for a model, and you would rather have that path used than the worst-case path found by the normal path tracing performed by the `extract_model` command, you can do so. Execute one or more `set_find_path` commands, then execute `extract_model` with the `-use_find_path` option. The tool first completes normal model path tracing, then performs the searches defined by the `set_find_path` commands. Any path found by a search using a `set_find_path` command replaces a path saved earlier, even if it is a subcritical path (not the worst-case path).

Debugging a NanoTime Model

NanoTime provides several features to help you debug a timing model.

- To create an ASCII version of the model (file *name*.lib) without creating the binary version (file *name*.db), use the `-debug lib_only` option of the `extract_model` command. This option saves time when you want to examine the contents of a model before creating the final version.
- To identify the worst-case paths used in the model, use the `-debug {paths lib}` or `-debug {paths lib_only}` option of the `extract_model` command. This option creates an ASCII version of the model (file *name*.lib) and a paths file (file *name*.paths) that lists the worst-case paths used in the timing model. The .lib file contains comment lines that show which timing arcs are generated from which paths.
- To create a Tcl collection of the paths used in the timing model, use the `get_timing_paths` command with the `-model_paths` option. By default, the collection includes only data paths. To get a collection of clock paths instead, add the `-clock_only` option to the command.
- To examine paths that have timing violations, set the `timing_save_violating_internal_model_paths` variable to `true` (the default is `false`).

Saving Extra Paths With Timing Violations

If NanoTime encounters timing violations when performing path tracing from clocks during model extraction, the tool issues MXTR-031 or MXTR-060 warning messages, as follows:

- MXTR-060 messages indicate paths that receive a latch error recovery adjustment. This includes most transparent setup check violations.
- MXTR-031 messages indicate paths that do not receive a latch error recovery adjustment. This includes most hold check violations, most nontransparent setup check violations, and most violations of user-defined setup checks that were created with the `create_timing_check` or `set_timing_check_attributes` commands without specifying the `-continue` option.

By default, NanoTime saves one path per startpoint-endpoint pair in the timing model. You can save more paths per startpoint-endpoint pair by using the `-npaths` option of the `extract_model` command. However, the additional paths might not be the violating paths.

To save the violating paths, set the `timing_save_violating_internal_model_paths` variable to `true` (the default is `false`). In this case, the tool saves all paths with timing violations, regardless of the setting of the `extract_model -npaths` option. Enabling this variable might increase memory usage and model file size.

The warning messages and additional paths are available only for paths that start from clock inputs. Timing violations that originate from nonclock inputs are not covered and might still exist within the design.

Note:

The extracted timing model is not changed as a result of enabling this variable. A timing model created in the presence of timing violations cannot represent the design correctly. The best practice is to resolve all timing violations by performing comprehensive timing analysis with the `trace_paths` and `report_paths` commands before creating an extracted timing model.

NanoTime issues one error message per pin. For iterative analysis such as signal integrity, differential skew, and multi-input switching analysis, the warning message is issued for the worst-case violation at a pin after all iterations are complete. However, NanoTime might save more than one timing path per pin. For each timing violation at a pin, the tool saves one path per pin for each clock startpoint from which the violating pin can be reached.

You can use the `report_paths` command after the `extract_model` command to examine the problem paths. To help filter paths for reporting, the following attributes of the `timing_path` object class are set to `true` for paths that are saved as a result of enabling the `timing_save_violating_internal_model_paths` variable: the `internal_modeling_violation` attribute for paths that receive an MXTR-031 warning and the `internal_modeling_violation_with_ler` attribute for paths that receive an MXTR-060 warning.

Testing a NanoTime Model

A timing model is valid only for the clock waveforms specified at the time of model creation. Varying the clock waveforms can change the transparency of the circuit and the paths modeled by NanoTime.

For testing purposes, NanoTime can create a Verilog file that instantiates the model. Use the `-test_design` option of the `extract_model` command, as shown in the following example:

```
extract_model -debug { lib paths } -test_design -name latch
```

This command produces a Verilog file named latch_test.v in addition to the model files.

You can test how the model is used in the hierarchy by reading in the latch.lib file and specifying the Verilog file as the netlist:

```
set link_path " * latch_lib.db $link_path"
read_lib latch.lib
set link_prefer_model_port { latch }
register_netlist -format spice {tech.sp}
register_netlist -format verilog {latch_test.v}
link_design test_test
```

To remove MIL and MOL timing checks on the zero delay paths, you should also source the SDC file that is created when you execute the `extract_model` command. This file contains commands to prevent the tracing of unneeded arcs in the model. In the current example, the unneeded arcs are the C to Q arcs (3) on the MIL and the unnecessary hold check on the MOL constraint arc.

The SDC file should be sourced before you run the `check_design` command:

```
source latch.sdc
check_design
```

The arcs produced when using the ETM are dependent on the input arrival time. First, consider the case when the signal at IN arrives after the transparency window. The following statements cause port IN to switch 0.3 ns after the falling edge of CLK:

```
create_clock -period 4.0 CLK
set_input_delay -clock CLK -clock_fall 0.3 {IN}
```

The maximum delay path report shows four paths:

```
         Path    Path
Slack    Delay   Type   Startpoint     Endpoint
------   ------  -----  ----------  --------------------------
3.6330   0.0000  D-L    f  IN       f  core.mil_d_IN_CLK_F_CLK
3.6972   0.0000  D-L    r  IN       r  core.mil_d_IN_CLK_F_CLK
3.7647   0.2353  C-O    r  CLK      f  OUT
3.7761   0.2239  C-O    r  CLK      r  OUT
```

Notice that the tlatch pins within the model become path points when the model is used in the hierarchy. The first two paths are setups to the MIL, which have positive slack values of almost a full period because the check is relative to the falling edge of the clock. The arrival time occurs before the transparency window of the latch, so the path stops at this point. The next two paths are from CLK to OUT.

Here are the details of the setup path for IN falling:

```
Startpoint:        IN (in port)
Endpoint:          core.mil_d_IN_CLK_F_CLK
Path Type:         max
Constraint:        model setup constraint arc: core (test_lib.test)

   Path     Incr  Adjust   Trans   NT     Point
--------  ------  ------  -------  --  --------------------------------
                  2.0000                 clock CLK (fall)
                  0.3000                 input external delay
 2.3000           0.0000   0.0800  D   f IN (in)
 2.3000   0.0000           0.0800  M   f core.IN (test)
 2.3000   0.0000           0.0800  L   f core.mil_d_IN_CLK_F_CLK (test)
 2.3000                                  data arrival time
          0.0000  2.3000                 Total

 6.0000           6.0000   0.0800  C   f CLK  {in}
 6.0000   0.0000           0.0800  C   f core.CLK (test)
 6.0000   0.0000           0.0800  C   f core.mil_c_IN_CLK_F_CLK (test)
          0.0000  6.0000                 Total
 5.9330  -0.0670                         setup time
 5.9330   0.0000                         clock uncertainty
 5.9330                                  data required time
-----------------------------------------------------------------
 5.9330                                  data required time
-2.3000                                  data arrival time
-----------------------------------------------------------------
 3.6330                                  slack (MET)
```

This table shows the setup to the MIL D input (the core.mil_d_IN_CLK_F_CLK entry) to be 3.633 ns, using .067 ns as the setup time from the MIL.

The report is different if the input IN is referenced to the rising edge of the clock and sufficient delay is added to make the transition occur after the falling edge of the clock. You can do this by using the following command:

```
set_input_delay -clock CLK 2.3 {IN}
```

The path report, as shown in the following example, shows the same four paths. Note that the `trace_latch_error_recovery` variable is set to `false` for this run. If you do not

change the value from `true` (the default), the timing is adjusted and you would also see the path to OUT in the report.

```
         Path    Path
Slack    Delay   Type   Startpoint     Endpoint
------   ------  -----  ----------  --------------------------
-0.3670  0.0000  D-L    f  IN       f  core.mil_d_IN_CLK_F_CLK
-0.3028  0.0000  D-L    r  IN       r  core.mil_d_IN_CLK_F_CLK
 3.7647  0.2353  C-O    r  CLK      f  OUT
 3.7761  0.2239  C-O    r  CLK      r  OUT
```

This result differs from the previous one in that the slack is negative for the arc from IN to the MIL. Because IN is referenced to the rising edge of the clock, the data is expected to arrive at the next falling edge of CLK.

Now consider the case when IN arrives during the latch transparency window. You get this condition by specifying IN to begin 0.3 ns after the rising edge of CLK:

```
set_input_delay -clock CLK 0.3 {IN}
```

The output report for max paths now shows eight paths:

```
         Path    Path
Slack    Delay   Type   Startpoint     Endpoint
------   ------  -----  ----------  --------------------------
 1.6330  0.0000  D-L    f  IN       f  core.mil_d_IN_CLK_F_CLK
 1.6972  0.0000  D-L    r  IN       r  core.mil_d_IN_CLK_F_CLK
 1.8358  0.2762  D-L    f  IN       f  core.mol_d_OUT_CLK_R_CLK
 1.8570  0.2211  D-L    r  IN       r  core.mol_d_OUT_CLK_R_CLK
 3.4642  0.2358  D-O    f  IN       f  OUT
 3.5131  0.1869  D-O    r  IN       r  OUT
 3.7647  0.2353  C-O    r  CLK      f  OUT
 3.7761  0.2239  C-O    r  CLK      r  OUT
```

The first two paths are setups to the MIL, and the next two are setups to the MOL. The fifth and sixth paths go from IN to OUT, and the final two paths go from CLK to OUT. Here is the path report for IN falling to OUT falling:

```
Startpoint:      IN (in port)
Endpoint:        OUT (out port)
Path Type:       max
Constraint:      set_output_delay check

Path    Incr     Adjust  Trans   NT Point                        Net
-------------------------------------------------------------------
                 0.0000             clock CLK (rise)
                 0.3000             input external delay
 0.3000          0.0000  0.0800  D  r IN (in)                     IN
 0.3000  0.0000          0.0800  M  r core.IN (test)              IN
 0.3000  0.0000          0.0800  L  r core.mil_d_IN_CLK_F_CLK (test)
  (cycle=1 clock=CLK high period=4.0000 open=-0.1095 r close=2.0000 f
                      pin=core.mil_c_IN_CLK_F_CLK)
 0.3000  0.0000          0.0800  M  r core.mil_q_IN_CLK_F_CLK (test)
 0.5211  0.2211          0.0800  L  r core.mol_d_OUT_CLK_R_CLK (test)
  (cycle=1 clock=CLK high period=4.0000 open=0.2516  r close=2.0000 f
                      pin=core.mol_c_OUT_CLK_R_CLK)
 0.4869 -0.0342          0.0141  M  r core.OUT (test)            OUT
 0.4869  0.0000          0.0141  O  r OUT (out)                  OUT
 0.4869                               data arrival time
         0.1869  0.3000               Total

 4.0000  4.0000                       clock CLK (rise)
 4.0000  0.0000                       output external delay
 4.0000  0.0000                       clock uncertainty
 4.0000                               data required time
-------------------------------------------------------------------
 4.0000                               data required time
-0.4869                               data arrival time
-------------------------------------------------------------------
 3.5131                               slack (MET)
```

Note that the path shows the MIL and MOL checkpoints with the clock timing at those points, including the delay from the input pin CLK to the MIL and MOL pins. Note also that the arc 4 delay from CLK to MIL C is included in the MIL clock pin:

```
cycle=1 clock=CLK high period=4.0000 open=0.1095 r close=2.0000 f
pin=core.mil_c_IN_CLK_F_CLK
```

The MOL clock includes the delay for the CLK to the MOL clock pin:

```
cycle=1 clock=CLK high period=4.0000 open=0.2516 r close=2.0000 f
pin=core.mol_c_OUT_CLK_R_CLK
```

13

Advanced Timing Model Options

This chapter describes ways that you can modify extracted timing models for special purposes.

This chapter includes the following sections:

- Merging Models
- Boundary Parasitics
- Full-Unate and Half-Unate Models
- Composite Current Source Models
- HSPICE Recalibration
- Context Characterization

Merging Models

A block can have different operating modes, such as test mode and normal operating mode. The timing requirements for a block can be quite different for different operating modes. In these cases, you need to extract a separate model for each mode.

Instead of keeping and using multiple extracted models, you can optionally merge them into a single 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.

The `merge_models` command merges multiple .db timing models into a single model. The generated model has moded timing arcs such that, in any given mode, the static timing behavior of the model is equivalent to one of the original models that was merged. The final merged model is saved in the specified file. PrimeTime and Design Compiler recognize the merged model with moded timing arcs for performing timing analysis.

You can only use the `extract_model` command to merge timing models generated by the NanoTime tool.

Using the merge_models Command

In the `merge_models` command, you specify the .db files of the models to be merged, the names of the modes corresponding to those models, the name of the new model being generated, an optional mode group name, and an optional tolerance value.

The `-tolerance` option is an absolute tolerance value that 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 is 40 ps and the minimum value is 1 ps.

When you merge models with a wide range of delay values, an absolute tolerance value might not be appropriate. You can use the `-percent_tolerance` option to define the maximum allowable percentage difference between two timing arcs. NanoTime considers both the `-tolerance` option and the `-percent_tolerance` option when comparing timing arcs.

Some tools that accept NanoTime models cannot handle more than one mode per timing arc (although PrimeTime can). To generate a merged model that can work with all tools, use the `-single_mode` option of 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.

Conditional Statements in Models

Timing models can have `when` and `sdf_cond` statements for all of the relevant constructs, such as delay arcs and pin based CCS receiver and noise models. If you use the `-when` option of the `extract_model` command, NanoTime evaluates the logic settings specified by the `set_case_analysis` command and generates the conditional statements in the model. Cases are mutually exclusive unless the `-exclude_from_when` option is used with the `set_case_analysis` command.

Models without the `when` and `sdf_cond` statements can be merged with models that do contain these statements. Note, however, that you must manually select the mode for a merged model that does not contain logic conditions. You need to ensure that logic conditions for the modes that are being merged are mutually exclusive.

If conditional statements are present in the models because of the use of the `-when` option of the `extract_model` command, then each mode has a condition associated with it in the modes group. Otherwise, only the modes with associated logic conditions have `when` statements in the mode definition.

The Library Compiler tool requires that distinct conditional statements be used for maximum and minimum delay arcs. Due to this requirement, the use of the `-when` option of the `extract_model` command generates `retain` constructs to represent minimum delay arc data as part of the maximum delay arcs in the model. If no maximum delay data exists, then the minimum delay data is represented as a separate arc.

When CCS timing arc data is present in the models being merged, nonlinear delay model (NLDM) data is used to determine which arcs can be merged. The output consists of both NLDM and CCS data. The accuracy of the merged model is controlled by the `-tolerance` and `-single_mode` options of the `merge_models` command.

Merging Models With Multiple Power or Ground Supplies

NanoTime follows the conventions of the Library Compiler tool, which requires that all unconnected signal pins be associated with a power or ground pin. If no connection is explicit in the design, a set of precedence rules governs how the pin is associated with power or ground. For information about the connection rules, see the *Library Compiler Timing, Signal Integrity, and Power Modeling User Guide*.

To see how unconnected pins are handled for a specific design, read the .lib model into the Library Compiler tool to observe any information or warning messages. The NanoTime tool does not display these messages. To avoid using the default connection rules, ensure that all pins are connected to appropriate power or ground pins through the design netlists or models.

In addition, Library Compiler rules specify that multiple rail voltages associated with a single pin cannot differ by more than 10 percent. If CCS models are being used, the current vectors are integrated to obtain voltage values. For NLDM models, rail voltages are extrapolated values that depend on the settings of the `rc_slew_*` variables, the `rc_input_threshold_*` variables, and the `rc_output_threshold_*` variables.

If you want to use corner-specific voltages in which the voltage values differ by more than 10 percent, you must maintain separate timing models for each corner. For example, you might want to use one voltage for maximum delay analysis and another voltage for minimum delay analysis.

Boundary Parasitics

You can handle boundary parasitics in two different ways: include their effects in the model or write them into a separate parasitics file.

Library Cell Models

By default, NanoTime generates a library cell model. This type of model replaces the entire design with a single, free-standing library cell that is easily used with other tools. The effects of boundary parasitics are included in the timing arcs of the model. The model consists of the library cell in .db (compiled) format and a command file in Synopsys Design Constraints (SDC) format that applies the applicable timing exceptions, such as false paths and multicycle paths, if any. The SDC file must be applied to the library cell when it is used at a higher level of the design hierarchy.

Figure 13-1 shows the contents of a library cell model (everything inside the dashed line).

Figure 13-1 Contents of a Library Cell Model

Models With Boundary Parasitics

To generate a core cell with a Verilog wrapper and boundary parasitics, use the `-extract_boundary_parasitics` option and specify the format of the parasitic data: `spef` (gzip-compressed Standard Parasitic Exchange Format or SPEF) or `spef_unzipped` (uncompressed SPEF). This type of model maintains the boundary parasitics of the original netlist.

Coupling capacitors between pins are preserved. Coupling capacitors into the model are preserved as capacitors to ground.

Maximum and minimum delay arcs are placed on the NMOS transistor input and output pins of a full-unate model when this option is used.

Figure 13-2 Contents of a Boundary Cell Model

To create a boundary cell timing model, follow this procedure (unrelated steps are not included):

1. Read in design files and link the design.
2. Set the `rc_reduction_exclude_boundary_nets` variable to `true` to prevent the boundary parasitics from being modified by the parasitic reduction operation.

   Alternatively, you can set the `rc_reduction_max_net_delta_delay` variable to 0 to prevent all parasitic reduction.
3. Read in the parasitic data.
4. Set the `timing_save_wire_delay` variable to `true`.
5. Execute the `check_design` command.
6. Execute the `extract_model -extract_boundary_parasitics` command.
7. (Optional) Add the `-ignore_ports` option if the ignored ports do not have parasitics.
8. (Optional) Add the `-bus` option to use bus naming in the generated Verilog output.
9. (Optional) Add the `-enable_arc_segmentation` option to preserve half-unate arcs.

Full-Unate and Half-Unate Models

The boundary-parasitic models produced by NanoTime expose multiple transistor pins in the core cell model for every boundary pin in the library cell model. Because of the innate switching behavior of transistors, these core cell boundary pins tend to have only a rise or fall transition defined, naturally making the arcs half-unate.

To ensure compatibility with the PrimeTime tool, the default NanoTime model-generation flow produces ETMs with full-unate boundary pins. NanoTime can generate both full-unate and half-unate models for a design. However, it is your responsibility to ensure that you generate a model that can be consumed by your tool.

To preserve the generation of half-unate arcs in ETMs, use the `extract_model` command with the `-enable_arc_segmentation` option. Use this option for flows in which the half-unate endpoints can be used for improved accuracy in the presence of asymmetric parasitics and unequal pin capacitances.

To produce full-unate timing models, NanoTime applies the following two transformations to the timing arcs, making the endpoints full-unate:

1. Maximum arcs are moved to PMOS pins, which usually have larger capacitances. Minimum arcs are moved to NMOS pins, which usually have smaller capacitances. This transformation assumes symmetric parasitics for the input and output nets.
2. Arc look-up tables are duplicated if the previous endpoint relocation alone does not sufficiently ensure rise and fall transitions for every boundary pin. Such arcs are marked with a special comment tag in the .lib file.

During full-unate model generation, NanoTime captures a single representative transistor pin in the timing model (Liberty and Verilog wrapper) when multiple transistors are in parallel. An example is when finger devices exist at the boundary port. In this case, the tool moves the device capacitances associated with the terminals of multiple parallel transistors or with the fingers belonging to the representative transistor to the boundary parasitics file and generates an MXTR-020 warning message for each instance.

The diagram in Figure 13-3 shows that the arcs are half-unate before endpoint relocation and full-unate after relocation. Note that the boundary output pins are marked as tristate since the net is multidriven.

Figure 13-3 Half-Unate Arcs Before and After Endpoint Relocation

Composite Current Source Models

Composite current source (CCS) models are more accurate than nonlinear delay models for advanced process technologies due to their ability to handle high-impedance nets and other nonmonotonic behavior. A CCS timing model represents the delays through the design block. A CCS noise model simulates the noise response by applying ramps and glitches to the inputs of the CCS timing model to derive the dynamic behavior of the design.

You must have a NanoTime Ultra license to create CCS models. In addition, you must have one or more HSPICE licenses to create CCS noise models.

Figure 13-1 summarizes how the NanoTime tools supports CCS models.

Table 13-1 NanoTime Support of CCS Models

CCS Model Type	Read and Use Model	Create Model
Timing	yes	yes
Noise	no	yes
Power	no	no

Figure 13-4 illustrates the structure of a CCS timing model. A CCS driver model uses a time- and voltage-dependent current source. The CCS driver lookup table is indexed by the input transition time and the output load capacitance, similar to a NLDM model. However, the values in the CCS model lookup table represent the driver output current, which is used for a more accurate timing analysis than the simple delay table.

In addition, the CCS model can include the time dimension, allowing a different current-versus-time model to be used for each combination of input transition time and output load capacitance.

Figure 13-4 CCS Timing Driver and Receiver Models

The CCS timing receiver model uses two different capacitor values rather than a single lumped capacitance. The first capacitance is used as the load up to the input delay threshold. When the input waveform reaches this threshold, the load is dynamically adjusted to the second capacitance value.

See Also

- Using CCS Timing Models

Creating CCS Timing Models

By default, the `extract_model` command produces a library cell using nonlinear delay model (NLDM) modeling only. To create a CCS timing model, use the `-library_elements` option and specify `{nldm ccs_timing}` as the list of element types, which generates both NLDM and CCS models.

An example of the command is as follows:

```
nt_shell> extract_model -name DMA -library_elements {nldm ccs_timing}
```

The generated CCS timing model is valid only for the single operating condition in effect at the time of model extraction. For modeling of different worst-case operating conditions, it is necessary to generate a separate model for each condition. Only input ports that have paths starting from them are given receiver models in the generated timing model.

CCS model generation is limited when dynamic simulation is used. When dynamic clock simulation is used, CCS receiver and driver models are not generated for the clock input pins and clock output pins. When dynamic delay simulation is used on the boundary of the model being extracted, CCS receiver and driver models are not generated for the input pins and output pins driven by dynamic delay simulation regions.

Driver Waveforms for Models

By default, NLDM models use linear ramp waveforms for all nets not specified for nonlinear waveform modeling. However, CCS models use the more accurate Synopsys predriver by default. When CCS models are specified in the `extract_model` command, both the NLDM and CCS models use the Synopsys predriver for input waveforms. As a result, an NLDM model generated without a CCS model might exhibit different delays than an NLDM model generated at the same time as a CCS model.

To maintain the same delays for NLDM models generated in both circumstances, you can use any of the following strategies:

- Use the ramp driver for CCS models and NLDM models created at the same time by setting the `model_ccs_characterization_driver_type` variable to `ramp`.
- Set the `sim_snps_predriver_mix_ratio` variable to 1, which sets the Synopsys predriver to a linear ramp, equivalent to that used in the NLDM models.
- Use the Synopsys predriver for all path tracing and all NLDM model creation by setting the `sim_use_snps_predriver` variable to `true` while keeping the `model_ccs_characterization_driver_type` variable set to its default of `snps_predriver`.

  In this case, the `sim_snps_predriver_mix_ratio` variable controls the waveforms for both types of models. You can use the default mixture of 0.5, specify ramp waveforms with a value of 1, or specify exponential waveforms with a value of 0.

The following variables affect the driver waveforms used for models:

- `model_ccs_measured_delay_tolerance`

  The acceptable difference between the measured delay from simulation and the delay obtained from the CCS waveform, expressed as a fraction of the measured delay from simulation. The default is 0.02.

- `model_ccs_waveform_segment_tolerance`

  The maximum allowed voltage difference between the simulation waveform and the CCS waveform used for selecting the CCS model point. The smaller the tolerance, the more points are used in waveform tables. The default is 0.005.

- `model_ccs_characterization_driver_type`

  The type of driver to be used at the cell inputs for characterization. It can be set to either `ramp` (for a simple ramp) or `snps_predriver` (for the standard Synopsys predriver). The default is `snps_predriver`.

- `sim_snps_predriver_mix_ratio`

  The properties of the Synopsys predriver, which is a mixture of linear and exponential behavior. A value of 0 produces an exponential waveform, while a value of 1 produces a linear ramp.

Creating CCS Noise Models

NanoTime uses composite current source (CCS) noise models to capture the noise information of digital circuits and to perform signal integrity noise analysis. To generate CCS noise models, NanoTime uses HSPICE analysis instead of the internal simulator.

A CCS noise model is a structural boundary model that describes

- The first channel-connected block (CCB) driven by the cell input
- The last channel-connected block driving the cell output

The CCS noise information is stored on either a pin or a timing arc. It consists of characterization data, which enables noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell. NanoTime supports pin-based CCS noise models.

Before generating a CCS noise model, you must set the `si_enable_noise_analysis` variable to `true`.

To generate the noise models, use the `-library_elements` option with the `extract_model` command and specify `{nldm ccs_timing ccs_noise}` as the list of element types. The `ccs_noise` element type cannot be used alone; all three element types are required.

For example, the following command generates NLDM timing, CCS timing, and CCS noise models:

```
nt_shell> extract_model -library_elements {nldm ccs_timing ccs_noise}
```

If the design consists of transistors and small timing models, NanoTime does not generate CCS noise models for the pins that are connected to the small timing models.

The `extract_model -library_elements {nldm ccs_timing ccs_noise}` command launches multiple HSPICE simulations during model extraction. If the simulations are successful, NanoTime removes all the intermediate SPICE files. To save the simulation files, use the `sim_file_cleanup` variable.

The HSPICE simulations are launched using the command specified in the `sim_hspice_cmd` variable. By default, the variable is set to the following string:

```
hspice -i <input_deck> -o <listing_file>
```

Note:

You can change the command that launches the simulation tool in your computing environment. However, the `<input_deck>` and `<listing_file>` tags are syntax terms that the NanoTime tool uses as variables to represent the correct input and output files. You must use these tags exactly as written, including the angle brackets and the wording inside the brackets.

Accuracy Considerations for Creating CCS Models

Observe the following recommendations to achieve the best accuracy when you are creating CCS models:

- Use nonlinear waveform analysis with the default mode of `accurate` for the entire design.
- Use a value of `1uA` for the `lib_current_unit` variable.

  Using a large unit for current might cause small values to lose precision. For example, a current value of 0.23759 mA would become 237.59 uA if the current unit specified in the `lib_current_unit` variable is set to `1uA`. In this case, no information is lost. However, if the specified current unit is set to `1A`, the value becomes 0.000238 A, losing some digits of precision.

HSPICE Recalibration

To produce timing models that meet HSPICE accuracy, specify the `-hspice_timing` option of the `extract_model` command. With this feature, the NanoTime tool first performs a conventional timing analysis, then runs HSPICE simulations on the model paths to improve the accuracy of the calculated delays. You must have a NanoTime Ultra license to use this feature.

NanoTime uses the `write_spice` command to generate the input files for HSPICE simulation. The methodology and limitations of the `write_spice` command also apply to HSPICE recalibration.

Usage notes are as follows:

- Paths that go through other timing models are not included.
- Analysis options that require dynamic updates during or after simulation are not fully covered. These analysis types include latch error recovery and delay and transition coefficient multipliers. The affected paths are simulated with HSPICE, but the analysis adjustments are not applied.
- Signal integrity analysis might not be accurate because the SPICE deck cannot properly capture the alignment of aggressor nets to victim nets.
- Transparency windows are not updated.
- If either dynamic clock simulation or dynamic delay simulation is enabled, the delays calculated during dynamic analysis are used for any paths or parts of paths that go through the dynamic simulation region.
- Setup and hold calculations might be different from those obtained from standard cell characterization tools. NanoTime uses the difference between two arrival edges reaching the logic level threshold (typically 50 percent), but other tools might use more complex calculations.

If you specify the `-hspice_timing` option, the model file provides an indication if NanoTime does not use HSPICE simulation for some of the model paths. The tool marks each affected path ID with a tilde character (~) in the comment line. For example, if path 26 contains a timing model, it is not analyzed with HSPICE. In this case, the comment line appears as follows:

```
/* comment : reference path 13 (11), checked path 26~ (25),
   reference path 14 (12), checked path 28 (27); */
```

See Also

- Chapter 11, “Correlation with HSPICE Analysis”

Generating Timing Models With HSPICE Simulations

Figure 13-5 shows the process NanoTime uses to generate timing models with the HSPICE tool when you use the `-hspice_timing` option.

Figure 13-5 Generating Timing Models With the HSPICE Simulator

When you use the `extract_model -hspice_timing` command, the NanoTime tool launches HSPICE simulations using the command specified in the `sim_hspice_cmd` variable. By default, the variable is set to the following string:

```
hspice -i <input_deck> -o <listing_file>
```

Note:

You can change the command that launches the simulation tool in your computing environment. However, the `<input_deck>` and `<listing_file>` tags are syntax terms that the NanoTime tool uses as variables to represent the correct input and output files. You must use these tags exactly as written, including the angle brackets and the wording inside the brackets.

By default, the `extract_model -hspice_timing` command analyzes the complete set of paths included in the timing model. To reduce runtime, you can optionally use the `set_hspice_timing_model_paths` command to reduce the number of paths to be analyzed with the HSPICE tool.

You must specify the paths before executing the `extract_model` command. To reset the path settings, use the `remove_hspice_timing_model_paths` command.

HSPICE Simulation Directories

When you initially execute the `extract_model -hspice_timing` command, the command creates an HSPICE simulation directory named hspice_timing under the nt_sim_data directory and a C0000 subdirectory under the hspice_timing directory. Each subsequent HSPICE simulation creates a new directory that is named by appending the -n suffix to C0000, such as C0000-1. The C0000-n directory contains subdirectories named for the ports of the generated timing models. Each port subdirectory includes the SPICE decks, simulation files, and simulation results associated with that port. Each port directory also contains all paths starting from that port and is indexed by the path_id. Figure 13-6 shows the simulation directory structure.

Figure 13-6 HSPICE Simulation Directory Structure

You can specify whether to keep the directories after the simulations by setting the `sim_file_cleanup` variable. Table 13-2 shows the available settings.

Table 13-2 Settings of the sim_file_cleanup Variable

Value	Operation
0	Keep all simulation files.
1	Keep all files in gzip-compressed format.
2	Keep only SPICE decks.
3	Remove all simulation files.

Table 13-2 Settings of the sim_file_cleanup Variable (Continued)

Value	Operation
4	Keep only files of failed simulations. This is the default.
5	Keep only SPICE decks in gzip-compressed format.
6	Keep only files of failed simulations in gzip-compressed format.

Creating SPICE Decks

When you run the `extract_model -hspice_timing` command, the command automatically creates SPICE decks for HSPICE simulations. Using data-driven analysis, the command includes the inline `.DATA` statement in the SPICE decks to bypass reading in the netlist and the settings for multiple simulations. The `.DATA` statement associates a list of parameter names with corresponding values in an array. HSPICE repeats the required simulations as an internal loop by varying slew and load for parameters and values defined in each `.DATA` statement.

For example, the following inline `.DATA` statement uses the datanm data name, the pnam1 parameter name, and the pval1 parameter value.

```
.DATA datanm pnam1 <pnam2 pnam3 ... pnamxxx >
+ pval1 <pval2 pval3 ... pvalxxx>
+ pval1’ <pval2’ pval3’ ... pvalxxx’>
.ENDDATA
```

The simulation results for each `.DATA` statement are stored in the .mt and .tr files.

The default of the `model_ccs_characterization_driver_type` variable applies during the simulations. The default is `pre_snpsdriver.`

To deliver better performance and reduce runtime, use the evaluated model cards for the HSPICE simulations rather than the original model files.

Speeding Up HSPICE Simulations

The flow of generating timing models with HSPICE accuracy requires many successive simulations. Therefore, the flow consumes longer runtime than the comparable flow of using the internal simulator. To reduce the runtime, NanoTime launches HSPICE simulations in advanced client/server mode. Starting the advanced client/server mode checks out one HSPICE license to do multiple simulations in sequence. In addition, HSPICE reads in the common files only one time in multiple simulations with different circuits.

If a distributed processing environment is available, HSPICE simulations can run in parallel via the common distributed processing library. For example, NanoTime might submit 10 simulation jobs to machine A and 5 simulation jobs to machine B. Each machine runs HSPICE in advanced client/server mode.

See Also

- Distributed Processing

Context Characterization

Context characterization can be used to perform timing analysis of a lower-level subdesign while observing the chip-level timing constraints. This is the general procedure:

1. Read the top-level design into NanoTime and apply the top-level timing constraints.
2. Characterize the timing context for the subdesign with the `characterize_context` command.
3. Generate a timing constraint script in Synopsys Design Constraints (SDC) format with the `write_context` command.
4. Read the subdesign into NanoTime.
5. Source the SDC script generated in step 3.
6. Analyze the timing of the subdesign.

Context characterization captures the timing context of instances of subdesigns in the top-level timing environment. The timing context of an instance includes clock waveform information, input arrival times, output required times, timing exceptions, logic constraints, and capacitive loads.

After the context of a subdesign has been characterized, you use the `write_context` command to write the timing context information as a script in SDC format.

For example, the following command derives the timing-related context information for instance I1:

```
nt_shell> characterize_context {I1}
```

After completion of context characterization, the following command writes the context information to a Synopsys Design Constraints (SDC) file called I1.ntsh in the current directory:

```
nt_shell> write_context -derive_file_name {I1}
```

The `characterize_context` command is a path-tracing process like the `trace_paths` command. It cannot be used after the `trace_paths` command. If you have already run the `trace_paths` command, you must first reset the design using the `reset_design` command and restart the analysis from the design linking stage. After you run the `check_design` command, you can then use the `characterize_context` command.

The characterize_context Command

The `characterize_context` command is a path-tracing process. Like the `trace_paths` command, it can be used only after the `check_design` command has been run. It cannot be used after the `trace_paths` command.

The `characterize_context` command must specify the name of at least one cell instance in the design. The contexts of the listed cells are characterized.

Use the `-max_only` or `-min_only` option to perform context characterization for only maximum-delay or only minimum-delay paths. Without these options, the command performs path tracing and context characterization for both types of paths.

The `-full_path_enumeration` option performs full path enumeration, without pruning. Due to the large runtime requirements, this option is practical only with very small designs and when used with the `set_find_path` command to limit the scope of the design being considered for characterization.

The `-npaths` option specifies the maximum number of paths to save in the path database for each startpoint-endpoint pair. The default is 1. Specify a larger number to save more paths, but use a value that is small enough to keep the runtime and memory usage reasonable. Using this option alone saves more paths at the endpoint, if that number of paths exists. However, they are not necessarily the worst paths, because the second worst path might have been pruned under the top worst path.

The `-keep_paths_within` option keeps timing paths in the path database that have a path delay close to the worst path delay of each startpoint-endpoint pair. By default, the `-keep_paths_within` option is set to 0.0, and NanoTime keeps only the single worst path per startpoint-endpoint pair. To keep more paths, set the `-keep_paths_within` option to a time value larger than 0.0, and set the `-npaths` option to a number larger than 1. This puts more timing information in the path database but also increases the runtime and memory usage. The `-full_path_enumeration` option is equivalent to the `-keep_paths_within` option set to a very large time value.

The `-pbsa` option invokes path-based slack adjustment, which adjusts the calculated path delays based on the lengths of the path segments traced in the timing check.

The `-debug_paths` option writes a file to the working directory that lists the specific worst-case timing paths in the design that are being used to generate the boundary constraints. This information is useful for debugging purposes.

The write_context Command

The `write_context` command writes the timing context of the specified instances as script in Synopsys Design Constraints (SDC) format. When you execute the SDC commands on the subdesign, the timing analysis tool initializes its timing environment to the information characterized from the context.

The command must specify the name of at least one cell instance in the design. The contexts of the listed cells are written to SDC files. The context of a cell can be written only after context characterization has been performed with the `characterize_context` command.

The `-timing` option causes the command to write only the timing information such as clocks, input delays, output delays, and timing exceptions. By default, all context information is written. Similarly, the `-environment` option causes the command to write only the environment-related information such as operating conditions and capacitive loads on input and output pins. The `-constant_inputs` option causes the command to write only the logic constants on input pins of the characterized instance.

The `-debug_paths` option writes a file to the working directory that lists the specific worst-case timing paths in the design that were used to generate the boundary constraints. This information can be used for debugging purposes.

The `-output` option specifies the name of the generated script file, whereas the `-derive_file_name` option derives the name of the generated script file from the instance name. You can use either `-output` or `-derive_file_name`, but not both. If you use neither, the script is written to standard output.

If you use the `-derive_file_name` option, you can also use the `-prefix`, `-extension`, `-script_directory`, and `-separator` options. The `-prefix` option specifies the prefix for the derived constraint file name. The `-extension` option specifies the extension for the derived constraint file name. If no extension is specified, the extension is ntsh. The `-separator` option specifies a single character to use as a hierarchical separator in constructing the derived constraint file name generated for each instance. If this option is not used, the default is the “at” sign (@). You cannot use the slash (/) or backslash (\) characters as name separators. The `-script_directory` option specifies the directory in which the constraint file is written. The default is the current directory.

14

Dynamic Simulation

Dynamic simulation is an optional feature that accurately calculates the behavior of complex portions of the design by using full circuit simulation.

This chapter contains the following sections:

- Overview of Dynamic Simulation
- Dynamic Clock Simulation
- Dynamic Delay Simulation

Overview of Dynamic Simulation

Standard NanoTime analysis uses a fast internal simulator to analyze the design. In some cases, a part of the design might be too complex for the internal simulator to get sufficiently accurate results. The NanoTime dynamic simulation feature analyzes a design region as a single unit to produce more accurate results, at the cost of additional runtime. Dynamic simulation fully accounts for interacting signals throughout the specified part of the design.

When you invoke dynamic simulation, NanoTime sets up a simulation of the designated part of the design, runs the simulation, and uses the results to represent the region during timing analysis of the rest of the design. The dynamic simulation applications are as follows:

- Dynamic clock simulation (DCS): NanoTime simulates the full clock network to determine the arrival times of clock signals at sequential elements.
- Dynamic delay simulation (DDS): NanoTime simulates a specified portion of the design using test vectors that the tool generates automatically or that you specify explicitly.

The Synopsys FineSim tool is the simulator for all dynamic simulations.

Important:
Do not change any settings of the FineSim tool.

General Dynamic Simulation Guidelines

To retain intermediate simulation files, set the `sim_finesim_keep_run_files` variable to `true`. These files can be used to run the FineSim tool on a standalone basis for the purpose of investigating setup issues or design errors. The files cannot be used to test path delay or transition time accuracy.

If you use TMI2 transistor models, you must provide a SPICE header file. The header file for TMI2 models must contain a .TEMP statement and the absolute paths to the models (not relative paths). No other SPICE statements are allowed in the file. Specify the header file name as follows:

- For dynamic clock simulation, use the `dcs_spice_header` variable.
- For dynamic delay simulation, use the `set_simulation_attributes` command with the `-spice_header` option.

If you use other transistor models, do not include a SPICE header file.

Dynamic Clock Simulation

The default internal simulator traces only one path at a time and does not retain logic state information from one path tracing iteration to the next. The simulator can handle clock trees consisting of simple buffers, multi-input gates, and pass gates by tracing one path at a time. However, if a clock generation circuit uses feedback loops, multiple paths, simultaneous switches, latches, flip-flops, clock edge chopping, or clock merging, the internal simulator might not be able to generate the correct clock waveforms. For these cases, you can analyze the entire clock circuit by using dynamic clock simulation.

Note:
Dynamic clock simulation does not work with multicorner analysis.

This section contains the following topics:

- Principles of Dynamic Clock Simulation
- Dynamic Clock Simulation Setup Procedure
- Side Input Logic
- Shared Pullup and Pulldown Transistors
- Self-Timed Circuits
- Signal Integrity Analysis for Dynamic Clock Simulation Regions
- Reporting Clock Arrivals for Dynamic Clock Simulation Regions
- Dynamic Clock Simulation Variables

Principles of Dynamic Clock Simulation

To use dynamic clock simulation on a straightforward clock tree with no dividers or multipliers, set the `dcs_enable_analysis` variable to `true`. This causes the clock tree to be simulated dynamically. By default, the simulation spans five clock cycles to ensure a stable result.

When using dynamic clock simulation on a clock tree with dividers or multipliers, you must make sure that all elements of the dividers are marked as clocks, using the `mark_clock_network` command with the `-force_propagation` option where necessary. Only the clock network is simulated dynamically, so all elements necessary for correct clock operation must be included in the network. For example, consider the divide-by-4 clock circuit in Figure 14-1.

Figure 14-1 Divide-by-4 Clock Circuit

Even if only div4 is used as a clock and not div2, div2 still must be marked as a clock for proper simulation of FF2. The output of each clock divider or multiplier must be declared to be a clock.

The commands for proper operation of this divider are:

```
nt_shell> mark_clock_network -force_propagation div2
...
nt_shell> create_generated_clock -name div4 \
          -source clkin -divide_by 4 FF2/MNQ/D
```

Instead of using the `mark_clock_network` command for div2, you could use another `create_generated_clock` command.

Because div2 and div4 are marked or defined as clocks, the clock attribute is propagated through the combinational logic so that the simulated clock network includes the inverters.

If the clock network contains latches along the control or reset paths that must be toggled to create the clock waveform, the latches also must be marked as clocks to be included in the clock network. Any side input control logic must be set to enable the clock waveforms. If a side input is a control signal that selects one of multiple operating frequencies, you must use the `set_case_analysis` command to place the circuit into a state that selects one operating frequency. The `trace_paths` command can use only one operating frequency.

Basic dynamic clock simulation produces one delay value for each path through the simulation region. This value is used in all timing checks that involve the path through the simulation region. However, if signal integrity analysis is enabled, both a minimum delay clock path and a maximum delay clock path are generated.

Dynamic Clock Simulation Setup Procedure

This procedure provides a basic setup for dynamic clock simulation.

1. Define the clock sources and clock topology structures.
2. Set the `dcs_enable_analysis` variable to `true` to enable dynamic clock simulation.
3. If you are using TMI2 transistor models, prepare the SPICE header file. Set the `dcs_spice_header` variable to point to the file.
4. If you are using signal integrity analysis but you want to disable SI analysis inside the dynamic clock simulation regions, set the `dcs_enable_si_analysis` variable to `false`.
5. If you are using signal integrity analysis and you want to consider timing windows and run the dynamic simulation during every signal integrity iteration, set the following variables:
   - Set the `timing_save_pin_arrival_and_transition` variable to `true`
   - Set the `dcs_enable_detailed_path_reporting` variable to `true`
   - Set the `dcs_enable_si_analysis_first_iteration_only` variable to `false`
6. (Optional) Exclude transistors and their fanouts from DCS by using the `mark_instance` command with the `-exclude_from_dcs` option.
7. (Optional) Terminate DCS in specific paths by using the `mark_clock_network` command with the `-end_dcs` option.
8. (Optional) Save information about clock arrivals in the critical path through the DCS region by setting the `dcs_enable_detailed_path_reporting` variable to `true`.
9. (Optional) Save information about clock arrivals at all timing points inside the DCS region by setting the `dcs_enable_detailed_path_reporting_complete` variable to `true`. (This setting results in increased runtime and memory consumption.)
10. Check the design using the `check_topology` and `check_design` commands.
11. If DCS-001 warning messages occur, address the side input warnings by setting logic on the side inputs with the `set_dcs_input` command. For self-timed circuits, include the `-synchronized_data` option. Run the `check_design` command again to obtain the corrected simulation.
12. Use the `report_clock_arrivals` command to verify that all necessary nodes have been marked as clocks. The report lists the nodes that have been marked as clocks, even though no timing analysis has been run yet.
13. If the nodes are correctly marked, proceed to step 15.

14. If some clock nets are not marked as clocks, mark them manually by using the `mark_clock_network` command with the `-force_propagation` option.

    An alternative approach is to create a generated clock on the clock net with the `create_generated_clock` command.

15. If there are nets incorrectly marked as clocks, stop clock propagation through them by using the `mark_clock_network` command with the `-stop_propagation` or `-end_dcs` options. In this case, execute a `reset_design` command and go back to step 6.

16. Execute the `trace_paths -clock_only` command to perform dynamic clock simulation and generate timing data for the entire clock tree.

17. Use the `report_clock_arrivals` command to verify the clock network simulation results. Compare the new results against the baseline results from the original analysis.

18. Reset the design using the `reset_design -paths` command, then run a full-chip analysis using the `trace_paths` command.

Side Input Logic

When setting up dynamic clock simulation regions, NanoTime sets nonclock side inputs to fixed logic states in these situations:

- A case analysis setting propagates to the side input.
- The side input is the clock gate enable signal (the input is set to the state that enables the clock gate).
- The side input state is specified with the `set_dcs_input` command.

If NanoTime cannot determine the correct logic state for a side input, it creates a waveform to simulate both high and low logic levels at that input. The tool issues a DCS-001 warning message whenever it generates a side input waveform.

If there are multiple side inputs, NanoTime generates waveforms to cover all possible combinations of side input logic states. This might cause long runtimes or inaccurate results from trying logically impossible side input combinations, so it is preferable to define the side input state.

When you use case analysis to set the side input, dynamic clock simulation is enabled; however, path tracing through the side input is disabled and the related timing checks are disabled. To avoid these limitations, use the `set_dcs_input` command to set the side input state. Dynamic clock simulation logic definition does not affect logic on other nets. The syntax of the command is as follows:

```
set_dcs_input -logic value pin_list
```

The Boolean variable *value* can be 0 or 1. The *pin_list* argument must be a list of nets, pins, or ports. To remove previously set DCS input logic settings, use the `remove_dcs_input` command with *pin_list* as the only argument.

The `dcs_input_logic_state` attribute for a net contains the logic state 0 or 1 if it was set by the `set_dcs_input` command; otherwise, it contains the value `X`.

During dynamic clock simulation, logic set by the `set_dcs_input` command takes precedence over logic set by case analysis.

Shared Pullup and Pulldown Transistors

Some circuits share a common pullup transistor and a common pulldown transistor for replicated structures that are only used one at a time. Therefore the common pullup and pulldown transistors are sized to be able to drive only part of the connected load adequately.

During dynamic clock simulation, if all of the load nets are active, the load capacitance is very high and consequently the simulated delays are unrealistically long.

There are two ways to ensure the correct operation for this situation:

- Specify the shared pullup or pulldown transistor by using the `mark_instance` command with the `-exclude_from_dcs` option. The argument of the option is the name of the transistor. You can cancel the assignment with the `erase_instance` command by using the same option and argument.

  NanoTime excludes the specified transistor, all other transistors in the same channel-connected block, and its transitive fanout from dynamic clock simulation. For this region, NanoTime employs the default delay calculation methods for path tracing.

  If you choose this approach, you might also need to mark other clock gates or set constraints to obtain correct analysis of parts of the circuit that are no longer part of the clock simulation region.

- Terminate dynamic clock simulation at internal pins or nets by using the `mark_clock_network` command with the `-end_dcs` option. The argument of the option is the input to the channel-connected block which contains the shared transistor. You can cancel the assignment with the `erase_clock_network` command by using the same option and argument.

Self-Timed Circuits

A self-timed circuit uses a derivative of its own output signal to feed back as an input. A general example is shown in Figure 14-2. This design style might be used to turn off an output signal at a specific time after it was generated. For example, a sense amplifier enable signal is generated by a self-timed circuit in some memory designs.

The self-timed signal input to the clock network is a side input of the dynamic clock simulation region. The correct waveform for this signal can only be obtained from path tracing, which occurs after the clock signal is acquired. Therefore you must perform a second iteration of dynamic clock simulation to correctly model this feedback circuit.

Figure 14-2 Self-Timed Circuit Feedback Loop

You must identify the clock network side input signal (signal B in Figure 14-2) by using the `set_dcs_input` command with the `-synchronized_data` option. The argument to the option is either a net name or a pin name; it must not be a clock net, but rather the input of a channel-connected block that has a clock net output.

The general flow is as follows:

1. Perform the original dynamic clock simulation run and treat the timed signal (B) as a side input with fixed logic.
2. Perform a partial path tracing operation from the self-timed circuit output (A) to the timed signal (B) to find the arrival times for this signal.

See Also

- Dynamic Clock Simulation for Memory Analysis

Signal Integrity Analysis for Dynamic Clock Simulation Regions

Signal integrity analysis is an optional NanoTime feature that analyzes how signals in one path affect the delays in nearby paths.

By default, dynamic clock simulation includes the effects of parasitic capacitances that reside completely within the simulation region, even if global signal integrity analysis is not enabled. When signal integrity analysis is enabled, the additional effects of capacitive coupling between nets inside the dynamic clock simulation region and nets outside the region are included, resulting in more accurate delays through the DCS region.

Signal integrity analysis within dynamic clock simulation regions is enabled by default, but does not apply unless you enable global signal integrity analysis by setting the `si_enable_analysis` variable to `true`. If you want to disable signal integrity analysis within dynamic clock simulation regions for compatibility with earlier NanoTime versions, set the `dcs_enable_si_analysis` variable to `false`.

To reduce runtime, only the logic considerations from the first iteration of signal integrity analysis are used as input for the dynamic clock simulation. Subsequent iterations of signal integrity analysis reuse the same delays.

To improve the accuracy at the cost of more runtime, you can elect to run the dynamic clock simulation during every signal integrity iteration by following these steps:

- Set the `timing_save_pin_arrival_and_transition` variable to `true`.

  This variable saves timing information inside the dynamic simulation region that is needed to determine signal integrity timing windows.

- Set the `dcs_enable_detailed_path_reporting` variable to `true`.

  This variable saves additional timing information inside the dynamic simulation region.

- Set the `dcs_enable_si_analysis_first_iteration_only` variable to `false`.

  This variable, when set to `false`, runs the dynamic simulation during every signal integrity iteration.

Reporting Clock Arrivals for Dynamic Clock Simulation Regions

The `report_clock_arrivals` command reports the arrival times of clock signals at sequential device pins or along intermediate points of a full clock tree. You can run the `report_clock_arrivals` command any time after the `check_design` command to view a list of the clock nets. However, to see the actual arrival times, you must run the `report_clock_arrivals` command after the `trace_paths` command.

The report lists the arrival times for clock nets in the design, including intermediate nets on clock paths at the level of channel-connected blocks.

However, by default, NanoTime does not save timing information at intermediate nets inside a dynamic clock simulation region. Instead, the DCS region is represented as a single timing arc from the input to the output.

You can report additional clock information in the DCS region by using the following variables. Both variables must be set before the `check_design` command. Both variables require a NanoTime Ultra license.

- The `dcs_enable_detailed_path_reporting` variable

  Set this variable to `true` to report the critical path through the DCS region as a sequence of arcs traversing internal timing points. The default is `false`.

  Setting this variable to `true` is required when you are performing multiple iterations of signal integrity analysis. In addition, setting the variable to `true` improves the ability of the path-based slack adjustment (PBSA) feature to identify the common point for pessimism reduction.

- The `dcs_enable_detailed_path_reporting_complete` variable

  Set this variable to `true` to report the clock arrivals on all internal nodes of the DCS region. The default is `false`. This variable has an effect only if you set the `dcs_enable_detailed_path_reporting` variable to `true`.

  Setting this variable to `true` might require many additional dynamic simulations to obtain the timing information at the internal DCS region nodes, resulting in increased runtime and memory consumption.

The use of detailed internal clock paths does not affect the total delay across the dynamic clock simulation region. However, path-based slack adjustments might be different due to the increased number of path levels and a more accurate common point determination. In addition, signal integrity analysis results might be affected due to the existence of more clock nets.

Note:

To report detailed clock paths, NanoTime must perform standard path tracing within the dynamic clock simulation region. However, the DCS delay is determined by the dynamic simulator and not the path tracing operation.

The tool might issue warning or error messages related to path tracing within the DCS region. In this case, the dynamic simulation delay is still accurate because it is not derived from the optional detailed path reporting step. If the errors or warnings disappear when detailed path reporting is disabled, the DCS delay is accurate.

Dynamic Clock Simulation Variables

The following variables control dynamic clock simulation:

The `dcs_enable_analysis` variable

Enables dynamic clock simulation. The default is `false`.

The `dcs_enable_internal_coupling_caps` variable

Enables the recognition of coupling capacitors that are internal to the dynamic clock simulation region. When this variable is set to `true` (the default), NanoTime takes coupling capacitances into account when calculating delays. Otherwise, NanoTime splits the capacitance into two separate grounded capacitors.

The `dcs_enable_network_reduction` variable

Allows elimination of inverters at the clock network outputs to reduce runtime.

The `dcs_spice_header` variable

Specifies the name of the SPICE header file included at the head of the SPICE-format netlist sent to the dynamic simulator.

The `dcs_number_of_cycles` variable

Specifies the number of reference clock cycles to simulate. This value is multiplied by the period of the reference clock to get the simulation time span. The default is 5, which results in the simulation of five cycles of the reference clock.

The `dcs_enable_detailed_path_reporting` variable

Enables detailed clock path reports for critical paths.

The `dcs_enable_detailed_path_reporting_complete` variable

Enables clock arrivals on all nodes in the dynamic simulation region, at a cost of runtime and memory.

The `dcs_enable_si_analysis` variable

Enables signal integrity analysis within dynamic clock simulation regions. The default is `true`. This variable has an effect only if the `si_enable_analysis` variable is `true`.

The `dcs_enable_si_analysis_first_iteration_only` variable

Specifies whether to run dynamic simulation using only the first iteration logic considerations, to save runtime. The default is `true`. Setting the variable to `false` runs dynamic simulation during every signal integrity iteration. This variable has an effect only if the `si_enable_analysis` and `dcs_enable_si_analysis` variables are `true`.

Dynamic Delay Simulation

Dynamic delay simulation can provide more accurate delays when multiple signals in a data path have complex interactions. An example application is a stage with multiple inputs that switch at the same time, or nearly the same time.

To use dynamic delay simulation, you mark the parts of the design to be simulated together. You can optionally specify a set of test vectors to reduce the simulation time or to specify simultaneous switching of different inputs. Otherwise, NanoTime generates its own comprehensive set of test vectors for simulation.

During path tracing, when NanoTime encounters a dynamic delay simulation region, the tool runs the dynamic simulator on those circuits and subsequently uses the resulting delay to represent the entire simulation region.

The increased accuracy comes at the cost of increased runtime. NanoTime features such as signal integrity analysis do not apply within the dynamic simulation region.

This section contains the following topics:

- Dynamic Delay Simulation Procedure
- Marking Dynamic Simulation Regions
- Reporting Dynamic Simulation Regions
- Dynamic Delay Simulation Vector Files
- Multivoltage Analysis Using Dynamic Simulation
- Dynamic Delay Simulation Commands

Dynamic Delay Simulation Procedure

An analysis session using dynamic delay simulation typically consists of the following steps:

1. Prepare the SPICE header file if you are using TMI2 transistor models. A header file is optional for other configurations. Point to the file by using the `set_simulation_attributes` command with the `-spice_header` option.
2. Read in and link the design, specify the topology, and run the `check_topology` command in the usual manner.
3. Mark the regions that must be simulated dynamically by using the `mark_simulation` command. Run the `check_design` command.
4. Report or change the simulation setup using commands such as the following:

   ```
   sizeof_collection [all_simulations]
   set sim [get_simulation -input in]
   report_simulation
   report_simulation $sim
   report_simulation $options $sim
   set_simulation_attributes $sim
   ```

5. If necessary, generate simulation vector template files by using a script similar to the following:

   ```
   foreach_in_collection sim \
   [all_simulations] {
     set outnet [index_collection \
        [get_attribute $sim output_nets] 0]
     set name [get_attribute $outnet full_name]
     report_simulation -template $sim > $name.vec
   }
   ```

6. Edit the simulation vector template files to specify the conditions for simulation, then apply the new files to the simulation region with the `set_simulation_attributes -vectors` command:

   ```
   set_simulation_attributes -vectors $name.vec $sim
   ```

7. Run path tracing to perform the simulations and complete the timing analysis.
8. Report the timing results with the `report_paths` command.

Marking Dynamic Simulation Regions

Specify a region in the design for simulation by using the `mark_simulation` command to list one or more nets in the design, as shown in the following example:

```
nt_shell> mark_simulation -net n13
Information: Created DDS region number 0 from 1 unique channel connected
             block. (DDS-006)
1
```

When you execute the `check_design` command, NanoTime finds the smallest channel-connected region in the design that includes the specified nets and sets up a simulation of that unit.

NanoTime evaluates each marked net individually for the minimum simulation region that can be built to simulate a delay arc through the net. The marked net might not be on the boundary of the simulation region. If the net is inside a channel-connected region, it is not visible as a timing point in the design.

The number of simulation regions might be less than the number of marked nets. The minimum simulation region that includes a marked net might overlap other marked nets or share a border with other simulation regions. NanoTime merges adjacent regions into a single simulation unit. Regions that are not merged are simulated separately.

If a channel-connected region receives a signal and the signal complement through an inverter, NanoTime adds the inverter to the region for better accuracy. An example of this type of circuit is a transmission gate using ctr and ctr_bar signals to control the pass transistors. If you do not want such circuitry added to the region, use the `-channel_expansion_only` option of the `mark_simulation` command. In that case, only the channel-connected paths containing the specified nets are included in the region.

Simulation regions cannot be chained together. A single `mark_simulation` command must be used to identify all of the stages to be simulated together, even if a stage is as simple as a single inverter.

By default, a region is limited to 10 inputs to prevent excessively long runtimes. To specify a larger or smaller limit, use the `-max_inputs` option of the `mark_simulation` command.

Reporting Dynamic Simulation Regions

A simulation region identified by the `check_design` command is represented as a simulation object, which can be accessed with the `get_simulations` and `all_simulations` commands. Each simulation region has one or more input nets, a single output net, and other attributes that are accessible with the `get_attribute` command. The output net acts as a unique identifier for the simulation region.

The `all_simulations` command creates a collection of all simulation objects in the design, whereas the `get_simulations` command creates a collection of simulation objects that have specified pins or nets at their inputs or outputs.

You can set a variable to a collection of simulation objects by using the `set` command, as in the following example:

```
nt_shell> set mysim [get_simulations -input in1]
```

You can get information about the simulation by using the `report_simulation` command:

```
nt_shell> report_simulation $mysim

Summary of Simulations
                                Num     Num     Num   Num
Outputs                         Outputs Inputs  Nets  TX
------------------------------- ------- ------- ----- -----
{net1 net2 net3 net4}             4       3      7     20
{net5 net6 net7}                  3       2      8     18
...
```

The `report_simulation -verbose` command provides the names of all of the nets and transistors involved in the simulation, as well as the name of the vector file, if used. An example is as follows:

```
nt_shell> report_simulation -verbose $mysim

Detailed Simulations Report

Attribute       Value
--------------- ----------------------------------------------
Type            data
Output Nets     {net1 net2 net3 net4}
Input Nets      {net25 net28 net32}
Internal Nets   {net120 net121 net122 ...}
Transistors     {path1.Mn0 path1.Mp0 path2.Mn0 ...}
Spice Header
Control Header
Vectors         /directory_abc/vector_file.vec
...
(additional similar sections)
...
```

The command can also be used to generate a SPICE deck or a vector template file by redirecting the command output into a file. The `-spice_deck` option writes out the region as a SPICE deck. The `-vectors` option writes out the region vectors. The `-template` option writes out a vector-file template for the region, which you can edit.

Use the `set_simulation_attributes` command to changes the attributes of a simulation object. For example, the following command sets the simulation runtime to 10,000 picoseconds and the vector file to v1.vec for the simulation object mysim:

```
nt_shell> set_simulation_attributes -time 10000 -vectors v1.vec $mysim
```

Dynamic Delay Simulation Vector Files

By default, NanoTime runs multiple simulations on the marked region to cover all combinations of values on the side inputs. This might result in long runtimes and wasted effort for meaningless combinations. For better performance, you can use a vector file to specify the side inputs to use for simulation.

For each simulation region, NanoTime searches for a valid vector in the vector file. A vector is valid if it contains the primary input signal and if that signal makes a transition according to the state table in the vector file.

Use the `report_simulation -template` command to create a template for editing. Syntax rules include the following:

- A semicolon at the beginning of a line denotes a comment. You might need to remove semicolons in a template file to customize it for your needs.
- A colon separates a section header from its arguments. The arguments should begin on the line after the section header.
- Objects in a list must be separated by spaces.
- Each section can appear only one time. Sections should appear in the same order as in the template file.

The following is a simple example of a vector file:

```
Outputs:
out_L

Signals:
IN_H iso_cpu IN_L

Vectors:
r 0 0
0 0 r

States:
out_H out_L

Initial conditions:
0.9 0
0 0.9

Options:
max min
```

The vector file sections are as follows:

- Outputs: The output net or nets
- Signals: Side input nets
- Vectors: The combinations of side input states to simulate

  The number of values on each line must equal the number of side input signals. Valid values are as follows:

  - 1 (logic 1, constant)
  - 0 (logic 0, constant)
  - r (rising transition)
  - f (falling transition)
  - X (expands to both 0 and 1)
  - * (expands to both r and f)

  The x and * entries help to reduce the size of the vector list. For example, a vector specification of X * expands to the four vectors 0 r, 0 f, 1 r, 1 f.
- States: (Optional) Internal nets on which to impose initial voltage conditions
- Initial Conditions: (Optional) Initial conditions (voltages) for the nets included in the States section

The number of initial condition values on each line must equal the number of nets. You can provide any number of sets of initial conditions for the same list of nets by providing additional lines in the Initial Conditions sections.

A separate simulation is performed for every combination of input vectors and initial conditions.

- Options: (Optional) Valid arguments are `min`, `max`, or both. The default is `max`, for maximum-delay simulation.

Multivoltage Analysis Using Dynamic Simulation

An example of a circuit that can benefit from dynamic delay simulation is a circuit with two or more supply voltages within the same stage, such as the level shifter in Figure 14-3.

Note:
Using differential circuit marking commands is an alternative method to analyze level shifter circuits. For more information, see Level Shifter Circuits in Chapter 7, "Differential Circuits."

Figure 14-3 Level Shifter Circuit With Input Stage

When using dynamic simulation on multivoltage designs, you must force consecutive stages into the same simulation region so that the dynamic simulation region does not end at any voltage or power domain boundaries. You can do this by adding a net from the stage before

the level shifter and a net from the level shifter into the same `mark_simulation` command, as follows:

```
nt_shell> mark_simulation -net { in out outb }
```

You can use dynamic simulation to analyze a multivoltage design that includes virtual supply or ground nets, such as the circuit in Figure 14-4.

To set up the virtual ground for this circuit, use the following command:

```
nt_shell> set_supply_net -virtual "vgnd"
```

You must also specify the correct state on the virtual supply enable device to ensure that the transistor is always on:

```
nt_shell> set_case_analysis 1 "en"
nt_shell> mark_simulation -net "Z"
```

Figure 14-4 Level Shifter With Virtual Ground

Dynamic Delay Simulation Commands

The following commands and variables control dynamic delay simulation:

The `mark_simulation` command

Specifies the portion of the design to be simulated. All transistors with a channel-connected path to the specified net become part of the same dynamic delay simulation region. If multiple nets are specified, adjacent regions are joined into a single region. This command must be used after the `check_topology` command but before the `check_design` command.

The `all_simulations` command

Creates a collection of all simulation objects in the design.

The `get_simulations` command

Creates a collection of simulation objects in the design based on a specified list of input or output pins or nets.

The `set_simulation_attributes` command

Sets the simulation parameters: SPICE header file name, total number of time units in the simulation run, vector file name, and list of simulation objects affected by the command. This command must be used after the `check_design` command but before the `trace_paths` command.

The `report_simulation` command

Reports information about a simulation object, including the associated SPICE deck, region definition, or region simulation vectors.

15

Parametric On-Chip Variation Analysis

Process and operating condition variations can cause physical chips to behave differently from timing analysis. Two NanoTime features model variations and provide more accurate timing analysis: parametric on-chip variation (POCV) and path-based slack adjustment (PBSA). Both features include clock path pessimism removal.

This chapter contains the following sections:

- Parametric On-Chip Variation Analysis Overview
- Using POCV in the NanoTime Flow
- POCV Delay Calculations
- Variation Parameters for Transistors
- Variation Parameters for Library Cells
- Variation Parameters for Other Devices
- Recommendations for Variation Parameters
- Reporting Parametric On-Chip Variation

Parametric On-Chip Variation Analysis Overview

Parametric on-chip variation (POCV) analysis accounts for local fluctuations in device characteristics by using statistical modeling techniques. Local variations are changes in device properties that cause differences in transistor behavior within the same die, the same circuit, or the same path. Advanced processes sometimes generate random fluctuations in properties such as line width or doping level.

You can use POCV and path-based slack adjustment (PBSA) analysis together or separately. Both techniques require a NanoTime Ultra license.

See Also

- Chapter 16, “Path-Based Slack Adjustment Analysis”

Using POCV in the NanoTime Flow

To use parametric on-chip variation analysis, perform the following steps:

1. Perform standard NanoTime setup up to and including the `check_design` command.
2. Define the variation characteristics of NMOS and PMOS reference transistors with the `set_variation_parameters` command with the `-type nmos` or `-type pmos` option and other options as needed to define a group of transistors with related variations. Use the command as many times as needed to represent the device types in the design.
3. (Optional) Set the thresholds for matching transistors to the reference transistors with the `tech_pocv_match_voltage_pct, tech_pocv_match_length_pct,` and `tech_pocv_match_wrapper_parameter_pct` variables.
4. Define the properties for unmatched transistors, library cells, and dynamic simulation regions with the `set_variation_parameters -type default` command.
5. (Optional) Define the global variation characteristics of library cells or dynamic simulation regions with the `set_variation_parameters` command and the `-type libcell`, `-type dcs`, `-type dds`, or `-type ntm_dds` option.
6. (Optional) Define the variation characteristics of specific library cells with the `set_libcell_variation_parameters` command.
7. (Optional) Specify the use of output transition variation in the analysis by setting the `timing_enable_slew_variation` variable to `true`.
8. (Optional) Specify variation values for specific timing arcs by using the `set_pin_variation` command.

9. Use the `report_variation` command to examine whether all devices in the design have an assigned variation value; make corrections as needed.
10. (Optional) Include RC delays in the variation analysis by setting the `timing_pocv_gate_delay_only` variable to `false`.
11. (Optional) Set the number of standard deviations of variation to use in the analysis with the `timing_pocv_sigma` and `timing_pocv_sigma_min` variables. Use a value of 3 or more to be more conservative or less than 3 to be more aggressive.
12. Use the `-pocv` option with the `trace_paths` or `extract_model` command. Consider saving more than one path because variation analysis might change the critical path.
13. (Optional) Examine the delay adjustment for a timing path by examining the `clock_variation`, `data_variation` and `slack_variation` attributes. These attributes are associated with the `timing_path` object type and are in user time units.
14. Use the `report_paths` command with the `-path_type full`, `-path_type full_clock`, or `-path_type full_clock_expanded` options to see delay adjustments for clock and data paths.

    (Optional) Use the `-variation` option of the `report_paths` command to add a column to the path report that displays the variation at each stage of a path. Even if you do not use the `-variation` option, the reported path delays still include the adjustments that result from the variation analysis.
15. (Optional) Use the `report_variation_calculation` command to view variation values and nominal delays for specific timing paths and details of adjustments due to variation.

POCV Delay Calculations

Parametric on-chip variation analysis works on a path by path basis.

First, you provide one-sigma variation values for various device types by using the `set_variation_parameters` command. NanoTime then calculates the amount of variation for an individual delay arc based on the variation for active switching devices included in the arc. Finally, for each complete path, NanoTime combines the component arc variation values in a statistical manner to determine the overall path variation and slack.

You control the analysis by defining device variation characteristics for specific types of transistors and library cells. In addition, you can specify the number of standard deviations to use in the delay analysis and choose whether or not to include wire delay in the analysis.

For example, for a specific transistor width and model, the one-sigma variation might be 10 percent. If the nominal delay for this transistor type is 25 ps, NanoTime calculates the one-sigma variation adjustment to be 2.5 ps. This value is scaled by the specified number of

standard deviations, then added to the nominal delay for maximum delay analysis or subtracted from the nominal delay for minimum delay analysis.

The circuit in Figure 15-1 is a simple example of path variation analysis. Details about how the device variations are set and used appear in the following sections.

Path1 traverses arc1 (through a PMOS transistor), arc3 (through a library cell), and arc4 (through a dynamic delay simulation region). Path2 traverses arc2 (through two NMOS transistors in series), arc3, and arc4. The nominal delay of each of the timing arcs is 10 ps.

Figure 15-1 Path Variation Calculation

Table 15-1 lists the one-sigma variation values set for each of the device types. Variation values are provided in the range of 0.0 to 1.0, representing 0 to 100 percent. Assume that this is a maximum delay analysis, which causes the nominal delay to increase.

The modified arc and path delays are shown in Table 15-2. The individual object delays are scaled by the number of standard deviations by using the `timing_pocv_sigma` variable, whose default is 3. The series configuration of the two NMOS transistors is taken into consideration by a scale factor in the transistor property definition command, which is described in the next section.

Path1 contains arcs through a PMOS transistor, a library cell, and a dynamic delay simulation region. A specific variation to use for dynamic simulation regions has not been set, therefore the default variation value of 0.05 is used for this arc.

Path2 contains arcs through two NMOS transistors, the library cell, and the dynamic simulation region.

Table 15-1 Device Variation Values for Example

Instance	Model	Variation
X1	libcell_37b	0.18
mn1, mn2	nch_n5	0.10
	series2 factor	1.18
mp1	pch_8607	0.25
mp2	pch_8772	0.13
DDS regions		not set
default		0.05

Table 15-2 Modified Maximum Delays for Arcs and Paths

Timing arc	Variation (ps)	Modified delay (ps)	Comments
arc1	(3*0.13) * 10 = 3.9	10 + 3.9 = 13.9	includes transistor mp2
arc2	(3*0.1) *10 / sqrt (1.18) = 2.8	10 + 2.8 = 12.8	includes transistors mn1 and mn2 in series
arc3	(3*0.18) * 10 = 5.4	10 + 5.4 = 15.4	includes instance X1
arc4	(3*0.05) * 10 = 1.5	10 + 1.5 = 11.5	includes the DDS region, but the default variation is used
path1	sqrt ($3.9^2 + 5.4^2 + 1.5^2$) = 6.83	30 + 6.83 = 36.83	includes arc1, arc3, and arc4
path2	sqrt ($2.8^2 + 5.4^2 + 1.5^2$) = 6.26	30 + 6.26 = 36.26	includes arc2, arc3, and arc4

The delays for path1 and path2 are calculated by combining the stage delays statistically as the square root of the sum of the square, reflecting the statistical likelihood that some of the variations in this path cancel. For example, the arithmetic sum of the stage delays for path 1 is 40.8 ps, but the statistical sum is 36.8 ps.

Wire Delay

You can choose whether to include wire delay in the variation calculation for a timing arc (stage). The `timing_pocv_gate_delay_only` variable is `true` by default, which means that wire delay is not included in the analysis. If you set the variable to `false`, the delay for a stage is computed by applying the variation value specified for the primary object in that stage (a transistor, library cell, or dynamic simulation region) to the combined gate delay and wire delay of the stage.

Variation Parameters for Specific Timing Arcs

You can override the global variation specifications and modify the variation values for specific timing arcs by using the `set_pin_variation` command. In this command, you specify the timing arc (or arcs) of interest and the new variation value. POCV analysis works as follows:

1. NanoTime calculates the nominal delay for the specified timing arc.
2. The tool multiplies the nominal delay by the specified variation value.
3. The tool multiplies the result by the number of standard deviations specified by the `timing_pocv_sigma` variable to determine the new stage variation.

For example, consider the circuit in Figure 15-1, whose calculated variations are listed in Table 15-1. The variation for arc1 is 3.9 ps, which is determined by the variation for PMOS transistor mp2. The following command changes the variation for arc1 to 5.0 ps:

```
set_pin_variation -variation -fall -target C 5.0 B
```

The `-fall` option is necessary to distinguish arc1 from arc2. The `-target` option is necessary to specify an endpoint for the new variation value. If the `-target` option is omitted, the new variation value applies to all paths that pass through B to OUT. A target net must be in the immediate fanout of the trigger net.

The following usage notes apply:

- For a timing arc whose variation is set by the `set_pin_variation` command, NanoTime ignores the variation values for transistors, library cells, or dynamic simulation regions

that make up that timing arc. In addition, the tool does not scale the variation values based on transistor parameters (length, width, and so on).

- You must specify a list of trigger objects (nets or transistor gate pins) to define the affected timing arc or arcs. You can optionally specify a list of target objects (nets, transistor gate pins, or output ports).
- You can optionally specify a supply voltage value. In this case, the new variation applies only to stages that have the specified voltage. Therefore, you can specify multiple `set_pin_variation` commands for the same timing arc if the commands have different voltage specifications. Voltage-specific commands take precedence over commands without a voltage option.

To remove settings made with the `set_pin_variation` command, use the `remove_pin_variation` command. The `remove_pin_variation` command does not set any variations to zero. The command removes changes made with the `set_pin_variation` command, which means that the path variation is calculated using the variations of objects (transistors, library cells, and so on) in that path.

To list variations set with the `set_pin_variation` command, use the `report_pin_variation` command.

The following is an example of the pin-based variation report:

```
****************************************
Report : report_pin_variation
Design : test
...
****************************************

Trigger                 Target          minmax  Dir     voltage variation
----------------------- --------------- ------- ------ -------- ---------
Xpre1.X10.MN0.G         --              min     rise     1.1000    0.1500
Xpre1.X10.MN0.G         --              min     fall     1.1000    0.1500
Xpre1.X10.MN0.G         --              max     rise     1.1000    0.1500
Xpre1.X10.MN0.G         --              max     fall     1.1000    0.1500
Xpre1.X10.MN0.G         --              min     fall     1.2000    0.3000
Xpre1.X10.MN0.G         --              max     fall     1.2000    0.3000
```

If a variation has been set on a pin, the tool creates one or more of the following pin attributes: `min_rise_pin_variation_override`, `min_fall_pin_variation_override`, `max_rise_pin_variation_override`, and `max_fall_pin_variation_override`. You can query these attributes on pins in the design to determine if a pin variation has been set. Each attribute contains a string the lists each override value and its corresponding target. If all targets map to the same value, the string is the value followed by `all_targets`. These attributes are valid after execution of the `check_design` command.

Variation Parameters for Transistors

To apply parametric on-chip variation, NanoTime requires information about the amount of delay variation to expect from transistors. You must specify the variation for at least one NMOS and one PMOS reference transistor. NanoTime uses this information to calculate the variation characteristics for paths that include these device types.

Specify transistor variation by using the `set_variation_parameters` command with the `-type` option set to either `nmos` or `pmos`. The argument of the `-variation` option is the one-sigma variation.

For transistors, the `set_variation_parameters` command must include the following options:

- Transistor type (`-type nmos` or `-type pmos`)
- Channel width in microns for standard transistors (`-width`) or the number of fins for FinFET transistors (`-nfin`)
- The delay variation (`-variation`)

The delay variation is the one-sigma variation, expressed as the ratio of the variation to the nominal delay for that device type. The value is typically between 0.0 (no variation) and 0.4 (40 percent variation).

Note:
By default, the NanoTime tool uses three standard deviations of variation for every device during path delay analysis. You can change the number of standard deviations to use by setting the `timing_pocv_sigma` and `timing_pocv_sigma_min` variables.

Matching Transistors to Variation Specifications

You can apply a variation value to a restricted group of transistors by using options of the `set_variation_parameters` command. Transistor matching is tested in the following order:

1. Model name, in alphabetical order (the `-transistor_model` option)
2. Transistor type (NMOS or PMOS)
3. Maximum or minimum delay analysis (the `-min` and `-max` options)
4. Supply voltage (the `-voltage` option)
5. Channel length in microns (the `-length` option)
6. Wrapper parameters (the `-wrapper_parameters` option); the largest number of parameter matches takes priority

If you use any of these options to identify a group of similar transistors, you must specify the variation for exactly one width or exactly one fin count for each combination of options. Variation values for similar transistors of different sizes are obtained by scaling by width or fin count.

By default, the channel length, supply voltage, and wrapper parameters of a transistor must be within 1 percent of the provided values to be considered a match. You can control the tolerances for matching by setting the `tech_pocv_match_length_pct`, `tech_pocv_match_voltage_pct`, and `tech_pocv_match_wrapper_parameter_pct` variables.

The cell attribute `has_transistor_variation` is set to `true` for transistors that have a variation match at either the maximum delay or minimum delay corner.

Important:

If NanoTime cannot find a variation definition to match the trigger device in a stage, the variation specified by the `-type default` option is used to represent the entire stage.

It is more accurate to define transistor characteristics explicitly rather than using the `-type default` option. When NanoTime uses the default variation, parameters such as the channel width or the number of fins are not considered for any of the transistors in the stage controlled by the trigger device.

Using Wrapper Parameters to Match Transistors

The `-wrapper_parameters` option for the `set_variation_parameters` command allows you to specify variation values for a more restricted set of devices than is possible with the built-in options.

The argument of the `-wrapper_parameters` option is a list of transistor model parameters and values specified in the following format:

```
-wrapper_parameters {param1=value1 param2=value2 ...}
```

For example, the following command assigns a one-sigma variation of 0.3 to NMOS transistors that have a channel width of 0.12 microns, a `multi` parameter equal to 2, and a `scale` parameter equal to 0.9:

```
set_variation_parameters -type nmos -width 0.12 \
                 -wrapper_parameters {multi=2 scale=0.9} -variation 0.3
```

In this example, the variation of 0.3 corresponds to a channel width of 0.12 microns. The variations of other transistors that have a `multi` parameter equal to 2 and a `scale` parameter equal to 0.9 are calculated by using the channel width to scale the variation.

Variation Values for Transistors in Series Stacks

Transistors in stacks do not behave the same as individual transistors. The number of series devices in a stack reduces the overall amount of variation in a stage due to the likelihood that variation effects of the individual transistors offset each other.

Use the `-series2`, `-series3`, and `-series4` options of the `set_variation_parameters` command to refine the analysis for stacks of two, three, or more than three transistors. The value is a number greater than one that represents the number of equivalent devices in the stack. Allowable values for the `-series2` option are 1.0 to 2.0, for the `-series3` option are 1.0 to 3.0, and for the `-series4` option are 1.0 to 4.0.

The modified arc delay is calculated as follows:

$$d_{mod} = \frac{d_{nominal} \times N \times V}{\sqrt{S}}$$

In this equation, d_{mod} is the modified delay, $d_{nominal}$ is the nominal delay, N is the number of standard deviations, V is the one-sigma device variation, and S is the series factor.

Variation Values as a Function of Input Slope

By default, NanoTime does not take input slope into account when calculating stage delay variations. In addition, the tool does not calculate output slew variations. However, transistor variations might change depending on the input slope conditions.

To enable transistor variation analysis as a function of input slope, set the `timing_enable_slew_variation` variable to `true`.

In addition, you must provide a list of values as arguments of the following options of the `set_variation_parameters` command:

- The `-input_transition` option, which specifies input slope values
- The `-variation` option, which specifies input delay variation values (a list of one-sigma variations corresponding to the input slope values)
- The `-transition_variation` option, which specifies output slope variations (a list of one-sigma variations corresponding to the input slope values)

The argument lists of the three options must contain the same number of values. The NanoTime tool uses the input slopes at the trigger pin to calculate the one-sigma delay variations and the one-sigma output slope variations of the stage. These values are used in variation calculations for subsequent stages in the path.

If the `timing_enable_slew_variation` variable is `false`, the `-variation` option must have only one variation value as an argument and the `-input_transition` and `-transition_variation` options cannot be specified.

For example, consider the variation values in Table 15-3. The delay and output slope variations are valid for the input slopes in the first column.

Table 15-3 Example Variation Values

Input Slope	Delay Variation	Output Slope Variation
12.5	0.055	0.066
25.0	0.050	0.048
37.5	0.052	0.032
50.0	0.058	0.020
125.0	0.090	0.018

Use the following commands to specify the variation values in Table 15-3 for the group of PMOS FinFET transistors that have model name pch_a1:

```
set_variation_parameters -type pmos -transistor_model pch_a1 -nfin 2 \
     -input_transition [list 12.5 25.0 37.5 50.0 125.0]
set_variation_parameters -type pmos -transistor_model pch_a1 -nfin 2 \
     -variation [list 0.055 0.050 0.052 0.058 0.090]
set_variation_parameters -type pmos -transistor_model pch_a1 -nfin 2 \
     -transition_variation [list 0.066 0.048 0.032 0.020 0.018]
```

The NanoTime tool interpolates within the table for intermediate values of input slope.

Variation Parameters for Library Cells

The NanoTime tool creates extracted timing models or ETMs (also known as library cells) to represent the input and output timing of a circuit. For hierarchical analysis, the tool also reads timing models as part of a larger design. A timing model always contains delay information for the timing arcs in the model, but it might also contain variation data.

Reading Variation Parameters from Library Cells

The NanoTime tool can read Liberty Variation Format (LVF) data in a library cell for the arc types listed in Table 15-4. The table shows the names of the corresponding data tables for the nominal values and variation values.

All tables are indexed by the same values of input slew and output load. All variation values are one-sigma values.

If the timing model file (.lib file) for a library cell contains variation parameters, the NanoTime tool automatically reads and uses them for POCV analysis, with the exception of the transition variation tables. To read and apply the output transition variation values, you must set the `timing_enable_slew_variation` variable to `true`.

Table 15-4 Timing Model Tables for Nominal and Variation Values

Arc type	Nominal values	Automatic read	Variation values	Automatic read
Delay	cell_rise	yes	ocv_sigma_cell_rise	yes
	cell_fall	yes	ocv_sigma_cell_fall	yes
Constraint	rise_constraint	yes	ocv_sigma_rise_constraint	yes
	fall_constraint	yes	ocv_sigma_fall_constraint	yes
Transition	rise_transition	yes	ocv_sigma_rise_transition	no
	fall_transition	yes	ocv_sigma_fall_transition	no
Retain	retaining_rise	yes	ocv_sigma_retaining_rise	yes
	retaining_fall	yes	ocv_sigma_retaining_fall	yes

Using Variation Parameters for Library Cells

Both the `set_pin_variation` and `set_libcell_variation_parameters` commands override the variation values contained in a library cell. The order of precedence for assigning variation values to library cells or to paths that contain library cells is as follows, from highest to lowest:

1. The `set_pin_variation` command, which specifies a variation value to apply to specific timing arcs. If the specified timing arcs contain library cells, all variation values defined for those library cells are ignored.
2. The `set_libcell_variation_parameters` command, which specifies up to two variation values (one for maximum delay paths and one for minimum delay paths) to use for all paths through the library cells named in the command
3. Variation values contained in the .lib file
4. The `set_variation_parameters -type libcell` command, which specifies the variation value to use for all other library cells
5. The `set_variation_parameters -type default` command, which specifies the variation value to use for all devices or instances not covered by other variation specifications

Saving Variation Data in Extracted Timing Models

By default, the NanoTime tool writes the nominal and variation values in separate tables (groups) in the timing model, following the LVF syntax, as listed in Table 15-4.

The following usage notes apply:

- To generate these tables, the `extract_model` command must be run with the `-pocv` option and the `model_generate_pocv_lvf` variable must be `true` (the default).
- Tables for retain arcs are created only if you use the `-when` option with the `extract_model` command.
- Transition variation tables are created only if the `timing_enable_slew_variation` variable is set to `true`.

By default, the `sigma_type` parameter for variation tables is set to `early_and_late`. If you set the `model_enable_split_early_late_sigma` variable to `true`, the NanoTime tool sets

the `sigma_type` parameter to either `early` or `late`. Table 15-5 shows how the variable affects the contents of the ocv_sigma_* variation tables.

Table 15-5 Effect of the model_enable_split_early_late_sigma Variable

Variable setting	Arc type	Value of sigma_type	Variations are calculated from this path
True	Minimum delay	early	Minimum delay
True	Minimum delay	late	Minimum delay
True	Maximum delay	early	Maximum delay
True	Maximum delay	late	Maximum delay
False	Minimum delay	early_and_late	Minimum delay
False	Maximum delay	early_and_late	Maximum delay

For best results when using a NanoTime timing model in the PrimeTime tool, set the `model_enable_split_early_late_sigma` variable to `true`.

When a timing model with variation is used in timing analysis, the total delay of a minimum delay path might be longer than the total delay of a maximum delay path if the variation values in the model are sufficiently large.

Combined Variation Values

Alternatively, you can generate a model that combines the worst-case variations with the nominal parameter values and writes the results into the `cell_rise`, `cell_fall`, `rise_constraint`, `fall_constraint`, `rise_transition`, and `fall_transition` tables. In this case, the model does not contain variation tables (the `ocv_sigma*` tables).

Specify this format by setting the `model_generate_pocv_lvf` variable to `false` (the default is `true`).

Variation Parameters for Other Devices

Specify the variation values for other devices as follows:

- Dynamic simulation regions

  Use the `set_variation_parameters` command with the `-type dcs` option to specify a variation value for the delay through dynamic clock simulation (DCS) regions. Use the `-type dds` option to specify the variation for dynamic delay simulation (DDS) regions.

The NanoTime tool does not analyze dynamic simulation regions at the transistor level. However, if the `dcs_enable_detailed_path_reporting` variable is true, the variation value specified with the `-type dcs` option is included in the reported delay for each arc of each path through the dynamic clock simulation region.

- Dynamic simulation regions in memory columns

  Use the `set_variation_parameters` command with the `-type ntm_dds` option to specify variation values for DDS regions in memory columns. Memory structures sometimes require larger variation values than other circuits.

- Other arc delays

  Use the `set_variation_parameters` command with the `-type default` option to define the variation for any arc delay that is not otherwise specified.

Important:
When you use the `-type` option with the `libcell`, `dcs`, `dds`, `ntm_dds`, or `default` settings, the only other options allowed are the `-variation`, `-min`, and `-max` options. If you specify any of the transistor-related options, the tool issues a CMDS-100 error and ignores the `set_variation_parameters` command.

Recommendations for Variation Parameters

One method to determine the variation parameters for POCV analysis is to use a dynamic device simulator such as the HSPICE tool. If you use foundry technology files that contain variation parameters to represent the expected amount of process variation, you can run the HSPICE tool with Monte Carlo analysis to simulate a test circuit many times. The percent standard deviation of the resulting delay values can be used as the one-sigma variation value in the NanoTime tool for the devices in the test circuit.

When specifying variation for dynamic clock simulation regions, use the variations corresponding to the following properties:

- The shortest path
- The smallest supply voltage
- The weakest or least active drivers

When specifying variation for dynamic delay simulation regions, use the variations corresponding to the following properties:

- The shortest path in the smallest dynamic simulation region
- The smallest supply voltage
- The weakest or least active drivers

When specifying variation for library cells,

- Set values for specific library cells
- Set a default to use for unmatched library cells
- Use the variation of the smallest supply voltage
- Use the largest value of any index from the rise or fall tables in a timing model or from side files used in PrimeTime advanced on-chip variation or parametric on-chip variation analysis

When specifying variation for transistors,

- Use the variation at the smallest channel width
- Start with the variation at the longest channel length, then characterize additional channel lengths later in the design cycle
- Start with the variation at the smallest supply voltage, then characterize additional supply voltages later
- Start with a loose mapping of similar transistors (for example, a 5 percent channel length and voltage match), then tighten them later
- Start with no definitions for the series stack scaling factors, then add them later

Reporting Parametric On-Chip Variation

The NanoTime tool provides several commands to report the details of variation analysis setup and the effects of variation analysis on path delays.

This section contains the following topics:

- The report_variation Command
- The report_variation_calculation Command
- Variation Analysis in the Path Report

The report_variation Command

The `report_variation` command helps you to analyze whether all devices in the design have been assigned a variation value.

Although you specify variation values in the `set_variation_parameters` command in the range of 0 to 1.0, representing 0 to 100 percent, this report lists variation values in percent.

An example of the variation report is as follows:

```
...
matched     model    nmos/ min/ Vdd    eff.    eff.width
transistors name     pmos  max  Vdd    length  / nfin    var pct series
----------- -------- ----- ---- ------ ------- --------- ------- -----------------
17          nch1     nmos  min  0.800  16.000  2.000     6.358*  1.180 2.180 3.180
17          nch2     nmos  max  0.800  16.000  2.000     6.266*  1.180 2.180 3.180
17          pch1     pmos  min  0.800  16.000  2.000     5.203*  1.180 2.180 3.180
17          pch2     pmos  max  0.800  16.000  2.000     5.235*  1.180 2.180 3.180
```

An asterisk appears after the variation value if, for this device type, the delay and output slope variations were specified as a function of input slopes.

You can report more detailed variation data by using the `-verbose` option with the `report_variation` command. The report contains additional columns for the input and output slopes. An example of the verbose variation report is as follows:

```
...
                                                                    trans
matched model  nmos/ min/ Vdd   eff.   eff.wid input              var
trans   name   pmos  max  Vdd   length  /nfin  trans   var pct  pct  series
------- ------ ----- ---- ----- ------ ------- ------- ------- ----- -----------------
17      nch1   nmos  min  0.80  16.00  2.000   {12.5 25.0 37.5 50.0 125.0}
                                                     {6.36 7.32 7.53 7.63 8.85}
                                                              {4.7 5.9 6.4 6.1 2.8}
                                                                   1.180 2.180 3.220
17      nch2   nmos  max  0.80  16.00  2.000   {12.5 25.0 37.5 50.0 125.0}
                                                     {6.27 7.32 7.59 7.65 8.70}
                                                              {4.6 5.8 6.3 6.1 2.9}
                                                                   1.180 2.180 3.220
17      pch1   pmos  min  0.80  16.00  2.000   {12.5 25.0 37.5 50.0 125.0}
                                                     {6.20 7.19 7.62 7.90 8.96}
                                                              {4.5 5.6 6.1 5.8 3.0}
                                                                   1.180 2.180 3.220
17      pch2   pmos  max  0.80  16.00  2.000   {12.5 25.0 37.5 50.0 125.0}
                                                     {6.15 7.11 7.52 7.80 8.52}
                                                              {4.4 5.7 6.0 5.7 3.2}
                                                                   1.180 2.180 3.220
```

Listing Unmatched Transistors

You can list transistors that do not have a matching variation assignment by using the `report_variation` command with the `-list_unmatched` option. The following options can be used with the `-list_unmatched` option:

- By default, the maximum number of transistors listed is 10. To increase this number, specify a new maximum number with the `-max_unmatched` option.
- To include the wrapper parameters for unmatched transistors, use the `-show_wrapper_parameters` option.

The report_variation_calculation Command

For specific timing paths, the `report_variation_calculation` command reports the nominal delays and the calculated variation values for the clock path, data path, and slack.

An example of the variation calculation report is as follows:

```
nt_shell> report_variation_calculation [get_timing_paths -max_paths 1]

ADJUSTMENT CALCULATIONS
Parameter                                Value
------------------------------ ----------------
timing_pocv_sigma                        1.000
timing_pocv_sigma_min                    0.900
timing_pocv_gate_delay_only              false
common net                                 ---
Nominal Common Path Delay                0.000
Nominal Data Path
    Gate Delay                           3.000
    Wire Delay                           0.000
Nominal Clock Path Delay
    Gate Delay                           2.000
    Wire Delay                           0.000
Data Path Variation                     -0.620
Clock Path Variation                     0.224
Slack Variation                         -0.660
```

Variation Analysis in the Path Report

NanoTime incorporates variation adjustments into all path reporting. To see the adjustments to clock or data paths, use the `report_paths` command with the `-path_type full`, `-path_type full_clock`, or `-path_type full_clock_expanded` options.

The path report contains the POCV delay adjustment in the “Adjust” column on the line immediately preceding the data arrival time. The label “POCV adjustment” appears on the same line.

You can optionally use the `-variation` option of the `report_paths` command to add a column to the path report that displays the variation at each stage of a path.

An example of a detailed path report that includes stage variation is as follows:

```
Startpoint:         in (inout port)
Endpoint:           XL1.XI2.MN1.g
Path Type:          max
Constraint:         latch setup (transparent)

   Path   Incr   Adjust    Var  NT      Point
-------  -----   ------ ------ ----  -- --------------------------------
                 11.000                 clock clk2 (fall)
                  1.000                 input external delay
 12.000           0.000          D    f in (inout)
 12.000  0.000           0.000        f X1.MP1.g (inv)
 13.000  1.000           0.100        r X2.MN1.g (inv)
 14.000  1.000           0.200        f XL1.XI1.MP1.g (inv)
 15.000  1.000           0.122   L    r XL1.XI2.MN1.g (inv)
 15.561           0.561                POCV adjustment
 15.561                                data arrival time
         3.000   12.561  0.255         Total

 17.000          17.000          C    f clk1 (inout)
 17.000  0.000           0.000   C    f X1C1.MP1.g (inv)
 18.000  1.000           0.300   C    r X2C1.MN1.g (inv)
 19.000  1.000           0.400   C    f XL1.MN1.g (lat)
         2.000   17.000  0.500         Total
 19.000  0.000                         setup time
 19.000  0.000                         clock uncertainty
 19.000                                data required time
----------------------------------------------------------------------
 19.000                                data required time
-15.561                                data arrival time
----------------------------------------------------------------------
  3.439                                 slack (MET)
```

The POCV adjustment is the statistical sum of the clock and data path variations. In this path report, the value of 0.561 is calculated as the square root of the sum of the squares of the data or launch path variation (0.255) and the clock or capture path variation (0.500).

The total variation offset is calculated as follows:

$$\mathrm{Var}_{\mathrm{total}} = \sqrt{\mathrm{Var}_{\mathrm{L}}^{2} + \mathrm{Var}_{\mathrm{C}}^{2}}$$

In this equation, $\mathrm{Var}_{\mathrm{total}}$ is the total variation offset, $\mathrm{Var}_{\mathrm{L}}$ is the launch path variation, and $\mathrm{Var}_{\mathrm{C}}$ is the capture path variation.

In the detailed path report, the column labeled *Var* shows the variation value used for each stage. The reported value is the variation of the path segment between the endpoint in the previous line and the endpoint in the current line. For example, the stage from device XL1.XI1.MP1.g to device XL1.XI2.MN1.g has a delay of 1.000 and a variation of 0.122. Device XL1.XI1.MP1.g is the trigger of this stage, therefore the stage variation does not apply to this device even though the values appear on the same line.

Stage variations are combined statistically as the square root of the sum of the squares to obtain the variation value for the complete path. Therefore, the value in the *Var* column on a line labeled *Total* is not an arithmetic sum like the other total values, but a statistical sum.

The reported arc delays are always the nominal arc delays, whether or not variation analysis is enabled and whether or not the `-variation` option is used with the `report_paths` command. All adjustments due to variation are combined into the single POCV adjustment value. If you enable POCV analysis but run the `report_paths` command without the `-variation` option, the variation column is not included, but the POCV adjustment remains.

16

Path-Based Slack Adjustment Analysis

Process and operating condition variations can cause physical chips to behave differently from timing analysis. Two NanoTime features model variations and provide more accurate timing analysis: parametric on-chip variation (POCV) and path-based slack adjustment (PBSA). Both features include clock path pessimism removal.

This chapter contains the following sections:

- Path-Based Slack Adjustment Analysis Overview
- How Path-Based Slack Adjustment Works
- Reporting Path-Based Slack Adjustment
- Common Point Definition and Pessimism Removal

Path-Based Slack Adjustment Analysis Overview

Path-based slack adjustment (PBSA) analysis accounts for global variations in device characteristics by adjusting path delay values based on user-specified scaling factors. It also provides indirect adjustment for local variations and a means of modifying global clock uncertainty values.

Global variations are correlated changes in device properties that affect the entire design, such as die-to-die, wafer-to-wafer, or lot-to-lot variations. For example, the gate oxide thickness might be different from one wafer to another due to deposition process variations.

You can use POCV and PBSA analysis together or separately. Both techniques require a NanoTime Ultra license.

This section contains the following topics:

- Using PBSA in the NanoTime Flow
- How Path-Based Slack Adjustment Works
- Reporting Path-Based Slack Adjustment

Using PBSA in the NanoTime Flow

To use path-based slack adjustment, perform the following steps.

1. Set the scaling factors that control the path adjustment calculations.
2. Reduce or eliminate the global uncertainty values set previously with the `set_clock_uncertainty` command.
3. Eliminate all `set_clock_latency` commands.
4. (Optional) Specify a minimum threshold for path-based adjustment of specific timing checks by setting the `pbsa_min_threshold` variable and using the `-selective_pbsa` option with the `set_timing_check_attributes` command. If the total adjustment for a path is below the specified threshold, NanoTime does not apply the adjustment.
5. (Optional) Use the `set_pbsa_override` command to modify the analysis for selected paths.
6. (Optional) Customize the variables that determine common point recognition and pessimism removal.
7. Use the `-pbsa` option with the `trace_paths` command.
8. (Optional) Use the `report_pbsa_calculation` option command to display the calculations for a list of paths.

See Also

- How Path-Based Slack Adjustment Works
- Reporting Path-Based Slack Adjustment
- Common Point Definition and Pessimism Removal

How Path-Based Slack Adjustment Works

Three types of scaling factors allow you to adjust the delay of a path based on the nominal delay, the number of simulation levels, or both. There are separate scaling factors for minimum and maximum delay analysis, launch and capture paths, and cell and wire delays.

Path-based slack adjustment is designed to be used with propagated clocking, which is the default clocking mode. If any clocks are set to use ideal latency (for example, with the `set_clock_latency` command), path-based slack adjustment cannot be used.

Adjustments for a Single Path

A simulation level is one channel-connected block or one library cell arc used as a simulation unit for timing analysis. For example, an inverter is one simulation unit or one level. The size of a circuit considered to be a simulation unit depends not only on the circuit topology, but also topology definitions (such as `mark_*` commands) and simulation constraints. In general, the longer the path, the larger the number of levels in that path.

A net from an input port to the input of a transistor or library cell does not count as a simulation level.

If the `timing_pbsa_gate_delay_only` variable is set to `false` (the default), NanoTime treats the cell and wire delays the same, as described in this section. If the variable is set to `true`, NanoTime adjusts cell delays as described in this section and wire delays as described in Independent Wire Delay and Cell Delay Adjustments.

Define L as the number of levels and T_{orig} as the original delay time, or the delay simulated without path-based slack adjustment. NanoTime calculates delay time adjustments by applying scaling factors to the original delay time and the number of levels.

The adjustments are added to the original delay time to calculate the adjusted delay time T_{adj} as follows:

$$T_{adj} = T_{orig} + (T_{orig} \times M_T) + (L \times M_L)$$

where M_T is the scaling factor for the delay time (also known as the delay scaling factor or derating factor) and M_L is the scaling factor for the number of levels (also known as the uncertainty per level).

The M_T factor increases or decreases the total delay of a path by a fixed multiplication factor. For example, you can use it to make the original path tracing results more conservative by increasing all path delays for maximum delay analysis and decreasing all delays for minimum delay analysis.

The M_L factor increases or decreases the total delay of a path by a fixed amount of time per simulation level. Typically, you would use a positive value for maximum delay analysis and a negative value for minimum delay analysis. However, this analysis tends to be too pessimistic for increasing stage count because local variations in device characteristics usually cancel each other (to some degree) from stage to stage.

To provide more flexibility, the level-based derating factor is available. It takes the number of simulation levels into account by modifying the delay scaling factor M_T. For example, you can specify that the delay of a two-stage path should be increased by 10 percent, but that the delay of a ten-stage path should be increased by only 8 percent.

If M_{TL} is the level-based derating factor, the new delay scaling factor M_{Tnew} is

$$M_{Tnew} = M_T + (L \times M_{TL})$$

The adjusted delay time is

$$T_{adj} = T_{orig} + (T_{orig} \times M_{Tnew}) + (L \times M_L)$$

NanoTime requires that M_{TL}, when used for maximum delay analysis, is a negative value to ensure that the overall percentage of delay scaling decreases with each additional stage.

You can also set a lower bound for the adjusted scaling factor by using the M_{floor} parameter. NanoTime adjusts the value of M_{Tnew} as follows:

$$M_{Tnew} = \max(M_{floor}, (M_T + L \times M_{TL}))$$

where max is a function that returns the largest value in the argument list. The value of M_{Tnew} cannot fall below the value of M_{floor}.

NanoTime expects the value of M_{floor} to be greater than 0 but less than the value of M_T. If M_{floor} is greater than M_T, NanoTime sets M_{floor} to M_T, which effectively prevents any level-based adjustments to M_T.

Similarly, for minimum delay analysis, NanoTime requires M_{TL} to be a positive value; the overall percentage of delay scaling starts at a large negative value then becomes less negative with each additional stage. You can set an upper bound for the adjusted scaling factor by using the $M_{ceiling}$ value. NanoTime adjusts the value of M_{Tnew} as follows:

$$M_{Tnew} = \min(M_{ceiling}, M_T + L \times M_{TL})$$

where min is a function that returns the smallest value in the argument list. The value of M_{Tnew} cannot rise above the value of $M_{ceiling}$. NanoTime expects the value of $M_{ceiling}$ to be less than 0 but greater than the value of M_T; otherwise, the tool sets $M_{ceiling}$ to M_T.

The final adjusted delay time is calculated as follows:

$$T_{adj} = T_{orig} + (T_{orig} \times M_{Tnew}) + (L \times M_L)$$

The following examples describe how to set the variables to adjust path delays. Apart from a limited amount of range checking, NanoTime does not check the values of scaling factors.

- Using the M_T factors

  To increase all delays in maximum analysis by 10 percent, set M_T to 0.1 and keep the M_L and M_{TL} factors set to 0 (the defaults), for the maximum delay analysis factors only.

  Similarly, to decrease all delays for minimum delay analysis by 5 percent, set M_T to -0.05 and keep the M_L and M_{TL} factors set to 0, for the minimum delay analysis factors only.

- Using the M_L factors

  To increase delays for maximum delay analysis by 5 ps per stage, set M_L to 5 and keep the M_T and M_{TL} factors set to 0 (the defaults).

  To decrease delays by 3 ps per stage, set M_L to -3 and keep the M_T and M_{TL} factors set to 0 (the defaults).

- Using the M_{TL} factors

  To increase delays on a customized per stage basis, first set M_L to 0. Set M_{TL} to a negative number that represents the decrease in percentage delay adjustment per level. Set M_T to the positive percentage delay adjustment to be used for a one-stage path, increased by the absolute value of M_{TL} (because this amount is subtracted during the calculation for one-stage paths). Finally, if you want to set an optional minimum value for the delay scaling factor, set M_{floor} to the minimum value.

  For example, if M_T = 0.1025 and M_{TL} = -0.0025, NanoTime adjusts the delay of a one-stage path by +10 percent, a two-stage path by +9.75 percent, a three-stage path by +9.5 percent, and so on, until the adjustment reaches the minimum allowed value.

  You can similarly decrease delays by setting M_{TL} to a positive value and $M_{ceiling}$ to the (optional) maximum value of the delay scaling factor. In this case M_T is the negative percentage delay adjustment to be used for a one-stage path, decreased by the absolute value of M_{TL}.

  For example, if M_T = -0.0525 and M_{TL} = +0.0025, NanoTime adjusts the delay of a one-stage path by -5 percent, a two-stage path by -4.75 percent, a three-stage path by -4.5 percent, and so on, until the adjustment reaches the maximum allowed value.

Independent Wire Delay and Cell Delay Adjustments

The total delay of a path includes cell delay and wire delay. You can apply separate scaling factors to cell and wire delay by setting the `timing_pbsa_gate_delay_only` variable to `true` and specifying the wire delay scaling factors.

For a single path, the original delay T_{orig} (the delay calculated without path-based slack adjustment) includes the cell delay TC and the wire delay TW:

$$T_{orig} = TC + TW$$

If the `timing_pbsa_gate_delay_only` variable is set to `false` (the default), NanoTime treats the cell and wire delays the same during path-based slack adjustment analysis. The calculation is described in Adjustments for a Single Path.

If the `timing_pbsa_gate_delay_only` variable is set to `true`, NanoTime adjusts the wire delay using the wire delay scaling factor M_W, as follows:

$$TW_{adj} = TW + (TW \times M_W)$$

NanoTime adjusts the cell delay portion of the path delay as described in Adjustments for a Single Path. The final adjusted path delay is the sum of the adjusted cell delay and the adjusted wire delay:

$$T_{adj} = TC_{adj} + TW_{adj}$$

Maximum delay analysis and minimum delay analysis use separate scaling factors. The wire delay scaling factor for maximum delay analysis is a number between 0 and 1. For example, to increase the wire delay of a path by 10 percent, set the factor to 0.10.

The wire delay scaling factor for minimum delay analysis is a number between -1 and 0. For example, to decrease the wire delay of a path by 10 percent, set the factor to -0.10.

Path Segments to Consider for Timing Checks

For path-based slack adjustment, NanoTime considers four path segments in each setup or hold timing check. The four segments are shown in Figure 16-1 and are labeled A, B, C, and D. Different multiplication factors apply to each path segment.

Figure 16-1 Timing Path Segments

Path segments A, B, C, and D represent conceptual paths as follows:

- Segment A – The shared or common portion of the launch clock path and capture clock path (if any), from the clock input port to the common point.
- Segment B – The portion of the launch clock path from the common point to the clock pin of the launching sequential element.
- Segment C – The portion of the capture clock path from the common point to the clock pin of the capturing sequential element.
- Segment D – The data path from the clock pin of the launching sequential element to the data input pin of the capturing sequential element.

Launch path analysis includes segments A, B, and D. Capture path analysis includes segments A and C.

If any segment of type A, B, or C contains chained generated clocks, they are included in the analysis. (Figure 16-1 shows a generated clock in segment A.) If dynamic clock simulation is in use to simulate complex clock circuits, all clock paths are included in the analysis.

Setup and Hold Slack Adjustments

A setup check determines if a data signal arrives early enough to be captured when the clock signal switches. The slack S_{setup} is the amount of time by which the data signal is early (negative slack means the data signal is late). The slack is also reduced by the clock uncertainty U. If T_{req} is the data required time, which is the time at which the clock signal arrives, and T_{arr} is the data arrival time, the setup slack is as follows:

$$S_{setup} = T_{req} - T_{arr} - U$$

Based on the paths in Figure 16-1, the data path includes segments A, B, and D. The clock path includes segments A and C. For a setup check, NanoTime uses the fastest clock path time (segments A and C) and slowest data arrival time (segments A, B, and D):

$$S_{setup} = (A_{min} + C_{min} + P) - (A_{max} + B_{max} + D_{max}) - U$$

P is the clock period, in which capture occurs on the next clock edge after the launch event. A_{min} is the minimum delay for path segment A, A_{max} is the maximum delay for path segment A, and similar notation represents delays for path segments B, C, and D.

However, the delays through path segment A (A_{min} and A_{max}) must be equal, regardless of whether path-based slack adjustment is enabled. Therefore, the A_{min} and A_{max} terms in the setup calculation cancel each other out. For this reason, the variables for path segment A have no effect on the analysis.

A hold check examines whether a data signal is stable enough that it does not switch too soon after being captured. The slack S_{hold} is the amount of time by which the data signal maintains its value after the clock edge (negative slack means the data signal switches too soon):

$$S_{hold} = T_{arr} - T_{req} - U$$

For a hold check, NanoTime uses the slowest clock path time (segments A and C) and fastest data arrival time (segments A, B, and D):

$$S_{hold} = (A_{min} + B_{min} + D_{min}) - (A_{max} + C_{max}) - U$$

As with the setup check, the delay through path segment A cannot equal both A_{min} and A_{max} concurrently. Therefore, the two delay values for segment A in the calculation of S_{hold} must be equal; consequently, they cancel each other out.

When path-based slack adjustment is enabled, NanoTime adjusts each of the path delays according to the values of the pbsa_* global variables.

The `pbsa_KUsetup` and `pbsa_KUhold` variables act as scaling factors for the clock uncertainty during setup or hold checks. When path-based slack adjustment is enabled, these variables are generally set to 0 to disable the use of global clock uncertainty values.

Variables That Control PBSA Analysis

Table 16-1 lists the path segment variables that control path-based slack adjustment calculations. There is a variable for each combination of path segment (A through D), type of constraint (max for setup checks and min for hold checks), and type of scaling factor (simple derating, uncertainty per level, level-based derating, and wire delay derating). Other variables define the upper or lower bounds for the level-based derating factors.

Note:

The usage of the PBSA variables is not affected by the `-force_max_max` or `-force_min_min` options of the `set_timing_check_attributes` command.

Table 16-1 PBSA Variables for Path Segments

Path	Wire delay factor (M_W)	Cell delay factor (M_T)	Cell delay uncertainty per level (M_L)	Cell delay level-based derating factor (M_{TL})

A	`pbsa_KAwiremax` [0.0, +1.0]
	`pbsa_KAwiremin` [-1.0, 0.0]
		`pbsa_KAmax` [-1.0, +1.0]
		`pbsa_KAmin` [-1.0, +1.0]
			`pbsa_KALmax`
			`pbsa_KALmin`
				`pbsa_KALpctmax` [-1.0, 0.0]
				`pbsa_KALpctmin` [0.0, +1.0]
				`pbsa_KAceilingmin`
				`pbsa_KAfloormax`

B	`pbsa_KBwiremax` [0.0, +1.0]
	`pbsa_KBwiremin` [-1.0, 0.0]
		`pbsa_KBmax` [-1.0, +1.0]
		`pbsa_KBmin` [-1.0, +1.0]
			`pbsa_KBLmax`
			`pbsa_KBLmin`
				`pbsa_KBLpctmax` [-1.0, 0.0]
				`pbsa_KBLpctmin` [0.0, +1.0]
				`pbsa_KBceilingmin`
				`pbsa_KBfloormax`

C	`pbsa_KCwiremax` [0.0, +1.0]
	`pbsa_KCwiremin` [-1.0, 0.0]
		`pbsa_KCmax` [-1.0, +1.0]
		`pbsa_KCmin` [-1.0, +1.0]
			`pbsa_KCLmax`
			`pbsa_KCLmin`
				`pbsa_KCLpctmax` [-1.0, 0.0]
				`pbsa_KCLpctmin` [0.0, +1.0]
				`pbsa_KCceilingmin`
				`pbsa_KCfloormax`

D	`pbsa_KDwiremax` [0.0, +1.0]
	`pbsa_KDwiremin` [-1.0, 0.0]
		`pbsa_KDmax` [-1.0, +1.0]
		`pbsa_KDmin` [-1.0, +1.0]
			`pbsa_KDLmax`
			`pbsa_KDLmin`
				`pbsa_KDLpctmax` [-1.0, 0.0]
				`pbsa_KDLpctmin` [0.0, +1.0]
				`pbsa_KDceilingmin`
				`pbsa_KDfloormax`

All of the segment-based variables are floating-point numbers that default to 0. Limits on the allowed values are indicated in square brackets in Table 16-1. If no limits are listed, then NanoTime does not check the values and you must ensure that the values are reasonable.

For completeness, the same parameters exist for all four path segments. However, the delay of segment A cancels in all setup and hold analysis. Therefore the segment A parameters do not affect the results.

Table 16-2 lists additional variables that affect the analysis.

Typically, the `KUsetup` and `KUhold` variables, which are the fixed-uncertainty scaling factors, are chosen to either enable or completely disable the global clock uncertainty defined with the `set_clock_uncertainty` command. A value of 0.0 disables the global uncertainty and is the appropriate value to use when path-based slack adjustment is intended to entirely replace the global uncertainty. On the other hand, to account for off-chip design-external uncertainty (such as source clock uncertainty), you should set `KUsetup` and `KUhold` to 1.0 and set the new external uncertainty values with the `set_clock_uncertainty` command.

The `pbsa_min_threshold` variable sets a minimum allowed amount of delay adjustment. It is applied only when the `set_timing_check_attributes` command includes the `-selective_pbsa` option.

Table 16-2 Other PBSA Variables

Variable	Default	Description
`timing_pbsa_gate_delay_only`	`false`	If the value is `false`, the cell delay scaling factor variables apply to the total path delay, including both wire and cell delay. If the value is `true`, separate scaling factors apply to the wire and cell delays.
`pbsa_KUsetup`	0.0	Multiplier for clock uncertainty values set with the `set_clock_uncertainty` command; used for setup checks.
`pbsa_KUhold`	0.0	Multiplier for clock uncertainty values set with the `set_clock_uncertainty` command; used for hold checks.
`pbsa_min_threshold`	0.0	Threshold for adjustments when selective PBSA is enabled.
`pbsa_enable_chained_generated_clocks`	`true`	If the value is `true`, chained generated clock segments are included in both the data and clock paths when determining the common segment.

Overriding Path-Based Slack Adjustments

You can selectively override the scaling performed by path-based slack adjustment by using the `set_pbsa_override` command. This command specifies which scaling factor changes to make and which paths are affected.

How the Override Scaling Factors Work

The `set_pbsa_override` command specifies the amount of additional delay to add to the nominal path delay. The delay is scaled by a factor of (1+*value*), where *value* can be positive or negative. For example:

- A value of 0.05 scales the nominal path delay by a factor of 1.05.
- A value of -0.05 scales the nominal path delay by a factor of 0.95.
- A value of zero results in a scaling factor of 1, which sets the delay to its nominal value (that is, the delay calculated without any adjustment).
- A value of -1 results in a scaling factor of zero, which sets the delay to zero. Values less than -1 are not permitted for the `-Amin_factor, -Bmin_factor, -Cmin_factor`, or `-Dmin_factor` options.

The scaling factors specified in the `set_pbsa_override` command override previously defined PBSA adjustments for the path segments in question. For example, if you set the `-Bmax_factor` option to 0.05, the standard PBSA variables for path segment B, such as the `pbsa_KBmax` and `pbsa_KBLmax` variables, are ignored.

Important:

If you use multiple `set_pbsa_override` commands for the same path, only the last command is honored. This means that if you want to use more than one of the override factor options (such as the `-Amin_factor` and `-Cmax_factor` options) for a path, you must use all of the necessary options in a single `set_pbsa_override` command.

You can use multiple `set_pbsa_override` commands if they apply to different paths.

How the Scaling Factors Apply to Path Segments

The `set_pbsa_override` command is a timing exception command. This command operates as follows:

1. The NanoTime tool finds one or more path segments that match the `-from`, `-through`, and `-to` type options used in the `set_pbsa_override` command.
2. The tool classifies a matched path segment as type A, B, C, or D, as shown in Figure 16-1.

3. The tool identifies the other path segments that correspond to the matched path segment. For example, if the matched path segment is type B, NanoTime identifies path segments of type A, C, and D that are associated with the B segment.
4. The scaling factors in the `set_pbsa_override` command are applied to the appropriate path segments, overriding any previously set standard PBSA variables.
5. If the `set_pbsa_override` command does not specify scaling factors for some of the path segments, NanoTime applies the standard PSBA variables to those path segments (if the variables were previously defined).
6. The path delay is calculated when the timing check at the end of path segment D is evaluated. At this point, the timing exception is satisfied and will not be applied at any subsequent timing checks or path endpoints.

   This means that if path tracing continues beyond the location of the timing check, standard PBSA adjustments are applied to any subsequent path.

The `remove_pbsa_override` cancels the effects of previous `set_pbsa_override` commands.

The `set_pbsa_override` and `remove_pbsa_override` commands must be specified after the `check_topology` command but before the `check_design` command.

See Also

- Using PBSA in the NanoTime Flow
- Reporting Path-Based Slack Adjustment

Reporting Path-Based Slack Adjustment

Invoking path-based slack adjustment changes the timing results, so the paths with the worst-case timing might be different from the worst-case paths calculated without POCV. By default, the `trace_paths` command saves only the single worst-case path per startpoint-endpoint pair. Therefore, if you want to compare the timing of paths with and without path-based slack adjustment, you might need to save more paths in the path database.

To increase the number of paths saved by the `trace_paths` command, set the `-keep_paths_within` option to a value slightly larger than the default of 0.0, and set the `-npaths` option to a value larger than 1:

```
nt_shell> trace_paths -keep_paths_within 0.15 -npaths 4
```

In this example, for each startpoint-endpoint pair, the `trace_paths` command keeps all paths that have a slack within 0.15 time units of the worst path, up to a maximum of four paths per startpoint-endpoint pair.

After path tracing is complete, the slack values reported by the `report_paths` command reflect the adjustments made by path-based slack adjustment. To see the adjustments, use the `report_paths` command with one of the `-path_type full`, `-path_type full_clock`, or `-path_type full_clock_expanded` options.

The following path report includes PBSA adjustments:

```
nt_shell> report_paths -max -path_type full -max_paths 1
 ...
    Path     Incr   Adjust NT    Point
-------- -------- -------- -- -- ------------------------------
   1.500             1.500 C   r clk2 (in)
   1.630    0.130          C   r Xadder.Xi0.X0.Mn0.G (inv2)
   1.702    0.072          C   f Xadder.Xi0.X1.Mp0.G (inv2)
   ...
   9.115    0.395              r Xbreg.Xreg50.Mn4.G (muxflop)
   9.412    0.297          L   f Xbreg.Xreg50.X5.Mp0.G (inv2)
   9.612             0.200       PBSA adjustment
   9.612                         data arrival time
            9.124    0.488       Total

   6.843    6.843                Xbreg.Xreg50.Mn0.G (muxflop)
   6.843    0.000                setup time
   6.843    0.000                clock uncertainty
   6.843                         data required time
-------------------------------------------------------------
   6.843                         data required time
  -9.612                         data arrival time
-------------------------------------------------------------
  -2.770                         slack (VIOLATED)
```

If the analysis involves differential circuits, the header of the path report includes the name of the common net and its differential complement, as follows:

```
Startpoint:        clk2 (in port)
Endpoint:          Xbreg.Xreg50.X5.Mp0.G
Path Type:         max
Constraint:        latch setup
PBSA Common Net:   b2clkp
PBSA Complement:   b2clkn
...
```

Reporting PBSA Calculation Details

To find out how an adjustment is calculated from the path segment characteristics and pbsa_* global variables, use the `report_pbsa_calculation` command. Given a path retrieved with the `get_timing_paths` command, the `report_pbsa_calculation` command reports the variable values, the original delay and level depth of the path segments, and the components of the slack adjustment calculations.

The report contains optional components that depend on settings of the PBSA variables. For example, wire delay scaling factors appear only if wire delay analysis is enabled. Common point delay details are shown if appropriate for the common point variable settings.

The clock path adjustment in the report includes any common mode adjustments. The data path adjustment includes clock uncertainty.

You can change the variable values and run the `report_pbsa_calculation` command again to see the changes to the adjustment values. However, changing the variables does not immediately affect the path database or the path reports generated by the `report_paths` command. To update the path database and path reports, use the `reset_design -paths` command, followed by the `trace_paths -pbsa` command.

The PBSA Adjustment Options

The adjustment options section of the PBSA calculation report shows the settings of the variables that control global aspects of the analysis. Table 16-3 lists the report labels and the associated variables or conditions.

Table 16-3 PBSA Adjustment Options

Label in report	Associated variable or condition
gate only	`timing_pbsa_gate_delay_only`
allow reconvergence	`pbsa_allow_reconvergent_common_net`
same switching dir	`pbsa_same_common_switching_direction`
chained genclks	`pbsa_allow_chained_generated_clocks`
min threshold	`pbsa_min_threshold`
same edge reconvergence	`pbsa_same_edge_reconvergence_only`
same direction delays	`pbsa_common_net_use_same_direction_delays`
include si deltas	`pbsa_include_common_si_deltas`

Table 16-3 PBSA Adjustment Options (Continued)

Label in report	Associated variable or condition
same edge zero cycle reconvergence	`pbsa_same_edge_zero_cycle_reconvergence_only`
zero cycle path	`true` if common path is a zero cycle path
diff. switching direction	`pbsa_differential_switching_direction`

Example PBSA Calculation Report

In the following example, the `get_timing_paths` command creates a collection of paths taken from the paths stored in the path database, based on the specified criteria. The `report_pbsa_calculation` command operates on this collection and displays slack adjustment information about the retrieved path.

```
nt_shell> report_pbsa_calculation [get_timing_paths -max -max_paths 1]

USER DEFINED K factors
Param       Value  Param       Value  Param       Value  Param      Value
---------- ------- ---------- ------- ---------- ------- --------- ------
KAmin      -0.500  KBmin      -0.500  KCmin      -0.500  KDmin     -0.500
KAmax       1.200  KBmax       1.200  KCmax       1.200  KDmax      1.200
KALmin      0.000  KBLmin      0.000  KCLmin      0.000  KDLmin     0.000
KALmax      0.000  KBLmax      0.000  KCLmax      0.000  KDLmax     0.000
KALpctmax  -0.100  KBLpctmax  -0.100  KCLpctmax  -0.100  KDLpctmax -0.100
KALpctmin   0.200  KBLpctmin   0.200  KCLpctmin   0.200  KCLpctmin  0.200
KAfloormax  0.500  KBfloormax  0.500  KCfloormax  0.500  KDfloormax 0.500
KAceilmin  -0.100  KBceilmin  -0.100  KCceilmin  -0.100  KDceilmin -0.100
KUsetup     1.000  KUhold      1.000
```

```
COMPUTED PATH PARAMETERS
Param          Value     Param        Value
-------------  --------- ------------ -----
A clk delay    0.000     A clk levels   0
Adelay         0.000     Alevels        0
Bdelay         1.513     Blevels        5
Cdelay         0.888     Clevels        4
Ddelay         2.849     Dlevels       12
Aclkwiredelay  0.001
Awiredelay     0.001
Bwiredelay     0.001
Cwiredelay     0.001
Dwiredelay     0.001
Aclksidelay    0.044    (presence depends on variable settings)
Asidelay       0.000    (presence depends on variable settings)
common net     c2
directions     mismatched
Aclkdelay_r    0.222    (presence depends on variable settings)
Adelay_r       0.137    (presence depends on variable settings)
Aclkdelay_f    1.176    (presence depends on variable settings)
Adelay_f       1.176    (presence depends on variable settings)

ADJUSTMENT OPTIONS
Option                              Value
----------------------------------- -----
gate only                           false
allow reconvergence                 false
same switching dir                  false
chained genclks                     true
min threshold                       0.000
same edge reconvergence             false
same direction delays               false
include si deltas                   false
same edge zero cycle reconvergence  false
zero cycle path                     false
diff. switching direction           ignore

ADJUSTMENT CALCULATIONS
Adjustment    Value
------------  ---------
type           setup
common mode    0.000
uncertainty    0.000
clock path     0.711
data path     45.684
total slack  -44.973
```

See Also

- Using PBSA in the NanoTime Flow
- How Path-Based Slack Adjustment Works

Common Point Definition and Pessimism Removal

The POCV and PBSA features both employ common path pessimism removal. Global variables control how the common point is determined.

Segment A in Figure 16-1 is the shared common portion of the launch and capture clock paths. The point at which the launch and capture clock paths diverge is the common point.

The variables used in common path analysis are included in the report generated with the `report_pbsa_calculation` command.

You can customize the common point determination and pessimism removal, as follows:

- Allow or prohibit reconvergent common nets

  Set the `pbsa_allow_reconvergent_common_net` variable to `true` to allow reconvergence. When the variable is `false` (the default), the search finds the first point of divergence (net X in Figure 16-2). When the variable is `true`, the search finds the last point of divergence (net Z in Figure 16-2).

Figure 16-2 Reconvergent Nets

- Consider the launch path and capture path switching directions
  - To require that switching directions must match at all common points (either divergent or reconvergent), set the `pbsa_same_common_switching_direction` variable to `true` (the default is `false`).
  - To require that switching directions must match only at a reconvergent common point, set the `pbsa_same_edge_reconvergence_only` variable to `true` (the default is `false`) and keep the `pbsa_allow_reconvergent_common_net` variable set to `true`.
- Allow or prohibit multicycle reconvergence for unresolved shapers

  To allow same-cycle same-edge reconvergence while disallowing multicycle same-edge reconvergence, set the `pbsa_same_edge_zero_cycle_reconvergence_only` variable to `true` (the default is `false`).

For a multicycle path, the launch and capture paths usually take different physical paths through the clock gate. In this case, standard common point pessimism removal might be optimistic. Set this variable to `true` to improve the accuracy.

This variable has an effect only if the `pbsa_allow_reconvergent_common_net`, `pbsa_same_common_switching_direction`, and `pbsa_same_edge_reconvergence_only` variables are all set to `true`.

The NanoTime tool detects zero-cycle paths only if the launch and capture clocks meet the following conditions:

- They originate from the same source clock.
- They originate from the same source clock edge.
- They originate from the same cycle.

- How to calculate the amount of pessimism removal

  To remove pessimism based on the minimum delay difference of the same-edge common net, set the `pbsa_common_net_use_same_direction_delays` variable to `true` (the default is `false`) and keep the `pbsa_same_common_switching_direction` variable set to `false` (the default).

  When this variable is `true`, NanoTime calculates the rise pessimism (the difference between the launch and capture path delays for a rising transition) and the fall pessimism (the difference between the launch and capture path delays for a falling transition) at the common net. The smaller of the two values is removed. This method is consistent with the PrimeTime tool calculation method.

Common Point Determination for Differential Circuits

Figure 16-3 shows a simplified example of a differential network. If switching direction is not a concern for this circuit, the common net is the differential pair gclkn and gclkp, and the common point is the pair of input pins to the inverters that drive nets gclkn and gclkp.

If you want to require specific switching behavior when you search for common nets in differential circuits, use the `pbsa_differential_switching_direction` variable. Accepted values for the variable are `opposite`, `same`, `automatic`, and `ignore` (the default).

The `automatic` value directs NanoTime to infer the switching direction of the common net pair based on path tracing analysis. The `ignore` value directs the tool to ignore the switching direction and provides backward compatibility with previous NanoTime versions. Under most circumstances, the `automatic` and `ignore` values are equivalent. The `same` and `opposite` values are specialized selections that should be used only with a detailed understanding of the behavior of the design.

Figure 16-3 Differential Circuit Common Point

The `pbsa_same_common_switching_direction` variable has no effect when analyzing differential circuits. The `pbsa_allow_reconvergent_common_net` variable has the same effect for both differential and nondifferential circuits: when the variable is `false`, the search finds the first common net or differential pair, and when the variable is `true`, the search finds the last common net or differential pair. Differential pairs must be marked explicitly with the `set_differential` command to be considered.

For path-based slack adjustment on differential nets, the `-skew_min` and `-skew_max` options of the `set_differential` command increase pessimism for minimum and maximum delay analysis.

Common Point Determination for Clock Gates

Clock gate circuits take multiple clock inputs from the same clock domain and produce a shaped clock output signal. By definition, clock gates (pulsers and shapers) contain reconvergent clock paths.

Figure 16-4 shows a simple clock gate with two clock paths. The clock inputs originate at the same clock domain (CLK). After the clock paths diverge at net X, they pass through a varying number of logic inversions before reconverging at net Y, the output net.

Figure 16-4 General Reconvergent Clock Gate

The common point variables apply to clock gates the same as they do to other reconvergent circuits, as follows:

- To allow the reconvergent net at the clock gate output (net Y) to be the common point, set the `pbsa_allow_reconvergent_common_net` variable to `true`.
- To require matching switching directions at all common points (whether net X or net Y), set the `pbsa_same_common_switching_direction` variable to `true` (the default is `false`).
- To require matching switching directions only at a reconvergent common point (net Y), set the `pbsa_same_edge_reconvergence_only` variable to `true` (the default is `false`) and leave the `pbsa_same_common_switching_direction` variable set to `false`.

Reducing Common Path Signal Integrity Optimism

If signal integrity analysis is enabled, NanoTime removes all common path pessimism by default, including pessimism resulting from multicycle timing checks. However, this analysis might be optimistic for multicycle checks if the aggressors change between cycles.

To change the default behavior, set the `pbsa_include_common_si_deltas` variable to `true` (the default is `false`). In this mode, pessimism arising from signal integrity delta delays can only be removed if the launch and capture devices are driven by the same clock edge (zero-cycle checks).

The NanoTime tool detects zero-cycle paths only if the launch and capture clocks meet the following conditions:

- They originate from the same source clock.
- They originate from the same source clock edge.
- They originate from the same cycle.

The `pbsa_include_common_si_deltas` variable has an effect only if signal integrity analysis is enabled.

Variables That Control Common Point Definition

Table 16-4 summarizes the variables that control common point definition.

Table 16-4 Common Point Control Variables

Variable	Description
`pbsa_allow_reconvergent_common_net` (default: `false`)	When `false`, the common net is the first point of divergence. When `true`, the common net is the last point of divergence.
`pbsa_same_common_switching_direction` (default: `false`)	When `true`, the common net must switch in the same direction for both launch and capture paths. When `false`, there is no requirement on the switching direction.
`pbsa_same_edge_zero_cycle_reconvergence_only` (default: `false`)	When `true`, same-edge same-cycle reconvergence is allowed but same-edge multicycle reconvergence is not allowed. When `false`, performs standard pessimism removal.
`pbsa_same_edge_reconvergence_only` (default: `false`)	When `true`, a reconvergent common net must switch in the same direction for both launch and capture paths, but mismatched edges are allowed if the common net is the first divergent net. When `true`, the switching direction requirements are applied to all common points.
`pbsa_common_net_use_same_direction_delays` (default: `false`)	When `true`, the amount of pessimism removal is the difference between the rise pessimism and the fall pessimism.
`pbsa_differential_switching_direction` (default: `ignore`)	Specifies the required switching behavior for elements of a differential pair to be considered a common net. Accepted values are `opposite`, `same`, `automatic`, and `ignore`.
`pbsa_include_common_si_deltas` (default: `false`)	When `true`, pessimism removal is performed only for same-cycle timing checks during signal integrity analysis. When `false`, all common path pessimism is removed.

17

Signal Integrity Analysis

Signal integrity (or quality) within an integrated circuit can deteriorate due to crosstalk, the phenomenon by which a signal in one path affects the signal in another path. NanoTime analysis can include the effects of crosstalk on transition arrival times, timing slack, and steady-state signal levels.

Signal integrity delay and noise analysis requires a NanoTime Ultra license.

This chapter contains the following sections:

- Signal Integrity Concepts
- Overview of Signal Integrity Analysis
- Timing Windows for Signal Integrity Analysis
- Setting Up Signal Integrity Analysis
- SI Delay Analysis Procedure
- SI Noise Analysis Procedure
- Fanout Noise Analysis Procedure

Signal Integrity Concepts

Signal integrity is the ability of an electrical signal to carry information reliably and to resist the effects of high-frequency electromagnetic interference from nearby signals. Signal integrity can be degraded by crosstalk, which is the electrical interaction between physically adjacent nets due to capacitive cross-coupling.

Figure 17-1 shows a view of two parallel metal wires in an integrated circuit for two circuit geometries.

Figure 17-1 Coupling Capacitance With Different Feature Sizes

At smaller circuit geometries, interconnect lines are closer together and taller, causing the coupling capacitance between lines to increase. At the same time, parasitic capacitance to the substrate decreases as the lines become narrower, and cell delays decrease as transistors become smaller. Therefore, crosstalk effects increase in importance with each new technology.

Crosstalk can affect signal delays by changing the times at which signal transitions occur. For example, consider the signal waveforms A, B, and C on three cross-coupled nets, shown in Figure 17-2.

Figure 17-2 Transition Slowdown or Speedup Caused by Crosstalk

Because of capacitive cross-coupling, the transitions on nets A and C can affect the time at which the transition occurs on net B. A rising-edge transition on net A at the time shown in Figure 17-2 might cause the transition to occur later on net B, possibly contributing to a setup violation for a path containing net B. Similarly, a falling-edge transition on net C might cause the transition to occur earlier on net B, possibly contributing to a hold violation for a path containing net B.

Crosstalk also causes noise on steady-state nets. Figure 17-3 shows a noise bump on net B caused by the transition on net A. Large noise spikes might cause incorrect logic values to be propagated to the next gate in the path.

Figure 17-3 Noise Bump Due to Crosstalk

Cross-Coupling Models

Figure 17-4 shows the physical layout for a small portion of an integrated circuit and a detailed model of the circuit that includes cross-coupled capacitance. Each physical interconnection has some distributed resistance along the conductor and some parasitic capacitance to the substrate (ground) and to adjacent nets. A parasitic extraction tool divides each net into subnets and represents the distributed resistance and capacitance as a set of discrete resistors and capacitors.

Figure 17-4 Detailed Model of Cross-Coupled Nets

A detailed model provides a very accurate prediction of crosstalk effects in simulation. For an actual integrated circuit, a model might have too many circuit elements to process in a practical amount of time. To ensure a reasonable runtime, NanoTime filters cross-coupling capacitors and aggressor nets to remove from consideration capacitors and aggressors that are too small to have a significant effect. You control the filtering thresholds by setting variables.

Overview of Signal Integrity Analysis

The types of signal integrity analysis are as follows:

- Delay analysis: changes in arrival times on victim nets due to aggressor net transitions
- Noise analysis: changes in signal levels on victim nets due to aggressor net transitions
- Fanout noise analysis: an extension of noise analysis; the effect of noise on the fanout of a victim net

You can run delay and noise analysis separately or together. Many variables apply to all types of SI analysis, such as the variables that control net filtering.

Note:
The terms crosstalk analysis and signal integrity analysis are used interchangeably to refer to delay analysis. Noise analysis is explicitly referred to as crosstalk noise analysis or signal integrity noise analysis.

How Signal Integrity Delay Analysis Works

Signal integrity analysis is designed to be pessimistic. The aggressor nets that can switch within the arrival time window of a victim net are all assumed to switch in a direction that maximizes pessimism, as follows:

- For minimum-timing analysis, the aggressors are all assumed to switch in the *same* direction as the victim, making the delay of the victim net as small as possible.
- For maximum-timing analysis, the aggressors are all assumed to switch in the *opposite* direction as the victim net, making the delay of the victim net as large as possible.

NanoTime determines the worst-case changes in delay values and uses this additional information to calculate and report total slack values. The tool also reports the sources and amounts of the worst crosstalk delays so that you can change the design or the layout.

Signal integrity analysis is an iterative process in which each path tracing pass obtains increasingly accurate results, as illustrated in Figure 17-5.

Figure 17-5 NanoTime SI Analysis Flow

Signal integrity iterations work as follows:

- First iteration

  The goal of the first iteration is to quickly obtain worst-case delay values so that NanoTime can intelligently select the nets for the next iteration.

  In the first iteration, the tool does not consider transition timing windows. In other words, every aggressor net is assumed to have a transition at the worst possible time to affect other nets.

  All iterations, including the first, take logic constraints into account, such as victim-to-aggressor logic constraints, to avoid analyzing conditions that cannot occur.

- Second and subsequent iterations

  NanoTime considers timing windows to avoid analyzing transitions that do not affect other nets. After each iteration, NanoTime selects a new set of nets for analysis in the next iteration.

The tool selects the aggressor-victim pairs that have overlapping arrival windows and a large enough coupling capacitance to require calculation of a revised stage delay. For aggressor-victim pairs not reselected, the previously calculated delay values are retained. In each iteration, changes in calculated delays and arrival windows might cause a different set of nets to be reselected.

After each path-tracing iteration, NanoTime checks the to see if the exit criteria are satisfied. By default, the tool performs three iterations, which typically provides good results with reasonable runtime.

Signal integrity analysis can be used at the same time as other iterative analysis features, such as timing-based multi-input switching analysis and differential skew analysis. If more than one type of iterative analysis is enabled, NanoTime updates the results for all enabled features during each path tracing iteration until all iteration exit criteria for the enabled features are satisfied.

Note:

In rare cases, oscillations between iterations might result, especially when strong aggressors are logically constrained to switch in a direction that removes pessimism. You should always use the worst-case results for your analysis and choose iteration exit criteria that produce the worst-case results.

How Signal Integrity Noise Analysis Works

NanoTime calculates the effects of noise from the following sources:

- Crosstalk noise (capacitive coupling)

  NanoTime considers the coupling capacitance between the aggressor nets and the victim net, the arrival windows of aggressor transitions, the drive characteristics of the aggressor nets, and the steady-state resistance characteristics of the victim net.

- Propagated noise

  Propagated noise on a victim net is caused by noise at an input of the cell that is driving the victim net. NanoTime calculates propagated noise at a cell output, given the noise bump at the cell input and the load on the cell output.

- User-defined injected noise

  You can inject noise on any pin or port in the design. User-defined noise can either override or add to any existing noise. User-defined noise that propagates forward might also affect the amount of fanout noise. However, the tool does not explicitly support user-defined fanout noise.

Four types of noise bumps are caused by aggressor net transitions: above the ground rail (above low), below the ground rail (below low), above the supply rail (above high), and below the supply rail (below high).

Noise bumps between the two rail voltages (above low and below high) might cause logic failure if they exceed the logic thresholds of the technology. Bumps outside of the rail voltages (below low and above high) might forward-bias pass gates at the inputs of flip-flops and latches, allowing incorrect values to be latched.

By default, NanoTime assumes a worst-case combination of aggressor and victim states and computes an injected noise waveform on the victim node. The maximum noise level is compared to the user-specified noise margins to determine whether a noise violation exists.

Timing Windows for Signal Integrity Analysis

The range of possible switching times defines a timing window for a net. Signal integrity effects occur only when the victim and aggressor timing windows overlap.

When an overlap occurs, NanoTime calculates the effect of a transition on the aggressor net that occurs at the same time as a transition on the victim net. The analysis takes into account the drive strengths and coupling characteristics of the two nets.

The tool calculates the delta delay for each load pin of a net. If the aggressor timing window only partially overlaps with the victim's timing window, the partial effect (smaller delta delay) is evaluated.

Figure 17-6 shows how timing windows can overlap. In this example, victim V1 is coupled with aggressor A1. The timing arrival windows are 2 ns to 6 ns for the aggressor, and 5 ns to 9 ns for the victim. Because the victim timing window overlaps with the aggressor timing window, even if only by a small amount, NanoTime calculates the delta delay of the victim net due to this aggressor.

Figure 17-6 Victim-Aggressor Switching Time Alignment

The variables in Table 17-1 affect how the NanoTime tool computes timing windows. The `si_timing_window_overlap_tolerance` variable specifies a margin in user time units where two timing windows are considered to be overlapping. Half of this value is added to the beginning and half to the end of a computed timing window. If some aggressors do not have valid timing windows, the `si_aggressor_transition_default_fall` and `si_aggressor_transition_default_rise` variables provide default transition times for them; this can happen if all paths were not exercised during path tracing.

Table 17-1 Variables for Logic Constraints in Crosstalk Analysis

Variable	Default
`si_timing_window_overlap_tolerance`	`0.0`
`si_aggressor_transition_default_fall`	`0.05`
`si_aggressor_transition_default_rise`	`0.05`

See Also

- Timing Window Overlap
- Timing Windows For Multiple Aggressors
- Inversion Constraints Between Aggressor Nets
- Timing Windows for Dangling Nets

Timing Window Overlap

The relationship between the arrival times of signals at aggressor and victim nets is very important for the determination of overlap windows in signal integrity analysis.

By default, the arrival time (switching time) of a signal is considered to be the time at which the signal rises to 50 percent of the supply voltage for a rising transition or falls to that level for a falling transition. The transition of a signal is defined to begin and end at ground or at the full supply voltage, extrapolated from the part of the transition defined by the RC slew variables. The overlap timing window spans the time between the beginning of the aggressor transition and the end of the victim transition.

Figure 17-7 and Figure 17-8 show four scenarios for a falling aggressor net signal and a rising victim net signal. The signal integrity risk is that the falling aggressor signal might pull down the victim signal, causing the victim net to switch later than expected. The most vulnerable part of the victim signal transition is the first half, before the signal reaches the 50 percent level. After this point, the aggressor net is less likely to affect the victim net.

Figure 17-7 Victim and Aggressor Net Switching Scenarios

NanoTime evaluates the SI delay effect as follows:

- In scenario (a), the aggressor net switches before the victim net, but the transitions do not overlap. There is no SI delay impact, because the capacitive coupling is only relevant during switching.
- In scenario (b), the aggressor net switches during the first half of the victim transition. In other words, the victim arrival time occurs within the overlap timing window. This situation produces the largest SI delay effect on the victim.
- In scenario (c), the aggressor arrival time occurs after the victim arrival time, during the second half of the victim transition, but the aggressor transition begins before the victim arrival time. (The victim arrival time occurs within the overlap timing window.) NanoTime includes this situation in SI delay analysis.
- In scenario (d), the aggressor arrival time occurs after the victim arrival time and the aggressor transition begins after the victim arrival time. In other words, the victim arrival time does not occur within the overlap timing window. Even though the aggressor and victim transitions overlap, this situation is considered to have minimal SI delay impact and NanoTime does not include it in SI delay analysis.

Figure 17-8 Victim and Aggressor Net Switching Scenarios

If an aggressor net and a victim net switch at nearly the same time, small changes in the design might cause the scenario to change from overlapping to nonoverlapping or vice versa, possibly leading to larger than expected changes in crosstalk effects. To avoid this occurrence, you can add an optional overlap tolerance to expand all timing windows.

The following settings affect the signal switching times and slews and therefore might change the timing window during signal integrity analysis:

- Timing model parameters such as the `slew_derate_from_library` parameter and the slew threshold parameters
- NanoTime variables such as the `rc_slew_*`, `rc_output_threshold_*`, and `rc_input_threshold_*` variables
- The `set_measurement_threshold` command

See Also

- Timing Windows for Signal Integrity Analysis
- Timing Windows For Multiple Aggressors
- Inversion Constraints Between Aggressor Nets
- Timing Windows for Dangling Nets

Timing Windows For Multiple Aggressors

When there are multiple aggressors in the design, NanoTime finds the combination of aggressors that produce the worst crosstalk effect and calculates the delta delay for that combination. An example of multiple aggressor nets with simple timing windows that do not overlap is shown in Figure 17-9.

In this example, the victim net V has three cross-coupled aggressors. Because aggressor A1's timing window does not overlap with the victim's timing window, only aggressors A2 and A3 are considered in the analysis. A2 and A3 do not overlap with each other; therefore, they cannot both affect the victim at the same time. NanoTime determines the stronger of the two aggressors and sets its switching to occur in the direction that increases pessimism.

Figure 17-9 Nonoverlapping Aggressor

If different aggressor nets are based on different clocks, the timing windows of multiple aggressors might overlap in a complex way that changes throughout the clock cycles. For example, in Figure 17-10, rising transitions of aggressor A1 are triggered by rising transitions of clock CK1, while aggressor A2 is controlled by clock CK2. At time t1, only aggressor A2 overlaps with the victim net transition. However, at time t2, aggressors A2 and A2 both overlap with the victim net, leading to a stronger signal integrity effect.

To avoid optimism, the NanoTime tool evaluates aggressor and victim timing windows across multiple clock cycles to determine the worst-case overlap.

Figure 17-10 Aggressors With Different Clock Periods

See Also

- Timing Windows for Signal Integrity Analysis
- Timing Window Overlap
- Inversion Constraints Between Aggressor Nets
- Timing Windows for Dangling Nets

Inversion Constraints Between Aggressor Nets

If A2 and A3 overlap, as shown in Figure 17-11, and there is no inversion constraint between them, both A2 and A3 are analyzed to switch in the direction of increased pessimism. Since the victim signal is rising in the overlapping timing window, the most pessimistic directions for the two aggressors are falling transitions for maximum delay analysis. (Conversely, the aggressors would rise for minimum delay analysis.)

Figure 17-11 Overlapping Aggressors With No Inversion Constraint

If there is an inversion constraint between aggressors A2 and A3, they switch in opposite directions, as shown in Figure 17-12. In this case, they exert a smaller total effect on the victim net. You can choose whether to consider this relationship in crosstalk delay analysis. If you enable aggressor logic pessimism reduction, and A2 is stronger than A3, NanoTime selects the stronger aggressor A2 to switch in the most pessimistic (falling) direction; the weaker aggressor A3 switches in the opposite (rising) direction. The effects of the two aggressors partially cancel so that the combined effect is the least pessimistic of the three scenarios.

Figure 17-12 Overlapping Aggressors With Inversion Constraint

See Also

- Timing Windows for Signal Integrity Analysis
- Timing Window Overlap
- Timing Windows For Multiple Aggressors
- Timing Windows for Dangling Nets

Timing Windows for Dangling Nets

Dangling nets (floating nets) are nets that are connected on only one end. Dangling nets are not valid endpoints during NanoTime path tracing. As a result, they do not have arrival times, transition times, or timing windows.

However, dangling nets might still act as aggressors. For signal integrity analysis, NanoTime assigns transition times based on the `si_aggressor_transition_default_rise` and `si_aggressor_transition_default_fall` variables. The default of both variables is 0.05 time units.

You can set these variables to modify the analysis, as follows:

- For a more conservative analysis, set both variables to a smaller value. The default transition value might not be suitable for all process technologies. A value that is too large (too slow) might cause overly optimistic results.
- For a less conservative analysis, set both variables to a larger value. A very large value effectively ignores all dangling nets.

By default, NanoTime uses the following assumptions about timing windows:

- For maximum delay analysis, aggressor timing windows always overlap with the victim.
- For minimum delay analysis, aggressor timing windows never overlap with the victim.

If these assumptions result in overly pessimistic results, you can force a dangling net to become a valid path endpoint for path tracing by identifying it as an output port and adding an output delay constraint.

See Also

- Timing Windows for Signal Integrity Analysis
- Timing Window Overlap
- Timing Windows For Multiple Aggressors
- Inversion Constraints Between Aggressor Nets

Setting Up Signal Integrity Analysis

The following preparation helps to achieve good results with a reasonable runtime:

- Make sure that your design passes normal static timing analysis without crosstalk analysis enabled. There should be no timing violations.
- Make sure that the parasitic extraction tool generates a standard SPEF or DSPF file with coupling capacitors, and that the settings generate a reasonable number of capacitors. If the extraction tool supports the filtering of small capacitors based on a threshold, it might be more efficient to let the extraction tool perform the filtering instead of reading many small capacitors into the NanoTime tool.
- NanoTime ignores any capacitor between different parts of the same net. If possible, configure the extraction tool to suppress generation of self-coupling capacitors.

The following steps are common to both delay and noise analysis:

1. Before executing the `link_design` command, enable one or more analysis types by setting the appropriate variables:
   - For crosstalk delay analysis, set the `si_enable_analysis` variable to `true`.
   - For crosstalk noise analysis, set the `si_enable_noise_analysis` variable to `true`.
   - For fanout noise analysis, set the `si_enable_noise_fanout_analysis` variable to `true`.
2. If you are using a SPICE netlist with embedded parasitics, use the `link_design` command with the `-keep_capacitive_coupling` option.
3. If you are reading parasitic data from a SPEF file:
   - Set the `parasitics_read_variation` variable to `true` if you want to read and use the minimum and maximum capacitance values in addition to the typical values. The default is `false`, which enables reading only the typical values.
   - Use the `read_parasitics` command with the `-keep_capacitive_coupling` option.
4. If you are reading parasitic data from a DSPF file, use the `read_parasitics` command with the `-keep_capacitive_coupling` option.
5. (Optional) Set the `si_enable_aggressor_logic_pessimism_reduction` variable to `true` to reduce pessimism. This variable affects both delay and noise analysis. You can define additional logic constraints to further reduce pessimism.
6. (Optional) Set variables to account for cross-capacitance variability.
7. (Optional) Set variables that affect net exclusion.

See Also

- Generating Parasitic Netlist Files With an Extraction Tool
- Accounting for Capacitance Variability
- Filtering Victim and Aggressor Nets
- Logic Constraints in Signal Integrity Analysis

Accounting for Capacitance Variability

The circuit layout and the process specifications determine nominal coupling capacitance values. However, variations in wafer processing cause changes to the coupling capacitances. To help account for these changes, you can add variability to the nominal coupling capacitance values that NanoTime uses for crosstalk delay and noise analysis.

You can apply capacitance variability in either of two mutually exclusive ways:

- Use minimum and maximum capacitance values obtained from a SPEF parasitics file to calculate capacitance ranges for individual nets.

  In this case, NanoTime calculates the total of the minimum, maximum, and typical capacitances on each net. The average percentage variations above and below the typical capacitance are applied to each of the individual capacitances on that net. All capacitances on a net receive the same percentage variation.

  Set the `parasitics_read_variation` variable to `true` to read net-specific capacitances from the SPEF file.

- Set global variables to specify one set of percentage variations to use throughout the design. These variations apply to the nominal coupling capacitance values read from a DSPF or SPEF file. If net-specific variations are read in, the net-specific values override the global values for those nets. The global values apply for all other nets.

  Two variables control the coupling capacitance variability:

  - The `parasitics_coupling_cap_variation_max` variable specifies the percentage variation above the nominal capacitance.
  - The `parasitics_coupling_cap_variation_min` variable specifies the percentage variation below the nominal capacitance.

  Both variables default to 0.0, which means that NanoTime does not consider any coupling capacitance variation. Both variables allow only positive values, and the `parasitics_coupling_cap_variation_min` variable must be less than 1.0 to prevent calculation of a negative capacitance.

The `coupling_capacitors` attribute is associated with a victim net and contains a list of the coupling capacitors acting on the victim net. The `coupling_capacitors_max` and `coupling_capacitors_min` attributes contain the values of those capacitors after adjustment for variation.

The list of aggressor nets acting on a victim net is available in the `effective_aggressors` attribute associated with that victim net. The `effective_aggressors_coupling_cap` attribute associated with the victim net contains the coupling capacitances associated with the effective aggressor nets. The `effective_aggressors_coupling_cap_max` and `effective_aggressors_coupling_cap_min` attributes contain the values of the coupling capacitances after adjustment for variation.

During delay and noise analysis, NanoTime applies the coupling capacitance variations to achieve the worst-case result.

For maximum SI delay calculation,

- Maximum coupling capacitance is used for aggressors that switch in the opposite direction from the victim.
- Minimum coupling capacitance is used for aggressors that switch in the same direction as the victim.
- Maximum coupling capacitance is used for nonswitching and filtered aggressors.

For minimum SI delay calculation,

- Maximum coupling capacitance is used for aggressors that switch in the same direction as the victim.
- Minimum coupling capacitance is used for aggressors that switch in the opposite direction from the victim.
- Minimum coupling capacitance is used for nonswitching and filtered aggressors.

For SI noise calculation,

- Maximum coupling capacitance is used for all switching aggressors.
- Minimum coupling capacitance is used for nonswitching and filtered aggressors.

See Also

- Setting Up Signal Integrity Analysis
- Filtering Victim and Aggressor Nets
- Logic Constraints in Signal Integrity Analysis

Filtering Victim and Aggressor Nets

NanoTime filters victim nets and aggressor nets that have a small effect on the analysis. The variables listed in Table 17-2 control the filtering thresholds. Set the variables to balance accuracy (less filtering) and runtime (more filtering).

Table 17-2 Crosstalk Parasitic Filtering Variables

Filter	Variable	Default
Victim net	`si_filter_total_aggr_xcap`	0.0
	`si_filter_total_aggr_xcap_to_gcap_ratio`	0.0
	`si_aggressive_filtering_total`	false
Aggressor net	`si_filter_per_aggr_xcap`	0.0
	`si_filter_per_aggr_xcap_to_gcap_ratio`	0.0
	`si_filter_per_aggr_to_average_aggr_xcap_ratio`	0.0
	`si_aggressive_filtering_per_aggr`	false

Filtering Victim Nets

A victim net is filtered if its total coupling capacitance is small in absolute terms or in comparison to its total ground capacitance. When a victim net is filtered, it is removed from analysis as a victim net. However, the same net can still be an aggressor to another net.

If the `si_aggressive_filtering_total` variable is `false`, both of the following conditions must be true for a net to be filtered. If the variable is `true`, only one condition must be met.

- The total coupling capacitance between the victim net and all aggressor nets is less than the value of the `si_filter_total_aggr_xcap` variable.
- The total coupling capacitance divided by the total ground capacitance of the net is less than the value of the `si_filter_total_aggr_xcap_to_gcap_ratio` variable.

Filtering Aggressor Nets

An aggressor net is filtered when the coupling capacitance between an aggressor net and victim net is small compared to the victim's total ground capacitance, compared to the victim's total coupling capacitance, or in absolute terms. When an aggressor net is filtered, it is removed from analysis for that victim net. However, the net can still be an aggressor to other nets.

If the `si_aggressive_filtering_per_aggr` variable is `false`, all of the following conditions must be true for a net to be filtered. If the variable is `true`, only one condition must be met.

- The total coupling capacitance between the aggressor net and the victim net is less than the value of the `si_filter_per_aggr_xcap` variable.
- The total coupling capacitance between the aggressor and victim nets divided by the total ground capacitance of the victim net is less than the value of the `si_filter_per_aggr_xcap_to_gcap_ratio` variable.
- The total coupling capacitance between the aggressor and victim nets divided by the average coupling capacitance between the victim net and all of its aggressor nets is less than the value of the `si_filter_per_aggr_to_average_aggr_xcap_ratio` variable.

If capacitance variation is enabled, NanoTime uses the largest calculated capacitance values during the filtering operation to maximize the number of aggressors kept for analysis.

See Also

- Setting Up Signal Integrity Analysis
- Accounting for Capacitance Variability
- Logic Constraints in Signal Integrity Analysis

Logic Constraints in Signal Integrity Analysis

Considering logic constraints during signal integrity analysis reduces delay pessimism. For example, a mutual exclusion constraint might specify that only one net in a group of aggressor nets can switch, resulting in a smaller effect on the victim net compared to simultaneous switching.

The NanoTime tool considers the following logic constraints during signal integrity delay and noise analysis:

- Constraints between victim nets and aggressor nets

  The tool automatically considers structurally-inferred (topology-induced) constraints.

- Constraints between aggressor nets

  By default, the tool does not consider logic constraints between aggressor nets. The aggressor nets for a single victim net are assumed to switch in the same direction, which might be pessimistic. You can enable the evaluation of aggressor logic constraints, which includes both structually-inferred and manually specified constraints.

Application of Logic Constraints

Set the `si_enable_aggressor_logic_pessimism_reduction` variable to `true` to consider logic constraints between aggressors.

If logic constraint evaluation is enabled, it is used during the first signal integrity iteration by default. This improves the accuracy of the first iteration at the cost of increased runtime. The accuracy of later iterations is not affected. You can disable this selection by setting the `si_enable_first_iteration_pessimism_reduction` variable to `false`, which might improve the total runtime for cases where SI analysis requires multiple iterations.

Table 17-3 and Table 17-4 summarize how the NanoTime tool evaluates logic constraints during delay and noise analysis. In these tables, V1, V2, and V3 represent the following variables:

- V1 is the `si_enable_first_iteration_pessimism_reduction` variable.
- V2 is the `si_enable_aggressor_logic_pessimism_reduction` variable.
- V3 is the `si_enable_noise_aggressor_windows_pessimism_reduction` variable.

Table 17-3 Crosstalk Delay Analysis Options

Iteration	Use timing windows	Aggressor-victim logic constraints	Aggressor-aggressor logic constraints
1	No	Yes, if V1 = true	Yes, if V1 = true and V2 = true
2 and up	Yes	Yes	Yes, if V2 = true

Table 17-4 Crosstalk Noise Analysis Options

Timing windows	Aggressor-victim logic constraints	Aggressor-aggressor logic constraints
Yes, if V2 = true or V3 = true	Inversion only	Yes, if V2 = true

Types of Logic Constraints

Inversion Constraints

The NanoTime tool infers simple inverting and noninverting logic from the design. For example, differential net pairs have an inverting relationship by definition. You can use the `set_logic_constraint -invert` command to specify additional inversion constraints.

If inversion constraints overlap, NanoTime selects a set of switching directions that satisfies the constraints while maximizing the coupling capacitance effect.

Mutual Exclusion Constraints

Among a set of aggressor nets, mutual exclusion constraints define a smaller subset of switching nets, resulting in a less pessimistic delay effect on the victim net.

The `-one_hot` and `-one_off` options of the `set_logic_constraint` command are mutually exclusive. The `-one_hot` logic constraint among a set of aggressors is treated the same as the `-at_most_one_hot` logic constraint.

If a mutual exclusion constraint specifies a set of nets that is a subset of the nets in a second mutual exclusion constraint, the constraint with the largest set of nets takes priority.

If some of the nets in a mutual exclusion constraint are also specified in an inversion constraint, the inversion constraint takes priority for those nets. The mutual exclusion constraint is honored for the remaining nets.

If mutual exclusion constraints partially overlap, NanoTime attempts to allocate the specified nets into a set of nonoverlapping mutual exclusion constraints that maximizes the amount of removed capacitance.

Constraints That Do Not Affect Signal Integrity Analysis

Constraints set with the `-nand` and `-nor` options of the `set_logic_constraint` command are not considered during signal integrity analysis.

A net that is set to a fixed logic state (0 or 1) is not used for signal integrity analysis. You can optionally use the `-enable_trace_from` option of the `set_case_analysis` command to allow path tracing in the fanout cone of a port or pin with a fixed logic state. However, the interaction of multiple `set_case_analysis` commands is complex. For signal integrity analysis, minimize the number of ports and pins that you set to a fixed logic state.

Precedence For Aggressor Logic Constraints

Specific aggressor nets might be included in multiple logic constraints. During signal integrity analysis, NanoTime applies only one constraint per aggressor net.

By default, inversion constraints have priority over mutual exclusion constraints, and user-specified constraints have priority over topology-induced constraints. In addition, you can elevate the priority of a user-specified logic constraint by using the `-crosstalk_force` option of the `set_logic_constraint` command in addition to the option that defines the constraint. The order of precedence for overlapping logic constraints is as follows, from highest to lowest priority:

1. Forced user-specified inversion constraints
2. Forced user-specified or topology-induced mutual exclusion constraints
3. Unforced user-specified inversion constraints
4. Unforced user-specified or topology-induced mutual exclusion constraints
5. Topology-induced inversion constraints

The following usage notes apply to the `-crosstalk_force` option:

- Forcing logic constraints might cause logic constraint conflicts.
- Forcing logic constraints might cause crosstalk delta delays to become more pessimistic or more optimistic.

Figure 17-13 illustrates overlapping user-specified inversion constraints. The NanoTime tool honors the forced constraint and ignores the unforced constraint.

By contrast, if the two constraints are both defined with the same status (both with or both without the `-crosstalk_force` option), NanoTime attempts to satisfy the constraints while also maximizing the aggressor impact based on the coupling capacitance. Forcing one constraint to take priority might result in a less pessimistic delay compared to letting the tool control the conditions.

Figure 17-13 Forced Inversion Constraint

In Figure 17-14, a user-specified inversion constraint overlaps with a mutual exclusion constraint. The user-specified inversion constraint takes priority. The mutual exclusion constraint is ignored because it has only one remaining net after the overlapping nets are removed from its scope. In a similar case where the mutual exclusion constraint has two or more remaining nets, the mutual exclusion constraint is applied to those remaining nets.

Figure 17-14 Overlapping Constraints

In Figure 17-15, the mutual exclusion constraint is defined with the `-crosstalk_force` option, which elevates its priority over the inversion constraint. In this case, the inversion constraint is ignored.

Figure 17-15 Forced Mutual Exclusion Constraint

To report logic constraints that have been set with the `set_logic_constraint` command, use the `report_logic_constraint` command. This report indicates whether a constraint is a forced constraint. To remove logic constraints that have been set with the `set_logic_constraint` command, use the `remove_logic_constraint` command.

Effect of Logic Constraints on Noise Analysis

The following additional considerations affect signal integrity noise analysis:

- Noise analysis runtime is highly dependent on the number and size of aggressor logic constraints. To trade off runtime versus the diminishing benefits of additional pessimism reduction, NanoTime ignores the analysis of logic constraints involving some weaker noise aggressors and considers such aggressors to switch independently.
- Aggressors with inversion logic constraints to other aggressors that switch in the opposite direction from the intended noise type are treated as nonswitching nets during noise analysis.

See Also

- Setting Up Signal Integrity Analysis
- Accounting for Capacitance Variability
- Filtering Victim and Aggressor Nets

SI Delay Analysis Procedure

In this mode, NanoTime calculates the changes in delays due to crosstalk. To perform SI delay analysis, follow this procedure:

1. Perform the setup steps common to all SI analysis, as described in Setting Up Signal Integrity Analysis.
2. (Optional) Specify nets to be included or excluded for analysis by using the `set_si_delay_analysis` command.
3. (Optional) Specify parameters that determine the accuracy and speed of the delay analysis effort, such as the criteria for ending the iterations and whether or not to analyze all paths.
4. Execute the `trace_paths` command at least one time for a conservative analysis, or two or more times to converge on more accurate crosstalk analysis results.
5. (Optional) Reselect nets for analysis and use the `continue_trace` command to run additional path tracing iterations.
6. Generate reports to examine the crosstalk delay effects.

Setting the Analysis Mode

By default, NanoTime analyzes SI delays for all paths through the victim net. The tool calculates the largest possible delta delay based on the victim and aggressor arrival windows, then applies that delay to all paths through the victim net. This approach guarantees a conservative analysis.

However, the analysis might be overly pessimistic because the worst delta delay might not result from the maximum delay path. To reduce pessimism and possibly reduce runtime, set the `si_xtalk_delay_analysis_mode` variable to `worst_path` (the default is `all_paths`). In this mode, NanoTime calculates only the crosstalk affecting the worst path.

This approach might allow inaccurate analysis of subcritical paths. In addition, if subcritical paths have very small positive slack values before crosstalk analysis, using the `worst_path` mode might fail to find timing violations after crosstalk analysis.

See Also

- Selecting or Excluding Nets for SI Delay Analysis
- Controlling SI Delay Iteration Exit Criteria
- Reporting SI Delay Analysis

Selecting or Excluding Nets for SI Delay Analysis

For the initial delay calculation iteration, NanoTime selects all cross-coupled nets for analysis. You can exclude specific nets as aggressors or as victims by using the `set_si_delay_analysis -exclude` command.

For subsequent delay calculation iterations, NanoTime reselects the aggressor-victim pairs that have overlapping arrival windows and a large enough coupling capacitance to require calculation of a revised stage delay. You can also have NanoTime reselect specific nets by using the `set_si_delay_analysis -reselect` command.

The `set_si_delay_analysis` command lets you include or exclude nets for crosstalk delay analysis in the following ways:

- Reselect specific nets in all subsequent iterations, even if they do not meet the usual reselection criteria
- Set specific nets to use infinite arrival windows
- Exclude specific nets as aggressors
- Exclude specific nets as victims
- Exclude specific aggressor-victim relationships between net pairs

Table 17-5 lists the variables that affect net selection and reselection.

Table 17-5 Variables That Affect Net Selection in SI Delay Analysis

Variable	Default
`si_xtalk_reselect_delta_delay`	`0.0`
`si_xtalk_reselect_delta_delay_ratio`	`0.0`
`si_timing_window_overlap_tolerance`	`0.0`
`si_aggressor_transition_default_fall`	`0.05`
`si_aggressor_transition_default_rise`	`0.05`
`si_exclusion_cap_factor_max`	`1.0`
`si_exclusion_cap_factor_min`	`1.0`

Net Reselection

NanoTime reselects any net that satisfies either or both of the following conditions:

- The change in stage delay (positive or negative) caused by crosstalk in the previous iteration exceeds the amount specified by the `si_xtalk_reselect_delta_delay` variable. For example, a setting of 5 causes a net to be reselected if its delay value changes by 5 ps or more from the previous iteration.
- The change in stage delay (either positive or negative) caused by crosstalk analysis in the previous iteration, when divided by the total stage delay, gives a ratio that exceeds the amount specified by the `si_xtalk_reselect_delta_delay_ratio` variable. For example, a setting of 0.95 causes a net to be reselected if its calculated stage delay is less than or equal to 95 percent of its previous value.

Stage delay is the delay of a stage (one cell and its fanout net). Crosstalk analysis in the previous iteration might have caused a change in the calculated worst-case delay: a decrease in delay for a minimum analysis or an increase in delay for a maximum analysis. If the amount of change is large enough, the net in that stage is reselected for further analysis.

Typically, the change in calculated delay becomes smaller and smaller for successive analysis iterations, as the analysis becomes more and more accurate. Reselecting nets based on the amount of delay change is a way to ensure accurate results.

The `set_si_delay_analysis` command lets you reselect specific nets in all subsequent iterations for crosstalk analysis, even if those nets do not meet the usual reselection criteria.

For example, you might want NanoTime to analyze reset signals (RESET1, RESET2, and so on) for crosstalk effects in all analysis iterations, even though they might not meet the usual reselection criteria. To ensure that these nets are reselected, enter the following:

```
nt_shell> set_si_delay_analysis -reselect [get_nets RESET*]
```

Note:
Forced reselection of nets might cause a significant increase in runtime.

To generate a report of the nets affected by the `set_si_delay_analysis` command, use the `report_si_delay_analysis` command. To remove the effects of the `set_si_delay_analysis` command, use the `remove_si_delay_analysis` command.

Excluding Nets From SI Analysis

You can exclude nets from signal integrity analysis by using the `-exclude` option of the `set_si_delay_analysis` command. Additional options allow you to refine the exclusion conditions.

Note:
If you exclude nets from signal integrity analysis, the NanoTime tool does not consider logic constraints between them. This includes logic constraints between excluded nets and logic constraints between excluded nets and included nets.

You can refine the exclusion conditions, as follows:

- To exclude maximum-delay (setup) or minimum-delay (hold) analysis, use the `set_si_delay_analysis` command with the `-exclude` option and either the `-max` or `-min` option.
- To exclude a specific net from consideration as a victim net, use the `-exclude` option with the `set_si_delay_analysis` command. In the following example, the clock net CLK1 is excluded. NanoTime does not consider net CLK1 to be a potential victim net for crosstalk delay analysis, thereby reducing the analysis runtime. However, CLK1 can still be considered as an aggressor net.

  ```
  nt_shell> set_si_delay_analysis -exclude -victims [get_nets CLK1]
  ```

- To exclude specific aggressor-victim net pair relationships, use the `-exclude` option with both the `-victims` and `-aggressors` options. For example, you might know that the scan clock signals (SCN_CLK_*) and clock network signals (CLK_NET_*) in your design do not affect each other for timing. Eliminate these crosstalk effects from consideration as follows:

  ```
  nt_shell> set_si_delay_analysis -exclude -aggressors \
        [get_nets SCN_CLK_*] -victims [get_nets CLK_NET*]
  nt_shell> set_si_delay_analysis -exclude -victims \
        [get_nets SCN_CLK_*] -aggressors [get_nets CLK_NET*]
  ```

  The first of these two commands removes from consideration any SCN_CLK_* signal as an aggressor to any CLK_NET* signal as a victim during crosstalk delay analysis. The second command removes from consideration the aggressor-victim relationships of these signals in the opposite direction. However, these signals can still be considered aggressors or victims relative to signals not mentioned in the commands. For example, CLK_NET1 can still be considered an aggressor to CLK_NET2 as a victim.

Effect of Removing Nets on Capacitance

Manually disabling or excluding aggressor nets, victim nets, or aggressor-victim net pairs from signal integrity analysis affects the treatment of coupling capacitors that exist between them. When you remove nets from the analysis, the coupling capacitors are split to ground.

You can modify the split capacitance values to increase or decrease the analysis pessimism related to the excluded nets. Split capacitances are multiplied by a coupling capacitance factor that comes from either of two sources. In all cases, the allowable range of values is 0.0 to 2.0 and the defaults are 1.0.

Set the coupling factor as follows:

- Use the `-coupling_factor` option together with the `-exclude` option of the `set_si_delay_analysis` command to define a coupling factor to use with a specific net.
- Set the `si_exclusion_cap_factor_min` and `si_exclusion_cap_factor_max` variables for minimum and maximum delay analysis. NanoTime uses these values whenever you use the `set_si_delay_analysis -exclude` command without a net-specific coupling factor.

For example, if the coupling capacitance between nets AGGR1 and VICT1 is 4 pF, the following command excludes both nets from crosstalk analysis and adds a 4 pF capacitor to ground from each of them:

```
nt_shell> set_si_delay_analysis -exclude -aggressors AGGR1 -victims VICT1
```

In the following example, the original 4 pF coupling capacitor is replaced by two 8 pF grounded capacitors (one on each net) during maximum delay analysis and 0 pF during minimum delay analysis:

```
nt_shell> set si_exclusion_cap_factor_max 2.0
nt_shell> set si_exclusion_cap_factor_min 0.0
nt_shell> set_si_delay_analysis -exclude -aggressors AGGR1 -victims VICT1
```

The next example demonstrates the use of different net-specific coupling factors for min and max analysis. The original 4 pF coupling capacitor is replaced by two 4.8 pF grounded capacitors (one on each net) during maximum delay analysis and two 3.2 pF grounded capacitors (one on each net) during minimum delay analysis.

```
nt_shell> set_si_delay_analysis -exclude -aggressors AGGR1 \
          -victims VICT1 -max -coupling_factor 1.2
nt_shell> set_si_delay_analysis -exclude -aggressors AGGR1 \
          -victims VICT1 -min -coupling_factor 0.8
```

Removing Net Exclusions

To reverse the effects of the `set_si_delay_analysis` command, use the `remove_si_delay_analysis` command. For example, suppose that the design has a victim net V that has three aggressors, A1, A2, and A3. To see a list of the aggressors for net V, use the following command:

```
nt_shell> get_attribute -class net [get_net V] aggressors
A1 A2 A3
```

To exclude the net V from consideration as a victim for crosstalk, use the following command. The excluded list for net V would be {A1 A2 A3}:

```
nt_shell> set_si_delay_analysis -exclude -victims V
```

Even though the exclusion on net V implies that A1 is excluded as an aggressor for net V, you cannot use the `remove_si_delay_analysis -aggressors` command to remove the

exclusion between nets V and A1. This is because no exclusion was directly set on net A1. Therefore, after the following command, the excluded list for net V is still {A1 A2 A3}:

```
nt_shell> remove_si_delay_analysis -aggressors A1
Warning: Cannot remove global separation or exclusion that
was not set on net(s) A1. (XTALK-107)
```

Similarly, even though the exclusion on net V implies that A2 is excluded as an aggressor for V, this exclusion cannot be removed by using the victim-aggressor option because no exclusion was directly set for the victim-aggressor pair V and A2. Therefore, after the following command, the excluded list for net V would still be {A1 A2 A3}:

```
nt_shell> remove_si_delay_analysis -victims V -aggressors A2
Warning: Cannot remove global separation or exclusion that
was not set on net(s) V A2. (XTALK-107)
```

Only exclusions that were applied directly can be removed. If you issue the following command, the effect of the `set_si_delay_analysis -exclude -victims V` command is reversed, and the excluded list is empty:

```
nt_shell> remove_si_delay_analysis -victims V
```

See Also

- SI Delay Analysis Procedure
- Controlling SI Delay Iteration Exit Criteria
- Reporting SI Delay Analysis

Controlling SI Delay Iteration Exit Criteria

NanoTime calculates crosstalk effects using an iterative process. The first iteration uses a fast, conservative analysis mode that does not consider the timing windows between aggressor and victim nets. In successive iterations, the results become increasingly accurate (less pessimistic) as the tool narrows down the transition windows, allowing more and more potential crosstalk effects to be eliminated because they do not overlap.

By default, NanoTime exits from the loop after three iterations, which provides good results in a reasonable amount of time. You can optionally specify other types of exit criteria to cause more iterations and obtain more accurate results at the expense of additional runtime.

You can use signal integrity analysis at the same time as timing-based multi-input switching analysis and differential skew analysis, which are also iterative. If f more than one type of iterative analysis is enabled, NanoTime updates the results for all enabled features during each path tracing iteration until all exit criteria are satisfied.

Table 17-6 lists the variables that control loop exit criteria.

Table 17-6 Crosstalk Delay Analysis Exit Criteria Variables

Variable	Default
`si_xtalk_exit_on_max_iteration_count`	3
`si_xtalk_exit_on_min_delta_delay`	0.0
`si_xtalk_exit_on_max_delta_delay`	0.0
`si_xtalk_exit_on_number_of_reevaluated_nets`	0
`si_xtalk_exit_on_reevaluated_nets_pct`	0.0
`si_xtalk_exit_on_coupled_reevaluated_nets_pct`	0.0
`timing_min_iteration_count` (affects all iterative analysis)	3

For signal integrity analysis, iterations are eligible to be terminated if no nets are reselected for the next iteration. This condition might occur if no cross-coupled nets meet the delta delay and slack reselection criteria. However, iterations might continue based on the requirements of timing-based multi-input switching analysis and differential skew analysis.

In addition, NanoTime terminates the crosstalk analysis loop after performing an iteration when any of the following conditions is true:

- The number of completed iterations equals the value of the `si_xtalk_exit_on_max_iteration_count` variable. The default is 3.
- The number of reselected nets is less than the value of the `si_xtalk_exit_on_number_of_reevaluated_nets` variable.
- The percentage of reselected nets, compared to the total number of nets, is less than the value of the `si_xtalk_exit_on_reevaluated_nets_pct` variable.
- The percentage of reselected nets, compared to the total number of cross-coupled nets, is less than the value of the `si_xtalk_exit_on_coupled_reevaluated_nets_pct` variable.
- All delta delays fall between the values of the `si_xtalk_exit_on_max_delta_delay` and `si_xtalk_exit_on_min_delta_delay` variables.

Be sure to specify an appropriate set of exit criteria. For example, if you set a large maximum iteration count without changing the other variables from their defaults, the analysis run could be very long, with little or no gain in accuracy.

You can terminate an analysis in progress by pressing Ctrl+C one time. NanoTime completes the current analysis iteration before exiting from the loop. Note that pressing Ctrl+C multiple times causes an exit from NanoTime upon completion of the loop.

Iteration Count

The `si_xtalk_exit_on_max_iteration_count` variable sets the maximum iteration count. To force a specific number of analysis iterations, set this variable without changing any of the other variables. For example,

```
nt_shell> set si_xtalk_exit_on_max_iteration_count 4
```

If you also change some of the other exit criteria variables from their defaults, NanoTime examines the results of each iteration to determine whether more iterations are needed. An analysis of just one iteration is possible, depending on the design and the exit criteria.

Number of Reselected Nets

You can terminate the analysis loop when the number of nets reselected for analysis is small. If the number of reselected nets is zero, the analysis always exits, regardless of the other variable settings.

You can set the following variables to control reselection thresholds:

- The `si_xtalk_exit_on_number_of_reevaluated_nets` variable, the absolute number of reselected nets
- The `si_xtalk_exit_on_reevaluated_nets_pct` variable, the percentage of reselected nets compared to the total number of nets
- The `si_xtalk_exit_on_coupled_reevaluated_nets_pct` variable, the percentage of reselected nets compared to the total number of cross-coupled nets

For example, to have NanoTime exit from the analysis loop when the number of reselected nets is less than 1.0 percent of the total number of nets in the design, use the following command:

```
nt_shell> set si_xtalk_exit_on_reevaluated_nets_pct 1.0
```

By default, all of these variables are set to zero, causing them to be ignored. If you choose to set these variables, you should also increase the maximum iteration count variable from its default of 3.

Delta Delay

When crosstalk analysis causes only a small change in calculated delays, it is an indication that the analysis is fairly accurate. You can terminate the analysis loop when the largest changes in calculated delay among all nets analyzed is within a specified range. You specify the thresholds separately for minimum (hold) delays and maximum (setup) delays.

NanoTime exits the analysis loop after completing the current iteration if the delta delay value for the next iteration falls within the range defined by the `si_xtalk_exit_on_max_delta_delay` and `si_xtalk_exit_on_min_delta_delay` variables.

Both variables default to zero, which means that the range between them is zero and no delays can meet the requirement. You must set at least one of the variables to a nonzero value to use delta delay as an exit criterion. Specify the minimum delay threshold as a negative value and the maximum delay threshold as a positive value, in library time units.

For example, to terminate the analysis loop when the delay change is between –1.0 and +2.0 time units, use the following commands:

```
nt_shell> set si_xtalk_exit_on_min_delta_delay -1.0
nt_shell> set si_xtalk_exit_on_max_delta_delay 2.0
```

For a design that has deltas of {-3, 1.5, 1.7}, the value -3 falls outside the window (-1.0 to 2.0), so NanoTime does not exit the analysis loop. If in the next iteration the failing value improves to 0.9, all deltas fall within the window and NanoTime terminates crosstalk analysis.

See Also

- SI Delay Analysis Procedure
- Selecting or Excluding Nets for SI Delay Analysis
- Reporting SI Delay Analysis

Reporting SI Delay Analysis

You can report SI delay results in the following ways:

- Retrieve and report attributes on specific nets, such as the following examples of net attributes related to SI delay analysis (this is not a complete list):

  ```
  aggressors
  number_of_aggressors
  effective_aggressors
  number_of_effective_aggressors
  coupling_capacitors
  number_of_coupling_capacitors
  total_coupling_capacitance
  effective_aggressors_coupling_cap
  effective_aggressors_max_fall_delta_delays
  effective_aggressors_max_fall_transitions
  si_max_fall_delta_delay
  si_max_fall_transition
  ```

- Use the `report_si_convergence` command to examine the results of a single analysis iteration.
- Use the `report_crosstalk_delay_sources` command to examine details of aggressor nets and their coupling capacitances.
- Use the `report_si_nets` command to list victim nets.
- Use the `-crosstalk_delta` option of the `report_paths` command to add SI information to a standard path report.

The Convergence Report

At the end of each iteration of crosstalk delay analysis, NanoTime displays a convergence report. You can generate this report explicitly with the `report_si_convergence` command, for example, to save the report to a file. However, NanoTime executes the `report_si_convergence` command only at the end of the final iteration. If you want to examine the results of an earlier iteration, you must set the maximum iteration count to a smaller number before the crosstalk analysis begins.

A convergence report includes the number of coupled nets, the number of effective nets (the ones that have not been filtered), and the values that control the iteration exit. The following report is an example of a convergence report:

```
************************************************
Report:  si convergence
Design:  ALU
Version: G-2012.06
Date:    Tue Aug 14 15:58:12 2012
************************************************
Convergence Criteria           Value    Limit    status
--------------------      ---------- -------- ---------
number of nets                  7463
number of coupled nets           256
number of effective nets         256

current iteration                  3        3       met
reevaluated nets                 255        0
reevaluated nets pct            3.42     0.00
coupled reevaluated nets pct   99.61     0.00
delta delay
  max                         0.5701   0.0000
  min                        -0.3745   0.0000
```

In this example, the only iteration exit criterion is the number of iterations. The default of 0.0 appears for the other limits if you do not explicitly set them.

The delta delay values are the changes in calculated minimum and maximum delays from one iteration to the next. The minimum delta delay is always written as a negative value. For example, if the minimum delay is -8.0 after the first iteration and -6.0 after the second iteration, the minimum delta delay is displayed as -2.0 even though the value changed by +2.0 units.

An example of a convergence report with delta delay exit criteria is shown here:

```
************************************************
Report:  si convergence
Design:  ALU
Version: G-2012.06
Date:    Wed Sep 26 16:06:12 2012
************************************************
Convergence Criteria           Value    Limit    status
--------------------      ---------- -------- ---------
number of nets                  7463
number of coupled nets           256
number of effective nets         256

current iteration                  2        4
reevaluated nets                 255        0
reevaluated nets pct            3.42     0.00
coupled reevaluated nets pct   99.61     0.00
delta delay                                          met
  max                         0.5728   0.6000
  min                        -0.3745  -0.3900
```

In this example, the analysis exits after the second iteration because the delta delay criteria have been met: both the maximum and minimum delay values fall between the limits. You set the limits with the `si_xtalk_exit_on_max_delta_delay` and `si_xtalk_exit_on_min_delta_delay` variables.

The Crosstalk Delay Sources Report

You can generate a report of the aggressor nets and their coupling capacitances by using the `report_crosstalk_delay_sources` command.

The following is an example of the default report. Aggressors are listed in order of decreasing maximum delta delay.

```
************************************************
Report:  si crosstalk delay sources
Design:  ALU
Version: G-2012.06
Date:    Wed Sep 12 10:18:57 2012
************************************************
                      rising   falling
coupling   percent aggressor aggressor max delta min delta
     cap  of total     trans     trans     delay     delay Victim net  aggressor net
-------- --------- --------- --------- --------- --------- ----------- ---------------
   0.005      3.35     1.304     0.689     0.573    -0.374 SM9         SM59
   0.005      3.07     0.336     0.370     0.573    -0.374 A[9]        A[5]
   0.005      2.80     0.531     0.283     0.546    -0.374 S[9]        S[3]
...
```

To add a column that displays the percentage of the total crosstalk delay contributed by each aggressor net, use the `-aggressor_contributions` option. To add columns that display the maximum and minimum delta delays as percentages of the total stage delay, use the `-delta_delay_ratio` option. To list the aggressors in order of decreasing maximum delta delay ratio, use the `-cost_type delta_delay_ratio` option.

If the crosstalk analysis includes coupling capacitance variation, the coupling cap column lists the range of capacitance values in the format [minimum capacitance, maximum capacitance]. However, percentage calculations are based on the nominal (typical) capacitance.

The SI Nets Report

You can generate a report of the victim nets that have crosstalk delays by using the `report_si_nets` command.

The following is an example of the default report.

```
************************************************
Report:  si nets
Design:  ALU
Version: G-2012.06
Date:    Mon Aug 20 17:05:14 2012
************************************************

Max Delta            Min Delta
  Delay   Max Ratio    Delay    Min Ratio   netname
--------- --------- ---------- ----------- --------------------------
    0.451     1.096     -0.269       0.702  B[19]
    0.389     1.075     -0.227       0.687  B[24]
    0.421     1.070     -0.259       0.710  B[25]
    0.414     0.964     -0.364       0.339  B[20]
...
```

SI Delays in Path Reports

Path reports generated by the `report_paths` command include signal integrity effects. To see detailed crosstalk information in the report, use the `-crosstalk_delta` option.

```
nt_shell> report_paths -max -max_paths 1 -path_type full -crosstalk_delta
```

The following is an example of the resulting report. The values in the Xtalk Delta column show the incremental change in arrival time due to crosstalk effects. The values in the Incr

column include the calculated delta delay. Note that the crosstalk delta value might be positive or negative.

```
Startpoint:        clk2 (in port)
Endpoint:          Xbreg.Xreg50.X5.Mp0.G
Path Type:         max
Constraint:        latch setup
                             Xtalk
  Path     Incr   Adjust    Delta NT    Point
------ -------- -------- -------- -- -- ------------------------------
 1.500             1.500          C   r clk2 (in)
 1.630    0.130             0.000 C   r Xaddsub.Xadder.Xi0.X0.Mn0.G (inv2)
 1.702    0.072             0.000 C   f Xaddsub.Xadder.Xi0.X1.Mp0.G (inv2)
 1.857    0.154             0.000 C   r Xaddsub.Xadder.Xb00.X0.Mn0.G (inv2)
 1.934    0.077             0.000 C   f Xaddsub.Xadder.Xb00.X1.Mp0.G (inv2)
 2.112    0.178             0.000 C   r Xaddsub.Xadder.Xla0.XP0.Mn0.G (Pterm)
 2.334    0.222             0.000 N1  f Xaddsub.Xadder.Xla0.XP0.X0.X0.Mp0.G
 ...
 6.918    0.290             0.000     r Xaddsub.Xadder.Xfa51.Mn3.G (fadd)
 7.594    0.676             0.000     r Xaddsub.Xadder.Xfa51.X6.X0.Mn0.G (inv)
 7.736    0.142             0.000     f Xaddsub.Xadder.Xfa51.X6.X1.Mp0.G (inv)
 8.466    0.729             0.000     r Xlc51.X5.Mn0.G (inv3)
 8.721    0.255             0.000     f Xls50.X5.Mp0.G (inv2)
 9.385    0.664             0.269     r Xbreg.Xreg50.Mn4.G (muxflop)
 9.682    0.297             0.000 L   f Xbreg.Xreg50.X5.Mp0.G (inv2)
 9.682                                  data arrival time
          9.395    0.287    0.269       Total

 6.803    6.803                         Xbreg.Xreg50.Mn0.G (muxflop)
 6.803    0.000                         setup time
 6.803    0.000                         clock uncertainty
 6.803                                  data required time
-----------------------------------------------------------------------
 6.803                                  data required time
-9.682                                  data arrival time
-----------------------------------------------------------------------
-2.879                                  slack (VIOLATED)
```

Note:

Crosstalk delta values for dynamic clock simulation (DCS) regions in path reports are reported as “n/a” instead of zero. The stage delay includes the dynamically simulated delay for the DCS region. However, the specific portion of the delay due to crosstalk is not separated.

If any stage of a path has a crosstalk delay value of “n/a,” the crosstalk delta value for the complete path is also reported as “n/a.”

See Also

- SI Delay Analysis Procedure
- Selecting or Excluding Nets for SI Delay Analysis
- Controlling SI Delay Iteration Exit Criteria

SI Noise Analysis Procedure

SI noise analysis models the impact of noise on steady-state nets. NanoTime assumes a worst-case combination of aggressor and victim states and computes an injected noise waveform on the victim net. The computed maximum noise voltage level is compared to the user-specified noise margins to determine whether a noise violation occurs.

Follow these steps to perform SI noise analysis:

1. Perform the steps that are common to all SI analysis, as described in Setting Up Signal Integrity Analysis. Make sure to set the enabling variables before executing the `link_design` command:
   - For crosstalk noise analysis, set the `si_enable_noise_analysis` variable to `true`.
   - For fanout noise analysis, set the `si_enable_noise_fanout_analysis` variable to `true`.
2. (Optional) Specify nets that are to be excluded for analysis by using the `-exclude` option with the `set_si_noise_analysis` command.

   You can report the excluded nets with the `report_si_noise_analysis` command and remove the exclusion with the `remove_si_noise_analysis` command.
3. Specify noise margins and choose whether or not to include noise beyond the rail voltages.
4. If you are analyzing fanout noise, specify the fanout noise margins and thresholds.
5. (Optional) Define global or local values for steady state resistances on library pins.
6. (Optional) Inject user-defined noise on a port or pin to replace or add to the previously calculated noise.
7. Execute the `update_noise` command after the `trace_paths` or `extract_model` command in the NanoTime flow.
8. Generate reports to understand noise sources and their impact on circuit elements.

See Also

- Injecting User-Defined Noise
- Calculating Noise on Nets Driven by Model Pins
- Setting Noise Margins
- Reporting SI Noise Analysis

Differences Between the Path Tracing and Model Extraction Flows

SI noise analysis can only be performed after a path tracing operation because path tracing creates the timing windows. NanoTime performs path tracing at both the `trace_paths` and `extract_model` commands. You can execute noise analysis after either one of these commands, but the implications are different for the two flows.

The primary effect on noise analysis is the accuracy of the timing windows. If the calculated timing windows are too small, the noise analysis tends to be overly optimistic. If the timing windows are too large, the noise analysis tends to be overly pessimistic.

The following conditions apply to the path tracing flow (executing the `update_noise` command after the `trace_paths` command):

- The timing windows are accurate in this case. However, small changes in input or output conditions or small changes to the design might cause noise violations. To avoid this, add pessimism to the boundary context by using the `set_input_delay` command and the `si_timing_window_overlap_tolerance` variable.
- Limits on the path transparency depth can affect the accuracy of timing windows on downstream nets. Check the `trace_transparency_depth_limit` variable with respect to your design.
- Timing windows within transparent loops might be incorrect if path tracing through those loops is not enabled. By default, NanoTime does not search through transparent loops. If your design contains transparent loops, set the `trace_transparent_loop_checking` and `trace_transparent_inverting_loops` variables appropriately.
- If latch error recovery is disabled, timing windows downstream from the latch net might be incorrectly limited. Check the `trace_latch_error_recovery` variable setting with respect to your design.
- If you use the `set_find_path` command to limit path tracing, paths that are not searched have infinite timing windows, which might lead to overly pessimistic results.
- If path tracing through precharge paths is disabled, timing windows on nets downstream from a precharge net might be incorrectly limited. Check the settings of all of the `timing_precharge_*` variables with respect to your design. Examples include the `timing_precharge_contention_recovery` and `timing_precharge_min_latch_transparency` variables.

The following conditions apply to the model extraction flow (executing the `update_noise` command after the `extract_model` command):

- NanoTime creates context-independent models by default. Nets between the primary inputs and the boundary input latches have infinite timing windows. Any net in a path downstream from a boundary input latch has a maximum allowable timing window based

on transparency from any input arrival. The resulting noise analysis might be overly pessimistic because of the large timing windows.

Alternatively, you can generate context-dependent timing models by using the `-context_dependent` option. In this case, you must ensure that the input context has enough margin to cover most of the environments in which the timing model could be used. Use the `set_input_delay` command to set the conditions.

- If you create transparent models (using the `-non_transparent` option) or models with ignored transparency (using the `-ignore_input_to_output_transparency` option), the timing windows from the output boundary latches to the primary outputs might shrink. Limiting the transparency depth (using the `-latch_level` option) can also reduce timing windows downstream from the affected nets. The resulting noise analysis might be overly optimistic because of the small timing windows.
- Ignoring paths from ports (using the `-ignore_ports` option) might remove or limit timing windows on downstream nets.
- Including only specific paths in the model (using the `-find_path_only` option) produces infinite timing windows on nets that are not searched.

Injecting User-Defined Noise

You can inject a noise bump at any port or pin in the design. The injected noise can replace the noise bump that NanoTime calculates, or it can be added to the calculated noise bump.

A simple triangle model represents the user-defined injected noise bump, as shown in Figure 17-16.

Three values define the noise model:

- Height in voltage
- Total width in time units
- Time-to-peak ratio, which is the time-to-peak divided by the total bump width. A value of 0.5 indicates a symmetric shape.

Figure 17-16 Triangle Model Noise Bump

Choose the width to make the triangle model area equal the actual noise bump area. Choose the time-to-peak ratio to make the area of the noise bump that occurs before the peak voltage equal that of the actual noise.

Use the `set_input_noise` command with a list of affected ports or pins to define injected noise. Specify the three geometry parameters of the triangular noise model by using the `-height`, `-width` , and `-time_to_peak_ratio` options. In addition, you can limit the type of noise with the appropriate combination of `-above`, `-below`, `-high`, and `-low` options.

If the `-add_noise` option is not present, user-defined injected noise overrides previously calculated noise. If the `-add_noise` option is present, NanoTime aligns the peak of the user-defined triangular injected noise with the peak of any injected noise that was previously calculated.

Use the `remove_input_noise` command to remove user-defined injected noise.

See Also

- SI Noise Analysis Procedure
- Calculating Noise on Nets Driven by Model Pins
- Setting Noise Margins
- Reporting SI Noise Analysis

Calculating Noise on Nets Driven by Model Pins

When the design contains black box models, NanoTime can calculate noise on victim nets that are driven by model pins. You can define the steady-state resistance values (linear holding resistances) at the library pins. If you do not specify any resistances, the tool uses the default of 0.1 ohms, which results in the smallest possible injection noise.

Set steady-state resistances globally or locally, as follows:

- Global settings

  Set the following variables, which all default to 0.1 ohms:

  - The `lib_pin_steady_state_resistance_above_low` variable for calculating noise above the ground rail
  - The `lib_pin_steady_state_resistance_above_high` variable for calculating noise above the supply rail
  - The `lib_pin_steady_state_resistance_below_low` variable for calculating noise below the ground rail
  - The `lib_pin_steady_state_resistance_below_high` variable for calculating noise below the supply rail

- Pin-specific settings

  Use the `set_steady_state_resistance` command to define resistances on specific library pins and override the global values. To remove a specific setting, use the `remove_steady_state_resistance` command.

The `steady_state_resistances` attribute for a library pin (associated with the `lib_pin` object class) contains all four resistances.

Note:
NanoTime does not perform noise analysis on an input or inout port if there is no drive resistance. The tool issues a warning message if this condition is detected.

See Also

- SI Noise Analysis Procedure
- Injecting User-Defined Noise
- Setting Noise Margins
- Reporting SI Noise Analysis

Setting Noise Margins

Noise margins are the test values that determine if calculated noise is in violation. By default, NanoTime noise margins are zero, which means that all detected noise is in violation. Set noise margins globally or locally, as follows:

- Global settings

  Set the following variables, which all default to zero:

  - The `si_noise_margin_above_low` variable for the margin above the ground rail
  - The `si_noise_margin_above_high` variable for the margin above the supply rail
  - The `si_noise_margin_below_low` variable for the margin below the ground rail
  - The `si_noise_margin_below_high` variable for the margin below the supply rail

- Object-specific settings

  Use the `set_noise_margin` command to define noise margins on specific objects (pins, ports and nets) and override the global noise margins.

By default, NanoTime checks and reports only the below high and above low types of noise. You can enable the checking and reporting of the other noise violations with the `set_noise_parameters -include_beyond_rails` command.

See Also

- SI Noise Analysis Procedure
- Injecting User-Defined Noise
- Calculating Noise on Nets Driven by Model Pins
- Reporting SI Noise Analysis

Reporting SI Noise Analysis

The following sections discuss signal integrity noise analysis reports:

- The report_noise Command
- The report_noise_violation_sources Command
- The report_noise_coverage Command

The report_noise Command

The `report_noise` command generates a report with the single worst violator for each of the four types of noise (above high, below low, and so on). The columns display the noise slack for each pin, the noise height (voltage), the type of noise (for example, above high), the name of the pin, and the name of the connected net. Noise slack is the amount by which the calculated noise exceeds the margin; negative slack indicates a violation.

The following is an example of a default noise report:

```
************************************************
Report:  si noise values
Design:  ALU
Version: G-2012.06
Date:    Thu Sep 27 11:21:55 2012
************************************************
Noise Types:
    AH - Above High, noise above the supply rail
    AL - Above Low, noise above the ground rail
    BH - Below High, noise below the supply rail
    BL - Below Low, noise below the ground rail

    slack    height type Pin                           Net
--------- --------- ---- --------------------------     ---------------
    0.000     0.200 AH   Xlc38.X5.Mn0.G                 Xlc38.S
   -0.541    -0.753 BH   Xareg.Xreg9.Mp4.G              S[9]
   -0.468     0.668 AL   Xaddsub.Xadder.XLa4.XG3.Mn1.G  B[19]
```

In this example, the worst slack for above high noise is 0.0, which means that there are no noise violations for this type of noise. The report does not list any pins for below low noise, which means that no pins exhibit this type of noise.

You can include all of the pins with noise margin violations (in other words, having negative slack) by using the `report_noise` command with the `-all_violators` option. To report specific noise types, use the `-above`, `-below`, `-low`, and `-high` options in combination. The `-nworst` *N* option displays the worst N values of each type of noise.

If you want to report all noise values and not just those with violations, use the `-slack_lesser_than` and `-height_greater_than` options to select pins based on the noise values.

An example of an expanded noise report is as follows:

```
nt_shell> report_noise
************************************************
Report:  si noise values
Design:  ALU
Version: G-2012.06
Date:    Thu Sep 27 13:26:55 2012
************************************************
Noise Types:
    AH - Above High, noise above the supply rail
    AL - Above Low, noise above the ground rail
    BH - Below High, noise below the supply rail
    BL - Below Low, noise below the ground rail

    slack    height type Pin                            Net
--------- --------- ---- --------------------------     ----------------
    0.000     0.200 AH   Xlc38.X5.Mn0.G                 Xlc38.S
    0.001     0.199 AH   Xlc38.X5.Mp0.G                 Xlc38.S
    0.005     0.195 AH   Xlc16.X5.Mn0.G                 Xlc16.S
    ...
   -0.953    -0.753 BH   Xareg.Xreg9.Mp4.G              S[9]
   -0.916    -0.716 BH   Xareg.Xreg22.Mp4.G             S[22]
   -0.893    -0.693 BH   Xareg.Xreg0.Mp4.G              S[0]
    ...
   -0.468     0.668 AL   Xaddsub.Xadder.XLa4.XG3.Mn1.G  B[19]
   -0.441     0.641 AL   Xaddsub.Xadder.Xla5.Xg0.Mn1.G  B[20]
   -0.424     0.624 AL   Xaddsub.Xadder.Xla6.XG1.Mn1.G  B[25]
    ...
```

The `-shape` option adds two columns to the report: the noise bump width and time-to-peak ratio. The values reported with the `-shape` option depend on how you used the `set_input_noise` command.

- Without the `-add_noise` option, the values correspond to the user-defined injected noise.
- With the `-add_noise` option, the values correspond to a triangular model that is fitted to a composite noise waveform consisting of user-defined noise and any other injection noise that NanoTime might have calculated.

The report_noise_violation_sources Command

To view noise violation sources, use the `report_noise_violation_sources` command. The report displays the coupling capacitance to the victim net, the percentage of the total coupling capacitance to the victim, and the minimum rising and falling aggressor transitions. The default report displays the single worst violator for each of the four types of noise (above high, below low, and so on), but you can display the N worst violators of each type by using the `-nworst N` option.

If you specify user-defined noise with the `set_input_noise` command, the reported noise source is only the user-defined noise. If you use the `set_input_noise` command with the `-add_noise` option, the user-defined bump is reported along with any aggressors that are also sources of the calculated injected noise.

An example report with the `-nworst 3` option is shown here:

```
nt_shell> report_noise_violation_sources
************************************************
Report:  si noise violation sources
Design:  ALU
Version: G-2012.06
Date:    Thu Sep 27 14:11:41 2012
************************************************
Aggressor Transition:
    R   - Rising
    F   - Falling
    D   - Use default transition
    INF - Infinite delay
type:
    AH - Above High, noise above the supply rail
    AL - Above Low, noise above the ground rail
    BH - Below High, noise below the supply rail
    BL - Below Low, noise below the ground rail

coupling  percent  aggressor    noise
     cap of total      trans    slack type Victim pin                      aggressor net
-------- -------- ---------- -------- ---- ------------------------------ -------------
   0.005     2.83  0.382 (R)   -0.448   AH Xadsub.Xadder.Xla4.XG3.Mn1.G  B[44]
   0.005     2.99  0.338 (R)   -0.430   AH Xadsub.Xadder.Xla5.XG0.Mn1.G  B[23]
   0.005     3.45  0.358 (R)   -0.425   AH Xlc12.Mn13.G                  A[30]
   ...
   0.005     2.80  0.283 (F)   -0.953   BH Xareg.Xreg9.Mp4.G             S[3]
   0.005     3.06  0.271 (F)   -0.916   BH Xareg.Xreg22.Mp4.G            S[38]
   0.005     3.17  0.273 (F)   -0.893   BH Xareg.Xreg0.Mp4.G             S[27]
   ...
   0.005     2.83  0.382 (R)   -0.468   AL Xadsub.Xadder.Xla4.Xg3.Mn1.G  B[44]
   0.005     2.99  0.338 (R)   -0.441   AL Xadsub.Xadder.Xla5.Xg0.Mn1.G  B[23]
   0.005     2.91  0.331 (R)   -0.424   AL Xadsub.Xadder.Xla6.Xg1.Mn1.G  B[40]
   ...
   0.005     2.80  0.283 (F)   -0.955   BL Xareg.Xreg9.Mp4.G             S[3]
   0.005     3.06  0.271 (F)   -0.926   BL Xareg.Xreg22.Mp4.G            S[38]
   0.005     3.17  0.273 (F)   -0.893   BL Xareg.Xreg0.Mp4.G             X[27]
   ...
```

You can include an optional column of data containing estimates of the percentage of the total noise height contributed by each individual aggressor net. To report the aggressor net contributions, use the `-aggressor_contributions` option.

You can also include an optional column that displays the timing windows for each of the individual noise aggressors by using the `-aggressor_windows` option.

An aggressor timing window description appears in one of the formats shown in Table 17-7, where *min* represents the smallest minimum delay arrival time, *max* represents the largest

maximum delay arrival time, *clk* is the name of the associated clock, and *dir* is R or F for the transition direction of the clock reference edge.

Table 17-7 Aggressor Timing Window Formats

Format	Description
INF	No timing window available; NanoTime uses an infinite timing window
(*min max*)	A timing window for a nonperiodic nonclock domain; both minimum and maximum delay arrival times exist for this window
(INF INF)	A timing window for a nonperiodic nonclock domain, but no arrival times exist for this window
(*min* INF)	A timing window for a nonperiodic nonclock domain; no maximum delay arrival times exist for this window
(INF *max*)	A timing window for a nonperiodic nonclock domain; no minimum delay arrival times exist for this window
(*min max clk dir*)	A timing window for a periodic clock domain; both minimum and maximum delay arrival times exist for this window
(INF INF c*lk dir*)	A timing window for a periodic clock domain, but no arrival times exist for this window
(*min* INF *clk dir*)	A timing window for a periodic clock domain; no maximum delay arrival times exist for this window
(INF *max clk dir*)	A timing window for a periodic clock domain; no minimum delay arrival times exist for this window

An example report with the `-aggressor_windows` option is shown here. A single aggressor might have more than one timing window, in which case the report shows all of them in the last column.

```
nt_shell> report_noise_violation_sources
   ...
coup  pct of  aggressor  noise                          aggressor aggressor
 cap   total      trans  slack type Victim pin          net       window
----- ------ ---------- ------ ---- ------------------- --------- ------------------
0.005   2.80  0.283 (F) -0.949   BH Xareg.Xreg9.Mp4.G   S[3]      (2.475 5.740 clk2 R)
0.005   3.06  0.271 (F) -0.916   BH Xareg.Xreg22.Mp4.G  S[6]      (INF 3.992 clk1 R)
0.005   3.17  0.273 (F) -0.893   BH Xareg.Xreg0.Mp4.G   S[0]      (1.070 3.180 MCLK R)
0.005   2.83  0.382 (R) -0.458   AL Xaddsub.Xadd.Mn1.G  B[44]     (INF INF clk1 R)
...
```

The report_noise_coverage Command

The `report_noise_coverage` command generates a report about injection noise analysis. This report includes the top nets of each hierarchical group as victim nets and provides additional information such as the total coupling capacitance, the total effective aggressor capacitance, and the number of effective aggressors.

The report format as follows:

```
        Sum of
Sum Of   Eff.
Coupling Aggr.    # of Eff.   BH        AL
Cap      Cap      Aggr.     Height    Height   Attributes                Victim
-------- -------- --------- --------- -------- ------------------------- -----------
0.010200 0.010200     6     -0.193381 0.191487 BHI ALI G2C TRI           bD0
0.010200 0.010200     6     -0.193381 0.191487 BHI ALI G2C TRI           bD1
0.006300 0.006300     5     -0.002235 0.004902 GHI ALI PFB               bout1
0.004000 0.004000     2         --       --    BHI ALI NGP               bbgclk1
...
```

The Attributes column includes codes that provide additional information. In this example, noise heights for net bbgclk1 are missing because the victim net doesn't have any gate pins, as indicated by code NGP in the Attributes column.

You can customize some of the report columns by using command options. For example, if you use the `-attribute_count` option, a number in parentheses appears after a code to indicate the number of occurrences of that item.

Table 17-8 lists the codes that might appear in the Attributes column. These codes appear in the legend of the report.

Table 17-8 Attributes in Noise Coverage Report

Code	Attribute	Code	Attribute
NC	No coupling capacitances	GFB	Net with gate-connected feedback device
NEA	No effective aggressors	CFB	Net with channel-connected feedback device
BNA	Beyond rail analysis not enabled	ILO	Input logic state turns off transistor
LS0	Logic state of 0 on victim	AHE	Above high excluded
LS1	Logic state of 1 on victim	ALE	Above low excluded
LA0	Logic state of 0 on aggressor	BHE	Below high excluded
LA1	Logic state of 1 on aggressor	BLE	Below low excluded
DSG	dont_search_through_gate	AHI	Above high inverted
DSC	dont_search_through_channel	ALI	Above low inverted
PI	Primary input	BHI	Below high inverted
LFB	Latch feedback net	BLI	Below low inverted
LAT	Latch net	G2C	Aggressor gate to channel victim coupling

Table 17-8 Attributes in Noise Coverage Report (Continued)

Code	Attribute	Code	Attribute
LEV	Level shifter cross-coupled net	C2G	Aggressor channel to gate victim coupling
TRI	Turnoff net	PUC	Pullup to VDD with aggressor from gate to channel net
REG	Register file bitcell net	PDC	Pulldown to ground with aggressor from gate to channel net
PRE	Precharge net	PUG	Pullup to VDD with aggressor from channel to gate net
PFB	Precharge feedback net	PDG	Pulldown to ground with aggressor from channel to gate net
RAM	RAM cell net	UNK	All other exceptions
NTP	No timing point		
NGP	No gate pin on net		

See Also

- SI Noise Analysis Procedure
- Injecting User-Defined Noise
- Calculating Noise on Nets Driven by Model Pins
- Setting Noise Margins

Fanout Noise Analysis Procedure

To assess the vulnerability of a design to propagated noise, you can analyze injected noise on a net through one fanout level of that net. The injected noise can be the noise calculated during SI noise analysis, a user-defined noise pulse, or a combination of the two.

The following limitations apply to fanout noise analysis:

- NanoTime analyzes fanout noise to a maximum depth of one level.
- Very large fanout channel-connected blocks might have to be limited in depth of exploration to save runtime.
- Fanout channel-connected blocks must be transistor level circuits. Fanout noise analysis does not include timing models on the fanout of victim nets.
- Fanout noise analysis does not take logic correlation into account.
- Noise analysis is not available for dynamic simulation stages.

A user-defined injected noise bump, if present, contributes to the trigger for calculating fanout noise and can therefore influence the amount of fanout noise. If you do not use the `-add_noise` option of the `set_input_noise` command, the user-defined injected noise bump is considered the trigger for fanout noise calculation. If you use the `-add_noise` option, the aligned sum of both the user-defined and calculated injected noise bumps is considered as the trigger for fanout noise calculation.

NanoTime analyzes the effect of noise on a victim net for each fanout stage that it drives. Noise is measured on the output of each fanout channel-connected block.

To perform fanout noise analysis, follow these steps:

1. Run standard crosstalk noise analysis, as described in SI Noise Analysis Procedure.
2. Set the `si_enable_noise_fanout_analysis` variable to `true`.
3. (Optional) Set thresholds to determine whether noise on the victim net should be considered for fanout noise analysis.
4. Set global noise margins to test fanout noise levels.
5. (Optional) Set noise margins for fanout noise tests on specific pins or nets.
6. Execute the `update_noise` command.
7. Report the noise analysis by running the `report_fanout_noise` command or by examining noise-related attributes on the fanout nets.
8. (Optional) Write a SPICE deck that includes injected and fanout noise.

Setting Fanout Noise Thresholds

To reduce simulation effort, NanoTime can filter out cases where fanout noise analysis is not important, or where the noise on a victim net is small enough to remove from consideration.

You can apply global or local thresholds to noise on a victim net, as follows:

- Set global thresholds with the `si_fanout_noise_threshold_above_low` and `si_fanout_noise_threshold_below_high` variables to define whether noise peaks above the ground rail or below the supply rail should be propagated.
- Use the `set_fanout_noise_threshold` command to define local thresholds for specific pins or nets.

In all cases, the threshold is a positive voltage value that is either added to the ground voltage or subtracted from the supply voltage. The threshold defaults are zero, which means that every noise peak is propagated forward for fanout analysis.

The following commands allow NanoTime to propagate only the noise bumps that are at least 0.05 V above the ground rail or at least 0.07 V below the supply rail:

```
nt_shell> set si_fanout_noise_threshold_above_low 0.05
nt_shell> set si_fanout_noise_threshold_below_high 0.07
```

The following command sets a fanout noise threshold of 0.2 V for peaks above the ground rail on pin Xxor2.NM1.g:

```
nt_shell> set_fanout_noise_threshold -above -low 0.2 Xxor2.MN1.g
```

Fanout noise does not consider noise outside of the rails. Only the `-above -low` and `-below -high` combinations are valid for the `set_fanout_noise_threshold` command.

Setting Fanout Noise Margins

Noise margins are the test values that determine if the fanout noise is in violation. There are two global fanout noise margin variables, as follows:

- The `si_fanout_noise_margin_above_low` variable specifies the allowable voltage above the ground rail.
- The `si_fanout_noise_margin_below_high` variable specifies the allowable voltage below the supply rail.

In both cases, the value is a positive voltage value that is a delta voltage which is either added to the ground voltage or subtracted from the supply voltage. Values of zero indicate that no noise is allowed and all detected noise is in violation.

Reporting Fanout Noise Analysis

The `report_fanout_noise` command reports the effect of noise waveforms that propagate from pins that have noise, through the driven channel-connected region, then to the fanout net. The original noise victim pin and the noise peaks are listed in the report. The fanout net and propagated noise values are also listed by type.

An example of a fanout noise report is as follows:

```
report_fanout_noise -significant_digits 5 -nworst 2
****************************************
Report : si fanout noise values
Design : ALU
...
****************************************
Noise Types:
    AL - Above Low, noise above the ground rail
    BH - Below High, noise below the supply rail

Fanout    Fanout  Fanout                      Trigger     Trigger
slack     height  Type   Fanout Net           Height AL   Height BH  Trigger Pin
-------- -------- ------ ------------------- ---------- -----------  -----------------
 0.10000  0.00000 BH     Xareg.Xreg9.lat1n   0.71663    -0.71459    Xareg.Xreg9.Mp4.G
 0.04984  0.05016 AL     Xareg.Xreg9.lat1n   0.71663    -0.71459    Xareg.Xreg9.Mp4.G
 0.10000  0.00000 BH     Xareg.Xreg22.lat1n  0.69491    -0.68661    Xareg.Xreg22.Mp4.G
 0.05235  0.04765 AL     Xareg.Xreg22.lat1n  0.69491    -0.68661    Xareg.Xreg22.Mp4.G
```

Options of the `report_fanout_noise` command allow you to modify the report:

- Use the `-slack_lesser_than` option to report a set of fanout nets whose noise slack value is less than the specified value.
- Use the `-height_greater_than` option to report a set of fanout nets whose noise height value is greater than the specified value.
- You can report fanout noise from specific objects.

If the `-shape` option of the `report_fanout_noise` command is specified, two additional columns (width and time-to-peak ratio) are included for reporting the properties of each fanout noise bump. An example report is as follows:

```
Noise Types:
  AL - Above Low, noise above the ground rail
  BH - Below Height, noise below the supply rail

fanout     fanout       time_to_peak Fanout  Fanout     Trigger    Trigger    Trigger
 slack     height    width     ratio  Type      Net   Height AL  Height BH        Pin
-------   -------   ------    ------  ----  --------   --------  --------- -----------
-0.0764   -0.0764   0.3914    0.4190   BH   Xfa19.Bn   0.51841   -0.5011   Xfa19.Mp0.G
 0.0000    0.0000   0.0000    0.0000   BH   Xfa19.Z    0.28036   -0.2528   Xfa19.Mn0.G
```

The noise slacks displayed by the `report_fanout_noise` are based on the peak heights of the fanout noise bumps.

18

Memory Circuit Analysis

The NanoTime tool performs timing analysis for embedded memory designs and reports critical paths without the need to specify input vectors or measurement statements.

This chapter contains the following sections:

- Memory Analysis Overview
- Using NanoTime to Analyze Memories
- Memory Topology Recognition
- Clock Propagation for Memories
- Case Analysis for Memories
- Timing Checks for Memories
- Path Tracing and Timing Analysis
- Memory Analysis Reporting
- Black Box Memory Analysis
- Memory-Specific Syntax

Memory Analysis Overview

Memory blocks present several challenges for static timing analysis. The NanoTime tool provides specialized features that extend its capabilities to memories.

NanoTime cannot analyze the memory core array of the following types of memory designs:

- Nonvolatile memories, including flash memory
- Dynamic memories such as DRAMs
- Content-addressable memory

However, you can analyze the peripheral logic of these memory designs.

Differences From Standard NanoTime Analysis

Analyzing memory circuits differs from analyzing digital logic in the following ways:

- Topology recognition

  Memory topologies include a large number of transistors connected to networks that can switch to intermediate voltage values. They also contain analog elements such as sense amplifiers that cannot be supported by specialized digital logic simulators.

- Simulation complexity

  NanoTime uses customized methods to determine which elements of the memory array to include in the simulation netlist and which side inputs are necessary.

- Accuracy and performance

  NanoTime uses specialized techniques to reduce both the netlist size and the number of simulations needed to measure critical timing paths.

- Timing constraint complexity

  Memory blocks use complex clocking schemes and require many timing constraints to validate correct behavior.

- Model extraction

  Functional modes such as read and write must be modeled correctly to represent the different modes of operation. The model extraction operation in NanoTime can capture user-specified functional mode settings to generate models with labeled timing arcs.

License Requirements

The license requirements are as follows:

- A specific license to enable memory analysis
- A minimum of three NanoTime or NanoTime Ultra licenses
  - If NanoTime Ultra features are not used, all licenses can be NanoTime licenses.
  - If NanoTime Ultra features are used, at least one NanoTime license and one NanoTime Ultra license are required; the remaining licenses can be of either type.
- At least one HSPICE license

Memory Analysis Modes

Figure 18-1 shows a simple 64-bit memory circuit. The NanoTime tool uses standard analysis techniques for the peripheral circuits, but employs the HSPICE dynamic simulation tool to analyze the bitcells with high accuracy. NanoTime can also analyze split memory arrays, such as the example in Figure 18-2.

Figure 18-1 HSPICE Analysis Region For Memory Array

Figure 18-2 Split Memory Array

You can optionally use a timing model to represent the memory array instead of simulating the array directly. This method (known as black box analysis) requires less runtime and memory consumption because delay analysis is performed on a much smaller number of paths. In most cases, delays through the memory array do not affect the final extracted timing model behavior. Figure 18-3 shows the reduced region of the circuit that receives HSPICE simulation.

Figure 18-3 HSPICE Analysis Region for Memory Black Box Analysis

Using NanoTime to Analyze Memories

Figure 18-4 shows the general design flow for a memory block. All steps represent NanoTime operations except for the first and last steps. The goal is to create a single extracted timing model that accurately represents all the operational modes of the memory block. The model can then be used in hierarchical timing analysis in the Synopsys NanoTime and PrimeTime tools, or in other third-party tools.

To achieve this goal, you must first analyze, debug, and extract a model for each operational mode (such as read and write modes) separately. Then you can use the individual models or merge them into a single model that contains conditional statements.

Figure 18-4 Memory Analysis Flow

Figure 18-5 shows the steps in the NanoTime analysis for the memory block. The phases are the same as for NanoTime analysis of other types of circuits. However, for memory circuits, the topology recognition and clock propagation steps are especially important. You must spend time in this phase to review topology and clock reports to ensure that the design is correct. The next few sections provide more detail about procedures in the topology recognition and clock propagation phases.

Your analysis plans might include features such as annotated parasitics, signal integrity analysis, or path-based slack adjustment. To simplify the troubleshooting process, verify that path tracing is successful on your design before you enable optional features.

Figure 18-5 NanoTime Analysis Flow

Memory Topology Recognition

NanoTime automatically recognizes many memory cell and peripheral circuit topologies. You must manually mark other circuits. The goal for your final analysis script is to perform all the topology operations at the same time. However, achieving this goal usually requires several iterations of interactive analysis to determine how to mark structures correctly. In this investigation phase, you can experiment with automatic and manual structure marking, observe the clock and data paths, and fix topology errors.

The black box analysis method, in which a timing model represents the memory array, follows a simplified procedure. For more information, see Black Box Memory Analysis.

Figure 18-6 shows a single memory column and its constituent circuit blocks.

For memory circuits, you must observe the following topology marking requirements:

- The following signals must be clock signals: wordlines, read and write mux select, memory precharge (bitline and sense amplifier) enable, sense amplifier enable, and write driver enable.
- For read mode analysis, the topology marking must yield a set of contiguously connected nets from the bitcell to the sense amplifier.
- For write mode analysis, the topology marking must define a chain of contiguously connected nets from the write data driver to the bitlines at the cell.
- For write mode analysis, at least one memory write control signal must exist, either a write mux enable or a write driver circuit enable. NanoTime issues an NTM-019 error message if the tool cannot detect a write control signal.

Figure 18-6 Memory Circuit Blocks

See Also

- Memory Object Creation
- Topology Recognition Procedure
- Bitline Tracking Circuits
- Memory Cell Topologies
- Peripheral Circuit Topologies

Memory Object Creation

Before carrying out topology recognition on the design, you must create a memory object and specify an analysis mode by using the `create_memory` command.

To analyze all or part of the bitcell array, you must set the `-mode` option to either `read` or `write`, as shown in the following example:

```
create_memory -name mem4 -mode read -bitcell_ports {nmos_bidi} top_cell
```

To skip analyzing the bitcells, use the black box analysis method by setting the `-mode` option to `skip_array`. For more information, see Black Box Memory Analysis.

In the `create_memory` command, the `-name` option specifies a name for the new memory block.

The `-bitcell_ports` option specifies a list of ports used by the bitcells in the memory array. The list contains one item for a single-port design or two items for a dual-port design. The valid port types are as follows:

- `nmos_read` (read)
- `nmos_write` (write)
- `nmos_bidi` (read and write)
- `two_nmos_read` (read)
- `single_ended_nmos_read` (read)
- `single_ended_nmos_read_differential_write` (read and write)
- `single_ended_two_nmos_read` (read)

Every memory object must include at least one read port type and one write port type. Therefore, a single-port design must use a port type that can both read and write.

The final argument is the top-level hierarchical cell in which to search for the memory core array. If you are defining more than one memory in the design, you must include the top-level cell in the `create_memory` command.

See Also

- Topology Recognition Procedure
- Bitline Tracking Circuits
- Memory Cell Topologies
- Peripheral Circuit Topologies

Topology Recognition Procedure

The general procedure for working with a memory circuit in the topology recognition stage is as follows:

1. Identify all of the bit cores in the memory block with the `create_memory` command. For example, the following command creates a memory object named *r_mem* found in the design hierarchy under top_cell, having up to two port types specified in *p_list* and using the read mode for analysis. For more information, see Memory Object Creation.

   ```
   create_memory -name r_mem -mode read -bitcell_ports {p_list} top_cell
   ```

2. Check the `topo_auto_search_class` variable to verify that it contains the structure types of interest for automatic topology recognition.
3. Mark structures needed for correct propagation of the clock network, such as clock gates. Note that NanoTime can automatically recognize many topologies.
4. (Optional) Set up dynamic clock simulation regions for complex circuits such as self-timed circuits. For more information, see Dynamic Clock Simulation for Memory Analysis.
5. (Optional) To enable path tracing through sense amplifier enable signals in pipelined memories, set the `timing_enable_path_through_sense_amplifier` variable to `true`. For more information, see Clocking for Pipelined Memories.
6. Execute the `match_topology` command to recognize topologies and propagate the clock network.
7. Examine reports generated by the `report_clock_network`, `report_topology`, and `report_clock_arrivals` commands to determine if the topologies and clock signals are correct.
8. Mark memory-specific topology structures that do not fit the automatic recognition pattern by using commands such as the `mark_sense_amp` command.

9. Mark nonstandard latches, precharge structures, and clock gates manually by using the `mark_latch`, `mark_precharge`, and `mark_clock_gate` commands.
10. Use the `match_topology` and reporting commands again to observe the changes.
11. Set transistor directions as needed for directional transistors. For nondirectional transistors, set the variable `topo_waive_nondirectional_transistor` to `true` to disable this requirement.
12. Repeat the marking, reporting, and `match_topology` steps to make changes and observe the effects.
13. Execute the `check_topology` command to complete the topology recognition phase.
14. Verify the topology by using the `report_memory, report_bitline, report_wordline, report_topology,` and `get_topology` commands.
15. Incorporate the successful steps into a script, then test the script in a new session.
16. Repeat the interactive analysis and script testing steps as needed until all topology errors are resolved.

See Also

- Memory Object Creation
- Bitline Tracking Circuits
- Memory Cell Topologies
- Peripheral Circuit Topologies

Bitline Tracking Circuits

Some memory designs include tracking or replica circuits to improve performance in the presence of process and operating condition variations. A typical tracking circuit includes a bit column with a small number of bitcells. The design matches the main memory array, but the replica circuit is not part of the array. Instead, the delay of the tracking bitlines is used to time the sense amplifier enable signal to optimize sense amplifier performance when reading data from the memory array.

Tracking bitcells should not be included in NanoTime analysis. However, the topology recognition step might automatically recognize the tracking bitcells. In this case, you must manually erase them.

If your design contains tracking bitcells, modify the general topology recognition procedure as follows:

1. Force NanoTime to recognize only memory cells with the following command:

   ```
   match_topology -structure_types {inverter ram}
   ```

2. Erase the tracking bitcells with the following command, where cell_xxx identifies the tracking bitcells:

   ```
   erase_ram cell_xxx
   ```

3. Execute the `create_memory` command and follow the procedure described in Memory Topology Recognition.

Figure 18-7 shows a simplified view of an SRAM clock generator. The external clock input CLK feeds a clock generator circuit, which in turn derives necessary clock signals such as WL for the wordlines. The sense amplifier enable (SAE) clock signal is delayed by the tracking bitline (TBL) circuit. There might also be a return loop after the TBL circuit to use the TBL delay time for other purposes, such as turning off the wordline clock.

Figure 18-7 Example SRAM Clock Generator

Figure 18-8 is an example of a bitline tracking circuit. Repeated cells similar to bitcells simulate the load on the bitlines. These cells might omit some of the transistors of an actual bitcell and might have floating inputs because they primarily function as load devices. The repeated cells optimize the timing of the SAE signal by causing a delay that is directly related to process variations in the memory array.

The TBL circuit input drives a PMOS pullup transistor and the pass transistor on one or more of the tracking bitcells. These special cells are configured to pull down the tracking bitline when the input signal is high.

Figure 18-8 Example Bitline Tracking Circuit

See Also

- Memory Object Creation
- Topology Recognition Procedure
- Memory Cell Topologies
- Peripheral Circuit Topologies

Memory Cell Topologies

This section contains the following topics:

- Six-Transistor Single-Port Cell With Differential Read and Write
- Seven-Transistor Dual-Port Cell
- Eight-Transistor Dual-Port Cell With Differential Read and Write
- Ten-Transistor Dual-Port Cell
- Eight-Transistor Dual-Port Cell With Differential Write and Single-Ended Two-NMOS Read

See Also

- Memory Object Creation
- Memory Topology Recognition
- Bitline Tracking Circuits
- Peripheral Circuit Topologies

Six-Transistor Single-Port Cell With Differential Read and Write

Figure 18-9 shows a standard six-transistor (6T) SRAM cell, which consists of two cross-coupled inverters. Transistors N1, N3, P1, and P2 constitute the basic storage cell (bitcell); they form a latch with two stable states that represent logic values 0 and 1. Transistors N2 and N4 are access transistors that connect the bitcell to the bitlines, which serve both to provide input to the cell and to read output from the cell. The wordline controls the access transistors. Figure 18-10 is an alternate representation of the same design, showing the inverters instead of the individual transistors.

Figure 18-9 6T Single-Port SRAM Cell

Figure 18-10 Gate Representation of 6T Cell

This is a single-port design, which means that each bitcell is only accessible by one pair of ports for read or write access.

To enable NanoTime to recognize this design, use the `create_memory` command as follows (this example is for a read operation):

```
nt_shell > create_memory -name mem_name -mode read \
                         -bitcell_ports {nmos_bidi}
```

A variation on this design is to use only one bitline for the read operation. The read operation for a cell of this type does not use a sense amplifier.

This configuration is a one-read, one-write (1R1W) design. Simultaneous reading and writing of the cell is not supported.

To enable NanoTime to recognize this design, use the `create_memory` command as follows (this example is for a read operation):

```
nt_shell > create_memory -name mem_name -mode read \
           -bitcell_ports {single_ended_nmos_read_differential_write}
```

Seven-Transistor Dual-Port Cell

Figure 18-11 shows a SRAM cell that is a variation on the basic 6T cell. It uses a differential write operation as before, with a pair of complementary bitlines connected to the cell by a wordline. However, this cell uses a single-ended read port. In this configuration, an additional wordline connects one side of the cell to an additional bitline that does not have a complement.

The read operation for a cell of this type does not use a sense amplifier.

This configuration is a one-read, one-write (1R1W) design. Simultaneous reading and writing of the cell is not supported.

Figure 18-11 7T Dual-Port Cell

To enable NanoTime to recognize this design, use the `create_memory` command as follows (this example is for a read operation):

```
nt_shell > create_memory -name mem_name -mode read \
                -bitcell_ports {nmos_write single_ended_nmos_read}
```

Eight-Transistor Dual-Port Cell With Differential Read and Write

A dual-port cell with eight transistors (8T cell) is similar to the 6T cell design, but it includes a second pair of access transistors that connects the bitcell to a new pair of bitlines. The new access transistors are controlled by an additional wordline. NanoTime recognizes the bitcell, the two wordlines, and the two pairs of bitlines.

Figure 18-12 shows an 8T dual-port SRAM cell. Either port can be used to read from or write to the bitcell.

The supported operations within one clock cycle are as follows:

- Reading from both ports (A and B)
- Reading or writing to one of the ports

Writing to both ports within the same clock cycle is an invalid operation.

Figure 18-12 8T Dual-Port SRAM Cell

To enable NanoTime to recognize this design, use the `create_memory` command as follows:

```
nt_shell > create_memory -name name -mode read \
                         -bitcell_ports {nmos_bidi nmos_bidi}
```

Ten-Transistor Dual-Port Cell

Figure 18-13 shows a ten-transistor (10T) design based on the standard 6T cell. Isolating the read and write operations is advantageous for data stability. Each latch output controls a pulldown transistor whose drain is connected to an access transistor, which in turn connects to bitline used exclusively for read operations. A new wordline controls the read access transistors.

This configuration is a one read, one write (1R1W) design. NanoTime recognizes the bitcell, the two wordlines, and the two bitlines. Simultaneous reading and writing of the cell is not supported.

Figure 18-13 10T Dual-Port SRAM Cell

To enable NanoTime to recognize this design, use the `create_memory` command as follows (this example is for a read operation):

```
nt_shell > create_memory -name name -mode read \
                         -bitcell_ports {nmos_write two_nmos_read}
```

Eight-Transistor Dual-Port Cell With Differential Write and Single-Ended Two-NMOS Read

Figure 18-14 shows a variation on the 8T dual-port cell. Isolating the read and write operations is advantageous for data stability. A separate wordline controls the read access for a single-ended read operation. NanoTime recognizes the bitcell, the two wordlines, and the three bitlines (two for write and one for read).

This configuration is a one read, one write (1R1W) design. Simultaneous reading and writing of the cell is not supported.

Figure 18-14 8T Dual-Port SRAM Cell

To enable NanoTime to recognize this design, use the `create_memory` command as follows (this example is for a read operation):

```
nt_shell > create_memory -name name -mode read \
                 -bitcell_ports {nmos_write single_ended_two_nmos_read}
```

Peripheral Circuit Topologies

This section contains the following topics:

- Precharge Circuits
- Sense Amplifiers
- Read and Write Multiplexers
- Write Drivers

See Also

- Memory Topology Recognition
- Memory Object Creation
- Bitline Tracking Circuits
- Memory Cell Topologies

Precharge Circuits

Precharging the bitlines to the supply voltage is a common technique for improving memory performance. NanoTime recognizes a three-transistor PMOS circuit of the pattern shown in Figure 18-15 as a precharge circuit. When the precharge enable signal is low, the PMOS pullup transistors allow the bitlines to charge up to the supply voltage and the equalization transistor keeps them tied together. During the read operation, the precharge enable signal is high, which disconnects the bitlines from the supply voltage and from each other.

Precharge circuit recognition is enabled by default. In rare cases, you might want to mark all precharge circuits manually. To disable automatic recognition, remove the `memory_precharge` structure type from the `topo_auto_search_class` variable.

Figure 18-15 Precharge Circuit

The structure in Figure 18-15 would be automatically recognized. To mark it manually, you would use the following commands:

```
set_transistor_direction -transistor bidi Mp2
mark_memory_precharge -precharge_clock {Mp0 Mp1 Mp2} \
                      -evaluate_net {bit bitb}
```

Sense Amplifiers

A sense amplifier detects a small voltage difference between a bitline and its complement, then amplifies the signal into a full logic 1 or 0 level signal for use as input to other circuit blocks. Each pair of bitlines (one bitline and its complement) requires one sense amplifier.

NanoTime recognizes sense amplifiers with the cross-coupled latch design shown in Figure 18-16. The three-transistor PMOS precharge circuit is not part of the sense amplifier pattern. During the read operation, the sense amplifier enable signal is raised after the bitcell sets the levels of the bitlines. The differential voltage of the bitlines then sets the logic state of the sense amplifier in a similar manner to the write operation on a bitcell.

Figure 18-16 Cross-Coupled Latch Sense Amplifier

The two PMOS transistors at the latch output nodes are optional; they isolate the bitlines from the sense amplifier output. NanoTime also recognizes sense amplifiers that do not contain the PMOS isolation transistors.

NanoTime recognizes this structure as a sense amplifier only if the sense amplifier enable signal connects to the clock net. The sense amplifier can have more than one sense amp enable transistor if the transistors are connected in parallel.

Sense amplifier recognition is enabled by default. To disable it, remove the `sense_amp` structure type from the `topo_auto_search_class` variable.

The structure in Figure 18-16 is automatically recognized. To mark it manually, you would use the following commands:

```
set_transistor_direction -transistor s2d {Mn0 Mn1}
mark_sense_amp -transistors {Mp0 Mp1 Mn0 Mn1 Mnsaen} \
                      -inputs {bit bitb} -enable {Mnsaen/g}
```

You can specify more than one transistor gate pin in the argument of the `-enable` option. For a set of parallel enable transistors, it is sufficient to specify any one of the parallel devices. NanoTime detects the parallel configuration and selects one device as a reference device. Other configurations of enable transistors are not supported.

Read and Write Multiplexers

For read mode, the bit-column mux refers to the multiplexer structure between the memory bits and the sense amplifiers (many inputs to one output). For write mode, the bit-column mux refers to the demultiplexer structure between the memory bits and the memory write driver circuits (one input to many outputs). The topology structure is the same and the topology recognition works for both circuits.

One bit-column mux is a set of pass transistors between multiple memory bits and one sense amplifier bit or one write driver, as shown in Figure 18-17.

Automatic multiplexer recognition is enabled by default. In most cases, automatic recognition is sufficient to recognize all mux topologies in a design.

NanoTime recognizes the pass transistors as bit-column mux structures if they satisfy these conditions:

- One side of each pass transistor connects to the memory bitlines.
- The other side connects to one sense amplifier bit or one write driver output. However, for single-ended read bitline multiplexers, the other side can connect to ordinary logic gates.
- All pass transistor gate pins connect to clock nets.
- All transistors are of the same type, either NMOS or PMOS.

Figure 18-17 Single-Level Bit-Column Multiplexer

In cases where a tree of two or more levels of pass gates occur between the memory cells and the sense amplifiers or write drivers, as shown in Figure 18-18, NanoTime recognizes the pass transistors as bit-column mux structures if they satisfy these conditions:

- The leaves of the tree are memory bitlines.
- The root of the tree is the sense amplifier or write driver output.
- All pass transistor gate pins connect to clock nets.
- All transistors in the tree are of the same type, either NMOS or PMOS.

Figure 18-18 Multiplexer Tree

To disable automatic recognition, remove the `mux` structure type from the `topo_auto_search_class` variable. This disables recognition of all multiplexer circuits throughout the design. To disable only bit-column multiplexer structures, set the `topo_auto_find_bit_column_mux` variable to `false` but leave the `mux` structure type in the `topo_auto_search_class` variable.

By default, all pass transistors within each multiplexer from the same output must have the same resistance. To allow a specified difference in transistor drive strength, set the `topo_mux_drive_res_ratio` variable to the maximum allowed ratio between the highest and lowest resistances in the multiplexer tree structure.

The setting of the `topo_mux_strict_enable_rules` variable does not apply for bit-column multiplexer recognition as it does for other multiplexer structures. During bit-column multiplexer recognition, NanoTime creates a logic constraint making all of the mux select inputs mutually exclusive. If this constraint is not feasible, the multiplexer cannot be recognized.

A simple multiplexer structure is shown in Figure 18-19.

Figure 18-19 Read Multiplexer

This structure would be automatically recognized. To mark it manually, use the following commands:

```
set_transistor_direction -transistor d2s { Mp0 Mp1 Mp2 Mp3}
mark_mux -output_net data  -select_pins { Mp0/g Mp2/g }
mark_mux -output_net datab -select_pins { Mp1/g Mp3/g }
set_logic_constraint -at_most_one_off { sel0 sel1 }
```

Write Drivers

The NanoTime tool does not recognize any write driver topologies automatically due to the large number of possible circuit configurations. You must mark all write drivers manually using the `mark_memory_write_circuit` command. This section provides several examples of write drivers and the commands for manually marking them.

The NanoTime tool requires all write driver circuits to be clocked.

Write Driver Example 1

Figure 18-20 is an example of a write driver circuit.

Figure 18-20 Write Driver Circuit 1

To mark this structure manually, use the following commands:

```
mark_cross_coupled_pmos -transistors { Mp0 Mp1 }
mark_cross_coupled_pmos -transistors { Mp2 Mp3 }
mark_memory_write_circuit -outputs bit0  -transistors Mn0 -enable Mn0/g
mark_memory_write_circuit -outputs bitb0 -transistors Mn1 -enable Mn1/g
mark_memory_write_circuit -outputs bit1  -transistors Mn2 -enable Mn2/g
mark_memory_write_circuit -outputs bitb1 -transistors Mn3 -enable Mn3/g
```

Write Driver Example 2

Figure 18-21 is another example of a write driver circuit.

Figure 18-21 Write Driver Circuit 2

To mark this structure manually, use the following commands:

```
set_transistor_direction -transistor d2s { Mpadb0 Mpadb1 Mpad0 Mpad1 }
mark_memory_write_circuit -outputs bit0  -transistors Mn0  -enable Mn0/g
mark_memory_write_circuit -outputs bit1  -transistors Mn1  -enable Mn1/g
mark_memory_write_circuit -outputs bitb0 -transistors Mnb0 -enable Mnb0/g
mark_memory_write_circuit -outputs bitb1 -transistors Mnb1 -enable Mnb1/g
```

The write driver circuit in Figure 18-21 contains clock-gating structures that must be correctly marked to ensure clock propagation and to obtain appropriate timing checks. To mark the clock-gating topologies, use the following commands:

```
mark_clock_gate -negative_enable {Mpd/g} \
                -clock {Mpad0/g Mnad0/g) -output {wd0}
mark_clock_gate -negative_enable {Mpd/g} \
                -clock {Mpad1/g Mnad1/g) -output {wd1}
mark_clock_gate -negative_enable {Mpdb/g} \
                -clock {Mpadb0/g Mnadb0/g) -output {wdb0}
mark_clock_gate -negative_enable {Mpdb/g} \
                -clock {Mpadb1/g Mnadb1/g) -output {wdb1}
```

Write Driver Example 3

Figure 18-22 is a third example of a write driver circuit.

Figure 18-22 Write Driver Circuit 3

To mark this structure manually, use the following commands:

```
set_logic_constraint -one_hot { N1 N2 }
mark_memory_write_circuit -outputs { out1 out2}  \
                          -transistors { X1/M* X2/M* } -inputs { N1 N2 }
mark_mux -output_net out1 -select_pins {Mn1/g Mn2/g}
mark_mux -output_net out2 -select_pins {Mn3/g Mn4/g}
```

Write Driver Example 4

Figure 18-23 is another example of a write driver circuit.

Figure 18-23 Write Driver Circuit 4

To mark this structure manually, use the following commands:

```
set_logic_constraint -one_hot { data datab }
set_logic_constraint -one_hot { be ben }
mark_memory_write_circuit -outputs { bit0 bitb0 }  \
        -inputs { data datab be ben } \
        -transistors { X1/m* X2/m* X3/m* X4/m* X5/m* X6/m* } \
        -enable { X5/m* X6/m* }
mark_memory_write_circuit -outputs { bit1 bitb1 }  \
        -inputs { data datab be ben } \
        -transistors { X1/m* X2/m* X3/m* X4/m* X7/m* X8/m* } \
        -enable { X7/m* X8/m* }
```

The signals we0, we0_n, we1, and we1_n must be clock signals.

Clock Propagation for Memories

NanoTime treats many memory structures as clocks. Make sure to set up clocks and clock propagation properties to obtain a correct timing analysis.

Use the `report_clock_arrivals` command to verify that clock endpoints have the expected waveforms. You might need to mark some topologies and networks using commands such as the `mark_clock_gate` command or the `mark_clock_network` command to obtain correct clock propagation.

The following circuits are expected to be part of the clock network:

- Wordline
- Bitline precharge
- Sense amplifier precharge
- Sense amplifier enable
- Read MUX select
- Write MUX select
- Write circuit enable

Dynamic clock simulation (DCS) is often used in memory characterization to obtain accurate signal arrivals. Dynamic simulation fully accounts for interacting signals throughout the designated part of the design.

This section includes the following topics:

- Clocking for Pipelined Memories
- Dynamic Clock Simulation for Memory Analysis

Clocking for Pipelined Memories

By default, NanoTime memory analysis assumes that the wordline signal initiates memory read operations. The wordline signal is used as a reference; all delays through the bit column go through the wordline input pin. No path tracing occurs from the read multiplexer or sense amplifier enable inputs.

In a pipelined clocking scheme, the sense amplifier enable signal occurs a half cycle later than the wordline and read multiplexer signals, as shown in Figure 18-24. NanoTime correctly incorporates the half cycle offset into the analysis. However, paths from the sense amplifier enable signal are not traced and are not available for reporting or for inclusion in an extracted timing model.

To enable path tracing through sense amplifier enable signals, set the `timing_enable_path_through_sense_amplifier` variable to `true` before executing the `check_topology` command. This variable enables path tracing through the sense amplifier enable signals. The tool assumes that the sense amplifier enable signal is initiated from the second clock edge and the other memory column inputs, such as the wordline enable signal, are initiated from the first clock edge.

For path reporting through the sense amplifier enable signal of a memory bit column, multiple combinations of wordline signals and read multiplexer signals are possible. NanoTime reports the combinations that cause the minimum and maximum delay through the selected bitcells.

Limitations are as follows:

- Paths through read multiplexer enable signals are not traced.
- Multiple transitions of a single input signal are not considered.

Figure 18-24 Pipelined Memory Clocking Scheme

Dynamic Clock Simulation for Memory Analysis

Dynamic clock simulation is often more complicated for memory designs than for general digital logic, as follows:

- Memory stages might contain a very large number of side inputs.
- Memories typically have multiple wordline driver circuits that share a common pullup transistor and a common pulldown transistor. During any memory access cycle, only one of the driver circuits is active at a time. Therefore, the common pullup and pulldown transistors are sized to be able to drive only one wordline adequately.
- SRAM sense amplifiers sometimes contain a self-timed circuit to produce the sense amplifier enable (SAE) signal. The closing edge of the sense amplifier enable signal is triggered by a special SAE_OFF signal, which is generated from the outputs of the sense amplifier. This creates a feedback loop that contains both a clock path (from SAE_OFF to SAE) and a datapath (from the sense amplifier output to SAE_OFF). The purpose of this signal is to automatically turn off the sense amplifier enable signal.

Dynamic Clock Simulation Setup Procedure

This general procedure includes steps needed to handle side inputs, sense amplifier feedback circuits, and common pullup or pulldown transistors. You can omit steps marked as optional if your simulation does not include these elements.

1. Enable dynamic clock simulation by setting the `dcs_enable_analysis` variable to `true`.
2. Define the clock sources and design topology.
3. (Optional) Exclude transistors and their fanouts from the DCS region by using the `mark_instance -exclude_from_dcs` command.
4. (Optional) Terminate dynamic clock simulation in specific paths by using the `mark_clock_network -end_dcs` command.
5. (Optional) If you use a pipelined clocking scheme, set the `timing_enable_path_through_sense_amplifier` variable to `true`.
6. (Optional) To save information about clock arrivals in the critical path through the DCS region, set the `dcs_enable_detailed_path_reporting` variable to `true`.
7. Execute the `check_topology` and `check_design` commands.
8. Examine the list of side inputs that cause DCS-001 warning messages.
9. (Optional) Set logic on the side inputs in the warning list with the `set_dcs_input` command. For self-timed circuits, include the `-synchronized_data` option.

10. If you set side inputs logic in Step 9, use the `check_design` command again to obtain the corrected simulation.
11. Use the `report_clock_arrivals` command to verify that the clock endpoints have the expected waveforms. If some nets are not marked as clocks, mark them manually by using the `mark_clock_network` command.
12. Execute the `trace_paths -clock_only` command.
13. Proceed with full path tracing.
14. Execute the `report_clock_arrivals` command to verify clock net behavior.

See Also

- Side Input Logic
- Shared Pullup and Pulldown Transistors
- Self-Timed Circuits

Case Analysis for Memories

Case analysis is the method of specifying read, write, and other operational modes. NanoTime allows you to define fixed logic for case analysis while performing full path tracing on those ports.

NanoTime can only perform one operation (read or write) at a time. This behavior is usually controlled by setting a constant on a single write-enable (WEN) port that is used to control writing into or reading from the memory macro. Depending on the design, additional case settings might be necessary, such as controls on an output-enable port or on scan-related ports.

To analyze a memory design, you typically run NanoTime in two passes, one for read mode and one for write mode. During these runs, some of the inputs (such as WEN) are set to fixed logic values. NanoTime does not generate timing paths from ports with fixed logic values or check timing constraints involving these paths. This prevents them from being properly represented in a timing model.

To include paths from fixed logic ports in path tracing, use the `set_case_analysis` command with the `-enable_trace_from` option. The argument of the command must be a boundary input port, not an internal pin or output port. This command temporarily removes the effect of the case analysis setting while performing path tracing from that port; it does not affect case logic settings on other ports.

The `set_case_analysis` command only affects path tracing. If you use the `set_case_analysis` command with the `-enable_trace_from` option before running the `check_topology` command, the logic setting is used during topology recognition and transistor direction setting.

Timing Checks for Memories

NanoTime creates memory-specific timing checks based on the memory topology. These timing checks are based on the user-specified functional mode (read or write).

Table 18-1, Table 18-2, and Table 18-3 list the memory-specific timing checks. Each timing check has a type (setup, hold, or pulsewidth), a checked pin, a reference pin, and a tag that is unique to the pair of pins involved. The label is the description that appears in reports.

To filter memory-related timing paths, use the `endpoint_type` attribute with the `get_timing_paths` command. The attribute values are of the form memory_column_inputXX, where XX is the timing check tag. For example,

```
set sram_maxpaths [filter_collection [get_timing_paths -max] \
      “endpoint_type = ~memory_column_inputM1”]
report_paths $sram_maxpaths > sram_maxpaths.rpt
```

Table 18-1 Memory Column Hold Checks

Type	Checked Pin	Reference Pin	Tag	Label
hold	bitline precharge	read mux	M1	precharge on after read mux off
hold	bitline precharge	wordline	M2	precharge on after word line off
hold	bitline precharge	write enable	M3	precharge on after write mux/enable off
hold	bitline precharge	sense amp enable	M4	precharge on after sense amp enable off
hold	sense amp out precharge	read mux	M5	sense amp out precharge on after read mux off (present only if sense amp output is precharged separately from inputs)
hold	sense amp out precharge	sense amp enable	M6	sense amp out precharge on after sense amp enable off
hold	sense amp precharge	read mux	M7	sense amp precharge on after read mux off
hold	sense amp precharge	sense amp enable	M8	sense amp precharge on after sense amp enable off
hold	data	wordline	M10	data valid after word line off
hold	write input	write enable	M11	input valid after write mux/enable off
hold	write mux/enable	wordline	M12	write mux/enable off after wordline off

Table 18-2 Memory Column Setup Checks

Type	Checked Pin	Reference Pin	Tag	Label
setup	bitline precharge	read mux	M1	precharge off before read mux on
setup	bitline precharge	wordline	M2	precharge off before word line on
setup	bitline precharge	write enable	M3	precharge off before write mux/enable on
setup	bitline precharge	sense amp enable	M4	precharge off before sense amp enable on
setup	sense amp out precharge	read mux	M5	sense amp precharge off before read mux on (present only if sense amp output is precharged separately from inputs)
setup	sense amp out precharge	sense amp enable	M6	sense amp precharge off before sense amp enable on
setup	sense amp precharge	read mux	M7	sense amp precharge off before read mux on
setup	sense amp precharge	sense amp enable	M8	sense amp precharge off before sense amp enable on
setup	sense amp enable	read mux	M9	1. read mux on before sense amp enable on 2. sense amp enable on before read mux off, which can be disabled by setting a variable
setup	data	wordline	M10	data valid before word line off
setup	input	write enable	M11	input valid before write mux/enable on
setup	write enable	wordline	M12	write mux/enable on before wordline on
setup	wordline	read mux	M13	word line on before read mux on (depends on a variable setting)
setup	read mux	wordline	M13	read mux on before word line on (depends on a variable setting)
setup	bitline	wordline	M16	bit line valid before word line on (reported only by setting a variable)
setup	bitline	precharge enable	M17	bitline charged before precharge off (reported only by setting a variable)

Table 18-3 Memory Column Pulse Width Timing Checks

Type	Checked Pin	Reference Pin	Tag	Label
pulsewidth	wordline	wordline	M14	word line pulse width
pulsewidth	sense amp enable	sense amp enable	M15	sense amp enable pulse width

You can add, remove, or modify several timing checks by setting NanoTime variables. These variables must be set before running the `check_design` command.

- Setup check M9

  By default, M9 includes two setup timing checks: (1) read mux on before sense amp enable on and (2) sense amp enable on before read mux off.

  The `timing_memory_remove_sense_amp_before_read_mux` variable controls whether the second M9 timing check is reported. The default is `false`, which means the timing check exists. To remove the timing check, set this variable to `true`. The first M9 timing check is not affected by this variable.

- Setup check M13 (wordline on before read mux on)

  By default, NanoTime requires the wordline signal to be on before the read mux signal. The `timing_memory_read_mux_before_wordline` variable controls this requirement; the default is `false`. To require the read mux signal to be on before the wordline signal, set the variable to `true`. The label of the new timing check is “read mux on before wordline on.”

- Setup check M16 (bit line valid before word line on)

  Set the `timing_memory_bitline_valid_before_wordline` variable to `true` to add timing check M16. Setup check M16 is a more conservative version of check M10. Timing check M10, which is automatically defined, requires the data into a bitcell to be valid before the wordline signal that controls the input into that bitcell is turned off. Check M16 requires that the bitline signal be valid before the wordline device is turned on.

- Setup check M17 (bitline charged before precharge off)

  Set the `timing_memory_measurement_bitline_precharge_percentage` variable to a nonzero value to add timing check M17. Setup check M17 verifies that the memory bitlines reach a precharged state in preparation for a subsequent read or write operation.

  The default is 0.0, which indicates that precharge timing validation does not take place. Specify the percentage of the supply voltage that represents a precharged state.

Figure 18-25 to Figure 18-29 illustrate which edges are involved for some of the memory circuit timing checks. These diagrams are for illustration only and do not imply any requirements for a memory design.

Figure 18-25 Read Setup Timing

Code	Description
M1	precharge off before read mux on
M2	precharge off before wordline on
M4	precharge off before SAE on
M7	sense amp precharge off before read mux on
M8	sense amp precharge off before SAE on
M9	read mux on before SAE on, SAE on before read mux off (can disable with a variable)
M13	wordline on before read mux on (can swap order by setting a variable)

Figure 18-26 Read Hold Timing

Code	Description
M1	precharge on after read mux off
M2	precharge on after wordline off
M4	precharge on after SAE off
M7	sense amp precharge on after read mux off
M8	sense amp precharge on after SAE off
M14	wordline pulse width
M15	sense amp enable pulse width

Figure 18-27 Write Setup Timing

Code	Description
M2	precharge off before wordline on
M3	precharge off before write mux/enable on
M10	data valid before wordline off
M11	input valid before write mux/enable on
M12	write mux/enable on before wordline on

Figure 18-28 Write Hold Timing

Code	Description
M2	precharge on after wordline off
M3	precharge on after write mux/enable off
M10	data valid after wordline off
M11	input valid after write mux/enable off
M12	write mux/enable off after wordline off

Figure 18-29 Memory Precharge Timing Check

Path Tracing and Timing Analysis

During timing analysis, NanoTime uses HSPICE for memory column simulations in the background to obtain accurate results. After timing analysis is complete, you can create extracted timing models. After models are available for all of the memory operational modes, you can merge the models for use in circuit-level timing analysis tools.

This section contains the following topics:

- General Setup
- Automatic HSPICE Simulation
- Model Extraction

General Setup

The steps of the general memory analysis procedure are as follows:

1. Load parasitics by using the `read_parasitics` command. You can use the `complete_net_parasitics` command to complete all nets in the memory macro that have incomplete RC networks.
2. (Optional) Use the `set_memory_analysis_type` command.
3. (Optional) Use the `set_memory_bit_selected` and `remove_memory_bit_selected` commands to create a set of bitcells for analysis.
4. Set other simulation parameters by using the `set_simulation_attributes` command.
5. Check the clock paths before running a full path trace by using the `trace_paths -clock_only` command, followed by the `report_clock_arrivals` command to determine whether all the expected clock endpoints are showing valid waveforms.
6. Use the `trace_paths` command to carry out the memory analysis.
7. Extract the timing model with the `extract_model` command.
8. After creating timing models for different operating modes, merge the models with the `merge_models` command.

The following commands specify analysis criteria for memories:

- `set_memory_analysis_type`
- `set_memory_bit_selected`
- `remove_memory_bit_selected`

The default behavior is to analyze one minimum and one maximum delay bit from each wordline and bitline. This is equivalent to specifying a value of 1 for the `-minmax` option of the `set_memory_analysis_type` command. You can analyze more bitcells by specifying an integer of 2 or greater per wordline and bitline.

If you enable pipelined clocking, NanoTime also analyzes one minimum and one maximum delay path from the sense amplifier enable signal through selected bitcells.

You can load parasitics by using the `read_parasitics` command. If you use the `complete_net_parasitics` command to complete all nets in the memory macro that have incomplete RC networks, you must use the command before executing the `set_memory_analysis_type` command.

You can specify the bitcells to use for analysis. To enable this capability, specify the `-custom` option of the `set_memory_analysis_type` command and use the `set_memory_bit_selected` and `remove_memory_bit_selected` commands.

Alternatively, bitcells can be selected or deselected from a collection of corresponding RAM topology objects. Create the collection by using the `get_topology` command, as follows:

```
set_memory_bit_selected [get_topology -structure_type ram -all]
```

This command selects all the bitcells for memory analysis. Use the `report_memory` command to check the number of bits selected for analysis.

See Also

- Clocking for Pipelined Memories

Automatic HSPICE Simulation

NanoTime automatically uses HSPICE simulation for bitcell analysis. HSPICE simulation is integrated with the delay calculation for bitlines, multiplexers, and sense amplifiers. You do not need to provide stimulus because NanoTime automatically sets up the HSPICE runs.

Use the `set_simulation_attributes` command to set up the HSPICE simulations.

NanoTime launches HSPICE simulations in advanced client/server mode. If a distributed processing environment is available, NanoTime runs HSPICE simulations in parallel if you set the `memory_enable_parallel_hspice` variable to `true`.

Model Extraction

You use the `extract_model` command to create timing models as in standard NanoTime flows. By default, paths from the first clock edge through the wordline to the memory read output are included in the model. If you enable pipelined clocking by setting the `timing_enable_path_through_sense_amplifier` variable to `true`, paths from the second clock edge through the sense amplifier enable to the memory read output are also included in the model.

You can create NLDM or CCS timing models. Model extraction captures the read or write mode onto the model arcs and contains the library information. Use model extraction commands such as the `set_model_input_transition_indexes` command and the `set_model_load_indexes` command to configure the table indexes and select the format and content of the timing model.

Use the `merge_models` command to merge timing and noise models generated by NanoTime for various analysis modes.

Note:

Do not use the `-mode` option when creating memory models because the `merge_models` command cannot merge moded models. Use the `-when` option instead.

If you are simulating many memory circuits with multiple operating modes, running multiple NanoTime sessions in parallel reduces turnaround time and provides methods to automate analysis setup and reporting with Tcl scripts.

See Also

- Clocking for Pipelined Memories
- Parallel NanoTime Runs
- Merging Models

Memory Analysis Reporting

After the memory analysis is complete, you can use the `report_paths` command to create minimum and maximum timing reports.

The following sections describe commands that provide specialized information about the memory circuit:

- The report_memory Command
- The report_bitline and report_wordline Commands
- The report_measurement Command

The report_memory Command

After you execute the `create_memory` command, you can display information about the memory with the `report_memory` command.

If you use the `-bitlines` and `-wordlines` options with the `report_memory` command, the report lists the nets identified as bitlines and wordlines. For example:

```
nt_shell> report_memory -bitlines -wordlines

********************************
Report: memory
Design: mem_6tsedw
...
********************************

             # Word  # Bit  # Bit  Analysis  Analysis    # Bits
Name          Lines  Lines  Cells  Mode      Type      Selected Top Cell
------------ ------ ------ ------  --------  --------- -------- ---------
read_mem          2      4      4  read      minmax(1)        4  --

Port types of memory ‘read_mem’:

     Port Type
     --------------------------------------------
     single_ended_nmos_read_differential_write

Wordlines of memory ‘read_mem’:

     Wordline Name
     --------------------------------------------
     w1_01_[0]
     w1_01_[10]
     ...

Bitlines of memory ‘read_mem’:

     Bitline Name                    Complement Name
     ----------------------------  -------------------------------
     b0_b1[100]                      b0_b1b[100]
     b0_b1[101]                      b0_b1b[101]
     ...
```

The report_bitline and report_wordline Commands

Use the `report_bitline` and `report_wordline` commands to obtain detailed information about the bitlines and wordlines of the selected memory cells.

The bitline report includes information about the associated precharge, multiplexer, and sense amplifier structures. In the following example, the `bitlines` attribute of the memory object is used to provide input to the `report_bitline` command:

```
nt_shell> report_bitline [get_attribute read_mem bitlines]

*********************
Report: bitline
Design: memtop
...
*********************
...
Attributes:
   s - selected

Memory bitline ‘Xmem/Xmem0/bit0’:

   Item                Value
   ------------------- ------------------------------------
   Memory              read_mem
   Bitline Type        bidirectional
   Complement Net      Xmem/bitb0
   Precharge Pin       Xmem/Xbitlinepre0/Mp0/g
   Precharge Net       Xmem/prech
   Precharge Txs       {Xmem/Xbitlinepre0/Mp0 Xmem/Xbitlinepre0/Mp1 ...}
   Num Bit Cells       2
   Read Mux Select     {Xmen/Xrdmux/Mp0/g Xmem/Xrdmux/Mp1/g}
   Read In Net         Xmem/data0
   Sense Amp Input     Xmem/data0
   Sense Amp Enable    Xmem/Xsenseamp-en/Mnsaen/g
   S-Amp Prech Pin     Xmem/Xsenseamp-en/Xpre/Mp0/g
   S-Amp Prech Net     Xmem/prechsamp
   Write Out Nets      Xmem/bit0
```

If you use the `-bitcells` option with the `report_bitline` command, the report also includes the bitcell name and wordline for every bitcell associated with the bitline. The additional information appears after the line with the number of bitcells. For example:

```
...
   Num Bit Cells        2
   Wordline     0       w1_01_[0]
   Bitcell      0    s  {Xmem/xcore_d4_w0/q1 Xmem/xcore_d4_w0/qb1}
   Wordline     1       w1_02_[1]
   Bitcell      1    s  {Xmem/xcore_d4_w1/q1 Xmem/xcore_d4_w1/qb1}
...
```

The wordline report includes the number of bitcells associated with the wordline. If you use the `-bitcells` option with the `report_wordline` command, the report also lists the bitcell and bitline names for every bitcell associated with the wordline. For example:

```
nt_shell> report_wordline ‘Xmem/rwaddr0’

*********************
Report: wordline
Design: memtop
...
*********************

Attributes:
   s - selected

Memory wordline ‘Xmem/rwaddr0’:

   Item                Value
   ------------------- -----------------------------------
   Memory              read_mem
   Num Bit Cells       2
   Bitlines    0       {Xmem/bit0 Xmem/bitb0}
   Bitcell     0     s {Xmem/Xsram6t_0/Xsram0/b Xmem/Xsram6t_0/Sxram0/bb}
   Bitlines    1       {Xmem/bit1 Xmem/bitb1}
   Bitcell     1     s {Xmem/Xsram6t_1/Xsram0/b Xmem/Xsram6t_1/Sxram0/bb}
```

The report_measurement Command

The `report_measurement` command reports the differential voltage across the sense amplifier inputs when the sense amplifier enable signal crosses a threshold. The differential voltage is compared to a check value and is considered to be a violation if it is below that value.

The following variables control the measurement:

- The `timing_memory_measurement_sense_amp_trigger_threshold_percentage` variable

  When the sense amplifier enable signal crosses the threshold defined by this variable, NanoTime measures the differential voltage across the two sense amplifier inputs. The threshold is a percentage of the supply voltage; the default is 0.5 or 50 percent.

- The `timing_memory_measurement_differential_voltage_percentage` variable

  This variable specifies the percentage of the supply voltage below which a measurement is considered to be a violation. The default is 0.1 or 10 percent. The test threshold is calculated by multiplying the supply voltage by the variable value. The calculated value appears in the report under the column heading “Check value.”

Measurement names are automatically generated. Violations against the check values are not indicated explicitly. However, you can use the `-violators` options with the `report_measurement` command to create a report that lists only the violations.

To limit the size of the report, you can also use the `-name` and `-trigger` options with the `report_measurement` command.

The following is an example of the output from the `report_measurement` command:

```
nt_shell> report_measurement -trigger -violators
****************************************
Report: measurement
Design: memtop
...
****************************************

****************************************
Report: differential_voltage_measurement
Design: memtop
...
****************************************

Trigger Switching Direction:
   f - Falling
   r - Rising


          Differen                      Keep
             -tial                Path value Check
Name       voltage Trigger        type  type value Switch net Measured nets
---------- ------- ---------  --- ---- ----- ----- ---------- -------------------
diffVolt_0  0.1435 Xmem/rwaddr0 r  max   min 0.096 Xmem/Xsenseamp/buf
                                                                {Xmem/data0 Xmem/datab0}
diffVolt_1  0.1489 Xmem/rwaddr0 r  max   max 0.096 Xmem/Xsenseamp/buf
                                                                {Xmem/data0 Xmem/datab0}
diffVolt_2  0.1202 Xmem/rwaddr0 r  min   min 0.096 Xmem/Xsenseamp/buf
                                                                {Xmem/data0 Xmem/datab0}
diffVolt_3  0.1255 Xmem/rwaddr0 r  min   max 0.096 Xmem/Xsenseamp/buf
                                                                {Xmem/data0 Xmem/datab0}

Total differential voltage measurements: 4
```

Black Box Memory Analysis

You can save runtime and memory by using a timing model to represent the bitcell array. In this mode, the NanoTime tool skips the bitcell simulations but performs bit column simulations without the bitcells, including the read multiplexer, the sense amplifier, and the bitline capacitances. These simulations are faster than full array simulations.

This analysis method is known as black box memory analysis or the skip-array mode. This method results in the following actions:

- The NanoTime tool assumes that the memory contains one bidirectional port, corresponding to port type `nmos_bidi`.
- The tool does not perform any write mode simulations. In other words, the wordlines, write multiplexer, write driver, and precharge circuits are not considered.
- The tool performs one read mode simulation that measures the delay from the sense amplifier enable to the sense amplifier output.
- Delays are not calculated through the memory array.
- Not all memory timing checks are available.

Most SRAM designs need only a single NanoTime run in skip-array mode to cover both the read and write paths. However, combining the read and write modes into a single run might introduce pessimism for some designs due to differences in arrival times between read and write operations, resulting in more pessimistic clock arrival windows or more signal integrity window overlap. For these designs, it might be advantageous to run the analysis once for read mode and once for write mode. Write mode is supported in skip-array analysis, but no paths are reported through the array and no timing checks are supported at the boundaries of the array.

Memory Object Creation

To use the black box analysis method, set the `-mode` option to `skip_array`, as shown in the following example:

```
create_memory -name memABC -mode skip_array -bitcell_ports nmos_bidi
```

The `-bitcell_ports` option must be set to `nmos_bidi`.

All other setup requirements for the memory analysis flow are still valid, especially the setup of the clock network for all inputs to the bit columns. The exception is wordline timing, because wordline arrivals are not used in this analysis mode.

Topology Recognition

To enable topology recognition of peripheral circuits, you must mark all bitlines in the array by using the `mark_memory_bitline` command, as follows:

- The `-memory` option must specify the same name specified by the `-name` option in the `create_memory` command.
- The argument is a list of exactly two objects (nets, ports, or pins) that constitute a bitline and its complement. The `is_bitline` attribute is set for a bitline and bitline complement marked by the `mark_memory_bitline` command.
- You can use multiple `mark_memory_bitline` commands to mark multiple pairs of bitlines. All bitlines in the memory must be marked.

Because the memory array is not analyzed, you must specify the high and low voltages at the sense amplifier cross-coupled nodes to be used when calculating the delay from the sense amplifier enable signal to the sense amplifier output. To set the differential voltages, use the `mark_sense_amp` command with the `-diff_high` and `-diff_low` options; both options must be used.

The `report_measurement` command does not report manually specified sense amplifier voltages. To examine these voltages, use one of the following methods:

- Use the `report_paths` command with the sense amplifier enable net name, as shown in the following example:

  ```
  report_paths -max -through [get_nets -hier *saen]
  ```

- Use the `report_attribute` command to list all attributes of the sense amplifier, as shown in the following example:

  ```
  set dv [get_topology -all -structure_type sense_amp]
  report_attribute -application $dv
  ```

- Use the `get_attribute` command to examine specific attributes of the sense amplifier, as shown in the following example:

  ```
  #net1 is the sense amplifier input net
  get_attribute [get_topology -of net1 -structure_type sense_amp] \
                differential_high_voltage
  get_attribute [get_topology -of net1 -structure_type sense_amp] \
                differential_low_voltage
  ```

Example Flow

An example flow for memory black box analysis is as follows. In this example, the `foreach_match` command is used to mark all of the bitlines in subcircuit abc.

```
# add cell library paths to link path
set search_path ". ./models $search_path"
set link_path " * 6tarray.db ..."
set link_prefer_model_port {*}
read_db 6tarray.db
...
register_netlist -format spice "./tech.sp ./6tarray.sp"
link_design -verbose -keep_capacitive_coupling "6tarray"
...
create_memory -name mem_blk -mode skip_array -bitcell_ports nmos_bidi
...
mark_sense_amp -diff_high 1.2 -diff_low 1.08 -transistors {...}
...
foreach_match abc -command {
   mark_memory_bitline {bl bl_b} -memory mem_blk
}
check_topology
...
```

Timing Checks

Table 18-4 lists the timing checks available with black box memory analysis.

Table 18-4 Timing Checks for Black Box Memory Analysis

Type	Checked Pin	Reference Pin	Tag	Label
setup	bitline precharge	read mux	M1	precharge off before read mux on
setup	bitline precharge	write enable	M3	precharge off before write mux/enable on
setup	bitline precharge	sense amp enable	M4	precharge off before sense amp enable on
setup	sense amp out precharge	read mux	M5	sense amp precharge off before read mux on (present only if sense amp output is precharged separately from inputs)
setup	sense amp out precharge	sense amp enable	M6	sense amp precharge off before sense amp enable on
setup	sense amp precharge	read mux	M7	sense amp precharge off before read mux on
setup	sense amp precharge	sense amp enable	M8	sense amp precharge off before sense amp enable on
setup	sense amp enable	read mux	M9	sense amp enable on before read mux off (can be disabled by setting a variable)
setup	input	write enable	M11	input valid before write mux/enable on
pulse width	sense amp enable	sense amp enable	M15	sense amp enable pulse width

Memory-Specific Syntax

Table 18-5 lists the NanoTime commands specific to memory analysis.

Table 18-5 Commands Specific to Memory Analysis

Command	Description
`create_memory`	Creates a memory object on the netlist for timing analysis
`get_memory`	Creates a collection of memory objects
`remove_memory`	Removes a memory object from design
`mark_memory_bitline`	Identifies bitlines for black-box memory array analysis
`mark_memory_precharge`	Identifies the elements of the memory precharge
`mark_memory_write_circuit`	Identifies the elements of the memory write circuit
`mark_sense_amp`	Identifies the elements of the memory sense amplifier
`erase_memory_precharge`	Erases a memory precharge topology
`erase_memory_write_circuit`	Erases a memory write circuit topology
`erase_sense_amp`	Erases a memory sense amplifier topology
`report_memory`	Reports structural information for a memory object
`report_bitline`	Reports bitline information for a memory object
`report_wordline`	Reports wordline information for a memory object
`report_measurement`	Reports differential voltage measurements
`set_memory_analysis_type`	Configures the memory for the most appropriate analysis type
`set_memory_bit_selected`	Marks subset of memory bits for analysis to reduce runtime
`remove_memory_bit_selected`	Deselects a memory bit from the analysis

Table 18-6 lists the NanoTime variables specific to memory analysis.

Table 18-6 Variables Specific to Memory Analysis

Variable	Description
`memory_enable_parallel_hspice`	Enables use of parallel processing for HSPICE analysis
`topo_auto_find_bit_column_mux`	Enables automatic recognition of column MUXs in memory topologies
`timing_enable_path_through_sense_amplifier`	Enables path tracing through sense amplifier enable signals in pipelined memories
`timing_memory_measurement_bitline_precharge_percentage`	Adds a timing check to verify that the bitlines reach a precharged state before the precharge enable signal is turned off
`timing_memory_measurement_differential_voltage_percentage`	Specifies the percentage of the supply voltage below which a sense amplifier differential voltage measurement is considered to be a violation
`timing_memory_measurement_sense_amp_trigger_threshold_percentage`	Specifies the percentage of the supply voltage that triggers a measurement of the voltage across the sense amplifier inputs
`timing_memory_read_mux_before_wordline`	Changes required order of signal arrivals for a setup check between the read MUX control and the wordline control
`timing_memory_remove_sense_amp_before_read_mux`	Removes timing check requiring the sense amplifier enable pin to be valid before the read MUX control signal
`timing_memory_bitline_valid_before_wordline`	Adds a timing check that requires the bitline signal at the input into the wordline-controlled device be valid before the wordline device is turned on
`timing_memory_simultaneous_read`	Allows simultaneous reading from both ports in a dual-port design
`timing_memory_use_preswitched_read_mux`	Allows the read MUX control signal to remain in a fixed ON state during simulation of a memory bit column

19

Object Attributes

An attribute is a string or value associated with an object in the design that carries some information about that object. You can write programs in Tcl to get attribute information from the design database and generate custom reports about the design.

The following sections describe attributes and how to use them:

- Using Attributes
- Creating Custom Reports With Attributes

NanoTime attributes are listed in the tables at the end of the chapter, organized by object class.

Using Attributes

Table 19-1 lists the commands for working with attributes in NanoTime.

Table 19-1 Attribute Commands

Attributes	Description
`list_attributes`	Lists the names of available attributes by object class.
`get_attribute`	Retrieves the value of one attribute associated with one object.
`report_attribute`	Displays the values of all attributes associated with one or more objects.

Listing Attribute Names

The `list_attributes` command displays an alphabetically sorted list of the attributes in NanoTime: the names of the available attributes, not the values of any attributes.

Note:
NanoTime does not support user-defined attributes or imported attributes.

NanoTime uses many attributes. It is often useful to limit the listing to a specific object class by using the `-class` option. You must include the `-application` option to show all attribute names. An example of an attribute list is shown here.

```
nt_shell> list_attributes -class pin -application

****************************************
Report : List of Attribute Definitions
...
****************************************
Properties:
    A - Application-defined
    U - User-defined
    I - Importable from design/library (for user-defined)

Attribute Name            Object     Type      Properties  Constraints
-----------------------------------------------------------------------
arrival_max_fall          pin        float     A
arrival_max_rise          pin        float     A
arrival_min_fall          pin        float     A
arrival_min_rise          pin        float     A
clock_latency_fall_max    pin        float     A
clock_latency_fall_min    pin        float     A
clock_latency_rise_max    pin        float     A
...
```

Reporting All Attributes for an Object

Use the `report_attribute` command to generate a report of attributes associated with specified objects in the design. You must use the `-application` option to show all of the attributes. For application attributes that are of the type *collection,* the name of the first object in the collection is displayed.

```
nt_shell> report_attribute [get_nets {in}] -application

****************************************
Report : Attribute
...
****************************************

Design    Object    Type      Attribute Name              Value
-----------------------------------------------------------------
Design_A   in       string    object_class                net
Design_A   in       string    base_name                   in
Design_A   in       string    full_name                   in
Design_A   in       boolean   is_supply                   false
Design_A   in       boolean   is_virtual_supply           false
Design_A   in       boolean   is_vdd                      false
Design_A   in       boolean   is_virtual_vdd              false
Design_A   in       boolean   is_ground                   false
Design_A   in       boolean   is_virtual_ground           false
Design_A   in       boolean   has_detailed_parasitics     false
Design_A   in       boolean   rc_annotated_segment        false
Design_A   in       string    logic_state                 X
Design_A   in       boolean   has_waveform                false
Design_A   in       boolean   is_nonlinear                false
Design_A   in       string    nonlinear_mode              fast
Design_A   in       float     netlist_capacitance         0.000000
Design_A   in       float     total_capacitance_fall_max  0.003869
```

Reading Attribute Values

To get a single specific attribute for a specific object, use the `get_attribute` command with arguments for the object and the attribute, as follows:

```
nt_shell> get_attribute Sn1 total_capacitance_fall_max
0.039744
```

To set a variable to an attribute value obtained from the design, enter

```
nt_shell> set cfm [get_attribute Sn1 total_capacitance_fall_max
Information: Defining new variable 'cfm'. (CMD-041)
0.039744
nt_shell> echo $cfm
0.039744
```

The following command obtains multiple attributes, creates a collection, and iterates through the collection:

```
foreach_in_collection snets [get_nets Sn*] {
  set maxcap [get_attribute $snets \
    total_capacitance_fall_max]
  set netname [get_attribute $snets full_name]
  echo "total cap. fall max of net $netname is $maxcap"
  }
```

The command finds the nets named Sn*, gets the value of the `maxcap` attribute, and reports the net names and corresponding `maxcap` values.

Creating Custom Reports With Attributes

NanoTime generates many different reports during the analysis flow. In addition, you can create your own reports by searching for specific groups of design objects and reporting attributes associated with those objects. This section provides some examples of custom reporting in NanoTime.

You can find some examples of custom timing reports in $SYNOPSYS/auxx/nt/scripts/modeling (where $SYNOPSYS is the installation directory for NanoTime).

Using Paths to Generate Custom Reports

Use the `get_timing_paths` command to create a collection of paths for custom reporting and other processing. You can assign these timing paths to a variable or pass them into another command. For a list of supported attributes on timing paths, see Table 19-21.

Each use of the `get_timing_paths` command has its own context. You cannot compare, add, or remove objects taken from different contexts. For example,

```
nt_shell> set paths1 [get_timing_paths -nworst 10]
...
nt_shell> set paths2 [get_timing_paths -nworst 100]
...
z
```

Even though the variables `paths1` and `paths2` contain some of the same objects, the collections cannot be compared because they come from different contexts. To do such comparisons, create one large collection containing all the objects of interest, and then perform filtering or other manipulation on that collection.

Use the `foreach_in_collection` command to iterate among the paths in the collection. The collection commands `index_collection`, `copy_collection`, `add_to_collection`,

and `remove_from_collection` are not applicable to timing path collections. You can use the `get_attribute` command to obtain information about the paths.

One attribute of a timing path is the points collection. A point corresponds to a pin or port along the path. See Table 19-22 for supported attributes for points of a timing path. Iterate through these points by using the `foreach_in_collection` command and get the attributes on them by using the `get_attribute` command.

The following procedure prints out the startpoint name, endpoint name, and the slack of the worst path in each path group.

```
proc custom_report_max_path_slack {} {
    echo [format "%-20s %-50s %-20s" "From" "To" "Slack"]
    echo
 "----------------------------------------------------------------------"
    foreach_in_collection path [get_timing_paths -max] {
        set slack [get_attribute $path slack]
        set startpoint [get_attribute $path startpoint]
        set endpoint [get_attribute $path endpoint]
        foreach_in_collection sp $startpoint {
          foreach_in_collection obj [get_attribute $sp object] {
            set spo [get_attribute $obj full_name]
          }
        }
        foreach_in_collection ep $endpoint {
          foreach_in_collection obj [get_attribute $ep object] {
            set epo [get_attribute $obj full_name]
          }
        }
      echo [format "%-20s %-50s %-20f" $spo $epo $slack]
    }
}
```

```
nt_shell> custom_report_max_path_slack

From                 To                                Slack
-------------------------------------------------------
clk2                 Xbreg.Xreg50.X5.Mp0.G            -2.728385
sub                  Xbreg.Xreg50.X5.Mp0.G            -2.528385
```

Using Arcs to Generate Custom Reports

To create a collection of path arcs for custom reporting and other processing, use the `arcs` attribute of the `timing_paths` object class. You can assign these timing arcs to a variable and retrieve attributes of each arc for further processing.

Use the `foreach_in_collection` command to iterate through the arcs in the collection. You can use the `get_attribute` command to obtain information about the arcs. However, you cannot copy, sort, or index the collection of arcs.

The following example shows a Tcl procedure to list the channel-connected switching nets in all the arcs of a single path.

```
proc report_switching_nets_of_path { args } {
  set results(-path_id) ""
  parse_proc_arguments -args $args results
  set anum 1
  set item 0
  foreach_in_collection tpath [get_timing_paths -path_id [split
$results(-path_id) ,]] {
    echo "\n==========";
    echo "Path ID [lindex [split $results(-path_id) ,] $item]";
    echo "==========";
    foreach_in_collection arc [get_attribute $tpath arcs] {
      echo "Arc $anum channel-connected switching nets"
      incr anum
      foreach_in_collection snet [get_attribute $arc switching_nets] {
        echo "\t[get_attribute $snet full_name]"   }
    }
    set anum 1
    incr item
  }
}

define_proc_attributes -info "Report all the channel-connected switching
 nets in a path delay arc" -define_args {
  { "-path_id" "Comma-separated list of path IDs; at least one path ID
 required" "{ <id1>, <id2> ... }" string required }
} report_switching_nets_of_path
```

```
nt_shell> report_switching_nets_of_path –path_id 60966
=============
Path ID 60966
=============
Arc 1 channel-connected switching nets
        HOLDA
Arc 2 channel-connected switching nets
  Xareg.hld13
Arc 3 channel-connected switching nets
  Xareg.hld50
```

Attributes of Object Class capture_window_edge

Table 19-2 Attributes of the capture_window_edge Object Class

Attribute name	Type	Description
`input_arrival_time`	float	Input arrival time with respect to the given capture clock domain. The time is defined as being after the capture clock edge.
`model_delay_arc`	collection	A collection containing one `model_delay_arc` object which is the critical timing arc from the specified input port that defines this window edge.
`object_class`	string	The class of the object, a constant equal to `capture_window_edge`.
`output_clock_domain`	collection	Launch clock domain of the last sequential element in the transparent path from the input port to the output port.

Attributes of Object Class cell

Table 19-3 Attributes of the cell Object Class

Attribute name	Type	Description
`base_name`	string	The leaf name of a cell. For example, the `base_name` of cell U1/U2/U3 is U3.
`bidirection_related_to_floating_output`	Boolean	The value is `true` if the transistor is bidirectional because it drives a floating output. Look for corresponding TOPO-055 warning.
`capacitance`	float	The capacitance for the capacitor element.
`channel_connected_transistors`	collection	A collection of transistors that are source/drain channel-connected to the transistor.
`full_name`	string	The complete name of a cell. For example, the full name cell U3 within cell U2 within cell U1 is U1/U2/U3. The `full_name` attribute is not affected by `current_instance`.
`has_parasitic_devices`	Boolean	The value is `true` if the transistor has associated parasitic devices.
`has_transitor_variation`	Boolean	The value is `true` if the cell has variation mapping
`is_capacitor`	Boolean	The value is `true` if the cell is a capacitor.
`is_clock`	Boolean	The value is `true` if the cell is part of the clock network.
`is_clock_gate`	Boolean	The value is `true` if the cell is a gated-clock device.
`is_differential_synchronizer`	Boolean	The value is `true` if the cell is part of a differential synchronizer.
`is_differential_synchronizer_tapped_transistor`	Boolean	The value is `true` if the cell is part of a differential synchronizer tapped transistor.
`is_feedback`	Boolean	The value is `true` if the cell is a feedback device.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`is_hierarchical`	Boolean	The value is `true` if the cell is part of a cell hierarchy.
`is_latch`	Boolean	The value is `true` if the cell is part of a latch structure.
`is_mux`	Boolean	The value is `true` if the cell is a multiplexer.
`is_nmos`	Boolean	The value is `true` if the cell is NMOS.
`is_parallel_non_reference`	Boolean	The value is `true` if the cell belongs to parallel transistors, but is not a parallel reference cell.
`is_parallel_reference`	Boolean	The value is `true` if the cell belongs to parallel transistors and is a parallel reference cell.
`is_pmos`	Boolean	The value is `true` if the cell is PMOS.
`is_precharge`	Boolean	The value is `true` if the cell is part of a precharge structure.
`is_pulldown`	Boolean	The value is `true` if the cell is a pulldown device.
`is_pullup`	Boolean	The value is `true` if the cell is a pullup device.
`is_ram`	Boolean	The value is `true` if the cell is part of a RAM structure.
`is_resistor`	Boolean	The value is `true` if the cell is part of a register file structure.
`is_tgate`	Boolean	The value is `true` if the cell is part of a tgate structure.
`is_timing_model`	Boolean	The value is `true` if the cell is represented by a timing model.
`is_transistor`	Boolean	The value is `true` if the cell is a transistor.
`is_weak_pullup`	Boolean	The value is `true` if the cell is a weak pullup device.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`is_xor`	Boolean	The value is `true` if the cell is part of an XOR structure.
`number_of_pins`	integer	The number of pins on the cell.
`object_class`	string	The class of the object, a constant equal to `cell`.
`parasitic_devices`	collection	A collection of parasitic devices associated with the transistor.
`power_switches`	collection	A collection of power switch topology objects that includes this cell. The cell is a sleeper device.
`ref_name`	string	The name of the design or library cell of which the cell is an instantiation. Also known as the reference name. The linker looks for a design or library cell by this name to resolve the reference.
`resistance`	float	The resistance for the resistor element.
`transistor_ad`	float	The junction area of the drain contact.
`transistor_adej`	float	For FinFET models; the drain junction area (square microns).
`transistor_adeo`	float	For FinFET models; the drain to substrate overlap area (square microns).
`transistor_as`	float	The junction area of the source contact.
`transistor_asej`	float	For FinFET models; the source junction area (square microns).
`transistor_aseo`	float	For FinFET models; the source to substrate overlap area (square microns).
`transistor_bulk_pin`	collection	The transistor bulk name.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_bvs_gain`	float	The gain value of the behavioral voltage source model associated with a transistor. The attribute is used in some device models that include a behavioral voltage source to model drain induced barrier lowering variations.
`transistor_cdsp`	float	For FinFET models; the constant drain to source fringe capacitance.
`transistor_cd_max`	float	The average drain diffusion capacitance calculated by NanoTime for maximum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cd_*` attributes have the same value.
`transistor_cd_max_clock`	float	The average drain diffusion capacitance calculated by NanoTime for maximum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cd_*` attributes have the same value.
`transistor_cd_min`	float	The average drain diffusion capacitance calculated by NanoTime for minimum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cd_*` attributes have the same value.
`transistor_cd_min_clock`	float	The average drain diffusion capacitance calculated by NanoTime for minimum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cd_*` attributes have the same value.
`transistor_cgdp`	float	For FinFET models; the constant gate to drain fringe capacitance.
`transistor_cgsp`	float	For FinFET models; the constant gate to source fringe capacitance.
`transistor_cg_max`	float	The average gate capacitance calculated by NanoTime for maximum data transistors in a multicorner analysis. In a single corner analysis, all the transistor_cg_* attributes have the same value.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_cg_max_clock`	float	The average gate capacitance calculated by NanoTime for maximum clock transistors in a multicorner analysis. In a single corner analysis, all the transistor_cg_* attributes have the same value.
`transistor_cg_min`	float	The average gate capacitance calculated by NanoTime for minimum data transistors in a multicorner analysis. In a single corner analysis, all the transistor_cg_* attributes have the same value.
`transistor_cg_min_clock`	float	The average gate capacitance calculated by NanoTime for minimum clock transistors in a multicorner analysis. In a single corner analysis, all the transistor_cg_* attributes have the same value.
`transistor_cov_max`	float	The average gate to source/drain diffusion overlap capacitance calculated by NanoTime for maximum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cov_*` attributes have the same value.
`transistor_cov_max_clock`	float	The average gate to source/drain diffusion overlap capacitance calculated by NanoTime for maximum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cov_*` attributes have the same value.
`transistor_cov_min`	float	The average gate to source/drain diffusion overlap capacitance calculated by NanoTime for minimum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cov_*` attributes have the same value.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_cov_min_clock`	float	The average gate to source/drain diffusion overlap capacitance calculated by NanoTime for minimum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cov_*` attributes have the same value.
`transistor_cs_max`	float	The average source diffusion capacitance calculated by NanoTime for maximum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cs_*` attributes have the same value.
`transistor_cs_max_clock`	float	The average source diffusion capacitance calculated by NanoTime for maximum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cs_*` attributes have the same value.
`transistor_cs_min`	float	The average source diffusion capacitance calculated by NanoTime for minimum data transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cs_*` attributes have the same value.
`transistor_cs_min_clock`	float	The average source diffusion capacitance calculated by NanoTime for minimum clock transistors in a multicorner analysis. In a single corner analysis, all the `transistor_cs_*` attributes have the same value.
`transistor_d`	float	For FinFET models; the cylinder diameter.
`transistor_delvto`	float	The threshold voltage shift parameter value.
`transistor_delvtrand`	float	For FinFET models; the threshold voltage shift.
`transistor_direction`	string	The transistor direction.
`transistor_dpf_m`	float	The multiplication factor.
`transistor_dpf_nf`	integer	The number of gate fingers for a fingered device.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_drain_pin`	collection	The transistor drain pin name.
`transistor_estimated_width`	float	For FinFET models; a calculated estimated width based on the number of gate fingers, the number of fins per finger, and the fin thickness and height.
`transistor_factuo`	string	The transistor's Penn State Philips model zero-field mobility prefactor.
`transistor_fpitch`	float	For FinFET models; the fin pitch (microns).
`transistor_gate_pin`	collection	The transistor gate pin name.
`transistor_ids0mult`	float	For FinFET models; the source-drain channel current multiplier.
`transistor_length`	float	The transistor length.
`transistor_lrsd`	float	For FinFET models; the length of the source/drain region (microns).
`transistor_model_name`	string	The transistor model name.
`transistor_mulid0`	float	The degradation or enhancement parameter value of a fingered device.
`transistor_mulu0`	float	The low-field mobility multiplier.
`transistor_netlist_m`	float	The multiplication factor. This value is reported based on the netlist parameter.
`transistor_netlist_nf`	integer	The number of gate fingers. This value is reported based on the netlist parameter.
`transistor_nfin`	float	For FinFET models; the number of fins per finger.
`transistor_ngcon`	float	For FinFET models; the number of gate contacts.
`transistor_nrd`	float	The number of drain squares.
`transistor_nrs`	float	The number of source squares.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_pd`	float	The junction periphery of the drain contact.
`transistor_pdej`	float	For FinFET models; the drain junction perimeter (microns).
`transistor_pdeo`	float	For FinFET models; the perimeter of the drain to substrate overlap region (microns).
`transistor_ps`	float	The junction periphery of the source contact.
`transistor_psej`	float	For FinFET models; the source junction perimeter (microns).
`transistor_pseo`	float	For FinFET models; the perimeter of the source to substrate overlap region (microns).
`transistor_sa1` through `transistor_sa10`	float	The distance from the source/drain diffusion edge to the poly gate edge on one side of the transistor. For an irregularly shaped transistor, up to 10 SA parameters can be assigned.
`transistor_sb1` through `transistor_sb10`	float	The distance from the source/drain diffusion edge to the poly gate edge on the other side of the transistor. For an irregularly shaped transistor, up to 10 SB parameters can be assigned.
`transistor_sc`	float	The distance to the single well edge.
`transistor_sca`	float	The integral of the first distribution function for scattered well dopant.
`transistor_scb`	float	The integral of the second distribution function for scattered well dopant.
`transistor_scc`	float	The integral of the third distribution function for scattered well dopant.
`transistor_source_pin`	collection	The transistor source pin name.
`transistor_stimod`	integer	The STI model selector parameter.

Table 19-3 Attributes of the cell Object Class (Continued)

Attribute name	Type	Description
`transistor_sw1` through `transistor_sw10`	float	The distance from one source/drain diffusion edge to the other source/drain diffusion edge for the transistor. For an irregularly shaped transistor, up to 10 SW parameters can be reported.
`transistor_tfin`	float	For FinFET models; the thickness of an individual fin (microns).
`transistor_u0mult`	float	For FinFET models; the mobility multiplier.
`transistor_width`	float	The transistor width.
`transistor_xl`	float	For FinFET models; the L offset for channel length due to mask or etch effects.
`user_defined_transistor_direction`	Boolean	The value is `true` if the transistor has a user-defined transistor direction.

Attributes of Object Class clock

Table 19-4 Attributes of the clock Object Class

Attribute name	Type	Description
`clock_latency_fall_max`	float	The maximum fall latency (insertion delay) for a clock. Set with the `set_clock_latency` command.
`clock_latency_fall_min`	float	The minimum fall latency (insertion delay) for a clock. Set with the `set_clock_latency` command.
`clock_latency_rise_max`	float	The maximum rise latency (insertion delay) for a clock. Set with the `set_clock_latency` command.
`clock_latency_rise_min`	float	The minimum rise latency (insertion delay) for a clock. Set with the `set_clock_latency` command.
`clock_source_latency_early_fall_max`	float	The maximum early fall source latency. Set with the `set_clock_latency` command.
`clock_source_latency_early_fall_min`	float	The minimum early fall source latency. Set with the `set_clock_latency` command.
`clock_source_latency_early_rise_max`	float	The maximum early rise source latency. Set with the `set_clock_latency` command.
`clock_source_latency_early_rise_min`	float	The minimum early rise source latency. Set with the `set_clock_latency` command.
`clock_source_latency_late_fall_max`	float	The maximum late fall source latency. Set with the `set_clock_latency` command.
`clock_source_latency_late_fall_min`	float	The minimum late fall source latency. Set with the `set_clock_latency` command.

Table 19-4 Attributes of the clock Object Class (Continued)

Attribute name	Type	Description
`clock_source_latency_late_rise_max`	float	The maximum late rise source latency. Set with the `set_clock_latency` command.
`clock_source_latency_late_rise_min`	float	The minimum late rise source latency. Set with the `set_clock_latency` command.
`divide_factor`	integer	The frequency division factor.
`duty_cycle`	float	The duty cycle of the clock waveform in percent.
`edge_shifts`	string	A list of three floating point numbers that describes the amount by which each edge in the generated clock waveform is shifted.
`edges`	string	A list of three integers that describes the generated clock waveform.
`full_name`	string	The name of the clock. It is either the name given with the `-name` option of the `create_clock` command, or the name of the first object to which the clock is attached.
`hold_uncertainty_fall`	float	The falling edge clock uncertainty (skew) of a clock used for hold and other minimum delay timing checks. Set with the `set_clock_uncertainty` command.
`hold_uncertainty_rise`	float	The rising edge clock uncertainty (skew) of a clock used for hold and other minimum delay timing checks. Set with the `set_clock_uncertainty` command.
`hold_uncertainty_same_cycle_fall`	float	The falling edge clock uncertainty (skew) of a clock used for hold and other minimum delay timing checks, for same cycle checking only. Set with the `set_clock_uncertainty` command.

Table 19-4 Attributes of the clock Object Class (Continued)

Attribute name	Type	Description
`hold_uncertainty_same_cycle_rise`	float	The rising edge clock uncertainty (skew) of a clock used for hold and other minimum delay timing checks, for same cycle checking only. Set with the `set_clock_uncertainty` command.
`is_generated`	Boolean	The value is `true` for a generated clock.
`is_inverted`	Boolean	The value is `true` if the generated clock is inverted with respect to the master clock.
`is_pulse`	Boolean	The value is `true` for a pulse clock.
`master_clock`	collection	The master clock that is used to make the generated clock when the master clock is a virtual clock.
`multiply_factor`	integer	The frequency multiplication factor.
`object_class`	string	The class of the object, a constant equal to `clock`.
`period`	float	The clock period or cycle time.
`primordial_waveform`	string	A pair of numbers that describes the clock's first rising and falling edge times in the very first clock cycle.
`propagated_clock`	Boolean	The value is `true` if clock latency is determined by propagating delays from the clock source to destination register clock pins. The value is `false` for ideal clocking.
`pulse_type`	string	The value is `high` for a clock that pulses high and `low` for a clock that pulses low.

Table 19-4 Attributes of the clock Object Class (Continued)

Attribute name	Type	Description
`setup_uncertainty_fall`	float	The falling edge clock uncertainty (skew) of a clock falling edge used for setup and other maximum delay timing checks. Set with the `set_clock_uncertainty` command.
`setup_uncertainty_rise`	float	The rising edge clock uncertainty (skew) of a clock rising edge used for setup and other maximum delay timing checks. Set with the `set_clock_uncertainty` command.
`setup_uncertainty_same_cycle_fall`	float	The falling edge clock uncertainty (skew) of a clock falling edge used for setup and other maximum delay timing checks, for same cycle checks only. Set with the `set_clock_uncertainty` command.
`setup_uncertainty_same_cycle_rise`	float	The rising edge clock uncertainty (skew) of a clock falling edge used for setup and other maximum delay timing checks, for same cycle checks only. Set with the `set_clock_uncertainty` command.
`sources`	collection	A collection of the source pins or ports of the clock.
`waveform`	string	Clock rise and fall time pairs.

Attributes of Object Class design

Table 19-5 Attributes of the design Object Class

Attribute name	Type	Description
`full_name`	string	The name of a design, typically the name specified with the `link_design` command.
`has_parasitics`	Boolean	The value is `true` if the design has parasitics.
`is_current`	Boolean	The value is `true` for the current design.
`is_netlist_valid`	Boolean	The value is `true` if the `link_design` phase is complete.
`is_path_restored`	Boolean	The value is `true` if a saved NanoTime session (saved by using the `save_session` command) has been restored by using the `restore_session` command.
`is_path_valid`	Boolean	The value is `true` if the `trace_paths` phase is complete.
`is_timing_graph_valid`	Boolean	The value is `true` if the `check_design` phase is complete.
`is_topology_valid`	Boolean	The value is `true` if the `check_topology` phase is complete.
`object_class`	string	The class of the object; a constant equal to `design`.

Attributes of Object Class launch_window_edge

Table 19-6 Attributes of the launch_window_edge Object Class

Attribute name	Type	Description
`input_clock_domain`	collection	The capture clock domain of the first sequential element in the transparent path from the input port to the output port.
`model_delay_arc`	collection	A collection containing one `model_delay_arc` object which is the critical timing arc to the specified output port that defines this window edge.
`object_class`	string	The class of the object, a constant equal to `launch_window_edge`.
`output_departure_time`	float	Output arrival time with respect to the given launch clock domain. The time is specified as following the launch clock edge.

Attributes of Object Class lib

Table 19-7 Attributes of the lib Object Class

Attribute name	Type	Description
`full_name`	string	The name of a library. For example, the `full_name` of library tech1 read in from /u/user/lib1.db is tech1. This name can be ambiguous because several libraries of the same name can be read in from different directories.

Attributes of Object Class lib_cell

Table 19-8 Attributes of the lib_cell Object Class

Attribute name	Type	Description
`base_name`	string	The name of a library cell. For example, the base_name of library cell tech1/AN2 is AN2.
`full_name`	string	The fully-qualified name of a library cell. This is the name of the library followed by the library cell name. For example, the `full_name` of library cell AN2 in library tech1 is tech1/AN2.
`function_id`	string	The name of the function that is created by Library Compiler.
`is_timing_model`	Boolean	The value is `true` if the cell or SPICE subcircuit is represented by a timing model instead of transistors.
`mog_func_id`	string	The name of the cell's function that is created by Library Compiler for multiple output gate (MOG) cells.
`number_of_pins`	integer	The number of pins on the library cell.
`object_class`	string	The class of the object, a constant equal to `lib_cell`.
`user_function_class`	string	This attribute is used to size complex cells that would otherwise not be sizable because of the limitations of library tools in generating the `function_id` attribute for those complex cells.

Attributes of Object Class lib_pin

Table 19-9 Attributes of the lib_pin Object Class

Attribute name	Type	Description
`direction`	string	The direction of a pin. The value can be `in`, `out`, `inout`, or `internal`.
`full_name`	string	The fully-qualified name of a library cell pin. This is the name of the library followed by the library cell name followed by a pin name. For example, the `full_name` of pin Z on library cell AN2 in library tech1 is tech1/AN2/Z.
`object_class`	string	The class of the object, a constant equal to `lib_pin`.
`steady_state_resistances`	string	Four linear holding resistance values used to calculate injection noise for above low, above high, below low, and below high noise types.

Attributes of Object Class lib_topology

Table 19-10 Attributes of the lib_topology Object Class

Attribute name	Type	Description
`base_name`	string	The name of the library topology element. For example, for a user-defined topology named muxflop, the attribute has the value of `muxflop`.
`full_name`	string	The full name of the library topology including the topology library name. For example, for a user-defined topology named muxflop residing in a topology library named user_topo_lib, the attribute has the value of user_topo_ lib.muxflop.
`library_base_name`	string	The name of the topology library. For example, for a user-defined topology library named user_topo_lib, the attribute has the value of user_topo_lib.
`library_full_name`	string	The full path of the topology library. For example, for a user-defined topology library named user_topo_lib residing at /USER/TOPODB/PATH, the attribute has the value of /USER/TOPODB/PATH/user_topo_lib.
`matched_count`	integer	The total number of times that the library topology element was matched in the source netlist.
`object_class`	string	The class of the object, a constant equal to `lib_topology`.
`search_enabled`	Boolean	The value is `true` if the library topology element has been enabled for search using the `set_search_enabled` or `create_lib_topology –enable_search` command.

Attributes of Object Class memory

Table 19-11 Attributes of the lib_topology Object Class

Attribute name	Type	Description
`analysis_mode`	string	Memory analysis mode; valid values are `read` and `write`.
`bitlines`	collection of nets	The bitlines of the memory.
`full_name`	string	The name of the memory.
`object_class`	string	The class of the object, a constant equal to `memory`.
`wordlines`	collection of nets	The wordlines of the memory.

Attributes of Object Class model_clock_domain

Table 19-12 Attributes of the model_clock_domain Object Class

Attribute name	Type	Description
`clock`	collection	The clock of the clock domain.
`is_pulse_end`	Boolean	The value is `true` if the clock domain represents the end of a pulse.
`is_pulse_start`	Boolean	The value is `true` if the clock domain represents the start of a pulse.
`object_class`	string	The class of the object, a constant equal to `model_clock_domain`.
`reference_clock_domain`	collection	The reference clock domain of the queried clock domain.
`reference_edge_shift`	float	The time shift from the reference clock domain to the queried clock domain.
`transition_direction`	string	The switching direction of the clock that is part of the clock domain. The value is either `rising` or `falling`.

Attributes of Object Class model_delay_arc

Table 19-13 Attributes of the model_delay_arc Object Class

Attribute name	Type	Description
`clock_cycle_count`	integer	The number of clock cycles for a transparent path.
`delay`	float	The propagation delay from an input port or an internal pin to the specified output port or an internal pin of the design.
`is_unate`	Boolean	The value is `true` if the path is noninverting.
`latch_transparency_begin_clock_domains`	collection	An ordered collection of opening clock domains at transparent latches on the timing path the model delay arc is based on.
`load_coefficient`	float	The effective resistance pulling up to the power supply rail and down to the ground rail, including driver and interconnect resistance.
`model_path_id`	integer	The unique path ID associated with the model arc.
`nominal_load`	float	The load applied at the output port during path search.
`nominal_slope`	float	The slope applied at the input port during path search.
`object_class`	string	The class of the object, a constant equal to `model_delay_arc`.
`output_slope`	float	The slope applied at the output port during path search.
`slope_coefficient`	float	The curve-fit coefficient used in adjusting the delay when the actual slope changes from the nominal slope.
`timing_path`	collection	A collection of paths that define the window edges.

Attributes of Object Class net

Table 19-14 Attributes of the net Object Class

Attribute name	Type	Description
`aggressors`	string	The list of nets acting as aggressors to the victim net for signal integrity analysis.
`base_name`	string	The leaf name of a net. For example, the base name of net i1/i1z1 is i1z1.
`bit_cells`	collection	The RAM topologies of the bitcells of a wordline or bitline net.
`clock_gate_checking`	string	Type of clock gate checking enforced on any clock gates that drive this net. Valid values are `non_strict` and `force_strict`.
`coupling_capacitors`	string	The list of coupling capacitors acting on the victim net for SI analysis. Each entry in the list consists of a pair of coupling capacitor node names and the associated capacitance.
`coupling_capacitors_max`	string	The coupling capacitors from the `coupling_capacitors` attribute after adjustment for maximum global variation.
`coupling_capacitors_min`	string	The coupling capacitors from the `coupling_capacitors` attribute after adjustment for minimum global variation.
`dcs_input_logic_state`	Boolean	The logic state to be used when the net is a side input for a dynamic clock simulation region.
`differential_complement`	string	If the net is part of a differential pair, this attribute contains the name of the other net in the pair.
`differential_override_clear`	collection	The list of gate pins of enable transistors that set the output net of a differential circuit to a logic low state.
`differential_override_preset`	collection	The list of gate pins of enable transistors that set the output net of a differential circuit to a logic high state.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`differential_skew_max`	float	The differential skew value set by the `-skew_max` option with the `set_differential` command.
`differential_skew_min`	float	The differential skew value set by the `-skew_min` option of the `set_differential` command.
`effective_aggressors`	string	The list of effective aggressor nets coupled to the victim net. The number of effective aggressors is determined by the SI crosstalk filtering variables.
`effective_aggressors_coupling_cap`	string	The list of coupling capacitances with one-to-one correspondence to the effective aggressors associated with the victim net.
`effective_aggressors_coupling_cap_max`	string	The coupling capacitors from the `effective_aggressors_coupling_cap` attribute after adjustment for maximum global variation.
`effective_aggressors_coupling_cap_min`	string	The coupling capacitors from the `effective_aggressors_coupling_cap` attribute after adjustment for minimum global variation.
`effective_aggressors_max_fall_delta_delays`	string	The list of delta delay contributions from each effective aggressor when the victim net is falling for maximum SI analysis.
`effective_aggressors_max_fall_transitions`	string	The list of the full rail-to-rail falling transition times of all the effective aggressor associated with the victim net for maximum SI analysis.
`effective_aggressors_max_rise_delta_delays`	string	The list of delta delay contributions from each effective aggressor when the victim net is rising for maximum SI analysis.
`effective_aggressors_max_rise_transitions`	string	The list of the full rail-to-rail rising transition times of all the effective aggressors associated with the victim net for maximum SI analysis.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`effective_aggressors_min_fall_delta_delays`	string	The list of delta delay contributions from each effective aggressor when the victim net is falling for minimum SI analysis.
`effective_aggressors_min_fall_transitions`	string	The list of full rail-to-rail falling transition times of all the effective aggressor associated with the victim net for minimum SI analysis.
`effective_aggressors_min_rise_delta_delays`	string	The list of delta delay contributions from each effective aggressor when the victim net is rising for minimum SI analysis.
`effective_aggressors_min_rise_transitions`	string	The list of full rail-to-rail rising transition time of all the effective aggressor associated with the victim net for minimum SI analysis.
`full_name`	string	The complete name of a net. For example, the full_name of net i1z1 within cell i1 is i1/i1z1. The `full_name` attribute is not affected by the current instance.
`group_name`	string	The name assigned to the group of nets to which the net belongs. Each net can belong to no more than one group.
`has_detailed_parasitics`	Boolean	This attribute is `true` if any part of the net has annotated detailed parasitics, even if only one segment of a net is at different levels of the hierarchy.
`has_waveform`	Boolean	The value is `true` if the net has a nonlinear waveform.
`high_impedance`	Boolean	The value is `true` if the initial conditions on this net have been changed as a result of setting the `timing_high_impedance_pessimism_reduction` variable to `true`.
`is_bitline`	Boolean	The value is `true` if the net is a bitline of a memory.
`is_clock`	Boolean	The value is `true` if the net is part of the clock network.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`is_differential`	Boolean	The value is `true` if the net is part of a differential pair.
`is_differential_cross_coupled`	Boolean	The value is `true` if the net is part of a differential cross-coupled pair.
`is_differential_cross_coupled_clear`	Boolean	The value is `true` if the net is part of a differential cross-coupled pair and the clamped state is low.
`is_differential_cross_coupled_preset`	Boolean	The value is `true` if the net is part of a differential cross-coupled pair and the clamped state is high.
`is_differential_reference`	Boolean	The value is `true` if the net is part of a differential pair and it is the reference object.
`is_end_dcs`	Boolean	The value is `true` if the net is specified as the endpoint of a dynamic clock simulation region with the `mark_clock_network -end_dcs` command.
`is_force_clock_propagation`	Boolean	The value is `true` if the net is a forced or an implied forced clock net.
`is_ground`	Boolean	The value is `true` if the net is a ground net.
`is_latch_input`	Boolean	The value is `true` if the net is a latch-input net.
`is_latch_net`	Boolean	The value is `true` if the net is a latch net.
`is_latch_output`	Boolean	The value is `true` if the net is a latch output net.
`is_memory_precharge_eval`	Boolean	The value is `true` if the net is an eval net of a memory precharge topology.
`is_memory_write`	Boolean	The value is `true` if the net is the output of a memory write circuit topology.
`is_mux`	Boolean	The value is `true` if the net is the output of a multiplexer.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`is_no_pulse`	Boolean	The value is `true` if the net does not have the pulse clock property, as specified with the `mark_clock_network -no_pulse` command.
`is_nonlinear`	Boolean	The value is `true` if the net is included for nonlinear waveform analysis.
`is_power_switch_output_enabled`	Boolean	The value is `true` if the power switch topology that connects the output net to a real supply net is turned on.
`is_precharge`	Boolean	The value is `true` if the net is a precharge net.
`is_precharge_input`	Boolean	The value is `true` if the net is a precharge input.
`is_ram`	Boolean	The value is `true` if the net is a RAM net.
`is_register`	Boolean	The value is `true` if the net is a RAM net in a register file structure.
`is_requires_clock`	Boolean	The value is `true` if the net is a clock net.
`is_sense_amp`	Boolean	The value is `true` if the net is the input of a sense amplifier topology.
`is_stop_clock_propagation`	Boolean	The value is `true` if the net is a stopped or an implied stopped clock net.
`is_storage_node`	Boolean	The value is `true` if the net is part of two channel-connected block (CCB) feedback loops.
`is_supply`	Boolean	The value is `true` if the net is a ground or power supply net.
`is_sync_data`	Boolean	The value is `true` if the net is marked with the `set_dcs_input -synchronized_data` command.
`is_vdd`	Boolean	The value is `true` if the net is a power supply net.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`is_virtual_ground`	Boolean	The value is `true` if the net is a virtual ground net. Virtual ground nets can be specified by using the `set_supply_net –virtual –gnd` command.
`is_virtual_supply`	Boolean	The value is `true` if the net is a virtual power supply net. Virtual power supply nets can be specified by using the `set_supply_net –virtual` command.
`is_virtual_vdd`	Boolean	The value is `true` if the net is a virtual power supply net. Virtual supply nets can be specified by using the `set_supply_net –virtual` command.
`is_wordline`	Boolean	The value is `true` if the net is a wordline of a memory.
`logic_check_is_disabled`	Boolean	The value is `true` for the net if logic checking is disabled for that net.
`logic_state`	string	The logic state value of the net. The valid values are X, 1, and 0.
`max_clock_dependent_voltage`	float	The rail voltage value specified on the input trigger net during maximum clock tracing.
`max_clock_voltage`	float	In a multicorner analysis, this attribute is the voltage specified on the ground or power supply net to calculate maximum delays for the clock path. In a single-corner analysis, this attribute has the same value as other maximum or minimum data or clock voltage attributes.
`max_dependent_voltage`	float	The rail voltage value specified on the input trigger net during maximum data tracing.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`max_voltage`	float	In a multicorner analysis, this attribute is the voltage specified on the ground or power supply net to calculate maximum delays for the data path. In a single-corner analysis, this attribute has the same value as other maximum or minimum data or clock voltage attributes.
`memory_bitline_type`	string	The value is one of the following: `bidirectional`, `read_only`, `write_only`, `single_ended_read`, `single_ended_read_differential_write`.
`min_clock_dependent_voltage`	float	The rail voltage value specified on the input trigger net during minimum clock tracing.
`min_clock_voltage`	float	In a multicorner analysis, this attribute is the voltage specified on the ground or power supply net to calculate minimum delays for the clock path. In a single-corner analysis, this attribute has the same value as other maximum or minimum data or clock voltage attributes.
`min_dependent_voltage`	float	The rail voltage value specified on the input trigger net during minimum data tracing.
`min_voltage`	float	In multicorner analysis, this attribute is the voltage specified on the ground or power supply net to calculate minimum delays for the data path. In single-corner analysis, this attribute is the same value as other maximum or minimum data or clock voltage attributes.
`netlist_capacitance`	float	If the `-disable_lumped_capacitance` option of the `link_design` command is not specified, nets that have purely capacitive loading in the netlist (without any resistors) have their capacitance assigned to the `netlist_capacitance` attribute of the net. This value is copied into the `wire_capacitance_*` attribute for the net. This capacitance can be overridden with the `set_load` command.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`nonlinear_mode`	string	The mode specified with the `set_nonlinear_waveform -mode` command to run nonlinear waveform analysis on the net. The valid modes are `fast`, `detect_only`, `efficient`, and `accurate`.
`number_of_aggressors`	integer	The total number of aggressors on the victim net.
`number_of_coupling_capacitors`	integer	The total number of coupling capacitors on the victim net.
`number_of_effective_aggressors`	integer	The total number of effective aggressors on the victim net.
`object_class`	string	The class of the object, a constant equal to `net`.
`pin_capacitance_fall_max`	float	The total capacitance of all the pins attached to the net during the maximum data propagation for a falling transition on the net.
`pin_capacitance_fall_max_clock`	float	The total capacitance of all the pins attached to the net during maximum clock propagation for a falling transition on the net.
`pin_capacitance_fall_min`	float	The total capacitance of all the pins attached to the net during minimum data propagation for a falling transition on the net.
`pin_capacitance_fall_min_clock`	float	The total capacitance of all the pins attached to the net during minimum clock propagation for a falling transition on the net.
`pin_capacitance_rise_max`	float	The total capacitance of all the pins attached to the net during maximum data propagation for a rising transition on the net.
`pin_capacitance_rise_max_clock`	float	The total capacitance of all the pins attached to the net during maximum clock propagation for a rising transition on the net.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`pin_capacitance_rise_min`	float	The total capacitance of all the pins attached to the net during minimum data propagation for a rising transition on the net.
`pin_capacitance_rise_min_clock`	float	The total capacitance of all the pins attached to the net during minimum clock propagation for a rising transition on the net.
`precharge_type`	string	The type of precharge domino structure associated with the net. Values are `nmos` and `pmos` for NMOS precharge and PMOS predischarge type domino structures, respectively.
`rc_annotated_segment`	Boolean	The value is `true` if the net has annotated RC parasitics.
`rc_network`	string	A string describing the parasitic data that has been back-annotated on the net, including resistor values in ohms and capacitor values in picofarads.
`si_fanout_noise_peak_above_low`	string	The maximum noise voltage bump above the ground voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage bump of 150mV above ground, the attribute value is {out1n 0.15}.
`si_fanout_noise_peak_below_high`	string	The maximum noise voltage dip below the supply voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage dip of 150mV below the supply voltage, the attribute value is {out1n -0.15}.
`si_fanout_noise_time_to_peak_ratio_above_low`	string	The time-to-peak ratio of the maximum noise voltage bump above the ground voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage bump above ground with a time-to-peak ratio of 0.45, the attribute value is {out1n 0.45}.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`si_fanout_noise_time_to_peak_ratio_below_high`	string	The time-to-peak ratio of the maximum noise voltage dip below the power supply voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage dip below the supply voltage with a time-to-peak ratio of 0.45,the attribute value is {out1n 0.45}.
`si_fanout_noise_width_above_low`	string	The width of the maximum noise voltage bump above the ground voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage bump above ground with a width of 0.15 ns, the attribute value is {out1n 0.15}.
`si_fanout_noise_width_below_high`	string	The width of the maximum noise voltage dip below the power supply voltage for each fanout net of the fanin net. For example, if net data1n introduces a voltage dip below the supply voltage with a width of 0.15 ns, the attribute value is {out1n 0.15}.
`si_is_selected`	Boolean	The value is `true` if the net has been selected for signal integrity delay or noise analysis.
`si_max_fall_delta_delay`	float	The delta delay adjustment for SI maximum delay analysis for a falling transition on the net.
`si_max_fall_transition`	float	The transition time between the slew threshold trip points for a falling transition on the net as a result of maximum SI analysis.
`si_max_rise_delta_delay`	float	The delta delay adjustment for SI maximum delay analysis for a rising transition on the net.
`si_max_rise_transition`	float	The transition time between the slew threshold trip points for a rising transition on the net as a result of maximum SI analysis.
`si_min_fall_delta_delay`	float	The delta delay adjustment for SI minimum delay analysis for a falling transition on the net.
`si_min_fall_transition`	float	The transition time between the slew threshold trip points for a falling transition on the net as a result of minimum SI analysis.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`si_min_rise_delta_delay`	float	The delta delay adjustment for SI minimum delay analysis for a rising transition on the net.
`si_min_rise_transition`	float	The transition time between the slew threshold trip points for a rising transition on the net as a result of minimum SI analysis.
`total_capacitance_fall_max`	float	The total pin and wire capacitance on the net during the maximum data propagation for a falling transition on the net.
`total_capacitance_fall_max_clock`	float	The total pin and wire capacitance on the net during the maximum clock propagation for a falling transition on the net.
`total_capacitance_fall_min`	float	The total pin and wire capacitance on the net during the minimum data propagation for a falling transition on the net.
`total_capacitance_fall_min_clock`	float	The total pin and wire capacitance on the net during the minimum clock propagation for a falling transition on the net.
`total_capacitance_rise_max`	float	The total pin and wire capacitance on the net during the maximum data propagation for a rising transition on the net.
`total_capacitance_rise_max_clock`	float	The total pin and wire capacitance on the net during the maximum clock propagation for a rising transition on the net.
`total_capacitance_rise_min`	float	The total pin and wire capacitance on the net during the minimum data propagation for a rising transition on the net.
`total_capacitance_rise_min_clock`	float	The total pin and wire capacitance on the net during the minimum clock propagation for a rising transition on the net.
`total_coupling_capacitance`	float	The total coupling capacitance from all the aggressors associated with the victim net.
`voltage`	float	The voltage if the net is a power or ground supply net.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`wire_capacitance_fall_max`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the maximum data propagation for a falling transition on the net.
`wire_capacitance_fall_max_clock`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the maximum clock propagation for a falling transition on the net.
`wire_capacitance_fall_min`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the minimum data propagation for a falling transition on the net.
`wire_capacitance_fall_min_clock`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the minimum clock propagation for a falling transition on the net.
`wire_capacitance_rise_max`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the maximum data propagation for a rising transition on the net.
`wire_capacitance_rise_max_clock`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the maximum clock propagation for a rising transition on the net.
`wire_capacitance_rise_min`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the minimum data propagation for a rising transition on the net.

Table 19-14 Attributes of the net Object Class (Continued)

Attribute name	Type	Description
`wire_capacitance_rise_min_clock`	float	The total wire capacitance applied to the net via netlist-embedded capacitance, parasitic back-annotation, or the `set_load` command during the minimum clock propagation for a rising transition on the net.

Attributes of Object Class parasitic_device

Table 19-15 Attributes of the parasitic_device Object Class

Attribute name	Type	Description
`ad`	float	The junction area of the drain contact for the parasitic transistor. The fingered transistor is considered a separate physical device.
`adej`	float	For FinFET models, the drain junction area (square microns).
`adeo`	float	For FinFET models, the drain to substrate overlap area (square microns).
`as`	float	The junction area of the source contact for the parasitic transistor. The fingered transistor is considered a separate physical device.
`asej`	float	For FinFET models, the source junction area (square microns).
`aseo`	float	For FinFET models, the source to substrate overlap area (square microns).
`cdsp`	float	For FinFET models, the constant drain to source fringe capacitance.
`cgdp`	float	For FinFET models, the constant gate to drain fringe capacitance.
`cgsp`	float	For FinFET models, the constant gate to source fringe capacitance.
`d`	float	For FinFET models, the cylinder diameter.
`delvto`	float	The threshold voltage shift parameter value for the fingered or nonfingered parasitic device.
`delvtrand`	float	For FinFET models, the threshold voltage shift.
`factuo`	string	The Penn State Philips model zero-field mobility prefactor of the parasitic device.
`fpitch`	float	For FinFET models, the fin pitch (microns).
`full_name`	string	The complete name of the fingered or nonfingered parasitic device.
`ids0mult`	float	For FinFET models, the source-drain channel current multiplier.
`length`	float	The length of the fingered or nonfingered parasitic device.
`lrsd`	float	For FinFET models, the length of the source/drain region.

Table 19-15 Attributes of the parasitic_device Object Class (Continued)

Attribute name	Type	Description
`model_name`	string	The fingered or nonfingered parasitic transistor model name.
`mulid0`	float	The degradation or enhancement parameter of the fingered device for the parasitic transistor.
`mulu0`	float	The low field mobility multiplier for the parasitic transistor.
`nfin`	float	For FinFET models; the number of fins per finger.
`ngcon`	float	For FinFET models; the number of gate contacts.
`nrd`	float	The number of drain squares parameter for the parasitic transistor.
`nrs`	float	The number of source squares parameter for the parasitic transistor.
`object_class`	string	The class of the object, a constant equal to `parasitic_device`.
`pd`	float	The junction periphery of the drain contact for the parasitic transistor.
`pdej`	float	For FinFET models; the drain junction perimeter.
`pdeo`	float	For FinFET models; the perimeter of the drain to substrate overlap region through oxide.
`ps`	float	The junction periphery of the source contact for the parasitic transistor.
`psej`	float	For FinFET models; the source junction perimeter.
`pseo`	float	For FinFET models; the perimeter of the source to substrate overlap region through oxide.
`sa1` through `sa10`	float	The distance from the source/drain diffusion edge to the poly gate edge on one side of the parasitic transistor. For an irregularly shaped transistor, up to 10 SA parameters can be assigned.
`sb1` through `sb10`	float	The distance from the source/drain diffusion edge to the poly gate edge on one side of the parasitic transistor. For an irregularly shaped transistor, up to 10 SB parameters can be assigned.
`sc`	float	The distance to the single well edge for the parasitic transistor.
`sca`	float	The integral of the first distribution function for scattered well dopant for the parasitic transistor.

Table 19-15 Attributes of the parasitic_device Object Class (Continued)

Attribute name	Type	Description
`scb`	float	The integral of the second distribution function for scattered well dopant for the parasitic transistor.
`scc`	float	The integral of the third distribution function for scattered well dopant for the parasitic transistor.
`stimod`	float	The STI model selector parameter for the parasitic transistor.
`sw1` through `sw10`	float	The distance from the source/drain diffusion edge to the other source/drain diffusion edge for the parasitic transistor. For an irregularly shaped transistor, up to 10 SW parameters can be assigned.
`tfin`	float	For FinFET models, the thickness of an individual fin (microns).
`u0mult`	float	For FinFET models; the mobility multiplier.
`width`	float	The width of the fingered or nonfingered parasitic device.
`xl`	float	For FinFET models; the L offset for channel length due to mask or etch effects.

Attributes of Object Class path_arc

Table 19-16 Attributes of the path_arc Object Class

Attribute name	Type	Description
`annotated_delay_delta`	float	The annotated delta delays for this arc (for example, SI delta delay adjustments).
`clock`	collection	If applicable, the clock reference controlling the arc.
`clock_pin`	collection	If applicable, the clock pin controlling the arc.
`clock_pin_close_edge_type`	string	If applicable, the clock pin closing edge type (`r` for rising or `f` for falling).
`clock_pin_close_edge_value`	float	If applicable, the arrival of the closing edge of the clock pin controlling the arc.
`clock_pin_cycle`	integer	If applicable, the clock cycle of the clock pin controlling the arc.
`clock_pin_is_pulse`	Boolean	If applicable, indicates if the clock on the clock pin controlling the arc has a clock pulse property.
`clock_pin_open_edge_type`	string	If applicable, the clock pin opening edge type (`r` for rising or `f` for falling).
`clock_pin_open_edge_value`	float	If applicable, the arrival of the opening edge of the clock pin controlling the arc.
`delay`	float	The delay value calculation of the arc after applying delay coefficients, if any.
`delay_adjustment`	float	The delay adjustment (delay delta) of this arc caused by applying delay coefficients with the `set_delay_coefficients` command. If negative, the delay value is reduced by applying the delay coefficients. If positive, the delay value is increased by applying the delay coefficients.
`from_point`	collection	The trigger or startpoint of the arc, typically a transistor gate pin.
`object_class`	string	The class of the object, a constant equal to `path_arc`.

Table 19-16 Attributes of the path_arc Object Class (Continued)

Attribute name	Type	Description
`simulation_delay`	float	The simulated delay of this arc calculated by NanoTime before applying delay coefficients, if any.
`simulation_transition`	float	The simulated output transition time of this arc calculated by NanoTime before applying transition coefficients, if any.
`switching_nets`	collection	A collection of internal nets along the on-chain path of the stage.
`to_point`	collection	The endpoint of the arc, typically a transistor gate pin.
`total_adjustments`	float	The total value of adjustments done on this arc. These include user and automatic adjustments, for example, specifications made by using the `latch_error_recovery` attribute.
`transistor_counts`	string	The count of NMOS and PMOS transistors used to calculate the delay of path arcs. The first value is the NMOS transistor count and the second value is the PMOS transistor count.
`transition_adjustment`	float	The output transition time adjustment (transition time delta) of this arc caused by applying transition coefficients with the `set_transition_coefficients` command. If negative, the transition time is reduced by applying the transition coefficients. If positive, the transition time is increased by applying the transition coefficients.
`wire_delay`	float	The delay from the driver pin (typically a transistor source/drain pin) to the endpoint of the arc (typically a transistor gate pin).
`wire_delay_pin`	collection	The driver pin used to calculate the wire delay.
`wire_transition`	float	The transition time of the driver pin used to calculate the wire delay.

Attributes of Object Class pin

Table 19-17 Attributes of the pin Object Class

Attribute name	Type	Description
`active_fanin_timing_vertices_fall`	collection	A collection of timing pins that start a timing arc ending on the pin with a falling edge.
`active_fanin_timing_vertices_rise`	collection	A collection of timing pins that start a timing arc ending on the pin with a rising edge.
`active_fanout_timing_vertices_fall`	collection	A collection of timing pins at the end of all timing arcs that start from the falling edge of the pin.
`active_fanout_timing_vertices_rise`	collection	A collection of timing pins at the end of all timing arcs that start from the rising edge of the pin.
`arrival_max_fall`	float	The maximum clock falling arrival time at the clock transistor leaf pin.
`arrival_max_rise`	float	The maximum clock rising arrival time at the clock transistor leaf pin.
`arrival_min_fall`	float	The minimum clock falling arrival time at the clock transistor leaf pin.
`arrival_min_rise`	float	The minimum clock rising arrival time at the clock transistor leaf pin.
`clock_latency_fall_max`	float	The user-specified maximum fall latency, or insertion delay, of a pin in the clock network.
`clock_latency_fall_min`	float	The user-specified minimum fall latency, or insertion delay, of a pin in the clock network.
`clock_latency_rise_max`	float	The user-specified maximum rise latency, or insertion delay, of a pin in the clock network.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`clock_latency_rise_min`	float	The user-specified minimum rise latency, or insertion delay, of a pin in the clock network.
`clock_source_latency_early_fall_max`	float	The maximum early fall source latency.
`clock_source_latency_early_fall_min`	float	The minimum early fall source latency.
`clock_source_latency_early_rise_max`	float	The maximum early rise source latency.
`clock_source_latency_early_rise_min`	float	The minimum early rise source latency.
`clock_source_latency_late_fall_max`	float	The maximum late fall source latency.
`clock_source_latency_late_fall_min`	float	The minimum late fall source latency.
`clock_source_latency_late_rise_max`	float	The maximum late rise source latency.
`clock_source_latency_late_rise_min`	float	The minimum late rise source latency.
`correlated_output_pins`	collection	A list of output pins for a single correlated region, specified by the `-outputs` option of the `mark_correlated_region` command.
`differential_complement`	string	If the pin is part of a differential pair, this attribute contains the name of the other pin in the pair.
`differential_skew_max`	float	The differential skew value set by the `-skew_max` option of the `set_differential` command.
`differential_skew_min`	float	The differential skew value set by the `-skew_min` option of the `set_differential` command.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`driver_model_type_max_fall`	string	The driver model type for the max fall operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`driver_model_type_max_rise`	string	The driver model type for the max rise operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`driver_model_type_min_fall`	string	The driver model type for the min fall operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`driver_model_type_min_rise`	string	The driver model type for the min rise operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`fanin_timing_vertices`	collection	The collection of pins in the fanin cloud of the transistor leaf pin.
`fanout_timing_vertices`	collection	The collection of pins in the fanout cloud of the transistor leaf pin.
`full_name`	string	The complete name of a pin to the top of the hierarchy. For example, the full name of pin Z on cell U2 within cell U1 is U1/U2/Z.
`hold_uncertainty_fall`	float	The clock uncertainty (skew) for falling edges at the specified destination clock pin for calculating hold time slack.
`hold_uncertainty_rise`	float	The clock uncertainty (skew) for rising edges at the specified destination clock pin for calculating hold time slack.
`is_clock`	Boolean	The value is `true` if the pin is on a clock network.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`is_correlated_output`	Boolean	The value is `true` if the pin is specified as a correlated output by the `-outputs` option of the `mark_correlated_region` command.
`is_differential`	Boolean	The value is `true` if the pin is part of a differential pair.
`is_differential_reference`	Boolean	The value is `true` if the pin is part of a differential pair and it is the reference object.
`is_latch_net_pin`	Boolean	The value is `true` if the pin is a latch net pin.
`is_parallel_non_reference`	Boolean	The value is `true` if the pin is a nonreference pin of a fingered device.
`is_parallel_reference`	Boolean	The value is `true` if the pin is the reference pin of a fingered device.
`is_port`	Boolean	The value is `true` if the pin is a primary port.
`is_timing_vertex`	Boolean	The value is `true` if the pin is a vertex on the timing graph. This is usually true for leaf transistor gate pins.
`lib_pin_name`	string	The leaf pin name. For example, the lib_pin_name of pin U2/U1/Z is Z.
`max_fall_arrival`	string	The arrival time for the longest path with a falling transition on a pin.
`max_fall_pin_variation_override`	string	The variation of the timing arc associated with this trigger pin for the maximum delay path and falling transitions, as specified by the `set_pin_variation` command.
`max_fall_transition`	string	The transition time for the longest path with a falling transition on a pin.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`max_rise_arrival`	string	The arrival time for the longest path with a rising transition on a pin.
`max_rise_pin_variation_override`	string	The variation of the timing arc associated with this trigger pin for the maximum delay path and rising transitions, as specified by the `set_pin_variation` command.
`max_rise_transition`	string	The transition time for the longest path with a rising transition on a pin.
`min_fall_arrival`	string	The arrival time for the shortest path with a falling transition on a pin.
`min_fall_pin_variation_override`	string	The variation of the timing arc associated with this trigger pin for the minimum delay path and falling transitions, as specified by the `set_pin_variation` command.
`min_fall_transition`	string	The transition time for the shortest path with a falling transition on a pin.
`min_rise_arrival`	string	The arrival time for the shortest path with a rising transition on a pin.
`min_rise_pin_variation_override`	string	The variation of the timing arc associated with this trigger pin for the minimum delay path and rising transitions, as specified by the `set_pin_variation` command.
`min_rise_transition`	string	The transition time for the shortest path with a rising transition on a pin.
`object_class`	string	The class of the object, a constant equal to `pin`.
`pin_capacitance_fall_max`	float	The pin capacitance for a falling transition during maximum delay datapath tracing.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`pin_capacitance_fall_max_clock`	float	The pin capacitance for a falling transition during maximum delay clock path tracing.
`pin_capacitance_fall_min`	float	The pin capacitance for a falling transition during minimum delay datapath tracing.
`pin_capacitance_fall_min_clock`	float	The pin capacitance for a falling transition during minimum delay clock path tracing.
`pin_capacitance_rise_max`	float	The pin capacitance for a rising transition during maximum delay datapath tracing.
`pin_capacitance_rise_max_clock`	float	The pin capacitance for a rising transition during maximum delay clock path tracing.
`pin_capacitance_rise_min`	float	The pin capacitance for a rising transition during minimum delay datapath tracing.
`pin_capacitance_rise_min_clock`	float	The pin capacitance for a rising transition during minimum delay clock path tracing.
`rc_slew_lower_threshold_pct_fall`	float	The threshold voltage that defines the time measurement endpoint of a falling transition, used for calculating the slew or transition time.
`rc_slew_lower_threshold_pct_rise`	float	The threshold voltage that defines the time measurement startpoint of a rising transition, used for calculating the slew or transition time.
`rc_slew_upper_threshold_pct_fall`	float	The threshold voltage that defines the time measurement startpoint of a falling transition, used for calculating the slew or transition time.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`rc_slew_upper_threshold_pct_rise`	float	The threshold voltage that defines the time measurement endpoint of a rising transition, used for calculating the slew or transition time.
`receiver_model_type_max_fall`	string	The receiver model type for the max fall operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`receiver_model_type_max_rise`	string	The receiver model type for the max rise operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`receiver_model_type_min_fall`	string	The receiver model type for the min fall operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`receiver_model_type_min_rise`	string	The receiver model type for the min rise operating condition. The value is `advanced` for CCS models or `basic` for NLDM models.
`setup_uncertainty_fall`	float	The clock uncertainty (skew) for a falling edge at the specified destination clock pin for calculating setup time slack.
`setup_uncertainty_rise`	float	The clock uncertainty (skew) for a rising edge at the specified destination clock pin for calculating setup time slack.
`si_noise_peak_above_high`	float	The calculated noise bump, in volts, above the VDD rail.
`si_noise_peak_above_low`	float	The calculated noise bump, in volts, above the ground rail.
`si_noise_peak_below_high`	float	The value of the calculated noise dip, in volts, below the VDD rail.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`si_noise_peak_below_low`	float	The value of the calculated noise dip, in volts, below the ground rail.
`si_noise_slack_above_high`	float	The difference in volts between the calculated noise bump above the VDD rail and the allowed noise specified by using the `si_noise_margin_above_high` variable.
`si_noise_slack_above_low`	float	The difference in volts between the calculated noise bump above the GND rail and the allowed noise specified by using the `si_noise_margin_above_low` variable.
`si_noise_slack_below_high`	float	The difference in volts between the calculated noise dip below the VDD rail and the allowed noise specified by using the `si_noise_margin_below_high` variable.
`si_noise_slack_below_low`	float	The difference in volts between the calculated noise dip below the GND rail and the allowed noise specified by using the `si_noise_margin_below_low` variable.
`si_noise_time_to_peak_ratio_above_high`	float	The time-to-peak ratio of the calculated noise bump above the VDD rail.
`si_noise_time_to_peak_ratio_above_low`	float	The time-to-peak-ratio of the calculated noise bump above the ground rail.
`si_noise_time_to_peak_ratio_below_high`	float	The time-to-peak ratio of the calculated noise dip below the VDD rail.
`si_noise_time_to_peak_ratio_below_low`	float	The time-to-peak ratio of the calculated noise dip below the ground rail.

Table 19-17 Attributes of the pin Object Class (Continued)

Attribute name	Type	Description
`si_noise_width_above_high`	float	The width of the calculated noise bump, in lib time units above the VDD rail.
`si_noise_width_above_low`	float	The width of the calculated noise bump, in lib time units above the ground rail.
`si_noise_width_below_high`	float	The width of the calculated noise dip, in lib time units below the VDD rail.
`si_noise_width_below_low`	float	The width of the calculated noise dip, in lib time units below the GND rail.

Attributes of Object Class port

Table 19-18 Attributes of the port Object Class

Attribute name	Type	Description
`clock_latency_fall_max`	float	The user-specified maximum fall latency (insertion delay) for clock networks through the port, set with the `set_clock_latency` command.
`clock_latency_fall_min`	float	The user-specified minimum fall latency (insertion delay) for clock networks through the port, set with the `set_clock_latency` command.
`clock_latency_rise_max`	float	The user-specified maximum rise latency (insertion delay) for clock networks through the port, set with the `set_clock_latency` command.
`clock_latency_rise_min`	float	The user-specified minimum rise latency (insertion delay) for clock networks through the port, set with the `set_clock_latency` command.
`clock_source_latency_early_fall_max`	float	The maximum early fall source latency, set with the `set_clock_latency` command.
`clock_source_latency_early_fall_min`	float	The minimum early fall source latency, set with the `set_clock_latency` command.
`clock_source_latency_early_rise_max`	float	The maximum early rise source latency, set with the `set_clock_latency` command.
`clock_source_latency_early_rise_min`	float	The minimum early rise source latency, set with the `set_clock_latency` command.
`clock_source_latency_late_fall_max`	float	The maximum late fall source latency, set with the `set_clock_latency` command.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`clock_source_latency_late_fall_min`	float	The minimum late fall source latency, set with the `set_clock_latency` command.
`clock_source_latency_late_rise_max`	float	The maximum late rise source latency, set with the `set_clock_latency` command.
`clock_source_latency_late_rise_min`	float	The minimum late rise source latency, set with the `set_clock_latency` command.
`clocks`	collection	A collection of clocks created on the port.
`differential_complement`	string	If the port is part of a differential pair, this attribute contains the name of the other port in the pair.
`differential_skew_max`	float	The differential skew value set by the `-skew_max` option of the `set_differential` command.
`differential_skew_min`	float	The differential skew value set by the `-skew_min` option of the `set_differential` command.
`direction`	string	The direction of a port. Values can be `in`, `out`, `inout`, or `internal`.
`drive_resistance_fall_max`	float	The linear drive resistance for falling delays and maximum conditions associated with an input or inout port.
`drive_resistance_fall_min`	float	The linear drive resistance for falling delays and minimum conditions associated with an input or inout port.
`drive_resistance_rise_max`	float	The linear drive resistance for rising delays and maximum conditions associated with an input or inout port.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`drive_resistance_rise_min`	float	The linear drive resistance for rising delays and minimum conditions associated with an input or inout port.
`fanin_timing_vertices`	collection	A collection of vertices for timing arcs to the port.
`fanout_timing_vertices`	collection	A collection of vertices for timing arcs from the port.
`full_name`	string	The name of the port.
`has_input_delay`	Boolean	This attribute is `true` if the input port has a `set_input_delay` command executed on that port.
`has_output_delay`	Boolean	This attribute is `true` if the input port has a `set_output_delay` command executed on that port.
`hold_uncertainty_fall`	float	The fall clock uncertainty through this port for hold checks on paths set with the `set_clock_uncertainty` command.
`hold_uncertainty_rise`	float	The rise clock uncertainty through this port for hold checks on paths set with the `set_clock_uncertainty` command.
`input_delay_fall_max`	string	The maximum fall delay defined on the input port by the `set_input_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`input_delay_fall_min`	string	The minimum fall delay defined on the input port by the `set_input_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`input_delay_rise_max`	string	The maximum rise delay defined on the input port by the `set_input_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`input_delay_rise_min`	string	The minimum rise delay defined on the input port by the `set_input_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`input_transition_fall_max`	float	The fixed transition time for falling delays, maximum conditions associated with an input or inout port, set with the `set_input_transition` command.
`input_transition_fall_min`	float	The fixed transition time for falling delays and minimum conditions associated with an input or inout port, set with the `set_input_transition` command.
`input_transition_rise_max`	float	The fixed transition time for rising delays and maximum conditions associated with an input or inout port, set with the `set_input_transition` command.
`input_transition_rise_min`	float	The fixed transition time for rising delays and minimum conditions associated with an input or inout port, set with the `set_input_transition` command.
`is_differential`	Boolean	The value is `true` if the port is part of a differential pair.
`is_differential_reference`	Boolean	The value is `true` if the port is part of a differential pair and it is the reference object.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`is_timing_vertex`	Boolean	The value is `true` if the port is a timing vertex or the startpoint or endpoint of a timing arc.
`model_input_ceff_fall_max`	float	The maximum fall ceff value looking in from the input port; this value is defined after the `extract_model` command.
`model_input_ceff_fall_min`	float	The minimum fall ceff value looking in from the input port; this value is defined after the `extract_model` command.
`model_input_ceff_rise_max`	float	The maximum rise ceff value looking in from the input port; this value is defined after the `extract_model` command.
`model_input_ceff_rise_min`	float	The minimum rise ceff value looking in from the input port; this value is defined after the `extract_model` command.
`model_input_internal_capacitance`	float	The lumped capacitance seen looking into the input port, including input net parasitics. but with all pass gates turned off.
`model_input_nominal_slope_fall_max`	float	The maximum fall nominal slope used for critical path detection.
`model_input_nominal_slope_fall_min`	float	The minimum fall nominal slope used for critical path detection.
`model_input_nominal_slope_rise_max`	float	The maximum rise nominal slope used for critical path detection.
`model_input_nominal_slope_rise_min`	float	The minimum rise nominal slope used for critical path detection.
`model_input_slope_fall_max`	float	The maximum fall slope used for critical path detection.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`model_input_slope_fall_min`	float	The minimum fall slope used for critical path detection.
`model_input_slope_rise_max`	float	The maximum rise slope used for critical path detection.
`model_input_slope_rise_min`	float	The minimum rise slope used for critical path detection.
`model_input_transition_indexes_max_fall`	string	The list of maximum fall input transition indexes.
`model_input_transition_indexes_max_rise`	string	The list of maximum rise input transition indexes.
`model_input_transition_indexes_min_fall`	string	The list of minimum fall input transition indexes.
`model_input_transition_indexes_min_rise`	string	The list of minimum rise input transition indexes.
`model_load_indexes_max`	string	A list of maximum capacitance indexes that are used on the output port; this list is defined after the `extract_model` command.
`model_load_indexes_min`	string	A list of minimum capacitance indexes that are used on the output port; this list is defined after the `extract_model` command.
`model_output_internal_capacitance`	float	The internal capacitance of the output port net. This is the driver plus the output port net cap, but without any port pin cap, it is defined after the `extract_model` command.
`model_output_intrinsic_slope_fall_max`	float	The maximum fall slope on the output port with no capacitance index loading on the port; this slope is defined after the `extract_model` command.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`model_output_intrinsic_slope_fall_min`	float	The minimum fall slope on the output port with no capacitance index loading on the port; this slope is defined after the `extract_model` command.
`model_output_intrinsic_slope_rise_max`	float	The maximum rise slope on the output port with no capacitance index loading on the port; this slope is defined after the `extract_model` command.
`model_output_intrinsic_slope_rise_min`	float	The minimum rise slope on the output port with no capacitance index loading on the port; this slope is defined after the `extract_model` command.
`model_output_load_fall_max`	float	The maximum fall pin capacitance on the output port. Set with the `set_load` command.
`model_output_load_fall_min`	float	The minimum fall pin capacitance on the output port. Set with the `set_load` command.
`model_output_load_rise_max`	float	The maximum rise pin capacitance on the output port. Set with the `set_load` command.
`model_output_load_rise_min`	float	The minimum rise pin capacitance on the output port. Set with the `set_load` command.
`model_output_nominal_load_fall_max`	float	The nominal maximum fall pin capacitance on the output port. Set with the `set_load` command.
`model_output_nominal_load_fall_min`	float	The nominal minimum fall pin capacitance on the output port. Set with the `set_load` command.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`model_output_nominal_load_rise_max`	float	The nominal maximum rise pin capacitance on the output port. Set with the `set_load` command.
`model_output_nominal_load_rise_min`	float	The nominal minimum rise pin capacitance on the output port. Set with the `set_load` command.
`model_output_reff_fall_max`	float	The maximum fall reff value of the output port net and driver; this value is defined after the `extract_model` command.
`model_output_reff_fall_min`	float	The minimum fall reff value of the output port net and driver; this value is defined after the `extract_model` command..
`model_output_reff_rise_max`	float	The maximum rise reff value of the output port net and driver; this value is defined after the `extract_model` command.
`model_output_reff_rise_min`	float	The minimum rise reff value of the output port net and driver; this value is defined after the `extract_model` command.
`model_output_reff_slope_fall_max`	float	The maximum fall reff value for output slope of the output port net and driver; this value is defined after the `extract_model` command.
`model_output_reff_slope_fall_min`	float	The minimum fall reff value for output slope of the output port net and driver; this value is defined after the `extract_model` command.
`model_output_reff_slope_rise_max`	float	The maximum rise reff value for output slope of the output port net and driver; this value is defined after the `extract_model` command.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`model_output_reff_slope_rise_min`	float	The minimum rise reff value for output slope of the output port net and driver; this value is defined after the `extract_model` command.
`object_class`	string	The class of the object, a constant equal to `port`.
`output_delay_fall_max`	string	The maximum fall delay defined on the output port by the `set_output_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`output_delay_fall_min`	string	The minimum fall delay defined on the output port by the `set_output_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`output_delay_rise_max`	string	The maximum rise delay defined on the output port by the `set_output_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`output_delay_rise_min`	string	The minimum rise delay defined on the output port by the `set_output_delay` command. The string is a list of up to three elements: the delay value, the clock definition (if used), and options (if any).
`phasing`	string	The phasing applied on the port with the `set_phasing` command.
`pin_capacitance_fall_max`	float	The pin capacitance on the port, set with the `set_load -pin_load` command.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`pin_capacitance_fall_max_clock`	float	The maximum fall clock pin capacitance on the port, set with the `set_load -pin_load` command.
`pin_capacitance_fall_min`	float	The minimum fall pin capacitance on the port, set with the `set_load -pin_load` command.
`pin_capacitance_fall_min_clock`	float	The minimum fall clock pin capacitance on the port, set with the `set_load -pin_load` command.
`pin_capacitance_rise_max`	float	The maximum rise pin capacitance on the port, set with the `set_load -pin_load` command.
`pin_capacitance_rise_max_clock`	float	The maximum rise clock pin capacitance on the port, set with the `set_load –pin_load` command.
`pin_capacitance_rise_min`	float	The minimum rise pin capacitance on the port, set with the `set_load -pin_load` command.
`pin_capacitance_rise_min_clock`	float	The minimum rise clock pin capacitance on the port, set with the `set_load -pin_load` command.
`rc_slew_lower_threshold_pct_fall`	float	The lower percent measurement point for a falling edge waveform at the port.
`rc_slew_lower_threshold_pct_rise`	float	The lower percent measurement point for a rising edge waveform at the port.
`rc_slew_upper_threshold_pct_fall`	float	The upper percent measurement point for a falling edge waveform at the port.
`rc_slew_upper_threshold_pct_rise`	float	The upper percent measurement point for a rising edge waveform at the port.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`setup_uncertainty_fall`	float	The fall clock uncertainty for setup checks on paths through the port set with the `set_clock_uncertainty` command.
`setup_uncertainty_rise`	float	The rise clock uncertainty for setup checks on paths through the port set with the `set_clock_uncertainty` command.
`wire_capacitance_fall_max`	float	The maximum fall net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_fall_max_clock`	float	The maximum fall clock net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_fall_min`	float	The minimum fall net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_fall_min_clock`	float	The minimum fall clock net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_rise_max`	float	The maximum rise net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.

Table 19-18 Attributes of the port Object Class (Continued)

Attribute name	Type	Description
`wire_capacitance_rise_max_clock`	float	The maximum rise clock net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_rise_min`	float	The minimum rise net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.
`wire_capacitance_rise_min_clock`	float	The minimum rise clock net wire capacitance set with the `set_load -wire_load` command on the net the port is attached to, or from netlist capacitors.

Attributes of Object Class simulation

Table 19-19 Attributes of the simulation Object Class

Attribute name	Type	Description
`input_nets`	collection	A collection of nets in the fanin cone of logic of the contiguous channel-connected simulation unit specified for dynamic delay simulation by using the `mark_simulation` command.
`output_nets`	collection	A collection of nets in the fanout cone of logic of the contiguous channel-connected simulation unit specified for dynamic delay simulation by using the `mark_simulation` command.

Attributes of Object Class timing_check

Table 19-20 Attributes of the timing_check Object Class

Attribute name	Type	Description
`check_value`	string	The check value specified for timing checks.
`from_pin`	collection	A collection of the clock pins associated with the timing check. For example, for the path through a latch, the attribute contains the latch clock pins on the device driving the latch net.
`from_transition_direction`	string	The transition direction on the `from_pin` of the timing check. Valid values are `rise` and `fall`.
`is_always_pbsa`	Boolean	The value is `true` when path-based slack adjustment is applied unconditionally by using the `set_timing_check_attributes –always_pbsa` command.
`is_continue`	Boolean	The value is `true` when the `-continue` option is used with `create_timing_check` or `set_timing_check_attributes` commands to modify the default path tracing behavior.
`is_continue_with_error_adjustment`	Boolean	The value is `true` if the timing check allows tracing to continue after a constraint violation.
`is_force_max_max`	Boolean	The value is `true` if both the reference path and checked path of the timing constraint are from maximum path tracing.
`is_force_min_min`	Boolean	The value is `true` if both the reference path and checked path of the timing constraint are from minimum path tracing.
`is_force_no_same_phase`	Boolean	The value is `true` when the `-force_no_same_phase` option is used with the `set_timing_check_attributes` command.

Table 19-20 Attributes of the timing_check Object Class (Continued)

Attribute name	Type	Description
`is_force_same_phase`	Boolean	The value is `true` when the `-force_no_same_phase` option is used with the `set_timing_check_attributes` command.
`is_intrace`	Boolean	The value is `true` if the timing check is evaluated during path trace.
`is_model_defined`	Boolean	The value is `true` if the timing check has a 2D table lookup for the constraint value.
`is_selective_pbsa`	Boolean	The value is `true` if selective path-based slack adjustment is applied using `set_timing_check_attributes –selective_pbsa` command.
`is_stop`	Boolean	The value is `true` if a timing check is overridden with a `–stop` option.
`is_tabular`	Boolean	The value is `true` if a timing check value is based on a lookup table, specified by the `set_timing_check_attributes` command, rather than a fixed check value.
`is_transparency_begin`	Boolean	The value is `true` if the timing check is of a transparency begin type. The transparency begin check is triggered off the rising edge of the clock, such as in the case of a high-enable latch.
`is_transparent`	Boolean	The value is `true` if the timing constraint is used to check for transparent propagation, such as in latch and precharge circuits.
`is_user_defined`	Boolean	The value is `true` if the timing check was manually defined using `create_timing_check` command.
`is_zero_positive_slack`	Boolean	The value is `true` if the timing check allows tracing to continue only on a successful constraint evaluation with zero or positive slack.
`label`	string	The label associated with the timing check.

Table 19-20 Attributes of the timing_check Object Class (Continued)

Attribute name	Type	Description
`margin`	float	The setup or hold margin specified for the timing check.
`object_class`	string	The class of the object, a constant equal to `timing_check`.
`structure_type`	string	The structure type associated with the timing check. For example, for a path through the latch, the `structure_type` attribute returns `latch`.
`to_pin`	collection	A collection of the data pins associated with the timing check. For example, for the path through a latch, the `to_pin` attribute returns the latch data pins on the device driving the latch net.
`to_transition_direction`	string	The transition direction on the `to_pin` of the timing check. Valid values are `rise` and `fall`.
`type`	string	The type of the timing check performed. Valid values are `setup` and `hold`.

Attributes of Object Class timing_path

Table 19-21 Attributes of the timing_path Object Class

Attribute name	Type	Description
`arcs`	collection	A collection of timing arcs along the timing path.
`arrival`	float	The arrival time at the endpoint of the timing path.
`capture_clock_primordial_time`	float	The arrival time of the master clock edge. A master clock defines the capture clock domain for the timing paths ending at a constraint.
`capture_cycle`	int	The capture clock cycle for the timing paths that end at a constraint.
`char_delay_values`	string	The delay values, encoded as a string, for a timing path used for custom model generation.
`char_input_slopes`	string	The input slope indexes, encoded as a string, for a timing path used for custom model generation.
`char_output_loads`	string	The output load indexes, encoded as a string, for a timing path used for custom model generation.
`char_output_slopes`	string	The output slopes, encoded as a string, for a timing path used for custom model generation.
`clock_cycle_number`	integer	The cycle number in which the timing check is performed. This number is 0 if the check is performed at the end of the cycle in which the data is launched.
`clock_pbsa`	float	The total path-based slack adjustment on the clock portion of the timing path.

Table 19-21 Attributes of the timing_path Object Class (Continued)

Attribute name	Type	Description
`clock_uncertainty`	float	The clock uncertainty value that is set between the clock domain and the constraint. This attribute applies only to timing paths that end at a constraint.
`clock_variation`	float	The adjustment, in user time units, made to the clock path when slack is computed during parametric on-chip variation analysis.
`common_net_pbsa`	collection	The clock net that is common on the launch and capture clock paths used during PBSA calculations.
`data_pbsa`	collection	Total path-based slack adjustment on the data portion of the timing path.
`data_variation`	float	The adjustment, in user time units, made to the data path when slack is computed during parametric on-chip variation analysis.
`delay`	float	The total path delay. This is the sum of all the incremental stage delays and the adjustment delays as seen in the `report_paths` command.
`delta_to_checked_pin`	float	The delta delay from the path endpoint to the checked pin of the timing constraint.
`endpoint`	collection	The timing endpoint name of the `timing_path`, for example, U1/U5/par_reg/D. It corresponds to the endpoint in the timing path report header.
`endpoint_capture_domain_edge_type`	string	The edge of the transparency window (closing or opening).
`endpoint_clock`	collection	The name of the clock at the path endpoint.

Table 19-21 Attributes of the timing_path Object Class (Continued)

Attribute name	Type	Description
`endpoint_clock_close_edge_type`	string	The type of clock edge (rise or fall) that closes (latches) the data.
`endpoint_clock_close_edge_value`	float	The value of the closing edge of the endpoint clock.
`endpoint_clock_is_inverted`	Boolean	The value is `true` if the endpoint clock has been inverted.
`endpoint_clock_is_propagated`	Boolean	The value is `true` if the endpoint clock is a propagated clock and `false` if it is an ideal clock.
`endpoint_clock_latency`	float	The latency of the endpoint clock. If the clock is propagated, it is the computed latency (or delay) from the clock source to the endpoint.
`endpoint_clock_pin`	collection	A collection of the clock pins at the endpoint of the timing path against which the actual setup or hold delays are calculated.
`endpoint_hold_time_value`	float	The value of the hold time at the timing endpoint.
`endpoint_is_level_sensitive`	Boolean	The value is `true` if the endpoint is a level-sensitive device, for example, a latch. The value is `false` if the endpoint is edge-triggered.
`endpoint_output_delay_value`	float	The value of the output delay of the timing endpoint.
`endpoint_setup_time_value`	float	The value of the setup time at the timing endpoint.
`endpoint_type`	string	The type of the timing path endpoint, for example output or latch.
`internal_modeling_violation`	Boolean	The value is `true` if the path contains a timing violation.

Table 19-21 Attributes of the timing_path Object Class (Continued)

Attribute name	Type	Description
`internal_modeling_violation_with_ler`	Boolean	The value is `true` if the path contains a timing violation and received a latch error recovery adjustment.
`is_clocked_loop`	Boolean	The value is `true` if the timing path is a transparent path that completes a loop back to the originating latch.
`launch_clock`	collection	A collection of launch clock objects for timing paths that end at a constraint.
`launch_clock_is_inverted`	Boolean	The value is `true` if the timing path launch clock is inverted.
`launch_clock_open_edge_type`	string	The edge direction of the last clock domain for timing paths that end at a constraint. The value is either `rising` or `falling`.
`launch_clock_primordial_time`	float	The arrival time of the master clock edge.
`launch_cycle`	integer	The launch clock cycle for the timing paths that end at a constraint.
`launch_domain_edge_type`	string	The edge direction of the launch domain. The value is either `rising` or `falling`.
`launch_latch_error_recovery`	float	The cumulative value of latch error recovery adjustments. An adjustment is made when the tool traces paths and a setup violation occurs at a transparent latch. A value of 0.0 is returned when no launch error recovery adjustments are made.
`minimum_effective_driving_transistor_width`	float	The smallest effective width among the transistors in the on-chain driving the output for each arc in a timing path.
`num_arcs`	integer	The number of arcs in a timing path.

Table 19-21 Attributes of the timing_path Object Class (Continued)

Attribute name	Type	Description
`object_class`	string	The class of the object, a constant equal to `timing_path`.
`path_id`	integer	The path ID of the timing path.
`path_type`	string	The type of the timing path. Valid values are `max` and `min`.
`points`	collection	A collection of the timing points that comprise a timing path. A single timing path can consist of many timing points.
`prefix_path_id`	string	The ID of the path to the generated clock startpoint.
`reference_timing_path`	collection	The reference timing path for the given checked timing path.
`required`	float	The required time value for the timing path. Corresponds to the data required time of a timing report.
`slack`	float	The slack of the timing path. Negative values represent violations. Corresponds to the slack of a timing report.
`slack_variation`	float	The adjustment, in user time units, made to the slack during parametric on-chip variation analysis.
`startpoint`	collection	The startpoint of the timing path. Corresponds to the startpoint in the header of a timing report.
`startpoint_clock`	string	The startpoint clock name of the timing path.
`startpoint_clock_is_inverted`	Boolean	The value is `true` if the startpoint clock is inverted.
`startpoint_clock_is_propagated`	Boolean	The value is `true` if the startpoint clock is a propagated clock and `false` if it is an ideal clock.

Table 19-21 Attributes of the timing_path Object Class (Continued)

Attribute name	Type	Description
`startpoint_clock_open_edge_type`	string	The type of clock edge (`rise` or `fall`) that launches the data.
`startpoint_clock_open_edge_value`	float	The opening edge of the startpoint clock.
`startpoint_input_delay_value`	float	The startpoint input delay.
`startpoint_is_asynchronous`	Boolean	The value is `true` if the timing path startpoint is an asynchronous port.
`startpoint_is_level_sensitive`	Boolean	The value is `true` if the startpoint is a level-sensitive device, such as a latch. The value is `false` if the startpoint is edge-triggered.
`startpoint_type`	string	The type of timing path startpoint, for example, the input for a path starting from an input port, clock for a path starting from a clock port, and so on.
`timing_check`	collection	A collection of timing checks for the timing path.
`total_latch_error_recovery`	float	The total latch error recovery adjustments of timing paths regardless of clock domains.

Attributes of Object Class timing_point

Table 19-22 Attributes of the timing_point Object Class

Attribute name	Type	Description
`adjustment`	float	The timing adjustment value associated with the timing point object.
`arrival`	float	The arrival time at the timing point. The total arrival time at a timing point is the accumulated delay of all the previous stages including input external delay.
`object`	collection	The object at this point in the timing path. This is the timing graph endpoint for the stage. It is usually an input or output port or the gate pin of a transistor. In a cell-based design, it is a .lib cell pin.
`object_class`	string	The class of the object, a constant equal to `timing_point`.
`rail_voltage`	float	The rail voltage for the associated timing point object.
`rise_fall`	string	Status of the timing point. Values are: `rise` if the timing point is a rising edge delay; `fall` if the timing point is a falling edge delay. The following values occur only for turnoff topologies: `z_rise` if the timing point is transitioning from 0 to high Z; `z_fall` if the timing point is transitioning from 1 to high Z; `rise_from_z` if the timing point is rising from a high Z state; `fall_from_z` if the timing point is falling from a high Z state.
`transition`	float	The transition time of the net associated with the timing point object.
`transition_full_swing`	float	The full rail-to-rail transition time for the net associated with the timing point object.
`type`	string	The node type associated with the timing point object. For example, for an input port, the value is `input`.
`vdd_value`	float	The rail voltage for the associated timing point object.
`wire_delay`	float	The wire delay for the associated timing point object.

Attributes of Object Class topology

Table 19-23 Attributes of the topology Object Class

Attribute name	Type	Description
`bitlines`	collection of nets	The bitlines of a bitcell in a memory topology.
`clock_gate_type`	string	The type of clock-gate structure. Valid values are: `pulse_generator`, `pulse_generator_unresolved`, `pulse_shaper`, `pulse_shaper_unresolved`, `timing_unresolved`, and `timing_resolved`.
`clocks`	collection	A collection of all the clock pins associated with the topology objects such as latch, precharge, and so on.
`controlling_nmos_pin_max`	string	In timing-based reconvergent clock-gate analysis, the gate pin of the NMOS transistor that controls the maximum delay path.
`controlling_nmos_pin_min`	string	In timing-based reconvergent clock-gate analysis, the gate pin of the NMOS transistor that controls the minimum delay path.
`controlling_pmos_pin_max`	string	In timing-based reconvergent clock-gate analysis, the gate pin of the PMOS transistor that controls the maximum delay path.
`controlling_pmos_pin_min`	string	In timing-based reconvergent clock-gate analysis, the gate pin of the PMOS transistor that controls the minimum delay path.
`differential_high_voltage`	float	The high voltage input to a sense amplifier in black-box memory analysis mode, specified with the `mark_sense_amp -diff_high` command.

Table 19-23 Attributes of the topology Object Class (Continued)

Attribute name	Type	Description
`differential_low_voltage`	float	The low voltage input to a sense amplifier in black-box memory analysis mode, specified with the `mark_sense_amp -diff_low` command.
`enable_pins`	collection	A collection of enable or select pins related to a turnoff or mux topology. The collection depends on the topology. • For tristate or turnoff topologies, the enable pins • For a passgate mux, all of the select pins • For a tgate mux, the select pins of the NMOS side
`evaluate_clock_transistors`	collection	A collection of all the automatically recognized or manually marked evaluate clock transistors in a precharge domino topology object.
`evaluate_data_transistors`	collection	A collection of all the automatically recognized or manually marked evaluate data transistors in a precharge domino topology object.
`evaluate_nets`	collection	A collection of all the automatically recognized or manually marked evaluate or precharge nets in a precharge domino topology object.
`evaluate_other_transistors`	collection	A collection of all the automatically recognized or manually marked evaluate other transistors in a precharge domino topology object.
`feed_forward_transistors`	collection	A collection of all the automatically recognized or manually marked feed-forward transistors in a latch topology object.
`feedback_transistors`	collection	A collection of all the automatically recognized or manually marked feedback transistors in a latch topology object.

Table 19-23 Attributes of the topology Object Class (Continued)

Attribute name	Type	Description
`hold_to`	string	The string that corresponds to the location where the hold check is performed in the topology object. Valid strings are `input`, `latch_net`, and `evaluate_net`.
`ignored_pins`	collection	For clock gates of type `pulse_generator` or `pulse_shaper`, some of the pins in the clock gating structure are ignored to prevent extraneous `false` delays. This attribute returns a list of these ignored pins.
`inputs`	collection	A collection of port, pin, or net objects specified as inputs by options such as `-inputs` or by automatic recognition.
`is_combinational`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked combinational element, for example, a nonsequential element.
`is_edge_triggered`	Boolean	The value is `true` if the topology object is a device manually marked with the `set_non_transparent –edge_triggered` command.
`is_footed_precharge`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked footed precharge domino structure.
`is_fullkeeper_precharge`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked full-keeper precharge domino structure.
`is_halfkeeper_precharge`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked half-keeper precharge domino structure.
`is_non_transparent`	Boolean	The value is `true` if the bitcell of a RAM topology is selected for analysis.
`is_selected_bit`	Boolean	The value is `true` if the bitcell of a memory topology is selected for analysis.

Table 19-23 Attributes of the topology Object Class (Continued)

Attribute name	Type	Description
`is_sequential`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked sequential element.
`is_three_state`	Boolean	The value is `true` if the topology object is an automatically recognized or a manually marked tri-state element.
`is_user_defined`	Boolean	The value is `true` if the topology object is a user-defined object. A user-defined object is one that was declared using an appropriate `mark_*` command.
`keeper_data_transistors`	collection	A collection of all the automatically recognized or manually marked keeper data transistors in a precharge domino topology object.
`keeper_other_transistors`	collection	A collection of all the automatically recognized or manually marked keeper other transistors in a precharge domino topology object.
`latch_net`	collection	A collection of latch nets in the latch topology object.
`lib_topology`	collection	The full name of the library topology including the topology library name. For example, for a user-defined topology named “muxflop” residing in topology library named “user_topo_lib”, the lib_topology attribute returns the string `user_topo_lib.muxflop`.
`negative_enable_pins`	collection	A collection of negative enable pins in an automatically recognized or a manually marked clock-gate structure. All the listed pins must be logic 0 to enable the clock on the output net.
`object_class`	string	The class of the object, a constant equal to `topology`.

Table 19-23 Attributes of the topology Object Class (Continued)

Attribute name	Type	Description
`output`	string	The output net name. Valid for clock gate, mux, and inverter topologies.
`outputs`	collection	A collection of port, pin, or net objects specified by the `-outputs` option of the user-specified commands or by automatic recognition.
`phasing`	string	The phasing of the timing checks for the sequential device or output port. The valid values are `same_cycle` and `next_cycle`.
`positive_enable_pins`	collection	A collection of positive enable pins in an automatically recognized or a manually marked clock gate structure. All the listed pins must be logic 1 to enable the clock on the output net.
`precharge_clock_transistors`	collection	A collection of all the automatically recognized or manually marked precharge clock transistors in a precharge domino topology object.
`precharge_data_transistors`	collection	A collection of all the automatically recognized or manually marked precharge data transistors in a precharge domino topology object.
`precharge_other_transistors`	collection	A collection of all the automatically recognized or manually marked precharge other transistors in a precharge domino topology object.
`precharge_type`	string	The type of precharge technology. Valid values are `nmos` for an NMOS precharge domino structure and `pmos` for a PMOS predischarge domino structure.
`setup_to`	string	The location where the setup check is performed in the topology object. Valid strings are `input`, `output`, `latch_net`, and `evaluate_net`.

Table 19-23 Attributes of the topology Object Class (Continued)

Attribute name	Type	Description
`structure_name`	string	The name associated with the topology object, for example `inv_ts`.
`structure_type`	string	The structure type associated with the topology object., for example `latch`.
`transistors`	collection	A collection of transistors that form the tgate structure of the topology object.
`wordlines`	collection of nets	The wordlines of a bitcell in a memory topology.

A

Tcl Command Interface

You can use the power and flexibility of the Tcl command language to write scripts and procedures to perform repetitive or complex tasks.

This appendix contains the following sections:

- Tcl Syntax and NanoTime Commands
- Redirecting and Appending Output
- Command Aliases
- Tcl Scripts
- Command History
- Suppressing Messages
- Variables
- Collections
- Lists
- Flow Control
- Procedures

Tcl Syntax and NanoTime Commands

The NanoTime user interface is based on the Tcl scripting language. Tcl versions up to and including version 8.5 are supported. Using Tcl, you can extend the NanoTime command language by writing reusable procedures. If you need more information about the Tcl language, consult books on the subject in the engineering section of your local bookstore or library.

Tcl has a straightforward syntax. Every Tcl script is viewed as a series of commands, separated by a new-line character or semicolon. Each command consists of a command name and a series of arguments.

NanoTime has two types of commands: application commands and built-in commands. Most built-in commands are intrinsic to Tcl. Their arguments do not necessarily conform to the NanoTime argument syntax. For example, many Tcl commands have options that do not begin with a dash, but have a value argument.

For example, the Tcl `string` command has a `compare` option with the following syntax:

`string compare` *`string1 string2`*

The characters listed in Table A-1 have special meaning for Tcl.

Table A-1 Tcl Special Characters

Character	Meaning
`$`	Dereferences a variable.
`( )`	Used for grouping expressions.
`[ ]`	Denotes a nested command.
`\`	Used for escape quoting.
`""`	Denotes weak quoting. Nested commands and variable substitutions still occur.
`{ }`	Denotes rigid quoting. There are no substitutions.
`;`	Ends a command.
`#`	Begins a comment.

Table A-2 lists some common tasks and the system commands for performing them.

Table A-2 Common Tasks and Their System Commands

To do this	Use this
List the current NanoTime working directory.	`pwd`
Change the NanoTime working directory to a specified directory or, if no directory is specified, to your home directory.	`cd` *`directory`*
List the files specified, or list all files in the working directory if no arguments are specified. Requires an `sh` program in your path.	`ls` *`directory_list`*
Search for a file using `search_path`.	which *`filename`*
Display the current date and time.	`date`
Execute an operating system command. You should always use the `exec` command before using the `sh` command. This Tcl built-in command has some limitations. For example, it doesn't perform any expansion. For more information, see the `exec` man page.	`exec` *`command`*
Execute an operating system command. Unlike `exec`, this command performs file name expansion. Requires an `sh` program in your path.	`sh` *`command`*
Return the value of an environment variable.	`getenv` *`name`*
Set the value of an environment variable. Any changes to environment variables apply only to the current NanoTime process and to any child processes launched by the current NanoTime process.	`setenv` *`name value`*
Display the value of one or all environment variables.	`printenv` *`var_pattern`*

Generally, NanoTime implements all the Tcl built-in commands. However, NanoTime adds semantics to some Tcl built-in commands and imposes restrictions on some elements of the language. Here are the differences:

- The Tcl `rename` command is limited to procedures you have created.
- The Tcl `load` command is not supported.
- You cannot create a command called `unknown`.
- The autoexec feature found in tclsh is not supported. However, autoload is supported.

- The Tcl `source` command has additional options `-echo` and `-verbose`, which are nonstandard to Tcl.
- The `history` command has additional options `-h` and `-r`, nonstandard to Tcl, and the form `history` *n*. For example, `history 5` lists the last five commands.
- The NanoTime command processor accepts words with bus (array) notation (words that have square brackets, such as `abus[0]`), so that Tcl does not try to execute the index as a nested command.

You can nest a command within another command (also known as command substitution) by enclosing the nested command within brackets `[]`.

Redirecting and Appending Output

You can direct the output of a command, procedure, or script to a specified file in two ways. The traditional redirection operators (> and >>) are available. In addition, the `redirect` command provides more functionality.

The following examples illustrate two methods to direct the output from a path tracing report into a file named temp.out.

- Using the redirection operators > and >>
  - To create a new file temp.out and write a path tracing report into the file:

    ```
    report_paths -nworst 3 > temp.out
    ```

  - To append additional content to the file temp.out:

    ```
    report_paths -slack_less_than 5 >> temp.out
    ```

- Using the `redirect` command
  - To create a new file temp.out and write a path tracing report into the file:

    ```
    redirect temp.out {report_paths -nworst 3}
    ```

  - To append additional content to the file temp.out:

    ```
    redirect -append temp.out {report_paths -slack_less_than 5}
    ```

The `redirect` command is more flexible than the redirection operators. The redirection operators > and >> are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

With the `redirect` command, you can redirect multiple commands or an entire script. This example redirects multiple `echo` commands:

```
redirect e.out {
    echo -n "Hello"
    echo "world"
}
```

You can also run the redirect command in the background by using the `-bg` option.

The Tcl built-in `puts` command does not respond to redirection. Use the NanoTime `echo` command, which responds to redirection.

Command Aliases

You can use aliases to create short forms for the commands you commonly use. For example, you can set up an alias to duplicate a command with a long string of options:

```
nt_shell> alias path_rpt2 "report_paths -max -nosplit -show_path_id"
```

After creating the alias, enter it as a command:

```
nt_shell> path_rpt2
```

NanoTime recognizes an alias only when it is the first word in the command line.

An alias definition takes effect immediately but lasts only until you exit the NanoTime session. To save commonly used alias definitions, store them in the .synopsys_nt.setup file.

You cannot use an existing command name as an alias name; however, aliases can refer to other aliases.

Tcl Scripts

You can use the `source` command to execute scripts in NanoTime. A script is a sequence of NanoTime and Tcl commands in a text file. The script file can be in ASCII or gzip-compressed format.

The syntax is as follows:

`nt_shell>` **`source [-echo] [-verbose]`** ***`script_file_name`***

By default, the `source` command executes the specified script without showing the commands or the system response to the commands. The `-echo` option causes each command in the script to be displayed as it is executed. The `-verbose` option causes the system response to each command to be displayed.

Lines in script files that begin with the pound sign (`#`) are comment lines . Inline comments use a semicolon to end the command, followed by the pound sign to begin the comment. For example,

```
# Set the new string
#
set newstr "New"; # This is a comment.
```

By default, the `source` command treats the *`script_file_name`* argument as an absolute file name. To direct NanoTime to look for the file using the search path, set the `sh_source_uses_search_path` variable to `true`. (The default is `false`.)

When you run a script, the `source` command is echoed to the command log file. By default, each command in the script is also written to the log file. To disable command logging, set the `sh_source_logging` variable to `false`. (The default is `true`.)

By default, when a syntax or semantic error occurs during execution of a command in a script, NanoTime stops processing the script. There are two variables that you can use to change the default behavior: `sh_continue_on_error` and `sh_script_stop_severity`.

To force NanoTime to continue processing the script regardless of error conditions, set the `sh_continue_on_error` variable to `true`. This setting is usually not desirable because the remainder of the script might not perform as expected if a command fails.

To direct NanoTime to stop the script only when certain kinds of messages are issued, use the `sh_script_stop_severity` variable, which is set to `none` by default. Set the variable to `E` to make the script stop on any message with error severity. Set it to `W` to make the script stop on any message with warning severity. The `sh_script_stop_severity` variable has no effect if the `sh_continue_on_error` variable is set to `true`.

Command History

The `history` command lists the commands used in a nt_shell session. With no arguments, the `history` command lists the last 20 commands that you entered.

The `history` command is complex and can generate several forms of output. For more information, see the `history` command man page.

The syntax is as follows:

`history [keep` *`count`*`] [-h] [-r] [`*`number_of_entries`*`]`

To review the last 20 commands you entered, enter

```
nt_shell> history
1 source basic.tcl
2 read_db middle.db
.
.
18 current_design middle
19 link
20 history
```

To change the length of the history buffer, use the `keep` option. For example, this command specifies a history of 50 commands:

```
nt_shell> history keep 50
```

To limit the history list to three commands and to display them in reverse order, enter

```
nt_shell> history -r 3
20 history info 3
19 link
18 current_design middle
```

You can also redirect the output of the `history` command to create a command script:

```
nt_shell> redirect my_script {history -h}
```

You can rerun and recall previously entered commands by using the exclamation point (`!`) operator. Table A-3 lists the shortcuts you can use to rerun commands. Place these shortcuts at the beginning of a command line. They cannot be part of a nested command or a script.

Table A-3 Shortcuts for Rerunning Commands

To rerun	Use
The last command	`!!`
The *n*th command from the last	`!-n`
The command numbered *n* (from a history list)	`!n`
The most recent command that started with `text` (`text` must begin with a letter or underscore and can contain numbers)	`!text`

You can modify and rerun the previous command executed using the `^x^y` format. This recalls the last command, replaces all instances of `x` with `y`, and then executes the command. This is different from many operating system shells, which only replace the first instance of `x` with `y`.

Suppressing Messages

NanoTime provides commands for suppressing and redisplaying warning and information messages. You cannot suppress error messages.

Table A-4 summarizes the commands for controlling the output of warning and information messages. You can use these commands within a procedure (for example, to turn off specific warnings). If you suppress a message *n* times, you must unsuppress the message the same number of times to enable its output again. Use these commands carefully. For more information, see the man pages.

Table A-4 Commands for Controlling Message Output

To do this	Use this command
Disable on-screen reporting of one or more messages	`suppress_message`
Enable on-screen reporting of previously disabled messages	`unsuppress_message`
Display the currently suppressed message IDs	`print_suppressed_messages`
Disable on-screen reporting of messages after a specified number of occurrences	`set_message_info`
Get information about the total number of messages generated, the number of occurrences of a specified message, or the limit set on reporting of a message	`get_message_info`
Display summary information about error, warning, and information messages that have occurred or have been limited with the `set_message_info` command	`print_message_info`

Variables

NanoTime uses two types of variables: application variables and user-defined variables. Application variables have predefined names and are used to control the behavior of NanoTime. User-defined variables can be created for a variety of purposes.

The `set` command specifies the value of a variable. NanoTime echoes the value:

```
nt_shell> set search_path ". /usr/synopsys/libraries"
. /usr/synopsys/libraries
nt_shell> set adir "/usr/local/lib"
/usr/local/lib
```

The dollar sign operator returns the value of a variable, which can then be used in another command:

```
nt_shell> set my_path "$adir $search_path"
/usr/local/lib . /usr/synopsys/libraries
```

You can set variables to the result of a command:

```
nt_shell> set x [get_ports *]
```

The unset command removes the variable setting:

```
nt_shell> unset adir
```

The `printvar` command lists variables. It takes as an argument a variable name, which can include wildcard characters. You can use the `-application` or `-user_defined` options to restrict the scope of the command.

For example, to list all variables that end in “path,” enter

```
nt_shell> printvar *path
 ...
library_path         = ". ./../libs"
link_path            = "*"
search_path          = ". ./../designs"
sh_source_uses_search_path = "false"
sh_user_man_path     = ""
```

By default, when you create a new variable by using the `set` command, a message appears:

```
nt_shell> set link_patg “*”

Information: Defining new variable 'link_patg'. (CMD-041)
```

The CMD-041 message indicates that you have created a new variable. This helps you to notice errors when you intend to set a system variable but type it incorrectly. The `sh_new_variable_message`, `sh_new_variable_message_in_proc`, and `sh_new_variable_message_in_script` variables control when NanoTime issues CMD-041 messages.

By default, the CMD-041 message is generated in interactive mode, but not during execution of procedures or scripts (when you are unlikely to be looking for messages). To enable generation of these messages during execution of procedures or scripts, set the variables appropriately. Doing so might cause a significant increase in runtime. For details, see the man pages for the variables.

Collections

NanoTime builds an internal database of the netlist and the attributes applied to the database. This database consists of several classes of objects, such as designs, libraries, ports, cells, nets, pins, and clocks. Most NanoTime commands operate on these objects.

A collection is a group of objects exported to the Tcl user interface. Collections have an internal representation (the objects) and a string representation. The string representation is generally used only for information and error messages.

You can create collections of objects, then apply a set of commands to interact with those collections. Collections can be homogeneous (contain objects of one type) or heterogeneous (contain objects of many types).

The collection commands are divided into three categories:

- Commands that create collections of objects for use by another command
- Commands that manipulate collections
- Commands that query objects for you to view

You can use wildcards and filtering criteria to narrow the focus of a collection. You can store collections in variables for use in setting attributes, or for performing custom reporting.

Creating Collections

The primary commands that create collections have the form `get_*` or `all_*`, where * is a type of object such as nets or cells. To list the available commands, use the `printvar get_*` or `printvar all_*` commands.

For example, the `get_cells` command creates a collection of cells in the design. Enter the following command to find the cells that begin with the letter “o” and reference an FD2 library cell:

```
nt_shell> get_cells "o*" -filter {ref_name == FD2}
{"o_reg1", "o_reg2", "o_reg3", "o_reg4"}
```

The output is not a list. The output is a display of the `get_cells` command result. The collection is not saved in a variable or passed to another command; it is deleted after being displayed.

You can pass a collection to a command either directly or by using a variable. For example, the following command creates a collection of pins, assigns it to the variable named pins, then uses the variable to query the cells connected to those pins:

```
nt_shell> set pins [get_pins o*/CP]
{"o_reg1/CP", "o_reg2/CP"}
nt_shell> get_cells -of_objects $pins
{"o_reg1", "o_reg2"}
```

To view the contents of a collection, use the `query_objects` command. This command searches for and displays objects in the NanoTime database. Do not use the `echo`, `puts`, or `printvar` commands, because they return only the string that represents the collection (a name that serves as a pointer to the collection). The `query_objects` command does not have a meaningful return value; it simply displays the objects found and returns the empty string.

You can control the format of the output with the `query_objects_format` variable. The default is `Legacy` which returns a list of items enclosed in quotation marks and delimited with commas. The alternative is `Tcl`, which eliminates the quotation marks and commas. For more information, see the `query_objects` and `query_objects_format` man pages.

When commands that create collections are issued from the command prompt, they implicitly query the collection. You can control how many objects are displayed from the collection by using the `collection_result_display_limit` variable.

The following commands demonstrate equivalent methods to display the same ports:

```
nt_shell> get_ports in*
{"in0", "in1", "in2"}
nt_shell> query_objects [get_ports in*]
{"in0", "in1", "in2"}
nt_shell> query_objects -class port in*
{"in0", "in1", "in2"}
```

The primary collection commands, such as the `get_cells` and `get_pins` commands, can take arguments that are collections of the same type. This is useful for writing procedures that can take either a pattern or a collection as an argument. For example, you can use syntax such as the following:

```
get_cells [get_cells u1]
```

Each collection created by a command such as `get_nets` has its own context. You cannot add to, remove from, or compare collections from different contexts such as different designs, different libraries, or different sets of paths created by different `get_timing_paths` commands. To compare objects, create one collection containing all the objects of interest, and then sort, filter, or manipulate the objects in that collection.

For example, the following commands create two different sets of paths that cannot be compared:

```
nt_shell> set paths [get_timing_paths ...]
nt_shell> set more_paths [get_timing_paths ...]
```

Even if the two collections represent the same paths, they are different objects, so comparing them reports a mismatch.

A collection is active only as long as it is referenced. Typically, a collection is referenced when a variable is set to the result of a command that creates it, or when it is passed as an argument to a command or a procedure. For example, if you save a collection of ports with

```
nt_shell> set myports [get_ports *]
```

then either of the following two commands deletes the collection referenced by the `myports` variable:

```
nt_shell> unset myports
```

```
nt_shell> set myports "newvalue"
```

Collections are implicitly deleted when the parent of the objects within the collection is deleted. For example, if a collection of ports is owned by a design, the collection is implicitly deleted when the design that owns the ports is deleted. When a collection is implicitly deleted, the variable that referenced the collection still holds a string representation of the collection. However, because the collection is gone, this string value is not useful.

Using Wildcard Characters

Most commands that create collections allow a list of patterns that contain wildcard characters. NanoTime uses some common matching operators, including

- `*` – Matches 0 to n characters
- `?` – Matches one character

Given ports `i0`, `i1`, `in0`, and `in1`, notice the queries in this sequence of commands:

```
nt_shell> get_ports i*
{"i0", "i1", "in0", "in1"}

nt_shell> query_objects [get_ports i?]
{"i0", "i1"}
```

Commands that create explicit collections, such as `get_cells`, allow you to search in the current instance or hierarchically by using the `-hierarchical` option. The rules for different kinds of searches are as follows:

- Using a wildcard pattern alone matches leaf names in the current instance. For example,

  `get_cells i1*`

- Using a wildcard pattern with the `-hierarchical` option matches leaf names at each level of the hierarchy.

  For example, to find all cells in the hierarchy with the leaf name containing `n1`:

  `prompt>` **`get_cells *n1* -hierarchical`**

- When you search for a wildcard pattern that contains the hierarchy separator, NanoTime breaks up the pattern around the hierarchy separator and matches each piece at progressively deeper levels of the hierarchy. For example, to find the cells in `i1` that begin with `i2`, and then return a collection of cells in each `i1/i2*` that has a leaf name of `n1`, enter

  `prompt> get_cells i1/i2*/n1`

If an object you specify has a name that contains a backslash character (`\`) or any wildcard characters (`*` or `?`), it is best to use the `-exact` option to find the object. Using the `-exact` option makes the primary collection creation command (such as the `get_cells` or `get_ports` commands) consider the *patterns* argument to be a list of exact strings rather than a list of patterns with wildcards. For example,

```
get_cells -exact [list {*cell*1}] ;# finds cell named *cell*1
get_cells -exact [list {a\b1}]    ;# finds cell named a\b1
```

Wildcard matching uses the same logic as the Tcl “string match” command. If you do not use the `-exact` option, you must use the backslash character to escape (mark) each wildcard character to be considered as a character and not as a wildcard. For example,

```
get_cells [list {\*cell\*1}]
get_cells [list {a\\b1}]
```

The *patterns* argument to the collection creation command is a list. If you do not supply a well-formed list, additional backslashes are necessary to communicate these special characters.

Filtering Collections

You can filter collections by using the `-filter` option with the primary commands that create collections, or you can use the `filter_collection` command.

Many commands that create collections accept a `-filter` option that specifies a filter expression. A filter expression is a string composed of a series of logical expressions that describe a set of constraints you want to place on a collection.

A subexpression of a filter expression is a comparison of an attribute name (such as area or direction) with a value (such as 43 or input) by means of an operator (such as `==` or `!=`).

The following command gets the cells in `U1` that have an area no greater than 12 or reference a design (or library cell) named `AN2P`, `AO2P`, and so on. The command then assigns the collection to the `my_cells` variable.

```
nt_shell> set my_cells [get_cells "U1/*" \
            -filter {area <= 12 || ref_name =~ "A*P"}]
```

The filter language supports the following logical operators:

- `AND` Logical AND (not case-sensitive)
- `&&` Logical AND
- `OR` Logical OR (not case-sensitive)
- `||` Logical OR

You can group logical expressions with parentheses to enforce order; otherwise, NanoTime evaluates the expressions from left to right.

The filter language supports the following relational operators:

- `==` Equal
- `!=` Not equal
- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to
- `=~` Matches pattern
- `!~` Does not match pattern

The filter language also supports the defined and undefined existence operators. An existence operator determines whether an attribute is defined for an object. For example,

```
sense == setup_clk_rise and defined(sdf_cond)
```

The right side of a relation can consist of a string or a number. You do not need to enclose strings in quotation marks. For example,

```
nt_shell> set iports [get_ports * -filter {direction == in}]
```

If an expression contains characters that are part of the filter language syntax, you must use curly braces to enclose the expression and double quotation marks to enclose string operands.

In the following example, parentheses are part of the filter language, so they are double quoted and the complete expression is grouped in curly braces:

```
nt_shell> set x [filter_collection $ports {full_name =~ "D(*)"}
```

Parsing a filter expression can fail because of

- Syntax problems
- An invalid attribute name
- A type mismatch between an attribute and the value

Here are the basic relational rules:

- String attributes can be compared using any operator.
- Numeric attributes cannot be compared with patterns.
- Boolean attributes can be compared only using `==` and `!=`. The value can be only true or false.

Commands that provide a `-filter` option also provide a `-regexp` option to enable you to change the filtering to full regular expressions. The `filter_collection` command also provides a `-regexp` option to enable full regular expressions.

The pattern match filter operators `=~` and `!~` behave differently in different circumstances. They do one of two types of pattern matching:

- Simple wildcard matching (the default)
- Full regular expression matching

The `filter_collection` command takes a collection and a filter expression as arguments. The result of the `filter_collection` command is a new collection, or an empty string if no objects match the criteria.

These two commands are equivalent:

```
get_cells * -filter "ref_name =~ A*P"
filter_collection [get_cells *] "ref_name =~ A*P"
```

If you derive several collections from one larger collection, using the `filter_collection` command might be more efficient than using the `-filter` option, as shown in this series of commands:

```
nt_shell> set acells [get_cells "*"]
{"u1", "u2", "u3", "u4"}
nt_shell> set ands [filter_collection $acells "ref_name =~ AN*"]
{"u1", "u2"}
nt_shell> set ors [filter_collection $acells "ref_name =~ OR*"]
{"u3", "u4"}
```

Optionally, the `filter_collection` command accepts the `-regexp` option, which changes `=~` and `!~` to perform real regular expression matching.

Using Implicit Collections as Arguments

Many NanoTime commands have arguments that accept a list of patterns. You can use the name of the individual object rather than an explicit collection such as `[get_ports out4]`, especially for singular objects. For example, the `set_input_delay` command looks for ports and pins, in that order. Therefore, the following command is valid:

```
nt_shell> set_input_delay 5.0 [out4 [get_pins U1/*] n*1]
```

In this example, `set_input_delay` first tries to find a port named `out4`, and if it is not found, tries to find a pin named `out4`. The collection created by the `get_pins` command is composed of pins by definition. The `n*1` pattern is matched against ports. If there is no match, it is matched against pins.

Iterating Over the Elements of a Collection

The `foreach_in_collection` command iterates over a collection. You can nest it within other control structures, including another `foreach_in_collection` command. The similar `foreach` command works on lists, not collections.

During each iteration, the iteration variable is set to a collection of exactly one object. Any command that accepts one of the collections also accepts the iteration variable because they are of the same type.

To generate a separate report for each cell, use these commands:

```
nt_shell> foreach_in_collection itr_var [get_cells o*] {
            redirect [format "%s%d" $fbase $ndx]
            report_cell
          }
```

The `foreach_in_collection` command is sensitive to changes in the netlist. If you are iterating over the elements of a collection, and netlist changes cause the collection to be deleted, the iteration terminates. NanoTime displays a message to indicate that the collection is deleted. For this reason, using commands like `swap_cell` within the `foreach_in_collection` command is not advisable. Instead, use the command to create a list of cells that you want to swap out, then use the `swap_cell` command at the end.

Removing From and Adding to a Collection

NanoTime allows you to remove objects from or add objects to a collection by using the `remove_from_collection` and `add_to_collection` commands.

You can remove objects by using the `filter_collection` command or by filtering objects when you initially create the collection. However, there are some tasks such as removing the elements in one collection from another collection that you cannot accomplish with filtering. Use the `remove_from_collection` and `add_to_collection` commands for this purpose.

The arguments to both commands are a collection and a specification of the objects that you want to add or remove. The specification can be a list of names, wildcard strings, or other collections.

The result of either command is a new collection or an empty string if the operation results in zero elements (in the case of the `remove_from_collection` command). The first argument (the base collection) is a read-only argument.

The commands take the form

```
add_to_collection ABC XY_spec [-unique]
remove_from_collection ABC XY_spec
```

where:

- ABC is the base collection. The base collection is copied to the result collection, and objects matching XY_spec are added to or removed from the new collection. The base collection can be the empty collection.
- XY_spec is a list of named objects or collections to add or remove. The object class of each element in this list must be the same as the base collection. If the name matches an existing collection, the collection is used. Otherwise, NanoTime searches for the objects in the database using the object class of the base collection.
- The `-unique` option eliminates duplicate objects from the resulting collection.

When the base collection argument is the empty collection, some special rules apply. If XY_spec is not empty, at least one homogeneous collection must exist in the XY_spec list. The first homogeneous collection in the XY_spec list becomes the base collection and sets the object class for the function.

This command sets the ports variable for collection of all ports except for CLOCK:

```
nt_shell> set ports [remove_from_collection [get_ports "*"] CLOCK]
```

This command results in a collection of all cells in the design except those in the top level of the hierarchy:

```
nt_shell> set lcells [remove_from_collection  \
            [get_cells * -hier] [get_cells *]]
```

You can add objects to a collection with the `add_to_collection` command:

```
nt_shell> set isos [add_to_collection [get_cells i*] [get_cells o*]]
```

Most collections are homogeneous because commands such as `get_ports` and `get_cells` create homogeneous collections. You can create heterogeneous collections with the `add_to_collection` command. For example,

```
nt_shell> set a [get_ports PH*]
{"PH1", "PH2"}
nt_shell> set b [get_pins -regexp {reg(0|1)/CP}]
{"reg0/CP", "reg1/CP"}
nt_shell> query_objects -verbose [add_to_collection $a $b]
{"port:PH1", "port:PH2", "pin:reg0/CP", "pin:reg1/CP"}
```

Collection Utility Commands

NanoTime provides additional commands for manipulating collections. Some collections, such as timing paths, cannot be sorted, indexed, copied, or compared.

- `sort_collection`

  You can sort a collection according to a list of attributes using the `sort_collection` command. Sorts are ascending by default. You can reverse the order using the `-descending` option. In an ascending sort of Boolean attributes, NanoTime first lists the objects with the attribute set to `false`, then the objects with the attribute set to `true`. Objects without the attribute follow the objects with the attribute.

  For example, enter the following command to sort by direction, then by full name.

  ```
  nt_shell> sort_collection [get_ports *] {direction full_name}
    {"in1", "in2", "out1", "out2"}
  ```

- `sizeof_collection`

  The `sizeof_collection` command returns the number of elements in a collection.

For example, to estimate the size of a design, enter

```
nt_shell> sizeof_collection [get_cells * -hier]
```

To determine whether a collection command yielded results, enter

```
nt_shell> if {[sizeof_collection [get_cells U2/U2]] != 0}
```

- `compare_collections`

  The `compare_collections` command compares the contents of two collections, object for object. You can specify that NanoTime compare the objects in order.

  If the objects in both collections are the same, the result is 0 (like the result of string compare). If the objects are different, the result is nonzero.

- `copy_collection`

  The `copy_collection` command duplicates a collection, resulting in a new collection. The base collection remains unchanged. The `copy_collection` command is an efficient mechanism for duplicating an existing collection. However, copying a collection and having multiple references to the same collection are significantly different.

  If you create a collection and save a reference to it in variable `c1`, assigning the value of `c1` to another variable `c2` creates a second reference to the same collection:

```
nt_shell> set c1 [get_cells "U1*"]
   {"U1", "U10"}
nt_shell> set c2 $c1
   {"U1", "U10"}
nt_shell> echo $c1 $c2
   _sel3 _sel3
```

  The previous commands do not copy the collection; only `copy_collection` creates a new collection that is a duplicate of the original.

  This command sequence shows the result of copying a collection:

```
nt_shell> set collection1 [get_cells "U1*"]
   {"U1", "U10"}
nt_shell> set collection2 [copy_collection $collection1]
   {"U1", "U10"}
nt_shell> compare_collections $collection1 $collection2
   0
```

- `index_collection`

  The `index_collection` command creates a collection of one object that is the *n*th object in another collection. Objects in a collection are numbered 0 through *n*-1.

  Although collections that result from commands such as `get_cells` are not really ordered, each has a predictable, repeatable order. The same command executed *n* times (for example, `get_cells *`) creates a collection of cells in the same order.

This example shows how to extract the first object in a collection.

```
nt_shell> set c1 [get_cells {u1 u2}]
   {"u1", "u2"}
nt_shell> index_collection $c1 0
   {"u1"}
```

Lists

Lists are an important part of Tcl; they are used to represent groups of objects. Tcl list elements can consist of strings or other lists. Do not use commas to separate list items.

Table A-5 shows the Tcl commands you can use with lists.

Table A-5 Tcl Commands to Use With Lists

Command	Task
`concat`	Concatenates two lists and returns a new list.
`join`	Joins elements of a list into a string.
`lappend`	Creates a new list by appending elements to a list (modifies the original list).
`lindex`	Returns a specific element from a list; this command returns the indexed element if it is there, or an empty string if it is not there.
`linsert`	Creates a new list by inserting elements into a list (it does not otherwise modify the list).
`list`	Returns a list formed from its arguments.
`llength`	Returns the number of elements in a list.
`lrange`	Extracts elements from a list.
`lreplace`	Replaces a specified range of elements in a list.
`lsearch`	Searches a list for a regular expression.
`lsort`	Sorts a list.
`split`	Splits a string into a list.

This is the preferred method of specifying a list:

```
[list a b c d]
```

The entries `"a b c d"`, `{a b c d}`, and `[list a b c d]` might work equally well. However, the suggested method is more reliable.

If you are attempting to perform command or variable substitution, the form with braces does not work. For example, if variable `a` is set to 5, these commands yield different results:

```
nt_shell> set b {c d $a [list $a z]}
c d $a [list $a z]

nt_shell> set b [list c d $a [list $a z]]
c d 5 {5 z}
```

Enclosing a set of words within curly braces `{}`, as in the first example, is called rigid quoting because variable, command, and backslash substitutions do not occur.

Enclosing a set of words within quotation marks is called weak quoting because variable, command, and backslash substitutions can still occur.

The NanoTime shell allows you to use bus syntax such as `blk[0]` even though square brackets `[]` are part of the language. Although nt_shell can help in some cases, rigidly quoting these strings, as in `{blk[0]}`, is always better.

In the following examples, the value of variable `a` is 5. Both commands are valid but have different results.

```
nt_shell> set s "temp = data[$a]"
temp = data[5]

nt_shell> set s {temp = data[$a]}
temp = data[$a]
```

Mark individual special characters with the backslash (`\`) so that they are interpreted literally. Note that the backslash character itself is very difficult to use from the user interface, so its use in netlist objects is discouraged. (The number of times a backslash needs to be marked depends on whether the option to which it is passed is a string or a list.)

Flow Control

Flow control commands, such as the `if`, `while`, `for`, `foreach`, `break`, `continue`, and `switch` commands, determine the execution order of other commands. You can use any NanoTime command in a flow control command.

Using the if Command

An `if` command has two or more arguments:

- An expression to evaluate
- A script to execute based on the result of the expression

You can extend the `if` command to contain an unlimited number of `elseif` clauses and one `else` clause.

- The `elseif` argument to the `if` command requires two additional arguments: an expression and a script.
- The `else` argument requires only a script.

This example shows the correct way to specify `elseif` and `else` clauses. Notice that the `else` and `elseif` clauses appear on the same line as the closing brace `}`. This syntax is required because a new line indicates a new command. If the `elseif` clause is on a separate line, it is treated as a command, which it is not. The `elseif` clause is an argument to the `if` command.

```
if {$x == 0} {
  echo "Equal"
} elseif {$x > 0} {
  echo "Greater"
} else {
  echo "Less"
}
```

The `switch` command is equivalent to an `if` tree that compares a variable to a number of values. One of a number of scripts is executed, based on the value of the variable:

```
switch $x {
a {incr t1}
b {incr t2}
c {incr t3}
}
```

Tcl supports expressions. However, arithmetic operators are not included as part of the base Tcl language syntax. Instead, use the `expr` command to evaluate expressions. The `expr` command can also evaluate logical and relational expressions.

This construction is correct:

```
set a [expr (12*$p)]
```

This construction is not correct:

```
set a (12 * $p);
```

If you are comparing strings, use `string compare $x $y` rather than `$x==$y`.

Loops

The `while` command is similar to the same construct in the C programming language. The `while` command has two arguments: an expression and a set of commands to perform inside the loop.

This example uses the `while` command to print squared values from 0 to 10:

```
set p 0
while {$p <= 10} {
  echo "$p squared is: [expr $p * $p]"; incr p
}
```

The `for` command is similar to the same construct in the C programming language. The `for` command has four arguments: an initialization instruction, a loop termination expression, an iteration instruction, and a set of loop commands.

This example shows how to write the `while` loop as a `for` loop:

```
for {set p  0} {$p <= 10} {incr p} {
  echo "$p squared is: [expr $p * $p]"
}
```

The `foreach` command is similar to the same construct in the C shell. This command iterates over the elements in a list. The `foreach` command has three arguments: an iteration variable, a list, and a set of commands to perform inside the loop.

You cannot use `foreach` to iterate over a NanoTime collection; attempting to do so deletes the collection.

This statement prints an array:

```
foreach el [lsort [array names a]] {
   echo "a\($el\) = $a($el)"
}
```

This statement searches in the search path for several files, then reports whether the files are directories:

```
foreach f [which {t1 t2 t3}] {
  echo -n "File $f is "
  if { [file isdirectory $f] == 0 } {
    echo -n "NOT "
  }
  echo "a directory"
}
```

The `break` and `continue` commands terminate a loop before the termination condition is reached, as follows:

- The `break` command causes the innermost loop to terminate.
- The `continue` command causes the current iteration of the innermost loop to terminate.

Procedures

To extend the Tcl command language, you can write reusable Tcl procedures. You can write new commands that can use an unlimited number of arguments, and the arguments can contain defaults. You can use a varying number of arguments.

This procedure prints the contents of an array:

```
proc array_print {arr} {
  upvar $arr a
  foreach el [lsort [array names a]] {
    echo "$arr\($el\) = $a($el)"
  }
}
```

NanoTime provides extensions to Tcl procedures that allow you to define the syntax of arguments. With these features, you can write extensions to NanoTime that look like commands, and you can parse the arguments to your procedure.

Keep in mind the following points about procedures:

- Procedures can use any supported NanoTime command or other procedure you define.
- Procedures can be recursive.
- Procedures can contain local variables and reference variables outside their scope.
- Arguments to procedures can be passed by value or by reference.

Creating Tcl Procedures

The `proc` command creates a new Tcl procedure.

You cannot create a procedure using the name of an existing built-in or application command. However, if a procedure with the name you specify exists, the new procedure usually replaces the existing procedure.

When you invoke the new command, NanoTime executes the contents of the body.

The `proc` command returns an empty string. When a procedure is invoked, the return value is the value specified in a return command. If the procedure does not execute an explicit return, the return value is the value of the last command executed in the body of the procedure. If an error occurs while the body of the procedure is running, the procedure returns that error.

When the procedure is invoked, a local variable is created for each formal argument to the procedure. The value of the local variable is the value of the corresponding argument in the invoking command or the default of the argument. Arguments with defaults are not specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that do not have defaults.

Except for one special case, no extra arguments can exist. The special case permits procedures with variable numbers of arguments. If the last formal argument has the name `args`, a call to the procedure can contain more actual arguments than the procedure has formal arguments. In this case, all actual arguments, starting at the one assigned to `args`, are combined into a list (as if you had used the `list` command). The combined value is assigned to the local variable `args`.

The arguments to Tcl procedures are named positional arguments. You can program positional arguments with defaults; you can include optional arguments by using the special argument name `args`.

When the body is executed, variable names typically refer to local variables, which are created automatically when referenced and deleted as the procedure returns. One local variable is automatically created for each argument of the procedure. You can only access global variables by invoking the `global` or the `upvar` command.

The syntax of the `proc` command is

`proc` *`name arguments body`*

The arguments of the command are as follows:

- *`name`* names the new procedure.
- *`arguments`* lists formal arguments for the procedure. Each list element specifies one argument. (The arguments list can be empty.) Each argument specifier is also a list with one or two fields. If there is one field in the specifier, it is the name of the argument. If

there are two fields in the specifier, the first field is the argument name and the second field is its default.

- *body* defines the procedure script.

To set up a default for an argument, make sure the argument is located in a sublist that contains two elements: the name of the argument and the default.

The following procedure reads a favorite library by default, but reads a specific library if its name is given:

```
proc read_lib_f {{lname favorite.db}} {
  read_db $lname
}
```

You can specify a varying number of arguments by using the `args` argument. You can specify that at least some arguments must be passed into a procedure; you can then deal with the remaining arguments as necessary.

For example, to report the square of at least one number, use

```
proc squares {num args} {
  set nlist $num
  append nlist ""
  append nlist $args
  foreach n $nlist {
    echo "Square of $n is [expr $n*$n]"
  }
}
```

This procedure adds two numbers and returns the sum:

```
nt_shell> proc plus {a b} { return [expr $a + $b]}
nt_shell> plus 5 6
11
```

Displaying Tcl Procedures

The `info` command with the `body` argument displays the body (contents) of a procedure.

If you define the procedure with the `hide_body` attribute, you cannot use the `info body` command to view the contents of the procedure.

The syntax is

`info body` *`proc_name`*

The *`proc_name`* argument is the name of the procedure.

This example shows the output of the `info body` command for a simple procedure named plus:

```
nt_shell> proc plus {a b} {return [expr $a + $b]}
nt_shell> info body plus
return [expr $a + $b]
nt_shell>
```

The `info` command with the `args` argument displays the names of the formal parameters of a procedure.

The syntax is

`info args` *`proc_name`*

The *`proc_name`* argument is the name of the procedure.

This example shows the output of the `info args` command for a simple procedure named plus:

```
nt_shell> proc plus {a b} { return [expr $a + $b] }
nt_shell> info args plus
a b
nt_shell>
```

B

Custom Compiler User Interface

The Synopsys Custom Compiler™ tool is a full-custom design environment that unifies design, simulation, layout, physical verification, parasitic analysis, and static timing analysis. Running NanoTime analysis within the Custom Compiler environment enhances productivity with features such as interactive runs and visualization of timing results.

The following sections describe how to run the NanoTime tool within Custom Compiler:

- Starting and Ending a NanoTime Session
- Initializing a NanoTime Run Directory
- Generating a Netlist
- Editing Constraints
- Editing Design Properties
- Initiating a NanoTime Run
- NanoTime Interactive Dialog Box
- Inspecting NanoTime Path Reports
- Detailed Path Reports
- Crossprobing Objects
- Timing Model Generation
- Hierarchical Analysis

Starting and Ending a NanoTime Session

You must have a Custom Compiler license to use the NanoTime interface to the Custom Compiler tool. NanoTime license requirements are the same as in a standalone NanoTime run.

Before starting the Custom Compiler tool, you must install the NanoTime interface. For installation instructions, see the *NanoTime Installation Notes* on SolvNet at the following address:

http://www.synopsys.com/Support/LI/Installation/Pages/default.aspx

To invoke the NanoTime tool within the Custom Compiler tool, perform the following steps:

1. At the operating system prompt, start the Custom Compiler tool.
2. Select the schematic tab.
3. Choose Tools > NanoTime. See Figure B-1.

Figure B-1 The Tools Menu in the Custom Compiler GUI

4. NanoTime appears as one of the choices on the main menu bar as shown in Figure B-2.

Figure B-2 NanoTime in the Custom Compiler Menu Bar

To exit from the NanoTime interface, choose Tools in the Schematic Editor and deselect NanoTime. Alternatively, to exit completely from the Custom Compiler session, click Choose File > Exit in the Custom Compiler Console window.

Initializing a NanoTime Run Directory

You must initialize a NanoTime run directory to associate it with a design. Run directories store netlists, constraints files, design properties, and outputs. You can initialize a design by creating a new run directory or by loading an existing run directory.

Creating a New Run Directory

To initialize a new run directory, perform the following steps:

1. In the design schematic that you want to analyze, choose NanoTime > Initialize.

   The Initialize Directory dialog box appears, as shown in Figure B-3.

Figure B-3 Initialize Directory Dialog Box

2. Confirm that the library, cell, and view information is correct.
3. Select the “New run directory” check box.
4. Select a run directory. You can browse or accept the directory shown.
5. Select a NanoTime command file to use as a template. The selected file is copied to your run directory and used for your NanoTime run. If no template files are defined, the tool creates one in the run directory.
6. (Optional) Specify setup files for each phase. The setup files should contain NanoTime commands to be applied during an interactive run.
7. Click OK.

Loading an Existing Run Directory

To load an existing run directory, perform the following steps:

1. In the design schematic you want to simulate, choose NanoTime > Initialize.
2. Confirm that the library, cell, and view information is correct.
3. Select the Load Run Directory check box, as shown in Figure B-3.
4. Browse to find the initialized run directory for the current design.
5. (Optional) Specify setup files for each phase.
6. Click OK.

Generating a Netlist

Before running NanoTime from the Custom Compiler tool, you must first generate a netlist from the interface. To create a netlist, perform the following steps:

1. In the design schematic that you want to analyze, choose NanoTime > Inputs > Netlist > Run Netlister. See Figure B-4.
2. A netlist file named netlist.final is created in the initialized run directory.
3. View the netlist by choosing NanoTime > Inputs > Netlist > Edit Netlist.

After the netlist is created, all functions in the NanoTime menu are enabled.

Figure B-4 Generating a Netlist in the Custom Compiler GUI

Editing Constraints

The NanoTime tool needs input, output, and clock constraints for timing analysis. The Edit Constraints dialog box, as shown in Figure B-5, allows you to create constraints and a template file for NanoTime timing analysis.

To edit constraints, perform the following steps:

1. In the design schematic that you want to analyze, choose NanoTime > Inputs > Edit Constraints.
2. The table lists NanoTime constraints saved in the pin properties of the Custom Compiler tool.
3. Edit the table with NanoTime commands.
4. Click OK to save the changes to the pin properties of the Custom Compiler tool.
5. Click Export to create the constraints.sdc file in the current run directory.

You can also edit the constraints.sdc file directly to change or add constraints. This file is read by the template command file.

Figure B-5 Edit NanoTime Constraints Dialog Box

	Name	Direction	set_input_delay	set_input_transition	set_output_delay	set_case_analysis	create_clock
1	addr2	input	0	0.16			
2	addr1	input	0	0.16			
3	addr0	input	0	0.16			
4	wen	input	0	0.0016			-period 2.5
5	wz	input				0	
6	vdd	input					
7	vss	input					
8	wdata0	input	0	0.16			
9	rdata0	output			0		
10	rdatab0	output			0		

Editing Design Properties

To get accurate results, NanoTime needs information about the design characteristics. For example, latch and precharge nodes should be marked using NanoTime commands. The Edit Design Properties dialog box (see Figure B-6) enables you to create, edit, and save the design properties.

Figure B-6 Edit Design Properties Dialog Box

To edit the design properties, perform the following steps:

1. Change the view in the same window to the subcircuit in which the properties are saved.
2. Choose NanoTime > Inputs > Edit Design Properties.
3. Confirm that the library, cell, and view information is correct.
4. In the Phase list, choose the NanoTime phase to which the properties are applied.

   In the example shown in Figure B-6, properties are applied to the Pre check_topology phase. Alternatively, you can apply the properties to a different phase by selecting another choice in the drop-down list.
5. Click the Add button for each property. The property names are given automatically. You can enter NanoTime commands.
6. Click OK or Apply to save the changes to the properties of the cell in the Custom Compiler tool.
7. Click Export to create files for each phase. The files are applied in the NanoTime run.

Initiating a NanoTime Run

In the Custom Compiler tool, you can invoke NanoTime analysis in interactive or batch mode. In interactive mode (the default), the dialog box guides you through each phase of timing analysis. In batch mode, the NanoTime tool launches in an xterm window as a separate process.

To invoke the NanoTime run, perform the following steps:

1. Open the design and choose NanoTime > Run.

   The Run NanoTime dialog box appears, as shown in Figure B-7.

Figure B-7 Run NanoTime Dialog Box

2. Confirm that the library, cell, and view information is correct.
3. Select the Run Netlister check box if the schematic has changed since the previous netlist run.
4. To run in batch mode, select the Batch check box.
5. In the Command file box, you can optionally browse to an existing command file.
6. In the Command history file box, browse to a file name if you want to save your history during an interactive run.
7. Click OK.

If you are running in batch mode, an xterm window appears, as shown in Figure B-8.

Figure B-8 Batch NanoTime Run

NanoTime Interactive Dialog Box

The Interactive Run dialog box, as shown in Figure B-9, guides you through the NanoTime flow. The dialog box has three sections: setup file selection, command execution, and a field to specify options for the command that completes the current phase. The phasing command changes as you move through the NanoTime flow.

Figure B-9 Interactive Run Dialog Box

The default setup file populates the dialog box during initialization. You can browse to select a different setup file or edit the loaded setup file by clicking Edit Setup File.

To add a command, click Add. To delete a command, click Delete. Each row in the table must have one command. Otherwise, to list multiline commands, use the backslash (\) continuation character or braces { }.

When you click Execute, NanoTime first sources the setup file. Next, the commands in the table execute from top to bottom. If a command in the list executes successfully, the command appears in green text.

Error conditions are highlighted in red, as shown in Figure B-9. Warnings are highlighted in yellow. You can continue execution after a warning message, but an error message stops the run. Look at the xterm window to see the detailed error message. If errors do not occur, the phase termination command is executed with the user-provided options.

Click Close to exit the NanoTime shell.

Inspecting NanoTime Path Reports

To create path reports that are readable by the Custom Compiler interface, you must execute the `create_cd_path_reports` command after the `trace_paths` or `extract_model` command. The `create_cd_path_reports` command is valid within the Custom Compiler environment, but not in the standalone NanoTime tool.

For example:

```
create_cd_path_reports [get_timing_path -max -max_paths 30] max.path
```

To inspect path reports, perform the following steps:

1. Choose NanoTime > Results > Inspect Path Reports. Browse to select a path report file.

   The NanoTime Path Report window appears, as shown in Figure B-10. The paths are listed in order of slack, beginning with the worst case.

Figure B-10 NanoTime Path Report

2. Click a path in the table.

   The path is highlighted in the Schematic Editor, as shown in Figure B-11. Only one path can be highlighted at a time. To obtain more information about the path, click Details.

Figure B-11 Path Highlighting in the Schematic View

Detailed Path Reports

In the Detailed Path Reports dialog box (shown in Figure B-12), you can see the path delays, nets, transition directions, and net types of each timing point. You can also highlight the nets in the path of interest.

Figure B-12 Detailed Path Reports Dialog Box

The checkboxes in the Show section display optional information.

To annotate information to the Schematic Editor, select the options in the Annotate section. For example, if you select Delay, the incremental delay values are annotated in the Schematic Editor.

When you click Crossprobe, the path is crossprobed between the Schematic Editor and the extracted layout view. To use the crossprobe feature, you must first initialize the parasitics, as shown in Figure B-13. The layout and schematic views of a crossprobed path are shown in Figure B-14.

Figure B-13 Initializing the Parasitics

Figure B-14 Highlighted Path in the Layout and Schematic Views

Crossprobing Objects

Use the Cross Probe Objects dialog box to specify design objects such as nets, instances, pins, and subcircuits to be probed. See Figure B-15.

Figure B-15 Cross Probe Objects Dialog Box

1. In the Netlist object name box, enter the name of the object to be probed.
2. If the object is hierarchical, enter the hierarchy separator.
3. Select the object type. Net, instance (transistors), and pin objects can be crossprobed with extracted layout.
4. Click Probe to highlight the object in the Schematic Editor.

When the extracted layout view is opened and the object type is a net or a transistor instance, the parasitics view (the starrc tab) is crossprobed with the schematic view and the layout view, as shown in Figure B-16.

Figure B-16 Cross Probe Objects Dialog Box

Timing Model Generation

The NanoTime interface in the Custom Compiler tool provides a quick way to generate timing models (.lib and .db files). After completing a NanoTime run, you can select Extract Model as the next NanoTime phase, as shown in Figure B-17.

Figure B-17 Generate a Timing Model

To create a timing model:

1. Confirm that the library, cell, and view information is correct.
2. In the Run directory box, browse to a run directory or use the default directory.
3. In the Spice model box, browse to a SPICE model file or use the default file.
4. In the Parasitic file box, browse to the boundary parasitics file from the NanoTime model generation, if it is available.
5. Click OK.

Hierarchical Analysis

The NanoTime interface in the Custom Compiler tool features hierarchical design analysis, which finds potential problems in the lowest levels of the design.

In hierarchical analysis, designs are assigned levels according to the hierarchy distance from leaf-level transistors. For example, an inverter with two transistors has a hierarchy level of 1. A buffer with two inverters has a hierarchy level of 2. The hierarchical analysis starts at level 1 designs and continues to higher-level designs.

If the NanoTime tool finds errors such as unresolved transistor direction, the hierarchical analysis stops and a dialog box opens, as shown in Figure B-18. After you resolve the problem or choose to skip debugging, the hierarchical analysis continues to the next design.

To start hierarchical analysis:

1. Choose NanoTime > Special Flows > Hierarchical Analysis.

   The NanoTime Hierarchical Analysis dialog box appears.

Figure B-18 NanoTime Hierarchical Analysis Dialog Box

2. Select a SPICE model file.
3. (Optional) Provide clock ports for each cell if the clocks for the cell are known and are different than those specified by the aliases.
4. Click OK.

C

Example latch.lib Model File

This appendix contains selected portions of the latch.lib model file used in the discussion of transparent latch constructs in Chapter 12, “Timing Models for Hierarchical Analysis.”

Example latch.lib Model File

```
library("latch") {
  technology (cmos) ;
  delay_model : table_lookup ;
  library_features ( report_delay_calculation ) ;
  date : "Thu Sep 1 13:29:59 2011" ;
  revision : "NanoTime Version F-2011.06-SP1" ;
  nom_process : 1.000000 ;
  nom_voltage : 1.200000 ;
  nom_temperature : 85.000000 ;
  operating_conditions( "typ"  ) {
      process : 1.000000 ;
      voltage : 1.200000 ;
      temperature : 85.000000 ;
   } /* current design opcond */
  default_operating_conditions : "typ" ;
  voltage_unit : "1V" ;
  time_unit : "1ns" ;
  capacitive_load_unit (1.000000, pf);
  slew_derate_from_library : 1.000000 ;
  slew_lower_threshold_pct_rise : 10.000000 ;
  slew_lower_threshold_pct_fall : 10.000000 ;
  slew_upper_threshold_pct_rise : 90.000000 ;
  slew_upper_threshold_pct_fall : 90.000000 ;
  input_threshold_pct_rise : 50.000000 ;
  input_threshold_pct_fall : 50.000000 ;
  output_threshold_pct_rise : 50.000000 ;
  output_threshold_pct_fall : 50.000000 ;
  k_process_cell_rise : 0.000000;
  k_process_cell_fall : 0.000000;
  k_volt_cell_rise : 0.000000;
  k_volt_cell_fall : 0.000000;
  k_temp_cell_rise : 0.000000;
  k_temp_cell_fall : 0.000000;
  k_process_rise_transition : 0.000000;
  k_process_fall_transition : 0.000000;
  k_volt_rise_transition : 0.000000;
  k_volt_fall_transition : 0.000000;
  k_temp_rise_transition : 0.000000;
  k_temp_fall_transition : 0.000000;
  default_fanout_load : 1.0;
  default_inout_pin_cap : 1.0;
  default_input_pin_cap : 1.0;
  default_output_pin_cap : 0.0;
  current_unit : "1mA";
  pulling_resistance_unit : "1kohm";
  comment : "ETM extracted by NanoTime Version F-2011.06-SP1
             for amd64 -- Sep 01, 2011" ;

  define(min_delay_flag, timing, boolean);
  define(program, library, string);
```

```
  define(original_pin, pin, string);
        /* Other user defined attributes. */
  program : nt_shell;
/* SCALAR table template is built-in */
/* 1-D table template f(in_trans) */
lu_table_template( f_itrans ) {
        variable_1 : input_net_transition;
}
        index_1 (" 0.0000, 1.0000 ");
/* 2-D table template f(in_trans, out_cap) */
lu_table_template( f_itrans_ocap ) {
        variable_1 : input_net_transition;
        variable_2 : total_output_net_capacitance;
        index_1 (" 0.0000, 1.0000 ");
        index_2 (" 0.0000, 1.0000 ");
}
/* 2-D table template f(d_trans, c_trans) */
lu_table_template( f_dtrans_ctrans ) {
        variable_1 : constrained_pin_transition;
        variable_2 : related_pin_transition;
        index_1 (" 0.0000, 1.0000 ");
        index_2 (" 0.0000, 1.0000 ");
}
cell( test ) {
  dont_use : true ;
  dont_touch : true ;
  interface_timing : true;
  timing_model_type : "extracted";
pin("IN") {
        direction : input ;
        max_transition : 0.320000 ;
        capacitance : 0.016291 ;
        rise_capacitance_range (0.016291, 0.016291);
        fall_capacitance_range (0.016291, 0.016291);
} /* end of pin IN */
pin("CLK") {
        direction : input ;
        clock : true ;
        max_transition : 0.320000 ;
        capacitance : 0.016291 ;
        rise_capacitance_range (0.016291, 0.016291);
        fall_capacitance_range (0.016291, 0.016291);
} /* end of pin CLK */
pin("OUT") {
        direction : output ;
        max_capacitance : 0.200000 ;
        min_capacitance : 0.050000 ;
        capacitance : 0.011393 ;
        rise_capacitance_range (0.011393, 0.011393);
        fall_capacitance_range (0.011393, 0.011393);
        timing () {
                min_delay_flag : true ;
                related_pin : "mol_c_OUT_CLK_R_CLK" ;
```

```
                timing_type : rising_edge ;
                /* comment : min mol c->out, path 12, path 11; */
                cell_rise( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.000000, 0.027703, 0.083494",\
                                  "0.000000, 0.027703, 0.083494",\
                                  "0.000000, 0.027703, 0.083494");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.074203, 0.134300, 0.254921",\
                                  "0.074203, 0.134300, 0.254921",\
                                  "0.074203, 0.134300, 0.254921");
                }
                cell_fall( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.001925, 0.020159, 0.057334",\
                                  "0.001782, 0.020017, 0.057191",\
                                  "0.001782, 0.020017, 0.057191");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.046753, 0.080618, 0.151719",\
                                  "0.046753, 0.080618, 0.151719",\
                                  "0.046753, 0.080618, 0.151719");
                }
        } /* end of arc mol_c_OUT_CLK_R_CLK_OUT_redg_min*/
        timing () {
                related_pin : "mol_c_OUT_CLK_R_CLK" ;
                timing_type : rising_edge ;
                /* comment : max mol c->out, path 9, path 10; */
                cell_rise( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.000000, 0.027703, 0.083494",\
                                  "0.000000, 0.027703, 0.083494",\
                                  "0.000000, 0.027703, 0.083494");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.074203, 0.134300, 0.254921",\
                                  "0.074203, 0.134300, 0.254921",\
                                  "0.074203, 0.134300, 0.254921");
                }
                cell_fall( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.001925, 0.020159, 0.057334",\
```

```
                                "0.001782, 0.020017, 0.057191",\
                                "0.001782, 0.020017, 0.057191");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.046753, 0.080618, 0.151719",\
                                "0.046753, 0.080618, 0.151719",\
                                "0.046753, 0.080618, 0.151719");
                }
        } /* end of arc mol_c_OUT_CLK_R_CLK_OUT_redg*/
        timing () {
                related_pin : "mol_d_OUT_CLK_R_CLK" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : mol d->out max launch arc, path 14 (delta),
                             path 16 (delta); */
                cell_rise( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "-0.006496, 0.021207, 0.076998",\
                                "0.007764, 0.035466, 0.091257",\
                                "0.026644, 0.054346, 0.110137");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.074208, 0.134301, 0.254921",\
                                "0.074208, 0.134301, 0.254921",\
                                "0.074208, 0.134301, 0.254921");
                }
                cell_fall( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "-0.022229, -0.003995, 0.033180",\
                                "-0.003811, 0.014423, 0.051597",\
                                "0.020224, 0.038458, 0.075633");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.061393, 0.111393, 0.211393");
                        values ( "0.046770, 0.080630, 0.151723",\
                                "0.046770, 0.080630, 0.151723",\
                                "0.046770, 0.080630, 0.151723");
                }
        } /* end of arc mol_d_OUT_CLK_R_CLK_OUT_una*/
} /* end of pin OUT */
pin("mil_d_IN_CLK_F_CLK") {
        direction : internal ;
        capacitance : 0.000000 ;
        tlatch("mil_c_IN_CLK_F_CLK") {
                edge_type : rising;
                tdisable : false;
```

```
        }
        timing () {
                related_pin : "IN" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : input->mil d; */
                cell_rise( scalar ){
                        values ( "0.000000");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.000000");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
        } /* end of arc IN_mil_d_IN_CLK_F_CLK_una*/
        timing () {
                related_pin : "mil_c_IN_CLK_F_CLK" ;
                timing_type : setup_falling ;
                /* comment : reference path 8, checked path 13,
                             reference path 7, checked path 15; */
                rise_constraint( f_dtrans_ctrans ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.080000, 0.160000, 0.320000");
                        values ( "0.002835, -0.015773, -0.040040",\
                                 "0.017094, -0.001515, -0.025782",\
                                 "0.035973, 0.017364, -0.006903");
                }
                fall_constraint( f_dtrans_ctrans ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.080000, 0.160000, 0.320000");
                        values ( "0.067005, 0.048397, 0.024130",\
                                 "0.085424, 0.066815, 0.042548",\
                                 "0.109459, 0.090850, 0.066583");
                }
        } /* end of arc mil_c_IN_CLK_F_CLK_mil_d_IN_CLK_F_CLK_stupf*/
        timing () {
                related_pin : "mil_c_IN_CLK_F_CLK" ;
                timing_type : hold_falling ;
                /* comment : reference path 3, checked path 17,
                             reference path 4, checked path 18; */
                rise_constraint( f_dtrans_ctrans ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.080000, 0.160000, 0.320000");
```

```
                        values ( "-0.002835, 0.015773, 0.040040",\
                                 "-0.017094, 0.001515, 0.025782",\
                                 "-0.035973, -0.017364, 0.006903");
                }
                fall_constraint( f_dtrans_ctrans ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        index_2 ( "0.080000, 0.160000, 0.320000");
                        values ( "-0.067005, -0.048397, -0.024130",\
                                 "-0.085424, -0.066815, -0.042548",\
                                 "-0.109459, -0.090850, -0.066583");
                }
        } /* end of arc mil_c_IN_CLK_F_CLK_mil_d_IN_CLK_F_CLK_hldf*/
} /* end of pin mil_d_IN_CLK_F_CLK */
pin("mil_c_IN_CLK_F_CLK") {
        direction : internal ;
        clock : true ;
        capacitance : 0.000000 ;
        timing () {
                related_pin : "CLK" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : mil clock feed; */
                cell_rise( scalar ){
                        values ( "0.000000");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.000000");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
        } /* end of arc CLK_mil_c_IN_CLK_F_CLK_una*/
} /* end of pin mil_c_IN_CLK_F_CLK */
pin("mil_q_IN_CLK_F_CLK") {
        direction : internal ;
        capacitance : 0.000000 ;
        timing () {
                related_pin : "mil_d_IN_CLK_F_CLK" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : mil d->q; */
                cell_rise( scalar ){
                        values ( "0.000000");
                }
```

```
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.000000");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
        } /* end of arc mil_d_IN_CLK_F_CLK_mil_q_IN_CLK_F_CLK_una*/
        timing () {
                related_pin : "mil_c_IN_CLK_F_CLK" ;
                timing_type : rising_edge ;
                /* comment : mil c->q; */
                cell_rise( scalar ){
                        values ( "0.000000");
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.000000");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                 "1.000000, 1.000000");
                }
        } /* end of arc mil_c_IN_CLK_F_CLK_mil_q_IN_CLK_F_CLK_una*/
} /* end of pin mil_q_IN_CLK_F_CLK */
pin("mol_d_OUT_CLK_R_CLK") {
        direction : internal ;
        capacitance : 0.000000 ;
        tlatch("mol_c_OUT_CLK_R_CLK") {
                edge_type : rising;
                tdisable : false;
        }
        timing () {
                related_pin : "mol_c_OUT_CLK_R_CLK" ;
                timing_type : setup_falling ;
                /* comment : mol c->d transparency end setup,
                             rise path 14, fall path 16; */
                rise_constraint( scalar ){
```

```
                        values ( "-0.378071");
                }
                fall_constraint( scalar ){
                        values ( "-0.412058");
                }
        } /* end of arc mol_c_OUT_CLK_R_CLK_mol_d_OUT_CLK_R_CLK_stupf*/
        timing () {
                related_pin : "mol_c_OUT_CLK_R_CLK" ;
                timing_type : hold_falling ;
                /* comment : mol c->d transparency end hold, rise path 6,
                             fall path 5; */
                rise_constraint( scalar ){
                        values ( "0.104808");
                }
                fall_constraint( scalar ){
                        values ( "0.082601");
                }
        } /* end of arc
             mol_c_OUT_CLK_R_CLK_mol_d_OUT_CLK_R_CLK_hldf_min*/
        timing () {
                related_pin : "mil_q_IN_CLK_F_CLK" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : IN:CLK:F:CLK -> OUT:CLK:R:CLK connecting max
                             arc, path 14, path 16; */
                cell_rise( scalar ){
                        values ( "0.221111");
                }
                rise_transition( f_itrans ){
                        index_1 ( "0.000000, 1.000000");
                        values ( "0.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.276249");
                }
                fall_transition( f_itrans ){
                        index_1 ( "0.000000, 1.000000");
                        values ( "0.000000, 1.000000");
                }
        } /* end of arc mil_q_IN_CLK_F_CLK_mol_d_OUT_CLK_R_CLK_una*/
} /* end of pin mol_d_OUT_CLK_R_CLK */
pin("mol_c_OUT_CLK_R_CLK") {
        direction : internal ;
        clock : true ;
        capacitance : 0.000000 ;
        timing () {
                related_pin : "CLK" ;
                timing_type : combinational ;
                timing_sense : positive_unate ;
                /* comment : mol clock feed; */
                cell_rise( f_itrans ){
                        index_1 ( "0.080000, 0.160000, 0.320000");
                        values ( " 0.251649, 0.266813, 0.286140");
```

```
                }
                rise_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                  "1.000000, 1.000000");
                }
                cell_fall( scalar ){
                        values ( "0.000000");
                }
                fall_transition( f_itrans_ocap ){
                        index_1 ( "0.000000, 1.000000");
                        index_2 ( "0.000000, 1.000000");
                        values ( "0.000000, 0.000000",\
                                  "1.000000, 1.000000");
                }
        } /* end of arc CLK_mol_c_OUT_CLK_R_CLK_una_min*/
} /* end of pin mol_c_OUT_CLK_R_CLK */
} /* end of cell */
} /* end of library */
```

Glossary

arc
A set of timing constraints associated with a single path.

asynchronous
The lack of a timing relationship between two different clocks or signals (transitions can occur at any time with respect to each other).

attribute
A string or value associated with an object in the design that carries some information about that object. For example, the `number_of_pins` attribute attached to a cell indicates the number of pins in the cell. You can find out the values of attributes by using the `get_attribute` command.

back-annotation
The process of applying detailed parasitic data to a design, allowing a more accurate timing analysis of the final circuit. The data comes from an external tool such as StarRC after completion of layout.

BSIM3/BSIM4
The industry-standard MOSFET SPICE models used for circuit simulation and CMOS technology development.

capture
The process of clocking data into a sequential element at a path endpoint, thus holding the data launched by an earlier clock edge at the path startpoint.

case analysis
The analysis of a design using fixed logic values on specified input ports.

cell
A component within a circuit with known timing characteristics, which can be used as an element in building a larger circuit for timing analysis.

channel-connected block (CCB)
A set of transistors whose sources and drains are connected together in a network, in series or in parallel, or both. The `check_design` command divides the design into these units for simulation purposes.

clock
A periodic signal that latches data into sequential elements. You define a signal to be a clock and specify its timing characteristics by using the `create_clock` or `create_generated_clock` command.

clock gating
The control of a clock signal by a combinational logic element such as a multiplexer or AND gate, to either shut down the clock at selected times or to modify the clock pulse characteristics.

clock propagation
The process of tracing clock signals from the clock input ports, through wires, buffers, inverters, and clock-gating circuitry, to the clock input pins of sequential elements.

clock path
A timing path that starts at a clock input port or a defined point inside a circuit and ends at a clock input pin of a sequential element.

clock latency
See latency.

clock reconvergence pessimism
An accuracy limitation of static timing analysis occurring when two different clock paths partially share a common physical path segment, and the analysis tool uses the minimum delay of the shared segment for one path and the maximum delay of the same segment for the other path. Correction of this inaccuracy is called clock reconvergence pessimism removal.

clock skew
See skew.

closing edge
The edge of a clock signal that latches data into a sequential element and ends the transparency of that element.

collection
A set of objects taken from the current design. For example, the `set mylist [get_clocks "CLK*"]` command creates a collection of clocks and sets a variable called mylist to the collection. Then you can use commands that operate on the collection, such as `remove_clock $mylist`. You can also generate and operate on a collection within a single command, such as `remove_clock [get_clocks "CLK*"]`.

combinational logic
A logic network that only contains elements that have no memory or internal state. Combinational logic can contain AND, OR, XOR, and inverter elements, but cannot contain flip-flops, latches, registers, or RAM.

common point
For a setup or hold timing check, the point in the design where the launch clock path and capture clock path diverge from a common or shared segment.

conservative
An analysis type or technique that favors pessimism (rather than optimism) to ensure detection of all violations.

constraint
A timing specification or requirement that applies to a path or a design. A timing constraint limits the allowed range of time during which signals can arrive at a device input or be valid at a device output.

context characterization
The process of using the `characterize_context` command to capture the timing context of a subdesign. This process allows standalone timing analysis of the subdesign in NanoTime. The timing context of an instance includes clock information, input arrival times, output delay times, timing exceptions, design rules, propagated constants, input drives, and capacitive loads.

context independent
Accurate in different contexts. This term refers to the usability, not behavior, of a timing model. For example, a timing model created with the `extract_model` command is context independent with respect to input data arrival time, input data transition time, data output required time, and output load. The model can be used accurately in different contexts because its behavior changes with the context in which it is used.

CPU time
The amount of time used by the central processing unit of the computer to accomplish a task, as reported by the tool. This is a measure of computation resources required for the task. CPU time is less than the actual elapsed time.

critical path
The path that has the largest violation (or the least timing slack) for a timing check.

current design
The loaded design currently considered to be the top-level design in a design hierarchy, on which NanoTime commands operate. You can specify the current design with the `current_design` command.

current instance
The instance (hierarchical cell) in a design hierarchy on which instance-specific commands operate by default. You can set the current instance with the `current_instance` command, which is similar to the cd operating system command.

D1 and D2 type domino
A type D1 domino precharge structure has a controlling clock on the evaluate stack, whereas a type D2 domino precharge structure does not.

data check
A timing check between two data signals, such as handshaking interface signals or recovery and removal checks between preset and clear pins. The two signals being checked are called the constrained and related signals. The related signal is the data signal treated as the clock in the timing relationship. For example, a setup data check verifies that the constrained signal is stable before the active edge of the related signal.

data path
A timing path that starts at a data launch point and ends at a data capture point. The term *path* usually means a data path, although there are other types of paths such as clock paths.

.db (database) format
A Synopsys file format used for storing designs, timing models, logic libraries, clock information, and detailed parasitics. NanoTime uses only the timing model information in these files, which it reads with the `read_library` command.

direct read
An implicit method of reading SPICE models into a NanoTime technology, invoked by using a .lib statement and including SPICE transistor models in the SPICE netlist files.

domino precharge circuit
A CMOS logic circuit that charges a node to a high voltage, which is then either left in the charged state or discharged upon arrival of an active clock edge, depending on the logic states at its inputs. A discharge event can discharge (or not discharge) a similar circuit downstream. A sequence of such circuits can evaluate a logic condition within a single clock cycle.

DSPF
Detailed Standard Parasitic Format, a format used to store circuit parasitic information for back-annotation.

endpoint
The place in a design where a timing path ends, where data gets captured by a clock edge or where the data must be available at a specific time. An endpoint must be at a sequential element or an output port.

erasing
The process of removing annotations on the design that identify circuit structures previously marked by the `match_topology` command or by commands such as `mark_inverter` and `mark_latch`.

evaluate phase
In a domino precharge circuit, the latter part of the clock cycle during which the precharge node is either discharged or not discharged, depending on the logic states of the inputs to the circuit.

exception
See timing exception.

exclusive clocks
Two clocks that are never active at the same time (for example, because they are multiplexed).

false path
A path that exists in a design that should not be analyzed for timing, such as a path between two multiplexed blocks that are never enabled at the same time.

flip-flop
A stable, nontransparent memory structure that holds a logic 0 or logic 1, such as a structure made of two NAND gates connected in a feedback configuration. The state of the memory structure can change only on active clock edges.

gated clock
A clock signal that can be modified by logic within the design, such as a clock that can be turned off to save power.

generated clock
A clock signal that is generated internally by the integrated circuit itself; a clock that does not come directly from an external source. An example of a generated clock is a divide-by-2 clock generated from the system clock, having half the frequency of the system clock. You specify the characteristics of a generated clock by using the `create_generated_clock` command.

hold constraint
A timing constraint that specifies how much time is necessary for data to be stable at the input of a device after the clock edge that clocks the data into the device. This constraint enforces a minimum delay on a timing path relative to a clock.

inout pin
A bidirectional pin of a cell; a pin that can operate as both an input and an output.

input delay
A constraint that specifies the minimum or maximum amount of delay from a clock edge to the arrival of a signal at a specified input port. NanoTime uses this information to check for timing violations at the input port and in the transitive fanout from that input port.

intellectual property
Information owned by one company that is licensed for use by another company. For example, one company might offer a license to use a chip submodule such as a microprocessor core in another company's application-specific circuit. The owner of the submodule design can provide the layout information, electrical specifications, and a timing model to the other company, without revealing the submodule netlist.

intersection transparency
A setup and hold checking mode in which NanoTime uses same-cycle checking when there is any amount of overlap between the transparency windows at the launch and capture latches. In the absence of this mode, NanoTime uses same-cycle checking only when the launch and capture edges occur at the same time.

latch
A memory structure that has two allowable stable states, 0 and 1. The term *latch* usually implies a structure that is transparent while the gate input is asserted. In other words, data passes through from the input to the output when the gate is open (when the gate input is asserted), and the value is "frozen" when the gate is closed. The inactive-to-active edge of the gate signal is called the opening edge. The active-to-inactive edge of the gate signal is called the closing edge.

latency
The amount of time that a clock signal takes to be propagated from the clock source to a specific point inside the design. The clock signal is "hidden" (latent) for this amount of time.

launch
The process of clocking data out of a latch or flip-flop at the startpoint of a path, releasing data that is captured by another clock edge at the endpoint of the path.

level (simulation level)
One channel-connected block (CCB) used as a simulation unit for timing analysis. For example, an inverter is typically considered one simulation unit or one level. An analysis using path-based slack adjustment (PBSA) can adjust the slack based on the number of simulation levels in the paths belonging to the timing check.

library
A database (.db file) containing timing models, read into NanoTime with the `read_library` command.

library cell
A timing model stored in a .db library file, which can be read into NanoTime and used in place of a netlist to represent a lower-level block in a hierarchical analysis.

library topology
A set of files that describe the pattern or subcircuit name and the list of NanoTime actions taken when the pattern or subcircuit is found. You can selectively enable or disable each topology within a library by using the `set_search_enabled` and `remove_search_enabled` commands.

linking
The process of resolving the hierarchy of a design, in which NanoTime builds a complete hierarchical representation of the whole design and stores that design in memory for analysis.

load
The amount of capacitance on a net that must be driven by the driver of the net when there is a transition on the net.

logic constraint
A fixed logical relationship specified for a set of nets in the design, such as an inverse relationship between two nets, or a “one-hot” relationship between a set of four nets (exactly one of the four nets is true at any given time).

man page
A description of a command, variable, or message code displayed by using the `man` command in NanoTime. For example, entering `man create_clock` at the nt_shell prompt displays the man page for the `create_clock` command. This is similar to the operating system man command (which is derived from the word “manual”).

marking
The process of annotating a design to indicate the locations of circuit structures such as inverters, multiplexers, latches, and domino precharge circuits. Marking can be done automatically with the `match_topology` command or manually with commands such as `mark_inverter` and `mark_latch`.

multicycle path
A path that is designed to take more than one clock cycle for the data to propagate from the startpoint to the endpoint. For proper analysis, you need to define such a path as a timing exception with the `set_multicycle_path` command.

N-domino circuit
A domino precharge circuit that has an NMOS evaluate stack and a precharge node that is first charged and then conditionally discharged during the evaluate phase.

net delay
The amount of delay from the driver of a net to a load connected to the net. This delay is the result of parasitic capacitance of the wire connection between the driver and load.

netlist
A circuit description (nets, components, and connections) provided to NanoTime in the form of data files. The most common formats are SPICE and Verilog with SPICE.

network latency
The amount of time a clock signal takes to propagate from the clock definition point in the design to the clock input of a sequential element.

next-cycle checking
Setup and hold timing checks that are based on the assumption that the capture clock edge occurs in the next clock cycle after the launch clock edge, rather than in the same clock cycle.

NMOS transistor
A transistor that uses n-type silicon for the source and drain and p-type silicon for the channel. A positive voltage on the gate induces an n-type conductive path through the channel.

no-change constraint
The amount of time that a data signal must remain unchanged before, during, and after a pulse of a control signal (for example, an address signal that must be stable during a write pulse). The data signal must be stable for a specified setup time before the leading edge of the control pulse, during the control pulse, and for a specified hold time after the trailing edge of the control pulse.

no-check timing exception
A timing exception set on a path to prevent timing checks from being performed on the path, while still allowing path tracing to continue through the specified endpoint.

nt_shell
The Tcl-based command-line environment of NanoTime that lets you enter commands, run scripts, and view results.

opening edge
The edge of a clock signal that asserts a gate input of a level-sensitive latch. The latch is transparent from the opening edge to the closing edge of the clock signal.

operating conditions
The process, voltage, and temperature conditions under which a circuit operates, which affect the timing characteristics of the cells used in the design. The `set_technology` command specifies the technologies (and their associated operating conditions) to use for analysis.

optimistic
An analysis type or technique with a known accuracy limitation, when the inaccuracy might indicate that a circuit has more timing slack than the real circuit. As a result, a violation in the real circuit might not be detected.

output delay
A constraint that specifies the minimum or maximum amount of delay from an output port to the external sequential device that captures data from that output port. This constraint establishes the times at which signals must be available at the output port to meet the setup and hold requirements of the external circuit.

P-domino circuit
A domino predischarge circuit, which has a PMOS evaluate stack and a predischarge node that is first discharged and then conditionally charged during the evaluate phase.

parasitic capacitance
Capacitance of a net due to the physical proximity between the net interconnections and adjacent nets or the substrate; or due to the charge storage effects of p-n junctions in the net.

parasitic resistance
Resistance of an interconnection or net due to the finite conductivity of the interconnect material.

path
A point-to-point sequence through a design that starts at a register clock pin or an input port, passes through any number of combinational logic elements and transparent latches, and ends at a sequential element or output port.

path-based slack adjustment (PBSA)
An optional adjustment feature in NanoTime for setup and hold slack calculation that estimates the worst-case delay uncertainty for a path based on the length of the path. Longer paths have more delay uncertainty, causing them to have a smaller reported slack.

path endpoint
See endpoint.

path startpoint
See startpoint.

path tracing
The process of calculating the delays along different paths in the design, performed by the `trace_paths` command in NanoTime.

PathMill
The previous-generation Synopsys tool for transistor-level static timing analysis; the predecessor of NanoTime.

pattern matching
The process of identifying circuit structures within a design that match a given network of interconnected transistors. This process is useful for identifying and marking structures that cannot be recognized by other means.

PBSA
See path-based slack adjustment (PBSA).

pessimistic
An analysis type or technique with a known accuracy limitation, when the inaccuracy might indicate that a circuit has less timing slack than the real circuit. As a result, a violation might be reported that would not actually exist in the real circuit.

pin
An input or output of a cell or transistor in a design.

PMOS transistor
A transistor that uses p-type silicon for the source and drain and n-type silicon for the channel. A negative voltage on the gate induces a p-type conductive path through the channel.

port
An input or output of a design. You specify timing constraints for ports with the `set_input_delay` and `set_output_delay` commands.

precharge circuit
A domino precharge circuit, or the part of such a circuit that charges a node to a high voltage in preparation for the evaluate phase.

precharge phase
In a domino precharge circuit, the part of the clock cycle during which the precharge node is charged in preparation for logical evaluation in the evaluate phase.

predischarge circuit
A circuit similar to a domino precharge circuit, except that the internal node is discharged (rather than charged) in the earlier part of the clock cycle and is conditionally charged (rather than discharged) during the evaluate phase of the clock cycle. Also known as a P-domino circuit.

preferred models
In a hierarchical analysis, a list of timing models that are to be used in place of netlists to represent lower-level blocks in the design hierarchy. The preferred models are specified with variables such as `link_prefer_model`.

PrimeTime
The Synopsys standalone, full-chip, gate-level static timing analysis tool. Like NanoTime, it performs setup and hold checks on timing paths and reports the worst-case slack values. However, it operates on gate-level rather than transistor-level netlists, and it depends on library characterization of gates rather than simulation to determine the circuit element delays.

procedure (Tcl)
A user-defined command created by the `proc` command in NanoTime. A procedure, after it is defined, can be executed like an ordinary command. A procedure can take arguments and can use local and externally defined variables.

pruning
A process during path tracing that removes from consideration the paths that are known not to be the worst-case paths. This feature reduces runtime without losing information about the worst-case paths.

pulse clock
A clock consisting of a sequence of pulses whose leading and trailing edges are both triggered by the same source clock edge. The pulse width is determined by the differences in delay from the clock source edge to the circuits that generate the leading and trailing edges of the pulse.

RAM
A memory structure consisting of two equal-strength inverters feeding back into each other. NanoTime terminates a path search when it reaches a RAM.

recovery constraint
The minimum amount of time required between an asynchronous control signal (such as the asynchronous clear input of a flip-flop) going inactive and a subsequent active clock edge. This is like a setup check for the active clock edge. If the active clock edge occurs too soon after the asynchronous input goes inactive, the register data is uncertain because the clocked input data might not be valid.

register
A memory structure consisting of a RAM structure driven from both sides, with an output taken from only one side.

registry
A list of data files that NanoTime is using or intends to use. For example, the netlist registry is a list of netlist files that have been specified with the `register_netlist` command. NanoTime has a netlist registry (for netlist files), technology registry (for technologies read implicitly from SPICE netlists or with `read_spice_model`), and a library registry (for .db timing model libraries read with `read_library`).

related signal
The data signal treated as a clock in a data-to-data timing check. The other signal being checked is called the constrained signal.

removal constraint
The minimum amount of time required between a clock edge that occurs while an asynchronous input is active and the subsequent removal of the asserted asynchronous control signal. This is like a hold check for the removal of the asynchronous control signal. If the asynchronous input signal goes inactive too soon after a clock edge, the register data is uncertain because the clocked input data might be valid and conflict with the asynchronous control signal.

same-cycle checking
Setup and hold timing checks that are based on the assumption that the capture clock edge occurs in the same clock cycle as the launch clock edge, rather than in the next clock cycle. This is the default checking mode in NanoTime.

script
A text file containing a sequence of nt_shell commands, which can be executed with the `source` command.

SDC
Synopsys Design Constraints, a standard command-file format for specifying timing constraints that can be shared between different tools such as Design Compiler, PrimeTime, and NanoTime. An SDC-format file can be executed as a command script in any of these tools.

segments A, B, C, D
The four path segments considered in a setup or hold check using path-based slack adjustment (PBSA). Path segment A is the shared or common portion of the launch clock path and capture clock path (if any), from the clock input port to the common point. Segment B is the portion of the launch clock path from the common point to the clock pin of the launching sequential element. Segment C is the portion of the capture clock path from the common point to the clock pin of the capturing sequential element. Segment D is the data path from the clock pin of the launching sequential element to the data input pin of the capturing sequential element.

sequential logic
A logic network that contains elements that have memory or internal state, such as flip-flops, latches, registers, or RAM.

setup constraint
A timing constraint that specifies how much time is necessary for data to be available at the input of a device before the clock edge that clocks the data into the device. This constraint enforces a maximum delay on a timing path relative to a clock.

simulation
The process of determining the behavior of a circuit over a period of time in response to a stimulus. NanoTime performs simulation of small circuit units in the design as a part of path tracing with the `trace_paths` command.

simulation level
One channel-connected block used as a simulation unit for timing analysis. For example, an inverter is typically considered one simulation unit or one level. An analysis using path-based slack adjustment (PBSA) can adjust the slack based on the number of simulation levels in the paths belonging to the timing check.

skew
The shift or variation from the nominal, expected time that a clock transition occurs; or the difference in the amount of shift between two different points in a clock network.

slack
The amount of time by which a violation is avoided. For example, if a signal must reach a cell input at no later than 8 ns and is determined to arrive at 5 ns, the slack is 3 ns. A slack of 0 means that the timing constraint is just barely satisfied. A negative slack indicates a timing violation.

slew
The amount of time it takes for a signal to change from low to high or from high to low; also known as transition time.

source latency
The amount of time a clock signal takes to propagate from its ideal waveform origin point to the clock definition point in the design.

SPEF
Standard Parasitic Exchange Format, a format used to store circuit parasitic information for back-annotation. This format is defined by Open Verilog International (OVI).

SPICE
Simulation Program with Integrated Circuit Emphasis, a time-based simulation tool that analyzes circuit operation at the level of transistors, capacitors, and resistors. Many different forms of this simulator are available from different sources.

startpoint
The place in a design where a timing path starts, where data is launched by a clock edge or where the data is expected to be available at a specific time. A startpoint must be either the clock pin of a sequential element or an input port.

static timing analysis
An efficient analysis technique that determines whether a circuit violates any timing requirements by considering clocks, path delays, and timing constraints. Compared with

full circuit simulation, static timing analysis is faster and more thorough, and does not require test vectors. However, it does not check the functionality of the circuit.

synchronous
A timing relationship between two clocks or signals in which specific transitions occur at the same time or have a phase relationship that stays fixed over time.

Tcl
Tool command language, a standard command scripting language on which the nt_shell interface is based. You can use this language to write scripts for performing complex tasks in NanoTime.

Tcl procedure
See procedure (Tcl).

technology
A set of SPICE transistor models read into NanoTime implicitly with the SPICE netlist or explicitly with the `read_spice_model` command. NanoTime uses a technology for SPICE simulation of the design. After it is read in, the `set_technology` command specifies which technology to use for tracing minimum data paths, maximum data paths, minimum clock paths, and maximum clock paths.

three-state
An output of a device that can be in any of three logical states: low (0), high (1), or the high-impedance state (Z).

timing arc
See arc.

timing exception
A path-level exception to the default timing behavior assumed by NanoTime, such as a false path or multicycle path. You must declare such exceptions for NanoTime to correctly perform timing checks on those paths.

timing model
A model created by the `extract_model` command. A timing model represents the timing characteristics of the block but does not contain any functional information. These models are used in hierarchical timing analysis to simplify analysis of large designs.

topology
The arrangement or configuration of transistors and connections that form a structure, or a structure that has been recognized by its configuration. NanoTime can recognize structures (transfer gates, latches, registers, inverters, and so on) based on the topology of the design.

topology library
A directory containing a set of topology definitions. There are two default topology libraries, called the common and global libraries. You can also create your own custom libraries. You can selectively enable or disable each library so that NanoTime spends time searching only for topologies expected to exist in the design.

topology recognition
The automated recognition of circuit structures such as inverters, multiplexers, latches, and domino precharge circuits from the configuration of interconnected transistors. This process is invoked by the `match_topology` command.

transfer gate
A circuit structure that either passes or blocks a signal based on the state of a control signal or a pair of complementary control signals.

transition time
The amount of time it takes for a signal to change from low to high or from high to low; also known as slew.

turnoff
A circuit structure consisting of two PMOS pullup transistors and two NMOS pulldown transistors connected in series, with complementary enable signals at the respective PMOS and NMOS transistor gates. NanoTime performs contention analysis at each turnoff structure.

type D1 and D2 domino
See D1 and D2 type domino.

uncertainty (clock uncertainty)
The amount of variation from the nominal, ideal-clock times at which clock edges occur.

variable
A NanoTime parameter that can be set to a string, integer, or floating-point value that affects the analysis. Variables can be set with the `set` command and reported with the `printvar` command.

weak pullup
A transistor that is turned on and connected to the power supply voltage at one end. NanoTime assumes that the transistor pulls up the net at the other end of the transistor when there is nothing pulling the net down.