Static Timing Analysis for Nanometer Designs

A Practical Approach

J. Bhasker • Rakesh Chadha

Static Timing Analysis for Nanometer Designs
A Practical Approach

Springer

J. Bhasker
eSilicon Corporation
1605 N. Cedar Crest Blvd.
Suite 615
Allentown, PA 18103, USA
jbhasker@esilicon.com

Rakesh Chadha
eSilicon Corporation
890 Mountain Ave
New Providence, NJ 07974, USA
rchadha@esilicon.com

ISBN 978-0-387-93819-6 e-ISBN 978-0-387-93820-2
DOI: 10.1007/978-0-387-93820-2

Library of Congress Control Number: 2009921502

Printed on acid-free paper.

springer.com

Contents

❑

Preface

Timing, timing, timing! That is the main concern of a digital designer charged with designing a semiconductor chip. What is it, how is it described, and how does one verify it? The design team of a large digital design may spend months architecting and iterating the design to achieve the required timing target. Besides functional verification, the timing closure is the major milestone which dictates when a chip can be released to the semiconductor foundry for fabrication. This book addresses the timing verification using static timing analysis for nanometer designs.

The book has originated from many years of our working in the area of timing verification for complex nanometer designs. We have come across many design engineers trying to learn the background and various aspects of static timing analysis. Unfortunately, there is no book currently available that can be used by a working engineer to get acquainted with the details of static timing analysis. The chip designers lack a central reference for information on timing, that covers the basics to the advanced timing verification procedures and techniques.

The purpose of this book is to provide a reference for both beginners as well as professionals working in the area of static timing analysis. The book

is intended to provide a blend of the underlying theoretical background as well as in-depth coverage of timing verification using static timing analysis. The book covers topics such as cell timing, interconnect, timing calculation, and crosstalk, which can impact the timing of a nanometer design. It describes how the timing information is stored in cell libraries which are used by synthesis tools and static timing analysis tools to compute and verify timing.

This book covers CMOS logic gates, cell library, timing arcs, waveform slew, cell capacitance, timing modeling, interconnect parasitics and coupling, pre-layout and post-layout interconnect modeling, delay calculation, specification of timing constraints for analysis of internal paths as well as IO interfaces. Advanced modeling concepts such as composite current source (CCS) timing and noise models, power modeling including active and leakage power, and crosstalk effects on timing and noise are described.

The static timing analysis topics covered start with verification of simple blocks particularly useful for a beginner to this area. The topics then extend to complex nanometer designs with concepts such as modeling of on-chip variations, clock gating, half-cycle and multicycle paths, false paths, as well as timing of source synchronous IO interfaces such as for DDR memory interfaces. Timing analyses at various process, environment and interconnect corners are explained in detail. Usage of hierarchical design methodology involving timing verification of full chip and hierarchical building blocks is covered in detail. The book provides detailed descriptions for setting up the timing analysis environment and for performing the timing analysis for various cases. It describes in detail how the timing checks are performed and provides several commonly used example scenarios that help illustrate the concepts. Multi-mode multi-corner analysis, power management, as well as statistical timing analyses are also described.

Several chapters on background reference materials are included in the appendices. These appendices provide complete coverage of SDC, SDF and SPEF formats. The book describes how these formats are used to provide information for static timing analysis. The SDF provides cell and interconnect delays for a design under analysis. The SPEF provides parasitic information, which are the resistance and capacitance networks of nets in a

design. Both SDF and SPEF are industry standards and are described in detail. The SDC format is used to provide the timing specifications or constraints for the design under analysis. This includes specification of the environment under which the analysis must take place. The SDC format is a defacto industry standard used for describing timing specifications.

The book is targeted for professionals working in the area of chip design, timing verification of ASICs and also for graduate students specializing in logic and chip design. Professionals who are beginning to use static timing analysis or are already well-versed in static timing analysis can use this book since the topics covered in the book span a wide range. This book aims to provide access to topics that relate to timing analysis, with easy-to-read explanations and figures along with detailed timing reports.

The book can be used as a reference for a graduate course in chip design and as a text for a course in timing verification targeted to working engineers. The book assumes that the reader has a background knowledge of digital logic design. It can be used as a secondary text for a digital logic design course where students learn the fundamentals of static timing analysis and apply it for any logic design covered in the course.

Our book emphasizes practicality and thorough explanation of all basic concepts which we believe is the foundation of learning more complex topics. It provides a blend of theoretical background and hands-on guide to static timing analysis illustrated with actual design examples relevant for nanometer applications. Thus, this book is intended to fill a void in this area for working engineers and graduate students.

The book describes timing for CMOS digital designs, primarily synchronous; however, the principles are applicable to other related design styles as well, such as for FPGAs and for asynchronous designs.

Book Organization

The book is organized such that the basic underlying concepts are described first before delving into more advanced topics. The book starts

with the basic timing concepts, followed by commonly used library modeling, delay calculation approaches, and the handling of noise and crosstalk for a nanometer design. After the detailed background, the key topics of timing verification using static timing analysis are described. The last two chapters focus on advanced topics including verification of special IO interfaces, clock gating, time borrowing, power management and multi-corner and statistical timing analysis.

Chapter 1 provides an explanation of what static timing analysis is and how it is used for timing verification. Power and reliability considerations are also described. Chapter 2 describes the basics of CMOS logic and the timing terminology related to static timing analysis.

Chapter 3 describes timing related information present in the commonly used library cell descriptions. Even though a library cell contains several attributes, this chapter focuses only on those that relate to timing, crosstalk, and power analysis. Interconnect is the dominant effect on timing in nanometer technologies and Chapter 4 provides an overview of various techniques for modeling and representing interconnect parasitics.

Chapter 5 explains how cell delays and paths delays are computed for both pre-layout and post-layout timing verification. It extends the concepts described in the preceding chapters to obtain timing of an entire design.

In nanometer technologies, the effect of crosstalk plays an important role in the signal integrity of the design. Relevant noise and crosstalk analyses, namely glitch analysis and crosstalk analysis, are described in Chapter 6. These techniques are used to make the ASIC behave robustly from a timing perspective.

Chapter 7 is a prerequisite for succeeding chapters. It describes how the environment for timing analysis is configured. Methods for specifying clocks, IO characteristics, false paths and multicycle paths are described in Chapter 7. Chapter 8 describes the timing checks that are performed as part of various timing analyses. These include amongst others - setup, hold and asynchronous recovery and removal checks. These timing checks are intended to exhaustively verify the timing of the design under analysis.

Chapter 9 focuses on the timing verification of special interfaces such as source synchronous and memory interfaces including DDR (Double Data Rate) interfaces. Other advanced and critical topics such as on-chip variation, time borrowing, hierarchical methodology, power management and statistical timing analysis are described in Chapter 10.

The SDC format is described in Appendix A. This format is used to specify the timing constraints of a design. Appendix B describes the SDF format in detail with many examples of how delays are back-annotated. This format is used to capture the delays of a design in an ASCII format that can be used by various tools. Appendix C describes the SPEF format which is used to provide the parasitic resistance and capacitance values of a design.

All timing reports are generated using PrimeTime, a static timing analysis tool from Synopsys, Inc. Highlighted text in reports indicates specific items of interest pertaining to the explanation in the accompanying text.

New definitions are highlighted in **bold**. Certain words are highlighted in *italics* just to keep the understanding that the word is special as it relates to this book and is different from the normal English usage.

Acknowledgments

We would like to express our deep gratitude to eSilicon Corporation for providing us the opportunity to write this book.

We also would like to acknowledge the numerous and valuable insights provided by Kit-Lam Cheong, Ravi Kurlagunda, Johnson Limqueco, Pete Jarvis, Sanjana Nair, Gilbert Nguyen, Chris Papademetrious, Pierrick Pedron, Hai Phuong, Sachin Sapatnekar, Ravi Shankar, Chris Smirga, Bill Tuohy, Yeffi Vanatta, and Hormoz Yaghutiel, in reviewing earlier drafts of the book. Their feedback has been invaluable in improving the quality and usefulness of this book.

Last, but not least, we would like to thank our families for their patience during the development of this book.

Dr. Rakesh Chadha
Dr. J. Bhasker

January 2009

CHAPTER

1

Introduction

This chapter provides an overview of the static timing analysis procedures for nanometer designs. This chapter addresses questions such as, what is static timing analysis, what is the impact of noise and crosstalk, how these analyses are used and during which phase of the overall design process are these analyses applicable.

1.1 Nanometer Designs

In semiconductor devices, metal interconnect traces are typically used to make the connections between various portions of the circuitry to realize the design. As the process technology shrinks, these interconnect traces have been known to affect the performance of a design. For deep submi-

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_1, © Springer Science + Business Media, LLC 2009

cron or nanometer process technologies[1], the coupling in the interconnect induces noise and crosstalk - either of which can limit the operating speed of a design. While the noise and coupling effects are negligible at older generation technologies, these play an important role in nanometer technologies. Thus, the physical design should consider the effect of crosstalk and noise and the design verification should then include the effects of crosstalk and noise.

1.2 What is Static Timing Analysis?

Static Timing Analysis (also referred as STA) is one of the many techniques available to verify the timing of a digital design. An alternate approach used to verify the timing is the timing simulation which can verify the functionality as well as the timing of the design. The term *timing analysis* is used to refer to either of these two methods - static timing analysis, or the timing simulation. Thus, timing analysis simply refers to the analysis of the design for timing issues.

The STA is *static* since the analysis of the design is carried out statically and does not depend upon the data values being applied at the input pins. This is in contrast to simulation based timing analysis where a stimulus is applied on input signals, resulting behavior is observed and verified, then time is advanced with new input stimulus applied, and the new behavior is observed and verified and so on.

Given a design along with a set of input clock definitions and the definition of the external environment of the design, the purpose of static timing analysis is to validate if the design can operate at the rated speed. That is, the design can operate safely at the specified frequency of the clocks without any timing violations. Figure 1-1 shows the basic functionality of static

1. Deep submicron refers to process technologies with a feature size of 0.25µm or lower. The process technologies with feature size below 0.1µm are referred to as *nanometer technologies*. Examples of such process technologies are 90nm, 65nm, 45nm, and 32nm. The finer process technologies normally allow a greater number of metal layers for interconnect.

timing analysis. The **DUA** is the design under analysis. Some examples of timing checks are setup and hold checks. A setup check ensures that the data can arrive at a flip-flop within the given clock period. A hold check ensures that the data is held for at least a minimum time so that there is no unexpected pass-through of data through a flip-flop: that is, it ensures that a flip-flop captures the intended data correctly. These checks ensure that the proper data is ready and available for capture and latched in for the new state.

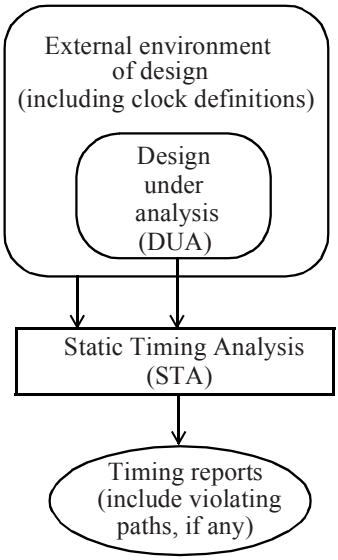


Figure 1-1 *Static timing analysis.*

The more important aspect of static timing analysis is that the entire design is analyzed once and the required timing checks are performed for all possible paths and scenarios of the design. Thus, STA is a complete and exhaustive method for verifying the timing of a design.

The design under analysis is typically specified using a hardware description language such as VHDL[1] or Verilog HDL[2]. The external environment, including the clock definitions, are specified typically using SDC[3] or an equivalent format. SDC is a timing constraint specification language. The timing reports are in ASCII form, typically with multiple columns, with each column showing one attribute of the path delay. Many examples of timing reports are provided as illustrations in this book.

1.3 Why Static Timing Analysis?

Static timing analysis is a complete and exhaustive verification of all timing checks of a design. Other timing analysis methods such as simulation can only verify the portions of the design that get exercised by stimulus. Verification through timing simulation is only as exhaustive as the test vectors used. To simulate and verify all timing conditions of a design with 10-100 million gates is very slow and the timing cannot be verified completely. Thus, it is very difficult to do exhaustive verification through simulation.

Static timing analysis on the other hand provides a faster and simpler way of checking and analyzing all the timing paths in a design for any timing violations. Given the complexity of present day ASICs[4], which may contain 10 to 100 million gates, the static timing analysis has become a necessity to exhaustively verify the timing of a design.

Crosstalk and Noise

The design functionality and its performance can be limited by noise. The noise occurs due to crosstalk with other signals or due to noise on primary inputs or the power supply. The noise impact can limit the frequency of

1. See [BHA99] in Bibliography.
2. See [BHA05] in Bibliography.
3. Synopsys Design Constraints: It is a defacto standard but a proprietary format of Synopsys, Inc.
4. Application-Specific Integrated Circuit.

operation of the design and it can also cause functional failures. Thus, a design implementation must be verified to be robust which means that it can withstand the noise without affecting the rated performance of the design.

Verification based upon logic simulation cannot handle the effects of crosstalk, noise and on-chip variations.

The analysis methods described in this book cover not only the traditional timing analysis techniques but also noise analysis to verify the design including the effects of noise.

1.4 Design Flow

This section primarily describes the CMOS[1] digital design flow in the context used in the rest of this book. A brief description of its applicability to FPGAs[2] and to asynchronous designs is also provided.

1.4.1 CMOS Digital Designs

In a CMOS digital design flow, the static timing analysis can be performed at many different stages of the implementation. Figure 1-2 shows a typical flow.

STA is rarely done at the RTL level as, at this point, it is more important to verify the functionality of the design as opposed to timing. Also not all timing information is available since the descriptions of the blocks are at the behavioral level. Once a design at the RTL level has been synthesized to the gate level, the STA is used to verify the timing of the design. STA can also be run prior to performing logic optimization - the goal is to identify the worst or critical timing paths. STA can be rerun after logic optimization to

1. Complimentary Metal Oxide Semiconductor.
2. Field Programmable Gate Array: Allows for design functionality to be programmed by the user after manufacture.

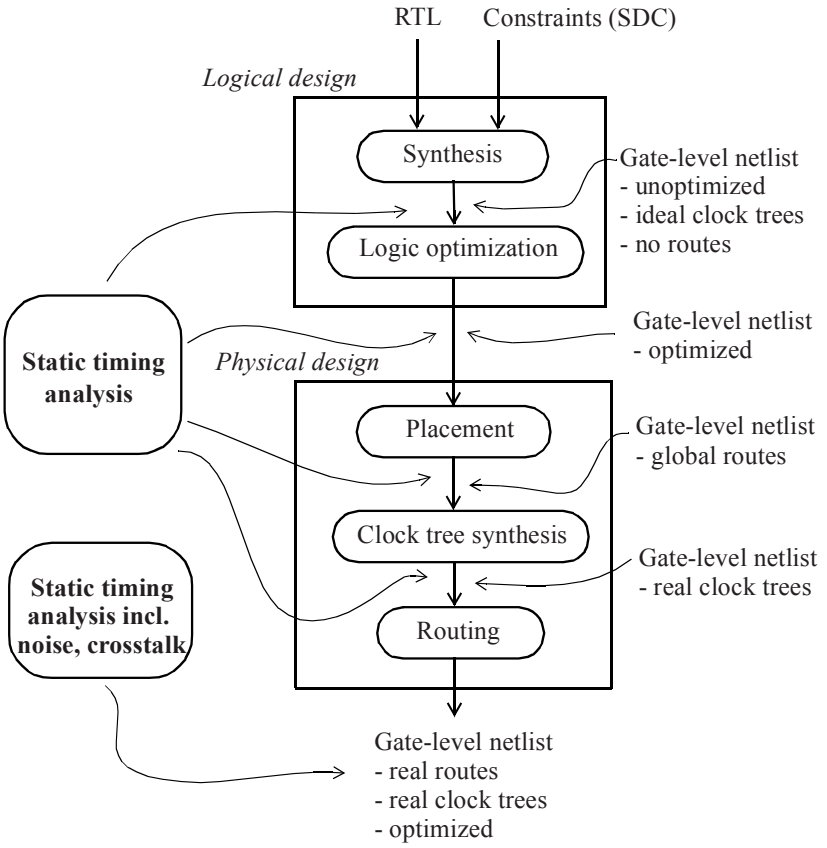


Figure 1-2 *CMOS digital design flow.*

see whether there are failing paths still remaining that need to be optimized, or to identify the critical paths.

At the start of the physical design, clock trees are considered as ideal, that is, they have zero delay. Once the physical design starts and after clock trees are built, STA can be performed to check the timing again. In fact,

during physical design, STA can be performed at each and every step to identify the worst paths.

In physical implementation, the logic cells are connected by interconnect metal traces. The parasitic RC (**R**esistance and **C**apacitance) of the metal traces impact the signal path delay through these traces. In a typical nanometer design, the parasitics of the interconnect can account for the majority of the delay and power dissipation in the design. Thus, any analysis of the design should evaluate the impact of the interconnect on the performance characteristics (speed, power, etc.). As mentioned previously, coupling between signal traces contributes to noise, and the design verification must include the impact of the noise on the performance.

At the logical design phase, ideal interconnect may be assumed since there is no physical information related to the placement; there may be more interest in viewing the logic that contributes to the worst paths. Another technique used at this stage is to estimate the length of the interconnect using a wireload model. The wireload model provides estimated RC based on the fanouts of a cell.

Before the routing of traces are finalized, the implementation tools use an estimate of the routing distance to obtain RC parasitics for the route. Since the routing is not finalized, this phase is called the *global route* phase to distinguish it from the *final route* phase. In the global route phase of the physical design, simplified routes are used to estimate routing lengths, and the routing estimates are used to determine resistance and capacitance that are needed to compute wire delays. During this phase, one can not include the effect of coupling. After the detailed routing is complete, actual RC values obtained from extraction are used and the effect of coupling can be analyzed. However, a physical design tool may still use approximations to help improve run times in computing RC values.

An extraction tool is used to extract the detailed parasitics (RC values) from a routed design. Such an extractor may have an option to obtain parasitics with small runtime and less accurate RC values during iterative optimization and another option for final verification during which very accurate RC values are extracted with a larger runtime.

To summarize, the static timing analysis can be performed on a gate-level netlist depending on:

i. How interconnect is modeled - ideal interconnect, wireload model, global routes with approximate RCs, or real routes with accurate RCs.

ii. How clocks are modeled - whether clocks are ideal (zero delay) or propagated (real delays).

iii. Whether the coupling between signals is included - whether any crosstalk noise is analyzed.

Figure 1-2 may seem to imply that STA is done outside of the implementation steps, that is, STA is done after each of the synthesis, logic optimization, and physical design steps. In reality, each of these steps perform integrated (and incremental) STA within their functionality. For example, the timing analysis engine within the logic optimization step is used to identify critical paths that the optimizer needs to work on. Similarly, the integrated timing analysis engine in a placement tool is used to maintain the timing of the design as layout progresses incrementally.

1.4.2 FPGA Designs

The basic flow of STA is still valid in an FPGA. Even though routing in an FPGA is constrained to channels, the mechanism to extract parasitics and perform STA is identical to a CMOS digital design flow. For example, STA can be performed assuming interconnects as ideal, or using a wireload model, assuming clock trees as ideal or real, assuming global routes, or using real routes for parasitics.

1.4.3 Asynchronous Designs

The principles of STA are applicable in asynchronous designs also. One may be more interested in timing from one signal in the design to another as opposed to doing setup and hold checks which may be non-existent. Thus, most of the checks may be point to point timing checks, or skew

checks. The noise analysis to analyze the glitches induced due to coupling are applicable for any design - asynchronous or synchronous. Also, the noise analysis impact on timing, including the effect of the coupling, is valid for asynchronous designs as well.

1.5 STA at Different Design Phases

At the logical level (gate-level, no physical design yet), STA can be carried out using:

i. Ideal interconnect or interconnect based on wireload model.

ii. Ideal clocks with estimates for latencies and jitter.

During the physical design phase, in addition to the above modes, STA can be performed using:

i. Interconnect - which can range from global routing estimates, real routes with approximate extraction, or real routes with signoff accuracy extraction.

ii. Clock trees - real clock trees.

iii. With and without including the effect of crosstalk.

1.6 Limitations of Static Timing Analysis

While the timing and noise analysis do an excellent job of analyzing a design for timing issues under all possible situations, the state-of-the-art still does not allow STA to replace simulation completely. This is because there are some aspects of timing verification that cannot yet be completely captured and verified in STA.

Some of the limitations of STA are:

i. *Reset sequence*: To check if all flip-flops are reset into their required logical values after an asynchronous or synchronous reset. This is something that cannot be checked using static timing analysis. The chip may not come out of reset. This is because certain declarations such as initial values on signals are not synthesized and are only verified during simulation.

ii. *X-handling*: The STA techniques only deal with the logical domain of logic-0 and logic-1 (or high and low), rise and fall. An unknown value *X* in the design causes indeterminate values to propagate through the design, which cannot be checked with STA. Even though the noise analysis within STA can analyze and propagate the glitches through the design, the scope of glitch analysis and propagation is very different than the *X* handling as part of the simulation based timing verification for nanometer designs.

iii. *PLL settings*: PLL configurations may not be loaded or set properly.

iv. *Asynchronous clock domain crossings*: STA does not check if the correct clock synchronizers are being used. Other tools are needed to ensure that the correct clock synchronizers are present wherever there are asynchronous clock domain crossings.

v. *IO interface timing*: It may not be possible to specify the IO interface requirements in terms of STA constraints only. For example, the designer may choose detailed circuit level simulation for the DDR[1] interface using SDRAM simulation models. The simulation is to ensure that the memories can be read from and written to with adequate margin, and that the DLL[2], if any, can be controlled to align the signals if necessary.

1. **D**ouble **D**ata **R**ate.
2. **D**elay **L**ocked **L**oop.

vi. *Interfaces between analog and digital blocks*: Since STA does not deal with analog blocks, the verification methodology needs to ensure that the connectivity between these two kinds of blocks is correct.

vii. *False paths*: The static timing analysis verifies that timing through the logic path meets all the constraints, and flags violations if the timing through a logic path does not meet the required specifications. In many cases, the STA may flag a logic path as a failing path, even though logic may never be able to propagate through the path. This can happen when the system application never utilizes such a path or if mutually contradictory conditions are used during the sensitization of the failing path. Such timing paths are called *false paths* in the sense that these can never be realized. The quality of STA results is better when proper timing constraints including false path and multicycle path constraints are specified in the design. In most cases, the designer can utilize the inherent knowledge of the design and specify constraints so that the false paths are eliminated during the STA.

viii. *FIFO pointers out of synchronization*: STA cannot detect the problem when two finite state machines expected to be synchronous are actually out of synchronization. During functional simulations, it is possible that the two finite state machines are always synchronized and change together in lock-step. However, after delays are considered, it is possible for one of the finite state machines to be out of synchronization with the other, most likely because one finite state machine comes out of reset sooner than the other. Such a situation can not be detected by STA.

ix. *Clock synchronization logic*: STA cannot detect the problem of clock generation logic not matching the clock definition. STA assumes that the clock generator will provide the waveform as specified in the clock definition. There could be a bad optimization performed on the clock generator logic that causes, for example, a large delay to be inserted on one of the paths that

may not have been constrained properly. Alternately, the added logic may change the duty cycle of the clock. The STA cannot detect either of these potential conditions.

x. *Functional behavior across clock cycles*: The static timing analysis cannot model or simulate functional behavior that changes across clock cycles.

Despite issues such as these, STA is widely used to verify timing of the design and the simulation (with timing or with unit-delay) is used as a back-up to check corner cases and more simply to verify the normal functional modes of the design.

1.7 Power Considerations

Power is an important consideration in the implementation of a design. Most designs need to operate within a power budget for the board and the system. The power considerations may also arise due to conforming to a standard and/or due to a thermal budget on the board or system where the chip must operate in. There are often separate limits for *total power* and for *standby power*. The standby power limits are often imposed for hand-held or battery operated devices.

The power and timing often go hand in hand in most practical designs. A designer would like to use faster (or higher speed) cells for meeting the speed considerations but would likely run into the limit on available power dissipation. Power dissipation is an important consideration in the choice of process technology and cell library for a design.

1.8 Reliability Considerations

The design implementation must meet the reliability requirements. As described in Section 1.4.1, the metal interconnect traces have parasitic RC limiting the performance of the design. Besides parasitics, the metal trace widths need to be designed keeping the reliability considerations into account. For example, a high speed clock signal needs to be wide enough to meet reliability considerations such as electromigration.

1.9 Outline of the Book

While static timing analysis may appear to be a very simple concept on the surface, there is a lot of background knowledge underlying this analysis. The underlying concepts range from accurate representation of cell delays to computing worst path delays with minimum pessimism. The concepts of computing cell delays, timing a combinational block, clock relationships, multiple clock domains and gated clocks form an important basis for static timing analysis. Writing a correct SDC for a design is indeed a challenge.

The book has been written in a bottom-up order - presenting the simple concepts first followed by more advanced topics in later chapters. The book begins by representing accurate cell delays (Chapter 3). Estimating or computing exact interconnect delays and their representation in an effective manner is the topic of Chapter 4. Computing delay of a path composed of cells and interconnect is the topic of Chapter 5. Signal integrity, that is the effect of signal switching on neighboring nets and how it impacts the delay along a path, is the topic of Chapter 6. Accurately representing the environment of the DUA with clock definitions and path exceptions is the topic of Chapter 7. The details of the timing checks performed in STA are described in Chapter 8. Modeling IO timing across variety of interfaces is the topic of Chapter 9. And finally, Chapter 10 dwells on advanced timing checks such as on-chip variations, clock gating checks, power management and statistical timing analysis. Appendices provide detailed descriptions of SDC (used to represent timing constraints), SDF (used to represent delays of cell and nets) and SPEF (used to represent parasitics).

Chapters 7 through 10 provide the heart of STA verification. The preceding chapters provide a solid foundation and a detailed description of the *nuts and bolts* knowledge needed for a better understanding of STA.

❑

CHAPTER

2

STA Concepts

This chapter describes the basics of CMOS technology and the terminology involved in performing static timing analysis.

2.1 CMOS Logic Design

2.1.1 Basic MOS Structure

The physical implementation of MOS transistors (NMOS[1] and PMOS[2]) is depicted in Figure 2-1. The separation between the *source* and *drain* regions is the *length* of the MOS transistor. The smallest length used to build a MOS

1. N-channel Metal Oxide Semiconductor.
2. P-channel Metal Oxide Semiconductor.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_2,

transistor is normally the smallest feature size for the CMOS technology process. For example, a 0.25µm technology allows MOS transistors with a channel length of 0.25µm or larger to be fabricated. By shrinking the channel geometry, the transistor size becomes smaller, and subsequently more transistors can be packed in a given area. As we shall see later in this chapter, this also allows the designs to operate at a greater speed.

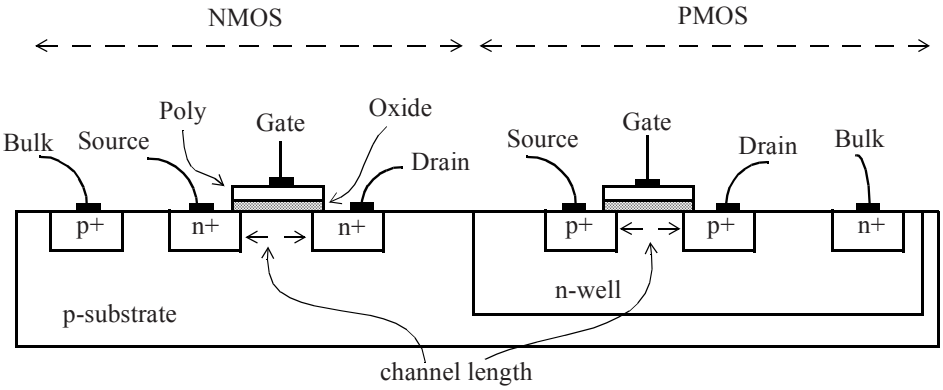


Figure 2-1 *Structure of NMOS and PMOS transistors.*

2.1.2 CMOS Logic Gate

A CMOS logic gate is built using NMOS and PMOS transistors. Figure 2-2 shows an example of a CMOS inverter. There are two stable states of the CMOS inverter depending upon the state of the input. When input *A* is low (at *Vss* or logic-0), the NMOS transistor is *off* and the PMOS transistor is *on*, causing the output *Z* to be pulled to *Vdd*, which is a logic-1. When input *A* is high (at *Vdd* or logic-1), the NMOS transistor is *on* and the PMOS transistor is *off*, causing the output *Z* to be pulled to *Vss*, which is a logic-0. In either of the two states described above, the CMOS inverter is stable and

does not draw any current[1] from the input *A* or from the power supply *Vdd*.

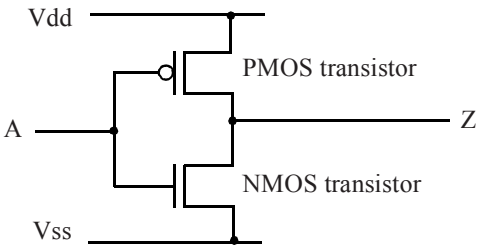


Figure 2-2 *A CMOS inverter.*

The characteristics of the CMOS inverter can be extended to any CMOS logic gate. In a CMOS logic gate, the output node is connected by a pull-up structure (made up of PMOS transistors) to *Vdd* and a pull-down structure (made up of NMOS transistors) to *Vss*. As an example, a two-input CMOS *nand* gate is shown in Figure 2-3. In this example, the pull-up structure is comprised of the two parallel PMOS transistors and the pull-down structure is made up of two series NMOS transistors.

For any CMOS logic gate, the pull-up and pull-down structures are complementary. For inputs at logic-0 or logic-1, this means that if the pull-up stage is turned *on*, the pull-down stage will be *off* and similarly if the pull-up stage is turned *off*, the pull-down stage will be turned *on*. The pull-down and pull-up structures are governed by the logic function implemented by the CMOS gate. For example, in a CMOS *nand* gate, the function controlling the pull-down structure is "*A and B*", that is, the pull-down is turned *on* when *A* and *B* are both at logic-1. Similarly, the function controlling the pull-up structure is "*not A or not B*", that is, the pull-up is turned *on* when either *A* or *B* is at logic-0. These characteristics ensure that the output node logic will be pulled to *Vdd* based upon the function controlling the pull-up structure. Since the pull-down structure is controlled by a comple-

1. Depending upon the specifics of the CMOS technology, there is a small amount of leakage current that is drawn even in steady state.

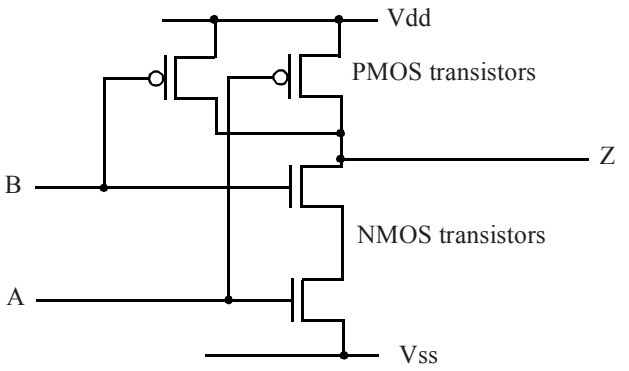


Figure 2-3 *CMOS two-input NAND gate.*

mentary function, the output node is at logic-0 when the pull-up structure function evaluates to 0.

For inputs at logic-0 or at logic-1, the CMOS logic gate does not draw any current from the inputs or from the power supply in steady state since the pull-up and pull-down structures can not both be *on*[1]. Another important aspect of CMOS logic is that the inputs pose only a capacitive load to the previous stage.

The CMOS logic gate is an inverting gate which means that a single switching input (rising or falling) can only cause the output to switch in the opposite direction, that is, the output can not switch in the same direction as the switching input. The CMOS logic gates can however be cascaded to put together a more complex logic function - inverting as well as non-inverting.

2.1.3 Standard Cells

Most of the complex functionality in a chip is normally designed using basic building blocks which implement simple logic functions such as *and*, *or*, *nand*, *nor*, a*nd-or-invert*, *or-and-invert* and *flip-flop*. These basic building

1. The pull-up and pull-down structures are both *on* only during switching.

blocks are pre-designed and referred to as **standard cells**. The functionality and timing of the standard cells is pre-characterized and available to the designer. The designer can then implement the required functionality using the standard cells as the building blocks.

The key characteristics of the CMOS logic gates described in previous subsection are applicable to all CMOS digital designs. All digital CMOS cells are designed such that there is no current drawn from power supply (except for leakage) when the inputs are in a stable logic state. Thus, most of the power dissipation is related to the activity in the design and is caused by the charging and discharging of the inputs of CMOS cells in the design.

What is a logic-1 or a logic-0? In a CMOS cell, two values *VIHmin* and *VILmax* define the limits. That is, any voltage value above *VIHmin* is considered as a **logic-1** and any voltage value below *VILmax* is considered as a **logic-0**. See Figure 2-4. Typical values for a CMOS 0.13µm inverter cell with 1.2V *Vdd* supply are 0.465V for *VILmax* and 0.625V for *VIHmin*. The *VIHmin* and *VILmax* values are derived from the DC transfer characteristics of the cell. The DC transfer characteristics are described in greater detail in Section 6.2.3.

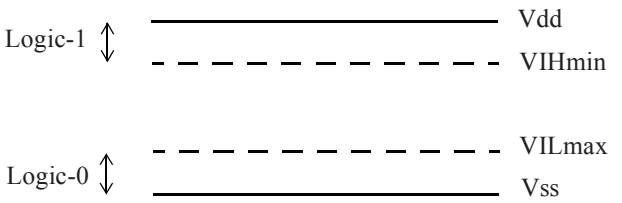


Figure 2-4 *CMOS logic levels.*

For more details on CMOS technology, refer to one of the relevant texts listed in the Bibliography.

2.2 Modeling of CMOS Cells

If a cell output pin drives multiple fanout cells, the total capacitance on the output pin of the cell is the sum of all the input capacitances of the cells that it is driving plus the sum of the capacitance of all the wire segments that comprise the net plus the output capacitance of the driving cell. Note that in a CMOS cell, the inputs to the cell present a capacitive load only.

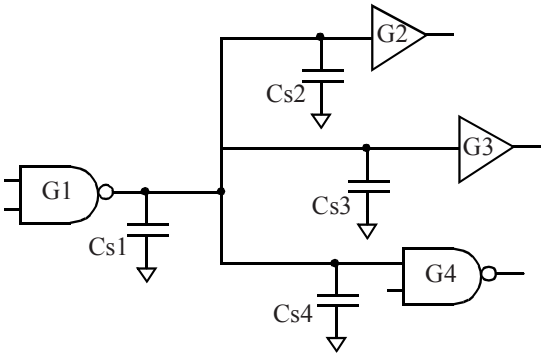


Figure 2-5 *Capacitance on a net.*

Figure 2-5 shows an example of a cell *G1* driving three other cells *G2*, *G3*, and *G4*. *Cs1*, *Cs2*, *Cs3* and *Cs4* are the capacitance values of wire segments that comprise the net. Thus:

```
Total cap (Output G1) = Cout(G1) + Cin(G2) + Cin(G3) +
                        Cin(G4) + Cs1 + Cs2 + Cs3 + Cs4
# Cout is the output pin capacitance of the cell.
# Cin is the input pin capacitance of the cell.
```

This is the capacitance that needs to be charged or discharged when cell *G1* switches and thus this total capacitance value impacts the timing of cell *G1*.

From a timing perspective, we need to model the CMOS cell to aid us in analyzing the timing through the cell. An input pin capacitance is specified

for each of the input pins. There can also be an output pin capacitance though most CMOS logic cells do not include the pin capacitance for the output pins.

When output is a logic-1, the pull-up structure for the output stage is *on*, and it provides a path from the output to *Vdd*. Similarly, when the output is a logic-0, the pull-down structure for the output stage provides a path from the output to *Vss*. When the CMOS cell switches state, the speed of the switching is governed by how fast the capacitance on the output net can be charged or discharged. The capacitance on the output net (Figure 2-5) is charged and discharged through the pull-up and pull-down structures respectively. Note that the channel in the pull-up and pull-down structures poses resistances for the output charging and discharging paths. The charging and discharging path resistances are a major factor in determining the speed of the CMOS cell. The inverse of the pull-up resistance is called the **output high drive** of the cell. The larger the output pull-up structure, the smaller the pull-up resistance and the larger the output high drive of the cell. The larger output structures also mean that the cell is larger in area. The smaller the output pull-up structures, the cell is smaller in area, and its output high drive is also smaller. The same concept for the pull-up structure can be applied for the pull-down structure which determines the resistance of the pull-down path and **output low drive**. In general, the cells are designed to have similar drive strengths (both large or both small) for pull-up and pull-down structures.

The output drive determines the maximum capacitive load that can be driven. The maximum capacitive load determines the maximum number of fanouts, that is, how many other cells it can drive. A higher output drive corresponds to a lower output pull-up and pull-down resistance which allows the cell to charge and discharge a higher load at the output pin.

Figure 2-6 shows an equivalent abstract model for a CMOS cell. The objective of this model is to abstract the timing behavior of the cell, and thus only the input and output stages are modeled. This model does not capture the intrinsic delay or the electrical behavior of the cell.

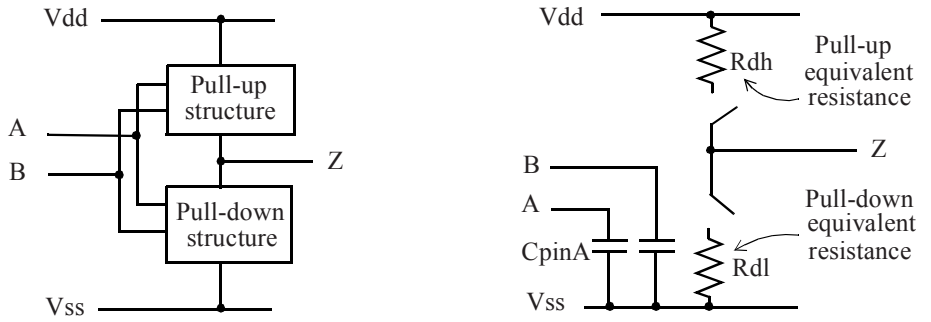


Figure 2-6 *CMOS cell and its electrically equivalent model.*

CpinA is the input pin capacitance of the cell on input *A*. *Rdh* and *Rdl* are the output drive resistances of the cell and determine the rise and fall times of the output pin *Z* based upon the load being driven by the cell. This drive also determines the maximum fanout limit of the cell.

Figure 2-7 shows the same net as in Figure 2-5 but with the equivalent models for the cells.

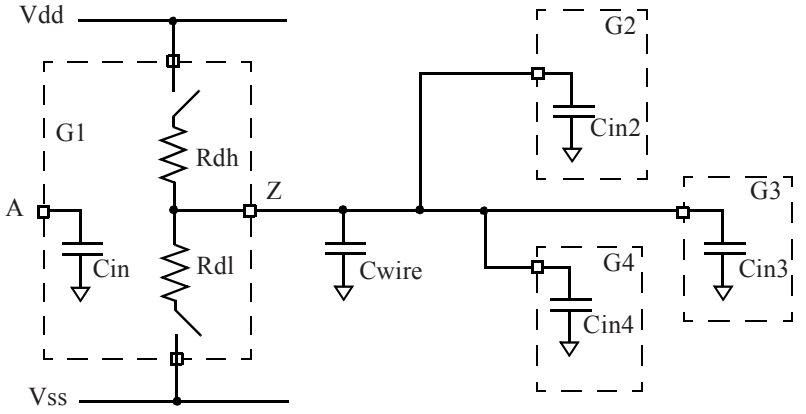


Figure 2-7 *Net with CMOS equivalent models.*

```
Cwire = Cs1 + Cs2 + Cs3 + Cs4
Output charging delay (for high or low) =
  Rout * (Cwire + Cin2 +Cin3 + Cin4)
```

In the above expression, *Rout* is one of *Rdh* or *Rdl* where *Rdh* is the output drive resistance for pull-up and *Rdl* is the output drive resistance for pull-down.

2.3 Switching Waveform

When a voltage is applied to the RC network as shown in Figure 2-8(a) by activating the *SW0* switch, the output goes to a logic-1. Assuming the output is at 0V when *SW0* is activated, the voltage transition at the output is described by the equation:

$$V = Vdd * [1 - e^{-t/(Rdh * Cload)}]$$

The voltage waveform for this rise is shown in Figure 2-8(b). The product *(Rdh * Cload)* is called the **RC time constant** - typically this is also related to the transition time of the output.

When the output goes from logic-1 to logic-0, caused by input changes disconnecting *SW0* and activating *SW1*, the output transition looks like the one shown in Figure 2-8(c). The output capacitance discharges through the *SW1* switch which is *on*. The voltage transition in this case is described by the equation:

$$V = Vdd * e^{-t/(Rdl * Cload)}$$

In a CMOS cell, the output charging and discharging waveforms do not appear like the RC charging and discharging waveforms of Figure 2-8 since the PMOS pull-up and the NMOS pull-down transistors are both *on* simultaneously for a brief amount of time. Figure 2-9 shows the *current* paths within a CMOS inverter cell for various stages of output switching from logic-1 to logic-0. Figure 2-9(a) shows the *current* flow when both the pull-

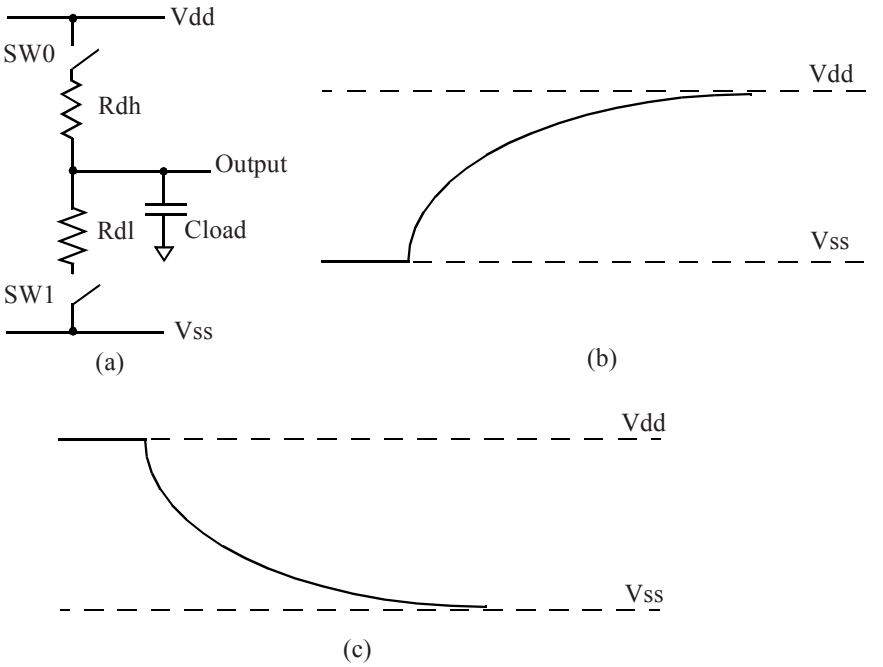
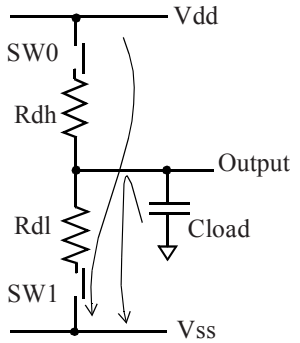


Figure 2-8 *RC charging and discharging waveforms.*

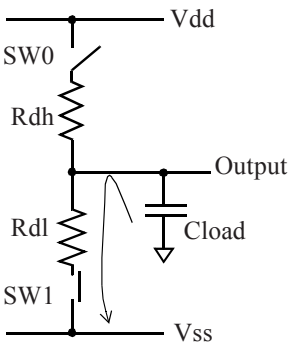
up and pull-down structures are *on*. Later, the pull-up structure turns *off* and the current flow is depicted in Figure 2-9(b). After the output reaches the final state, there is no current flow as the capacitance *Cload* is completely discharged.

Figure 2-10(a) shows a representative waveform at the output of a CMOS cell. Notice how the transition waveforms curve asymptotically towards the *Vss* rail and the *Vdd* rail, and the linear portion of the waveform is in the middle.

In this text, we shall depict some waveforms using a simplistic drawing as shown in Figure 2-10(b). It shows the waveform with some transition time, which is the time needed to transition from one logic state to the other. Fig-



(a) Cell is switching.
(Pull-up, pull-down both *on*)



(b) Cell is discharging to logic-0.
(Pull-up *off*, pull-down *on*)

Figure 2-9 *Current flow for a CMOS cell output stage.*

ure 2-10(c) shows the same waveforms using a transition time of 0, that is, as ideal waveforms. We shall be using both of these forms in this text interchangeably to explain the concepts, though in reality, each waveform has its real edge characteristics as shown in Figure 2-10(a).

2.4 Propagation Delay

Consider a CMOS inverter cell and its input and output waveforms. The **propagation delay** of the cell is defined with respect to some measurement points on the switching waveforms. Such points are defined using the following four variables:

```
# Threshold point of an input falling edge:
input_threshold_pct_fall : 50.0;
# Threshold point of an input rising edge:
input_threshold_pct_rise : 50.0;
# Threshold point of an output falling edge:
output_threshold_pct_fall : 50.0;
```

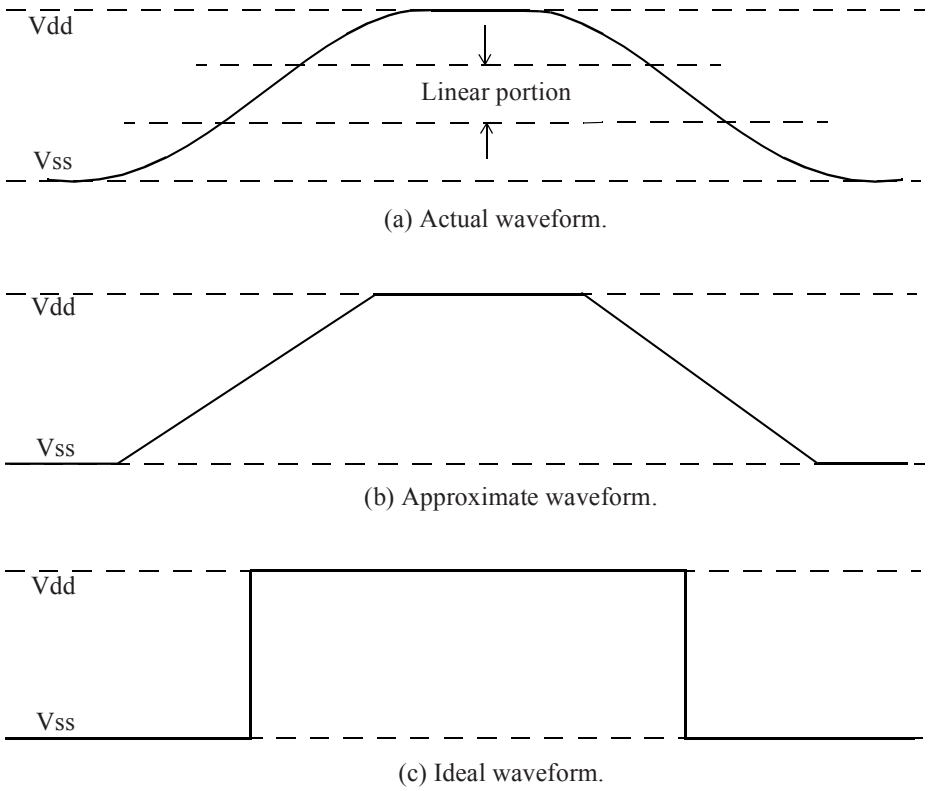


Figure 2-10 *CMOS output waveforms.*

```
# Threshold point of an output rising edge:
output_threshold_pct_rise : 50.0;
```

These variables are part of a command set used to describe a cell library (this command set is described in Liberty[1]). These threshold specifications are in terms of the percent of *Vdd*, or the power supply. Typically 50% threshold is used for delay measurement for most standard cell libraries.

1. See [LIB] in Bibliography.

Rising edge is the transition from logic-0 to logic-1. Falling edge is the transition from logic-1 to logic-0.

Consider the example inverter cell and the waveforms at its pins shown in Figure 2-11. The propagation delays are represented as:

i. Output fall delay (*Tf*)

ii. Output rise delay (*Tr*)

In general, these two values are different. Figure 2-11 shows how these two propagation delays are measured.

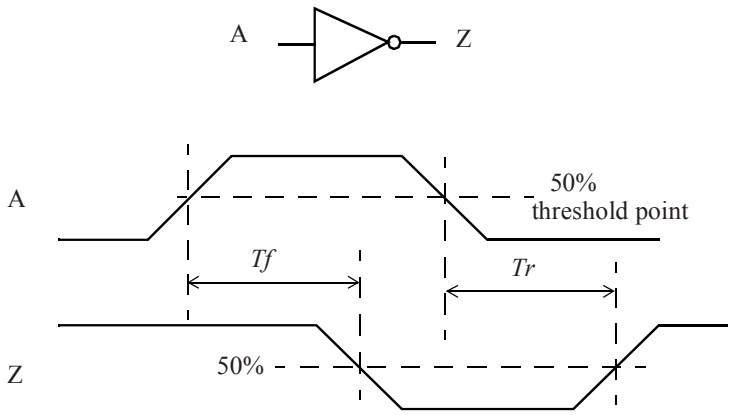


Figure 2-11 *Propagation delays.*

If we were looking at ideal waveforms, propagation delay would simply be the delay between the two edges. This is shown in Figure 2-12.

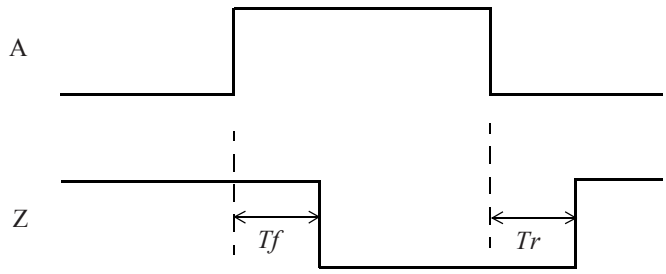


Figure 2-12 *Propagation delay using ideal waveforms.*

2.5 Slew of a Waveform

A slew rate is defined as a rate of change. In static timing analysis, the rising or falling waveforms are measured in terms of whether the transition is slow or fast. The slew is typically measured in terms of the **transition time**, that is, the time it takes for a signal to transition between two specific levels. Note that the transition time is actually inverse of the slew rate - the larger the transition time, the slower the slew, and vice versa. Figure 2-10 illustrates a typical waveform at the output of a CMOS cell. The waveforms at the ends are asymptotic and it is hard to determine the exact start and end points of the transition. Consequently, the transition time is defined with respect to specific threshold levels. For example, the slew threshold settings can be:

```
# Falling edge thresholds:
slew_lower_threshold_pct_fall : 30.0;
slew_upper_threshold_pct_fall : 70.0;
# Rising edge thresholds:
slew_lower_threshold_pct_rise : 30.0;
slew_upper_threshold_pct_rise : 70.0;
```

These values are specified as a percent of *Vdd*. The threshold settings specify that falling slew is the difference between the times that falling edge

reaches 70% and 30% of *Vdd*. Similarly, the settings for rise specify that the rise slew is the difference in times that the rising edge reaches 30% and 70% of *Vdd*. Figure 2-13 shows this pictorially.

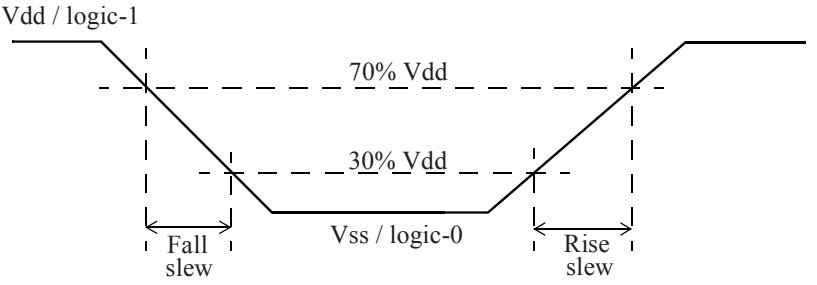


Figure 2-13 *Rise and fall transition times.*

Figure 2-14 shows another example where the slew on a falling edge is measured 20-80 (80% to 20%) and that on the rising edge is measured 10-90 (10% to 90%). Here are the threshold settings for this case.

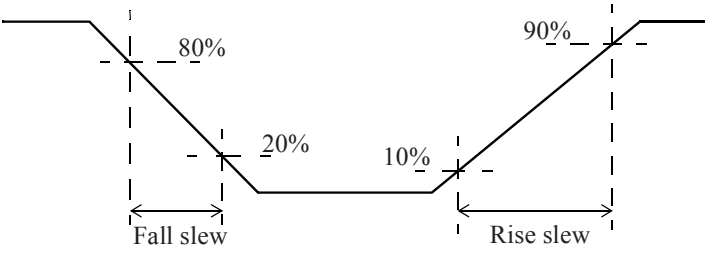


Figure 2-14 *Another example of slew measurement.*

```
# Falling edge thresholds:
slew_lower_threshold_pct_fall : 20.0;
slew_upper_threshold_pct_fall : 80.0;
```

```
# Rising edge thresholds:
slew_lower_threshold_pct_rise : 10.0;
slew_upper_threshold_pct_rise : 90.0;
```

2.6 Skew between Signals

Skew is the difference in timing between two or more signals, maybe data, clock or both. For example, if a clock tree has 500 end points and has a skew of 50ps, it means that the difference in latency between the longest path and the shortest clock path is 50ps. Figure 2-15 shows an example of a clock tree. The beginning point of a clock tree typically is a node where a clock is defined. The end points of a clock tree are typically clock pins of synchronous elements, such as flip-flops. **Clock latency** is the total time it takes from the clock source to an end point. **Clock skew** is the difference in arrival times at the end points of the clock tree.

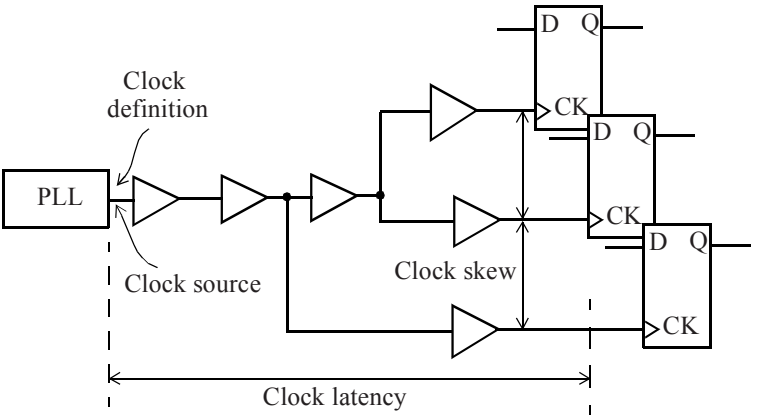


Figure 2-15 *Clock tree, clock latency and clock skew.*

An **ideal clock tree** is one where the clock source is assumed to have an infinite drive, that is, the clock can drive infinite sources with no delay. In addition, any cells present in the clock tree are assumed to have zero delay. In

the early stages of logical design, STA is often performed with ideal clock trees so that the focus of the analysis is on the data paths. In an ideal clock tree, clock skew is 0ps by default. Latency of a clock tree can be explicitly specified using the **set_clock_latency** command. The following example models the latency of a clock tree:

```
set_clock_latency 2.2 [get_clocks BZCLK]
  # Both rise and fall latency is 2.2ns.
  # Use options -rise and -fall if different.
```

Clock skew for a clock tree can also be implied by explicitly specifying its value using the **set_clock_uncertainty** command:

```
set_clock_uncertainty 0.250 -setup [get_clocks BZCLK]
set_clock_uncertainty 0.100 -hold [get_clocks BZCLK]
```

The *set_clock_uncertainty* specifies a window within which a clock edge can occur. The uncertainty in the timing of the clock edge is to account for several factors such as clock period jitter and additional margins used for timing verification. Every real clock source has a finite amount of jitter - a window within which a clock edge can occur. The clock period jitter is determined by the type of clock generator utilized. In reality, there are no ideal clocks, that is, all clocks have a finite amount of jitter and the clock period jitter should be included while specifying the clock uncertainty.

Before the clock tree is implemented, the clock uncertainty must also include the expected clock skew of the implementation.

One can specify different clock uncertainties for setup checks and for hold checks. The hold checks do not require the clock jitter to be included in the uncertainty and thus a smaller value of clock uncertainty is generally specified for hold.

Figure 2-16 shows an example of a clock with a setup uncertainty of 250ps. Figure 2-16(b) shows how the uncertainty takes away from the time avail-

able for the logic to propagate to the next flip-flop stage. This is equivalent to validating the design to run at a higher frequency.

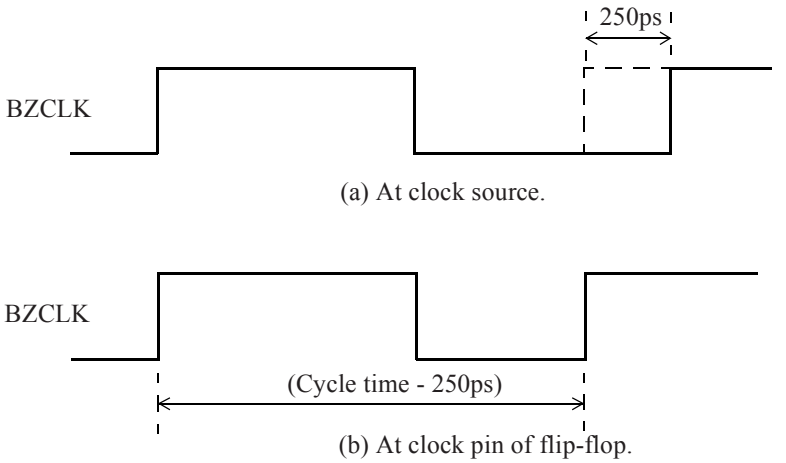


Figure 2-16 *Clock setup uncertainty.*

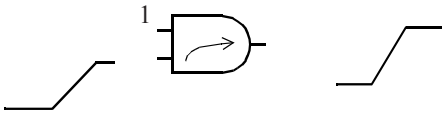
As specified above, the *set_clock_uncertainty* can also be used to model any additional margin. For example, a designer may use a 50ps timing margin as additional pessimism during design. This component can be added and included in the *set_clock_uncertainty* command. In general, before the clock tree is implemented, the *set_clock_uncertainty* command is used to specify a value that includes clock jitter plus estimated clock skew plus additional pessimism.

```
set_clock_latency 2.0 [get_clocks USBCLK]
set_clock_uncertainty 0.2 [get_clocks USBCLK]
  # The 200ps may be composed of 50ps clock jitter,
  # 100ps clock skew and 50ps additional pessimism.
```

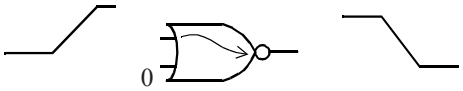
We shall see later how *set_clock_uncertainty* influences setup and hold checks. It is best to think of clock uncertainty as an offset to the final slack calculation.

2.7 Timing Arcs and Unateness

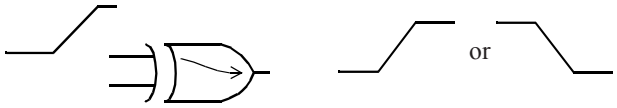
Every cell has multiple timing arcs. For example, a combinational logic cell, such as *and*, *or*, *nand*, *nor*, *adder* cell, has timing arcs from each input to each output of the cell. Sequential cells such as flip-flops have timing arcs from the clock to the outputs and timing constraints for the data pins with respect to the clock. Each timing arc has a timing sense, that is, how the output changes for different types of transitions on input. The timing arc is **positive unate** if a rising transition on an input causes the output to rise (or not to change) and a falling transition on an input causes the output to fall (or not to change). For example, the timing arcs for *and* and *or* type cells are positive unate. See Figure 2-17(a).



(a) Positive unate arc.



(b) Negative unate arc.



(c) Non-unate arc.

Figure 2-17 *Timing sense of arcs.*

A **negative unate** timing arc is one where a rising transition on an input causes the output to have a falling transition (or not to change) and a fall-

ing transition on an input causes the output to have a rising transition (or not to change). For example, the timing arcs for *nand* and *nor* type cells are negative unate. See Figure 2-17(b).

In a **non-unate** timing arc, the output transition cannot be determined solely from the direction of change of an input but also depends upon the state of the other inputs. For example, the timing arcs in an *xor* cell (exclusive-or) are non-unate.[1] See Figure 2-17(c).

Unateness is important for timing as it specifies how the edges (transitions) can propagate through a cell and how they appear at the output of the cell.

One can take advantage of the non-unateness property of a timing arc, such as when an *xor* cell is used, to invert the polarity of a clock. See the example in Figure 2-18. If input *POLCTRL* is a logic-0, the clock *DDRCLK* on output of the cell *UXOR0* has the same polarity as the input clock *MEMCLK*. If *POLCTRL* is a logic-1, the clock on the output of the cell *UXOR0* has the opposite polarity as the input clock *MEMCLK*.

2.8 Min and Max Timing Paths

The total delay for the logic to propagate through a logic path is referred to as the **path delay**. This corresponds to the sum of the delays through the various logic cells and nets along the path. In general, there are multiple paths through which the logic can propagate to the required destination point. The actual path taken depends upon the state of the other inputs along the logic path. An example is illustrated in Figure 2-19. Since there are multiple paths to the destination, the maximum and minimum timing to the destination points can be obtained. The paths corresponding to the maximum timing and minimum timing are referred to as the max path and min path respectively. A **max path** between two end points is the path with the largest delay (also referred to as the **longest path**). Similarly, a **min**

1. It is possible to specify state-dependent timing arcs for an *xor* cell which are positive unate and negative unate. This is described in Chapter 3.

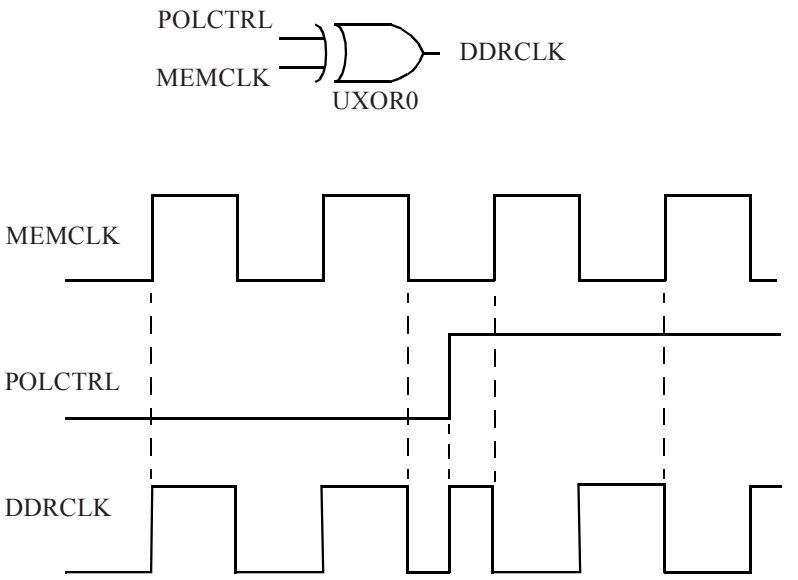


Figure 2-18 *Controlling clock polarity using non-unate cell.*

path is the path with the smallest delay (also referred to as the **shortest path**). Note that the longest and shortest refer to the cumulative delay of the path, not to the number of cells in the path.

Figure 2-19 shows an example of a data path between flip-flops. A max path between flip-flops *UFF1* and *UFF3* is assumed to be the one that goes through *UNAND0*, *UBUF2*, *UOR2* and *UNAND6* cells. A min path between the flip-flops *UFF1* and *UFF3* is assumed to be the one that goes through the *UOR4* and *UNAND6* cells. Note that in this example, the max and min are with reference to the destination point which is the *D* pin of the flip-flop *UFF3*.

A max path is often called a **late path**, while a min path is often called an **early path**.

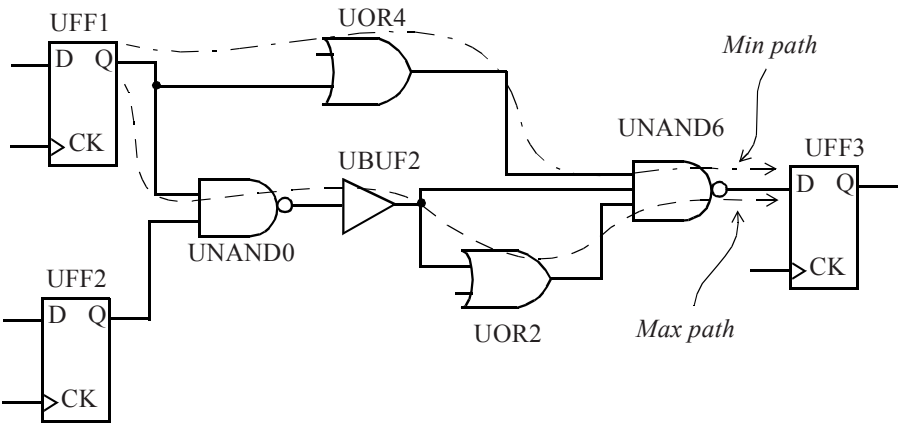


Figure 2-19 *Max and min timing paths.*

When a flip-flop to flip-flop path such as from *UFF1* to *UFF3* is considered, one of the flip-flops launches the data and the other flip-flop captures the data. In this case, since *UFF1* launches the data, *UFF1* is referred to as the **launch** flip-flop. And since *UFF3* captures the data, *UFF3* is referred to as the **capture** flip-flop. Notice that the launch and capture terminology are always with reference to a flip-flop to flip-flop path. For example, *UFF3* would become a launch flip-flop for the path to whatever flip-flop captures the data produced by *UFF3*.

2.9 Clock Domains

In synchronous logic design, a periodic clock signal latches the new data computed into the flip-flops. The new data inputs are based upon the flip-flop values from a previous clock cycle. The latched data thus gets used for computing the data for the next clock cycle.

A clock typically feeds a number of flip-flops. The set of flip-flops being fed by one clock is called its **clock domain**. In a typical design, there may be

more than one clock domain. For example, 200 flip-flops may be clocked by *USBCLK* and 1000 flip-flops may be fed by clock *MEMCLK*. Figure 2-20 depicts the flip-flops along with the clocks. In this example, we say that there are two clock domains.

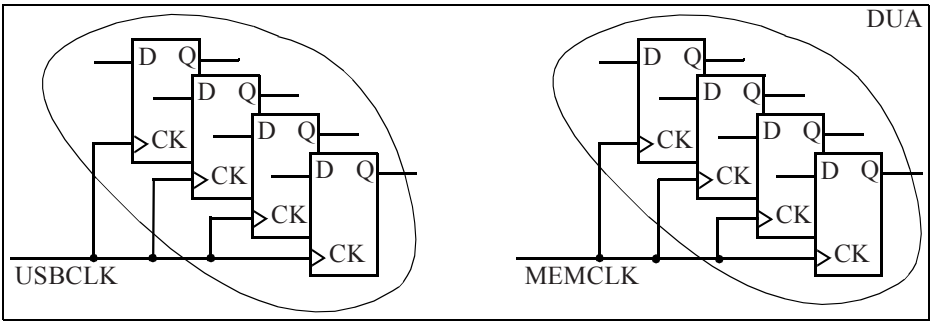


Figure 2-20 *Two clock domains.*

A question of interest is whether the clock domains are related or independent of each other. The answer depends on whether there are any data paths that start from one clock domain and end in the other clock domain. If there are no such paths, we can safely say that the two clock domains are independent of each other. This means that there is no timing path that starts from one clock domain and ends in the other clock domain.

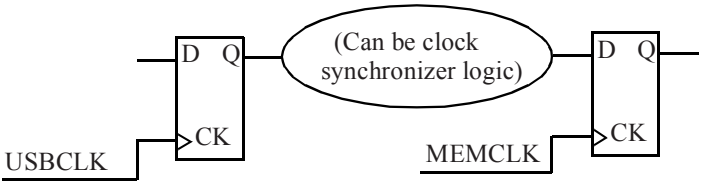


Figure 2-21 *Clock domain crossing.*

If indeed there are data paths that cross between clock domains (see Figure 2-21), a decision has to be made as to whether the paths are real or not. An example of a real path is a flip-flop with a 2x speed clock driving into a flip-flop with a 1x speed clock. An example of a false path is where the designer has explicitly placed clock synchronizer logic between the two clock domains. In this case, even though there appears to be a timing path from one clock domain to the next, it is not a real timing path since the data is not constrained to propagate through the synchronizer logic in one clock cycle. Such a path is referred to as a false path - not real - because the clock synchronizer ensures that the data passes correctly from one domain to the next. False paths between clock domains can be specified using the *set_false_path* specification, such as:

```
set_false_path -from [get_clocks USBCLK] \
  -to [get_clocks MEMCLK]
# This specification is explained in more detail in Chapter 8.
```

Even though it is not depicted in Figure 2-21, a clock domain crossing can occur both ways, from *USBCLK* clock domain to *MEMCLK* clock domain, and from *MEMCLK* clock domain to *USBCLK* clock domain. Both scenarios need to be understood and handled properly in STA.

What is the reason to discuss paths between clock domains? Typically a design has a large number of clocks and there can be a myriad number of paths between the clock domains. Identifying which clock domain crossings are real and which clock crossings are not real is an important part of the timing verification effort. This enables the designer to focus on validating only the real timing paths.

Figure 2-22 shows another example of clock domains. A multiplexer selects a clock source - it is either one or the other depending on the mode of operation of the design. There is only one clock domain, but two clocks, and these two clocks are said to be mutually-exclusive, as only one clock is active at one time. Thus, in this example, it is important to note that there can never be a path between the two clock domains for *USBCLK* and *USBCLKx2* (assuming that the multiplexer control is static and that such paths do not exist elsewhere in the design).

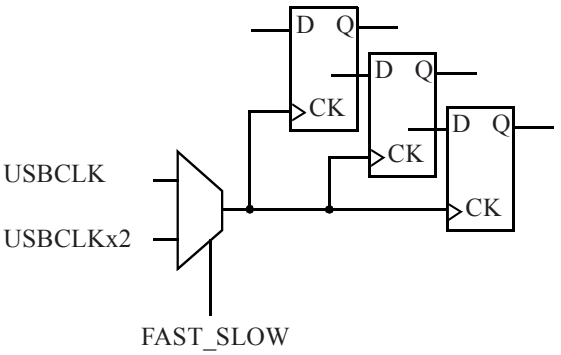


Figure 2-22 *Mutually-exclusive clocks.*

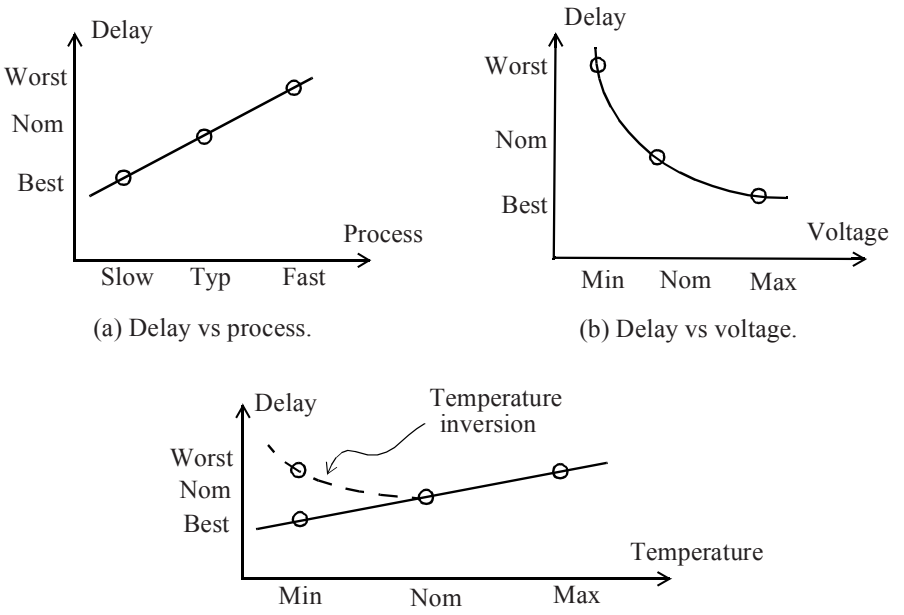
2.10 Operating Conditions

Static timing analysis is typically performed at a specific operating condition[1]. An operating condition is defined as a combination of *Process*, *Voltage* and *Temperature* (PVT). Cell delays and interconnect delays are computed based upon the specified operating condition.

There are three kinds of manufacturing process models that are provided by the semiconductor foundry for digital designs: *slow* process models, *typical* process models, and *fast* process models. The *slow* and *fast* process models represent the extreme corners of the manufacturing process of a foundry. For robust design, the design is validated at the extreme corners of the manufacturing process as well as environment extremes for temperature and power supply. Figure 2-23(a) shows how a cell delay changes with the process corners. Figure 2-23(b) shows how cell delays vary with power supply voltage, and Figure 2-23(c) shows how cell delays can vary

1. STA may be performed on a design with cells that have different voltages. We shall see later how these are handled. STA can also be performed statistically, which is described in Chapter 10.

with temperature. Thus it is important to decide the operating conditions that should be used for various static timing analyses.



(a) Delay vs process.

(b) Delay vs voltage.

(c) Delay vs temperature.

Figure 2-23 *Delay variations with PVT.*

The choice of what operating condition to use for STA is also governed by the operating conditions under which cell libraries are available. Three standard operating conditions are:

i. *WCS (Worst-Case Slow)*: Process is *slow*, temperature is highest (say 125C) and voltage is lowest (say nominal 1.2V minus 10%). For nanometer technologies that use low power supplies, there can be another worst-case slow corner that corresponds to the *slow* process, lowest power supply, and lowest temperature. The delays at low temperatures are not always smaller than the de-

lays at higher temperatures. This is because the device threshold voltage (Vt) margin with respect to the power supply is reduced for nanometer technologies. In such cases, at low power supply, the delay of a lightly loaded cell is higher at low temperatures than at high temperatures. This is especially true of high Vt (higher threshold, larger delay) or even standard Vt (regular threshold, lower delay) cells. This anomalous behavior of delays increasing at lower temperatures is called *temperature inversion*. See Figure 2-23(c).

ii. *TYP (Typical)*: Process is *typical*, temperature is nominal (say 25C) and voltage is nominal (say 1.2V).

iii. *BCF (Best-Case Fast)*: Process is *fast*, temperature is lowest (say -40C) and voltage is highest (say nominal 1.2V plus 10%).

The environment conditions for power analysis are generally different than the ones used for static timing analysis. For power analysis, the operating conditions may be:

i. *ML (Maximal Leakage)*: Process is *fast*, temperature is highest (say 125C) and the voltage is also the highest (say 1.2V plus 10%). This corner corresponds to the maximum leakage power. For most designs, this corner also corresponds to the largest active power.

ii. *TL (Typical Leakage)*: Process is *typical*, temperature is highest (say 125C) and the voltage is nominal (say 1.2V). This refers to the condition where the leakage is representative for most designs since the chip temperature will be higher due to power dissipated in normal operation.

The static timing analysis is based on the libraries that are loaded and linked in for the STA. An operating condition for the design can be explicitly specified using the **set_operating_conditions** command.

```
set_operating_conditions "WCCOM" -library mychip
# Use the operating condition called WCCOM defined in the
# cell library mychip.
```

The cell libraries are available at various operating conditions and the operating condition chosen for analysis depends on what has been loaded for the STA.

❑

CHAPTER

3

Standard Cell Library

This chapter describes timing information present in library cell descriptions. A cell could be a standard cell, an IO buffer, or a complex IP such as a USB core.

In addition to timing information, the library cell description contains several attributes such as cell area and functionality, which are unrelated to timing but are relevant during the RTL synthesis process. In this chapter, we focus only on the attributes relevant to the timing and power calculations.

A library cell can be described using various standard formats. While the content of various formats is essentially similar, we have described the library cell examples using the Liberty syntax.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_3, © Springer Science + Business Media, LLC 2009

The initial sections in this chapter describe the linear and the non-linear timing models followed by advanced timing models for nanometer technologies which are described in Section 3.7.

3.1 Pin Capacitance

Every input and output of a cell can specify capacitance at the pin. In most cases, the capacitance is specified only for the cell inputs and not for the outputs, that is, the output pin capacitance in most cell libraries is 0.

```
pin (INP1) {
  capacitance: 0.5;
  rise_capacitance: 0.5;
  rise_capacitance_range: (0.48, 0.52);
  fall_capacitance: 0.45;
  fall_capacitance_range: (0.435, 0.46);
  . . .
}
```

The above example shows the general specification for the pin capacitance values for the input *INP1*. In its most basic form, the pin capacitance is specified as a single value (0.5 units in above example). (The capacitance unit is normally picofarad and is specified in the beginning of the library file). The cell description can also specify separate values for *rise_capacitance* (0.5 units) and *fall_capacitance* (0.45 units) which refer to the values used for rising and falling transitions at the pin *INP1*. The *rise_capacitance* and *fall_capacitance* values can also be specified as a range with the lower and upper bound values being specified in the description.

3.2 Timing Modeling

The cell timing models are intended to provide accurate timing for various instances of the cell in the design environment. The timing models are nor-

mally obtained from detailed circuit simulations of the cell to model the actual scenario of the cell operation. The timing models are specified for each timing arc of the cell.

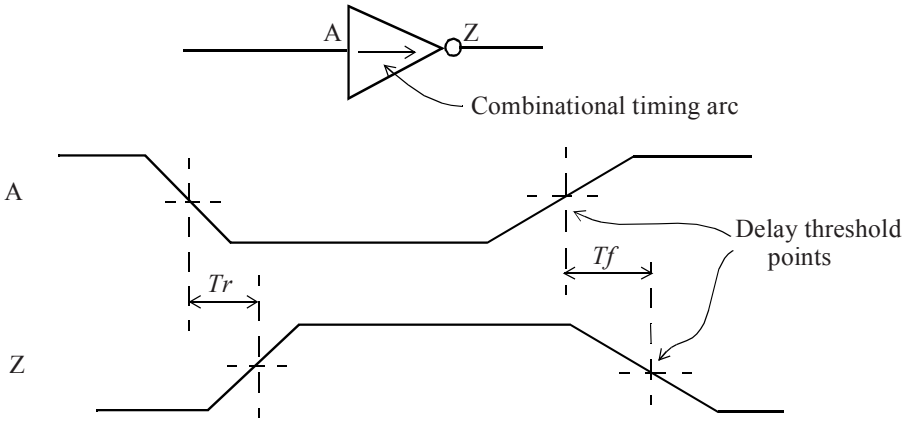


Figure 3-1 *Timing arc delays for an inverter cell.*

Let us first consider timing arcs for a simple inverter logic cell shown in Figure 3-1. Since it is an inverter, a rising (falling) transition at the input causes a falling (rising) transition at the output. The two kinds of delay characterized for the cell are:

- *Tr* : Output rise delay
- *Tf* : Output fall delay

Notice that the delays are measured based upon the threshold points defined in a cell library (see Section 2.4), which is typically 50% *Vdd*. Thus, delays are measured from input crossing its threshold point to the output crossing its threshold point.

The delay for the timing arc through the inverter cell is dependent on two factors:

i. the output load, that is, the capacitance load at the output pin of the inverter, and

ii. the transition time of the signal at the input.

The delay values have a direct correlation with the load capacitance - the larger the load capacitance, the larger the delay. In most cases, the delay increases with increasing input transition time. There are a few scenarios where the input threshold (used for measuring delay) is significantly different from the internal switching point of the cell. In such cases, the delay through the cell may show non-monotonic behavior with respect to the input transition time - a larger input transition time may produce a smaller delay especially if the output is lightly loaded.

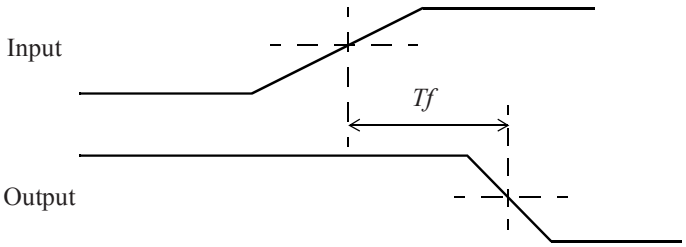
The slew at the output of a cell depends mainly upon the output capacitance - output transition time increases with output load. Thus, a large slew at the input (large transition time) can improve at the output depending upon the cell type and its output load. Figure 3-2 shows cases where the transition time at the output of a cell can improve or deteriorate depending on the load at the output of the cell.

3.2.1 Linear Timing Model

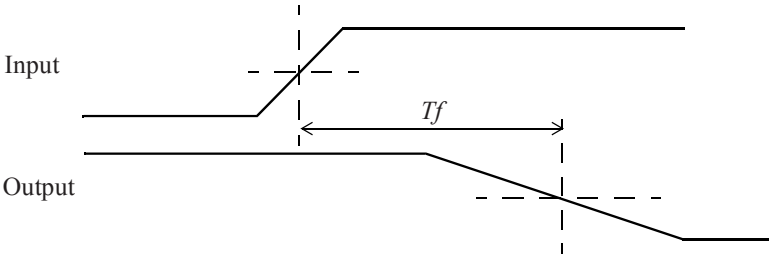
A simple timing model is a *linear delay model*, where the delay and the output transition time of the cell are represented as linear functions of the two parameters: input transition time and the output load capacitance. The general form of the linear model for the delay, *D*, through the cell is illustrated below.

```
D = D0 + D1 * S + D2 * C
```

where *D0*, *D1*, *D2* are constants, *S* is the input transition time, and *C* is the output load capacitance. The linear delay models are not accurate over the range of input transition time and output capacitance for submicron tech-



(a) Slew improves.



(b) Slew degrades (due to large output load).

Figure 3-2 *Slew changes going through a cell.*

nologies, and thus most cell libraries presently use the more complex models such as the non-linear delay model.

3.2.2 Non-Linear Delay Model

Most of the cell libraries include table models to specify the delays and timing checks for various timing arcs of the cell. Some newer timing libraries for nanometer technologies also provide current source based advanced timing models (such as CCS, ECSM, etc.) which are described later in this chapter. The table models are referred to as **NLDM** (**N**on-**L**inear **D**elay **M**odel) and are used for delay, output slew, or other timing checks. The table models capture the delay through the cell for various combinations of

input transition time at the cell input pin and total output capacitance at the cell output.

An NLDM model for delay is presented in a two-dimensional form, with the two independent variables being the input transition time and the output load capacitance, and the entries in the table denoting the delay. Here is an example of such a table for a typical inverter cell:

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP1";
    timing_sense : negative_unate;
    cell_rise(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values (  /*  0.16      0.35     1.43 */ \
        /* 0.1 */   "0.0513,  0.1537,  0.5280", \
        /* 0.3 */   "0.1018,  0.2327,  0.6476", \
        /* 0.7 */   "0.1334,  0.2973,  0.7252");
      }
    cell_fall(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values (  /*  0.16      0.35      1.43 */ \
        /* 0.1 */   "0.0617,  0.1537,   0.5280", \
        /* 0.3 */   "0.0918,  0.2027,   0.5676", \
        /* 0.7 */   "0.1034,  0.2273,   0.6452");
    }
```

In the above example, the delays of the output pin *OUT* are described. This portion of the cell description contains the rising and falling delay models for the timing arc from pin *INP1* to pin *OUT*, as well as the *max_transition* allowed time at pin *OUT*. There are separate models for the rise and fall delays (for the output pin) and these are labeled as *cell_rise* and *cell_fall* respectively. The type of indices and the order of table lookup indices are described in the lookup table template *delay_template_3x3*.

```
lu_table_template(delay_template_3x3) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("1000, 1001, 1002");
  index_2 ("1000, 1001, 1002");
}/* The input transition and the output capacitance can be
    in either order, that is, variable_1 can be the output
    capacitance. However, these designations are usually
    consistent across all templates in a library. */
```

This lookup table template specifies that the first variable in the table is the input transition time and the second variable is the output capacitance. The table values are specified like a nested loop with the first index (*index_1*) being the outer (or least varying) variable and the second index (*index_2*) being the inner (or most varying) variable and so on. There are three entries for each variable and thus it corresponds to a 3-by-3 table. In most cases, the entries for the table are also formatted like a table and the first index (*index_1*) can then be treated as a row index and the second index (*index_2*) becomes equivalent to the column index. The index values (for example 1000) are dummy placeholders which are overridden by the actual index values in the *cell_fall* and *cell_rise* delay tables. An alternate way of specifying the index values is to specify the index values in the template definition and to not specify them in the *cell_rise* and *cell_fall* tables. Such a template would look like this:

```
lu_table_template(delay_template_3x3) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("0.1, 0.3, 0.7");
  index_2 ("0.16, 0.35, 1.43");
}
```

Based upon the delay tables, an input fall transition time of 0.3ns and an output load of 0.16pf will correspond to the rise delay of the inverter of 0.1018ns. Since a falling transition at the input results in the inverter output

rise, the table lookup for the rise delay involves a falling transition at the inverter input.

This form of representing delays in a table as a function of two variables, transition time and capacitance, is called the *non-linear* delay model, since non-linear variations of delay with input transition time and load capacitance are expressed in such tables.

The table models can also be 3-dimensional - an example is a flip-flop with complementary outputs, *Q* and *QN*, which is described in Section 3.8.

The NLDM models are used not only for the delay but also for the transition time at the output of a cell which is characterized by the input transition time and the output load. Thus, there are separate two-dimensional tables for computing the output rise and fall transition times of a cell.

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP";
    timing_sense : negative_unate;
    rise_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values ( /*   0.16      0.35     1.43 */ \
        /* 0.1 */  "0.0417,  0.1337,  0.4680", \
        /* 0.3 */  "0.0718,  0.1827,  0.5676", \
        /* 0.7 */  "0.1034,  0.2173,  0.6452");
    }
    fall_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7"); /* Input transition */
      index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
      values ( /*    0.16      0.35     1.43 */ \
        /* 0.1 */   "0.0817,  0.1937,  0.7280", \
        /* 0.3 */   "0.1018,  0.2327,  0.7676", \
        /* 0.7 */   "0.1334,  0.2973,  0.8452");
    }
```

```
      . . .
    }
    . . .
  }
```

There are two such tables for transition time: *rise_transition* and *fall_transition*. As described in Chapter 2, the transition times are measured based on the specific slew thresholds, usually 10%-90% of the power supply.

As illustrated above, an inverter cell with an NLDM model has the following tables:

- Rise delay
- Fall delay
- Rise transition
- Fall transition

Given the input transition time and output capacitance of such a cell, as shown in Figure 3-3, the rise delay is obtained from the *cell_rise* table for 15ps input transition time (falling) and 10fF load, and the fall delay is obtained from the *cell_fall* table for 20ps input transition time (rising) and 10fF load.

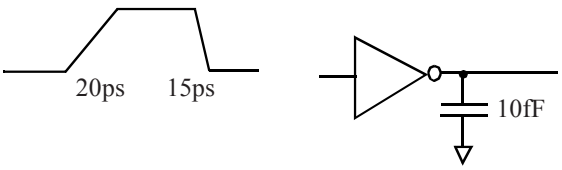


Figure 3-3 *Transition time and capacitance for computing cell delays.*

Where is the information which specifies that the cell is inverting? This information is specified as part of the *timing_sense* field of the timing arc. In some cases, this field is not specified but is expected to be derived from the pin function.

For the example inverter cell, the timing arc is *negative_unate* which implies that the output pin transition direction is opposite (negative) of the input pin transition direction. Thus, the *cell_rise* table lookup corresponds to the falling transition time at the input pin.

Example of Non-Linear Delay Model Lookup

This section illustrates the lookup of the table models through an example. If the input transition time and the output capacitance correspond to a table entry, the table lookup is trivial since the timing value corresponds directly to the value in the table. The example below corresponds to a general case where the lookup does not correspond to any of the entries available in the table. In such cases, two-dimensional interpolation is utilized to provide the resulting timing value. The two nearest table indices in each dimension are chosen for the table interpolation. Consider the table lookup for fall transition (example table specified above) for the input transition time of 0.15ns and an output capacitance of 1.16pF. The corresponding section of the fall transition table relevant for two-dimensional interpolation is reproduced below.

```
fall_transition(delay_template_3x3) {
  index_1 ("0.1, 0.3 . . .");
  index_2 (". . . 0.35, 1.43");
  values ( \
    ". . . 0.1937, 0.7280", \
    ". . . 0.2327, 0.7676"
    . . .
```

In the formulation below, the two *index_1* values are denoted as x_1 and x_2; the two *index_2* values are denoted as y_1 and y_2 and the corresponding table values are denoted as T_{11}, T_{12}, T_{21} and T_{22} respectively.

If the table lookup is required for (x_0, y_0), the lookup value T_{00} is obtained by interpolation and is given by:

$$T_{00} = x_{20} * y_{20} * T_{11} + x_{20} * y_{01} * T_{12} + x_{01} * y_{20} * T_{21} + x_{01} * y_{01} * T_{22}$$

where

$$x_{01} = (x_0 - x_1) / (x_2 - x_1)$$
$$x_{20} = (x_2 - x_0) / (x_2 - x_1)$$
$$y_{01} = (y_0 - y_1) / (y_2 - y_1)$$
$$y_{20} = (y_2 - y_0) / (y_2 - y_1)$$

Substituting 0.15 for *index_1* and 1.16 for *index_2* results in the fall_transition value of:

$$T_{00} = 0.75 * 0.25 * 0.1937 + 0.75 * 0.75 * 0.7280 + 0.25 * 0.25 * 0.2327 + 0.25 * 0.75 * 0.7676 = 0.6043$$

Note that the equations above are valid for interpolation as well as extrapolation - that is when the indices (x_0, y_0) lie outside the characterized range of indices. As an example, for the table lookup with 0.05 for *index_1* and 1.7 for *index_2*, the fall transition value is obtained as:

$$T_{00} = 1.25 * (-0.25) * 0.1937 + 1.25 * 1.25 * 0.7280 + (-0.25) * (-0.25) * 0.2327 + (-0.25) * 1.25 * 0.7676 = 0.8516$$

3.2.3 Threshold Specifications and Slew Derating

The slew[1] values are based upon the measurement thresholds specified in the library. Most of the previous generation libraries (0.25µm or older) used 10% and 90% as measurement thresholds for slew or transition time.

1. Slew is same as transition time.

The slew thresholds are chosen to correspond to the linear portion of the waveform. As technology becomes finer, the portion where the actual waveform is most linear is typically between 30% and 70% points. Thus, most of the newer generation timing libraries specify slew measurement points as 30% and 70% of *Vdd*. However, because the transition times were previously measured between 10% and 90%, the transition times measured between 30% and 70% are usually doubled for populating the library. This is specified by the *slew derate factor* which is typically specified as 0.5. The slew thresholds of 30% and 70% with slew derate as 0.5 results in equivalent measurement points of 10% and 90%. An example settings of threshold is illustrated below.

```
/* Threshold definitions */
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 : 0.5;
```

The above settings specify that the transition times in the library tables have to be multiplied by 0.5 to obtain the transition times which correspond to the slew threshold (30-70) settings. This means that the values in the transition tables (as well as corresponding index values) are effectively 10-90 values. During characterization, the transition is measured at 30-70 and the transition data in the library corresponds to extrapolation of measured values to 10% to 90% ((70 - 30)/(90 - 10) = 0.5).

Another example with a different set of slew threshold settings may contain:

```
/* Threshold definitions 20/80/1 */
slew_lower_threshold_pct_fall : 20.0;
```

```
slew_upper_threshold_pct_fall : 80.0;
slew_lower_threshold_pct_rise : 20.0;
slew_upper_threshold_pct_rise : 80.0;
/* slew_derate_from_library not specified */
```

In this example of 20-80 slew threshold settings, there is no *slew_derate_from_library* specified (implies a default of 1.0), which means that the transition time data in the library is not derated. The values in the transition tables correspond directly to the 20-80 characterized slew values. See Figure 3-4.

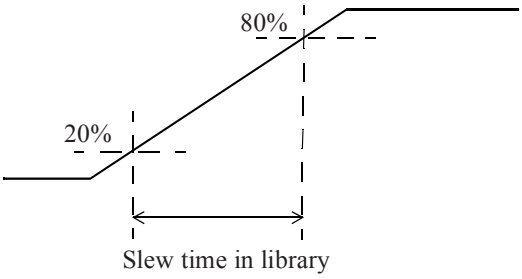


Figure 3-4 *No derating of slew.*

Here is another example of slew threshold settings in a cell library.

```
slew_lower_threshold_pct_rise : 20.00;
slew_upper_threshold_pct_rise : 80.00;
slew_lower_threshold_pct_fall : 20.00;
slew_upper_threshold_pct_fall : 80.00;
slew_derate_from_library : 0.6;
```

In this case, the *slew_derate_from_library* is set to 0.6 and characterization slew trip points are specified as 20% and 80%. This implies that transition table data in the library corresponds to 0% to 100% ((80 - 20)/(100 - 0) = 0.6) extrapolated values. This is shown in Figure 3-5.

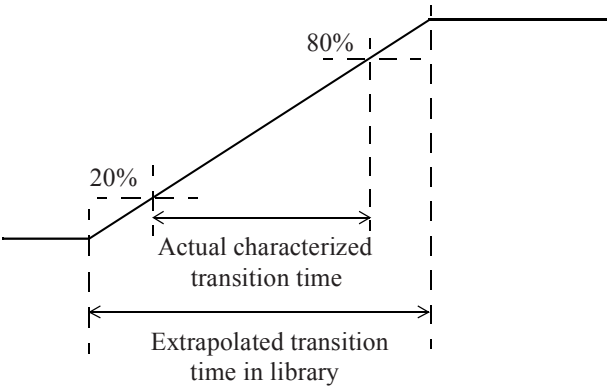


Figure 3-5 *Slew derating applied.*

When slew derating is specified, the slew value internally used during delay calculation is:

```
library_transition_time_value * slew_derate
```

This is the slew used internally by the delay calculation tool and corresponds to the characterized slew threshold measurement points.

3.3 Timing Models - Combinational Cells

Let us consider the timing arcs for a two-input *and* cell. Both the timing arcs for this cell are *positive_unate*; therefore an input pin rise corresponds to an output rise and vice versa.

For the two-input *and* cell, there are four delays:

- *A* -> *Z*: Output rise
- *A* -> *Z*: Output fall
- *B* -> *Z*: Output rise

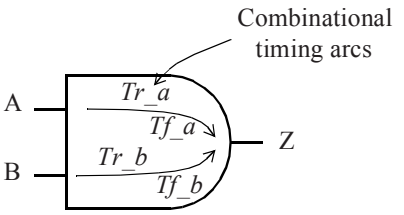


Figure 3-6 *Combinational timing arcs.*

- *B* -> *Z*: Output fall

This implies that for the NLDM model, there would be four table models for specifying delays. Similarly, there would be four such table models for specifying the output transition times as well.

3.3.1 Delay and Slew Models

An example of a timing model for input *INP1* to output *OUT* for a three-input *nand* cell is specified as follows.

```
pin (OUT) {
  max_transition : 1.0;
  timing() {
    related_pin : "INP1";
    timing_sense : negative_unate;
    cell_rise(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7");
      index_2 ("0.16, 0.35, 1.43");
      values ( \
        "0.0513, 0.1537, 0.5280", \
        "0.1018, 0.2327, 0.6476", \
        "0.1334, 0.2973, 0.7252");
      }
    rise_transition(delay_template_3x3) {
      index_1 ("0.1, 0.3, 0.7");
```

```
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0417, 0.1337, 0.4680", \
          "0.0718, 0.1827, 0.5676", \
          "0.1034, 0.2173, 0.6452");
      }
      cell_fall(delay_template_3x3) {
        index_1 ("0.1, 0.3, 0.7");
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0617, 0.1537, 0.5280", \
          "0.0918, 0.2027, 0.5676", \
          "0.1034, 0.2273, 0.6452");
      }
      fall_transition(delay_template_3x3) {
        index_1 ("0.1, 0.3, 0.7");
        index_2 ("0.16, 0.35, 1.43");
        values ( \
          "0.0817, 0.1937, 0.7280", \
          "0.1018, 0.2327, 0.7676", \
          "0.1334, 0.2973, 0.8452");
      }
      . . .
    }
    . . .
  }
```

In this example, the characteristics of the timing arc from *INP1* to *OUT* are described using two cell delay tables, *cell_rise* and *cell_fall*, and two transition tables, *rise_transition* and *fall_transition*. The output *max_transition* value is also included in the above example.

Positive or Negative Unate

As described in Section 2.7, the timing arc in the *nand* cell example is negative unate which implies that the output pin transition direction is opposite (negative) of the input pin transition direction. Thus, the *cell_rise* table

lookup corresponds to the falling transition time at the input pin. On the other hand, the timing arcs through an *and* cell or *or* cell are positive unate since the output transition is in the same direction as the input transition.

3.3.2 General Combinational Block

Consider a combinational block with three inputs and two outputs.

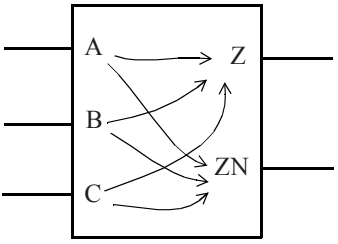


Figure 3-7 *General combinational block.*

A block such as this can have a number of timing arcs. In general, the timing arcs can be from each input to each output of the block. If the logic path from input to output is non-inverting or positive unate, then the output has the same polarity as the input. If it is an inverting logic path or negative unate, the output has an opposite polarity to input; thus, when the input rises, the output falls. These timing arcs represent the propagation delays through the block.

Some timing arcs through a combinational cell can be positive unate as well as negative unate. An example is the timing arc through a two-input *xor* cell. A transition at an input of a two-input *xor* cell can cause an output transition in the same or in the opposite transition direction depending on the logic state of the other input of the cell. The timing for these arcs can be described as non-unate or as two different sets of positive unate and negative unate timing models which are state-dependent. Such state-dependent tables are described in greater detail in Section 3.5.

3.4 Timing Models - Sequential Cells

Consider the timing arcs of a sequential cell shown in Figure 3-8.

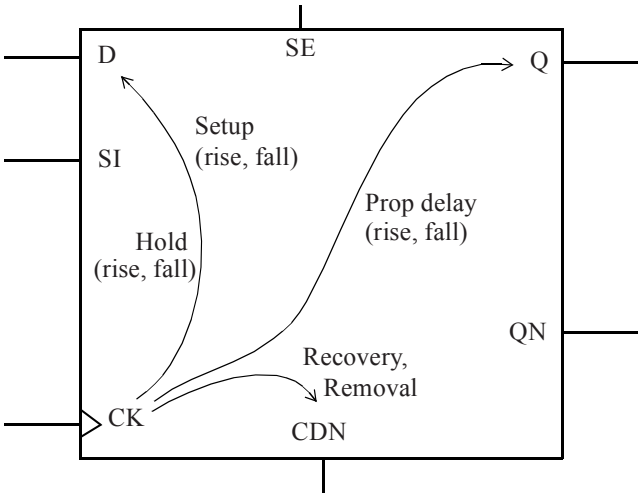


Figure 3-8 *Sequential cell timing arcs.*

For synchronous inputs, such as pin *D* (or *SI, SE*), there are the following timing arcs:

i. Setup check arc (rising and falling)

ii. Hold check arc (rising and falling)

For asynchronous inputs, such as pin *CDN*, there are the following timing arcs:

i. Recovery check arc

ii. Removal check arc

For synchronous outputs of a flip-flop, such as pins *Q* or *QN*, there is the following timing arc:

i. CK-to-output propagation delay arc (rising and falling)

All of the synchronous timing arcs are with respect to the **active edge** of the clock, the edge of the clock that causes the sequential cell to capture the data. In addition, the clock pin and asynchronous pins such as clear, can have pulse width timing checks. Figure 3-9 shows the timing checks using various signal waveforms.

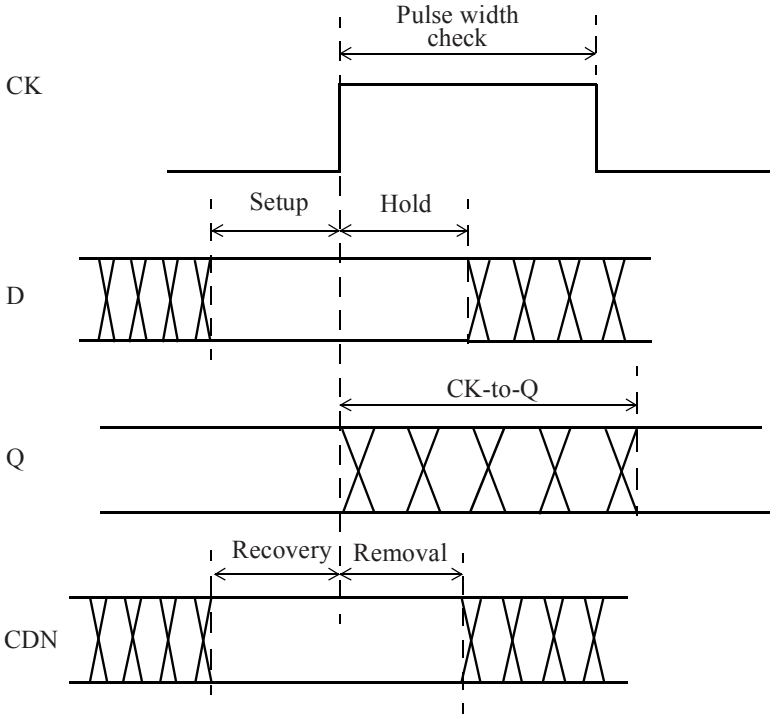


Figure 3-9 *Timing arcs on an active rising clock edge.*

3.4.1 Synchronous Checks: Setup and Hold

The setup and hold synchronous timing checks are needed for proper propagation of data through the sequential cells. These checks verify that the data input is unambiguous at the active edge of the clock and the proper data is latched in at the active edge. These timing checks validate if the data input is stable around the active clock edge. The minimum time before the active clock when the data input must remain stable is called the *setup time*. This is measured as the time interval from the latest data signal crossing its threshold (normally 50% of *Vdd*) to the active clock edge crossing its threshold (normally 50% of *Vdd*). Similarly, the hold time is the minimum time the data input must remain stable just after the active edge of the clock. This is measured as the time interval from the active clock edge crossing its threshold to the earliest data signal crossing its threshold. As mentioned previously, the active edge of the clock for a sequential cell is the rising or falling edge that causes the sequential cell to capture data.

Example of Setup and Hold Checks

The setup and hold constraints for a synchronous pin of a sequential cell are normally described in terms of two-dimensional tables as illustrated below. The example below shows the setup and hold timing information for the data pin of a flip-flop.

```
pin (D) {
  direction : input;
  . . .
  timing () {
    related_pin : "CK";
    timing_type : "setup_rising";
    rise_constraint ("setuphold_template_3x3") {
      index_1("0.4, 0.57, 0.84"); /* Data transition */
      index_2("0.4, 0.57, 0.84"); /* Clock transition */
      values( /*    0.4      0.57     0.84 */ \
        /* 0.4 */   "0.063,  0.093,   0.112", \
        /* 0.57 */  "0.526,  0.644,   0.824", \
        /* 0.84 */  "0.720,  0.839,   0.930");
```

```
    }
    fall_constraint ("setuphold_template_3x3") {
     index_1("0.4, 0.57, 0.84"); /* Data transition */
     index_2("0.4, 0.57, 0.84"); /* Clock transition */
     values( /*    0.4     0.57     0.84 */ \
       /* 0.4 */   "0.762,  0.895,   0.969", \
       /* 0.57 */  "0.804,  0.952,   0.166", \
       /* 0.84 */  "0.159,  0.170,   0.245");
    }
  }
}

timing () {
  related_pin : "CK";
  timing_type : "hold_rising";
  rise_constraint ("setuphold_template_3x3") {
    index_1("0.4, 0.57, 0.84"); /* Data transition */
    index_2("0.4, 0.57, 0.84"); /* Clock transition */
    values(   /*    0.4      0.57     0.84 */ \
      /* 0.4 */   "-0.220,  -0.339,  -0.584", \
      /* 0.57 */  "-0.247,  -0.381,  -0.729", \
      /* 0.84 */  "-0.398,  -0.516,  -0.864");
  }
  fall_constraint ("setuphold_template_3x3") {
    index_1("0.4, 0.57, 0.84"); /* Data transition */
    index_2("0.4, 0.57, 0.84");/* Clock transition */
    values(   /*    0.4         0.57     0.84 */ \
      /* 0.4 */    "-0.028,    -0.397,  -0.489", \
      /* 0.57 */   "-0.408,    -0.527,  -0.649", \
      /* 0.84 */   "-0.705,    -0.839,  -0.580");
  }
}
```

The example above shows setup and hold constraints on the input pin *D* with respect to the rising edge of the clock *CK* of a sequential cell. The two-dimensional models are in terms of the transition times at the *constrained_pin* (*D*) and the *related_pin* (*CK*). The lookup for the two-

dimensional table is based upon the template *setuphold_template_3x3* described in the library. For the above example, the lookup table template *setuphold_template_3x3* is described as:

```
lu_table_template(setuphold_template_3x3) {
  variable_1 : constrained_pin_transition;
  variable_2 : related_pin_transition;
  index_1 ("1000, 1001, 1002");
  index_2 ("1000, 1001, 1002");
}
/* The constrained pin and the related pin can be in either or-
der, that is, variable_1 could be the related pin transition.
However, these designations are usually consistent across all
templates in a library. */
```

Like in previous examples, the setup values in the table are specified like a nested loop with the first index, *index_1*, being the outer (or least varying) variable and the second index, *index_2*, being the inner (or most varying) variable and so on. Thus, with a *D* pin rise transition time of 0.4ns and *CK* pin rise transition time of 0.84ns, the setup constraint for the rising edge of the *D* pin is 0.112ns - the value is read from the *rise_constraint* table. For the falling edge of the *D* pin, the setup constraint will examine the *fall_constraint* table of the setup tables. For lookup of the setup and hold constraint tables where the transition times do not correspond to the index values, the general procedure for *non-linear model* lookup described in Section 3.2 is applicable.

Note that the *rise_constraint* and *fall_constraint* tables of the setup constraint refer to the *constrained_pin*. The clock transition used is determined by the *timing_type* which specifies whether the cell is rising edge-triggered or falling edge-triggered.

Negative Values in Setup and Hold Checks

Notice that some of the hold values in the example above are negative. This is acceptable and normally happens when the path from the pin of the flip-flop to the internal latch point for the data is longer than the corresponding

path for the clock. Thus, a negative hold check implies that the data pin of the flip-flop can change ahead of the clock pin and still meet the hold time check.

The setup values of a flip-flop can also be negative. This means that at the pins of the flip-flop, the data can change after the clock pin and still meet the setup time check.

Can both setup and hold be negative? No; for the setup and hold checks to be consistent, the sum of setup and hold values should be positive. Thus, if the setup (or hold) check contains negative values - the corresponding hold (or setup) should be sufficiently positive so that the setup plus hold value is a positive quantity. See Figure 3-10 for an example with a negative hold value. Since the setup has to occur prior to the hold, setup plus hold is a positive quantity. The setup plus hold time is the width of the region where the data signal is required to be steady.

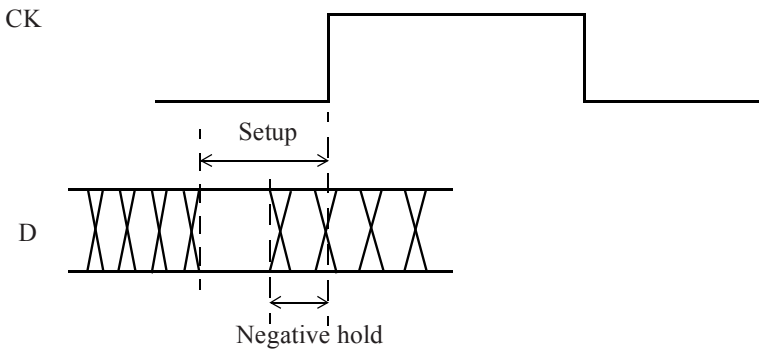


Figure 3-10 *Negative value for hold timing check.*

For flip-flops, it is helpful to have a negative hold time on scan data input pins. This gives flexibility in terms of clock skew and can eliminate the need for almost all buffer insertion for fixing hold violations in scan mode (*scan mode* is the one in which flip-flops are tied serially forming a scan chain - output of flip-flop is typically connected to the scan data input pin of the next flip-flop in series; these connections are for testability).

Similar to the setup or hold check on the synchronous data inputs, there are constraint checks governing the asynchronous pins. These are described next.

3.4.2 Asynchronous Checks

Recovery and Removal Checks

Asynchronous pins such as asynchronous clear or asynchronous set override any synchronous behavior of the cell. When an asynchronous pin is active, the output is governed by the asynchronous pin and not by the clock latching in the data inputs. However, when the asynchronous pin becomes inactive, the active edge of the clock starts latching in the data input. The asynchronous recovery and removal constraint checks verify that the asynchronous pin has returned unambiguously to an inactive state at the next active clock edge.

The **recovery time** is the minimum time that an asynchronous input is stable after being de-asserted before the next active clock edge.

Similarly, the **removal time** is the minimum time after an active clock edge that the asynchronous pin must remain active before it can be de-asserted.

The asynchronous removal and recovery checks are described in Section 8.6 and Section 8.7 respectively.

Pulse Width Checks

In addition to the synchronous and asynchronous timing checks, there is a check which ensures that the pulse width at an input pin of a cell meets the minimum requirement. For example, if the width of pulse at the clock pin is smaller than the specified minimum, the clock may not latch the data properly. The pulse width checks can be specified for relevant synchronous and asynchronous pins also. The minimum pulse width checks can be specified for high pulse and also for low pulse.

Example of Recovery, Removal and Pulse Width Checks

An example of recovery time, removal time, and pulse width check for an asynchronous clear pin *CDN* of a flip-flop is given below. The recovery and removal checks are with respect to the clock pin *CK*. Since the recovery and removal checks are defined for an asynchronous pin being de-asserted, only rise constraints exists in the example below. The minimum pulse width check for the pin *CDN* is for a low pulse. Since the *CDN* pin is active low, there is no constraint for the high pulse width on this pin and is thus not specified.

```
pin(CDN) {
  direction : input;
  capacitance : 0.002236;
  . . .
  timing() {
    related_pin : "CDN";
    timing_type : min_pulse_width;
    fall_constraint(width_template_3x1) { /*low pulse check*/
     index_1 ("0.032, 0.504, 0.788"); /* Input transition */
     values ( /*    0.032     0.504    0.788 */ \
                  "0.034,    0.060,   0.377");
    }
  }
  timing() {
    related_pin : "CK";
    timing_type : recovery_rising;
    rise_constraint(recovery_template_3x3) { /* CDN rising */
     index_1 ("0.032, 0.504, 0.788"); /* Data transition */
     index_2 ("0.032, 0.504, 0.788"); /* Clock transition */
     values( /*     0.032     0.504    0.788 */ \
       /* 0.032 */  "-0.198,   -0.122,  0.187", \
       /* 0.504 */  "-0.268,   -0.157,  0.124", \
       /* 0.788 */  "-0.490,   -0.219,  -0.069");
    }
  }
```

```
  timing() {
    related_pin : "CP";
    timing_type : removal_rising;
    rise_constraint(removal_template_3x3) { /* CDN rising */
      index_1 ("0.032, 0.504, 0.788"); /* Data transition */
      index_2 ("0.032, 0.504, 0.788"); /* Clock transition */
      values( /*     0.032   0.504    0.788 */ \
        /* 0.032 */  "0.106,  0.167,   0.548", \
        /* 0.504 */  "0.221,  0.381,   0.662", \
        /* 0.788 */  "0.381,  0.456,   0.778");
    }
  }
}
```

3.4.3 Propagation Delay

The propagation delay of a sequential cell is from the active edge of the clock to a rising or falling edge on the output. Here is an example of a propagation delay arc for a negative edge-triggered flip-flop, from clock pin *CKN* to output *Q*. This is a non-unate timing arc as the active edge of the clock can cause either a rising or a falling edge on the output *Q*. Here is the delay table:

```
timing() {
  related_pin : "CKN";
  timing_type : falling_edge;
  timing_sense : non_unate;
  cell_rise(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7"); /* Clock transition */
    index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
    values ( /*    0.16    0.35     1.43 */ \
      /* 0.1 */   "0.0513,  0.1537,  0.5280", \
      /* 0.3 */   "0.1018,  0.2327,  0.6476", \
      /* 0.7 */   "0.1334,  0.2973,  0.7252");
  }
```

```
  rise_transition(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0417, 0.1337, 0.4680", \
      "0.0718, 0.1827, 0.5676", \
      "0.1034, 0.2173, 0.6452");
  }
  cell_fall(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0617, 0.1537, 0.5280", \
      "0.0918, 0.2027, 0.5676", \
      "0.1034, 0.2273, 0.6452");
  }
  fall_transition(delay_template_3x3) {
    index_1 ("0.1, 0.3, 0.7");
    index_2 ("0.16, 0.35, 1.43");
    values ( \
      "0.0817, 0.1937, 0.7280", \
      "0.1018, 0.2327, 0.7676", \
      "0.1334, 0.2973, 0.8452");
  }
}
```

As in the earlier examples, the delays to the output are expressed as two-dimensional tables in terms of input transition time and the capacitance at the output pin. However in this example, the input transition time to use is the falling transition time at the *CKN* pin since this is a falling edge-triggered flip-flop. This is indicated by the construct *timing_type* in the example above. A rising edge-triggered flip-flop will specify *rising_edge* as its *timing_type*.

```
  timing() {
    related_pin : "CKP";
    timing_type : rising_edge;
```

```
    timing_sense : non_unate;
    cell_rise(delay_template_3x3) {
      . . .
    }
    . . .
  }
```

3.5 State-Dependent Models

In many combinational blocks, the timing arcs between inputs and outputs depend on the state of other pins in the block. These timing arcs between input and output pins can be positive unate, negative unate, or both positive as well as negative unate arcs. An example is the *xor* or *xnor* cell where the timing to the output can be positive unate or negative unate. In such cases, the timing behaviors can be different depending upon the state of other inputs of the block. In general, multiple timing models depending upon the states of the pins are described. Such models are referred to as **state-dependent models**.

XOR, XNOR and Sequential Cells

Consider an example of a two-input *xor* cell. The timing path from an input *A1* to output *Z* is positive unate when the other input *A2* is logic-0. When the input *A2* is logic-1, the path from *A1* to *Z* is negative unate. These two timing models are specified using state-dependent models. The timing model from *A1* to *Z* when *A2* is logic-0 is specified as follows:

```
pin (Z) {
  direction : output;
  max_capacitance : 0.0842;
  function : "(A1^A2)";
  timing() {
    related_pin : "A1";
    when : "!A2";
    sdf_cond : "A2 == 1'b0";
```

```
    timing_sense : positive_unate;
    cell_rise(delay_template_3x3) {
     index_1 ("0.0272, 0.0576, 0.1184"); /* Input slew */
     index_2 ("0.0102, 0.0208, 0.0419"); /* Output load */
     values( \
       "0.0581, 0.0898, 0.2791", \
       "0.0913, 0.1545, 0.2806", \
       "0.0461, 0.0626, 0.2838");
   }
   . . .
}
```

The state-dependent condition is specified using the *when* condition. While the cell model excerpt only illustrates the *cell_rise* delay, other timing models (*cell_fall, rise_transition* and *fall_transition* tables) are also specified with the same *when* condition. A separate timing model is specified for the other *when* condition - for the case when *A2* is logic-1.

```
timing() {
  related_pin : "A1";
  when : "A2";
  sdf_cond : "A2 == 1'b1";
  timing_sense : negative_unate;
  cell_fall(delay_template_3x3) {
    index_1 ("0.0272, 0.0576, 0.1184");
    index_2 ("0.0102, 0.0208, 0.0419");
    values( \
      "0.0784, 0.1019, 0.2269", \
      "0.0943, 0.1177, 0.2428", \
      "0.0997, 0.1796, 0.2620");
  }
  . . .
}
```

The *sdf_cond* is used to specify the condition of the timing arc that is to be used when generating SDF - see the example in Section 3.9 and the *COND* construct described in Appendix B.

State-dependent models are used for various types of timing arcs. Many sequential cells specify the setup or hold timing constraints using state-dependent models. An example of a scan flip-flop using state-dependent models for hold constraint is specified next. In this case, two sets of models are specified - one when the scan enable pin *SE* is active and another when the scan enable pin is inactive.

```
pin (D) {
  . . .
  timing() {
    related_pin : "CK";
    timing_type : hold_rising;
    when : "!SE";
    fall_constraint(hold_template_3x3) {
      index_1("0.08573, 0.2057, 0.3926");
      index_2("0.08573, 0.2057, 0.3926");
      values("-0.05018, -0.02966, -0.00919",\
             "-0.0703, -0.05008, -0.0091",\
             "-0.1407, -0.1206, -0.1096");
    }
    . . .
  }
}
```

The above model is used when the *SE* pin is at logic-0. A similar model is specified with the *when* condition *SE* as logic-1.

Some timing relationships are specified using both state-dependent as well as non-state-dependent models. In such cases, the timing analysis will use the state-dependent model if the state of the cell is known and is included in one of the state-dependent models. If the state-dependent models do not cover the condition of the cell, the timing from the non-state-dependent model is utilized. For example, consider a case where the hold constraint is

specified by only one *when* condition for *SE* at logic-0 and no separate state-dependent model is specified for *SE* at logic-1. In such a scenario, if the *SE* is set to logic-1, the hold constraint from the non-state-dependent model is used. If there is no non-state-dependent model for the hold constraint, there will *not* be any active hold constraint!

State-dependent models can be specified for any of the attributes in the timing library. Thus state-dependent specifications can exist for power, leakage power, transition time, rise and fall delays, timing constraints, and so on. An example of the state dependent leakage power specification is given below:

```
leakage_power() {
  when : "A1 !A2";
  value : 259.8;
}
leakage_power() {
  when : "A1 A2";
  value : 282.7;
}
```

3.6 Interface Timing Model for a Black Box

This section describes the timing arcs for the IO interfaces of a black box (an arbitrary module or block). A timing model captures the timing for the IO interfaces of the black box. The black box interface model can have combinational as well as sequential timing arcs. In general, these arcs can also be state-dependent.

For the example shown in Figure 3-11, the timing arcs can be placed under the following categories:

- *Input to output combinational arc*: This corresponds to a direct combinational path from input to output, such as from the input port *FIN* to the output port *FOUT*.

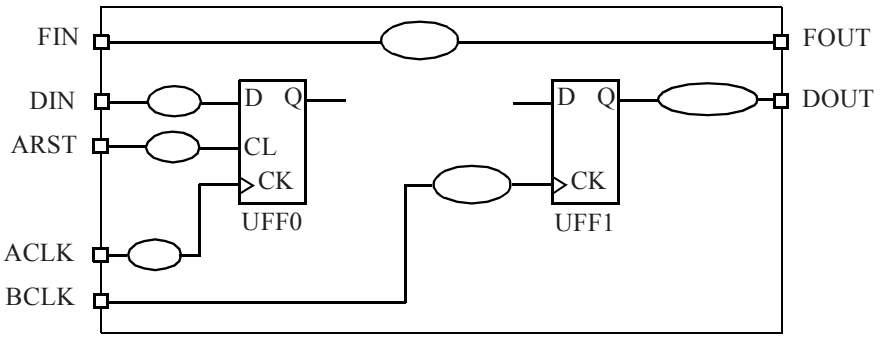


Figure 3-11 *Design modeled with interface timing.*

- *Input sequential arc*: This is described as setup or hold time for the input connected to a *D*-pin of the flip-flop. In general, there can be combinational logic from the input of the block before it is connected to a *D*-pin of the flip-flop. An example of this is a set-up check at port *DIN* with respect to clock *ACLK*.

- *Asynchronous input arc*: This is similar to the recovery or removal timing constraint for the input asynchronous pins of a flip-flop. An example is the input *ARST* to the asynchronous clear pin of flip-flop *UFF0*.

- *Output sequential arc*: This is similar to the output propagation timing for the clock to output connected to *Q* of the flip-flop. In general, there can be combinational logic between the flip-flop output and the output of the module. An example is the path from clock *BCLK* to the output of flip-flop *UFF1* to the output port *DOUT*.

In addition to the timing arcs above, there can also be pulse width checks on the external clock pins of the black box. It is also possible to define internal nodes and to define generated clocks on these internal nodes as well as

specify timing arcs to and from these nodes. In summary, a black box model can have the following timing arcs:

i. Input to output timing arcs for combinational logic paths.

ii. Setup and hold timing arcs from the synchronous inputs to the related clock pins.

iii. Recovery and removal timing arcs for the asynchronous inputs to the related clock pins.

iv. Output propagation delay from clock pins to the output pins.

The interface timing model as described above is not intended to capture the internal timing of the black box, but only the timing of its interfaces.

3.7 Advanced Timing Modeling

The timing models, such as NLDM, represent the delay through the timing arcs based upon output load capacitance and input transition time. In reality, the load seen by the cell output is comprised of capacitance as well as interconnect resistance. The interconnect resistance becomes an issue since the NLDM approach assumes that the output loading is purely capacitive. Even with non-zero interconnect resistance, these NLDM models have been utilized when the effect of interconnect resistance is small. In presence of resistive interconnect, the delay calculation methodologies retrofit the NLDM models by obtaining an equivalent **effective capacitance** at the output of the cell. The "effective" capacitance methodology used within delay calculation tools obtains an equivalent capacitance that has the same delay at the output of the cell as the cell with RC interconnect. The effective capacitance approach is described as part of delay calculation in Section 5.2.

As the feature size shrinks, the effect of interconnect resistance can result in large inaccuracy as the waveforms become highly non-linear. Various modeling approaches provide additional accuracy for the cell output drivers. Broadly, these approaches obtain higher accuracy by modeling the out-

put stage of the driver by an equivalent current source. Examples of these approaches are - **CCS** (**C**omposite **C**urrent **S**ource), or ECSM (**E**ffective **C**urrent **S**ource **M**odel). For example, the CCS timing models provide the additional accuracy for modeling cell output drivers by using a time-varying and voltage-dependent current source. The timing information is provided by specifying detailed models for the receiver pin capacitance[1] and output charging currents under different scenarios. The details of the CCS model are described next.

3.7.1 Receiver Pin Capacitance

The receiver pin capacitance corresponds to the input pin capacitance specified for the NLDM models. Unlike the pin capacitance for the NLDM models, the CCS models allow separate specification of receiver capacitance in different portions of the transitioning waveform. Due to interconnect RC and the equivalent input non-linear capacitance due to the Miller effect from the input devices within the cell, the receiver capacitance value varies at different points on the transitioning waveform. This capacitance is thus modeled differently in the initial (or leading) portion of waveform versus the trailing portion of the waveform.

The receiver pin capacitance can be specified at the pin level (like in NLDM models) where all timing arcs through that pin use that capacitance value. Alternately, the receiver capacitance can be specified at the timing arc level in which case different capacitance models can be specified for different timing arcs. These two methods of specifying the receiver pin capacitance are described next.

1. Input pin capacitance - equivalent to *pin capacitance* in NLDM.

Specifying Capacitance at the Pin Level

When specified at the pin level, an example of a one-dimensional table specification for receiver pin capacitance is given next.

```
pin (IN) {
  . . .
  receiver_capacitance1_rise ("Lookup_table_4") {
  index_1: ("0.1, 0.2, 0.3, 0.4"); /* Input transition */
  values("0.001040, 0.001072, 0.001074, 0.001085");
}
```

The *index_1* specifies the indices for input transition time for this pin. The one-dimensional table in *values* specifies the receiver capacitance for rising waveform at an input pin for the leading portion of the waveform.

Similar to the *receiver_capacitance1_rise* shown above, the *receiver_capacitance2_rise* specifies the rise capacitance for the trailing portion of the input rising waveform. The fall capacitances (pin capacitance for falling input waveform) are specified by the attributes *receiver_capacitance1_fall* and *receiver_capacitance2_fall* respectively.

Specifying Capacitance at the Timing Arc Level

The receiver pin capacitance can also be specified with the timing arc as a two-dimensional table in terms of input transition time and output load. An example of the specification at the timing arc level is given below. This example specifies the receiver pin rise capacitance for the leading portion of the waveform at pin *IN* as a function of transition time at input pin *IN* and load at output pin *OUT*.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN"-;
    . . .
```

```
    receiver_capacitance1_rise ("Lookup_table_4x4") {
      index_1("0.1, 0.2, 0.3, 0.4"); /* Input transition */
      index_2("0.01, 0.2, 0.4, 0.8"); /* Output capacitance */
      values("0.001040-, 0.001072-, 0.001074-, 0.001075", \
             "0.001148-, 0.001150-, 0.001152-, 0.001153", \
             "0.001174-, 0.001172-, 0.001172-, 0.001172", \
             "0.001174-, 0.001171-, 0.001177-, 0.001174");
    }
    . . .
  }
  . . .
}
```

The above example specifies the model for the *receiver_capacitance1_rise*. The library includes similar definitions for the *receiver_capacitance2_rise, receiver_capacitance1_fall,* and *receiver_capacitance2_fall* specifications.

The four different types of receiver capacitance types are summarized in the table below. As illustrated above, these can be specified at the pin level as a one-dimensional table or at the timing level as a two-dimensional table.

Capacitance type	*Edge*	*Transition*
Receiver_capacitance1_rise	Rising	Leading portion of transition
Receiver_capacitance1_fall	Falling	Leading portion of transition
Receiver_capacitance2_rise	Rising	Trailing portion of transition
Receiver_capacitance2_fall	Falling	Trailing portion of transition

Table 3-12 *Receiver capacitance types.*

3.7.2 Output Current

In the CCS model, the non-linear timing is represented in terms of output current. The output current information is specified as a lookup table that is dependent on input transition time and output load.

The output current is specified for different combinations of input transition time and output capacitance. For each of these combinations, the output current waveform is specified. Essentially, the waveform here refers to output current values specified as a function of time. An example of the output current for falling output waveform, specified using *output_current_fall*, is as shown next.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN"-;
    . . .
    output_current_fall () {
      vector ("LOOKUP_TABLE_1x1x5") {
        reference_time : 5.06; /* Time of input crossing
          threshold */
        index_1("0.040"); /* Input transition */
        index_2("0.900"); /* Output capacitance */
        index_3("5.079e+00, 5.093e+00, 5.152e+00,
                 5.170e+00, 5.352e+00");/* Time values */
        /* Output charging current: */
        values("-5.784e-02, -5.980e-02, -5.417e-02,
                 -4.257e-02, -2.184e-03");
      }
      . . .
    }
    . . .
  }
  . . .
}
```

The *reference_time* attribute refers to the time when the input waveform crosses the delay threshold. The *index_1* and *index_2* refer to the input transition time and the output load used and *index_3* is the time. The *index_1* and *index_2* (the input transition time and output capacitance) can have only one value each. The *index_3* refers to the time values and the table values refer to the corresponding output current. Thus, for the given input transition time and output load, the output current waveform as a function of time is available. Additional lookup tables for other combinations of input transition time and output capacitance are also specified.

Output current for rising output waveform, specified using *output_current_rise*, is described similarly.

3.7.3 Models for Crosstalk Noise Analysis

This section describes the CCS models for the crosstalk noise (or glitch) analysis. These are described as **CCSN** (**CCS N**oise) models. The CCS noise models are structural models and are represented for different CCBs (Channel Connected Blocks) within the cell.

What is a CCB? The **CCB** refers to the source-drain channel connected portion of a cell. For example, single stage cells such as an *inverter*, *nand* and *nor* cells contain only one CCB - the entire cell is connected through using one channel connection region. Multi-stage cells such as *and* cells, or *or* cells, contain multiple CCBs.

The CCSN models are usually specified for the first CCB driven by the cell input, and the last CCB driving the cell output. These are specified using steady state current, output voltage and propagated noise models.

For single stage combinational cells such as *nand* and *nor* cells, the CCS noise models are specified for each timing arc. These cells have only one CCB and thus the models are from input pins to the output pin of the cell.

An example model for a *nand* cell is described below:

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN1";
    . . .
    ccsn_first_stage() { /* First stage CCB */
      is_needed : true;
      stage_type : both; /*CCB contains pull-up and pull-down*/
      is_inverting : true;
      miller_cap_rise : 0.8;
      miller_cap_fall : 0.5;
      dc_current (ccsn_dc) {
        index_1 ("-0.9, 0, 0.5, 1.35, 1.8"); /* Input voltage */
        index_2 ("-0.9, 0, 0.5, 1.35, 1.8"); /* Output voltage*/
        values ( \
          "1.56, 0.42, . . ."); /* Current at output pin */
      }
      . . .
      output_voltage_rise () {
        vector (ccsn_ovrf) {
          index_1 ("0.01"); /* Rail-to-rail input transition */
          index_2 ("0.001"); /* Output net capacitance */
          index_3 ("0.3, 0.5, 0.8"); /* Time */
          values ("0.27, 0.63, 0.81");
        }
        . . .
      }
      output_voltage_fall () {
        vector (ccsn_ovrf) {
          index_1 ("0.01"); /* Rail-to-rail input transition */
          index_2 ("0.001"); /* Output net capacitance */
          index_3 ("0.2, 0.4, 0.6"); /* Time */
          values ("0.81, 0.63, 0.27");
        }
        . . .
```

```
        }
        propagated_noise_low () {
          vector (ccsn_pnlh) {
          index_1 ("0.5"); /* Input glitch height */
          index_2 ("0.6"); /* Input glitch width */
          index_3 ("0.05"); /* Output net capacitance */
          index_4 ("0.3, 0.4, 0.5, 0.7"); /* Time */
          values ("0.19, 0.23, 0.19, 0.11");
        }
        propagated_noise_high () {
          . . .
      }
    }
  }
```

We now describe the attributes of the CCS noise model. The attribute *ccsn_first_stage* indicates that the model is for the first stage CCB of the *nand* cell. As mentioned before, the *nand* cell has only one CCB. The attribute *is_needed* is almost always true with the exception being that for non-functional cells such as load cells and antenna cells. The *stage_type* with value *both* specifies that this stage has both pull-up and pull-down structures. The *miller_cap_rise* and *miller_cap_fall* represent the Miller capacitances[1] for the rising and falling output transitions respectively.

DC Current

The *dc_current* tables represents the DC current at the output pin for different combinations of input and output pin voltages. The *index_1* specifies the input voltage and *index_2* specifies the output voltage. The *values* in the two-dimensional table specify the DC current at the CCB output. The input voltages and output currents are all specified in library units (normally *Volt* and *mA*). For the example CCS noise model from input *IN1* to *OUT* of the *nand* cell, an input voltage of -0.9V and output voltage of 0V, results in the DC current at the output of 0.42mA.

1. Miller capacitance accounts for the increase in the equivalent input capacitance of an inverting stage due to amplification of capacitance between the input and output terminals.

Output Voltage

The *output_voltage_rise* and *output_voltage_fall* constructs contain the timing information for the CCB output rising and falling respectively. These are specified as multi-dimensional tables for the CCB output node. The multi-dimensional tables are organized as multiple tables specifying the rising and falling output voltages for different input transition time and output net capacitances. Each table has *index_1* specifying the rail-to-rail input transition time rate and *index_2* specifying the output net capacitance. The *index_3* specifies the times when the output voltage crosses specific threshold points (in this case 30%, 70% and 90% of *Vdd* supply of 0.9V). In each multi-dimensional table, the voltage crossing points are fixed and the time-values when the CCB output node crosses the voltage is specified in *index_3*.

Propagated Noise

The *propagated_noise_high* and *propagated_noise_low* models specify multi-dimensional tables which provide noise propagation information through the CCB. These models characterize the crosstalk glitch (or noise) propagation from an input to the output of the CCB. The characterization uses symmetric triangular waveform at the input. The multi-dimensional tables for *propagated_noise* are organized as multiple tables specifying the glitch waveform at the output of the CCB. These multi-dimensional tables contain:

i. input glitch magnitude (in *index_1*),

ii. input glitch width (in *index_2*),

iii. CCB output net capacitance (in *index_3*), and

iv. time (in *index_4*).

The CCB output voltage (or the noise propagated through CCB) is specified in the table values.

Noise Models for Two-Stage Cells

Just like the single stage cells, the CCS noise models for two-stage cells such as *and* cells and *or* cells, are normally described as part of the timing arc. Since these cells contain two separate CCBs, the noise models are specified for *ccsn_first_stage* and another for *ccsn_last_stage* separately. For example, for a two-input *and* cell, the CCS noise model is comprised of separate models for the first stage and for the last stage. This is illustrated next.

```
pin (OUT) {
  . . .
  timing () {
    related_pin : "IN1";
    . . .
    ccsn_first_stage() {
      /* IN1 to internal node between stages */
      . . .
    }
    ccsn_last_stage() { /* Internal node to output */
      . . .
    }
    . . .
  }
  timing () {
    related_pin : "IN2";
    . . .
    ccsn_first_stage() {
      /* IN2 to internal node between stages */
      . . .
    }
    ccsn_last_stage() {
      /* Internal node to output */
      /* Same as from IN1 */
      . . .
    }
    . . .
```

```
    }
    . . .
  }
```

The model within the *ccsn_last_stage* specified for *IN2* is the same as the model in *ccsn_last_stage* described for *IN1*.

Noise Models for Multi-stage and Sequential Cells

The CCS noise models for complex combinational or sequential cells are normally described as part of the pin specification. This is different from the one-stage or two-stage cells such as *nand*, *nor*, *and*, *or* where the CCS noise models are normally specified on a pin-pair basis as part of the timing arc. Complex multi-stage and sequential cells are normally described by a *ccsn_first_stage* model for all input pins and another *ccsn_last_stage* model at the output pins. The CCS noise models for these cells are not part of the timing arc but are normally specified for a pin.

If the internal paths between inputs and outputs are up to two CCB stages, the noise models can also be represented as part of the pin-pair timing arcs. In general, a multi-stage cell description can have some CCS noise models specified as part of the pin-pair timing arc while some other noise models can be specified with the pin description.

An example below has the CCS noise models specified with the pin description as well as part of the timing arc.

```
pin (CDN) {
  . . .
}
pin (CP) {
  . . .
  ccsn_first_stage() {
    . . .
  }
}
pin (D) {
```

```
    . . .
    ccsn_first_stage() {
      . . .
    }
  }
  pin (Q) {
    . . .
     timing() {
      related_pin : "CDN";
      . . .
      ccsn_first_stage() {
        . . .
      }
      ccsn_last_stage() {
        . . .
      }
    }
  }
  pin (QN) {
    . . .
    ccsn_last_stage() {
      . . .
    }
  }
```

Note that some of the CCS models for the flip-flop cell above are defined with the pins. Those defined with the pin specification at input pins are designated as *ccsn_first_stage,* and the CCS model at the output pin *QN* is designated as *ccsn_last_stage*. In addition, two-stage CCS noise models are described as part of the timing arc for *CDN* to *Q*. This example thus shows that a cell can have the CCS models designated as part of pin specification and as part of the timing group.

3.7.4 Other Noise Models

Apart from the CCS noise models described above, some cell libraries can provide other models for characterizing noise. Some of these models were utilized before the advent of the CCS noise models. These additional model are not required if the CCS noise models are available. We describe below some of these earlier noise models for completeness.

Models for DC margin: The DC margin refers to the largest DC variation allowed at the input pin of the cell which would keep the cell in its steady condition, that is, without causing a glitch at the output. For example, DC margin for input low refers to the largest DC voltage value at the input pin without causing any transition at the output.

Models for noise immunity: The noise immunity models specify the glitch magnitude that can be allowed at an input pin. These are normally described in terms of two-dimensional tables with the glitch width and output capacitance as the two indices. The values in the table correspond to the glitch magnitude that can be allowed at the input pin. This means that any glitch smaller than the specified magnitude and width will not propagate through the cell. Different variations of the noise immunity models can be specified such as:

i. *noise_immunity_high*

ii. *noise_immunity_low*

iii. *noise_immunity_above_high* (overshoot)

iv. *noise_immunity_below_low* (undershoot).

3.8 Power Dissipation Modeling

The cell library contains information related to power dissipation in the cells. This includes active power as well as standby or leakage power. As the names imply, the active power is related to the activity in the design whereas the standby power is the power dissipated in the standby mode, which is mainly due to leakage.

3.8.1 Active Power

The active power is related to the activity at the input and output pin of the cell. The active power in the cell is due to charging of the output load as well as internal switching. These two are normally referred to as *output switching power* and *internal switching power* respectively.

The output switching power is independent of the cell type and depends only upon the output capacitive load, frequency of switching and the power supply of the cell. The internal switching power depends upon the type of the cell and this value is thus included in the cell library. The specification of the internal switching power in the library is described next.

The internal switching power is referred to as *internal power* in the cell library. This is the power consumption within the cell when there is activity at the input or the output of the cell. For a combinational cell, an input pin transition can cause the output to switch and this results in internal switching power. For example, an inverter cell consumes power whenever the input switches (has a rise or fall transition at the input). The internal power is described in the library as:

```
pin (Z1) {
  . . .
  power_down_function : "!VDD + VSS";
  related_power_pin : VDD;
  related_ground_pin : VSS;
  internal_power () {
    related_pin : "A";
```

```
    power (template_2x2) {
     index_1 ("0.1, 0.4"); /* Input transition */
     index_2 ("0.05, 0.1"); /* Output capacitance */
     values ( /*   0.05    0.1 */ \
       /* 0.1 */   "0.045,  0.050", \
       /* 0.4 */   "0.055,  0.056");
    }
   }
  }
```

The example above shows the power dissipation from input pin *A* to the output pin *Z1* of the cell. The 2x2 table in the template is in terms of input transition at pin *A* and the output capacitance at pin *Z1*. Note that while the table includes the output capacitance, the table values only correspond to the internal switching and do not include the contribution of the output capacitance. The values represent the internal energy dissipated in the cell for each switching transition (rise or fall). The units are as derived from other units in the library (typically voltage is in volts (V) and capacitance is in picofarads (pF), and this maps to energy in picojoules (pJ)). The internal power in the library thus actually specifies the internal energy dissipated per transition.

In addition to the power tables, the example above also illustrates the specification of the power pins, ground pins, and the power down function which specifies the condition when the cell can be powered off. These constructs allow for multiple power supplies in the design and scenarios where different supplies may be powered down. The following illustration shows the power pin specification for each cell.

```
cell (NAND2) {
  . . .
  pg_pin (VDD) {
    pg_type : primary_power;
    voltage_name : COREVDD1;
    . . .
  }
```

```
    pg_pin (VSS) {
      pg_type : primary_ground;
      voltage_name : COREGND1;
      . . .
    }
  }
```

The power specification syntax allows for separate constructs for rise and fall power (referring to the output sense). Just like the timing arcs, the power specification can also be state-dependent. For example, the state-dependent power dissipation for an *xor* cell can be specified as dependent on the state of various inputs.

For combinational cells, the switching power is specified on an input-output pin pair basis. However for a sequential cell such as a flip-flop with complementary outputs, *Q* and *QN*, the *CLK*->*Q* transition also results in a *CLK*->*QN* transition. Thus, the library can specify the internal switching power as a three-dimensional table, which is shown next. The three dimensions in the example below are the input slew at *CLK* and the output capacitances at *Q* and *QN* respectively.

```
pin (Q) {
  . . .
  internal_power() {
  related_pin : "CLK";
  equal_or_opposite_output : "QN";
  rise_power(energy_template_3x2x2) {
    index_1 ("0.02, 0.2, 1.0"); /* Clock transition */
    index_2 ("0.005, 0.2"); /* Output Q capacitance */
    index_3 ("0.005, 0.2"); /* Output QN capacitance */
    values ( /*  0.005   0.2 */ /*   0.005    0.2 */ \
      /* 0.02 */ "0.060,  0.070",   "0.061,   0.068", \
      /* 0.2 */  "0.061,  0.071",   "0.063,   0.069", \
      /* 1.0 */  "0.062,  0.080",   "0.068,   0.075");
  }
  fall_power(energy_template_3x2x2) {
    index_1 ("0.02, 0.2, 1.0");
```

```
    index_2 ("0.005, 0.2");
    index_3 ("0.005, 0.2");
    values ( \
     "0.070, 0.080", "0.071, 0.078", \
     "0.071, 0.081", "0.073, 0.079", \
     "0.066, 0.082", "0.068, 0.085");
  }
}
```

Switching power can be dissipated even when the outputs or the internal state does not have a transition. A common example is the clock that toggles at the clock pin of a flip-flop. The flip-flop dissipates power with each clock toggle - typically due to switching of an inverter inside of the flip-flop cell. The power due to clock pin toggle is dissipated even if the flip-flop output does not switch. Thus for sequential cells, the input pin power refers to the power dissipation internal to the cell, that is, when the outputs do not transition. An example of the input pin power specification follows.

```
cell (DFF) {
  . . .
  pin (CLK) {
    . . .
    rise_power () {
     power (template_3x1) {
       index_1 ("0.1, 0.25, 0.4"); /* Input transition */
       values ( /*    0.1     0.25     0.4 */ \
                   "0.045,  0.050,   0.090");
     }
    }
    fall_power () {
     power (template_3x1) {
       index_1 ("0.1, 0.25, 0.4");
       values ( \
         "0.045, 0.050, 0.090");
     }
    }
  }
```

```
    . . .
}
```

This example shows the power specification when the *CLK* pin toggles. This represents the power dissipation due to clock switching even when the output does not switch.

Double Counting Clock Pin Power?

Note that a flip-flop also contains the power dissipation due to *CLK*->*Q* transition. It is thus important that the values in the *CLK*->*Q* power specification tables do not include the contribution due to the *CLK* internal power corresponding to the condition when the output *Q* does not switch.

The above guideline refers to consistency of usage of power tables by the application tool and ensures that the internal power specified due to clock input is not double-counted during power calculation.

3.8.2 Leakage Power

Most standard cells are designed such that the power is dissipated only when the output or state changes. Any power dissipated when the cell is powered but there is no activity is due to non-zero leakage current. The leakage can be due to subthreshold current for MOS devices or due to tunneling current through the gate oxide. In the earlier generations of CMOS process technologies, the leakage power has been negligible and has not been a major consideration during the design process. However, as the technology shrinks, the leakage power is becoming significant and is no longer negligible in comparison to active power.

As described above, the leakage power contribution is from two phenomena: subthreshold current in the MOS device and gate oxide tunneling. By using high Vt cells[1], one can reduce the subthreshold current; however,

1. The high Vt cells refer to cells with higher threshold voltage than the standard for the process technology.

there is a trade-off due to reduced speed of the high Vt cells. The high Vt cells have smaller leakage but are slower in speed. Similarly, the low Vt cells have larger leakage but allow greater speed. The contribution due to gate oxide tunneling does not change significantly by switching to high (or low) Vt cells. Thus, a possible way to control the leakage power is to utilize high Vt cells. Similar to the selection between high Vt and standard Vt cells, the strength of cells used in the design is a trade-off between leakage and speed. The higher strength cells have higher leakage power but provide higher speed. The trade-off related to power management are described in detail in Section 10.6.

The subthreshold MOS leakage has a strong non-linear dependence with respect to temperature. In most process technologies, the subthreshold leakage can grow by 10x to 20x as the device junction temperature is increased from 25C to 125C. The contribution due to gate oxide tunneling is relatively invariant with respect to temperature or the Vt of the devices. The gate oxide tunneling which was negligible at process technologies 100nm and above, has become a significant contributor to leakage at lower temperatures for 65nm or finer technologies. For example, gate oxide tunneling leakage may equal the subthreshold leakage at room temperature for 65nm or finer process technologies. At high temperatures, the subthreshold leakage continues to be the dominant contributor to leakage power.

Leakage power is specified for each cell in the library. For example, an inverter cell may contain the following specification:

```
cell_leakage_power : 1.366;
```

This is the leakage power dissipated in the cell - the leakage power units are as specified in the header of the library, typically in nanowatts. In general, the leakage power depends upon the state of the cell and state dependent values can be specified using the *when* condition.

For example, an *INV1* cell can have the following specification:

```
cell_leakage_power : 0.70;
leakage_power() {
  when : "!I";
  value : 1.17;
}

leakage_power() {
  when : "I";
  value : 0.23;
}
```

where *I* is the input pin of the *INV1* cell. It should be noted that the specification includes a default value (outside of the *when* conditions) and that the default value is generally the average of the leakage values specified within the *when* conditions.

3.9 Other Attributes in Cell Library

In addition to the timing information, a cell description in the library specifies area, functionality and the SDF condition of the timing arcs. These are briefly described in this section; for more details the reader is referred to the Liberty manual.

Area Specification

The **area** specification provides the area of a cell or cell group.

```
area : 2.35;
```

The above specifies that the area of the cell is 2.35 area units. This can represent the actual silicon area used by the cell or it can be a relative measure of the area.

Function Specification

The **function** specification specifies the functionality of a pin (or pin group).

```
pin (Z) {
  function: "IN1 & IN2";
  . . .
}
```

The above specifies the functionality of the *Z* pin of a two-input *and* cell.

SDF Condition

The SDF condition attribute supports the Standard Delay Format (SDF) file generation and condition matching during backannotation. Just as the *when* specifies the condition for the state-dependent models for timing analysis, the corresponding specification for state-dependent timing usage for SDF annotation is denoted by *sdf_cond*.

This is illustrated by the following example:

```
timing() {
  related_pin : "A1";
  when : "!A2";
  sdf_cond : "A2 == 1'b0";
  timing_sense : positive_unate;
  cell_rise(delay_template_7x7) {
    . . .
  }
}
```

3.10 Characterization and Operating Conditions

A cell library specifies the characterization and operating conditions under which the library is created. For example, the header of the library may contain the following:

```
nom_process : 1;
nom_temperature : 0;
nom_voltage : 1.1;
voltage_map(COREVDD1, 1.1);
voltage_map(COREGND1, 0.0);
operating_conditions("BCCOM"){
  process : 1;
  temperature : 0;
  voltage : 1.1;
  tree_type : "balanced_tree";
}
```

The nominal environmental conditions (designated as *nom_process*, *nom_temperature* and *nom_voltage*) specify the process, voltage and temperature under which the library was characterized. The operating conditions specify the conditions under which the cells from this library will get used. If the characterization and operating conditions are different, the timing values obtained during delay calculation need to be derated; this is accomplished by using the derating factor (k-factors) specified in the library.

The usage of derating to obtain timing values at a condition other than what was used for characterization introduces inaccuracy in timing calculation. The derating procedure is employed only if it is not feasible to characterize the library at the condition of interest.

What is the Process Variable?

Unlike temperature and voltage which are physical quantities, the process is not a quantifiable quantity. It is likely to be one of *slow*, *typical* or *fast* processes for the purposes of digital characterization and verification. Thus,

what does the process value of 1.0 (or any other value) mean? The answer is given below.

The library characterization is a time-consuming process and characterizing the library for various process corners can take weeks. The process variable setting allows a library characterized at a specific process corner be used for timing calculation for a different process corner. The k-factors for process can be used to derate the delays from the characterized process to the target process. As mentioned above, the use of derating factors introduces inaccuracy during timing calculation. Derating across process conditions is especially inaccurate and is rarely employed. To summarize, the only function of specifying different process values (say 1.0 or any other) is to allow derating across conditions which is rarely (if ever) employed.

3.10.1 Derating using K-factors

As described, the derating factors (referred to as **k-factors**) are used to obtain delays when the operating conditions are different from the characterization conditions. The k-factors are approximate factors. An example of the k-factors in a library are as specified below:

```
/* k-factors */
k_process_cell_fall             : 1;
k_process_cell_leakage_power    : 0;
k_process_cell_rise             : 1;
k_process_fall_transition       : 1;
k_process_hold_fall             : 1;
k_process_hold_rise             : 1;
k_process_internal_power        : 0;
k_process_min_pulse_width_high  : 1;
k_process_min_pulse_width_low   : 1;
k_process_pin_cap               : 0;
k_process_recovery_fall         : 1;
k_process_recovery_rise         : 1;
k_process_rise_transition       : 1;
k_process_setup_fall            : 1;
k_process_setup_rise            : 1;
```

```
k_process_wire_cap              : 0;
k_process_wire_res              : 0;
k_temp_cell_fall                : 0.0012;
k_temp_cell_rise                : 0.0012;
k_temp_fall_transition          : 0;
k_temp_hold_fall                : 0.0012;
k_temp_hold_rise                : 0.0012;
k_temp_min_pulse_width_high     : 0.0012;
k_temp_min_pulse_width_low      : 0.0012;
k_temp_min_period               : 0.0012;
k_temp_rise_propagation         : 0.0012;
k_temp_fall_propagation         : 0.0012;
k_temp_recovery_fall            : 0.0012;
k_temp_recovery_rise            : 0.0012;
k_temp_rise_transition          : 0;
k_temp_setup_fall               : 0.0012;
k_temp_setup_rise               : 0.0012;
k_volt_cell_fall                : -0.42;
k_volt_cell_rise                : -0.42;
k_volt_fall_transition          : 0;
k_volt_hold_fall                : -0.42;
k_volt_hold_rise                : -0.42;
k_volt_min_pulse_width_high     : -0.42;
k_volt_min_pulse_width_low      : -0.42;
k_volt_min_period               : -0.42;
k_volt_rise_propagation         : -0.42;
k_volt_fall_propagation         : -0.42;
k_volt_recovery_fall            : -0.42;
k_volt_recovery_rise            : -0.42;
k_volt_rise_transition          : 0;
k_volt_setup_fall               : -0.42;
k_volt_setup_rise               : -0.42;
```

These factors are used to obtain timing when the process, voltage or temperature for the operating conditions during delay calculation are different from the nominal conditions in the library. Note that *k_volt* factors are negative, implying that the delays reduce with increasing voltage supply,

whereas the *k_temp* factors are positive, implying that the delays normally increase with increasing temperature (except for cells exhibiting temperature inversion phenomenon described in Section 2.10). The k-factors are used as follows:

```
Result with derating = Original_value *
  ( 1 + k_process * DELTA_Process
      + k_volt * DELTA_Volt
      + k_temp * DELTA_Temp)
```

For example, assume that a library is characterized at 1.08V and 125C with *slow* process models. If the delays are to be obtained for 1.14V and 100C, the cell rise delays for the *slow* process models can be obtained as:

```
Derated_delay = Library_delay *
  ( 1 + k_volt_cell_rise * 0.06
      - k_temp_cell_rise * 25)
```

Assuming the k_factors outlined above are used, the previous equation maps to:

```
Derated_delay = Library_delay * (1 - 0.42 * 0.06 - 0.0012 * 25)
              = Library_delay * 0.9448
```

The delay at the derated condition works out to about 94.48% of the original delay.

3.10.2 Library Units

A cell description has all values in terms of library units. The units are declared in the library file using the Liberty command set. Units for voltage, time, capacitance, and resistance are declared as shown in the following example:

```
library("my_cell_library") {
  voltage_unit : "1V";
```

```
    time_unit : "1ns";
    capacitive_load_unit (1.000000, pf);
    current_unit : 1mA;
    pulling_resistance_unit : "1kohm";
    . . .
  }
```

In this text, we shall assume that the library time units are in nanoseconds (ns), voltage is in volts (V), internal power is in picojoules (pJ) per transition, leakage power is in nanowatts (nW), capacitance values are in picofarad (pF), resistance values are in Kohms and area units are square micron (μm^2), unless explicitly specified to help with an explanation.

❑

CHAPTER

4

Interconnect Parasitics

This chapter provides an overview of various techniques for handling and representing interconnect parasitics for timing verification of the designs. In digital designs, a wire connecting pins of standard cells and blocks is referred to as a net. A net typically has only one driver while it can drive a number of fanout cells or blocks. After physical implementation, the net can travel on multiple metal layers of the chip. Various metal layers can have different resistance and capacitance values. For equivalent electrical representation, a net is typically broken up into segments with each segment represented by equivalent parasitics. We refer to an interconnect trace as a synonym to a segment, that is, it is part of a net on a specific metal layer.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_4,

4.1 RLC for Interconnect

The interconnect resistance comes from the interconnect traces in various metal layers and vias in the design implementation. Figure 4-1 shows example nets traversing various metal layers and vias. Thus, the interconnect resistance can be considered as resistance between the output pin of a cell and the input pins of the fanout cells.

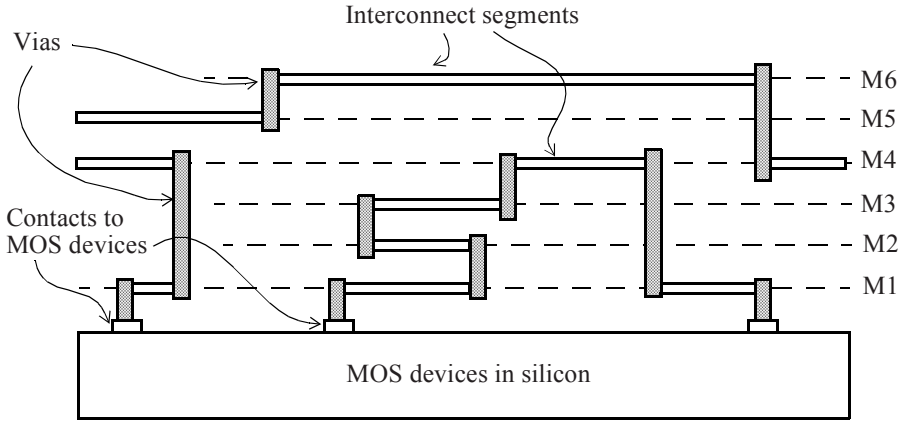
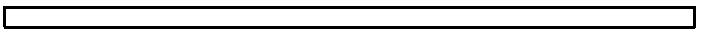


Figure 4-1 *Nets on metal layers.*

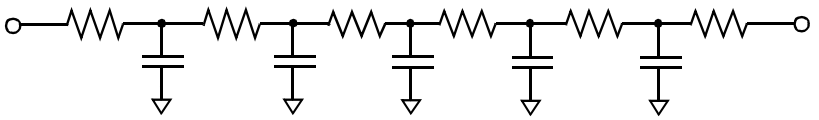
The interconnect capacitance contribution is also from the metal traces and is comprised of grounded capacitance as well as capacitance between neighboring signal routes.

The inductance arises due to *current* loops. Typically the effect of inductance can be ignored within the chip and is only considered for package and board level analysis. In chip level designs, the *current* loops are narrow and short - which means that the *current* return path is through a power or ground signal routed in close proximity. In most cases, the on-chip inductance is not considered for the timing analysis. Any further description of on-chip inductance analysis is beyond the scope of this book. Representation of interconnect resistance and capacitance is described next.

The resistance and capacitance (**RC**) for a section of interconnect trace is ideally represented by a distributed RC tree as shown in Figure 4-2. In this figure, the total resistance and capacitance of the RC tree - R_t and C_t respectively - correspond to $R_p * L$ and $C_p * L$ where R_p, C_p are per unit length values of interconnect resistance and capacitance for the trace and L is the trace length. The R_p, C_p values are typically obtained from the extracted parasitics for various configurations and is provided by the ASIC foundry.



(a) Trace of length L.



(b) Distributed RC tree.

Figure 4-2 *Interconnect trace.*

$$R_t = R_p * L$$
$$C_t = C_p * L$$

The RC interconnect can be represented by various simplified models. These are described in the subsections below.

T-model

In the T-model representation, the total capacitance C_t is modeled as connected halfway in the resistive tree. The total resistance R_t is broken in two sections (each being $R_t/2$), with the C_t connection represented at the midpoint of the resistive tree as shown in Figure 4-3.

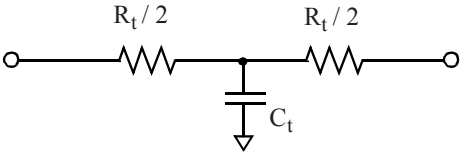


Figure 4-3 *T-model representation.*

Pi-model

In the Pi-model shown in Figure 4-4, the total capacitance C_t is broken into two sections (each being $C_t / 2$) and connected on either side of the resistance.

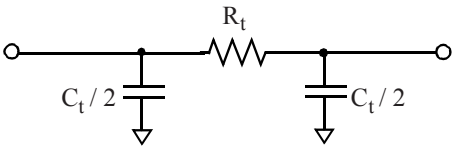


Figure 4-4 *Pi-model representation.*

More accurate representations of the distributed RC tree are obtained by breaking R_t and C_t into multiple sections. For N-sections, each of the intermediate sections of R and C are R_t / N and C_t / N. The end sections can be modeled along the concept of the T-model or the Pi-model. Figure 4-5 shows such an N-section with the end sections modeled using a T-model, while Figure 4-6 shows an N-section with the end sections modeled using the Pi-model.

With the broad overview of modeling of RC interconnect, we now describe how the parasitic interconnects are utilized during the pre-layout phase (by estimation) or post-layout phase (by detailed extraction). The next section describes the modeling of parasitic interconnect during the pre-layout process.

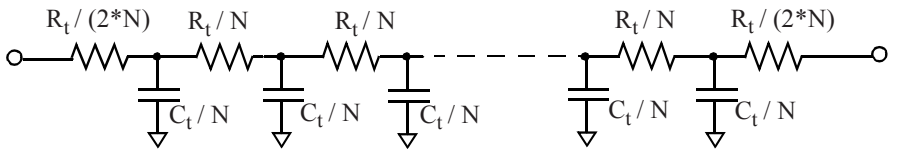


Figure 4-5 *R_t / (2 * N) on ends plus (N-1) sections of R_t / N.*

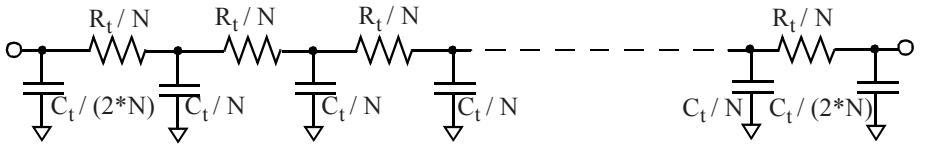


Figure 4-6 *C_t / (2 * N) on ends plus (N-1) sections of C_t / N.*

4.2 Wireload Models

Prior to floorplanning or layout, wireload models can be used to estimate capacitance, resistance and the area overhead due to interconnect. The wireload model is used to estimate the length of a net based upon the number of its fanouts. The wireload model depends upon the area of the block, and designs with different areas may choose different wireload models. The wireload model also maps the estimated length of the net into the resistance, capacitance and the corresponding area overhead due to routing.

The average wire length within a block correlates well with the size of the block; average net length increases as the block size is increased. Figure 4-7 shows that for different areas (chip or block size), different wireload models would typically be used in determining the parasitics. Thus, the figure depicts a smaller capacitance for the smaller sized block.

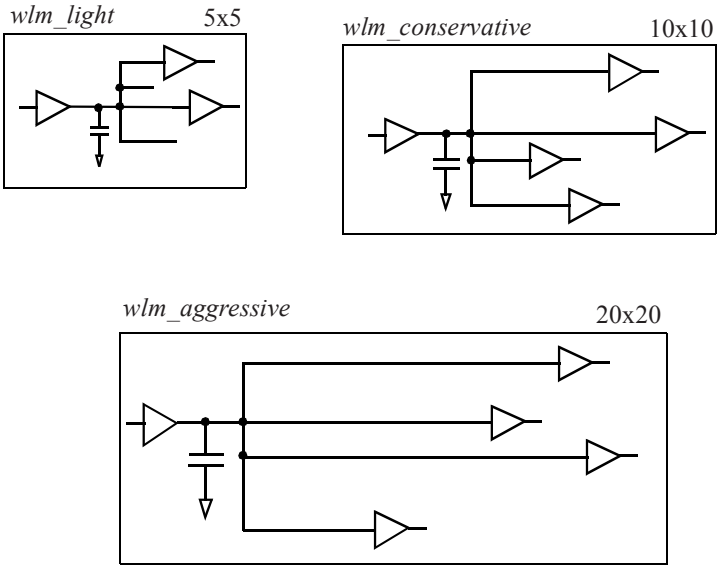


Figure 4-7 *Different wireload models for different areas.*

Here is an example of a wireload model.

```
wire_load ("wlm_conservative") {
  resistance : 5.0;
  capacitance : 1.1;
  area : 0.05;
  slope : 0.5;
  fanout_length (1, 2.6);
  fanout_length (2, 2.9);
  fanout_length (3, 3.2);
  fanout_length (4, 3.6);
  fanout_length (5, 4.1);
}
```

Resistance is resistance per unit length of the interconnect, *capacitance* is capacitance per unit length of the interconnect, and *area* is area overhead per

unit length of the interconnect. The *slope* is the extrapolation slope to be used for data points that are not specified in the fanout length table.

The wireload model illustrates how the length of the wire can be described as a function of fanout. The example above is depicted in Figure 4-8. For any fanout number not explicitly listed in the table, the interconnect length is obtained using linear extrapolation with the specified slope. For example, a fanout of 8 results in the following:

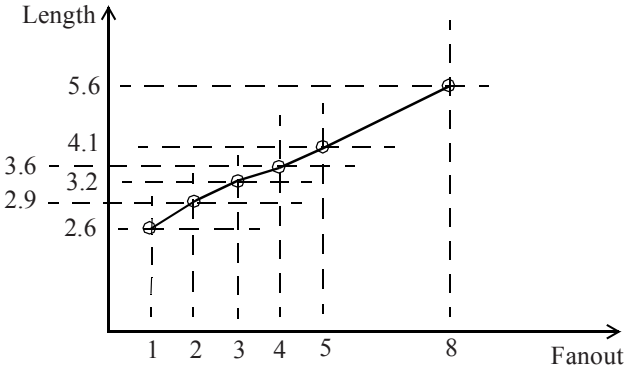


Figure 4-8 *Fanout vs wire length.*

```
Length = 4.1 + (8 - 5) * 0.5 = 5.6 units
Capacitance = Length * cap_coeff(1.1) = 6.16 units
Resistance = Length * res_coeff(5.0) = 28.0 units
Area overhead due to interconnect = Length * area_coeff(0.05)
  = 0.28 area units
```

The units for the length, capacitance, resistance and area are as specified in the library.

4.2.1 Interconnect Trees

Once the resistance and capacitance estimates, say R_{wire} and C_{wire}, of the pre-layout interconnect are determined, the next question is on the structure of the interconnect. How is the interconnect RC structure located with respect to the driving cell? This is important since the interconnect delay from a driver pin to a load pin depends upon how the interconnect is structured. In general, the interconnect delay depends upon the interconnect resistance and capacitance along the path. Thus, this delay can be different depending on the topology assumed for the net.

For pre-layout estimation, the interconnect RC tree can be represented using one of the following three different representations (see Figure 4-9). Note that the total interconnect length (and thus the resistance and capacitance estimates) is the same in each of the three cases.

- *Best-case tree*:

  In the best-case tree, it is assumed that the destination (load) pin is physically adjacent to the driver. Thus, none of the wire resistance is in the path to the destination pin. All of the wire capacitance and the pin capacitances from other fanout pins still act as load on the driver pin.

- *Balanced tree*:

  In this scenario, it is assumed that each destination pin is on a separate portion of the interconnect wire. Each path to the destination sees an equal portion of the total wire resistance and capacitance.

- *Worst-case tree*:

  In this scenario, it is assumed that all the destination pins are together at the far end of the wire. Thus each destination pin sees the total wire resistance and the total wire capacitance.

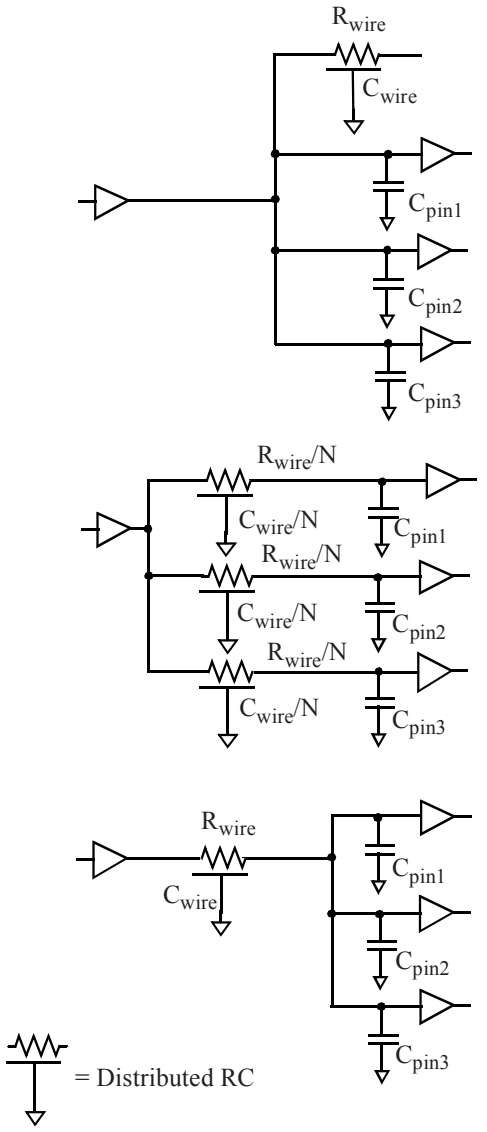


Figure 4-9 *RC tree representations used during pre-layout.*

4.2.2 Specifying Wireload Models

A wireload model is specified using the following command:

```
set_wire_load_model "wlm_cons" -library "lib_stdcell"
# Says to use the wireload model wlm_cons present in the
# cell library lib_stdcell.
```

When a net crosses a hierarchical boundary, different wireload models can be applied to different parts of the net in each hierarchical boundary based upon the **wireload mode**. These wireload modes are:

i. top

ii. enclosed

iii. segmented

The wireload mode can be specified using the *set_wire_load_mode* specification as shown below.

```
set_wire_load_mode enclosed
```

In the *top* wireload mode, all nets within the hierarchy inherit the wireload model of the top-level, that is, any wireload models specified in lower-level blocks are ignored. Thus, the top-level wireload model takes precedence. For the example shown in Figure 4-10, the *wlm_cons* wireload model specified in block *B1* takes precedence over all the other wireload models specified in blocks *B2*, *B3* and *B4*.

In the *enclosed* wireload mode, the wireload model of the block that fully encompasses the net is used for the entire net. For the example shown in Figure 4-11, the net *NETQ* is subsumed in block *B2* and thus the wireload model of block *B2*, *wlm_light*, is used for this net. Other nets which are fully contained in block *B3* use the *wlm_aggr* wireload model, whereas nets fully contained within block *B5* use the *wlm_typ* wireload model.

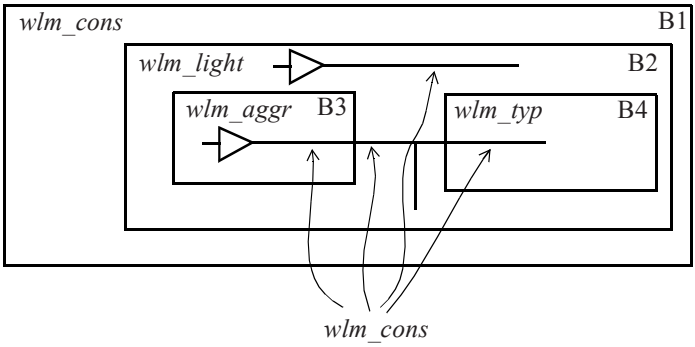


Figure 4-10 *TOP wireload mode.*

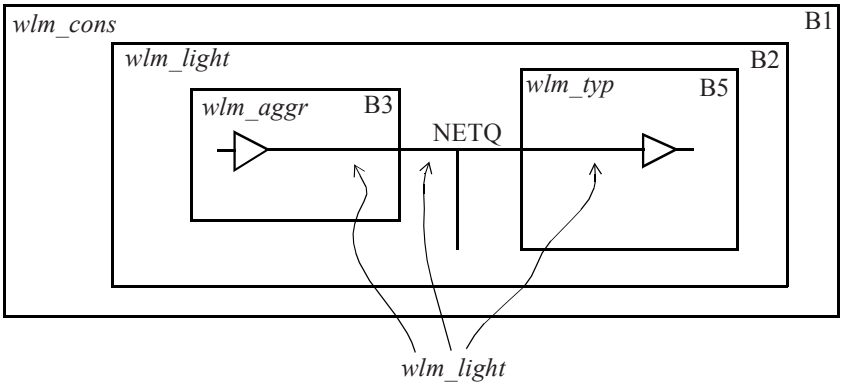


Figure 4-11 *ENCLOSED wireload mode.*

In the *segmented* wireload mode, each segment of the net gets its wireload model from the block that encompasses the net segment. Each portion of the net uses the appropriate wireload model within that level. Figure 4-12 illustrates an example of a net *NETQ* that has segments in three blocks. The interconnect for the fanout of this net within block *B3* uses the wireload model *wlm_aggr*, the segment of net within block *B4* utilizes the wireload

model *wlm_typ*, and the segment within block *B2* uses the wireload model *wlm_light*.

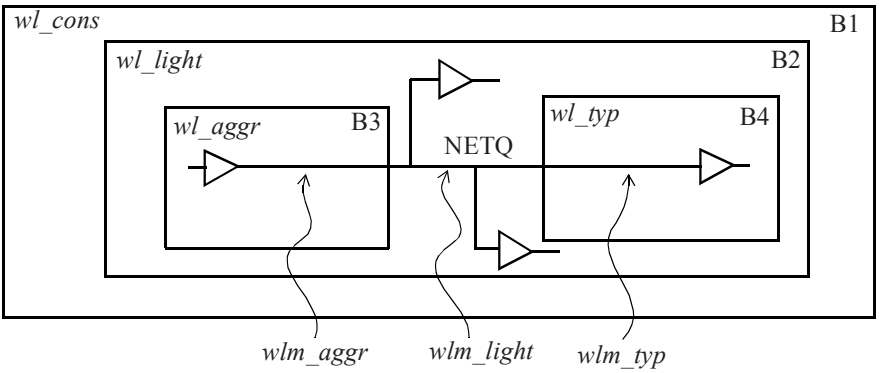


Figure 4-12 *SEGMENTED wireload mode.*

Typically a wireload model is selected based upon the chip area of the block. However these can be modified or changed at the user's discretion. For example, one can select the wireload model *wlm_aggr* for a block area between 0 and 400, the wireload model *wlm_typ* for area between 400 and 1000, and the wireload model *wlm_cons* for area 1000 or higher. Wireload models are typically defined in a cell library - however a user can define a custom wireload model as well. A **default wireload model** may optionally be specified in the cell library as:

```
default_wire_load: "wlm_light";
```

A **wireload selection group**, which selects a wireload model based upon area, is defined in a cell library. Here is one such example:

```
wire_load_selection (WireAreaSelGrp){
  wire_load_from_area(0, 50000, "wlm_light");
  wire_load_from_area(50000, 100000, "wlm_cons");
  wire_load_from_area(100000, 200000, "wlm_typ");
```

```
    wire_load_from_area(200000, 500000, "wlm_aggr");
  }
```

A cell library can contain many such selection groups. A particular one can be selected for use in STA by using the *set_wire_load_selection_group* specification.

```
set_wire_load_selection_group WireAreaSelGrp
```

This section described the modeling of estimated parasitics before the physical implementation, that is, during the pre-layout phase. The next section describes the representation of parasitics extracted from the layout.

4.3 Representation of Extracted Parasitics

Parasitics extracted from a layout can be described in three formats:

i. Detailed Standard Parasitic Format (DSPF)

ii. Reduced Standard Parasitic Format (RSPF)

iii. Standard Parasitic Extraction Format (SPEF)

Some tools provide a proprietary binary representation of the parasitics, such as SBPF; this helps in keeping the file size smaller and speeds up the reading of the parasitics by the tools. A brief description of each of the above three formats follows.

4.3.1 Detailed Standard Parasitic Format

In the DSPF representation, the detailed parasitics are represented in SPICE[1] format. The SPICE *Comment* statements are used to indicate the cell

1. Format that is readable by a circuit simulator, such as SPICE. Refer to [NAG75] or any book on analog integrated circuits design or simulation for further information.

types, cell pins and their capacitances. The resistance and capacitance values are in standard SPICE syntax and the cell instantiations are also included in this representation. The advantage of this format is that the DSPF file can be used as an input to a SPICE simulator itself. However, the drawback is that the DSPF syntax is too detailed and verbose with the result that the total file size for a typical block is very large. Thus, this format is rarely employed in practice for anything but a relatively small group of nets.

Here is an example DSPF file that describes the interconnect from a primary input *IN* to the input pin *A* of buffer *BUF*, and another net from output pin *OUT* of *BUF* to the primary output pin *OUT*.

```
.SUBCKT TEST_EXAMPLE OUT IN
* Net Section
*|GROUND_NET VSS
*|NET IN 4.9E-02PF
*|P (IN I 0.0 0.0 4.1)
*|I (BUF1:A BUF A I 0.0 0.7 4.3)
C1 IN VSS 2.3E-02PF
C2 BUF1:A VSS 2.6E-02PF
R1 IN BUF1:A 4.8E00
*|NET OUT 4.47E-02PF
*|S (OUT:1 8.3 0.7)
*|P (OUT O 0.0 8.3 0.0)
*|I (BUF1:OUT BUF1 OUT O 0.0 4.9 0.7)
C3 BUF1:OUT VSS 3.5E-02PF
C4 OUT:1 VSS 4.9E-03PF
C5 OUT VSS 4.8E-03PF
R2 BUF1:OUT OUT:1 12.1E00
R3 OUT:1 OUT 8.3E00
*Instance Section
X1 BUF1:A BUF1:OUT BUF
.ENDS
```

The nonstandard SPICE statements in DSPF are comments that start with "*|" and have the following format:

```
*|I(InstancePinName InstanceName PinName PinType PinCap X Y)
*|P(PinName PinType PinCap X Y)
```

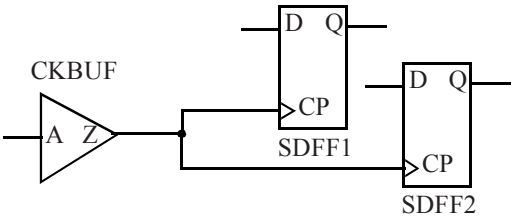
```
*|NET NetName NetCap
*|S(SubNodeName X Y)
*|GROUND_NET NetName
```

4.3.2 Reduced Standard Parasitic Format

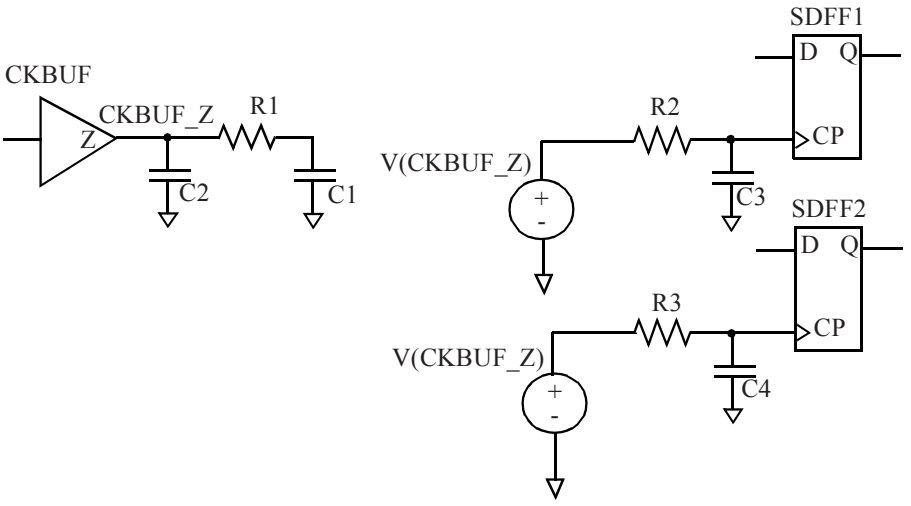
In the RSPF representation, the parasitics are represented in a reduced form. The reduced format involves a voltage source and a controlled current source. The RSPF format is also a SPICE file since it can be read into a SPICE-like simulator. The RSPF format requires that the detailed parasitics are reduced and mapped into the reduced format. This is thus a drawback of the RSPF representation since the focus of the parasitic extraction process is normally on the extraction accuracy and not on the reduction to a compact format like RSPF. One other limitation of the RSPF representation is that the bidirectional signal flow cannot be represented in this format.

Here is an example of an RSPF file. The original design and the equivalent representation are shown in Figure 4-13.

```
* Design Name : TEST1
* Date : 7 September 2002
* Time : 02:00:00
* Resistance Units : 1 ohms
* Capacitance Units : 1 pico farads
*| RSPF 1.0
*| DELIMITER "_"
.SUBCKT TEST1 OUT IN
*| GROUND_NET VSS
*|NET CP 0.075PF
*|DRIVER CKBUF_Z CKBUF Z
*|S (CKBUF_Z_OUTP 0.0 0.0)
R1 CKBUF_Z CKBUF_Z_OUTP 8.85
C1 CKBUF_Z_OUTP VSS 0.05PF
C2 CKBUF_Z VSS 0.025PF
*|LOAD SDFF1_CP SDFF1 CP
*|S (SDFF1_CP_INP 0.0 0.0)
E1 SDFF1_CP_INP VSS CKBUF_Z VSS 1.0
R2 SDFF1_CP_INP SDFF1_CP 52.0
C3 SDFF1_CP VSS 0.1PF
```



(a) Example logic.



(b) RSPF representation.

Figure 4-13 *RSPF example representation.*

```
*|LOAD SDFF2_CP SDFF2 CP
*|S (SDFF2_CP_INP 0.0 0.0)
E2 SDFF2_CP_INP VSS CKBUF_Z VSS 1.0
R3 SDFF2_CP_INP SDFF2_CP 43.5
C4 SDFF2_CP VSS 0.1PF
*Instance Section
X1 SDFF1_Q SDFF1_QN SDFF1_D SDFF1_CP SDFF1_CD VDD VSS SDFF
X2 SDFF2_Q SDFF2_QN SDFF2_D SDFF2_CP SDFF2_CD VDD VSS SDFF
X3 CKBUF_Z CKBUF_A VDD VSS CKBUF
.ENDS
.END
```

This file has the following features:

- The pin-to-pin interconnect delays are modeled at each of the fanout cell inputs with a capacitor of value 0.1pF (*C3* and *C4* in the example) and a resistor (*R2* and *R3*). The resistor value is chosen such that the RC delay corresponds to the pin-to-pin interconnect delay. The PI segment load at the output of the driving cell models the correct cell delay through the cell.
- The RC elements at the gate inputs are driven by ideal voltage sources (*E1* and *E2*) that are equal to the voltage at the output of the driving gate.

4.3.3 Standard Parasitic Exchange Format

The SPEF is a compact format which allows the representation of the detailed parasitics. An example of a net with two fanouts is shown below.

```
*D_NET NET_27 0.77181
*CONN
*I *8:Q O *L 0 *D CELL1
*I *10:I I *L 12.3
*CAP
1 *9:0 0.00372945
2 *9:1 0.0206066
3 *9:2 0.035503
4 *9:3 0.0186259
5 *9:4 0.0117878
6 *9:5 0.0189788
7 *9:6 0.0194256
8 *9:7 0.0122347
9 *9:8 0.00972101
10 *9:9 0.298681
11 *9:10 0.305738
12 *9:11 0.0167775
*RES
1 *9:0 *9:1 0.0327394
2 *9:1 *9:2 0.116926
3 *9:2 *9:3 0.119265
4 *9:4 *9:5 0.0122066
```

```
5 *9:5 *9:6 0.0122066
6 *9:6 *9:7 0.0122066
7 *9:8 *9:9 0.142205
8 *9:9 *9:10 3.85904
9 *9:10 *9:11 0.142205
10 *9:12 *9:2 1.33151
11 *9:13 *9:6 1.33151
12 *9:1 *9:9 1.33151
13 *9:5 *9:10 1.33151
14 *9:12 *8:Q 0
15 *9:13 *10:I 0
*END
```

The units of the parasitics *R* and *C* are specified at the beginning of the SPEF file. A more detailed description of the SPEF is provided in Appendix C. Due to its compactness and completeness of representation, SPEF is the format of choice for representing the parasitics in a design.

4.4 Representing Coupling Capacitances

The previous section illustrated the cases where the capacitances of the net are represented as grounded capacitances. Since most of the capacitances in the nanometer technologies are sidewall capacitances, the proper representation for these capacitances is the signal to signal coupling capacitance.

The representation of coupling capacitances in DSPF is as an add-on to the original DSPF standard and is thus not unique. The coupling capacitances are replicated between both sets of coupled nets. This implies that the DSPF cannot be directly read into SPICE because of the duplication of the coupled capacitance in both set of nets. Some tools which output DSPF resolved this discrepancy by including half of the coupling capacitance in both of the coupled nets.

The RSPF is a reduced representation and thus not amenable to representing coupling capacitances.

The SPEF standard handles the coupling capacitances in a uniform and unambiguous manner and is thus the extraction format of choice when crosstalk timing is of interest. Further, the SPEF is a compact representation in terms of file size and is used for representing parasitics with and without coupling.

As described in Appendix C, one of the mechanisms to manage the file size is by having a name directory in the beginning of the file. Many extraction tools now specify a directory of net names (mapping net names to indices) in the beginning of the SPEF file so that the verbosity of repeating the net names is avoided. This reduces the file size significantly. An example with the name directory is shown in the appendix on SPEF.

4.5 Hierarchical Methodology

The large and complex designs generally require hierarchical methodology during the physical design process for the parasitic extraction and timing verification. In such scenarios, the parasitics of a block are extracted at the block level and can then be utilized at the higher levels of hierarchy.

The layout extracted parasitics of a block may be utilized for timing verification with another block whose layout has not been completed. In this scenario, layout extracted parasitics of layout-complete blocks are often used with wireload model based parasitic estimates for the pre-layout blocks.

In the case of the hierarchical flow, where the top level layout is complete but the blocks are still represented as black boxes (pre-layout), wireload model based parasitic estimates can be used for the lower level blocks along with the layout extracted parasitics for the top level. Once the layout of the blocks is complete, layout extracted parasitics for the top and the blocks can be stitched together.

Block Replicated in Layout

If a design block is replicated multiple times in layout, the parasitic extraction for one instantiation can be utilized for all instantiations. This requires that the layout of the block be identical in all respects for various instantiations of the block. For example, there should be no difference in terms of layout environment as seen from the nets routed within the block. This implies that the block level nets are not capacitively coupled with any nets outside the block. One way this can be achieved is by ensuring that no top level nets are routed over the blocks and there is adequate shielding or spacing for the nets routed near the boundary of the block.

4.6 Reducing Parasitics for Critical Nets

This section provides a brief outline of the common techniques to manage the impact of parasitics for critical nets.

Reducing Interconnect Resistance

For critical nets, it is important to maintain low slew values (or fast transition times), which implies that the interconnect resistance should be reduced. Typically, there are two ways of achieving low resistance:

- *Wide trace*: Having a trace wider than the minimum width reduces interconnect resistance without causing a significant increase in the parasitic capacitance. Thus, the overall RC interconnect delay and the transition times are reduced.
- *Routing in upper (thicker) metals*: The upper metal layer(s) normally have low resistivity which can be utilized for routing the critical signals. The low interconnect resistance reduces the interconnect delay as well as the transition times at the destination pins.

Increasing Wire Spacing

Increasing the spacing between traces reduces the amount of coupling (and total) capacitance of the net. Large coupling capacitance increases the crosstalk whose avoidance is an important consideration for nets routed in adjacent traces over a long distance.

Parasitics for Correlated Nets

In many cases, a group of nets have to be matched in terms of timing. An example is the data signals within a byte lane of a high speed DDR interface. Since it is important that all signals within a byte lane see identical parasitics, the signals are all routed in the same metal layer. For example, while metal layers *M2* and *M3* have the same average and the same statistical variations, the variations are independent so that the parasitic variations in these two metal layers do not track each other. Thus, if it is important for timing to match for critical signals, the routing must be identical in each metal layer.

❑

CHAPTER

5

Delay Calculation

This chapter provides an overview of the delay calculation of cell-based designs for the pre-layout and post-layout timing verification. The previous chapters have focused on the modeling of the interconnect and the cell library. The cell and interconnect modeling techniques are utilized to obtain timing of the design.

5.1 Overview

5.1.1 Delay Calculation Basics

A typical design comprises of various combinational and sequential cells. We use an example logic fragment shown in Figure 5-1 to describe the concept of delay calculation.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_5, © Springer Science + Business Media, LLC 2009

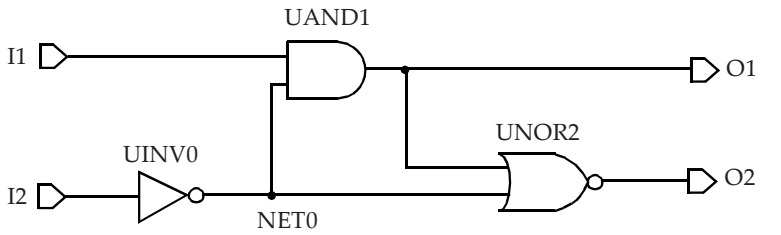


Figure 5-1 *Schematic of an example logic block.*

The library description of each cell specifies the pin capacitance values for each of the input pins[1]. Thus, every net in the design has a capacitive load which is the sum of the pin capacitance loads of every fanout of the net plus any contribution from the interconnect. For the purposes of simplicity, the contributions from the interconnect are not considered in this section - those are described in the later sections. Without considering the interconnect parasitics, the internal net *NET0* in Figure 5-1 has a net capacitance which is comprised of the input pin capacitances from the *UAND1* and *UNOR2* cells. The output *O1* has the pin capacitance of the *UNOR2* cell plus any capacitive loading for the output of the logic block. Inputs *I1* and *I2* have pin capacitances corresponding to the *UAND1* and *UINV0* cells. With such an abstraction, the logic design in Figure 5-1 can be described by an equivalent representation shown in Figure 5-2.

As described in Chapter 3, the cell library contains NLDM timing models for various timing arcs. The non-linear models are represented as two-dimensional tables in terms of input transition time and output capacitance. The output transition time of a logic cell is also described as a two-dimensional table in terms of input transition and total output capacitance at the net. Thus, if the input transition time (or slew) is specified at the inputs of the logic block, the output transition time and delays through the timing arcs of the *UINV0* cell and *UAND1* cell (for the input *I1*) can be obtained from the library cell descriptions. Extending the same approach through the fanout cells, the transition times and delays through the other timing

1. The standard cell libraries normally do not specify pin capacitances for cell outputs.

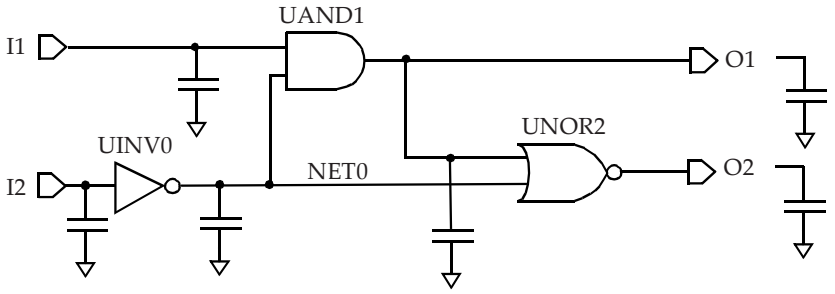


Figure 5-2 *Logic block representation depicting capacitances.*

arc of the *UAND1* cell (from *NET0* to *O1*) and through the *UNOR2* cell can be obtained. For a multi-input cell (such as *UAND1*), different input pins can provide different values of output transition times. The choice of transition time used for the fanout net depends upon the slew merge option and is described in Section 5.4. Using the approach described above, the delays through any logic cell can be obtained based upon the transition time at the input pins and the capacitance present at the output pins.

5.1.2 Delay Calculation with Interconnect

Pre-layout Timing

As described in Chapter 4, the interconnect parasitics are estimated using wireload models during the pre-layout timing verification. In many cases, the resistance contribution in the wireload models is set to 0. In such scenarios, the wireload contribution is purely capacitive and the delay calculation methodology described in the previous section is applicable to obtain the delays for all the timing arcs in the design.

In cases where the wireload models include the effect of the resistance of the interconnect, the NLDM models are used with the total net capacitance for the delay through the cell. Since the interconnect is resistive, there is additional delay from the output of the driving cell to the input pin of the

fanout cell. The interconnect delay calculation process is described in Section 5.3.

Post-layout Timing

The parasitics of the metal traces map into an RC network between driver and destination cells. Using the example of Figure 5-1, the interconnect resistance of the nets is illustrated in Figure 5-3. An internal net such as *NET0* in Figure 5-1 maps into multiple subnodes as shown in Figure 5-3. Thus, the output load of the inverter cell *UINV0* is comprised of an RC structure. Since the NLDM tables are in terms of input transition and output capacitance, the resistive load at the output pin implies that the NLDM tables are not directly applicable. Using the NLDM with resistive interconnect is described in the next section.

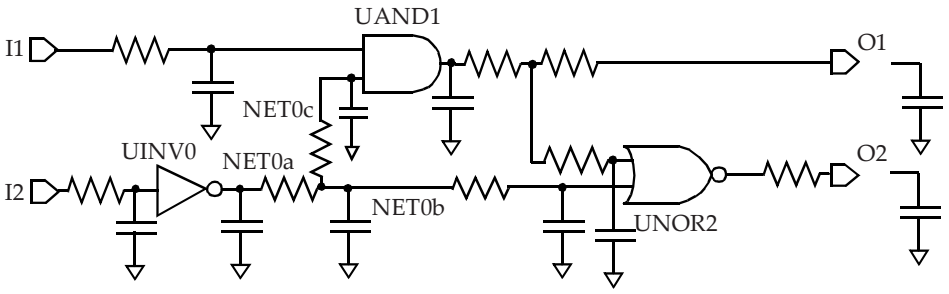


Figure 5-3 *Logic block representation depicting resistive nets.*

5.2 Cell Delay using Effective Capacitance

As described above, the NLDM models are not directly usable when the load at the output of the cell includes the interconnect resistance. Instead, an "effective" capacitance approach is employed to handle the effect of resistance.

The effective capacitance approach attempts to find a single capacitance that can be utilized as the equivalent load so that the original design as well the design with equivalent capacitance load behave similarly in terms of timing at the output of the cell. This equivalent single capacitance is termed as **effective capacitance**.

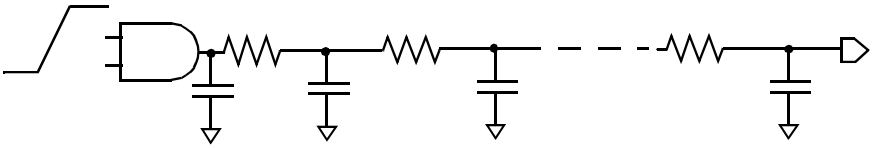
Figure 5-4(a) shows a cell with an RC interconnect at its fanout. The RC interconnect is represented by an equivalent RC PI-network shown in Figure 5-4(b). The concept of effective capacitance is to obtain an equivalent output capacitance *Ceff* (shown in Figure 5-4(c)) which has the same delay through the cell as the original design with the RC load. In general, the cell output waveform with the RC load is very different from the waveform with a single capacitive load.

Figure 5-5 shows representative waveforms at the output of the cell with total capacitance, effective capacitance and the waveform with the actual RC interconnect. The effective capacitance *Ceff* is selected so that the delay (as measured at the midpoint of the transition) at the output of the cell in Figure 5-4(c) is the same as the delay in Figure 5-4(a). This is illustrated in Figure 5-5.

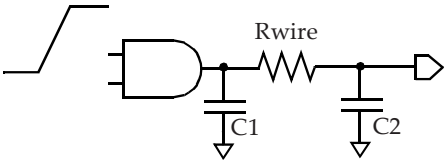
In relation to the PI-equivalent representation, the effective capacitance can be expressed as:

```
Ceff = C1 + k * C2, 0 <= k <= 1
```

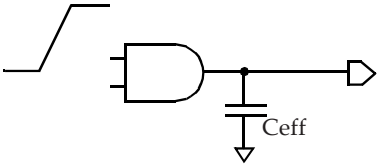
where *C1* is the near-end capacitance and *C2* is the far-end capacitance as shown in Figure 5-4(b). The value of *k* lies between 0 and 1. In the scenario where the interconnect resistance is negligible, the effective capacitance is nearly equal to the total capacitance. This is directly explainable by setting *R* to 0 in Figure 5-4(b). Similarly, if the interconnect resistance is relatively large, the effective capacitance is almost equal to the near-end capacitance *C1* (Figure 5-4(b)). This can be explained by increasing *R* to the limiting case where *R* becomes infinite (essentially an open circuit).



(a) RC interconnect.



(b) PI model.



(c) Effective capacitance.

Figure 5-4 *RC interconnect with effective capacitance.*

The effective capacitance is a function of:

i. the driving cell, and

ii. the characteristics of the load or specifically the input impedance of the load as seen from the driving cell.

For a given interconnect, a cell with a weak output drive will see a larger effective capacitance than a cell with a strong drive. Thus, the effective capacitance will be between the minimum value of *C1* (for high interconnect resistance or strong driving cell) and the maximum value which is the

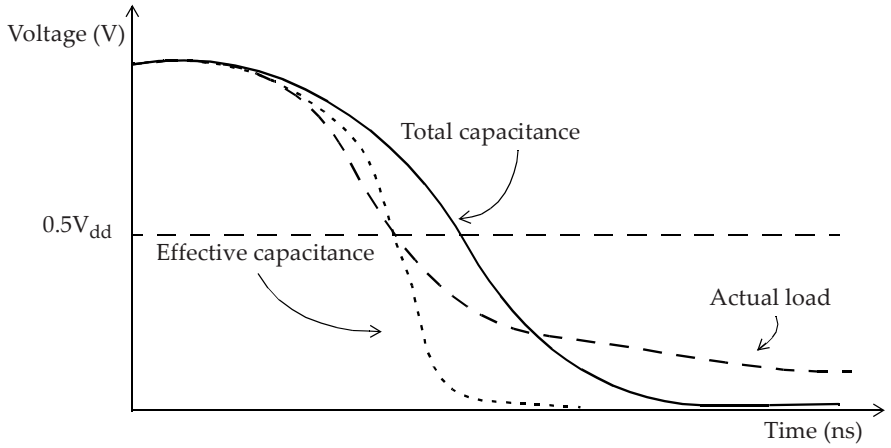


Figure 5-5 *Waveform at the output of the cell with various loads.*

same as the total capacitance when the interconnect resistance is negligible or the driving cell is weak. Note that the destination pin transitions later than the output of the driving cell. The phenomenon of near-end capacitance charging faster than the far-end capacitance is also referred to as the *resistive shielding effect* of the interconnect, since only a portion of the far-end capacitance is seen by the driving cell.

Unlike the computation of the delay by direct lookup of the NLDM models in the library, the delay calculation tools obtain the effective capacitance by an iterative procedure. In terms of the algorithm, the first step is to obtain the driving point impedance seen by the cell output for the actual RC load. The driving point impedance for the actual RC load is calculated using any of the methods such as second order AWE or Arnoldi algorithms[1]. The next step in computing effective capacitance is equating the charge transferred until the midpoint of the transition in the two scenarios. The charge transferred at the cell output when using the actual RC load (based upon driving point impedance) is matched with the charge transfer when using

1. See [ARN51] in Bibliography.

the effective capacitance as the load. Note that the charge transfer is matched only until the midpoint of the transition. The procedure starts with an estimate of the effective capacitance and then iteratively updates the estimate. In most practical scenarios, the effective capacitance value converges within a small number of iterations.

The effective capacitance approximation is thus a good model for computing delay through the cell. However, the output slew obtained using effective capacitance does not correspond to the actual waveform at the cell output. The waveform at the cell output, especially for the trailing half of the waveform, is not represented by the effective capacitance approximation. Note that in a typical scenario, the waveform of interest is not at the cell output but at the destination points of the interconnect which are the input pins of the fanout cells.

There are various approaches to compute the delay and the waveform at the destination points of the interconnect. In many implementations, the effective capacitance procedure also computes an equivalent Thevenin voltage source for the driving cell. The Thevenin source comprises of a ramp source with a series resistance *Rd* as shown in Figure 5-6. The series resistance *Rd* corresponds to the pull-down (or pull-up) resistance of the output stage of the cell.

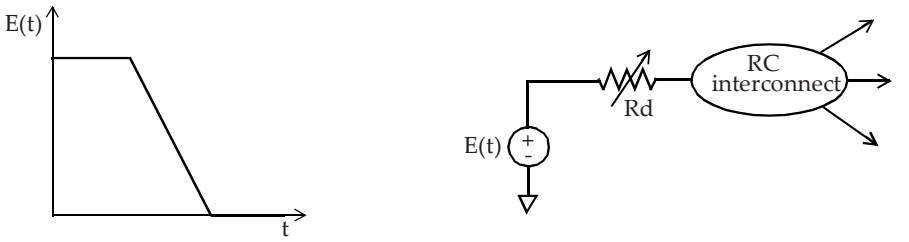


Figure 5-6 *Thevenin source model for driving cell.*

This section described the computation of the delay through the driving cell with an RC interconnect using the effective capacitance. The effective capacitance computation also provides the equivalent Thevenin source

model which is then used to obtain the timing through the RC interconnect. The process of obtaining timing through the interconnect is described next.

5.3 Interconnect Delay

As described in Chapter 4, the interconnect parasitics of a net are normally represented by an RC circuit. The RC interconnect can be pre-layout or post-layout. While the post-layout parasitic interconnect can include coupling to neighboring nets, the basic delay calculation treats all capacitances (including coupling capacitances) as capacitances to ground. An example of parasitics of a net along with its driving cell and a fanout cell is shown in Figure 5-7.

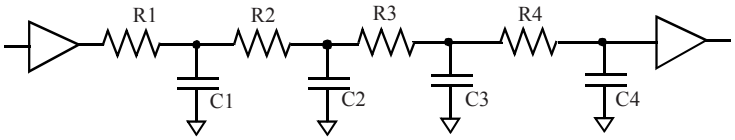


Figure 5-7 *Parasitics of a net.*

Using the effective capacitance approach, the delay through the driving cell and through the interconnect are obtained separately. The effective capacitance approach provides the delay through the driving cell as well as the equivalent Thevenin source at the output of the cell. The delay through the interconnect is then computed separately using the Thevenin source. The interconnect portion has one input and as many outputs as the destination pins. Using the equivalent Thevenin voltage source at the input of the interconnect, the delays to each of the destination pins are computed. This is illustrated in Figure 5-6.

For pre-layout analysis, the RC interconnect structure is determined by the tree type, which in turn determines the net delay. The three types of interconnect tree representations are described in detail in Section 4.2. The selected tree type is normally defined by the library. In general, the worst-

case slow library would select the worst-case tree type since that tree type provides the largest interconnect delay. Similarly, the best-case tree structure, which does not include any resistance from the source pin to the destination pins, is normally selected for the best-case fast corner. The interconnect delay for the best-case tree type is thus equal to zero. The interconnect delay for the typical-case tree and worst-case tree are handled just like for the post-layout RC interconnect.

Elmore Delay

Elmore delays are applicable for RC trees. What is an RC tree? An **RC tree** meets the following three conditions:

- Has a single input (source) node.
- Does not have any resistive loops.
- All capacitances are between a node and ground.

Elmore delay can be considered as finding the delay through each segment, as the *R* times the downstream capacitance, and then taking the sum of the delays from the root to the sink.

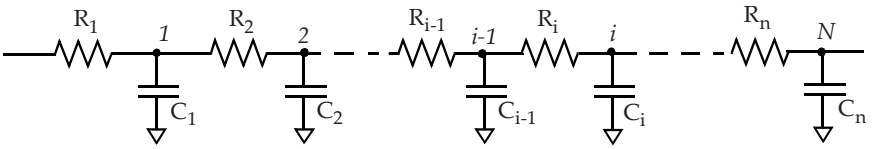


Figure 5-8 *Elmore delay model.*

The delays to various intermediate nodes are represented as:

```
Td1 = C1 * R1;
Td2 = C1 * R1 + C2 * (R1 + R2);
. . .
Tdn = Σ(i=1,N) Ci (Σ(j=1,i) Rj); # Elmore delay equation
```

Elmore delay is mathematically identical to considering the first moment of the impulse response[1]. We now apply the Elmore delay model to a simplified representation of a wire with R_{wire} and C_{wire} as the parasitics, plus a load capacitance C_{load} to model the pin capacitance at the far-end of the wire. The equivalent RC network can be simplified as a PI network model or a T-representation as shown in Figure 4-4 and Figure 4-3 respectively. Either representation provides the net delay (based upon Elmore delay equation) as:

$$R_{wire} * (C_{wire} / 2 + C_{load})$$

This is because the C_{load} sees the entire wire resistance in its charging path, whereas the C_{wire} capacitance sees $R_{wire} / 2$ for the T-representation, and $C_{wire} / 2$ sees R_{wire} in its charging path for the PI-representation. The above approach can be extended to a more complex interconnect structure also.

An example of Elmore delay calculation for a net using wireload model with balanced RC tree (and also worst-case RC tree) is given below.

Using a balanced tree model, the resistance and capacitance of a net is divided equally among the branches of the net (assuming a fanout of *N*). For a branch with pin load *Cpin*, the delay using balanced tree is:

$$\text{net delay} = (R_{wire} / N) * (C_{wire} / (2 * N) + C_{pin})$$

In the worst-case tree model, the resistance and entire capacitance of the net is considered for each endpoint of the net. Here *Cpins* is the total pin load from all fanouts:

$$\text{Net delay} = R_{wire} * (C_{wire} / 2 + C_{pins})$$

Figure 5-9 shows an example design.

1. See [RUB83] in Bibliography.

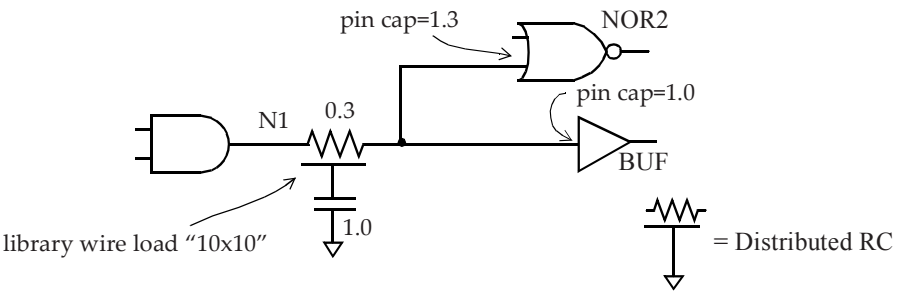


Figure 5-9 *Net delay for worst-case tree using Elmore model.*

If we use the worst-case tree model to calculate the delay for net *N1*, we get:

$$\text{Net delay} = R_{wire} * (C_{wire}/2 + C_{pins})$$
$$= 0.3 * (0.5 + 2.3) = 0.84$$

If we use the balanced tree model, we get the following delays for the two branches of the net *N1*:

$$\text{Net delay to NOR2 input pin} = (0.3/2) * (0.5/2 + 1.3)$$
$$= 0.2325$$

$$\text{Net delay to BUF input pin} = (0.3/2) * (0.5/2 + 1.0)$$
$$= 0.1875$$

Higher Order Interconnect Delay Estimation

As described above, the Elmore delay is the first moment of the impulse response. The AWE (Asymptotic Waveform Evaluation), Arnoldi or other methods match higher order moments of response. By considering the higher order estimates, greater accuracy for computing the interconnect delays is obtained.

Full Chip Delay Calculation

So far, this chapter has described the computation of delays for a cell and the interconnect at the output of a cell. Thus, given the transition at the inputs of the cell, the delays through the cell and the interconnect at the output of the cell can be computed. The transition time at the far-end of the interconnect (destination or sink point) is the input to the next stage and this process is repeated throughout the entire design. The delays for each timing arc in the design are thus calculated.

5.4 Slew Merging

What happens when multiple slews arrive at a common point, such as in the case of a multi-input cell or a multi-driven net? Such a common point is referred to as a *slew merge point*. Which slew is chosen to propagate forward at the slew merge point? Consider the 2-input cell shown in Figure 5-10.

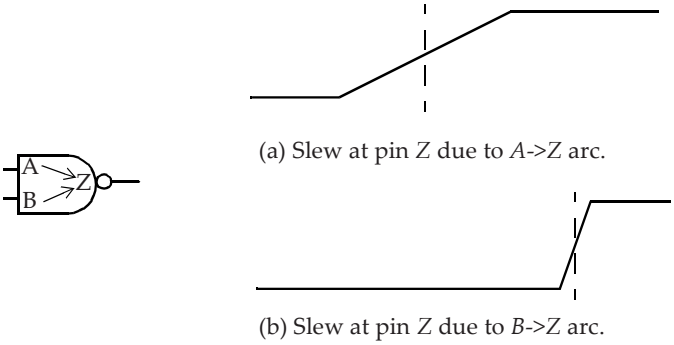


Figure 5-10 *Slews at merge point.*

The slew at pin *Z* due to signal changing on pin *A* arrives early but is slow to rise (slow slew). The slew at pin *Z* due to signal changing on pin *B* arrives late but is fast to rise (fast slew). At the slew merge point, such as pin

Z, which slew should be selected for further propagation? Either of these slew values may be correct depending upon the type of timing analysis (max or min) being performed as described below.

There are two possibilities when doing max path analysis:

i. *Worst slew propagation*: This mode selects the worst slew at the merge point to propagate. This would be the slew in Figure 5-10(a). For a timing path that goes through pins *A*->*Z*, this selection is exact, but is pessimistic for any timing path that goes through pins *B*->*Z*.

ii. *Worst arrival propagation*: This mode selects the worst arrival time at the merge point to propagate. This corresponds to the slew in Figure 5-10(b). The slew chosen in this case is exact for a timing path that goes through pins *B*->*Z* but is optimistic for a timing path that goes through pins *A*->*Z*.

Similarly, there are two possibilities when doing min path analysis:

i. *Best slew propagation*: This mode selects the best slew at the merge point to propagate. This would be the slew in Figure 5-10(b). For a timing path that goes through pins *B*->*Z*, this selection is exact, but the selection is smaller for any timing path that goes through pins *A*->*Z*. For the paths going through *A*->*Z*, the path delays are smaller than the actual values and is thus pessimistic for min path analysis.

ii. *Best arrival propagation*: This mode selects the best arrival time at the merge point to propagate. This corresponds to the slew in Figure 5-10(a). The slew chosen in this case is exact for a timing path that goes through pins *A*->*Z* but the selection is larger than the actual values for a timing path that goes through pins *B*->*Z*. For the paths going through *B*->*Z*, the path delays are larger than the actual values and is thus optimistic for min path analysis.

A designer may perform delay calculation outside of the static timing analysis environment for generating an SDF. In such cases, the delay calculation tools normally use the worst slew propagation. The resulting SDF is adequate for max path analysis but may be optimistic for min path analysis.

Most static timing analysis tools use worst and best slew propagation as their default as it bounds the analysis by being conservative. However, it is possible to use the exact slew propagation when a specific path is being analyzed. The exact slew propagation may require enabling an option in the timing analysis tool. Thus, it is important to understand what slew propagation mode is being used by default in a static timing analysis tool, and understand situations when it may be overly pessimistic.

5.5 Different Slew Thresholds

In general, a library specifies the slew (transition time) threshold values used during characterization of the cells. The question is, what happens when a cell with one set of slew thresholds drives other cells with different set of slew threshold settings? Consider the case shown in Figure 5-11 where a cell characterized with 20-80 slew threshold drives two fanout cells; one with a 10-90 slew threshold and the other with a 30-70 slew threshold and a slew derate of 0.5.

The slew settings for cell *U1* are defined in the cell library as follows:

```
slew_lower_threshold_pct_rise : 20.00
slew_upper_threshold_pct_rise : 80.00

slew_derate_from_library : 1.00
input_threshold_pct_fall : 50.00
output_threshold_pct_fall : 50.00
input_threshold_pct_rise : 50.00
output_threshold_pct_rise : 50.00
```

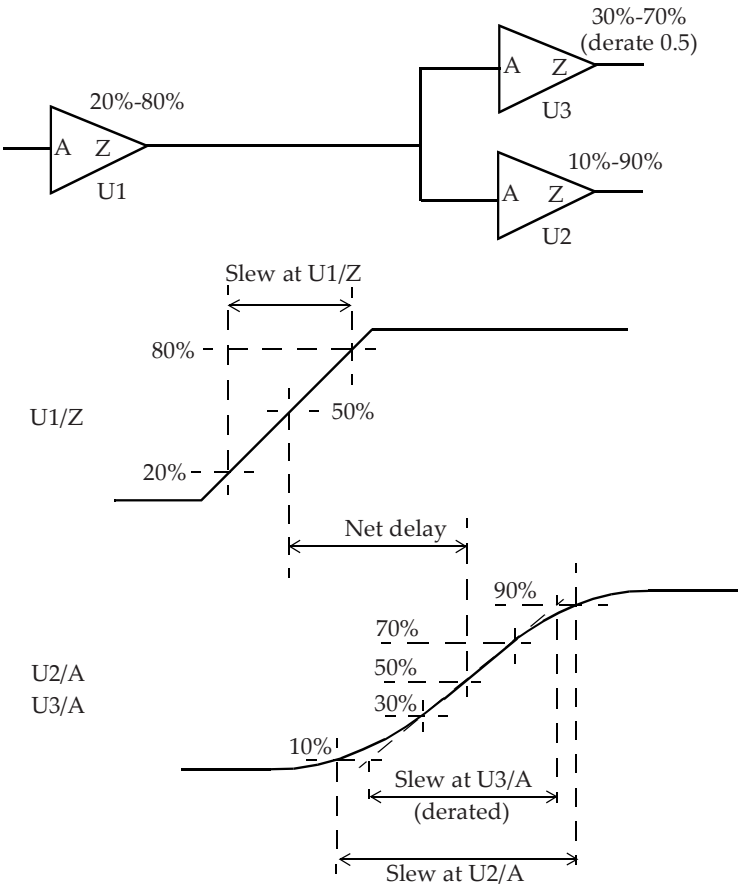


Figure 5-11 *Different slew trip points.*

```
slew_lower_threshold_pct_fall : 20.00
slew_upper_threshold_pct_fall : 80.00
```

The cell *U2* from another library can have the slew settings defined as:

```
slew_lower_threshold_pct_rise : 10.00
slew_upper_threshold_pct_rise : 90.00

slew_derate_from_library : 1.00
slew_lower_threshold_pct_fall : 10.00
slew_upper_threshold_pct_fall : 90.00
```

The cell *U3* from yet another library can have the slew settings defined as:

```
slew_lower_threshold_pct_rise : 30.00
slew_upper_threshold_pct_rise : 70.00

slew_derate_from_library : 0.5
slew_lower_threshold_pct_fall : 30.00
slew_upper_threshold_pct_fall : 70.00
```

Only the slew related settings for *U2* and *U3* are shown above; the delay related settings for input and output thresholds are 50% and not shown above. The delay calculation tools compute the transition times according to the slew thresholds of the cells connecting the net. Figure 5-11 shows how the slew at *U1/Z* corresponds to the switching waveform at this pin. The equivalent Thevenin source at *U1/Z* is utilized to obtain the switching waveforms at the inputs of the fanout cells. Based upon the waveforms at *U2/A* and *U3/A* and their slew thresholds, the delay calculation tools compute the slews at *U2/A* and at *U3/A*. Note that the slew at *U2/A* is based upon 10-90 settings whereas the slew used for *U3/A* is based upon 30-70 settings which is then derated based upon the *slew_derate* of 0.5 as specified in the library. This example illustrates how the slew at the input of the fanout cell is computed based upon the switching waveform and the slew threshold settings of the fanout cell.

During the pre-layout design phase where the interconnect resistance may not be considered, the slews at a net with different thresholds can be com-

puted in the following manner. For example, the relationship between the 10-90 slew and the 20-80 slew is:

```
slew2080 / (0.8 - 0.2) = slew1090 /(0.9 - 0.1)
```

Thus, a slew of 500ps with 10-90 measurement points corresponds to a slew of (500ps * 0.6) / 0.8 = 375ps with 20-80 measurement points. Similarly, a slew of 600ps with 20-80 measurement points corresponds to a slew of (600ps * 0.8) / 0.6 = 800ps with 10-90 measurement points.

5.6 Different Voltage Domains

A typical design may use different power supply levels for different portions of the chip. In such cases, level shifting cells are used at the interface between different power supply domains. A level shifting cell accepts input at one supply domain and provides output at the other supply domain. As an example, a standard cell input can be at 1.2V and its output can be at a reduced power supply, which may be 0.9V. Figure 5-12 shows an example.

Notice that the delay is calculated from the 50% threshold points. These points are at different voltages for different pins of the interface cells.

5.7 Path Delay Calculation

Once all the delays for each timing arc are available, the timing through the cells in the design can be represented as a timing graph. The timing through the combinational cells can be represented as timing arcs from inputs to outputs. Similarly, the interconnect is represented with corresponding arcs from the source to each destination (or sink) point represented as a separate timing arc. Once the entire design is annotated by corresponding arcs, computing the path delay involves adding up all the net and cell timing arcs along the path.

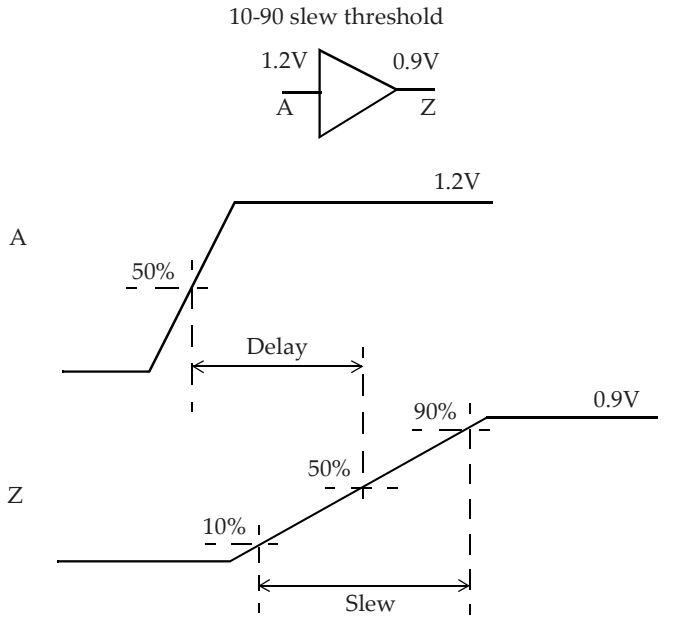


Figure 5-12 *Cell with different input and output voltages.*

5.7.1 Combinational Path Delay

Consider the three inverters in series as shown in Figure 5-13. While considering paths from net *N0* to net *N3*, we consider both rising edge and falling edge paths. Assume that there is a rising edge at net *N0*.

The transition time (or slew) at the input of the first inverter may be specified; in the absence of such a specification, a transition time of 0 (corresponding to an ideal step) is assumed. The transition time at the input *UINVa/A* is determined by using the interconnect delay model as specified in the previous section. The same delay model is also used in determining the delay, *Tn0*, for net *N0*.

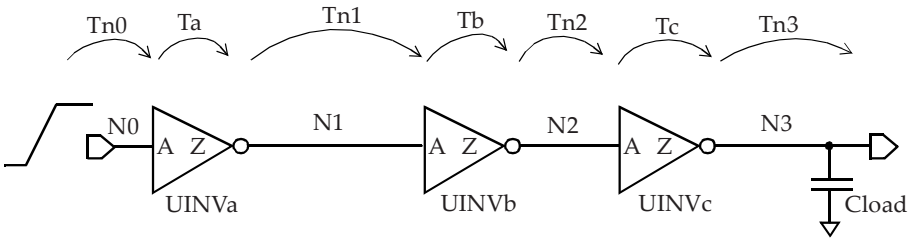


Figure 5-13 *Timing a combinational path.*

The effective capacitance at the output *UINVa/Z* is obtained based upon the RC load at the output of *UINVa*. The transition time at input *UINVa/A* and the equivalent effective load at output *UINVa/Z* is then used to obtain the cell output fall delay.

The equivalent Thevenin source model at pin *UINVa/Z* is used to determine the transition time at pin *UINVb/A* by using the interconnect model. The interconnect model is also used to determine the delay, *Tn1*, on the net *N1*.

Once the transition time at input *UINVb/A* is known, the process for calculating the delay through *UINVb* is similarly utilized. The RC interconnect at *UINVb/Z*, and the pin capacitance of pin *UINVc/A* are used to determine the effective load at *N2*. The transition time at *UINVb/A* is used to determine the rise delay through the inverter *UINVb*, and so on.

The load at the last stage is determined by any explicit load specification provided, or in the absence of which only the wire load of net *N3* is used.

The above analysis assumed a rising edge at net *N0*. Similar analysis can be carried out for the case of a falling edge on net *N0*. Thus, in this simple example, there are two timing paths with the following delays:

$$T_{fall} = Tn0_{rise} + Ta_{fall} + Tn1_{fall} + Tb_{rise} + Tn2_{rise} + Tc_{fall} + Tn3_{fall}$$

$$T_{rise} = Tn0_{fall} + Ta_{rise} + Tn1_{rise} + Tb_{fall} + Tn2_{fall} + Tc_{rise} + Tn3_{rise}$$

In general, the rise and fall delays through the interconnect can be different because of different Thevenin source models at the output of the driving cell.

5.7.2 Path to a Flip-flop

Input to Flip-flop Path

Consider the timing of the path from input *SDT* to flip-flop *UFF1* as shown in Figure 5-14.

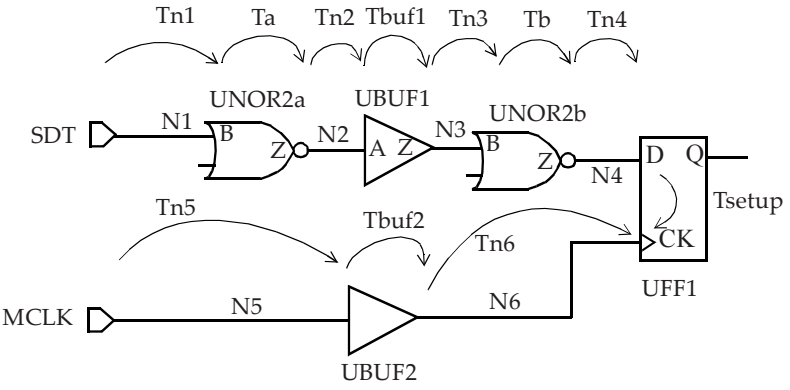


Figure 5-14 *Path to a flip-flop.*

We need to consider both rising edge and falling edge paths. For the case of a rising edge on input *SDT,* the data path delay is:

$$Tn1_{rise} + Ta_{fall} + Tn2_{fall} + Tbuf1_{fall} + Tn3_{fall} + Tb_{rise} + Tn4_{rise}$$

Similarly, for a falling edge on input *SDT*, the data path delay is:

$$Tn1_{fall} + Ta_{rise} + Tn2_{rise} + Tbuf1_{rise} + Tn3_{rise} + Tb_{fall} + Tn4_{fall}$$

The capture clock path delay for a rising edge on input *MCLK* is:

$$Tn5_{rise} + Tbuf2_{rise} + Tn6_{rise}$$

Flip-flop to Flip-flop Path

An example of a data path between two flip-flops and corresponding clock paths is shown in Figure 5-15.

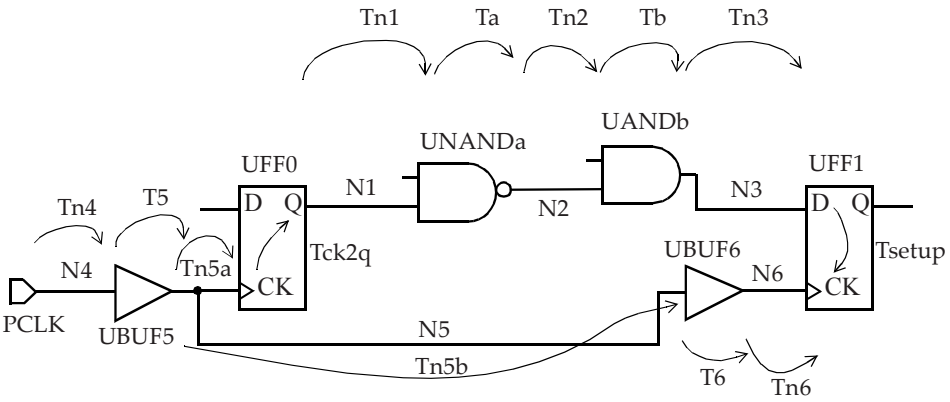


Figure 5-15 *Flip-flop to flip-flop.*

The data path delay for a rising edge on *UFF0/Q* is:

$$Tck2q_{rise} + Tn1_{rise} + Ta_{fall} + Tn2_{fall} + Tb_{fall} + Tn3_{fall}$$

The launch clock path delay for a rising edge on input *PCLK* is:

$$Tn4_{rise} + T5_{rise} + Tn5a_{rise}$$

The capture clock path delay for a rising edge on input *PCLK* is:

$$Tn4_{rise} + T5_{rise} + Tn5b_{rise} + T6_{rise} + Tn6_{rise}$$

Note that unateness of the cell needs to be considered as the edge direction may change as it goes through a cell.

5.7.3 Multiple Paths

Between any two points, there can be many paths. The longest path is the one that takes the longest time; this is also called the worst path, a late path or a max path. The shortest path is the one that takes the shortest time; this is also called the best path, an early path or a min path.

See the logic and the delays through the timing arcs in Figure 5-16. The longest path between the two flip-flops is through the cells *UBUF1*, *UNOR2*, and *UNAND3*. The shortest path between the two flip-flops is through the cell *UNAND3*.

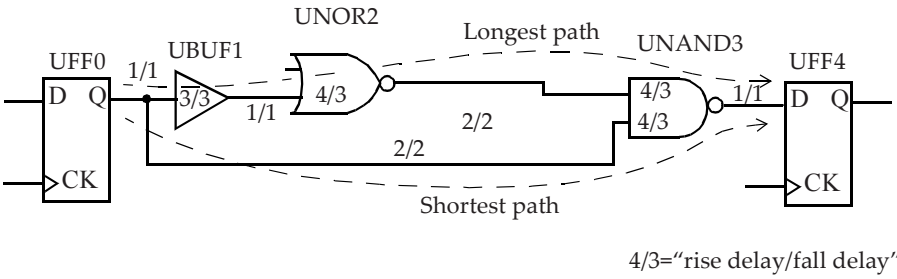


Figure 5-16 *Longest and shortest paths.*

5.8 Slack Calculation

Slack is the difference between the required time and the time that a signal arrives. In Figure 5-17, the data is required to be stable at time 7ns for the setup requirement to be met. However data becomes stable at 1ns. Thus, the slack is 6ns (= 7ns - 1ns).

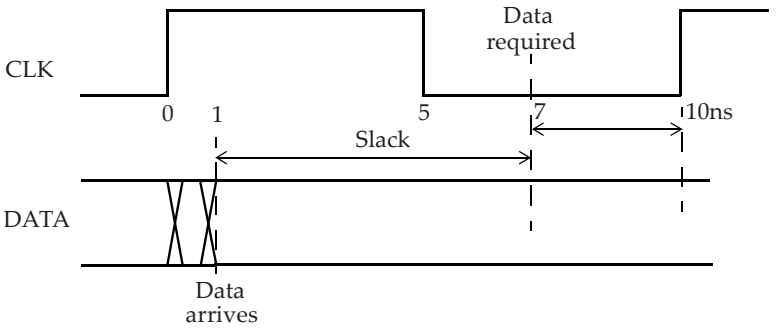


Figure 5-17 *Slack.*

Assuming that the data required time is obtained from the setup time of a capture flip-flop,

```
Slack = Required_time - Arrival_time
Required_time = Tperiod - Tsetup(capture_flip_flop)
              = 10 - 3 = 7ns
Arrival_time = 1ns
Slack = 7 - 1 = 6ns
```

Similarly if there is a skew requirement of 100ps between two signals and the measured skew is 60ps, the slack in skew is 40ps (= 100ps - 60ps).

❑

CHAPTER

6

Crosstalk and Noise

This chapter describes the signal integrity aspects of an ASIC in nanometer technologies. In deep submicron technologies, crosstalk plays an important role in the signal integrity of the design. The crosstalk noise refers to unintentional coupling of activity between two or more signals. Relevant noise and crosstalk analysis techniques, namely glitch analysis and crosstalk analysis, allow these effects to be included during static timing analysis and are described in this chapter. These techniques can be used to make the ASIC behave robustly.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_6, © Springer Science + Business Media, LLC 2009

6.1 Overview

Noise refers to undesired or unintentional effects affecting the proper operation of the chip. In nanometer technologies, the noise can impact in terms of functionality or in terms of timing of the devices.

Why noise and signal integrity?

There are several reasons why the noise plays an important role in the deep submicron technologies:

- *Increasing number of metal layers*: For example, a 0.25µm or 0.3µm process has four or five metal layers and it increases to ten or higher metal layers in the 65nm and 45nm process geometries. Figure 4-1 depicts the multiple layers of the metal interconnect.
- *Vertically dominant metal aspect ratio*: This means that the wires are thin and tall unlike the wide and thin in the earlier process geometries. Thus, a greater proportion of the capacitance is comprised of sidewall coupling capacitance which maps into wire to wire capacitance between neighboring wires.
- *Higher routing density due to finer geometry*: Thus, more metal wires are packed in close physical proximity.
- *Larger number of interacting devices and interconnects*: Thus, greater number of active standard cells and signal traces are packed in the same silicon area causing a lot more interactions.
- *Faster waveforms due to higher frequencies*: Fast edge rates cause more current spikes as well as greater coupling impact on the neighboring traces and cells.
- *Lower supply voltage*: The supply voltage reduction leaves little margin for noise.

In this chapter, we study the effect of crosstalk noise in particular. The crosstalk noise refers to unintentional coupling of activity between two or more signals. The crosstalk noise is caused by the capacitive coupling between neighboring signals on the die. This results in switching activity on a net to cause unintentional effects on the coupled signals. The affected sig-

nal is called the **victim**, and the affecting signals are termed as **aggressors**. Note that two coupled nets can affect each other, and often a net can be a victim as well as an aggressor.

Figure 6-1 shows an example of a few signal traces coupled together. The distributed RC extraction of the coupled interconnect is depicted along with several drivers and fanout cells. In this example, nets *N1* and *N2* have *Cc1* + *Cc4* as coupling capacitance between them, whereas *Cc2* + *Cc5* is the coupling capacitance between nets *N2* and *N3*.

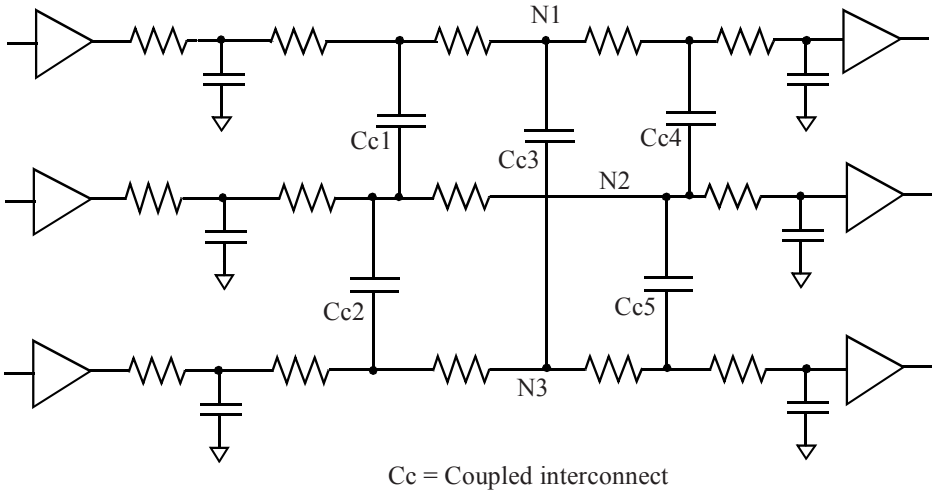


Figure 6-1 *Example of coupled interconnect.*

Broadly, there are two types of noise effects caused by crosstalk - **glitch**[1], which refers to noise caused on a steady victim signal due to coupling of switching activity of neighboring aggressors, and change in **timing** (crosstalk delta delay), caused by coupling of switching activity of the victim with the switching activity of the aggressors. These two types of crosstalk noise are described in the next two sections.

1. Some analysis tools refer to glitch as *noise*. Similarly, some tools use *crosstalk* to refer to crosstalk effect on delay.

6.2 Crosstalk Glitch Analysis

6.2.1 Basics

A steady signal net can have a glitch (positive or negative) due to charge transferred by the switching aggressors through the coupling capacitances. A positive glitch induced by crosstalk from a rising aggressor net is illustrated in Figure 6-2. The coupling capacitance between the two nets is depicted as one lumped capacitance *Cc* instead of distributed coupling, this is to simplify the explanation below without any loss of generality. In typical representations of the extracted netlist, the coupling capacitance may be distributed across multiple segments as seen previously in Section 6.1.

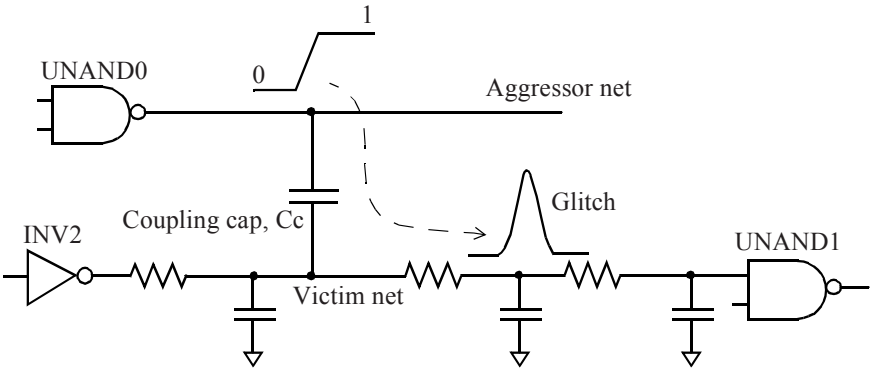


Figure 6-2 *Glitch due to an aggressor.*

In this example, the *NAND2* cell *UNAND0* switches and charges its output net (labeled *Aggressor*). Some of the charge is also transferred to the victim net through the coupling capacitance *Cc* and results in the positive glitch. The amount of charge transferred is directly related to the coupling capacitance, *Cc,* between the aggressor and the victim net. The charge transferred on the grounded capacitances of the victim net causes the glitch at that net. The steady value on the victim net (in this case, 0 or low) is restored because the transferred charge is dissipated through the pull-down stage of the driving cell *INV2*.

The magnitude of the glitch caused is dependent upon a variety of factors. Some of these factors are:

i. *Coupling capacitance between the aggressor net and victim*: The greater the coupling capacitance, the larger the magnitude of the glitch.

ii. *Slew of the aggressor net*: The faster the slew at the aggressor net, the larger the magnitude of glitch. In general, faster slew is because of higher output drive strength for the cell driving the aggressor net.

iii. *Victim net grounded capacitance*: The smaller the grounded capacitance on the victim net, the larger the magnitude of the glitch.

iv. *Victim net driving strength*: The smaller the output drive strength of the cell driving the victim net, the larger the magnitude of the glitch.

Overall, while the steady value on the victim net gets restored, the glitch can affect the functionality of the circuit for the reasons stated below.

- The glitch magnitude may be large enough to be seen as a different logic value by the fanout cells (e.g. a victim at logic-0 may appear as logic-1 for the fanout cells). This is especially critical for the sequential cells (flip-flops, latches) or memories, where a glitch on the clock or asynchronous set/reset can be catastrophic to the functionality of the design. Similarly, a glitch on the data signal at the latch input can cause incorrect data to be latched which can also be catastrophic if the glitch occurs when the data is being clocked in.
- Even if the victim net does not drive a sequential cell, a wide glitch may be propagated through the fanouts of the victim net and reach a sequential cell input with catastrophic consequences for the design.

6.2.2 Types of Glitches

There are different types of glitches.

Rise and Fall Glitches

The discussion in the previous subsection illustrates a positive or **rise glitch** on a victim net which is steady low. An analogous case is of a negative glitch on a steady high signal. A falling aggressor net induces a **fall glitch** on a steady high signal.

Overshoot and Undershoot Glitches

What happens when a rising aggressor couples to a victim net which is steady high? There is still a glitch which takes the victim net voltage above its steady high value. Such a glitch is called an **overshoot glitch**. Similarly, a falling aggressor when coupled to a steady low victim net causes an **undershoot glitch** on the victim net.

All four cases of glitches induced by crosstalk are illustrated in Figure 6-3.

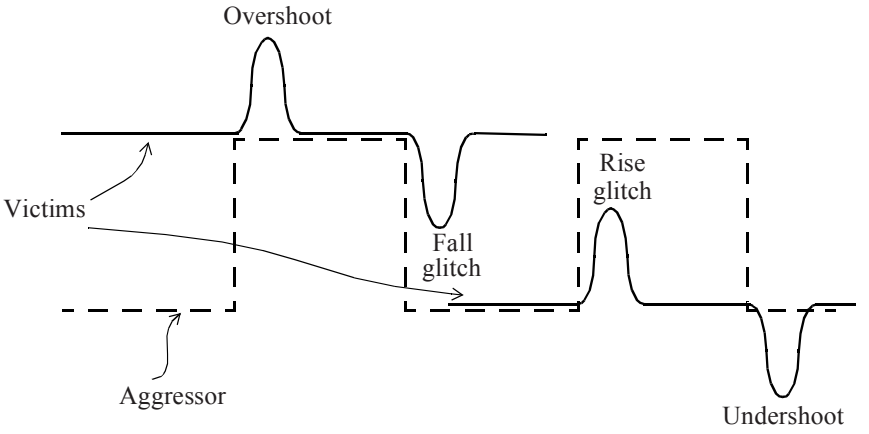


Figure 6-3 *Types of glitches.*

As described in the previous subsection, the glitch is governed by the coupling capacitance, aggressor slew and the drive strength of the victim net. The glitch computation is based upon the amount of current injected by the switching aggressor, the RC interconnect for the victim net, and the output impedance of the cell driving the victim net. The detailed glitch calculation is based upon the library models; the related noise models for the calculation are part of the standard cell library models described in Chapter 3. The output *dc_current* models in Section 3.7 relate to the output impedance of the cell.

6.2.3 Glitch Thresholds and Propagation

How can it be determined whether a glitch at a net can be propagated through the fanout cells? As discussed in an earlier subsection, a glitch caused by coupling from a switching aggressor can propagate through the fanout cell depending upon the fanout cell and glitch attributes such as glitch height and glitch width. This analysis can be based upon DC or AC noise thresholds. The DC noise analysis only examines the glitch magnitude and is conservative whereas the AC noise analysis examines other attributes such as glitch width and fanout cell output load. Various threshold metrics used in the DC and AC analyses of the glitches are described below.

DC Thresholds

The *DC noise margin* is a check used for glitch magnitude and refers to the DC noise limits on the input of a cell while ensuring proper logic functionality. For example, the output of an inverter cell may be high (that is, stay above the minimum value of V*OH*) as long as the input stays below the max value of *VIL* for the cell. Similarly, the output of the inverter cell may be low (that is, stay below *VOL* maximum value) as long as the input stays above the *VIH* minimum value. These limits are obtained based upon the DC transfer characteristics[1] of the cell and may be populated in the cell library.

1. See [DAL08] in Bibliography.

VOH is the range of output voltage that is considered as a logic-one or high. *VIL* is the range of input voltage that is considered a logic-zero or low. *VIH* is the range of input voltage that is considered as a logic-one. *VOL* is the range of output voltage that is considered as a logic-zero. An example of the input-output DC transfer characteristics of an inverter cell is given in Figure 6-4.

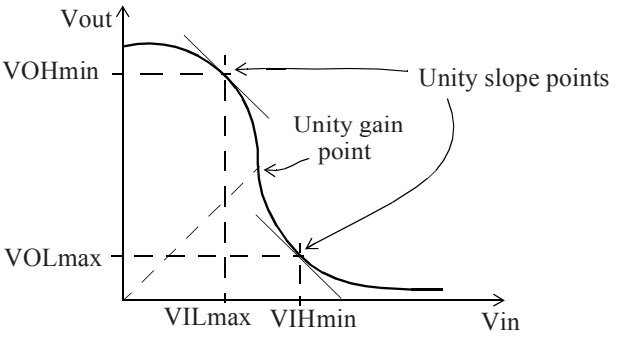


Figure 6-4 *DC transfer characteristics of an inverter cell.*

The *VILmax* and *VIHmin* limits are also referred to as DC margin limits. The DC margins based upon *VIH* and *VIL* are steady state noise limits. These can thus be used as a filter for determining whether a glitch will propagate through the fanout cell. The DC noise margin limits are applicable for every input pin of a cell. In general, the DC margin limits are separate for *rise_glitch* (input low) and *fall_glitch* (input high). Models for DC margin can be specified as part of the cell library description. A glitch below the DC margin limit (for example, a rise glitch below the *VILmax* of the fanout pins) cannot be propagated through the fanout irrespective of the width of the glitch. Thus, a conservative glitch analysis checks that the peak voltage level (for all glitches) meets the *VIL* and the *VIH* levels of the fanout cells. As long as all nets meet the *VIL* and *VIH* levels for the fanout cells in spite of any glitches, it can be concluded that the glitches have no impact on the functionality of the design (since the glitches cannot cause the output to change).

Figure 6-5 shows an example of DC margin limits. The DC noise margin can also be fixed to the same limit for all nets in the design. One can set the largest tolerable noise (or glitch) magnitude, above which noise can be propagated through the cell to the output pin. Typically this check ensures that the glitch level is less than *VILmax* and greater than *VIHmin*. The height is often expressed as a percent of the power supply. Thus, if the DC noise margin is set to 30%, that indicates that any glitch height greater than 30% of the voltage swing is identified as a potential glitch that can propagate through the cell and potentially impact the functionality of the design.

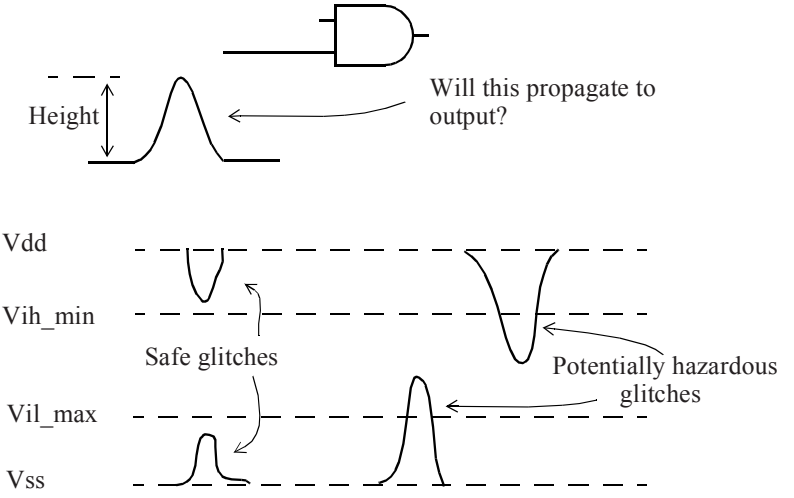


Figure 6-5 *Glitch check based upon DC noise margin.*

Not all glitches with magnitude larger than the DC noise margin can change the output of a cell. The width of the glitch is also an important consideration in determining whether the glitch will propagate to the output or not. A narrow glitch at a cell input will normally not cause any impact at the output of a cell. However, DC noise margin uses only a constant worst-case value irrespective of the signal noise width. See Figure 6-6. This provides a noise rejection level that is a very conservative estimate of the noise tolerance of a cell.

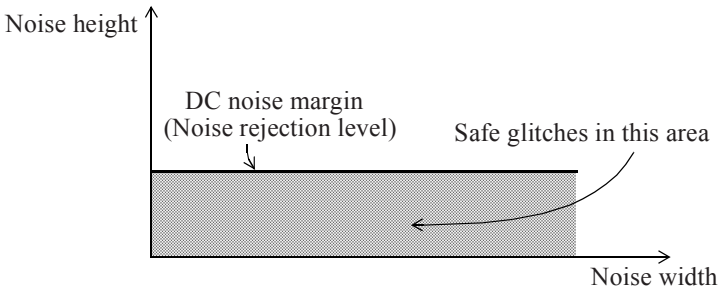


Figure 6-6 *DC noise rejection level.*

AC Thresholds

As described in the subsection above, the DC margin limits for glitch analysis are conservative since these analyze the design under the worst-case conditions. The DC margin limits verify that even if the glitch is arbitrarily wide, it will not affect the proper operation of the design.

In most cases, a design may not pass the conservative DC noise analysis limits. Therefore it becomes imperative to verify the impact of glitches with respect to the glitch width and the output load of the cell. In general, if the glitch is narrow or if the fanout cell has a large output capacitance, the glitch would have no effect on the proper functional operation. Both the effect of glitch width and the output capacitance can be explained in terms of the inertia of the fanout cell. In general, a single stage cell will stop any input glitch which is much narrower than the delay through the cell. This is because with a narrow glitch, the glitch is over before the fanout cell can respond to it. Thus, a very narrow glitch does not have any effect on the cell. Since the output load increases the delay through the cell, increasing the output load has the effect of minimizing the impact of glitch at the input - though it has the adverse effect of increasing the cell delay.

The AC noise rejection is illustrated in Figure 6-7 (for a fixed output capacitance). The dark shaded region represents *good* or *acceptable* glitches since these are either too narrow or too short, or both, and thus have no effect on the functional behavior of the cell. The lightly shaded region represents *bad*

or *unacceptable* glitches since these are too wide or too tall, or both, and thus such a glitch at the cell input affects the output of the cell. In the limiting case of very wide glitches, the glitch threshold corresponds to the DC noise margin as shown in Figure 6-7.

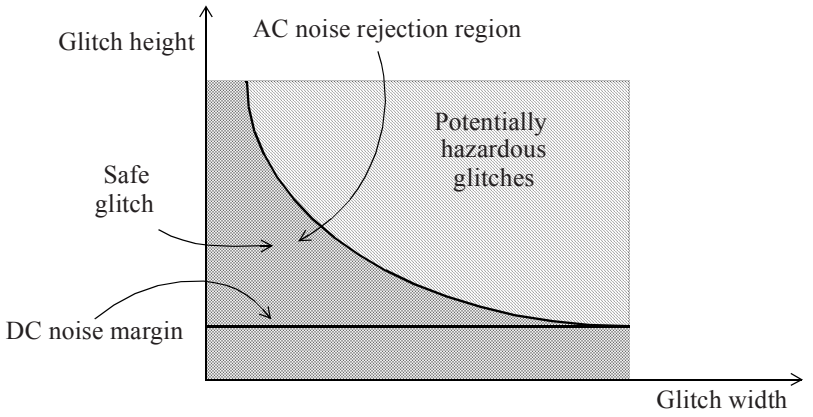


Figure 6-7 *AC noise rejection region.*

For a given cell, increasing the output load increases the noise margin since it increases the inertial delay and the width of the glitch that can pass through the cell. This phenomenon is illustrated through an example below. Figure 6-8(a) shows an unloaded inverter cell with a positive glitch at its input. The input glitch is taller than the DC margin of the cell and causes a glitch at the inverter output. Figure 6-8(b) shows the same inverter cell with some load at its output. The same input glitch at its input results in a much smaller glitch at the output. If the output load of the inverter cell is even higher, as in Figure 6-8(c), the output of the inverter cell does not have any glitch. Thus, increasing the load at the output makes the cell more immune to noise propagating from the input to the output.

As described above, the glitches below the AC threshold (the AC noise rejection region in Figure 6-7) can be ignored or the fanout cell can be considered to be immune from such a glitch. The AC threshold (or noise immunity) region depends upon the output load and the glitch width. As

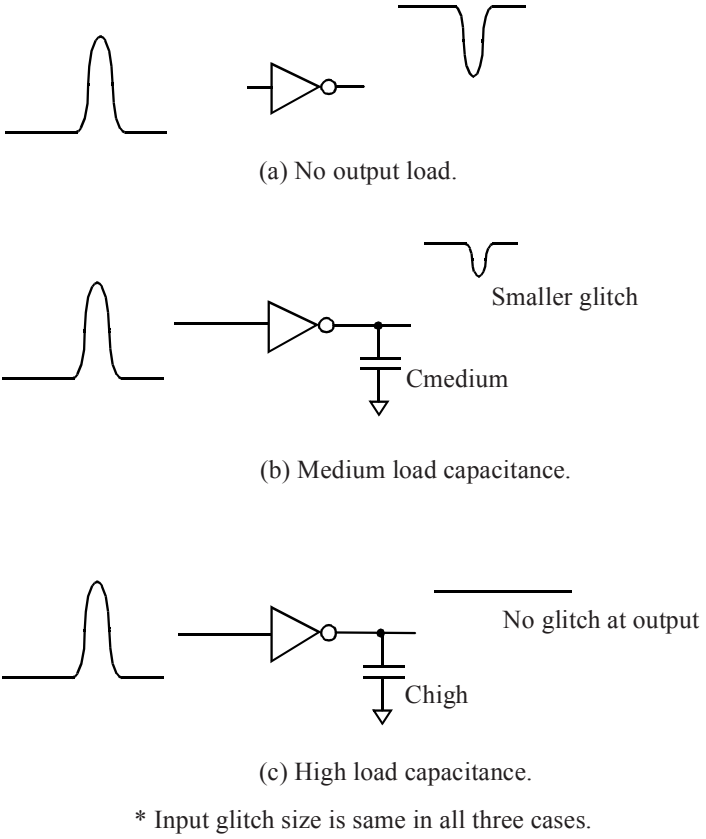


Figure 6-8 *Output load determines size of propagated glitch.*

described in Chapter 3, noise immunity models include the effect of AC noise rejection described above. The *propagated_noise* models described in Section 3.7 capture the effect of AC noise threshold in addition to modeling the propagation through the cell.

What happens if the glitches are larger than the AC threshold? In such a case where the glitch magnitude exceeds the AC threshold, the glitch at the cell input produces another glitch at the output of the cell. The output glitch height and width is a function of input glitch width and height as well as the output load. This information is characterized in the cell library which contains detailed tables or functions for the output glitch magnitude and width as a function of the input pin glitch magnitude, glitch width and the load at the output pin. The glitch propagation is governed by *propagated_noise* models which are included in the library cell description. The *propagated_noise (low* and *high)* models are described in detail in Chapter 3.

Based upon the above, the glitch is computed at the output of the fanout cell and the same checks (and glitch propagation to the fanout) are followed at the fanout net and so on.

While we have used the generic term *glitch* in the discussion above, it should be noted that this applies separately to *rise glitch* (modeled by *propagated_noise_high;* or *noise_immunity_high* in earlier models), *fall glitch* (modeled by *propagated_noise_low;* or *noise_immunity_low* in earlier models), *overshoot glitch* (modeled by *noise_immunity_above_high*) and *undershoot glitch* (modeled by *noise_immunity_below_low*) as described in the previous sections.

To summarize, different inputs of a cell have different limits on the glitch threshold which is a function of glitch width and output capacitance. These limits are separate for input high (low transition glitch) and for input low (high transition glitch). The noise analysis examines the peak as well as the width of the glitch and analyzes whether it can be neglected or whether it can propagate to fanouts.

6.2.4 Noise Accumulation with Multiple Aggressors

Figure 6-9 depicts the coupling due to a single aggressor net switching and introducing a crosstalk glitch on the victim net. In general, a victim net may be capacitive-coupled to many nets. When multiple nets switch concurrently, the crosstalk coupling noise effect on the victim is compounded due to multiple aggressors.

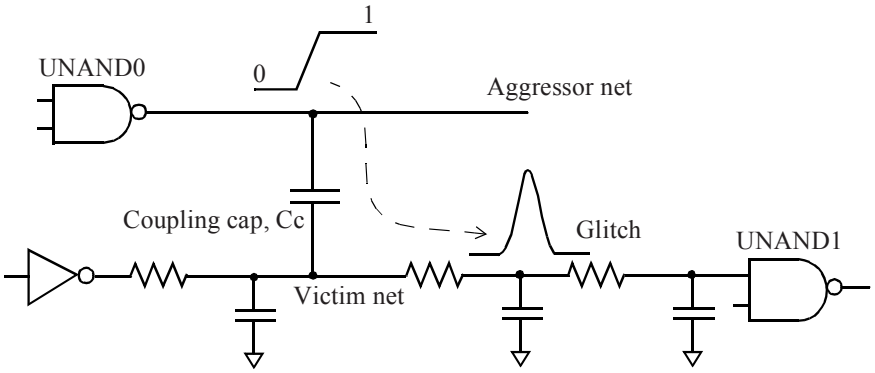


Figure 6-9 *Glitch from single aggressor.*

Most analyses for coupling due to multiple aggressors add the glitch effect due to each aggressor and compute the cumulative effect on the victim. This may appear conservative, however it does indicate the worst-case glitch on the victim. An alternate approach is the use of RMS (*root-mean-squared*) approach. When using the RMS option, the magnitude of the glitch on the victim is computed by taking the root-mean-square of the glitches caused by individual aggressors.

6.2.5 Aggressor Timing Correlation

For crosstalk glitch due to multiple aggressors, the analysis must include the timing correlation of the aggressor nets and determine whether the multiple aggressors can switch concurrently. The STA obtains this information from the timing windows of the aggressor nets. During timing

analysis, the *earliest* and the *latest* switching times of the nets are obtained. These times represent the *timing windows* during which a net may switch within a clock cycle. The switching windows (rising and falling) provide the necessary information on whether the aggressor nets can switch together.

Based upon whether the multiple aggressors can switch concurrently, the glitches due to individual aggressors are combined for the victim net. As a first step, the glitch analysis computes the four types of glitches (rise, fall, undershoot, and overshoot) separately for each potential aggressor. The next step combines the glitch contributions from the various individual aggressors. The multiple aggressors can combine separately for each type of glitch. For example, consider a victim net *V* coupled to aggressor nets *A1*, *A2*, *A3* and *A4*. During analysis, it is possible that *A1*, *A2*, and *A4* contribute to rising and overshoot glitches, whereas only *A2* and *A3* contribute to undershoot and falling glitches.

Consider another example where four aggressor nets can cause a rising glitch when the aggressor nets transition. Figure 6-10 shows the timing windows and the glitch magnitude caused by each aggressor net. Based upon the timing windows, the glitch analysis determines the worst possible combination of aggressor switching which results in the largest glitch. In this example, the switching window region is divided in four bins - each bin shows the possible aggressors switching. The glitch contribution from each aggressor is also depicted in Figure 6-10. Bin 1 has *A1* and *A2* switching which can result in a glitch magnitude of 0.21 (= 0.11 + 0.10). Bin 2 has *A1*, *A2*, and *A3* switching which can result in a glitch magnitude of 0.30 (= 0.11 + 0.10 + 0.09). Bin 3 has *A1* and *A3* switching which can result in a glitch magnitude of 0.20 (= 0.11 + 0.09). Bin 4 has *A3* and *A4* switching which can result in a glitch magnitude of 0.32 (= 0.09 + 0.23).

Thus, bin 4 has the worst possible glitch magnitude of 0.32. Note that an analysis without using timing windows will predict a combined glitch magnitude of 0.53 (= 0.11 + 0.10 + 0.09 + 0.23) which can be overly pessimistic.

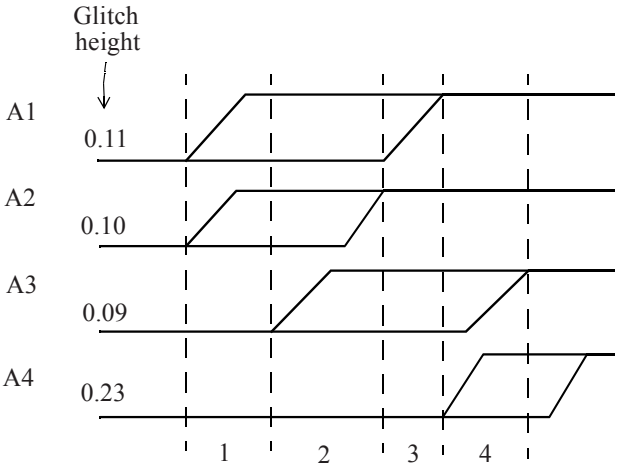


Figure 6-10 *Switching windows and glitch magnitudes from multiple aggressors.*

6.2.6 Aggressor Functional Correlation

For multiple aggressors, the use of timing windows reduces the pessimism in the analysis by considering the switching window during which a net can possibly switch. In addition, another factor to be considered is the *functional correlation* between various signals. For example, the scan[1] control signals only switch during the scan mode and are steady during functional or mission mode of the design. Thus, the scan control signals cannot cause a glitch on other signals during the functional mode. The scan control signals can only be aggressors during the scan mode. In some cases, the test and functional clocks are mutually exclusive such that the test clock is active only during testing when the functional clocks are turned off. In these designs, the logic controlled by test clocks and the logic controlled by functional clocks create two disjoint sets of aggressors. For such cases, the ag-

1. DFT test mode.

gressors controlled by test clocks cannot be combined with the other aggressors controlled by functional clocks for worst-case noise computation. Another example of functional correlation is two aggressors which are the complement (logical inverse) of each other. For such cases, both signal and its complement cannot be switching in the same direction for crosstalk noise computation.

Figure 6-11 shows an example of net *N1* having coupling with three other nets *N2*, *N3* and *N4*. In functional correlation, the functionality of the nets needs to be considered. Assume net *N4* is a constant (for example, a mode setting net) and thus cannot be an aggressor on net *N1*, in spite of its coupling. Assume *N2* is a net that is part of debug bus, but is in steady state in functional mode. Thus, net *N2* cannot be an aggressor for net *N1*. Assuming net *N3* carries functional data, only net *N3* can be considered as a potential aggressor for net *N1*.

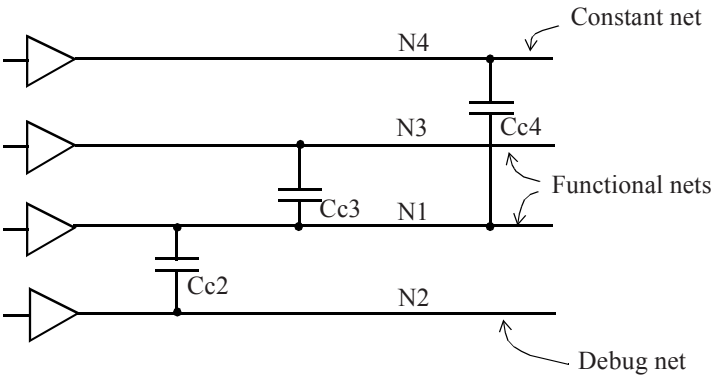


Figure 6-11 *Three couplings but only one aggressor.*

6.3 Crosstalk Delay Analysis

6.3.1 Basics

The capacitance extraction for a typical net in a nanometer design consists of contributions from many neighboring conductors. Some of these are grounded capacitances while many others are from traces which are part of other signal nets. The grounded as well as inter-signal capacitances are illustrated in Figure 6-1. All of these capacitances are considered as part of the total net capacitance during the basic delay calculation (without considering any crosstalk). When the neighboring nets are steady (or not switching), the inter-signal capacitances can be treated as grounded. When a neighboring net is switching, the charging current through the coupling capacitance impacts the timing of the net. The equivalent capacitance seen from a net can be larger or smaller based upon the direction of the aggressor net switching. This is explained in a simple example below.

Figure 6-12 shows net *N1* which has a coupling capacitance *Cc* to a neighboring net (labeled *Aggressor*) and a capacitance *Cg* to ground. This example assumes that the net *N1* has a rising transition at the output and considers different scenarios depending on whether or not the aggressor net is switching at the same time.

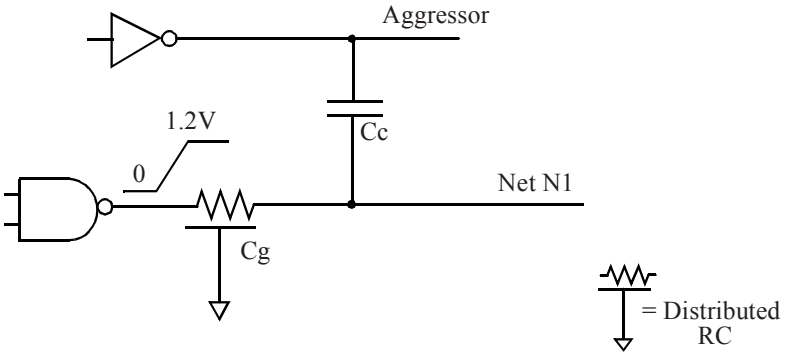


Figure 6-12 *Crosstalk impact example.*

The capacitive charge required from the driving cell in various scenarios can be different as described next.

i. **Aggressor net steady**. In this scenario, the driving cell for the net *N1* provides the charge for *Cg* and *Cc* to be charged to *Vdd*. The total charge provided by the driving cell of this net is thus *(Cg + Cc) * Vdd*. The base delay calculation obtains delays for this scenario where no crosstalk is considered from the aggressor nets. Table 6-13 shows the charge on *Cg* and *Cc* before and after the switching of *N1* for this scenario.

Capacitance		*Before rising transition at net N1*	*After rising transition at net N1*
`Grounded Cap, Cg`		`V(Cg) = 0`	`V(Cg) = Vdd`
`Coupling Cap, Cc`	`Aggressor net steady LOW`	`V(Cc) = 0`	`V(Cc) = Vdd`
	`Aggressor net steady HIGH`	`V(Cc) = - Vdd`	`V(Cc) = 0`

Table 6-13 *Base delay calculation - no crosstalk.*

ii. **Aggressor switching in same direction**. In this scenario, the driving cell is aided by the aggressor switching in the same direction. If the aggressor transitions at the same time with the same slew (identical transition time), the total charge provided by the driving cell is only (*Cg * Vdd)*. If the slew of the aggressor net is faster than that of *N1*, the actual charge required can be even smaller than (*Cg * Vdd)* since the aggressor net can provide charging current for *Cg* also. Thus, the required charge from the driving cell with the aggressor switching in same direction is smaller than the corresponding charge for the steady aggressor described in Table 6-13. Therefore, the aggressor switching in the same direction results in a smaller delay for the switching net

N1; the reduction in delay is labeled as **negative crosstalk delay**. See Table 6-14. This scenario is normally considered for min path analysis.

Capacitance	*Before rising transition at net N1 and aggressor net*	*After rising transition at net N1 and aggressor net*
`Grounded Cap, Cg`	`V(Cg) = 0`	`V(Cg) = Vdd`
`Coupling Cap, Cc`	`V(Cc) = 0`	`V(Cc) = 0`

Table 6-14 *Aggressor switching in same direction - negative crosstalk.*

iii. **Aggressor switching in opposite direction**. In this scenario, the coupling capacitance is charged from *-Vdd* to *Vdd*. Thus, the charge on coupling capacitance changes by *(2 * Cc * Vdd)* before and after the transitions. This additional charge is provided by both the driving cell of net *N1* as well as the aggressor net. This scenario results in a larger delay for the switching net *N1;* the increase in delay is labeled as **positive crosstalk delay**. See Table 6-15. This scenario is normally considered for max path analysis.

Capacitance	*Before transition at net N1 and aggressor net (net N1 is low; aggressor net is high)*	*After transition (net N1 is high; and aggressor net is low)*
`Grounded Cap, Cg`	`V(Cg) = 0`	`V(Cg) = Vdd`
`Coupling Cap, Cc`	`V(Cc) = - Vdd`	`V(Cc) = Vdd`

Table 6-15 *Aggressor switching in opposite direction - positive crosstalk.*

The above example illustrates the charging of *Cc* in various cases and how it can impact the delay of the switching net (labelled as *N1*). The example considers only a rising transition at net *N1*, however similar analysis holds for falling transitions also.

6.3.2 Positive and Negative Crosstalk

The base delay calculation (without any crosstalk) assumes that the driving cell provides all the necessary charge for rail-to-rail transition of the total capacitance of a net, *Ctotal* (= *Cground* + *Cc*). As described in the previous subsection, the charge required for the coupling capacitance *Cc* is larger when the coupled (aggressor) net and victim net are switching in the opposite directions. The aggressor switching in the opposite direction increases the amount of charge required from the driving cell of the victim net and increases the delays for the driving cell and the interconnect for the victim net.

Similarly, when the coupled (aggressor) net and the victim net are switching in the same direction, the charge on *Cc* remains the same before and after the transitions of the victim and aggressor. This reduces the charge required from the driving cell of the victim net. The delays for the driving cell and the interconnect for the victim net are reduced.

As described above, concurrent switching of victim and aggressor affects the timing of the victim transition. Depending upon the switching direction of the aggressor, the crosstalk delay effect can be positive (slow down the victim transition) or negative (speed up the victim transition).

An example of positive crosstalk delay effect is shown in Figure 6-16. The aggressor net is rising at the same time when the victim net has a falling transition. The aggressor net switching in opposite direction increases the delay for the victim net. The positive crosstalk impacts the driving cell as well as the interconnect - the delay for both of these gets increased.

The case of negative crosstalk delay is illustrated in Figure 6-17. The aggressor net is rising at the same time as the victim net. The aggressor net switching in the same direction as the victim reduces the delay of the vic-

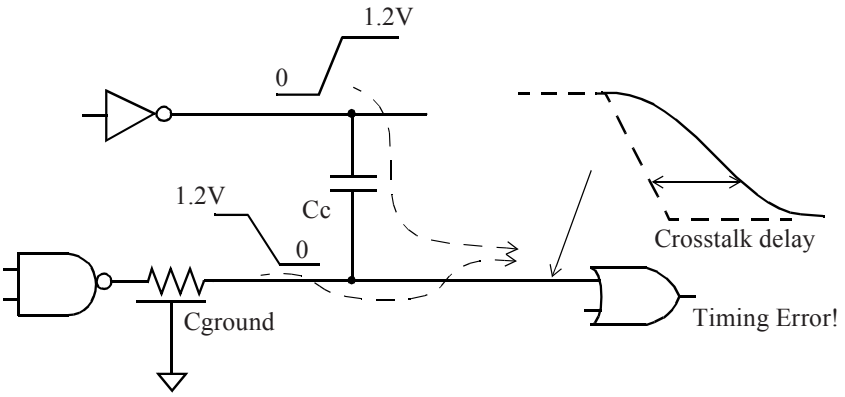


Figure 6-16 *Positive crosstalk delay.*

tim net. As before, the negative crosstalk affects the timing of the driving cell as well as the interconnect - the delay for both of these is reduced.

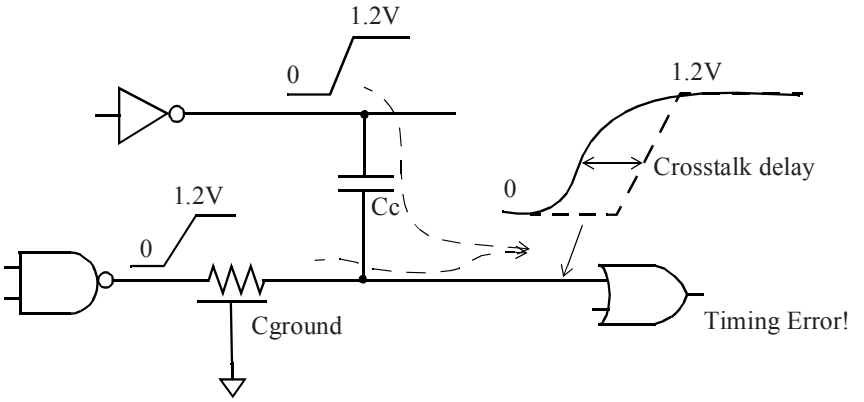


Figure 6-17 *Negative crosstalk delay.*

Note that the worst positive and worst negative crosstalk delays are computed separately for rise and fall delays. The worst set of aggressors for the

rise max, rise min, fall max, fall min delays with crosstalk are, in general, different. This is described in the subsections below.

6.3.3 Accumulation with Multiple Aggressors

The crosstalk delay analysis with multiple aggressors involves accumulating the contributions due to crosstalk for each of the aggressors. This is similar to the analysis for crosstalk glitches described in Section 6.2. When multiple nets switch concurrently, the crosstalk delay effect on the victim gets compounded due to multiple aggressors.

Most analyses for coupling due to multiple aggressors add the incremental contribution from each aggressor and compute the cumulative effect on the victim. This may appear conservative, however it does indicate the worst-case crosstalk delay on the victim.

Similar to the analysis of multiple aggressors for crosstalk glitch analysis, contributions can also be added using root-mean-squared (RMS) which is less pessimistic than the straight sum of individual contributions.

6.3.4 Aggressor Victim Timing Correlation

The handling of timing correlation for crosstalk delay analysis is conceptually similar to the timing correlation for crosstalk glitch analysis described in Section 6.2. The crosstalk can affect the delay of the victim, only if the aggressor can switch at the same time as the victim. This is determined using the timing windows of the aggressor and the victim. As described in Section 6.2, the *timing windows* represent the *earliest* and the *latest* switching times during which a net may switch within a clock cycle. If the timing windows of the aggressor and the victim overlap, the crosstalk effect on delay is computed. For multiple aggressors, the timing windows for multiple aggressors are also analyzed similarly. The possible effect in various timing bins is computed and the timing bin with the worst crosstalk delay impact is considered for delay analysis.

Consider the example below where three aggressor nets can impact the timing of the victim net. The aggressor nets (*A1*, *A2*, *A3*) are capacitively coupled to the victim net (*V*) and also their timing windows overlap with that of the victim. Figure 6-18 shows the timing windows and the possible crosstalk delay impact caused by each aggressor. Based upon the timing windows, the crosstalk delay analysis determines the worst possible combination of the aggressor switching that causes the largest crosstalk delay impact. In this example, the timing window overlap region is divided into three bins - each bin shows the possible aggressors switching. Bin 1 has *A1* and *A2* switching which can result in crosstalk delay impact of 0.26 (= 0.12 + 0.14). Bin 2 has *A1* switching which can result in crosstalk delay impact of 0.14. Bin 3 has *A3* switching which can result in crosstalk delay impact of 0.23. Thus, bin 1 has the worst possible crosstalk delay impact of 0.26.

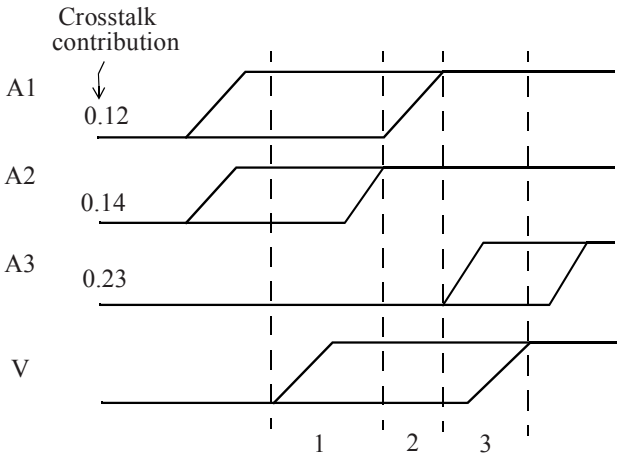


Figure 6-18 *Timing windows and crosstalk contributions from various aggressors.*

As indicated previously, the crosstalk delay analysis computes the four types of crosstalk delays separately. The four types of crosstalk delays are **positive rise delay** (rise edge moves forward in time), **negative rise delay** (rise edge moves backward in time), **positive fall delay** and **negative fall delay**. In general, a net can have different sets of aggressors in each of these

four cases. For example, a net can be coupled to aggressors *A1*, *A2*, *A3* and *A4*. During crosstalk delay analysis, it is possible that *A1*, *A2*, *A4* contribute to positive rise and negative fall delay contributions whereas *A2* and *A3* contribute to negative rise and positive fall delay contributions.

6.3.5 Aggressor Victim Functional Correlation

In addition to timing windows, crosstalk delay calculation can consider the *functional correlation* between various signals. For example, the scan control signals only switch during the scan mode and are steady during functional or mission mode of the design. Thus, the scan control signals can-not be aggressors during the functional mode. The scan control signals can only be aggressors during the scan mode in which case these signals can–not be combined with the other functional signals for worst-case noise computation.

Another example of functional correlation is a scenario where two aggressors are complements of each other. For such cases, both signal and its complement can never be switching in the same direction for crosstalk noise computation. This type of functional correlation information, when available, can be utilized so that the crosstalk analysis results are not pessimistic by ensuring that only the signals which can actually switch together are included as aggressors.

6.4 Timing Verification Using Crosstalk Delay

The following four types of crosstalk delay contributions are computed for every cell and interconnect in the design:

i. Positive rise delay (rise edge moves forward in time)

ii. Negative rise delay (rise edge moves backward in time)

iii. Positive fall delay (fall edge moves forward in time)

iv. Negative fall delay (fall edge moves backward in time)

The crosstalk delay contributions are then utilized during timing analysis for the verification of the max and min paths (setup and hold checks). The clock path for the launch and capture flip-flops are handled differently. The details of the data path and clock path analyses for the setup and hold checks are described in this section.

6.4.1 Setup Analysis

The STA with crosstalk analysis verifies the design with the worst-case crosstalk delays for the data path and the clock paths. Consider the logic shown in Figure 6-19 where crosstalk can occur at various nets along the data path and along the clock paths. The worst condition for setup check is when both the launch clock path and the data path have positive crosstalk and the capture clock path has negative crosstalk. The positive crosstalk contributions on launch clock path and data path delay the arrival of data at the capture flip-flop. In addition, the negative crosstalk on capture clock path results in capture flip-flop being clocked early.

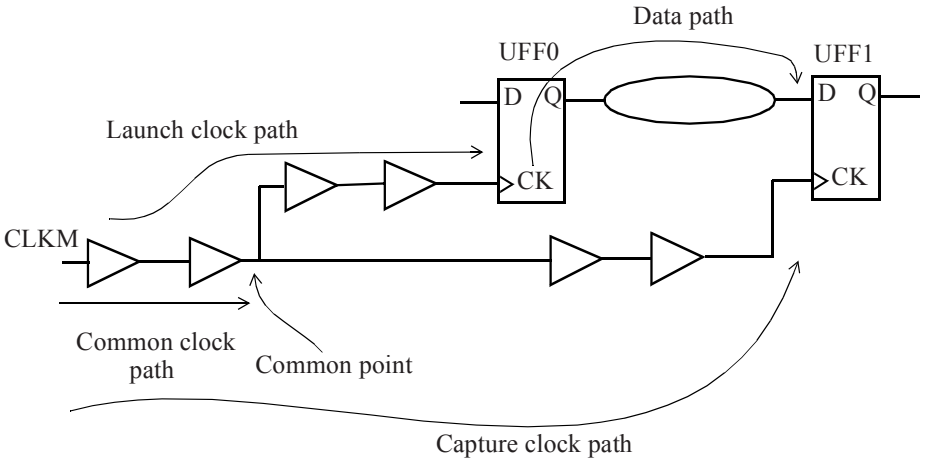


Figure 6-19 *Crosstalk for data path and clock path.*

Based upon above description, the setup (or max path) analysis assumes that:

- Launch clock path sees positive crosstalk delay so that the data is launched late.
- Data path sees positive crosstalk delay so that it takes longer for the data to reach the destination.
- Capture clock path sees negative crosstalk delay so that the data is captured by the capture flip-flop early.

Since the launch and capture clock edges for a setup check are different (normally one clock cycle apart), the common clock path (Figure 6-19) can have different crosstalk contributions for the launch and capture clock edges.

6.4.2 Hold Analysis

The worst-case hold (or min path) analysis for STA is analogous to the worst-case setup analysis described in the preceding subsection. Based upon the logic shown in Figure 6-19, the worst condition for hold check occurs when both the launch clock path and the data path have negative crosstalk and the capture clock path has positive crosstalk. The negative crosstalk contributions on launch clock path and data path result in early arrival of the data at the capture flip-flop. In addition, the positive crosstalk on capture clock path results in capture flip-flop being clocked late.

There is one important difference between the hold and setup analyses related to crosstalk on the common portion of the clock path. The launch and capture clock edge are normally the same edge for the hold analysis. The clock edge through the common clock portion cannot have different crosstalk contributions for the launch clock path and the capture clock path. Therefore, the worst-case hold analysis removes the crosstalk contribution from the common clock path.

The worst-case hold (or min path) analysis for STA with crosstalk assumes:

- Launch clock (not including the common path) sees negative crosstalk delay so that the data is launched early.
- Data path sees negative crosstalk delay so that it reaches the destination early.
- Capture clock (not including the common path) sees positive crosstalk delay so that the data is captured by the capture flip-flop late.

As described above, the crosstalk impact on the common portion of the clock tree is *not* considered for the hold analysis. The positive crosstalk contribution of the launch clock and negative crosstalk contribution of the capture clock are only computed for the *non-common* portions of the clock tree. In STA reports for hold analysis, the common clock path may show different crosstalk contributions for the launch clock path and the capture clock path. However, the crosstalk contributions from the common clock path are removed as a separate line item labeled as common path pessimism removal. Examples of common path pessimism removal in STA reports are provided in Section 10.1.

As described in the preceding subsection, the setup analysis concerns two different edges of the clock which may potentially be impacted differently in time. Thus, the common path crosstalk contributions are considered for both the launch and the capture clock paths during setup analysis.

The clock signals are critical since any crosstalk on the clock tree directly translates into clock jitter and impacts the performance of the design. Thus, special considerations should be adopted for reducing crosstalk on the clock signals. A common noise avoidance method is shielding of the clock tree - this is discussed in detail in Section 6.6.

6.5 Computational Complexity

A large nanometer design is generally too complex to allow for every coupling capacitance to be analyzed with reasonable turnaround time. The parasitic extraction of a typical net contains coupling capacitances to many neighboring signals. A large design will normally require appropriate settings for the parasitic extraction and crosstalk delay and glitch analyses. These settings are selected to provide acceptable accuracy for the analyses while ensuring that the CPU requirements remain feasible. This section describes some of the techniques that can be used for the analysis of a large nanometer design.

Hierarchical Design and Analysis

Hierarchical methodology for verifying a large design was introduced in Section 4.5. A similar approach is also applicable for reducing the complexity of extraction and analyses.

For a large design, it is normally not practical to obtain parasitic extraction in one run. The parasitics for each hierarchical block can be extracted separately. This in turn requires that a hierarchical design methodology be used for the design implementation. This implies that there be no coupling between signals inside the hierarchical block and signals outside the block. This can be achieved either with no routing over the block or by adding a shield layer over the block. In addition, signal nets should not be routed close to the boundary of the block and any nets routed close to the boundary of the block should be shielded. This avoids any coupling with the nets from other blocks.

Filtering of Coupling Capacitances

Even for a medium sized block, the parasitics will normally include a large number of very small coupling capacitances. The small coupling capacitances can be filtered during extraction or during the analysis procedures.

This filtering can be based upon the following criteria:

i. *Small value*: Very small coupling capacitances, for example, below 1fF, can be ignored for the crosstalk or noise analysis. During extraction, the small couplings can be treated as grounded capacitances.

ii. *Coupling ratio*: The impact of coupling on a victim is based upon the relative value of the coupling capacitance to the total capacitance of the victim net. Aggressor nets with a small coupling ratio, for example, below 0.001, can be excluded from crosstalk delay or glitch analyses.

iii. *Lumping small aggressors together*: Multiple aggressors with very small contributions can be mapped to one larger virtual aggressor. This can be pessimistic but can simplify the analysis. Some of the possible pessimism can be mitigated by switching a subset of the aggressors. The exact subset of switching aggressors can be determined by statistical methods.

6.6 Noise Avoidance Techniques

The preceding sections described the impact and analysis of crosstalk effects. In this section, we describe some noise avoidance techniques which can be utilized in the physical design phase.

i. *Shielding:* This method requires that shield wires are placed on either side of the critical signals. The shields are connected to power or ground rails. The shielding of critical signals ensures that there are no active aggressors for the critical signals since the nearest neighbors in the same metal layer are shield traces at a fixed potential. While there can be some coupling from routes in the different metal layers, most of the coupling capacitances are due to the capacitive coupling in the same layer. Since the immediate metal layers (above and below) would normally be routed orthogonally, the capacitive coupling across layers is

minimized. Thus, placing shield wires in the same metal layer ensures that there is minimal coupling for the critical signals. In cases where shielding with ground or power rails is not possible due to routing congestion, signal with low switching activity such as scan control which are fixed during functional mode can be routed as immediate neighbors for the critical signals. These shielding approaches ensure that there is no crosstalk due to capacitive coupling of the neighbors.

ii. Wire spacing: This reduces the coupling to the neighboring nets.

iii. Fast slew rate: A fast slew rate on the net implies that the net is less susceptible to crosstalk and is inherently immune to crosstalk effects.

iv. Maintain good stable supply: This is important not for crosstalk but for minimizing jitter due to power supply variations. Significant noise can be introduced on the clock signals due to noise on the power supply. Adequate decoupling capacitances should be added to minimize noise on the power supply.

v. Guard ring: A guard ring (or double guard ring) in the substrate helps in shielding the critical analog circuitry from digital noise.

vi. Deep n-well: This is similar to the above as having deep n-well[1] for the analog portions helps prevent noise from coupling to the digital portions.

vii. Isolating a block: In a hierarchical design flow, routing halos can be added to the boundary of the blocks; furthermore, isolation buffers could be added to each of the IO of the block.

❑

1. See [MUK86] in Bibliography.

CHAPTER

7

Configuring the STA Environment

This chapter describes how to set up the environment for static timing analysis. Specification of correct constraints is important in analyzing STA results. The design environment should be specified accurately so that STA analysis can identify all the timing issues in the design. Preparing for STA involves amongst others, setting up clocks, specifying IO timing characteristics, and specifying false paths and multicycle paths. It is important to understand this chapter thoroughly before proceeding with the next chapter on timing verification.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_7,

7.1 What is the STA Environment?

Most digital designs are synchronous where the data computed from the previous clock cycle is latched in the flip-flops at the active clock edge. Consider a typical synchronous design shown in Figure 7-1. It is assumed that the Design Under Analysis (DUA) interacts with other synchronous designs. This means that the DUA receives the data from a clocked flip-flop and outputs data to another clocked flip-flop external to the DUA.

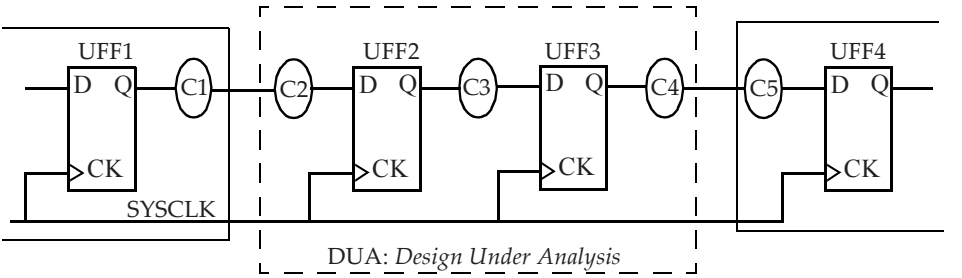


Figure 7-1 *A synchronous design.*

To perform STA on this design, one needs to specify the clocks to the flip-flops, and timing constraints for all paths leading into the design and for all paths exiting the design.

The example in Figure 7-1 assumes that there is only one clock and *C1*, *C2*, *C3*, *C4*, and *C5* represent combinational blocks. The combinational blocks *C1* and *C5* are outside of the design being analyzed.

In a typical design, there can be multiple clocks with many paths from one clock domain to another. The following sections describe how the environment is specified in such scenarios.

7.2 Specifying Clocks

To define a clock, we need to provide the following information:

i. *Clock source*: it can be a port of the design, or be a pin of a cell inside the design (typically that is part of a clock generation logic).

ii. *Period*: the time period of the clock.

iii. *Duty cycle*: the high duration (positive phase) and the low duration (negative phase).

iv. *Edge times*: the times for the rising edge and the falling edge.

Figure 7-2 shows the basic definitions. By defining the clocks, all the internal timing paths (all flip-flop to flip-flop paths) are constrained; this implies that all internal paths can be analyzed with just the clock specifications. The clock specification specifies that a flip-flop to flip-flop path must take one cycle. We shall later describe how this requirement (of one cycle timing) can be relaxed.

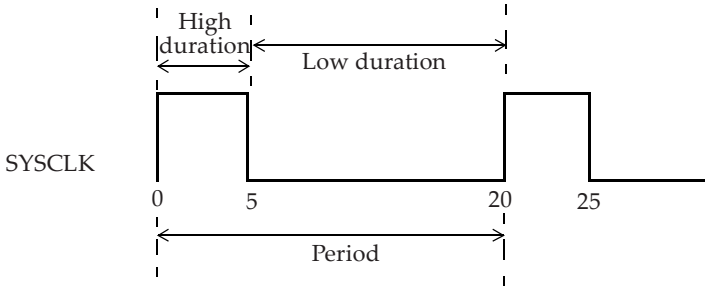


Figure 7-2 *A clock definition.*

Here is a basic clock specification[1].

```
create_clock \
  -name SYSCLK \
  -period 20 \
  -waveform {0 5} \
  [get_ports[2] SCLK]
```

The name of the clock is *SYSCLK* and is defined at the port *SCLK*. The period of *SYSCLK* is specified as 20 units - the default time unit is nanoseconds if none has been specified. (In general, the time unit is specified as part of the technology library.) The first argument in the waveform specifies the time at which rising edge occurs and the second argument specifies the time at which the falling edge occurs.

There can be any number of edges specified in a waveform option. However all the edges must be within one period. The edge times alternate starting from the first rising edge after time zero, then a falling edge, then a rising edge, and so on. This implies that all time values in the edge list must be monotonically increasing.

```
-waveform {time_rise time_fall time_rise time_fall ...}
```

In addition, there must be an even number of edges specified. The *waveform* option specifies the waveform within one clock period, which then repeats itself.

If no *waveform* option is specified, the default is:

```
-waveform {0, period/2}
```

1. The *specification* and *constraint* are used as synonyms to each other. These are all part of the SDC specifications.
2. See appendix on SDC regarding scenarios when object access commands, such as *get_ports* and *get_clocks,* should be used.

Here is an example of a clock specification with no waveform specification (see Figure 7-3).

```
create_clock -period 5 [get_ports SCAN_CLK]
```

In this specification, since no *-name* option is specified, the name of the clock is the same as the name of the port, which is *SCAN_CLK*.

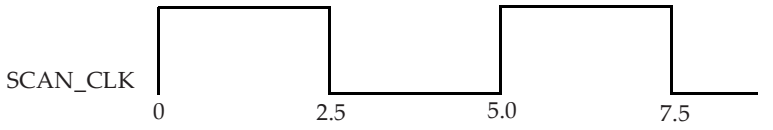


Figure 7-3 *Clock specification example.*

Here is another example of a clock specification in which the edges of the waveform are in the middle of a period (see Figure 7-4).

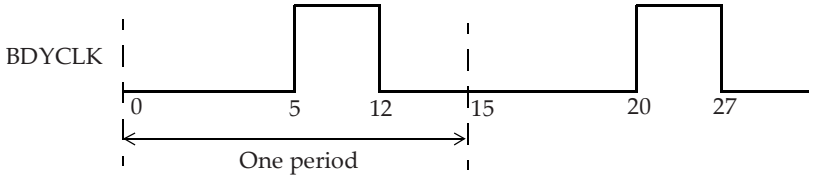


Figure 7-4 *Clock specification with arbitrary edges.*

```
create_clock -name BDYCLK -period 15 \
  -waveform {5 12} [get_ports GBLCLK]
```

The name of the clock is *BDYCLK* and it is defined at the port *GBLCLK*. In practice, it is a good idea to keep the clock name the same as the port name.

Here are some more clock specifications.

```
# See Figure 7-5(a):
create_clock -period 10 -waveform {5 10} [get_ports FCLK]
  # Creates a clock with the rising edge at 5ns and the
  # falling edge at 10ns.

# See Figure 7-5(b):
create_clock -period 125 \
  -waveform {100 150} [get_ports ARMCLK]
  # Since the first edge has to be rising edge,
  # the edge at 100ns is specified first and then the
  # falling edge at 150ns is specified. The falling edge
  # at 25ns is automatically inferred.
```

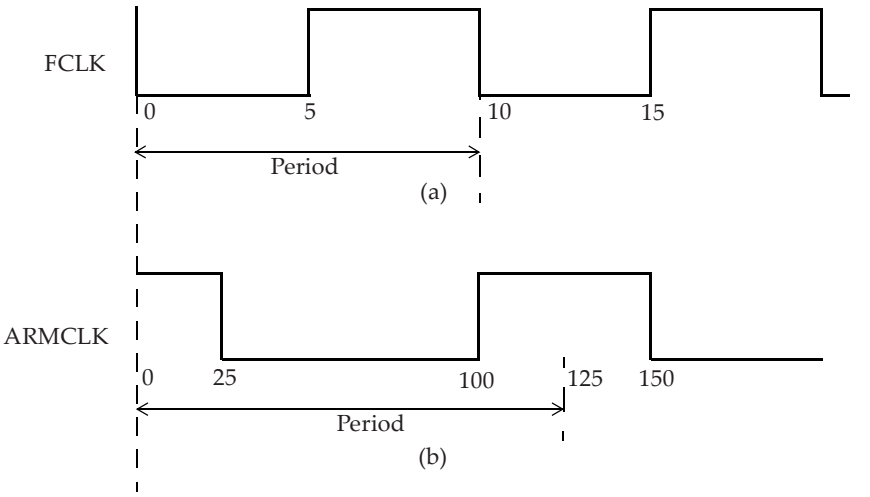


Figure 7-5 *Example clock waveforms.*

```
# See Figure 7-6(a):
create_clock -period 1.0 -waveform {0.5 1.375} MAIN_CLK
  # The first rising edge and the next falling edge
```

```
  # is specified. Falling edge at 0.375ns is inferred
  # automatically.

# See Figure 7-6(b):
create_clock -period 1.2 -waveform {0.3 0.4 0.8 1.0} JTAG_CLK
# Indicates a rising edge at 300ps, a falling edge at 400ps,
# a rising edge at 800ps and a falling edge at 1ns, and this
# pattern is repeated every 1.2ns.

create_clock -period 1.27 \
  -waveform {0 0.635} [get_ports clk_core]
```

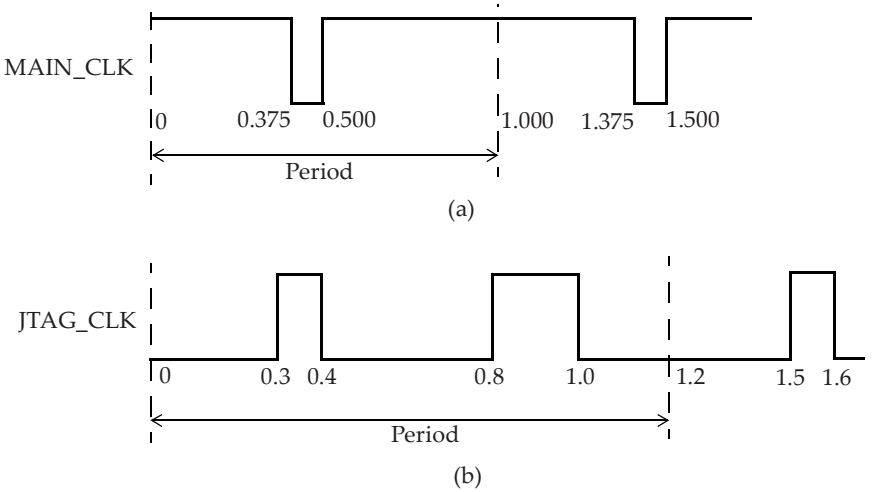


Figure 7-6 *Example with general clock waveforms.*

```
create_clock -name TEST_CLK -period 17 \
  -waveform {0 8.5} -add [get_ports {ip_io_clk[0]}]
  # The -add option allows more than one clock
  # specification to be defined at a port.
```

In addition to the above attributes, one can optionally specify the transition time (slew) at the source of the clock. In some cases, such as the output of some PLL[1] models or an input port, the tool cannot compute the transition time automatically. In such cases, it is useful to explicitly specify the transition time at the source of the clock. This is specified using the **set_clock_transition** specification.

```
set_clock_transition -rise 0.1 [get_clocks CLK_CONFIG]
set_clock_transition -fall 0.12 [get_clocks CLK_CONFIG]
```

This specification applies only for ideal clocks and is disregarded once the clock trees are built, at which point, actual transition times at the clock pins are used. If a clock is defined on an input port, use the *set_input_transition* specification (see Section 7.7) to specify the slew on the clock.

7.2.1 Clock Uncertainty

The timing uncertainty of a clock period can be specified using the **set_clock_uncertainty** specification. The uncertainty can be used to model various factors that can reduce the effective clock period. These factors can be the clock jitter and any other pessimism that one may want to include for timing analysis.

```
set_clock_uncertainty -setup 0.2 [get_clocks CLK_CONFIG]
set_clock_uncertainty -hold 0.05 [get_clocks CLK_CONFIG]
```

Note that the clock uncertainty for setup effectively reduces the available clock period by the specified amount as illustrated in Figure 7-7. For hold checks, the clock uncertainty for hold is used as an additional timing margin that needs to be satisfied.

1. Phase-locked loop: commonly used in an ASIC to generate high-frequency clocks.

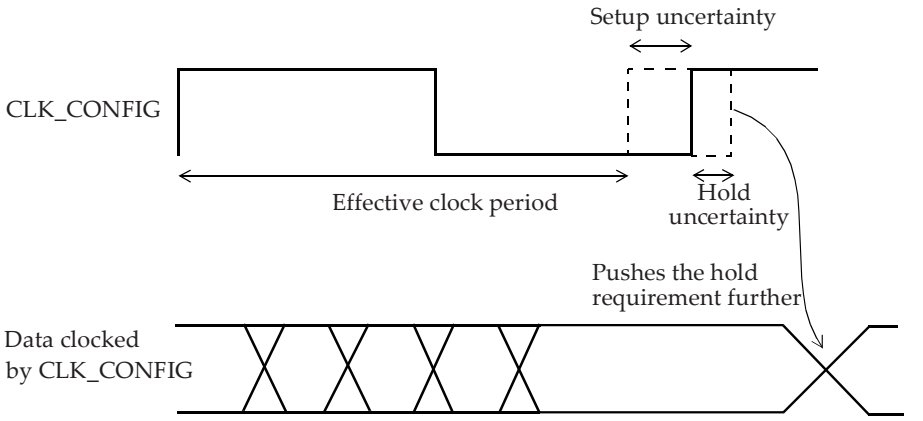


Figure 7-7 *Specifying clock uncertainty.*

The following commands specify the uncertainty to be used on paths crossing the specified clock boundaries, called **inter-clock uncertainty**.

```
set_clock_uncertainty -from VIRTUAL_SYS_CLK -to SYS_CLK \
  -hold 0.05
set_clock_uncertainty -from VIRTUAL_SYS_CLK -to SYS_CLK \
  -setup 0.3
set_clock_uncertainty -from SYS_CLK -to CFG_CLK -hold 0.05
set_clock_uncertainty -from SYS_CLK -to CFG_CLK -setup 0.1
```

Figure 7-8 shows a path between two different clock domains, *SYS_CLK* and *CFG_CLK*. Based on the above inter-clock uncertainty specifications, 100ps is used as an uncertainty for setup checks and 50ps is used as an uncertainty for hold checks.

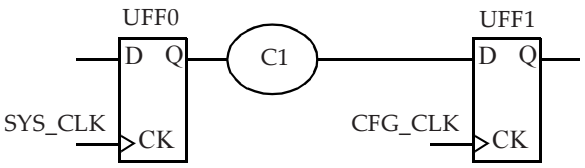


Figure 7-8 *Inter-clock paths.*

7.2.2 Clock Latency

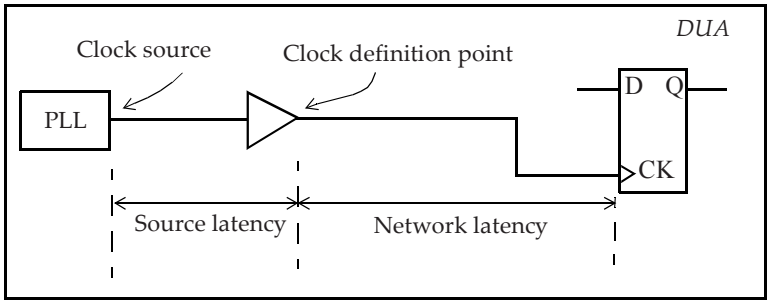
Latency of a clock can be specified using the *set_clock_latency* command.

```
# Rise clock latency on MAIN_CLK is 1.8ns:
set_clock_latency 1.8 -rise [get_clocks MAIN_CLK]
# Fall clock latency on all clocks is 2.1ns:
set_clock_latency 2.1 -fall [all_clocks]
# The -rise, -fall refer to the edge at the clock pin of a
# flip-flop.
```

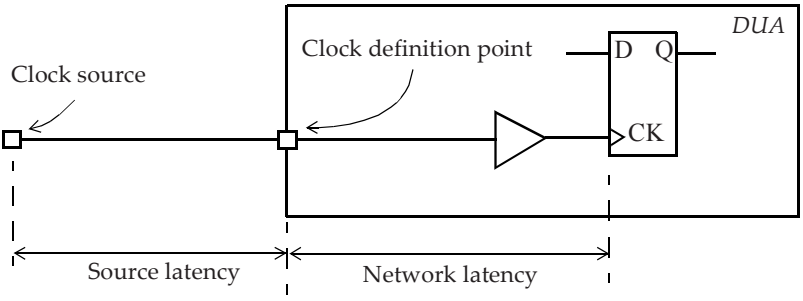
There are two types of clock latencies: **network latency** and **source latency**. Network latency is the delay from the clock definition point (*create_clock*) to the clock pin of a flip-flop. Source latency, also called **insertion delay**, is the delay from the clock source to the clock definition point. Source latency could represent either on-chip or off-chip latency. Figure 7-9 shows both the scenarios. The total clock latency at the clock pin of a flip-flop is the sum of the source and network latencies.

Here are some example commands that specify source and network latencies.

```
# Specify a network latency (no -source option) of 0.8ns for
# rise, fall, max and min:
set_clock_latency 0.8 [get_clocks CLK_CONFIG]
# Specify a source latency:
set_clock_latency 1.9 -source [get_clocks SYS_CLK]
# Specify a min source latency:
```



(a) On-chip clock source.



(b) Off-chip clock source.

Figure 7-9 *Clock latencies.*

```
set_clock_latency 0.851 -source -min [get_clocks CFG_CLK]
# Specify a max source latency:
set_clock_latency 1.322 -source -max [get_clocks CFG_CLK]
```

One important distinction to observe between source and network latency is that once a clock tree is built for a design, the network latency can be ignored (assuming *set_propagated_clock* command is specified). However, the source latency remains even after the clock tree is built. The network latency is an estimate of the delay of the clock tree prior to clock tree synthesis. After clock tree synthesis, the total clock latency from clock source to a

clock pin of a flip-flop is the source latency plus the actual delay of the clock tree from the clock definition point to the flip-flop.

Generated clocks are described in the next section and virtual clocks are described in Section 7.9.

7.3 Generated Clocks

A **generated clock** is a clock derived from a master clock. A master clock is a clock defined using the *create_clock* specification.

When a new clock is generated in a design that is based on a master clock, the new clock can be defined as a generated clock. For example, if there is a divide-by-3 circuitry for a clock, one would define a generated clock definition at the output of this circuitry. This definition is needed as STA does not know that the clock period has changed at the output of the divide-by logic, and more importantly what the new clock period is. Figure 7-10 shows an example of a generated clock which is a divide-by-2 of the master clock, *CLKP*.

```
create_clock -name CLKP 10 [get_pins UPLL0/CLKOUT]
  # Create a master clock with name CLKP of period 10ns
  # with 50% duty cycle at the CLKOUT pin of the PLL.
create_generated_clock -name CLKPDIV2 -source UPLL0/CLKOUT \
  -divide_by 2 [get_pins UFF0/Q]
  # Creates a generated clock with name CLKPDIV2 at the Q
  # pin of flip-flop UFF0. The master clock is at the CLKOUT
  # pin of PLL. And the period of the generated clock is double
  # that of the clock CLKP, that is, 20ns.
```

Can a new clock, that is, a master clock, be defined at the output of the flip-flop instead of a generated clock? The answer is yes, that it is indeed possible. However, there are some disadvantages. Defining a master clock instead of a generated clock creates a new clock domain. This is not a problem in general except that there are more clock domains to deal with

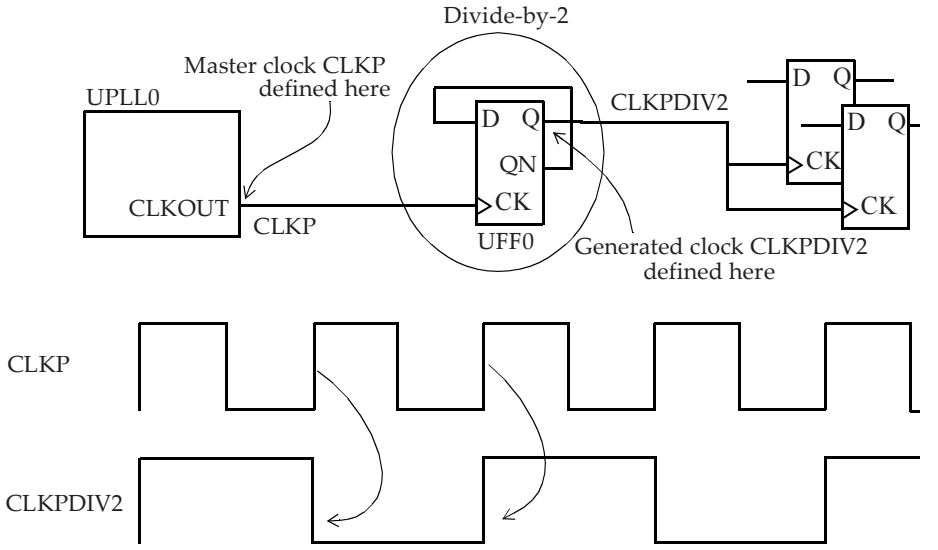


Figure 7-10 *Generated clock at output of divider.*

in setting up the constraints for STA. Defining the new clock as a generated clock does not create a new clock domain, and the generated clock is considered to be in phase with its master clock. The generated clock does not require additional constraints to be developed. Thus, one must attempt to define a new internally generated clock as a *generated clock* instead of deciding to declare it as another master clock.

Another important difference between a master clock and a generated clock is the notion of clock origin. In a master clock, the origin of the clock is at the point of definition of the master clock. In a generated clock, the clock origin is that of the master clock and not that of the generated clock. This implies that in a clock path report, the start point of a clock path is always the master clock definition point. This is a big advantage of a generated clock over defining a new master clock as the source latency is not automatically included for the case of a new master clock.

Figure 7-11 shows an example of a multiplexer with clocks on both its inputs. In this case, it is not necessary to define a clock on the output of the multiplexer. If the select signal is set to a constant, the output of the multiplexer automatically gets the correct clock propagated. If the select pin of the multiplexer is unconstrained, both the clocks propagate through the multiplexer for the purposes of the STA. In such cases, the STA may report paths between *TCLK* and *TCLKDIV5*. Note that such paths are not possible as the select line can select only one of the multiplexer inputs. In such a case, one may need to set a false path or specify an exclusive clock relationship between these two clocks to avoid incorrect paths being reported. This of course assumes that there are no paths between *TCLK* and *TCLKDIV5* elsewhere in the design.

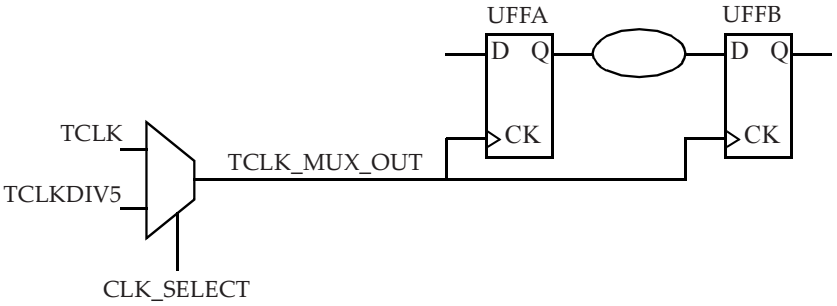


Figure 7-11 *Multiplexer selecting between two clocks.*

What happens if the multiplexer select signal is not static and can change during device operation? In such cases, clock gating checks are inferred for the multiplexer inputs. Clock gating checks are explained in Chapter 10; these checks ensure that the clocks at the multiplexer inputs switch safely with respect to the multiplexer select signal.

Figure 7-12 shows an example where the clock *SYS_CLK* is gated by the output of a flip-flop. Since the output of the flip-flop may not be a constant, one way to handle this situation is to define a generated clock at the output of the *and* cell which is identical to the input clock.

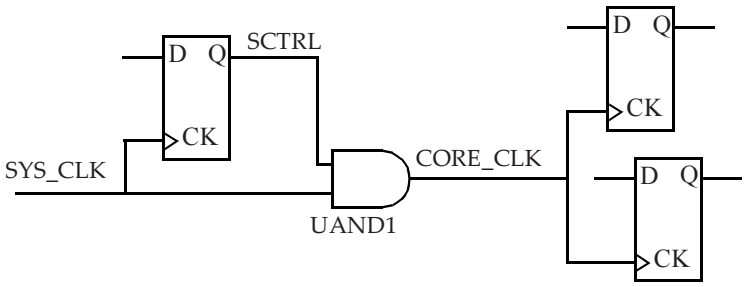


Figure 7-12 *Clock gated by a flip-flop.*

```
create_clock 0.1 [get_ports SYS_CLK]
  # Create a master clock of period 100ps with 50%
  # duty cycle.
create_generated_clock -name CORE_CLK -divide_by 1 \
  -source SYS_CLK [get_pins UAND1/Z]
  # Create a generated clock called CORE_CLK at the
  # output of the and cell and the clock waveform is
  # same as that of the master clock.
```

The next example is of a generated clock that has a frequency higher than that of the source clock. Figure 7-13 shows the waveforms.

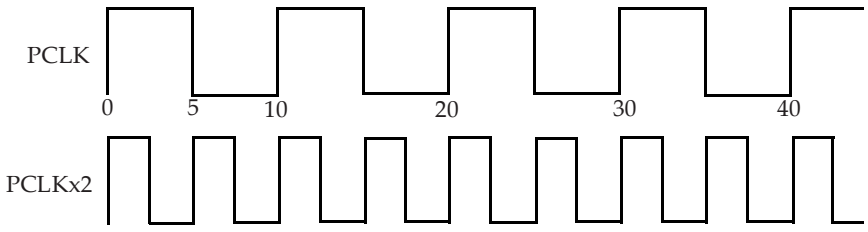


Figure 7-13 *Master clock and multiply-by-2 generated clock.*

```
create_clock -period 10 -waveform {0 5} [get_ports PCLK]
  # Create a master clock with name PCLK of period 10ns
  # with rise edge at 0ns and fall edge at 5ns.
create_generated_clock -name PCLKx2 \
  -source [get_ports PCLK] \
  -multiply_by 2 [get_pins UCLKMULTREG/Q]
  # Creates a generated clock called PCLKx2 from the
  # master clock PCLK and the frequency is double that of
  # the master clock. The generated clock is defined at the
  # output of the flip-flop UCLKMULTREG.
```

Note that the *-multiply_by* and the *-divide_by* options refer to the frequency of the clock, even though a clock period is specified in a master clock definition.

Example of Master Clock at Clock Gating Cell Output

Consider the clock gating example shown in Figure 7-14. Two clocks are fed to an *and* cell. The question is what is at the output of the *and* cell. If the input to the *and* cell are both clocks, then it is safe to define a new main clock at the output of the *and* cell, since it is highly unlikely that the output of the cell has any phase relationship with either of the input clocks.

```
create_clock -name SYS_CLK -period 4 -waveform {0 2} \
  [get_pins UFFSYS/Q]
create_clock -name CORE_CLK -period 12 -waveform {0 4} \
  [get_pins UFFCORE/Q]
create_clock -name MAIN_CLK -period 12 -waveform {0 2} \
  [get_pins UAND2/Z]
  # Create a master clock instead of a generated clock
  # at the output of the and cell.
```

One drawback of creating clocks at the internal pins is that it impacts the path delay computation and forces the designer to manually compute the source latencies.

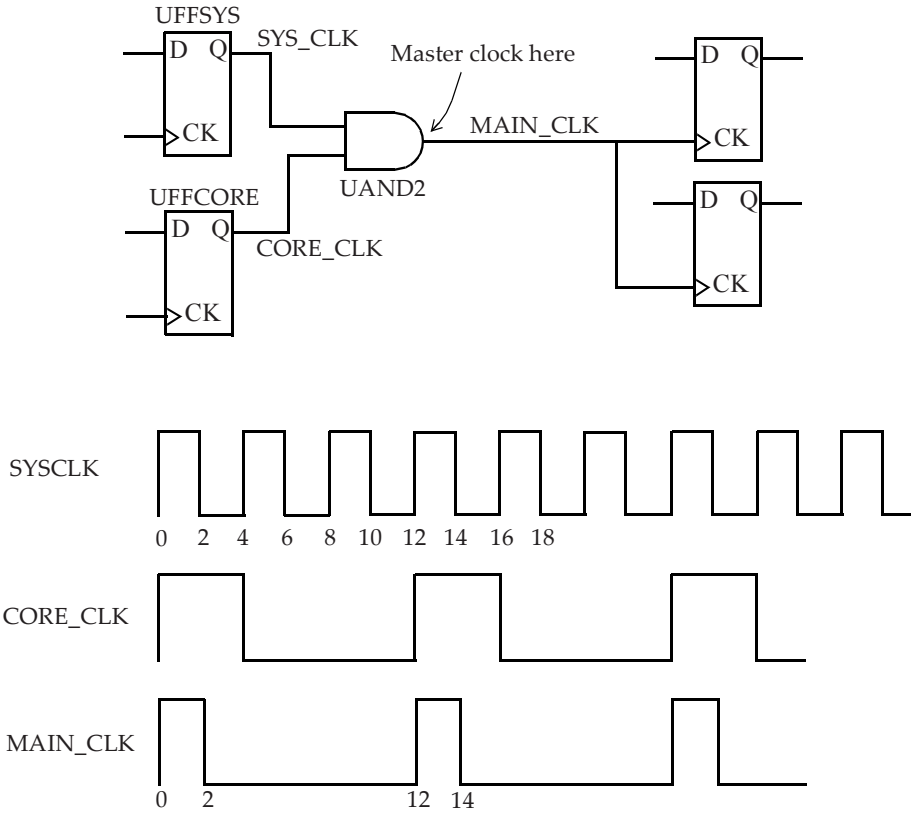


Figure 7-14 *Master clock at output of logic gate.*

Generated Clock using Edge and Edge_shift Options

Figure 7-15 shows an example of generated clocks. A divide-by-2 clock in addition to two out-of-phase clocks are generated. The waveforms for the clocks are also shown in the figure.

The clock definitions for this example are given below. The generated clock definition illustrates the **-edges** option, which is another way to define a generated clock. This option takes a list of edges {*rise*, *fall*, *rise*} of the source

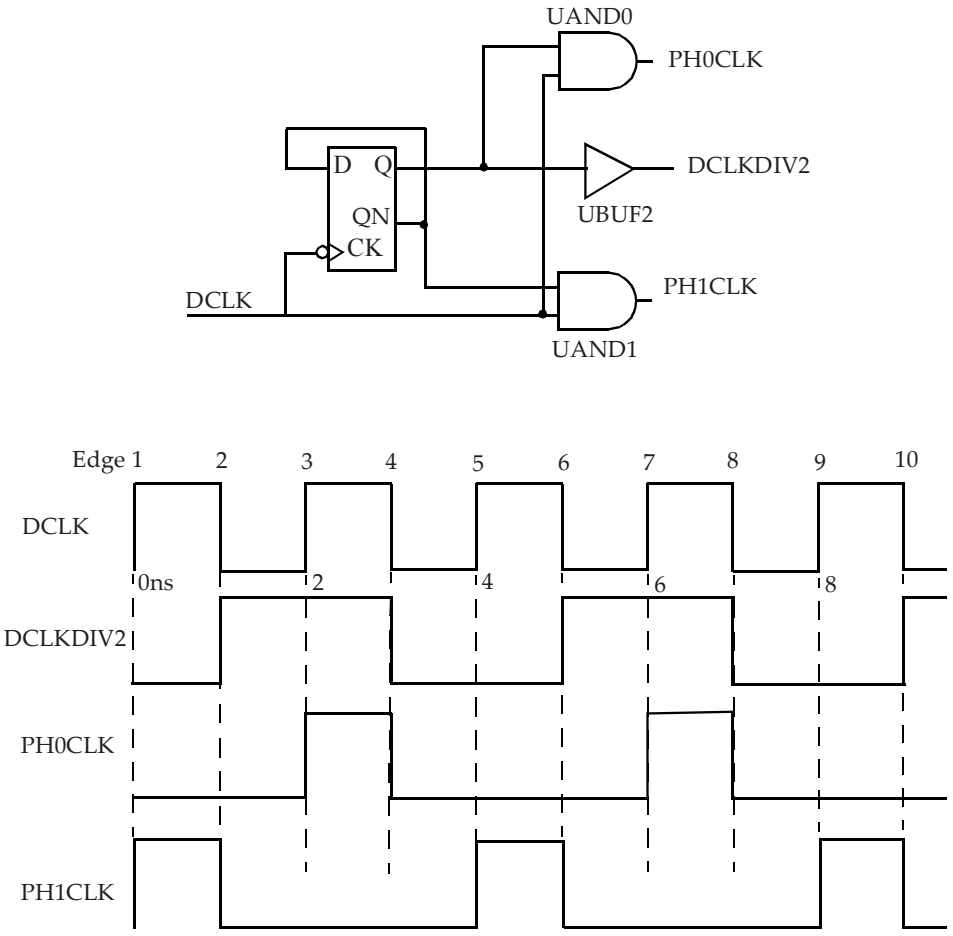


Figure 7-15 *Clock generation.*

master clock to form the new generated clock. The first rise edge of the master clock is the first edge, the first fall edge is edge 2, the next rise edge is edge 3, and so on.

```
create_clock 2 [get_ports DCLK]
  # Name of clock is DCLK, has period of 2ns with a
  # rise edge at 0ns and a fall edge at 1ns.
create_generated_clock -name DCLKDIV2 -edges {2 4 6} \
  -source DCLK [get_pins UBUF2/Z]
  # The generated clock with name DCLKDIV2 is defined at
  # the output of the buffer. Its waveform is formed by
  # having a rise edge at edge 2 of the source clock,
  # fall edge at edge 4 of the source clock and the next
  # rise edge at edge 6 of the source clock.
create_generated_clock -name PH0CLK -edges {3 4 7} \
  -source DCLK [get_pins UAND0/Z]
  # The generated clock PH0CLK is formed using
  # the 3, 4, 7 edges of the source clock.
create_generated_clock -name PH1CLK -edges {1 2 5} \
  -source DCLK [get_pins UAND1/Z]
  # The generated clock with name PH1CLK is defined at
  # the output of the and cell and is formed with
  # edges 1, 2 and 5 of the source clock.
```

What if the first edge of the generated clock is a falling edge? Consider the generated clock *G3CLK* shown in Figure 7-16. Such a generated clock can be defined by specifying the edges 5, 7 and 10, as shown in the following clock specification. The falling edge at 1ns is inferred automatically.

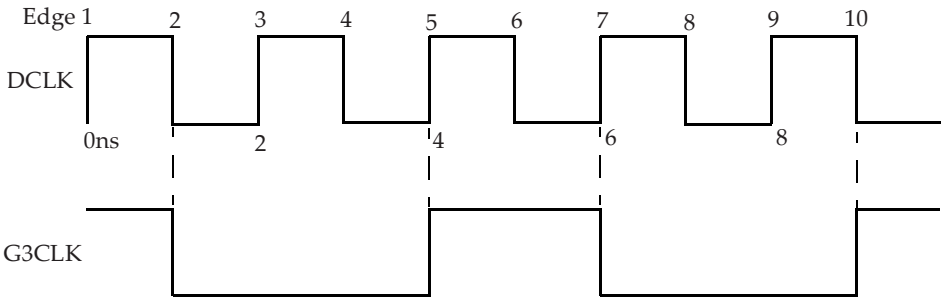


Figure 7-16 *Generated clock with a falling edge as first edge.*

```
create_generated_clock -name G3CLK -edges {5 7 10} \
  -source DCLK [get_pins UAND0/Z]
```

The **-edge_shift** option can be used in conjunction with the *-edges* option to specify any shift of the corresponding edges to form the new generated waveform. It specifies the amount of shift (in time units) for each edge in the edge list. Here is an example that uses this option.

```
create_clock -period 10 -waveform {0 5} [get_ports MIICLK]
create_generated_clock -name MIICLKDIV2 -source MIICLK \
  -edges {1 3 5} [get_pins UMIICLKREG/Q]
  # Create a divide-by-2 clock.
create_generated_clock -name MIIDIV2 -source MIICLK \
  -edges {1 1 5} -edge_shift {0 5 0} [get_pins UMIIDIV/Q]
  # Creates a divide-by-2 clock with a duty cycle different
  # from the source clock's value of 50%.
```

The list of edges in the edge list must be in non-decreasing order, though the same edge can be used for two entries to indicate a clock pulse independent of the source clocks' duty cycle. The *-edge_shift* option in the above example specifies that the first edge is obtained by shifting (edge 1 of source clock) by 0ns, the second edge is obtained by shifting (edge 1 of source clock) by 5ns and the third edge is obtained by shifting (edge 5 of source clock) by 0ns. Figure 7-17 shows the waveforms.

Generated Clock using Invert Option

Here is another example of a generated clock; this one uses the **-invert** option.

```
create_clock -period 10 [get_ports CLK]
create_generated_clock -name NCLKDIV2 -divide_by 2 -invert \
  -source CLK [get_pins UINVQ/Z]
```

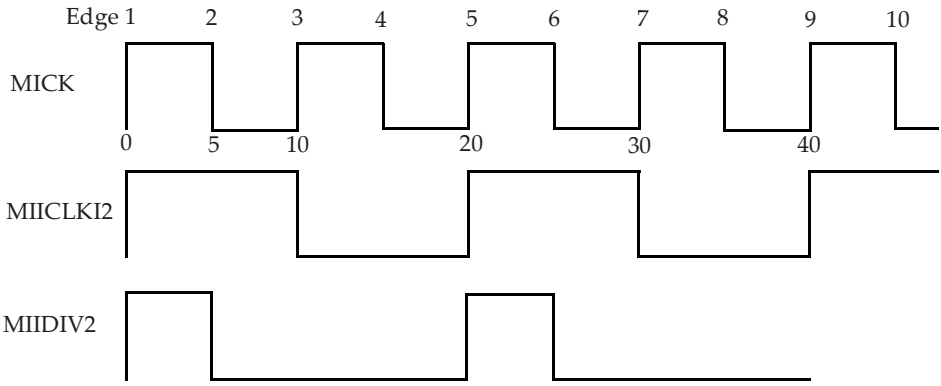


Figure 7-17 *Generated clocks using -edge_shift option.*

The *-invert* option applies the inversion to the generated clock after all other generated clock options are applied. Figure 7-18 shows a schematic that generates such an inverted clock.

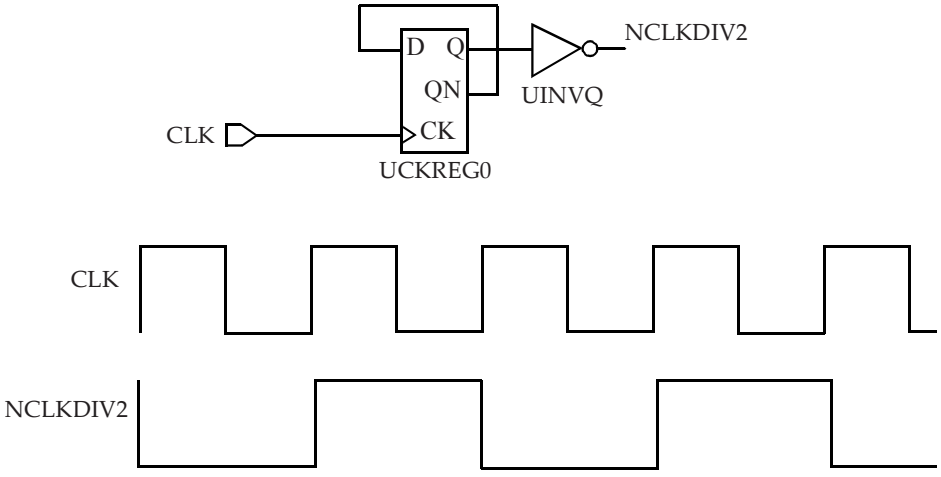


Figure 7-18 *Inverting a clock.*

Clock Latency for Generated Clocks

Clock latencies can be specified for generated clocks as well. A source latency specified on a generated clock specifies the latency from the definition of the master clock to the definition of the generated clock. The total clock latency to a clock pin of a flop-flop being driven by a generated clock is thus the sum of the source latency of the master clock, the source latency of the generated clock and the network latency of the generated clock. This is shown in Figure 7-19.

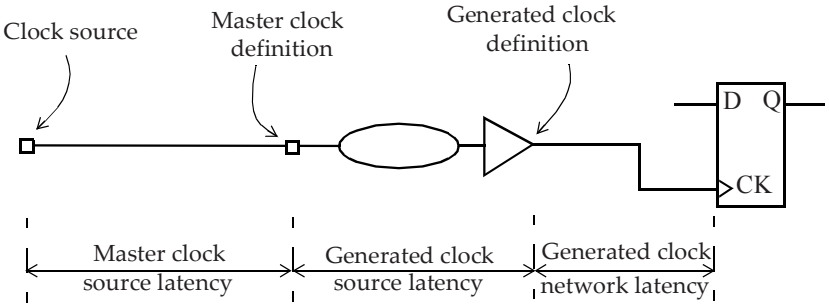


Figure 7-19 *Latency on generated clock.*

A generated clock can have another generated clock as its source, that is, one can have generated clocks of generated clocks, and so on. However, a generated clock can have only one master clock. More examples of generated clocks are described in later chapters.

Typical Clock Generation Scenario

Figure 7-20 shows a scenario of how a clock distribution may appear in a typical ASIC. The oscillator is external to the chip and produces a low frequency (10-50 MHz typical) clock which is used as a reference clock by the on-chip PLL to generate a high-frequency low-jitter clock (200-800 MHz typical). This PLL clock is then fed to a clock divider logic that generates the required clocks for the ASIC.

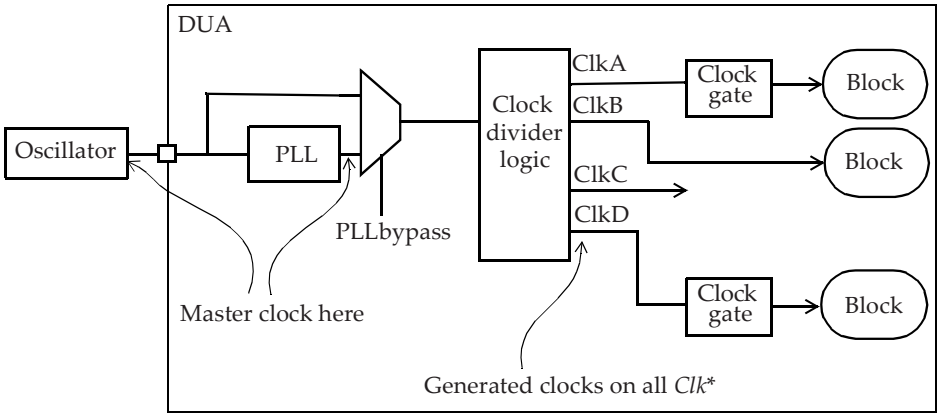


Figure 7-20 *Clock distribution in a typical ASIC.*

On some of the branches of the clock distribution, there may be clock gates that are used to turn off the clock to an inactive portion of the design to save power when necessary. The PLL can also have a multiplexer at its output so that the PLL can be bypassed if necessary.

A master clock is defined for the reference clock at the input pin of the chip where it enters the design, and a second master clock is defined at the output of the PLL. The PLL output clock has no phase relationship with the reference clock. Therefore, the output clock should *not* be a generated clock of the reference clock. Most likely, all clocks generated by the clock divider logic are specified as generated clocks of the master clock at the PLL output.

7.4 Constraining Input Paths

This section describes the constraints for the input paths. The important point to note here is that STA cannot check any timing on a path that is not constrained. Thus, all paths should be constrained to enable their analysis. Examples where one may not care about some logic and can leave such in-

puts unconstrained are described in later chapters. For example, one may not care about timing through inputs that are strictly control signals, and may determine that there is no need to specify the checks described in this section. However, this section assumes that we want to constrain the input paths.

Figure 7-21 shows an input path of the design under analysis (DUA). Flip-flop *UFF0* is external to the design and provides data to the flip-flop *UFF1* which is internal to the design. The data is connected through the input port *INP1*.

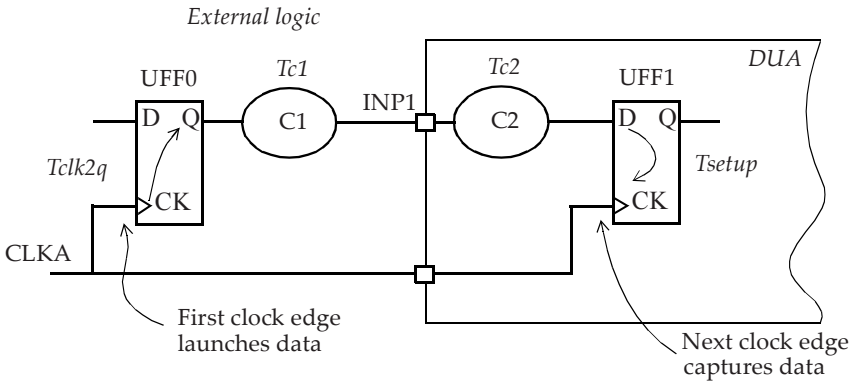


Figure 7-21 *Input port timing path.*

The clock definition for *CLKA* specifies the clock period, which is the total amount of time available between the two flip-flops *UFF0* and *UFF1*. The time taken by the external logic is *Tclk2q*, the *CK* to *Q* delay of the launch flip-flop *UFF0*, plus *Tc1*, the delay through the external combinational logic. Thus, the delay specification on an input pin *INP1* defines an external delay of *Tclk2q* plus *Tc1*. This delay is specified with respect to a clock, *CLKA* in this example.

Here is the input delay constraint.

```
set Tclk2q  0.9
set Tc1     0.6
set_input_delay -clock CLKA -max [expr Tclk2q + Tc1] \
  [get_ports INP1]
```

The constraint specifies that the external delay on input *INP1* is 1.5ns and this is with respect to the clock *CLKA*. Assuming the clock period for *CLKA* is 2ns, then the logic for *INP1* pin has only 500ps (= 2ns - 1.5ns) available for propagating internally in the design. This input delay specification maps into the input constraint that *Tc2* plus *Tsetup* of *UFF1* must be less than 500ps for the flip-flop *UFF1* to reliably capture the data launched by flip-flop *UFF0*. Note that the external delay above is specified as a max quantity.

Let us consider the case when we want to consider both max and min delays, as shown in Figure 7-22. Here are the constraints for this example.

```
create_clock -period 15 -waveform {5 12} [get_ports CLKP]
set_input_delay -clock CLKP -max 6.7 [get_ports INPA]
set_input_delay -clock CLKP -min 3.0 [get_ports INPA]
```

The max and min delays for *INPA* are derived from the *CLKP* to *INPA* delays. The max and min delays refer to the longest and shortest path delays respectively. These may also normally correspond to the worst-case slow (max timing corner) and the best-case fast (min timing corner). Thus, the max delay corresponds to the longest path delay at the max corner and the min delay corresponds to the shortest path delay at the min corner. In our example, 1.1ns and 0.8ns are the max and the min delay values for the *Tck2q*. The combinational path delay *Tc1* has a max delay of 5.6ns and a min delay of 2.2ns. The waveform on *INPA* shows the window in which the data arrives at the design input and when it is expected to be stable. The max delay from *CLKP* to *INPA* is 6.7ns (= 1.1ns + 5.6ns). The min delay is 3ns (= 0.8ns + 2.2ns). These delays are specified with respect to the active edge of the clock. Given the external input delays, the available setup time

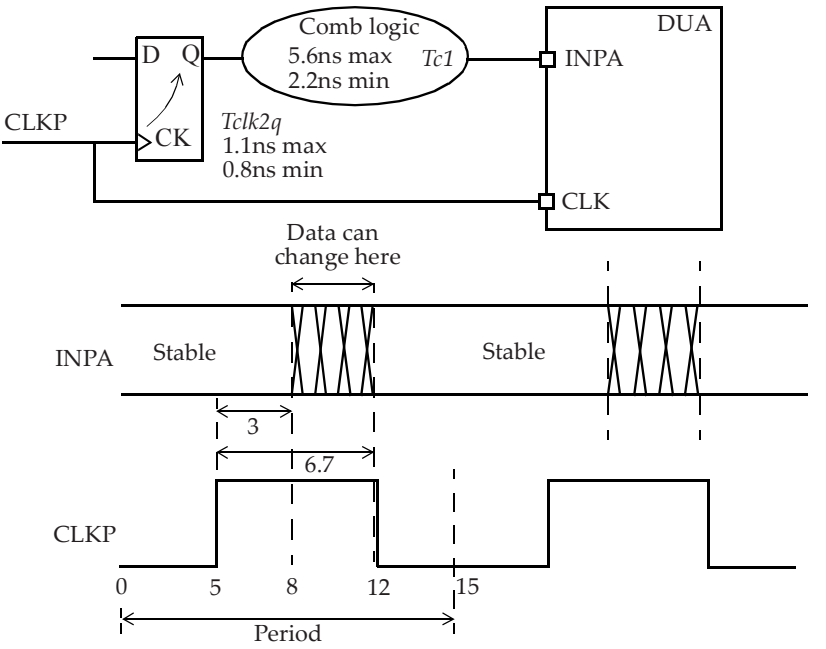


Figure 7-22 *Max and min delays on input port.*

internal to the design is the min of 8.3ns (= 15ns - 6.7ns) at the slow corner and 12ns (= 15ns - 3.0ns) at the fast corner. Thus, 8.3ns is the available time to reliably capture the data internal to the DUA.

Here are some more examples of input constraints.

```
set_input_delay -clock clk_core 0.5 [get_ports bist_mode]
set_input_delay -clock clk_core 0.5 [get_ports sad_state]
```

Since the max or min options are not specified, the value of 500ps applies to both the max and min delays. This external input delay is specified with respect to the rising edge of clock *clk_core* (the **-clock_fall** option has to be

used if the input delay is specified with respect to the falling edge of the clock).

7.5 Constraining Output Paths

This section describes the constraints for the output paths with the help of three illustrative examples below.

Example A

Figure 7-23 shows an example of a path through an output port of the design under analysis. *Tc1* and *Tc2* are the delays through the combinational logic.

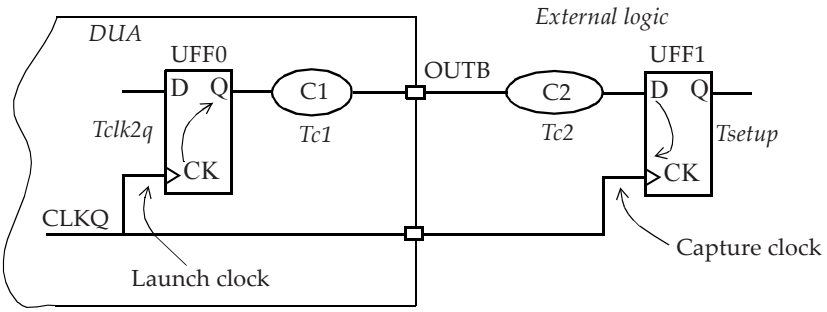


Figure 7-23 *Output port timing path for example A.*

The period for the clock *CLKQ* defines the total available time between the flip-flops *UFF0* and *UFF1*. The external logic has a total delay of *Tc2* plus *Tsetup*. This total delay, *Tc2* + *Tsetup*, has to be specified as part of the output delay specification. Note that the output delay is specified relative to the capture clock. Data must arrive at the external flip-flop *UFF1* in time to meet its setup requirement.

```
set Tc2     3.9
set Tsetup  1.1
set_output_delay -clock CLKQ -max [expr Tc2 + Tsetup] \
  [get_ports OUTB]
```

This specifies that the max external delay relative to the clock edge is *Tc2* plus *Tsetup*; and should correspond to the delay of 5ns. A min delay can be similarly specified.

Example B

Figure 7-24 shows an example with both min and max delays. The max path delay is 7.4ns (= max *Tc2* plus *Tsetup* = 7 + 0.4). The min path delay is -0.2ns (= min *Tc2* minus *Thold* = 0 - 0.2). Therefore the output specifications are:

```
create_clock -period 20 -waveform {0 15} [get_ports CLKQ]
set_output_delay -clock CLKQ -min -0.2 [get_ports OUTC]
set_output_delay -clock CLKQ -max 7.4 [get_ports OUTC]
```

The waveforms in Figure 7-24 show when *OUTC* has to be stable so that it is reliably captured by the external flip-flop. This depicts that the data must be ready at the output port before the required stable region starts and must remain stable until the end of the stable region. This maps into a requirement on the timing of the logic to the output port *OUTC* inside the DUA.

Example C

Here is another example that shows input and output specifications. This block has two inputs, *DATAIN* and *MCLK*, and one output *DATAOUT*. Figure 7-25 shows the intended waveforms.

```
create_clock -period 100 -waveform {5 55} [get_ports MCLK]
set_input_delay 25 -max -clock MCLK [get_ports DATAIN]
set_input_delay 5 -min -clock MCLK [get_ports DATAIN]
```

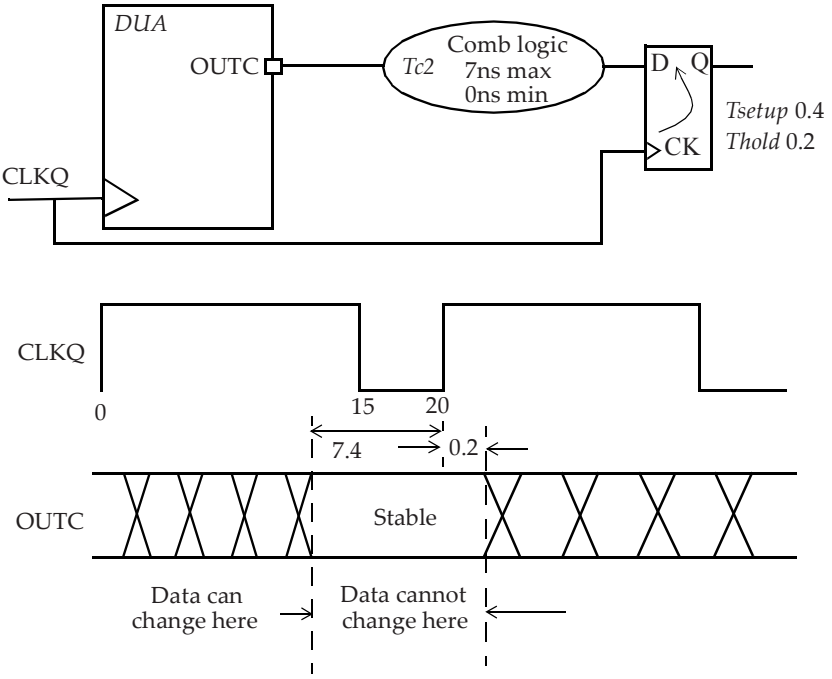


Figure 7-24 *Max and min delays on the example B output path.*

```
set_output_delay 20 -max -clock MCLK [get_ports DATAOUT]
set_output_delay -5 -min -clock MCLK [get_ports DATAOUT]
```

7.6 Timing Path Groups

Timing paths in a design can be considered as a collection of paths. Each path has a startpoint and an endpoint. See Figure 7-26 for some example paths.

In STA, the paths are timed based upon valid startpoints and valid endpoints. Valid startpoints are: input ports and clock pins of synchronous de-

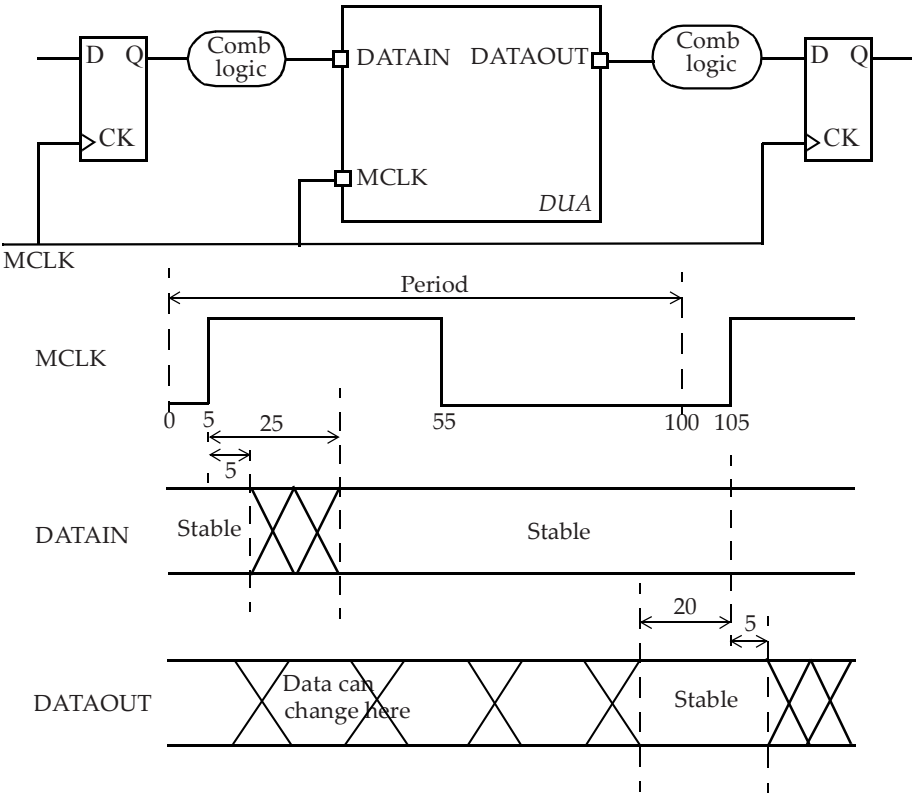


Figure 7-25 *Example C with input and output specifications.*

vices, such as flip-flops and memories. Valid endpoints are output ports and data input pins of synchronous devices. Thus, a valid timing path can be:

i. from an input port to an output port,

ii. from an input port to an input of a flip-flop or a memory,

iii. from the clock pin of a flip-flop or a memory to an input of flip-flop or a memory,

iv. from the clock pin of a flip-flop to an output port,

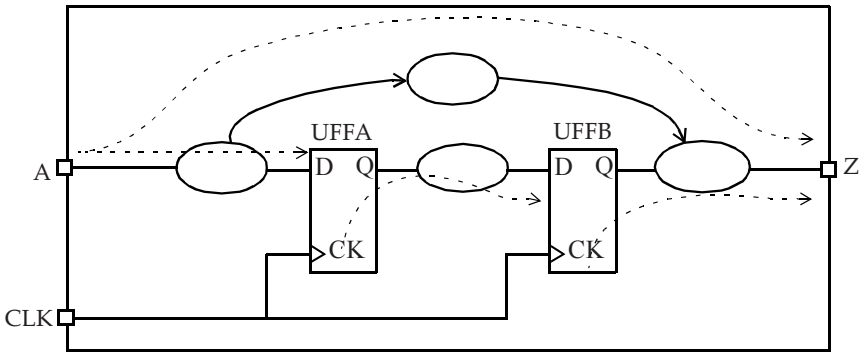


Figure 7-26 *Timing paths.*

v. from the clock pin of a memory to an output port, and so on.

The valid paths in Figure 7-26 are:

- input port *A* to *UFFA/D*,
- input port *A* to output port *Z*,
- *UFFA/CLK* to *UFFB/D*, and
- *UFFB/CLK* to output port *Z*.

Timing paths are sorted into **path groups** by the clock associated with the endpoint of the path. Thus, each clock has a set of paths associated with it. There is also a **default path group** that includes all non-clocked (asynchronous) paths.

In the example of Figure 7-27, the path groups are:

- *CLKA* group: Input port *A* to *UFFA/D*.
- *CLKB* group: *UFFA/CK* to *UFFB/D*.
- *DEFAULT* group: Input port *A* to output port *Z*, *UFFB/CK* to output port *Z*.

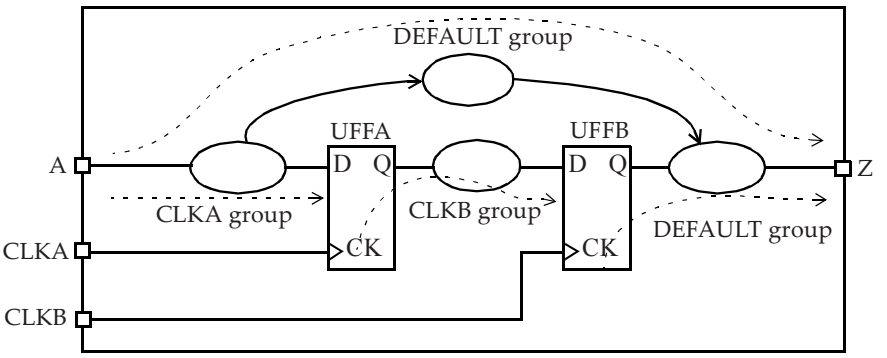


Figure 7-27 *Path groups.*

The static timing analysis and reporting are typically performed on each path group separately.

7.7 Modeling of External Attributes

While *create_clock*, *set_input_delay* and *set_output_delay* are enough to constrain all paths in a design for performing timing analysis, these are not enough to obtain accurate timing for the IO pins of the block. The following attributes are also required to accurately model the environment of a design. For inputs, one needs to specify the slew at the input. This information can be provided using:

- set_drive[1]
- set_driving_cell
- set_input_transition

1. This command is obsolete and not recommended.

For outputs, one needs to specify the capacitive load seen by the output pin. This is specified by using the following specification:

- set_load

7.7.1 Modeling Drive Strengths

The **set_drive** and **set_driving_cell** specifications are used to model the drive strength of the external source that drives an input port of the block. In absence of these specifications, by default, all inputs are assumed to have an infinite drive strength. The default condition implies that the transition time at the input pins is 0.

The *set_drive* explicitly specifies a value for the drive resistance at the input pin of the DUA. The smaller the drive value, the higher the drive strength. A resistance value of 0 implies an infinite drive strength.

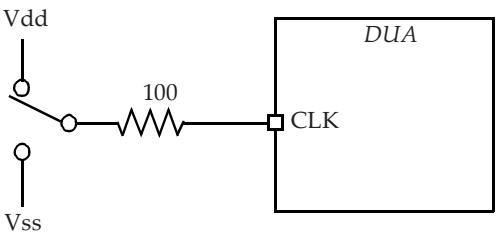


Figure 7-28 *Representation for set_drive specification example.*

```
set_drive 100 UCLK
  # Specifies a drive resistance of 100 on input UCLK.

# Rise drive is different from fall drive:
set_drive -rise 3 [all_inputs]
set_drive -fall 2 [all_inputs]
```

The drive of an input port is used to calculate the transition time at the first cell. The drive value specified is also used to compute the delay from the input port to the first cell in the presence of any RC interconnect.

```
Delay_to_first_gate =
  (drive * load_on_net) + interconnect_delay
```

The **set_driving_cell** specification offers a more convenient and accurate approach in describing the drive capability of a port. The *set_driving_cell* can be used to specify a cell driving an input port.

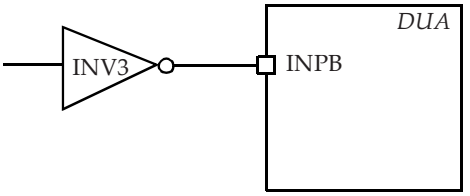


Figure 7-29 *Representation for set_driving_cell specification example.*

```
set_driving_cell -lib_cell INV3 \
  -library slow [get_ports INPB]
  # The input INPB is driven by an INV3 cell
  # from library slow.

set_driving_cell -lib_cell INV2 \
  -library tech13g [all_inputs]
  # Specifies that the cell INV2 from a library tech13g is
  # the driving cell for all inputs.

set_driving_cell -lib_cell BUFFD4 -library tech90gwc \
  [get_ports {testmode[3]}]
  # The input testmode[3] is driven by a BUFFD4 cell
  # from library tech90gwc.
```

Like the drive specification, the driving cell of an input port is used to calculate the transition time at the first cell and to compute the delay from the input port to the first cell in the presence of any interconnect.

One caveat of the *set_driving_cell* specification is that the incremental delay of the driving cell due to the capacitive load on the input port is included as an additional delay on the input.

As an alternate to the above approaches, the **set_input_transition** specification offers a convenient way of expressing the slew at an input port. A reference clock can optionally be specified. Here is the specification for the example shown in Figure 7-30 along with additional examples.

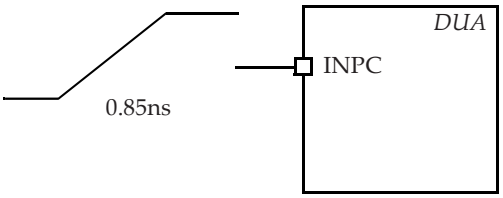


Figure 7-30 *Representation for set_input_transition specification example.*

```
set_input_transition 0.85 [get_ports INPC]
# Specifies an input transition of 850ps on port INPC.

set_input_transition 0.6 [all_inputs]
# Specifies a transition of 600ps on all input ports.

set_input_transition 0.25 [get_ports SD_DIN*]
# Specifies a transition of 250ps on all ports with
# pattern SD_DIN*.
# Min and max values can optionally be specified using
# the -min and -max options.
```

In summary, a slew value at an input is needed to determine the delay of the first cell in the input path. In the absence of this specification, an ideal transition value of 0 is assumed, which may not be realistic.

7.7.2 Modeling Capacitive Load

The **set_load** specification places a capacitive load on output ports to model the external load being driven by the output port. By default, the capacitive load on ports is 0. The load can be specified as an explicit capacitance value or as an input pin capacitance of a cell.

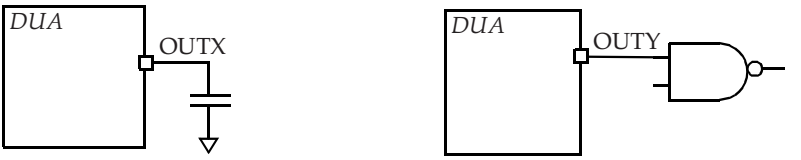


Figure 7-31 *Capacitive load on output port.*

```
set_load 5 [get_ports OUTX]
  # Places a 5pF load on output port OUTX.

set_load 25 [all_outputs]
# Sets 25pF load capacitance on all outputs.

set_load -pin_load 0.007 [get_ports {shift_write[31]}]
  # Place 7fF pin load on the specified output port.
  # A load on the net connected to the port can be
  # specified using the -wire_load option.
  # If neither -pin_load nor -wire_load option is used,
  # the default is the -pin_load option.
```

It is important to specify the load on outputs since this value impacts the delay of the cell driving the output. In the absence of such a specification, a load of 0 is assumed which may not be realistic.

The *set_load* specification can also be used for specifying a load on an internal net in the design. Here is an example:

```
set_load 0.25 [get_nets UCNT5/NET6]
  # Sets the net capacitance to be 0.25pF.
```

7.8 Design Rule Checks

Two of the frequently used design rules for STA are *max transition* and *max capacitance*. These rules check that all ports and pins in the design meet the specified limits for transition time[1] and capacitance. These limits can be specified using:

- set_max_transition
- set_max_capacitance

As part of the STA, any violations to these design rules are reported in terms of slack. Here are some examples.

```
set_max_transition 0.6 IOBANK
# Sets a limit of 600ps on IOBANK.

set_max_capacitance 0.5 [current_design]
# Max capacitance is set to 0.5pf on all nets in
# current design.
```

The capacitance on a net is calculated by taking the sum of all the pin capacitances plus any IO load plus any interconnect capacitance on the net. Figure 7-32 shows an example.

```
Total cap on net N1 =
  pin cap of UBUF1:pin/A +
```

1. As mentioned earlier, the terms “slew” and “transition time” are used interchangeably in this text.

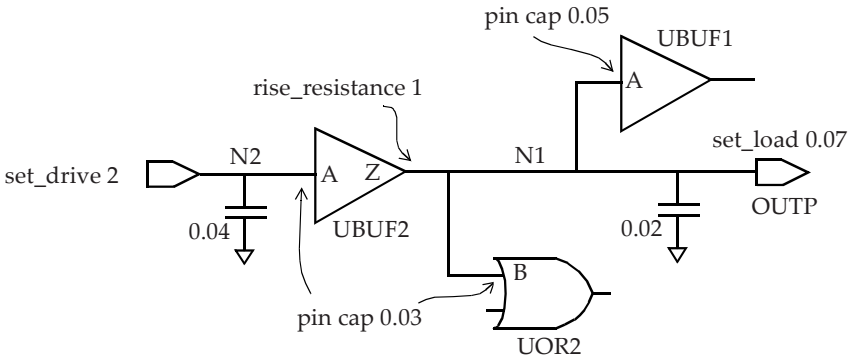


Figure 7-32 *Capacitance on various nets.*

```
  pin cap of UOR2:pin/B +
  load cap specified on output port OUTP +
  wire/routing cap
  = 0.05 + 0.03 + 0.07 + 0.02
  = 0.17pF

Total cap on net N2 =
  pin cap of UBUF2/A +
  wire/routing cap from input to buffer
  = 0.04 + 0.03
  = 0.07pF
```

Transition time is computed as part of the delay calculation. For the example of Figure 7-32 (assuming linear delay model for UBUF2 cell),

```
Transition time on pin UBUF2/A =
  drive of 2[1] * total cap on net N2
  = 2 * 0.07 = 0.14ns = 140ps
```

1. The library units for drive is assumed to be in Kohms.

```
Transition time on output port OUTP =
  drive resistance of UBUF2/Z * total cap of net N1 =
  1 * 0.17 = 0.17ns = 170ps
```

There are other design rule checks that can also be specified for a design. These are: **set_max_fanout** (specifies a fanout limit on all pins in design), **set_max_area** (for a design); however these checks apply for synthesis and not for STA.

7.9 Virtual Clocks

A **virtual clock** is a clock that exists but is not associated with any pin or port of the design. It is used as a reference in STA analysis to specify input and output delays relative to a clock. An example where virtual clock is applicable is shown in Figure 7-33. The design under analysis gets its clock from *CLK_CORE*, but the clock driving input port *ROW_IN* is *CLK_SAD*. How does one specify the IO constraint on input port *ROW_IN* in such cases? The same issue occurs on the output port *STATE_O*.

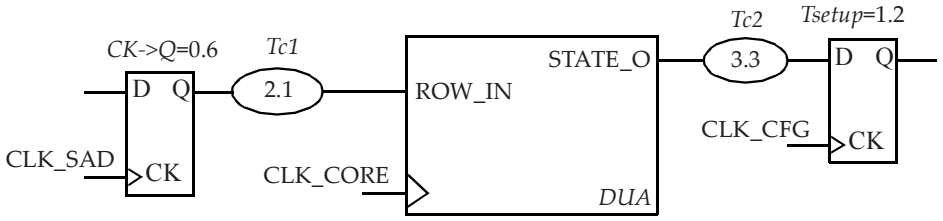


Figure 7-33 *Virtual clocks for CLK_SAD and CLK_CFG.*

To handle such cases, a virtual clock can be defined with no specification of the source port or pin. In the example of Figure 7-33, the virtual clock is defined for *CLK_SAD* and *CLK_CFG*.

```
create_clock -name VIRTUAL_CLK_SAD -period 10 -waveform {2 8}
```

```
create_clock -name VIRTUAL_CLK_CFG -period 8 \
  -waveform {0 4}
create_clock -period 10 [get_ports CLK_CORE]
```

Having defined these virtual clocks, the IO constraints can be specified relative to this virtual clock.

```
set_input_delay -clock VIRTUAL_CLK_SAD -max 2.7 \
  [get_ports ROW_IN]

set_output_delay -clock VIRTUAL_CLK_CFG -max 4.5 \
  [get_ports STATE_O]
```

Figure 7-34 shows the timing relationships on the input path. This constrains the input path in the design under analysis to be 5.3ns or less.

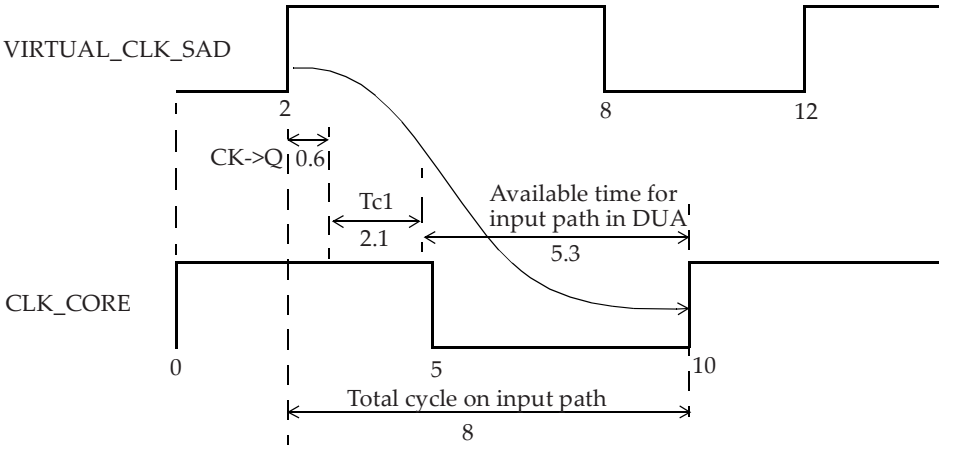


Figure 7-34 *Virtual clock and core clock waveform for input path.*

Figure 7-35 shows the timing relationships on the output path. This constrains the output path in the design under analysis to be 3.5ns or less.

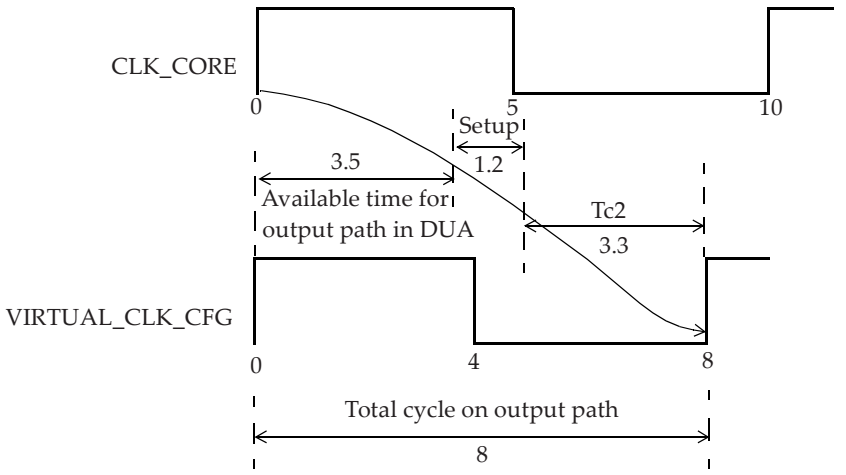


Figure 7-35 *Virtual clock and core clock waveform for output path.*

The *-min* option, when specified in the *set_input_delay* and *set_output_delay* constraints, is used for verifying the fast (or min) paths. The use of virtual clocks is just one approach to constrain the inputs and outputs (IO); a designer may choose other methods to constrain the IOs as well.

7.10 Refining the Timing Analysis

Four common commands that are used to constrain the analysis space are:

i. *set_case_analysis*: Specifies constant value on a pin of a cell, or on an input port.

ii. *set_disable_timing*: Breaks a timing arc of a cell.

iii. *set_false_path*: Specifies paths that are not real which implies that these paths are not checked in STA.

iv. *set_multicycle_path*: Specifies paths that can take longer than one clock cycle.

The *set_false_path* and *set_multicycle_path* specifications are discussed in greater detail in Chapter 8.

7.10.1 Specifying Inactive Signals

In a design, certain signals have a constant value in a specific mode of the chip. For example, if a chip has DFT logic in it, then the *TEST* pin of the chip should be at 0 in normal functional mode. It is often useful to specify such constant values to STA. This helps in reducing the analysis space in addition to not reporting any paths that are irrelevant. For example, if the *TEST* pin is not set as a constant, some odd long paths may exist that would never be true in functional mode. Such constant signals are specified by using the **set_case_analysis** specification.

```
set_case_analysis 0 TEST

set_case_analysis 0 [get_ports {testmode[3]}]
set_case_analysis 0 [get_ports {testmode[2]}]
set_case_analysis 0 [get_ports {testmode[1]}]
set_case_analysis 0 [get_ports {testmode[0]}]
```

If a design has many functional modes and only one functional mode is being analyzed, case analysis can be used to specify the actual mode to be analyzed.

```
set_case_analysis 1 func_mode[0]
set_case_analysis 0 func_mode[1]
set_case_analysis 1 func_mode[2]
```

Note that the case analysis can be specified on any pin in the design. Another common application of case analysis is when the design can run on multiple clocks, and the selection of the appropriate clock is controlled by multiplexers. To make STA analysis easier and reduce CPU run time, it is beneficial to do STA for each clock selection separately. Figure 7-36 shows an example of the multiplexers selecting different clocks with different settings.

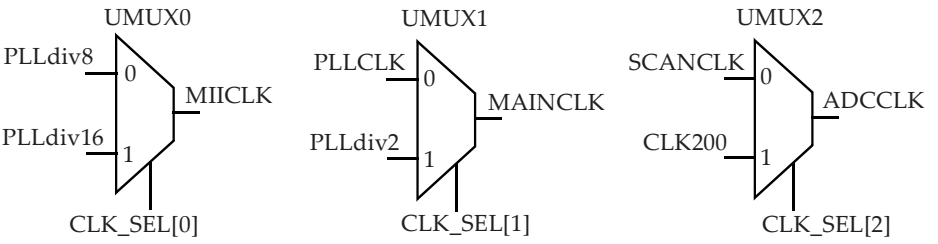


Figure 7-36 *Selecting clock mode for timing analysis.*

```
set_case_analysis 1 UCORE/UMUX0/CLK_SEL[0]
set_case_analysis 1 UCORE/UMUX1/CLK_SEL[1]
set_case_analysis 0 UCORE/UMUX2/CLK_SEL[2]
```

The first *set_case_analysis* causes *PLLdiv16* to be selected for *MIICLK*. The clock path for *PLLdiv8* is blocked and does not propagate through the multiplexer. Thus, no timing paths are analyzed using clock *PLLdiv8* (assuming that the clock does not go to any flip-flip prior to the multiplexer). Similarly, the last *set_case_analysis* causes *SCANCLK* to be selected for *ADCCLK* and the clock path for *CLK200* is blocked.

7.10.2 Breaking Timing Arcs in Cells

Every cell has timing arcs from its inputs to outputs, and a timing path may go through one of these cell arcs. In some situations, it is possible that a certain path through a cell cannot occur. For example, consider the scenario where a clock is connected to the select line of a multiplexer and the output of the multiplexer is part of a data path. In such a case, it may be useful to break the timing arc between the select pin and the output pin of the multiplexer. An example is shown in Figure 7-37. The path through the select line of multiplexer is not a valid data path. Such a timing arc can be broken by using the **set_disable_timing** SDC command.

```
set_disable_timing -from S -to Z [get_cells UMUX0]
```

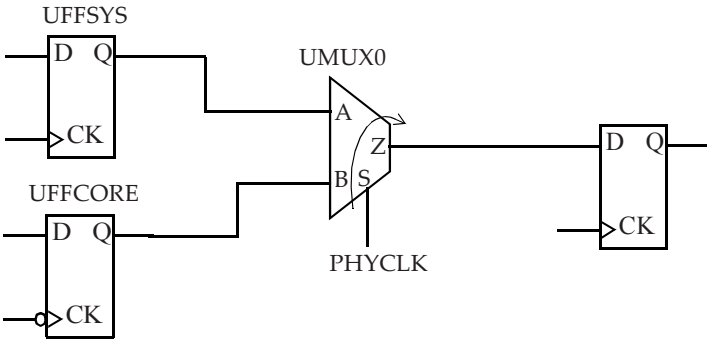


Figure 7-37 *An example timing arc to be disabled.*

Since the arc no longer exists, there are consequently fewer timing paths to analyze. Another example of a similar usage is to disable the minimum clock pulse width check of a flip-flop.

One should use caution when using the *set_disable_timing* command as it removes *all* timing paths through the specified pins. Where possible, it is preferable to use the *set_false_path* and the *set_case_analysis* commands.

7.11 Point-to-Point Specification

Point-to-point paths can be constrained by using the **set_min_delay** and **set_max_delay** specifications. These constrain the path delay between the from-pin and the to-pin to the values specified in the constraint. This constraint overrides any default single cycle timing paths and any multicycle path constraints for such paths. The **set_max_delay** constraint specifies the maximum delay for the specified path(s), while the **set_min_delay** constraint specifies the minimum delay for the specified path(s).

```
set_max_delay 5.0 -to UFF0/D
  # All paths to D-pin of flip-flop should take 5ns max.
```

```
set_max_delay 0.6 -from UFF2/Q -to UFF3/D
  # All paths between the two flip-flops should take a
  # max of 600ps.
set_max_delay 0.45 -from UMUX0/Z -through UAND1/A -to UOR0/Z
  # Sets max delay for the specified paths.
set_min_delay 0.15 -from {UAND0/A UXOR1/B} -to {UMUX2/SEL}
```

In the above examples, one needs to be careful that using non-standard startpoint and endpoint internal pins will force these to be the start and end points and will segment a path at those points.

One can also specify similar point-to-point constraints from one clock to another clock.

```
set_max_delay 1.2 -from [get_clocks SYS_CLK] \
  -to [get_clocks CFG_CLK]
  # All paths between these two clock domains are restricted
  # to a max of 1200ps.
set_min_delay 0.4 -from [get_clocks SYS_CLK] \
  -to [get_clocks CFG_CLK]
  # The min delay between any path between the two
  # clock domains is specified as 400ps.
```

If there are multiple timing constraints on a path, such as clock frequency, *set_max_delay* and *set_min_delay*, the most restrictive constraint is the one always checked. Multiple timing constraints can be caused by some global constraints being applied first and then some local constraints applied later.

7.12 Path Segmentation

Breaking up a timing path into smaller paths that can be timed is referred to as **path segmentation**.

A timing path has a startpoint and an endpoint. Additional startpoints and endpoints on a timing path can be created by using the *set_input_delay* and the *set_output_delay* specifications. The *set_input_delay*, which defines a startpoint, is typically specified on an output pin of a cell, while the *set_output_delay*, which defines a new endpoint, is typically specified on an input pin of a cell. These specifications define a new timing path which is a subset of the original timing path.

Consider the path shown in Figure 7-38. Once a clock is defined for *SYSCLK*, the timing path that is timed is from *UFF0/CK* to *UFF1/D*. If one is interested in reporting only the path delay from *UAND2/Z* to *UAND6/A*, then the following two commands are applicable:

```
set STARTPOINT [get_pins UAND2/Z]
set ENDPOINT [get_pins UAND6/A]
set_input_delay 0 $STARTPOINT
set_output_delay 0 $ENDPOINT
```

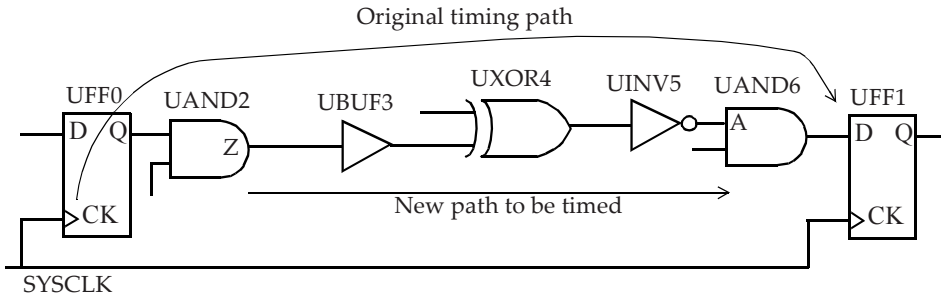


Figure 7-38 *Path segmentation.*

Defining these constraints causes the original timing path from *UFF0/CK* to *UFF1/D* to be segmented and creates an internal startpoint and an internal endpoint at *UAND2/Z* and *UAND6/A* respectively. A timing report would now show this new path explicitly. Note that two additional timing paths are also created automatically, one from *UFF0/CK* to *UAND2/Z* and another from *UAND6/A* to *UFF1/D*. Thus the original timing path has been broken up into three segments, each of which is timed separately.

The *set_disable_timing*, *set_max_delay* and *set_min_delay* commands also cause timing paths to get segmented.

❑

CHAPTER

8

Timing Verification

This chapter describes the checks that are performed as part of static timing analysis. These checks are intended to exhaustively verify the timing of the design under analysis.

The two primary checks are the setup and hold checks. Once a clock is defined at the clock pin of a flip-flop, setup and hold checks are automatically inferred for the flip-flop. The timing checks are generally performed at multiple conditions including the worst-case slow condition and best-case fast condition. Typically, the worst-case slow condition is critical for setup checks and best-case fast condition is critical for hold checks - though the hold checks may be performed at the worst-case slow condition also.

The examples presented in this chapter assume that the net delays are zero; this is done for simplicity and does not alter the concepts presented herein.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_8, © Springer Science + Business Media, LLC 2009

8.1 Setup Timing Check

A **setup timing check** verifies the timing relationship between the clock and the data pin of a flip-flop so that the setup requirement is met. In other words, the setup check ensures that the data is available at the input of the flip-flop before it is clocked in the flip-flop. The data should be stable for a certain amount of time, namely the setup time of the flip-flop, before the active edge of the clock arrives at the flip-flop. This requirement ensures that the data is captured reliably into the flip-flop. Figure 8-1 shows the set-up requirement of a typical flip-flop. A setup check verifies the setup requirement of the flip-flop.

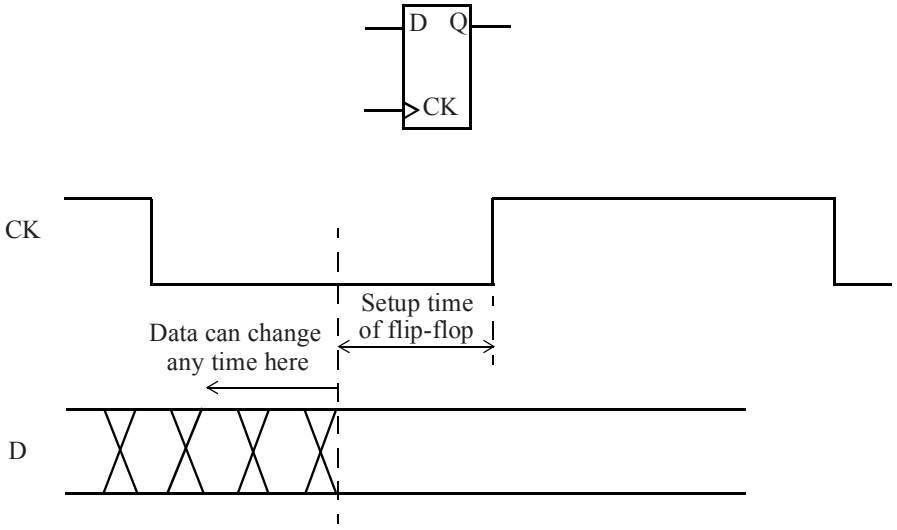


Figure 8-1 *Setup requirement of a flip-flop.*

In general, there is a launch flip-flop - the flip-flop that launches the data, and a capture flip-flop - the flip-flop that captures the data whose setup time must be satisfied. The setup check validates the long (or max) path from the launch flip-flop to the capture flip-flop. The clocks to these two flip-flops can be the same or can be different. The setup check is from the first active edge of the clock in the launch flip-flop to the closest following

active edge of the capture flip-flop. The setup check ensures that the data launched from the previous clock cycle is ready to be captured after one cycle.

We now examine a simple example, shown in Figure 8-2, where both the launch and capture flip-flops have the same clock. The first rising edge of clock *CLKM* appears at time T_{launch} at launch flip-flop. The data launched by this clock edge appears at time $T_{launch} + T_{ck2q} + T_{dp}$ at the *D* pin of the flip-flop *UFF1*. The second rising edge of the clock (setup is normally checked after one cycle) appears at time $T_{cycle} + T_{capture}$ at the clock pin of the capture flip-flop *UFF1*. The difference between these two times must be larger than the setup time of the flip-flop, so that the data can be reliably captured in the flip-flop.

The setup check can be mathematically expressed as:

$$T_{launch} + T_{ck2q} + T_{dp} < T_{capture} + T_{cycle} - T_{setup}$$

where T_{launch} is the delay of the clock tree of the launch flip-flop *UFF0*, T_{dp} is the delay of the combinational logic data path and T_{cycle} is the clock period. $T_{capture}$ is the delay of the clock tree for the capture flip-flop *UFF1*.

In other words, the total time it takes for data to arrive at the *D* pin of the capture flip-flop must be less than the time it takes for the clock to travel to the capture flip-flop plus a clock cycle delay minus the setup time.

Since the setup check poses a max[1] constraint, the setup check *always* uses the longest or the max timing path. For the same reason, this check is normally verified at the slow corner where the delays are the largest.

1. It imposes an *upper bound* on the data path delay.

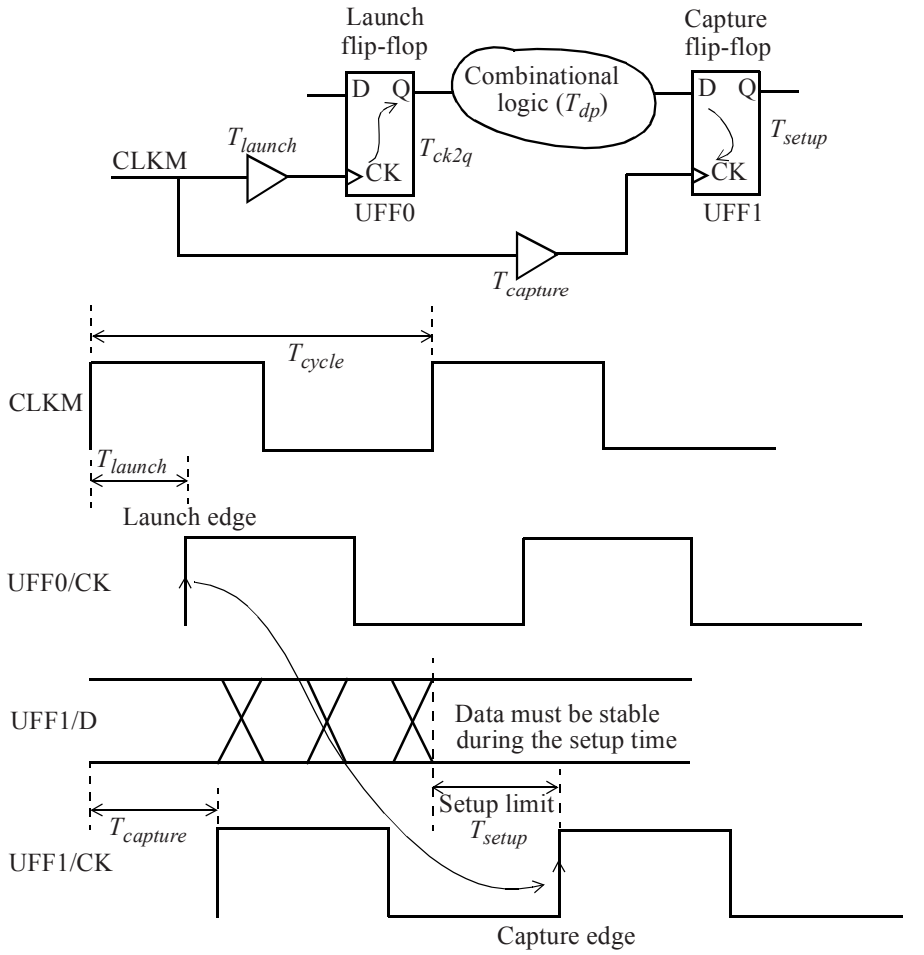


Figure 8-2 *Data and clock signals for setup timing check.*

8.1.1 Flip-flop to Flip-flop Path

Here is a path report of a setup check.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKM (rise edge)                   0.00       0.00
clock network delay (ideal)              0.00       0.00
UFF0/CK (DFF )                           0.00       0.00 r
UFF0/Q (DFF ) <-                         0.16       0.16 f
UNOR0/ZN (NR2  )                         0.04       0.20 r
UBUF4/Z (BUFF  )                         0.05       0.26 r
UFF1/D (DFF )                            0.00       0.26 r
data arrival time                                   0.26

clock CLKM (rise edge)                  10.00      10.00
clock network delay (ideal)              0.00      10.00
clock uncertainty                       -0.30       9.70
UFF1/CK (DFF )                                      9.70 r
library setup time                      -0.04       9.66
data required time                                  9.66
-----------------------------------------------------------
data required time                                  9.66
data arrival time                                  -0.26
-----------------------------------------------------------
slack (MET)                                         9.41
```

The report shows that the launch flip-flop (specified by *Startpoint*) has instance name *UFF0* and it is triggered by the rising edge of clock *CLKM*. The capture flip-flop (specified by *Endpoint*) is *UFF1* and is also triggered by the rising edge of clock *CLKM*. The *Path Group* line indicates that it belongs to the path group *CLKM*. As discussed in the previous chapter, all paths in a design are categorized into path groups based on the clock of the capture flip-flop. The *Path Type* line indicates that the delays shown in this report

are all max path delays indicating that this is a setup check. This is because setup checks correspond to the max (or longest path) delays through the logic. Note that the hold checks correspond to the min (or shortest path) delays through the logic.

The *Incr* column specifies the incremental cell or net delay for the port or pin indicated. The *Path* column shows the cumulative delay for the arrival and the data required paths. Here is the clock specification used for this example.

```
create_clock -name CLKM -period 10 -waveform {0 5} \
  [get_ports CLKM]
set_clock_uncertainty -setup 0.3 [all_clocks]
set_clock_transition -rise 0.2 [all_clocks]
set_clock_transition -fall 0.15 [all_clocks]
```

The launch path takes 0.26ns to get to the *D* pin of flip-flop *UFF1* - this is the arrival time at the input of the capture flip-flop. The capture edge (which is one cycle away since this is a setup check) is at 10ns. A clock uncertainty of 0.3ns was specified for this clock - thus, the clock period is reduced by the uncertainty margin. The clock uncertainty includes the variation in cycle time due to jitter in the clock source and any other timing margin used for analysis. The setup time of the flip-flop 0.04ns (called *library setup time*), is deducted from the total capture path yielding a required time of 9.66ns. Since the arrival time is 0.26ns, there is a positive slack of 9.41ns on this timing path. Note that the difference between the required time and arrival time may appear to be 9.40ns - however the actual value is 9.41ns which appears in the report. The anomaly exists because the report shows only two digits after the decimal whereas the internally computed and stored values have greater precision than those reported.

What is the *clock network delay* in the timing report and why is it marked as *ideal*? This line in the timing report indicates that the clock trees are treated as *ideal*, that any buffers in the clock path are assumed to have zero delay. Once the clock trees are built, the clock network can be marked as *propagated* - which causes the clock paths to show up with real delays, as shown in the next example timing report. The 0.11ns delay is the clock network de-

lay on the launch clock and the 0.12ns delay is the clock network delay on the capture flip-flop.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKM (rise edge)                   0.00       0.00
clock network delay (propagated)         0.11       0.11
UFF0/CK (DFF )                           0.00       0.11 r
UFF0/Q (DFF ) <-                         0.14       0.26 f
UNOR0/ZN (NR2  )                         0.04       0.30 r
UBUF4/Z (BUFF  )                         0.05       0.35 r
UFF1/D (DFF )                            0.00       0.35 r
data arrival time                                   0.35

clock CLKM (rise edge)                  10.00      10.00
clock network delay (propagated)         0.12      10.12
clock uncertainty                       -0.30       9.82
UFF1/CK (DFF )                                      9.82 r
library setup time                      -0.04       9.78
data required time                                  9.78
-----------------------------------------------------------
data required time                                  9.78
data arrival time                                  -0.35
-----------------------------------------------------------
slack (MET)                                         9.43
```

The timing path report can optionally include the expanded clock paths, that is, with the clock trees explicitly shown. Here is such an example.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max
```

```
Point                                          Incr       Path
-----------------------------------------------------------------
clock CLKM (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKM (in)                                      0.00       0.00 r
UCKBUF0/C (CKB  )                              0.06       0.06 r
UCKBUF1/C (CKB  )                              0.06       0.11 r
UFF0/CK (DFF )                                 0.00       0.11 r
UFF0/Q (DFF ) <-                               0.14       0.26 f
UNOR0/ZN (NR2  )                               0.04       0.30 r
UBUF4/Z (BUFF  )                               0.05       0.35 r
UFF1/D (DFF )                                  0.00       0.35 r
data arrival time                                         0.35

clock CLKM (rise edge)                        10.00      10.00
clock source latency                           0.00      10.00
CLKM (in)                                      0.00      10.00 r
UCKBUF0/C (CKB  )                              0.06      10.06 r
UCKBUF2/C (CKB  )                              0.07      10.12 r
UFF1/CK (DFF )                                 0.00      10.12 r
clock uncertainty                             -0.30       9.82
library setup time                            -0.04       9.78
data required time                                        9.78
-----------------------------------------------------------------
data required time                                        9.78
data arrival time                                        -0.35
-----------------------------------------------------------------
slack (MET)                                               9.43
```

Notice that the clock buffers, *UCKBUF0*, *UCKBUF1* and *UCKBUF2* appear in the path report above and provide details of how the clock tree delays are computed.

How is the delay of the first clock cell *UCKBUF0* computed? As described in previous chapters, the cell delay is calculated based on the input transition time and the output capacitance of the cell. Thus, the question is what transition time is used at the input of the first cell in the clock tree. The transition time (or slew) on the input pin of the first clock cell can be explicitly specified using the *set_input_transition* command.

```
set_input_transition -rise 0.3 [get_ports CLKM]
set_input_transition -fall 0.45 [get_ports CLKM]
```

In the *set_input_transition* specification shown above, we specified the input rise transition time to be 0.3ns and the fall transition time to be 0.45ns. In the absence of the input transition specifications, ideal slew is assumed at the origin of the clock tree, which implies that both the rise and fall transition times are 0ns.

The "r" and "f" characters in the timing report indicate the rising (and falling) edge of the clock or data signal. The previous path report shows the path starting from the falling edge of *UFF0/Q* and ending on the rising edge of *UFF1/D*. Since *UFF1/D* can be either 0 or 1, there can be a path ending at the falling edge of *UFF1/D* as well. Here is such a path.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------------
clock CLKM (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
CLKM (in)                                0.00       0.00 r
UCKBUF0/C (CKB  )                        0.06       0.06 r
UCKBUF1/C (CKB  )                        0.06       0.11 r
UFF0/CK (DFF )                           0.00       0.11 r
UFF0/Q (DFF ) <-                         0.14       0.26 r
UNOR0/ZN (NR2  )                         0.02       0.28 f
UBUF4/Z (BUFF  )                         0.06       0.33 f
UFF1/D (DFF )                            0.00       0.33 f
data arrival time                                   0.33

clock CLKM (rise edge)                  10.00      10.00
clock source latency                     0.00      10.00
CLKM (in)                                0.00      10.00 r
UCKBUF0/C (CKB  )                        0.06      10.06 r
UCKBUF2/C (CKB  )                        0.07      10.12 r
UFF1/CK (DFF )                           0.00      10.12 r
```

```
clock uncertainty                          -0.30       9.82
library setup time                         -0.03       9.79
data required time                                     9.79
-----------------------------------------------------------
data required time                                     9.79
data arrival time                                     -0.33
-----------------------------------------------------------
slack (MET)                                            9.46
```

Note that the edge at the clock pin of the flip-flop (called the **active edge**) remains unchanged. It can only be a rising or falling active edge, depending upon whether the flip-flop is rising-edge triggered or falling-edge triggered respectively.

What is *clock source latency*? This is also called *insertion delay* and is the time it takes for a clock to propagate from its source to the clock definition point of the design under analysis as depicted in Figure 8-3. This corresponds to the latency of the clock tree that is outside of the design. For example, if this design were part of a larger block, the clock source latency specifies the delay of the clock tree up to the clock pin of the design under analysis. This latency can be explicitly specified using the *set_clock_latency* command.

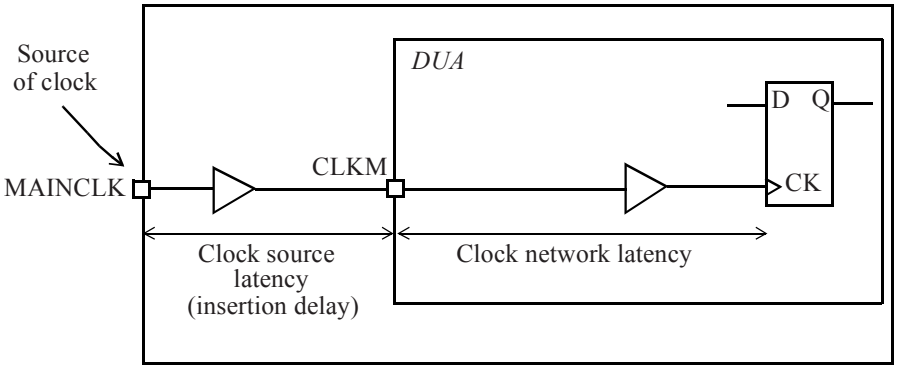


Figure 8-3 *The two types of clock latencies.*

```
set_clock_latency -source -rise 0.7 [get_clocks CLKM]
set_clock_latency -source -fall 0.65 [get_clocks CLKM]
```

In the absence of such a command, a latency of 0 is assumed. That was the assumption used in earlier path reports. Note that the source latency does not affect paths that are internal to the design and have the same launch clock and capture clock. This is because the same latency gets added to both the launch clock path and the capture clock path. However this latency does impact timing paths that go through the inputs and outputs of the design under analysis.

Without the *-source* option, the *set_clock_latency* command defines the *clock network latency* - this is the latency from the clock definition point in the *DUA* to the clock pin of a flip-flop. The clock network latency is used to model the delay through the clock path before the clock trees are built, that is, prior to clock tree synthesis. Once a clock tree is built and is marked as *propagated*, this clock network latency specification is ignored. The *set_clock_latency* command can be used to model the delay from the master clock to one of its generated clocks as described in Section 7.3. This command is also used to model off-chip clock latency when clock generation logic is not part of the design.

8.1.2 Input to Flip-flop Path

Here is an example path report through an input port to a flip-flop. Figure 8-4 shows the schematic related to the input path and the clock waveforms.

```
Startpoint: INA (input port clocked by VIRTUAL_CLKM)
Endpoint: UFF2 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock VIRTUAL_CLKM (rise edge)           0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                     2.55       2.55 f
```

```
INA (in) <-                              0.00       2.55 f
UINV1/ZN (INV  )                         0.02       2.58 r
UAND0/Z (AN2  )                          0.06       2.63 r
UINV2/ZN (INV  )                         0.02       2.65 f
UFF2/D (DFF )                            0.00       2.65 f
data arrival time                                   2.65

clock CLKM (rise edge)                  10.00      10.00
clock source latency                     0.00      10.00
CLKM (in)                                0.00      10.00 r
UCKBUF0/C (CKB  )                        0.06      10.06 r
UCKBUF2/C (CKB  )                        0.07      10.12 r
UCKBUF3/C (CKB  )                        0.06      10.18 r
UFF2/CK (DFF )                           0.00      10.18 r
clock uncertainty                       -0.30       9.88
library setup time                      -0.03       9.85
data required time                                  9.85
-----------------------------------------------------------
data required time                                  9.85
data arrival time                                  -2.65
-----------------------------------------------------------
slack (MET)                                         7.20
```

The first thing to notice is *input port clocked by VIRTUAL_CLKM*. As discussed in Section 7.9, this clock can be considered as an imaginary (virtual) flip-flop outside of the design that is driving the input port *INA* of the design. The clock of this virtual flip-flop is *VIRTUAL_CLKM*. In addition, the max delay from the clock pin of this virtual flip-flop to the input port *INA* is specified as 2.55ns - this appears as *input external delay* in the report. Both of these parameters are specified using the following SDC commands.

```
create_clock -name VIRTUAL_CLKM -period 10 -waveform {0 5}
set_input_delay -clock VIRTUAL_CLKM \
  -max 2.55 [get_ports INA]
```

Notice that the definition of the virtual clock *VIRTUAL_CLKM* does not have any pin from the design associated with it; this is because it is considered to be defined outside of the design (it is *virtual*). The input delay spec-

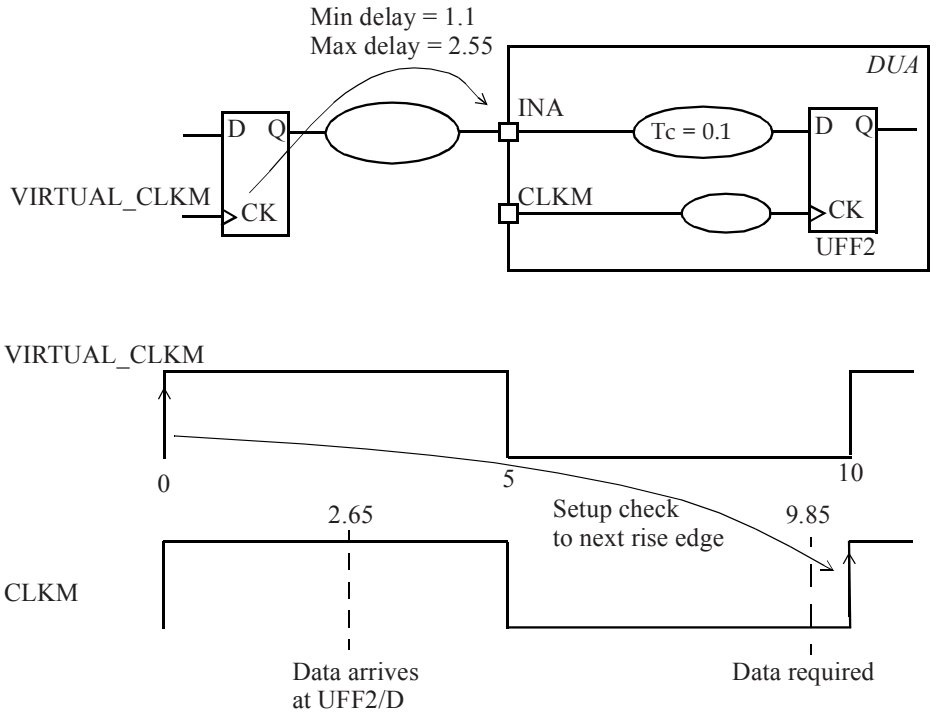


Figure 8-4 *Setup check for the path through input port.*

ification, *set_input_delay*, specifies the delay with respect to the virtual clock.

The input path starts from the port *INA*; how does one compute the delay of the first cell *UINV1* connected to port *INA*? One way to accomplish this is by specifying the driving cell of the input port *INA*. This driving cell is used to determine the drive strength and thus the slew on the port *INA*, which is then used to compute the delay of the cell *UINV1*. In the absence of any slew specification on the input port *INA*, the transition at the port is assumed to be ideal, which corresponds to a transition time of 0ns.

```
set_driving_cell -lib_cell BUFF \
 -library lib013lwc [get_ports INA]
```

Figure 8-4 also shows how the setup check is done. The time by which data must arrive at *UFF2/D* is 9.85ns. However, the data arrives at 2.65ns, thus the report shows a positive slack of 7.2ns on this path.

Input Path with Actual Clock

Input arrival times can be specified with respect to an actual clock also; these do not necessarily have to be specified with respect to a virtual clock. Examples of actual clocks are clocks on internal pins in the design, or on input ports. Figure 8-5 depicts an example where the input constraint on port *CIN* is specified relative to a clock on input port *CLKP*. This constraint is specified as:

```
set_input_delay -clock CLKP -max 4.3 [get_ports CIN]
```

Here is the input path report corresponding to this specification.

```
Startpoint: CIN (input port clocked by CLKP)
Endpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
Path Group: CLKP
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKP (rise edge)                   0.00       0.00
clock network delay (propagated)         0.00       0.00
input external delay                     4.30       4.30 f
CIN (in)                                 0.00       4.30 f
UBUF5/Z (BUFF  )                         0.06       4.36 f
UXOR1/Z (XOR2  )                         0.10       4.46 r
UFF4/D (DFF )                            0.00       4.46 r
data arrival time                                   4.46

clock CLKP (rise edge)                  12.00      12.00
clock source latency                     0.00      12.00
```

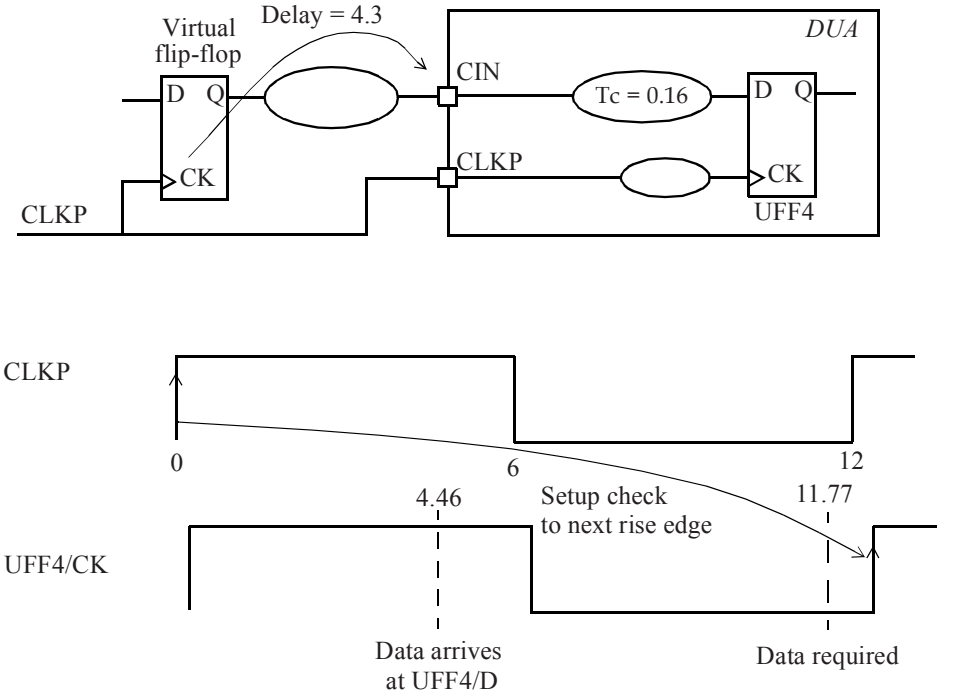


Figure 8-5 *Path through input port using core clock.*

```
CLKP (in)                                    0.00      12.00 r
UCKBUF4/C (CKB  )                            0.06      12.06 r
UCKBUF5/C (CKB  )                            0.06      12.12 r
UFF4/CK (DFF )                               0.00      12.12 r
clock uncertainty                           -0.30      11.82
library setup time                          -0.05      11.77
data required time                                     11.77
-------------------------------------------------------------
data required time                                     11.77
data arrival time                                      -4.46
-------------------------------------------------------------
slack (MET)                                             7.31
```

Notice that the *Startpoint* specifies the reference clock for the input port to be *CLKP* as expected.

8.1.3 Flip-flop to Output Path

Similar to the input port constraint described above, an output port can be constrained either with respect to a virtual clock, or an internal clock of the design, or an input clock port, or an output clock port. Here is an example that shows the output pin *ROUT* constrained with respect to a virtual clock. The output constraint is as follows:

```
set_output_delay -clock VIRTUAL_CLKP \
  -max 5.1 [get_ports ROUT]
set_load 0.02 [get_ports ROUT]
```

To determine the delay of the last cell connected to the output port correctly, one needs to specify the load on this port. The output load is specified above using the *set_load* command. Note that the port *ROUT* may have load contribution internal to the *DUA* and the *set_load* specification provides the additional load, which is the load contribution from outside the DUA. In the absence of the *set_load* specification, a value of 0 for the external load is assumed (which may not be realistic as this design would most probably be used in some other design). Figure 8-6 shows the timing path to the virtual flip-flop that has the virtual clock.

The path report through the output port is shown next.

```
Startpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: ROUT (output port clocked by VIRTUAL_CLKP)
  Path Group: VIRTUAL_CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
```

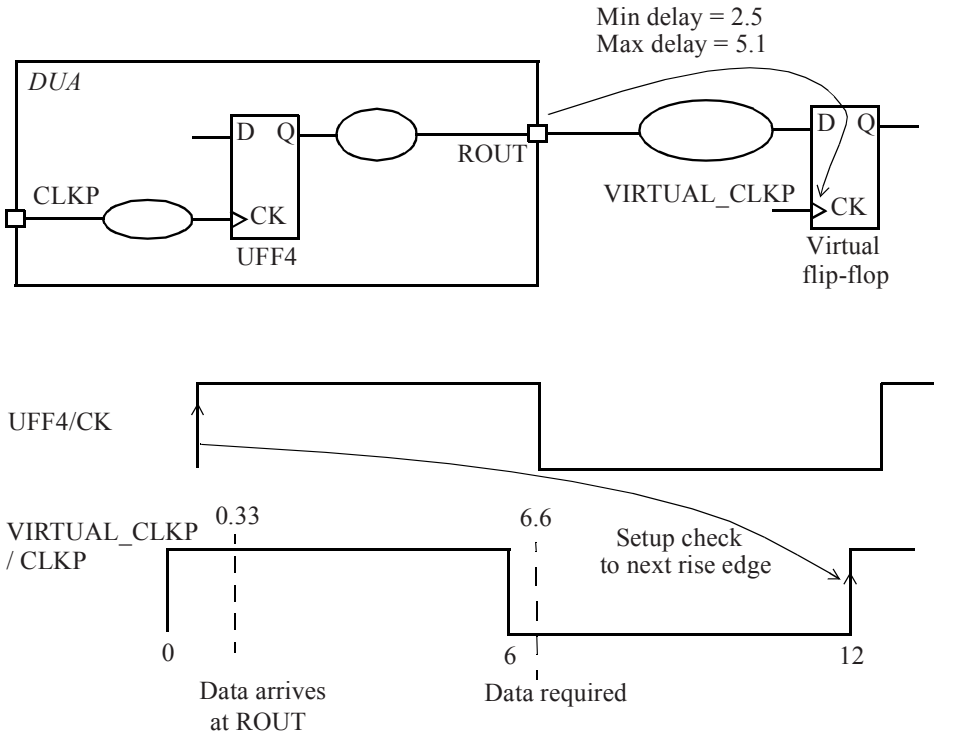


Figure 8-6 *Setup check for path through output port.*

```
UCKBUF4/C (CKB  )                        0.06       0.06 r
UCKBUF5/C (CKB  )                        0.06       0.12 r
UFF4/CK (DFF )                           0.00       0.12 r
UFF4/Q (DFF )                            0.13       0.25 r
UBUF3/Z (BUFF  )                         0.09       0.33 r
ROUT (out)                               0.00       0.33 r
data arrival time                                   0.33

clock VIRTUAL_CLKP (rise edge)          12.00      12.00
clock network delay (ideal)              0.00      12.00
clock uncertainty                       -0.30      11.70
output external delay                   -5.10       6.60
data required time                                  6.60
```

```
-------------------------------------------------------------
data required time                                    6.60
data arrival time                                    -0.33
-------------------------------------------------------------
slack (MET)                                           6.27
```

Notice that the output delay specified appears as *output external delay* and behaves like a required setup time for the virtual flip-flop.

8.1.4 Input to Output Path

The design can have a combinational path going from an input port to an output port. This path can be constrained and timed just like the input and output paths we saw earlier. Figure 8-7 shows an example of such a path. Virtual clocks are used to specify constraints on both input and output ports.

Here are the input and output delay specifications.

```
set_input_delay -clock VIRTUAL_CLKM \
  -max 3.6 [get_ports INB]
set_output_delay -clock VIRTUAL_CLKM \
  -max 5.8 [get_ports POUT]
```

Here is a path report that goes through the combinational logic from input *INB* to output *POUT*. Notice that any internal clock latencies, if present, have no effect on the path report.

```
Startpoint: INB (input port clocked by VIRTUAL_CLKM)
Endpoint: POUT (output port clocked by VIRTUAL_CLKM)
Path Group: VIRTUAL_CLKM
Path Type: max

Point                                    Incr       Path
-------------------------------------------------------------
clock VIRTUAL_CLKM (rise edge)           0.00       0.00
clock network delay (ideal)              0.00       0.00
```

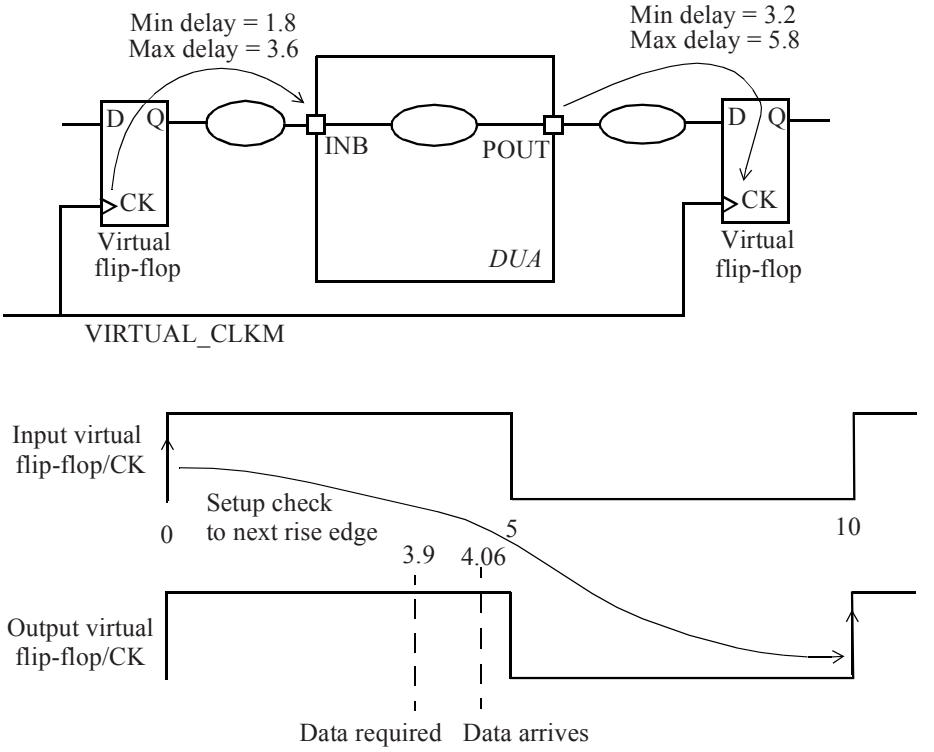


Figure 8-7 *Combinational path from input to output port.*

```
input external delay                    3.60       3.60 f
INB (in) <-                             0.00       3.60 f
UBUF0/Z (BUFF  )                        0.05       3.65 f
UBUF1/Z (BUFF  )                        0.06       3.72 f
UINV3/ZN (INV  )                        0.34       4.06 r
POUT (out)                              0.00       4.06 r
data arrival time                                  4.06

clock VIRTUAL_CLKM (rise edge)         10.00      10.00
clock network delay (ideal)             0.00      10.00
```

```
clock uncertainty                          -0.30        9.70
output external delay                      -5.80        3.90
data required time                                      3.90
-----------------------------------------------------------
data required time                                      3.90
data arrival time                                      -4.06
-----------------------------------------------------------
slack (VIOLATED)                                       -0.16
```

8.1.5 Frequency Histogram

If one were to plot a frequency histogram of setup slack versus number of paths for a typical design, it would look like the one shown in Figure 8-8. Depending upon the state of the design, whether it has been optimized or not, the zero slack line would be more towards the right for an unoptimized design and more towards the left for an optimized design. For a design that has zero violations, that is no paths with negative slack, the entire curve would be to the right of the zero slack line.

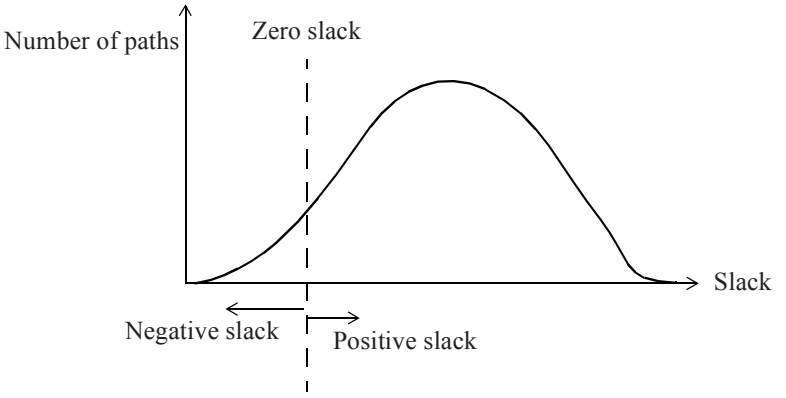


Figure 8-8 *Frequency histogram of timing slack in paths.*

Here is a histogram shown in a textual form that can often be produced by a static timing analysis tool.

```
{-INF   375   0}
{375    380   237}
{380    385   425}
{385    390   1557}
{390    395   1668}
{395    400   1559}
{400    405   1244}
{405    410   1079}
{410    415   941}
{415    420   431}
{420    425   404}
{425    430   1}
{430    +INF  0}
```

The first two indices denote the slack range and the third index is the number of paths within that slack range, for example, there are 941 paths with slack in the range of 410ps to 415ps. The histogram indicates that this design has no failing paths, that is all paths have positive slack, and that the most critical path has a positive slack between 375ps and 380ps.

Designs that are tough to meet timing would have their hump of the histogram more towards the left, that is, have many paths with slack closer to zero. One other observation that can be made by looking at a frequency histogram is on the ability to further optimize the design to achieve zero slack, that is, how difficult it is to close timing. If the number of failing paths is small and the negative slack is also small, the design is relatively close to meeting the required timing. However, if the number of failing paths is large and the negative slack magnitude is also large, this implies that the design would require a lot of effort to meet the required timing.

8.2 Hold Timing Check

A **hold timing check** ensures that a flip-flop output value that is changing does not pass through to a capture flip-flop and overwrite its output before the flip-flop has had a chance to capture its original value. This check is based on the hold requirement of a flip-flop. The hold specification of a flip-flop requires that the data being latched should be held stable for a specified amount of time after the active edge of the clock. Figure 8-9 shows the hold requirement of a typical flip-flop.

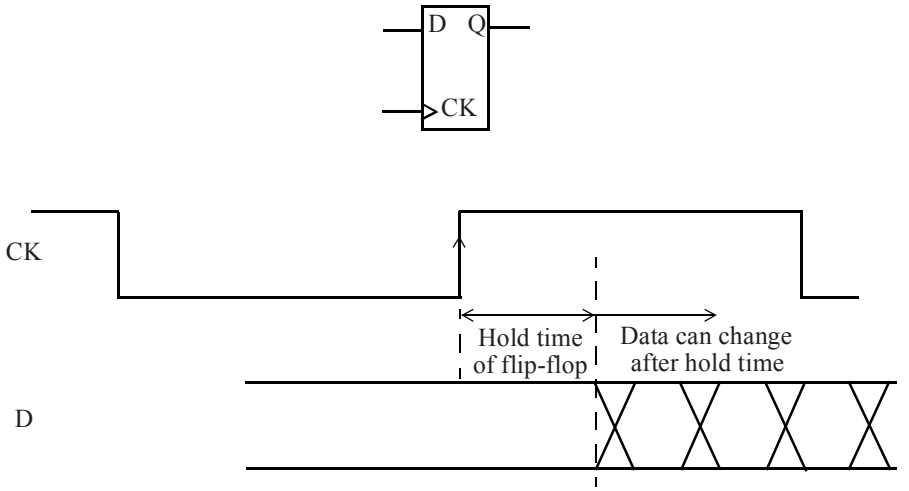


Figure 8-9 *Hold requirement of a flip-flop.*

Just like the setup check, a hold timing check is between the launch flip-flop - the flip-flop that launches the data, and the capture flip-flop - the flip-flop that captures the data and whose hold time must be satisfied. The clocks to these two flip-flops can be the same or can be different. The hold check is from one active edge of the clock in the launch flip-flop to the same clock edge at the capture flip-flop. Thus, a hold check is independent of the clock period. The hold check is carried out on each active edge of the clock of the capture flip-flop.

We now examine a simple example, shown in Figure 8-10, where both the launch and the capture flip-flops have the same clock.

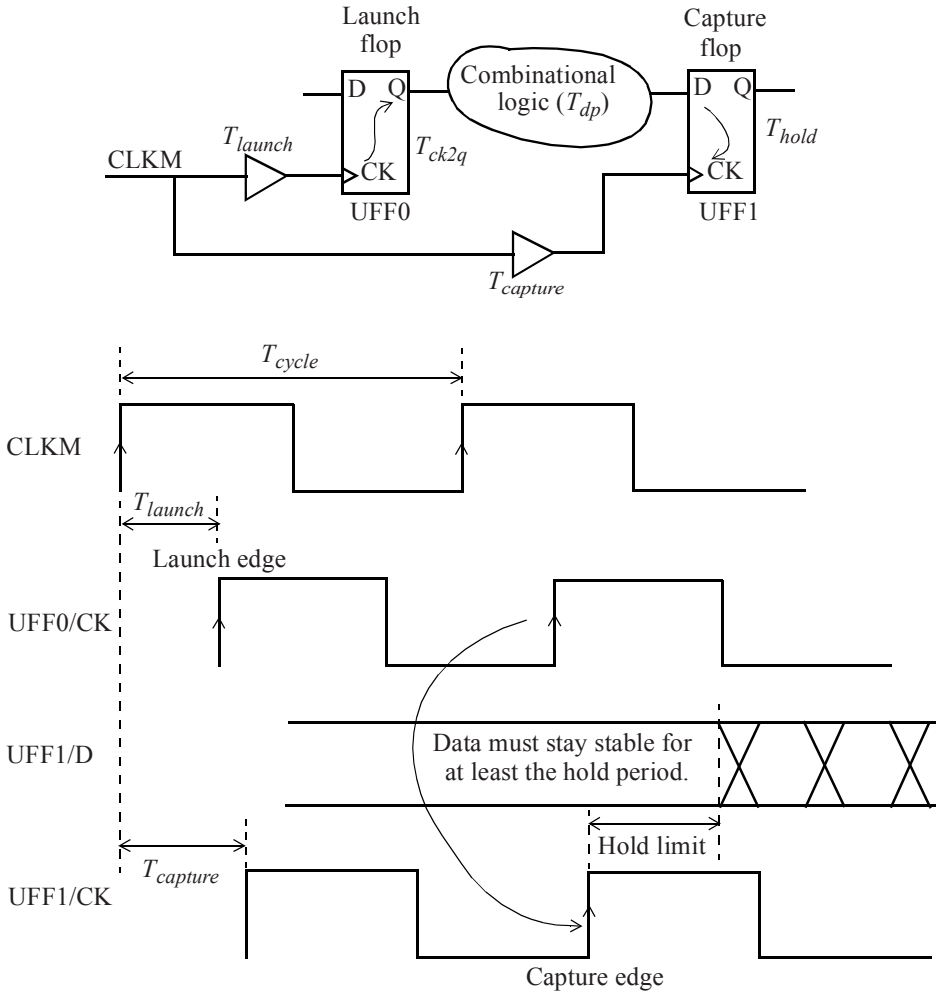


Figure 8-10 *Data and clock signals for hold timing check.*

Consider the second rising edge of clock *CLKM*. The data launched by the rising edge of the clock takes $T_{launch} + T_{ck2q} + T_{dp}$ time to get to the *D* pin of the capture flip-flop *UFF1*. The same edge of the clock takes $T_{capture}$ time to get to the clock pin of the capture flip-flop. The intention is for the data from the launch flip-flop to be captured by the capture flip-flop in the next clock cycle. If the data is captured in the same clock cycle, the intended data in the capture flip-flop (from the previous clock cycle) is overwritten. The hold time check is to ensure that the intended data in the capture flip-flop is not overwritten. The hold time check verifies that the difference between these two times (data arrival time and clock arrival time at capture flip-flop) must be larger than the hold time of the capture flip-flop, so that the previous data on the flip-flop is not overwritten and the data is reliably captured in the flip-flop.

The hold check can be mathematically expressed as:

$$T_{launch} + T_{ck2q} + T_{dp} > T_{capture} + T_{hold}$$

where T_{launch} is the delay of the clock tree of the launch flip-flop, T_{dp} is the delay in the combinational logic data path and $T_{capture}$ is the delay on the clock tree for the capture flip-flop. In other words, the total time required for data launched by a clock edge to arrive at the *D* pin of the capture flip-flop must be larger than the time required for the same edge of the clock to travel to the capture flip-flop plus the hold time. This ensures that *UFF1/D* remains stable until the hold time of the flip-flop after the rising edge of the clock on its clock pin *UFF1/CK*.

The hold checks impose a lower bound or min constraint for paths to the data pin on the capture flip-flop; the fastest path to the *D* pin of the capture flip-flop needs to be determined. This implies that the hold checks are *always* verified using the shortest paths. Thus, the hold checks are typically performed at the fast timing corner.

Even when there is only one clock in the design, the clock tree can result in the arrival times of the clocks at the launch and capture flip-flops to be substantially different. To ensure reliable data capture, the clock edge at the

capture flip-flop must arrive before the data can change. A hold timing check ensures that (see Figure 8-11):

- Data from the subsequent launch edge must not be captured by the setup receiving edge.
- Data from the setup launch edge must not be captured by the preceding receiving edge.

These two hold checks are essentially the same if both the launch and capture clock belong to the same clock domain. However, when the launch and capture clocks are at different frequencies or in different clock domains, the above two conditions may map into different constraints. In such cases, the worst hold check is the one that is reported. Figure 8-11 shows these two checks pictorially.

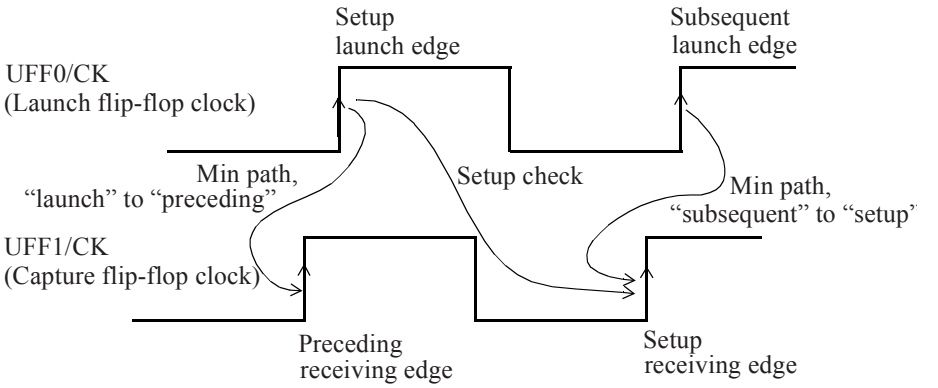


Figure 8-11 *Two hold checks for one setup check.*

UFF0 is the launch flip-flop and *UFF1* is the capture flip-flop. The setup check is between the *setup launch edge* and the *setup receiving edge*. The *subsequent launch edge* must not propagate data so fast that the *setup receiving edge* does not have time to capture its data reliably. In addition, the *setup launch edge* must not propagate data so fast that the *preceding receiving edge* does not get a chance to capture its data. The worst hold check corresponds

to the most restrictive hold check amongst various scenarios described above.

More general clocking such as for multicycle paths and multi-frequency paths are discussed later in Section 8.3 and Section 8.8 respectively. The discussion covers the relationships between setup checks and hold checks, especially, how hold checks are derived from the setup check relationships. While setup violations can cause the operating frequency of the design to be lowered, the hold violations can kill a design, that is, make the design inoperable at any frequency. Thus it is very important to understand the hold timing checks and resolve any violations.

8.2.1 Flip-flop to Flip-flop Path

This section illustrates the flip-flop to flip-flop hold path based upon the example depicted in Figure 8-2. Here is the path report of a hold timing check for the example from Section 8.1 for the setup check path.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UNOR0/ZN (NR2  )                         0.02       0.28 f
  UBUF4/Z (BUFF  )                         0.06       0.33 f
  UFF1/D (DFF )                            0.00       0.33 f
  data arrival time                                   0.33

  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
```

```
CLKM (in)                                0.00       0.00 r
UCKBUF0/C (CKB  )                        0.06       0.06 r
UCKBUF2/C (CKB  )                        0.07       0.12 r
UFF1/CK (DFF )                           0.00       0.12 r
clock uncertainty                        0.05       0.17
library hold time                        0.01       0.19
data required time                                  0.19
-----------------------------------------------------------
data required time                                  0.19
data arrival time                                  -0.33
-----------------------------------------------------------
slack (MET)                                         0.14
```

Notice that the *Path Type* is described as *min* indicating that the cell delay values along the shortest path are used which corresponds to hold timing checks. The *library hold time* specifies the hold time of flip-flop *UFF1*. (The hold time for flip-flops can also be negative as explained earlier in Section 3.4.) Notice that both the capture and receive timing is computed at the rising edge (active edge of flip-flop) of clock *CLKM*. The timing report shows that the earliest time the new data can arrive at *UFF1* while enabling the previous data to be safely captured is 0.19ns. Since the new data arrives at 0.33ns, the report shows a positive hold slack of 0.14ns.

Figure 8-12 shows the times of the clock signals at the launch and capture flip-flops along with the earliest allowed and actual arrival time for the data at the capture flip-flop. Since the data arrives later than the data required time (which for hold is the earliest allowed), the hold condition is met.

Hold Slack Calculation

An interesting point to note is the difference in the way the slack is computed for setup and hold timing reports. In the setup timing reports, the arrival time and the required time are computed and the slack is computed to be the required time minus arrival time. However in the hold timing reports, when we compute required time minus arrival time, a negative re-

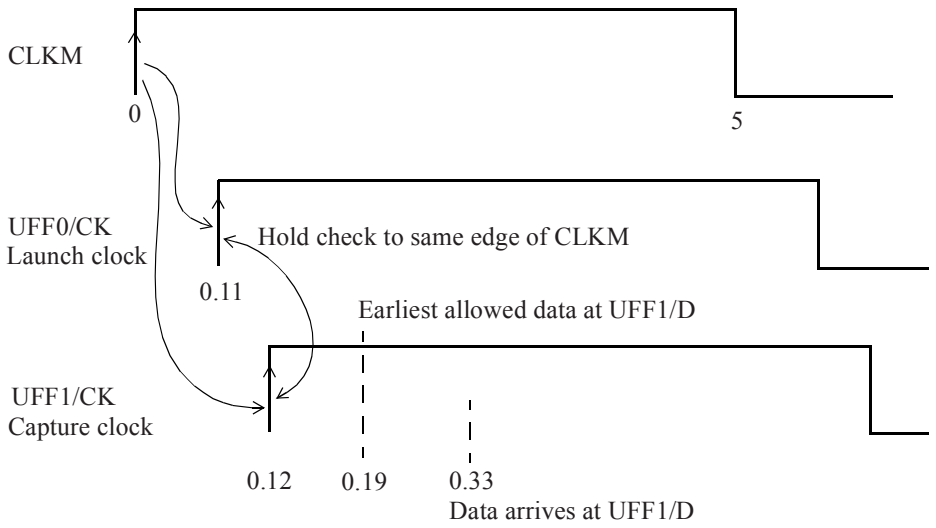


Figure 8-12 *Clock waveforms for hold timing check on internal path.*

sult translates into a positive slack (means hold constraint is satisfied), while a positive result translates into a negative slack (means hold constraint is not satisfied).

8.2.2 Input to Flip-flop Path

The hold timing check from an input port is described next. See Figure 8-4 for an example. The min delay on the input port is specified using a virtual clock as:

```
set_input_delay -clock VIRTUAL_CLKM \
  -min 1.1 [get_ports INA]
```

Here is the hold timing report.

```
Startpoint: INA (input port clocked by VIRTUAL_CLKM)
Endpoint: UFF2 (rising edge-triggered flip-flop clocked by CLKM)
```

```
Path Group: CLKM
Path Type: min

Point                                    Incr       Path
-----------------------------------------------------------
clock VIRTUAL_CLKM (rise edge)           0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                     1.10       1.10 f
INA (in) <-                              0.00       1.10 f
UINV1/ZN (INV  )                         0.02       1.13 r
UAND0/Z (AN2  )                          0.06       1.18 r
UINV2/ZN (INV  )                         0.02       1.20 f
UFF2/D (DFF )                            0.00       1.20 f
data arrival time                                   1.20

clock CLKM (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
CLKM (in)                                0.00       0.00 r
UCKBUF0/C (CKB  )                        0.06       0.06 r
UCKBUF2/C (CKB  )                        0.07       0.12 r
UCKBUF3/C (CKB  )                        0.06       0.18 r
UFF2/CK (DFF )                           0.00       0.18 r
clock uncertainty                        0.05       0.23
library hold time                        0.01       0.25
data required time                                  0.25
-----------------------------------------------------------
data required time                                  0.25
data arrival time                                  -1.20
-----------------------------------------------------------
slack (MET)                                         0.95
```

The *set_input_delay* appears as *input external delay*. The hold check is done at time 0 between rising edge of *VIRTUAL_CLKM* and rising edge of *CLKM*. The required arrival time for data to be captured by *UFF2* without violating its hold time is 0.25ns - this indicates that the data should arrive after 0.25ns. Since the data can arrive only at 1.2ns, this shows a positive slack of 0.95ns.

8.2.3 Flip-flop to Output Path

Here is a hold timing check at an output port. See Figure 8-6 for the example. The output port specification appears as:

```
set_output_delay -clock VIRTUAL_CLKP \
  -min 2.5 [get_ports ROUT]
```

The output delay is specified with respect to a virtual clock. Here is the hold report.

```
Startpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: ROUT (output port clocked by VIRTUAL_CLKP)
  Path Group: VIRTUAL_CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.06       0.06 r
  UCKBUF5/C (CKB  )                        0.06       0.12 r
  UFF4/CK (DFF )                           0.00       0.12 r
  UFF4/Q (DFF )                            0.13       0.25 f
  UBUF3/Z (BUFF  )                         0.08       0.33 f
  ROUT (out)                               0.00       0.33 f
  data arrival time                                   0.33

  clock VIRTUAL_CLKP (rise edge)           0.00       0.00
  clock network delay (ideal)              0.00       0.00
  clock uncertainty                        0.05       0.05
  output external delay                   -2.50      -2.45
  data required time                                 -2.45
  -----------------------------------------------------------
  data required time                                 -2.45
  data arrival time                                  -0.33
  -----------------------------------------------------------
  slack (MET)                                         2.78
```

Notice that the *set_output_delay* appears as *output external delay*.

Flip-flop to Output Path with Actual Clock

Here is a path report of a hold timing check to an output port. See Figure 8-13. The output min delay is specified with respect to a real clock.

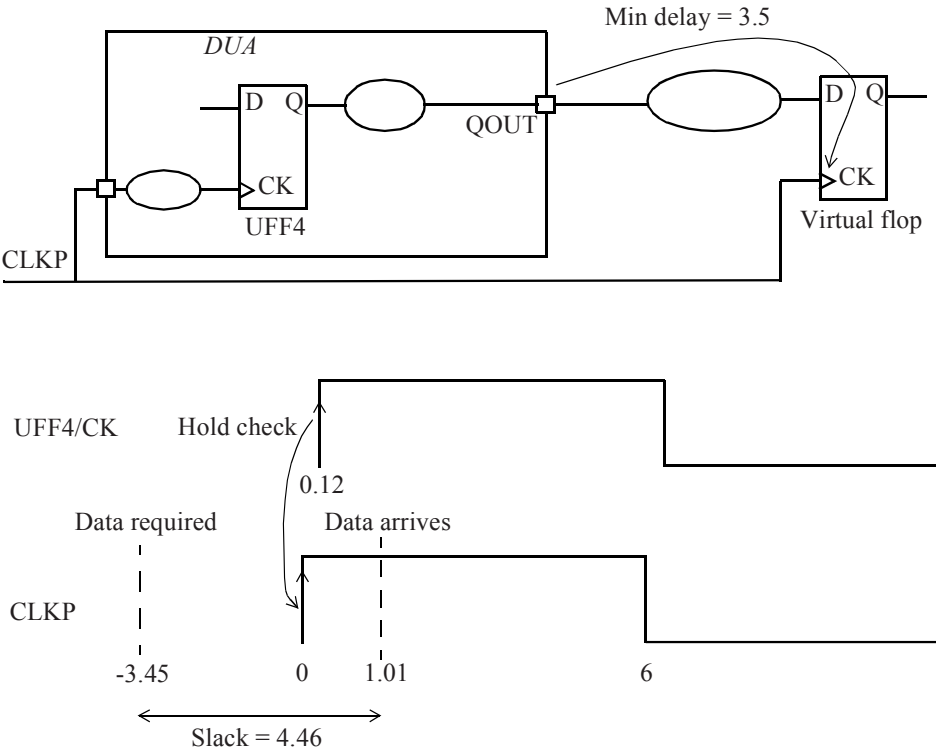


Figure 8-13 *Path through output port.*

```
set_output_delay -clock CLKP -min 3.5 [get_ports QOUT]
set_load 0.55 [get_ports QOUT]
```

Here is the hold timing report.

```
Startpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: QOUT (output port clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.06       0.06 r
  UCKBUF5/C (CKB  )                        0.06       0.12 r
  UFF4/CK (DFF )                           0.00       0.12 r
  UFF4/Q (DFF )                            0.14       0.26 r
  UINV4/ZN (INV  )                         0.75       1.01 f
  QOUT (out)                               0.00       1.01 f
  data arrival time                                   1.01

  clock CLKP (rise edge)                   0.00       0.00
  clock network delay (propagated)         0.00       0.00
  clock uncertainty                        0.05       0.05
  output external delay                   -3.50      -3.45
  data required time                                 -3.45
  -----------------------------------------------------------
  data required time                                 -3.45
  data arrival time                                  -1.01
  -----------------------------------------------------------
  slack (MET)                                         4.46
```

The hold timing check is performed on the rising edge (active edge of flip-flop) of clock *CLKP*. The above report indicates that the flip-flop to output has a positive slack of 4.46ns for the hold time.

8.2.4 Input to Output Path

Here is a hold timing check on an input to output path shown in Figure 8-7. The specifications on the ports are:

```
set_load -pin_load 0.15 [get_ports POUT]
set_output_delay -clock VIRTUAL_CLKM \
  -min 3.2 [get_ports POUT]
set_input_delay -clock VIRTUAL_CLKM \
  -min 1.8 [get_ports INB]
set_input_transition 0.8 [get_ports INB]
```

```
Startpoint: INB (input port clocked by VIRTUAL_CLKM)
Endpoint: POUT (output port clocked by VIRTUAL_CLKM)
Path Group: VIRTUAL_CLKM
Path Type: min

Point                                          Incr       Path
-----------------------------------------------------------------
clock VIRTUAL_CLKM (rise edge)                 0.00       0.00
clock network delay (ideal)                    0.00       0.00
input external delay                           1.80       1.80 r
INB (in) <-                                    0.00       1.80 r
UBUF0/Z (BUFF  )                               0.04       1.84 r
UBUF1/Z (BUFF  )                               0.06       1.90 r
UINV3/ZN (INV  )                               0.22       2.12 f
POUT (out)                                     0.00       2.12 f
data arrival time                                         2.12

clock VIRTUAL_CLKM (rise edge)                 0.00       0.00
clock network delay (ideal)                    0.00       0.00
clock uncertainty                              0.05       0.05
output external delay                         -3.20      -3.15
data required time                                       -3.15
-----------------------------------------------------------------
data required time                                       -3.15
data arrival time                                        -2.12
-----------------------------------------------------------------
slack (MET)                                               5.27
```

The specification on the output port is specified with respect to a virtual clock and hence the hold check is performed on the rising (default active) edge of that virtual clock.

8.3 Multicycle Paths

In some cases, the combinational data path between two flip-flops can take more than one clock cycle to propagate through the logic. In such cases, the combinational path is declared as a **multicycle path**. Even though the data is being captured by the capture flip-flop on every clock edge, we direct STA that the relevant capture edge occurs after the specified number of clock cycles.

Figure 8-14 shows an example. Since the data path can take up to three clock cycles, a setup multicycle check of three cycles should be specified. The multicycle setup constraints specified to achieve this are given below.

```
create_clock -name CLKM -period 10 [get_ports CLKM]
set_multicycle_path 3 -setup \
  -from [get_pins UFF0/Q] \
  -to [get_pins UFF1/D]
```

The setup multicycle constraint specifies that the path from *UFF0/CK* to *UFF1/D* can take up to three clock cycles to complete for a setup check. This implies that the design utilizes the required data from *UFF1/Q* only every third cycle instead of every cycle.

Here is a setup path report that has the multicycle constraint specified.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max
```

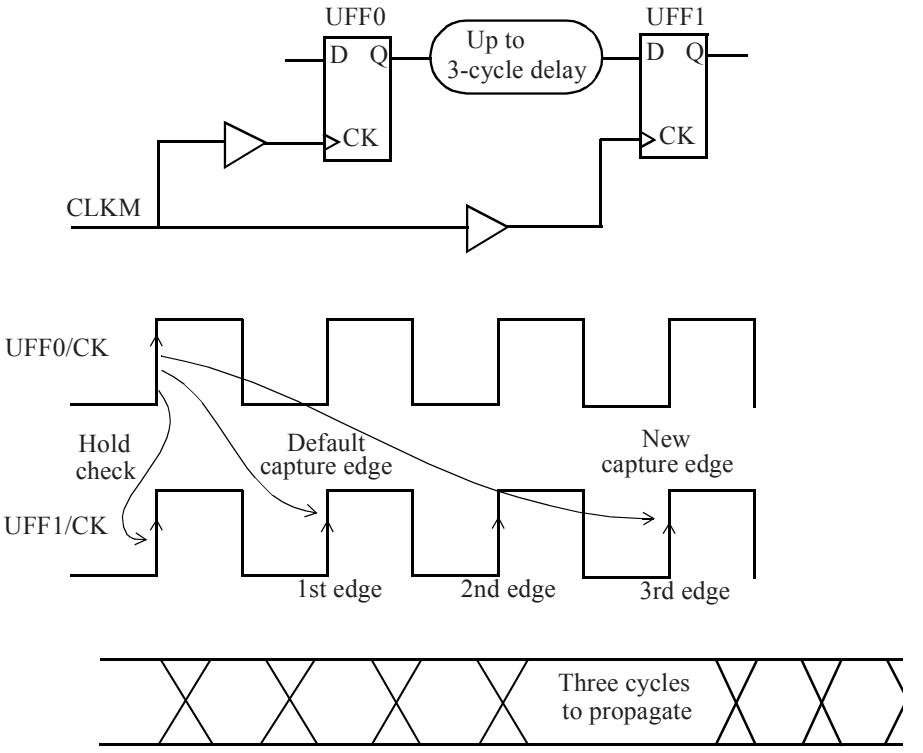


Figure 8-14 *A three-cycle multicycle path.*

```
Point                                          Incr       Path
-----------------------------------------------------------------
clock CLKM (rise edge)                         0.00       0.00
clock network delay (propagated)               0.11       0.11
UFF0/CK (DFF )                                 0.00       0.11 r
UFF0/Q (DFF ) <-                               0.14       0.26 f
UNOR0/ZN (NR2  )                               0.04       0.30 r
UBUF4/Z (BUFF  )                               0.05       0.35 r
UFF1/D (DFF )                                  0.00       0.35 r
data arrival time                                         0.35
```

```
clock CLKM (rise edge)                    30.00      30.00
clock network delay (propagated)           0.12      30.12
clock uncertainty                         -0.30      29.82
UFF1/CK (DFF )                                       29.82 r
library setup time                        -0.04      29.78
data required time                                   29.78
-------------------------------------------------------------
data required time                                   29.78
data arrival time                                    -0.35
-------------------------------------------------------------
slack (MET)                                          29.43
```

Notice that the clock edge for the capture flip-flop is now three clock cycles away, at 30ns.

We now examine the hold timing check on the example multicycle path. In most common scenarios, we would want the hold check to stay as it was in a single cycle setup case, which is the one shown in the Figure 8-14. This ensures that the data is free to change anywhere in between the three clock cycles. A hold multicycle of two is specified to get the same behavior of a hold check as in a single cycle setup case. This is because in the absence of such a hold multicycle specification, the default hold check is done on the active edge prior to the setup capture edge which is not the intent. We need to move the hold check two cycles prior to the default hold check edge and hence a hold multicycle of two is specified. The intended behavior is shown in Figure 8-15. With the multicycle hold, the min delay for the data path can be less than one clock cycle.

```
set_multicycle_path 2 -hold -from [get_pins UFF0/Q] \
  -to [get_pins UFF1/D]
```

The number of cycles denoted on a multicycle hold specifies how many clock cycles to move back from its default hold check edge (which is one active edge prior to the setup capture edge). Here is the path report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKP)
```

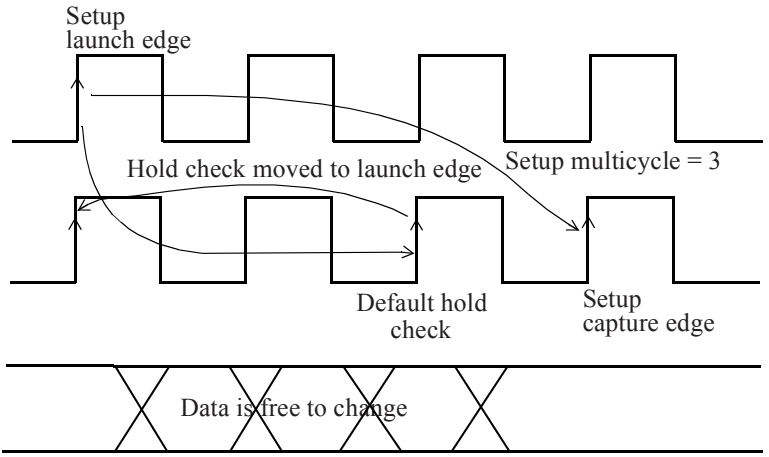


Figure 8-15 *Hold check moved back to launch edge.*

```
Path Group: CLKP
Path Type: min

Point                                          Incr       Path
-----------------------------------------------------------------
clock CLKP (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKP (in)                                      0.00       0.00 r
UCKBUF4/C (CKB  )                              0.07       0.07 r
UFF0/CK (DFF )                                 0.00       0.07 r
UFF0/Q (DFF ) <-                               0.15       0.22 f
UXOR1/Z (XOR2  )                               0.07       0.29 f
UFF1/D (DFF )                                  0.00       0.29 f
data arrival time                                         0.29

clock CLKP (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKP (in)                                      0.00       0.00 r
UCKBUF4/C (CKB  )                              0.07       0.07 r
UCKBUF5/C (CKB  )                              0.06       0.13 r
UFF1/CK (DFF )                                 0.00       0.13 r
clock uncertainty                              0.05       0.18
library hold time                              0.01       0.19
```

```
  data required time                                      0.19
  ------------------------------------------------------------
  data required time                                      0.19
  data arrival time                                      -0.29
  ------------------------------------------------------------
  slack (MET)                                             0.11
```

Since this path has a setup multicycle of three, its default hold check is on the active edge prior to the capture edge. In most designs, if the max path (or setup) requires *N* clock cycles, it is not feasible to achieve the min path constraint to be greater than (*N*-1) clock cycles. By specifying a multicycle hold of two cycles, the hold check edge is moved back to the launch edge (at 0ns) as shown in the above path report.

Thus, in most designs, a multicycle setup specified as *N* (cycles) should be accompanied by a multicycle hold constraint specified as *N-1* (cycles).

What happens when a multicycle setup of *N* is specified but the corresponding *N-1* multicycle hold is missing? In such a case, the hold check is performed on the edge one cycle prior to the setup capture edge. The case of such a hold check on a multicycle setup of 3 is shown in Figure 8-16.

This imposes a restriction that data can only change in the one cycle before the setup capture edge as the figure shows. Thus the data path must have a min delay of at least two clock cycles to meet this requirement. Here is such a path report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
```

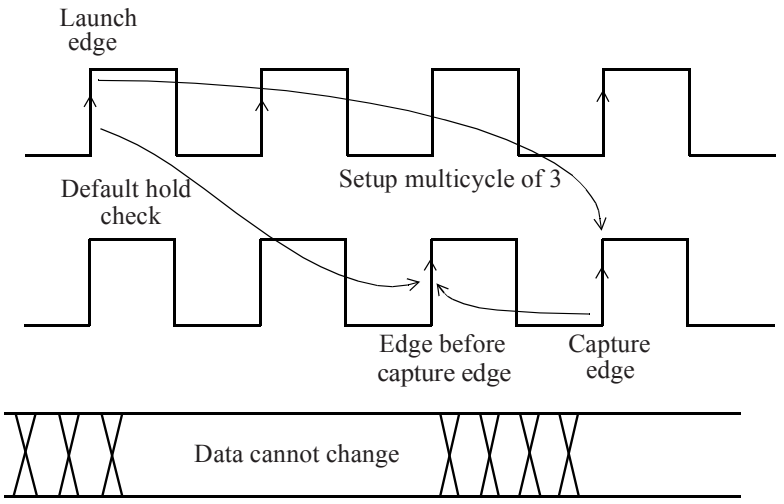


Figure 8-16 *Default hold check in a multicycle path.*

```
UCKBUF1/C (CKB  )                          0.06       0.11 r
UFF0/CK (DFF )                             0.00       0.11 r
UFF0/Q (DFF ) <-                           0.14       0.26 r
UNOR0/ZN (NR2  )                           0.02       0.28 f
UBUF4/Z (BUFF  )                           0.06       0.33 f
UFF1/D (DFF )                              0.00       0.33 f
data arrival time                                     0.33

clock CLKM (rise edge)                    20.00      20.00
clock source latency                       0.00      20.00
CLKM (in)                                  0.00      20.00 r
UCKBUF0/C (CKB  )                          0.06      20.06 r
UCKBUF2/C (CKB  )                          0.07      20.12 r
UFF1/CK (DFF )                             0.00      20.12 r
clock uncertainty                          0.05      20.17
library hold time                          0.01      20.19
data required time                                   20.19
-----------------------------------------------------------
data required time                                   20.19
data arrival time                                    -0.33
-----------------------------------------------------------
```

```
slack (VIOLATED)                                    -19.85
```

Notice from the path report that the hold is being checked one clock edge prior to the capture edge, leading to a large hold violation. In effect, the hold check is requiring a minimum delay in the combinational logic of at least two clock cycles.

Crossing Clock Domains

Let us consider the case when there is a multicycle between two different clocks of the same period. (The case where the clock periods are different is described later in this chapter.)

Example I:

```
create_clock -name CLKM \
  -period 10 -waveform {0 5} [get_ports CLKM]
create_clock -name CLKP \
  -period 10 -waveform {0 5} [get_ports CLKP]
```

The setup multicycle multiplier represents the number of clock cycles for a given path. This is illustrated in Figure 8-17. The default setup capture edge is always one cycle away. A setup multicycle of 2 puts the capture edge two clock cycles away from the launch edge.

The hold multicycle multiplier represents the number of clock cycles before the setup capture edge that the hold check will occur, regardless of the launch edge. This is illustrated in Figure 8-18. The default hold check is one cycle before the setup capture edge. A hold multicycle specification of 1 puts the hold check one cycle prior to the default hold check, and thus becomes two cycles prior to the capture edge.

```
set_multicycle_path 2 \
  -from [get_pins UFF0/CK] -to [get_pins UFF3/D]
# Since no -hold option is specified, the default option
```

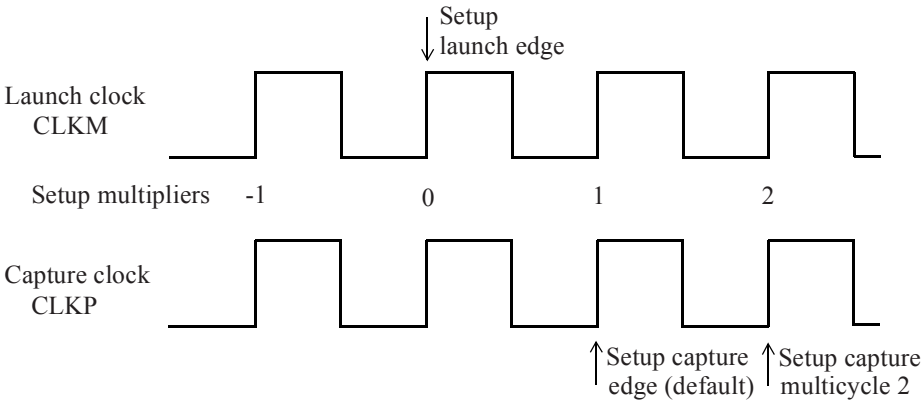


Figure 8-17 *Capture clock edges for various setup multicycle multiplier settings.*

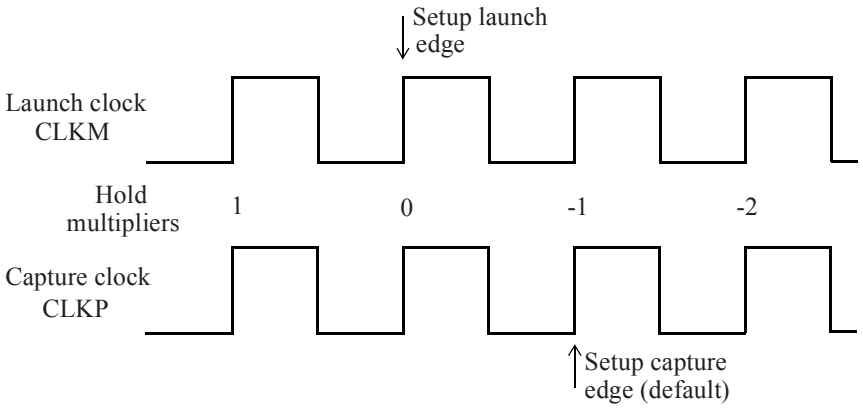


Figure 8-18 *Capture clock edges for various hold multicycle multiplier settings.*

```
   # -setup is assumed. This implies that the setup
   # multiplier is 2 and the hold multiplier is 0.
```

Here is the setup path report corresponding to the multicycle specification.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 f
  UINV0/ZN (INV  )                         0.03       0.28 r
  UFF3/D (DFF )                            0.00       0.28 r
  data arrival time                                   0.28

  clock CLKP (rise edge)                  20.00      20.00
  clock source latency                     0.00      20.00
  CLKP (in)                                0.00      20.00 r
  UCKBUF4/C (CKB  )                        0.06      20.06 r
  UFF3/CK (DFF )                           0.00      20.06 r
  clock uncertainty                       -0.30      19.76
  library setup time                      -0.04      19.71
  data required time                                 19.71
  -----------------------------------------------------------
  data required time                                 19.71
  data arrival time                                  -0.28
  -----------------------------------------------------------
  slack (MET)                                        19.43
```

Note that the path group specified in a path report is always that of the capture flip-flop, in this case, *CLKP*.

Next is the hold timing check path report. The hold multiplier defaults to 0 and thus the hold check is carried out at 10ns, one clock cycle prior to the capture edge.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UINV0/ZN (INV  )                         0.02       0.28 f
  UFF3/D (DFF )                            0.00       0.28 f
  data arrival time                                   0.28

  clock CLKP (rise edge)                  10.00      10.00
  clock source latency                     0.00      10.00
  CLKP (in)                                0.00      10.00 r
  UCKBUF4/C (CKB  )                        0.06      10.06 r
  UFF3/CK (DFF )                           0.00      10.06 r
  clock uncertainty                        0.05      10.11
  library hold time                        0.02      10.12
  data required time                                 10.12
  -----------------------------------------------------------
  data required time                                 10.12
  data arrival time                                  -0.28
  -----------------------------------------------------------
  slack (VIOLATED)                                   -9.85
```

The above report shows the hold violation which can be removed by setting a multicycle hold of 1. This is illustrated in a separate example next.

Example II:

Another example of a multicycle specified across two different clock domains is given below.

```
set_multicycle_path 2 \
  -from [get_pins UFF0/CK] -to [get_pins UFF3/D] -setup
set_multicycle_path 1 \
  -from [get_pins UFF0/CK] -to [get_pins UFF3/D] -hold
# The -setup and -hold options are explicitly specified.
```

Here is the setup path timing report for the multicycle setup of 2.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 f
  UNAND0/ZN (ND2  )                        0.03       0.29 r
  UFF3/D (DFF )                            0.00       0.29 r
  data arrival time                                   0.29

  clock CLKP (rise edge)                  20.00      20.00
  clock source latency                     0.00      20.00
  CLKP (in)                                0.00      20.00 r
  UCKBUF4/C (CKB  )                        0.07      20.07 r
  UFF3/CK (DFF )                           0.00      20.07 r
  clock uncertainty                       -0.30      19.77
  library setup time                      -0.04      19.72
  data required time                                 19.72
  -----------------------------------------------------------
```

```
  data required time                                    19.72
  data arrival time                                     -0.29
  ------------------------------------------------------------
  slack (MET)                                           19.44
```

Here is the hold check timing path report for the multicycle hold of 1.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UNAND0/ZN (ND2  )                        0.03       0.29 f
  UFF3/D (DFF )                            0.00       0.29 f
  data arrival time                                   0.29

  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UFF3/CK (DFF )                           0.00       0.07 r
  clock uncertainty                        0.05       0.12
  library hold time                        0.02       0.13
  data required time                                  0.13
  ------------------------------------------------------------
  data required time                                  0.13
  data arrival time                                  -0.29
  ------------------------------------------------------------
  slack (MET)                                         0.16
```

Note that the example reports for the setup and hold checks in this section are for the same timing corner. However, the setup checks are generally hardest to meet (have the lowest slack) at the worst-case slow corner whereas the hold checks are generally hardest to meet (have the lowest slack) at the best-case fast corner.

8.4 False Paths

It is possible that certain timing paths are not real (or not possible) in the actual functional operation of the design. Such paths can be turned off during STA by setting these as false paths. A false path is ignored by the STA for analysis.

Examples of false paths could be from one clock domain to another clock domain, from a clock pin of a flip-flop to the input of another flip-flop, through a pin of a cell, through pins of multiple cells, or a combination of these. When a false path is specified through a pin of a cell, all paths that go through that pin are ignored for timing analysis. The advantage of identifying the false paths is that the analysis space is reduced, thereby allowing the analysis to focus only on the real paths. This helps cut down the analysis time as well. However, too many false paths which are wildcarded using the *through* specification can slow down the analysis.

A false path is set using the **set_false_path** specification. Here are some examples.

```
set_false_path -from [get_clocks SCAN_CLK] \
  -to [get_clocks CORE_CLK]
# Any path starting from the SCAN_CLK domain to the
# CORE_CLK domain is a false path.

set_false_path -through [get_pins UMUX0/S]
# Any path going through this pin is false.
```

```
set_false_path \
  -through [get_pins SAD_CORE/RSTN]]
# The false path specifications can also be specified to,
# through, or from a module pin instance.

set_false_path -to [get_ports TEST_REG*]
# All paths that end in port named TEST_REG* are false paths.

set_false_path -through UINV/Z -through UAND0/Z
# Any path that goes through both of these pins
# in this order is false.
```

Few recommendations on setting false paths are given below. To set a false path between two clock domains, use:

```
set_false_path -from [get_clocks clockA] \
  -to [get_clocks clockB]
```

instead of:

```
set_false_path -from [get_pins {regA_*}/CP] \
  -to [get_pins {regB_*}/D]
```

The second form is much slower.

Another recommendation is to minimize the usage of *-through* options, as it adds unnecessary runtime complexity. The *-through* option should only be used where it is absolutely necessary and there is no alternate way to specify the false path.

From an optimization perspective, another guideline is to not use a false path when a multicycle path is the real intent. If a signal is sampled at a known or predictable time, no matter how far out, a multicycle path specification should be used so that the path has some constraint and gets optimized to meet the multicycle constraint. If a false path is used on a path

that is sampled many clock cycles later, optimization of the remaining logic may invariably slow this path even beyond what may be necessary.

8.5 Half-Cycle Paths

If a design has both negative-edge triggered flip-flops (active clock edge is falling edge) and positive-edge triggered flip-flops (active clock edge is rising edge), it is likely that half-cycle paths exist in the design. A half-cycle path could be from a rising edge flip-flop to a falling edge flip-flop, or vice versa. Figure 8-19 shows an example when the launch is on the falling edge of the clock of flip-flop *UFF5*, and the capture is on the rising edge of the clock of flip-flop *UFF3*.

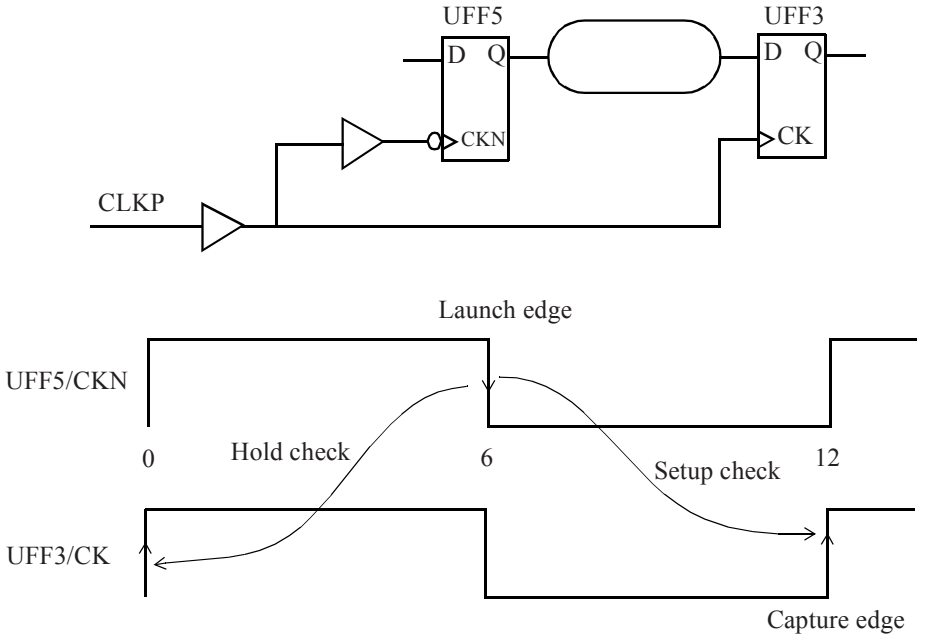


Figure 8-19 *A half-cycle path.*

Here is the setup timing check path report.

```
Startpoint: UFF5(falling edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (fall edge)                   6.00       6.00
  clock source latency                     0.00       6.00
  CLKP (in)                                0.00       6.00 f
  UCKBUF4/C (CKB  )                        0.06       6.06 f
  UCKBUF6/C (CKB  )                        0.06       6.12 f
  UFF5/CKN (DFN  )                         0.00       6.12 f
  UFF5/Q (DFN  ) <-                        0.16       6.28 r
  UNAND0/ZN (ND2  )                        0.03       6.31 f
  UFF3/D (DFF )                            0.00       6.31 f
  data arrival time                                   6.31

  clock CLKP (rise edge)                  12.00      12.00
  clock source latency                     0.00      12.00
  CLKP (in)                                0.00      12.00 r
  UCKBUF4/C (CKB  )                        0.07      12.07 r
  UFF3/CK (DFF )                           0.00      12.07 r
  clock uncertainty                       -0.30      11.77
  library setup time                      -0.03      11.74
  data required time                                 11.74
  -----------------------------------------------------------
  data required time                                 11.74
  data arrival time                                  -6.31
  -----------------------------------------------------------
  slack (MET)                                         5.43
```

Note the edge specification in the *Startpoint* and *Endpoint*. The falling edge occurs at 6ns and the rising edge occurs at 12ns. Thus, the data gets only a half-cycle, which is 6ns, to propagate to the capture flip-flop.

While the data path gets only half-cycle for setup check, an extra half-cycle is available for the hold timing check. Here is the hold timing path.

```
Startpoint: UFF5(falling edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (fall edge)                   6.00       6.00
  clock source latency                     0.00       6.00
  CLKP (in)                                0.00       6.00 f
  UCKBUF4/C (CKB  )                        0.06       6.06 f
  UCKBUF6/C (CKB  )                        0.06       6.12 f
  UFF5/CKN (DFN  )                         0.00       6.12 f
  UFF5/Q (DFN  ) <-                        0.16       6.28 r
  UNAND0/ZN (ND2  )                        0.03       6.31 f
  UFF3/D (DFF )                            0.00       6.31 f
  data arrival time                                   6.31

  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UFF3/CK (DFF )                           0.00       0.07 r
  clock uncertainty                        0.05       0.12
  library hold time                        0.02       0.13
  data required time                                  0.13
  -----------------------------------------------------------
  data required time                                  0.13
  data arrival time                                  -6.31
  -----------------------------------------------------------
  slack (MET)                                         6.18
```

The hold check always occurs one cycle prior to the capture edge. Since the capture edge occurs at 12ns, the previous capture edge is at 0ns, and hence the hold gets checked at 0ns. This effectively adds a half-cycle margin for hold checking and thus results in a large positive slack on hold.

8.6 Removal Timing Check

A **removal timing check** ensures that there is adequate time between an active clock edge and the release of an asynchronous control signal. The check ensures that the active clock edge has no effect because the asynchronous control signal remains active until *removal time* after the active clock edge. In other words, the asynchronous control signal is released (becomes inactive) well after the active clock edge so that the clock edge can have no effect. This is illustrated in Figure 8-20. This check is based on the removal time specified for the asynchronous pin of the flip-flop. Here is an excerpt from the cell library description corresponding to the removal check.

```
pin(CDN) {
  . . .
  timing() {
    related_pin : "CK";
    timing_type : removal_rising;
    . . .
  }
}
```

Like a hold check, it is a min path check except that it is on an asynchronous pin of a flip-flop.

```
Startpoint: UFF5(falling edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF6 (removal check against rising-edge clock CLKP)
  Path Group: **async_default**
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (fall edge)                   6.00       6.00
  clock source latency                     0.00       6.00
  CLKP (in)                                0.00       6.00 f
  UCKBUF4/C (CKB  )                        0.06       6.06 f
  UCKBUF6/C (CKB  )                        0.07       6.13 f
  UFF5/CKN (DFN  )                         0.00       6.13 f
```

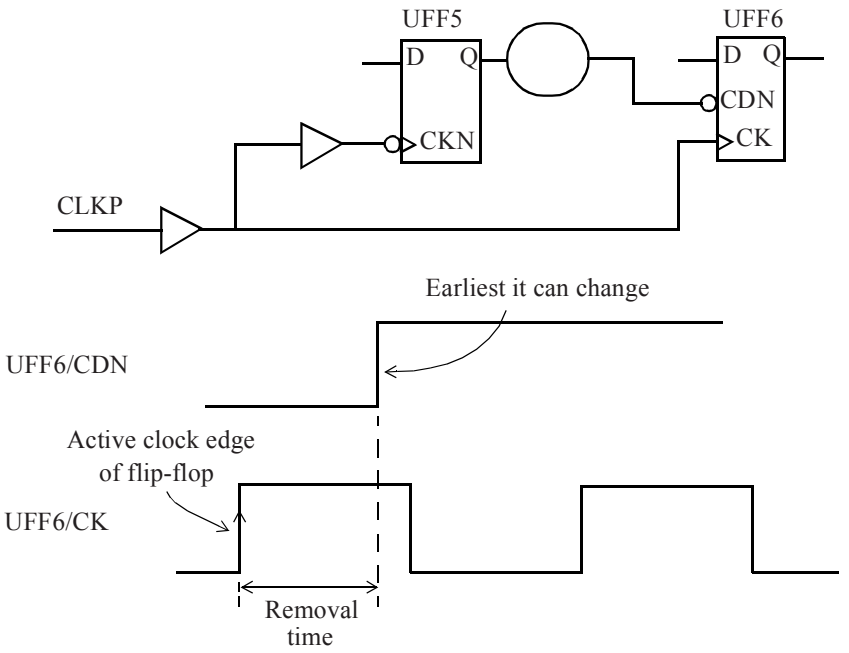


Figure 8-20 *Removal timing check.*

```
UFF5/Q (DFN  )                          0.15       6.28 f
UINV8/ZN (INV  )                        0.03       6.31 r
UFF6/CDN (DFCN  )                       0.00       6.31 r
data arrival time                                  6.31

clock CLKP (rise edge)                  0.00       0.00
clock source latency                    0.00       0.00
CLKP (in)                               0.00       0.00 r
UCKBUF4/C (CKB  )                       0.07       0.07 r
UCKBUF6/C (CKB  )                       0.07       0.14 r
UCKBUF7/C (CKB  )                       0.05       0.19 r
UFF6/CK (DFCN  )                        0.00       0.19 r
clock uncertainty                       0.05       0.24
library removal time                    0.19       0.43
data required time                                 0.43
-----------------------------------------------------------------
```

```
data required time                                    0.43
data arrival time                                    -6.31
-------------------------------------------------------------
slack (MET)                                           5.88
```

The *Endpoint* shows that it is removal check. It is on the asynchronous pin *CDN* of flip-flop *UFF6*. The removal time for this flip-flop is listed as *library removal time* with a value of 0.19ns.

All asynchronous timing checks are assigned to the *async default* path group.

8.7 Recovery Timing Check

A **recovery timing check** ensures that there is a minimum amount of time between the asynchronous signal becoming inactive and the next active clock edge. In other words, this check ensures that after the asynchronous signal becomes inactive, there is adequate time to recover so that the next active clock edge can be effective. For example, consider the time between an asynchronous reset becoming inactive and the clock active edge of a flip-flop. If the active clock edge occurs too soon after the release of reset, the state of the flip-flop may be unknown. The recovery check is illustrated in Figure 8-21. This check is based upon the recovery time specified for the asynchronous pin of the flip-flop in its cell library file, an excerpt of which is shown below.

```
pin(RSN) {
  . . .
  timing() {
    related_pin : "CK";
    timing_type : recovery_rising;
    . . .
  }
}
```

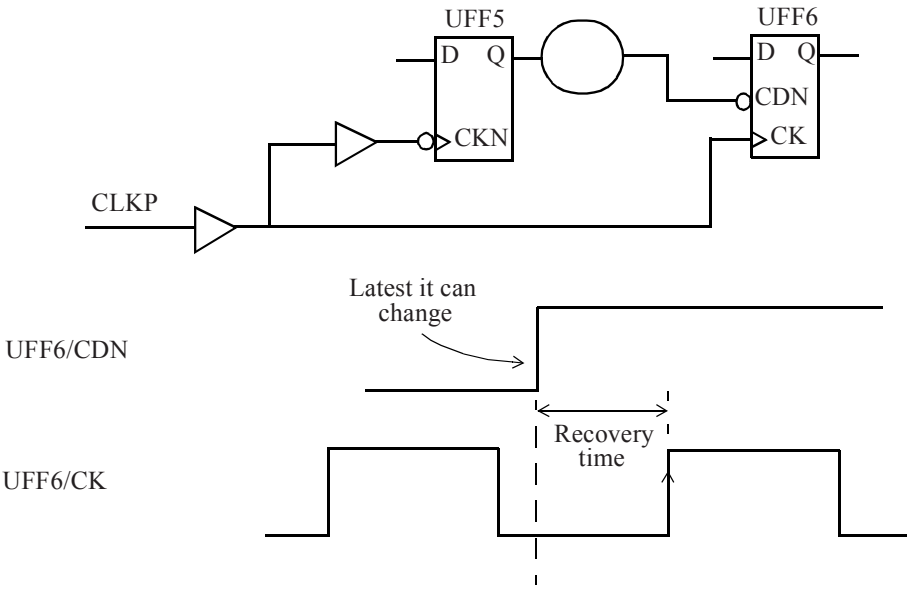


Figure 8-21 *Recovery timing check.*

Like a setup check, this is a max path check except that it is on an asynchronous signal.

Here is a recovery path report.

```
Startpoint: UFF5(falling edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF6 (recovery check against rising-edge clock CLKP)
  Path Group: **async_default**
  Path Type: max

  Point                                          Incr       Path
  ---------------------------------------------------------------
  clock CLKP (fall edge)                         6.00       6.00
  clock source latency                           0.00       6.00
  CLKP (in)                                      0.00       6.00 f
```

```
UCKBUF4/C (CKB  )                        0.06       6.06 f
UCKBUF6/C (CKB  )                        0.07       6.13 f
UFF5/CKN (DFN  )                         0.00       6.13 f
UFF5/Q (DFN  )                           0.15       6.28 f
UINV8/ZN (INV  )                         0.03       6.31 r
UFF6/CDN (DFCN  )                        0.00       6.31 r
data arrival time                                   6.31

clock CLKP (rise edge)                  12.00      12.00
clock source latency                     0.00      12.00
CLKP (in)                                0.00      12.00 r
UCKBUF4/C (CKB  )                        0.07      12.07 r
UCKBUF6/C (CKB  )                        0.07      12.14 r
UCKBUF7/C (CKB  )                        0.05      12.19 r
UFF6/CK (DFCN  )                         0.00      12.19 r
clock uncertainty                       -0.30      11.89
library recovery time                    0.09      11.98
data required time                                 11.98
-----------------------------------------------------------
data required time                                 11.98
data arrival time                                  -6.31
-----------------------------------------------------------
slack (MET)                                         5.67
```

The *Endpoint* shows that it is recovery check. The recovery time for the *UFF6* flip-flop is listed as the *library recovery time* with a value of 0.09ns. Recovery checks also belong to the *async default* path group.

8.8 Timing across Clock Domains

8.8.1 Slow to Fast Clock Domains

Let us examine the setup and hold checks when a path goes from a slower clock domain to a faster clock domain. This is shown in Figure 8-22.

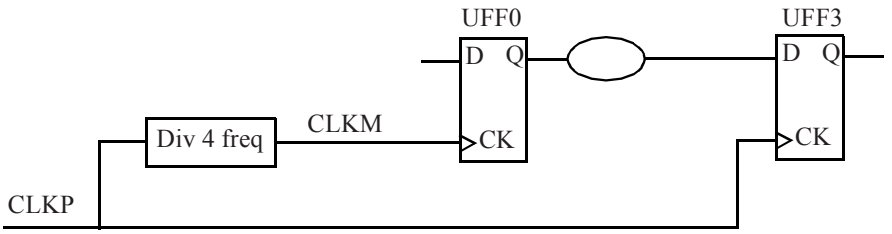


Figure 8-22 *Path from a slow clock to a faster clock.*

Here are the clock definitions for our example.

```
create_clock -name CLKM \
  -period 20 -waveform {0 10} [get_ports CLKM]
create_clock -name CLKP \
  -period 5 -waveform {0 2.5} [get_ports CLKP]
```

When the clock frequencies are different for the launch flip-flop and the capture flip-flop, STA is performed by first determining a common base period. An example of a message produced when STA is performed on such a design with the above two clocks is given below. The faster clock is expanded so that a common period is obtained.

```
Expanding clock 'CLKP' to base period of 20.00
(old period was 5.00, added 6 edges).
```

Figure 8-23 shows the setup check. By default, the most constraining setup edge relationship is used, which in this case is the very next capture edge. Here is a setup path report that shows this.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max
```

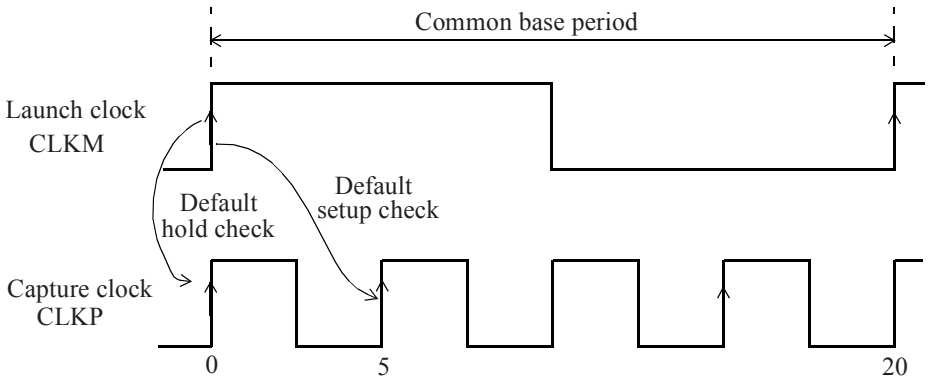


Figure 8-23 *Setup and hold checks with slow to fast path.*

```
Point                                          Incr       Path
-----------------------------------------------------------------
clock CLKM (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKM (in)                                      0.00       0.00 r
UCKBUF0/C (CKB  )                              0.06       0.06 r
UCKBUF1/C (CKB  )                              0.06       0.11 r
UFF0/CK (DFF )                                 0.00       0.11 r
UFF0/Q (DFF ) <-                               0.14       0.26 f
UNAND0/ZN (ND2  )                              0.03       0.29 r
UFF3/D (DFF )                                  0.00       0.29 r
data arrival time                                         0.29

clock CLKP (rise edge)                         5.00       5.00
clock source latency                           0.00       5.00
CLKP (in)                                      0.00       5.00 r
UCKBUF4/C (CKB  )                              0.07       5.07 r
UFF3/CK (DFF )                                 0.00       5.07 r
clock uncertainty                             -0.30       4.77
library setup time                            -0.04       4.72
data required time                                        4.72
-----------------------------------------------------------------
data required time                                        4.72
data arrival time                                        -0.29
-----------------------------------------------------------------
slack (MET)                                               4.44
```

Notice that the launch clock is at time 0ns while the capture clock is at time 5ns.

As discussed earlier, hold checks are related to the setup checks and ensure that the data launched by a clock edge does not interfere with the previous capture. Here is the hold check timing report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UNAND0/ZN (ND2  )                        0.03       0.29 f
  UFF3/D (DFF )                            0.00       0.29 f
  data arrival time                                   0.29

  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UFF3/CK (DFF )                           0.00       0.07 r
  clock uncertainty                        0.05       0.12
  library hold time                        0.02       0.13
  data required time                                  0.13
  -----------------------------------------------------------
  data required time                                  0.13
  data arrival time                                  -0.29
  -----------------------------------------------------------
  slack (MET)                                         0.16
```

In the above example, we can see that the launch data is available every fourth cycle of the capture clock. Let us assume that the intention is not to capture data on the very next active edge of *CLKP*, but to capture on every 4th capture edge. This assumption gives the combinational logic between the flip-flops four periods of *CLKP* to propagate, which is 20ns. We can do this by setting the following multicycle specification:

```
set_multicycle_path 4 -setup \
  -from [get_clocks CLKM] -to [get_clocks CLKP] -end
```

The *-end* specifies that the multicycle of 4 refers to the end point or the capture clock. This multicycle specification changes the setup and hold checks to the ones shown in Figure 8-24. Here is the setup report.

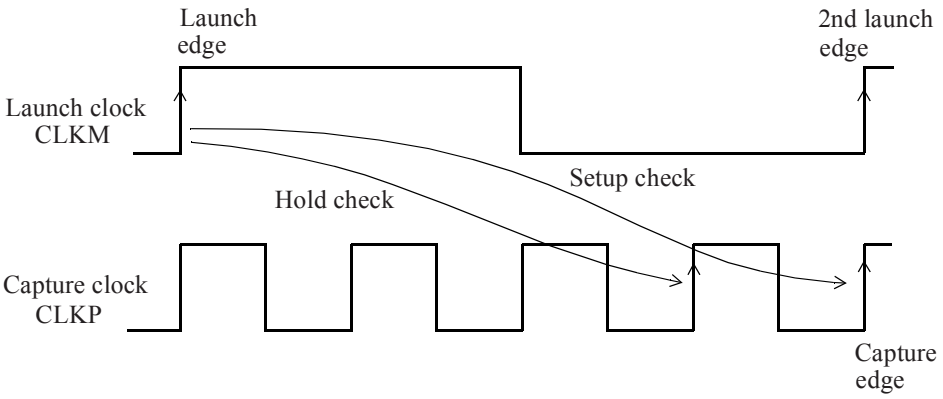


Figure 8-24 *Multicycle of 4 between clock domains.*

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  ------------------------------------------------------------
```

```
  clock CLKM (rise edge)                    0.00       0.00
  clock source latency                      0.00       0.00
  CLKM (in)                                 0.00       0.00 r
  UCKBUF0/C (CKB  )                         0.06       0.06 r
  UCKBUF1/C (CKB  )                         0.06       0.11 r
  UFF0/CK (DFF )                            0.00       0.11 r
  UFF0/Q (DFF ) <-                          0.14       0.26 f
  UNAND0/ZN (ND2  )                         0.03       0.29 r
  UFF3/D (DFF )                             0.00       0.29 r
  data arrival time                                    0.29

  clock CLKP (rise edge)                   20.00      20.00
  clock source latency                      0.00      20.00
  CLKP (in)                                 0.00      20.00 r
  UCKBUF4/C (CKB  )                         0.07      20.07 r
  UFF3/CK (DFF )                            0.00      20.07 r
  clock uncertainty                        -0.30      19.77
  library setup time                       -0.04      19.72
  data required time                                  19.72
  ------------------------------------------------------------
  data required time                                  19.72
  data arrival time                                   -0.29
  ------------------------------------------------------------
  slack (MET)                                         19.44
```

Figure 8-24 shows the hold check - note that the hold check is derived from the setup check and defaults to one cycle preceding the intended capture edge. Here is the hold timing report. Notice that the hold capture edge is at 15ns, one cycle prior to the setup capture edge.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                     Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                    0.00       0.00
  clock source latency                      0.00       0.00
  CLKM (in)                                 0.00       0.00 r
```

```
UCKBUF0/C (CKB  )                          0.06       0.06 r
UCKBUF1/C (CKB  )                          0.06       0.11 r
UFF0/CK (DFF )                             0.00       0.11 r
UFF0/Q (DFF ) <-                           0.14       0.26 r
UNAND0/ZN (ND2  )                          0.03       0.29 f
UFF3/D (DFF )                              0.00       0.29 f
data arrival time                                     0.29

clock CLKP (rise edge)                    15.00      15.00
clock source latency                       0.00      15.00
CLKP (in)                                  0.00      15.00 r
UCKBUF4/C (CKB  )                          0.07      15.07 r
UFF3/CK (DFF )                             0.00      15.07 r
clock uncertainty                          0.05      15.12
library hold time                          0.02      15.13
data required time                                   15.13
-------------------------------------------------------------
data required time                                   15.13
data arrival time                                    -0.29
-------------------------------------------------------------
slack (VIOLATED)                                    -14.84
```

In most designs, this is not the intended check, and the hold check should be moved all the way back to where the launch edge is. We do this by setting a hold multicycle specification of 3.

```
set_multicycle_path 3 -hold \
  -from [get_clocks CLKM] -to [get_clocks CLKP] -end
```

The cycle of 3 moves the hold checking edge *back* three cycles, that is, to time 0ns. The distinction with a setup multicycle is that in setup, the setup capture edge moves forward by the specified number of cycles from the default setup capture edge; in a hold multicycle, the hold check edge moves backward from the default hold check edge (one cycle before setup edge). The *-end* option implies that we want to move the endpoint (or capture edge) back by the specified number of cycles, which is that of the capture clock. Instead of *-end,* the other choice, the *-start* option, specifies the number of launch clock cycles to move by; the *-end* option specifies the

number of capture clock cycles to move by. The *-end* is the default for a multicycle setup and the *-start* is the default for multicycle hold.

With the additional multicycle hold specification, the clock edge used for hold timing checks is moved back one cycle, and the checks look like those shown in Figure 8-25. The hold report with the multicycle hold specification is as follows.

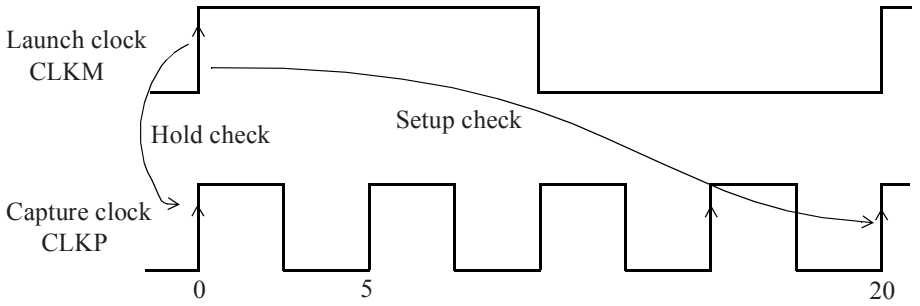


Figure 8-25 *Hold time relaxed with multicycle hold specification.*

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DF  )                           0.00       0.11 r
  UFF0/Q (DF  ) <-                         0.14       0.26 r
  UNAND0/ZN (ND2  )                        0.03       0.29 f
  UFF3/D (DF  )                            0.00       0.29 f
  data arrival time                                   0.29
```

```
clock CLKP (rise edge)                    0.00       0.00
clock source latency                      0.00       0.00
CLKP (in)                                 0.00       0.00 r
UCKBUF4/C (CKB  )                         0.07       0.07 r
UFF3/CK (DF  )                            0.00       0.07 r
clock uncertainty                         0.05       0.12
library hold time                         0.02       0.13
data required time                                   0.13
-----------------------------------------------------------
data required time                                   0.13
data arrival time                                   -0.29
-----------------------------------------------------------
slack (MET)                                          0.16
```

In summary, if a setup multicycle of *N* cycles is specified, then most likely a hold multicycle of *N*-1 cycles should also be specified. A good rule of thumb for multi-frequency multicycle path specification in the case of paths between slow to fast clock domains is to use the *-end* option. With this option, the setup and hold checks are adjusted based upon the clock cycles of the fast clock.

8.8.2 Fast to Slow Clock Domains

In this subsection, we consider examples where the data path goes from a fast clock domain to a slow clock domain. The default setup and hold checks are as shown in Figure 8-26 when the following clock definitions are used.

```
create_clock -name CLKM \
  -period 20 -waveform {0 10} [get_ports CLKM]
create_clock -name CLKP \
  -period 5 -waveform {0 2.5} [get_ports CLKP]
```

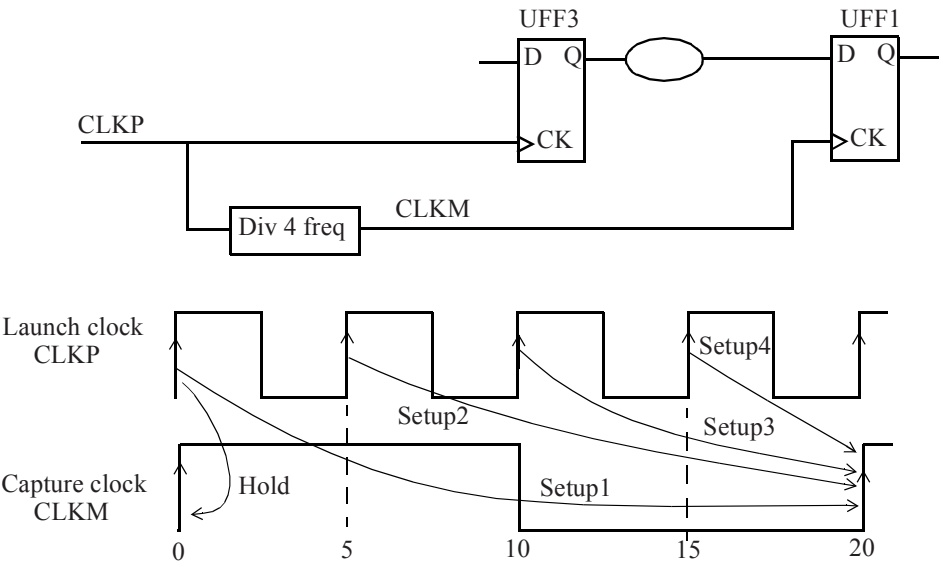


Figure 8-26 *Path from fast clock to slow clock domain.*

There are four setup timing checks possible; see *Setup1*, *Setup2*, *Setup3* and *Setup4* in the figure. However, the most restrictive one is the *Setup4* check. Here is the path report of this most restrictive path. Notice that the launch clock edge is at 15ns and the capture clock edge is at 20ns.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max

  Point                                          Incr       Path
  ---------------------------------------------------------------
  clock CLKP (rise edge)                        15.00      15.00
  clock source latency                           0.00      15.00
  CLKP (in)                                      0.00      15.00 r
  UCKBUF4/C (CKB  )                              0.07      15.07 r
  UFF3/CK (DFF )                                 0.00      15.07 r
```

```
  UFF3/Q (DFF ) <-                          0.15      15.22 f
  UNOR0/ZN (NR2  )                          0.05      15.27 r
  UBUF4/Z (BUFF  )                          0.05      15.32 r
  UFF1/D (DFF )                             0.00      15.32 r
  data arrival time                                   15.32

  clock CLKM (rise edge)                   20.00      20.00
  clock source latency                      0.00      20.00
  CLKM (in)                                 0.00      20.00 r
  UCKBUF0/C (CKB  )                         0.06      20.06 r
  UCKBUF2/C (CKB  )                         0.07      20.12 r
  UFF1/CK (DFF )                            0.00      20.12 r
  clock uncertainty                        -0.30      19.82
  library setup time                       -0.04      19.78
  data required time                                  19.78
  ------------------------------------------------------------
  data required time                                  19.78
  data arrival time                                  -15.32
  ------------------------------------------------------------
  slack (MET)                                         4.46
```

Similar to the setup checks, there are four hold checks possible. Figure 8-26 shows the most restrictive hold check which ensures that the capture edge at 0ns does not capture the data being launched at 0ns. Here is the timing report for this hold check.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min

  Point                                     Incr       Path
  ------------------------------------------------------------
  clock CLKP (rise edge)                    0.00       0.00
  clock source latency                      0.00       0.00
  CLKP (in)                                 0.00       0.00 r
  UCKBUF4/C (CKB  )                         0.07       0.07 r
  UFF3/CK (DFF )                            0.00       0.07 r
  UFF3/Q (DFF ) <-                          0.16       0.22 r
  UNOR0/ZN (NR2  )                          0.02       0.25 f
```

```
UBUF4/Z (BUFF  )                          0.06       0.30 f
UFF1/D (DFF )                             0.00       0.30 f
data arrival time                                    0.30

clock CLKM (rise edge)                    0.00       0.00
clock source latency                      0.00       0.00
CLKM (in)                                 0.00       0.00 r
UCKBUF0/C (CKB  )                         0.06       0.06 r
UCKBUF2/C (CKB  )                         0.07       0.12 r
UFF1/CK (DFF )                            0.00       0.12 r
clock uncertainty                         0.05       0.17
library hold time                         0.01       0.19
data required time                                   0.19
-------------------------------------------------------------
data required time                                   0.19
data arrival time                                   -0.30
-------------------------------------------------------------
slack (MET)                                          0.12
```

In general, a designer may specify the data path from the fast clock to the slow clock to be a multicycle path. If the setup check is relaxed to provide two cycles of the faster clock for the data path, the following is included for this multicycle specification:

```
set_multicycle_path 2 -setup \
  -from [get_clocks CLKP] -to [get_clocks CLKM] -start

set_multicycle_path 1 -hold \
  -from [get_clocks CLKP] -to [get_clocks CLKM] -start
# The -start option refers to the launch clock and is
# the default for a multicycle hold.
```

In this case, Figure 8-27 shows the clock edges used for the setup and hold checks. The *-start* option specifies that the unit for the number of cycles (2 in this case) is that of the launch clock (*CLKP* in this case). The setup multicycle of 2 moves the launch edge one edge prior to the default launch edge, that is, at 10ns instead of the default 15ns. The hold multicycle ensures that

the capture of the earlier data can reliably occur at 0ns due to the launch edge also at 0ns.

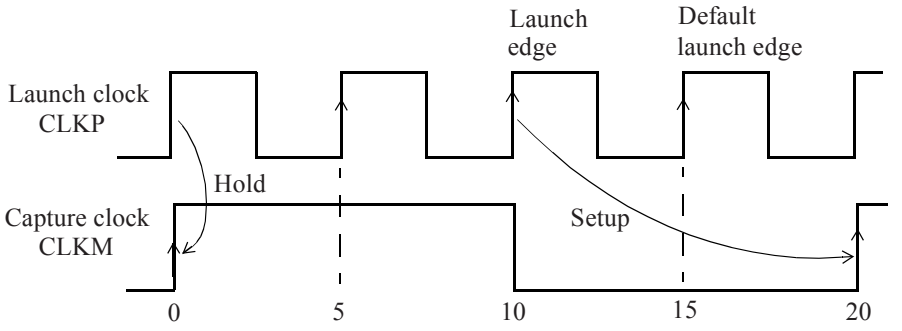


Figure 8-27 *Setup multicycle of 2.*

Here is the setup path report. As expected, the launch clock edge is at 10ns and the capture clock edge is at 20ns.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------------
  clock CLKP (rise edge)                  10.00      10.00
  clock source latency                     0.00      10.00
  CLKP (in)                                0.00      10.00 r
  UCKBUF4/C (CKB  )                        0.07      10.07 r
  UFF3/CK (DFF )                           0.00      10.07 r
  UFF3/Q (DFF ) <-                         0.15      10.22 f
  UNOR0/ZN (NR2  )                         0.05      10.27 r
  UBUF4/Z (BUFF  )                         0.05      10.32 r
  UFF1/D (DFF )                            0.00      10.32 r
  data arrival time                                  10.32
```

```
  clock CLKM (rise edge)                  20.00      20.00
  clock source latency                     0.00      20.00
  CLKM (in)                                0.00      20.00 r
  UCKBUF0/C (CKB  )                        0.06      20.06 r
  UCKBUF2/C (CKB  )                        0.07      20.12 r
  UFF1/CK (DFF )                           0.00      20.12 r
  clock uncertainty                       -0.30      19.82
  library setup time                      -0.04      19.78
  data required time                                 19.78
  -----------------------------------------------------------
  data required time                                 19.78
  data arrival time                                 -10.32
  -----------------------------------------------------------
  slack (MET)                                         9.46
```

Here is the hold path timing report. The hold check is at 0ns where both the capture and launch clocks have rising edges.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UFF3/CK (DFF )                           0.00       0.07 r
  UFF3/Q (DFF ) <-                         0.16       0.22 r
  UNOR0/ZN (NR2  )                         0.02       0.25 f
  UBUF4/Z (BUFF  )                         0.06       0.30 f
  UFF1/D (DFF )                            0.00       0.30 f
  data arrival time                                   0.30

  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF2/C (CKB  )                        0.07       0.12 r
```

```
UFF1/CK (DFF )                          0.00       0.12 r
clock uncertainty                       0.05       0.17
library hold time                       0.01       0.19
data required time                                 0.19
-----------------------------------------------------------
data required time                                 0.19
data arrival time                                 -0.30
-----------------------------------------------------------
slack (MET)                                        0.12
```

Unlike the case of paths from slow to fast clock domains, a good rule of thumb for multi-frequency multicycle path specification in the case of paths from fast to slow clock domains is to use the *-start* option. The setup and hold checks are then adjusted based upon the fast clock.

8.9 Examples

In this section, we describe different scenarios of launch and capture clocks and show how the setup and hold checks are performed. Figure 8-28 shows the configuration for the examples.

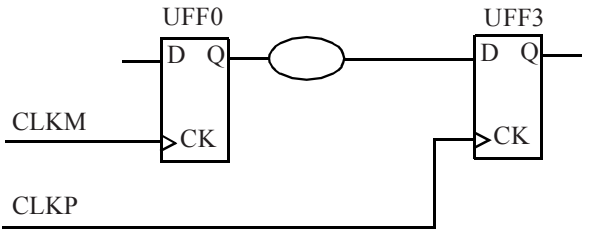


Figure 8-28 *Launch and capture clocks with different relationships.*

Half-cycle Path - Case 1

In this example, the two clocks have the same period but are of opposite phase. Here are the clock specifications - the waveforms are shown in Figure 8-29.

```
create_clock -name CLKM \
  -period 20 -waveform {0 10} [get_ports CLKM]
create_clock -name CLKP \
  -period 20 -waveform {10 20} [get_ports CLKP]
```

The setup check is from a launch edge, at 0ns, to the next capture edge, at 10ns. A half-cycle margin is available for the hold check which validates whether the data launched at 20ns is not captured by the capture edge at 10ns. Here is the setup path report.

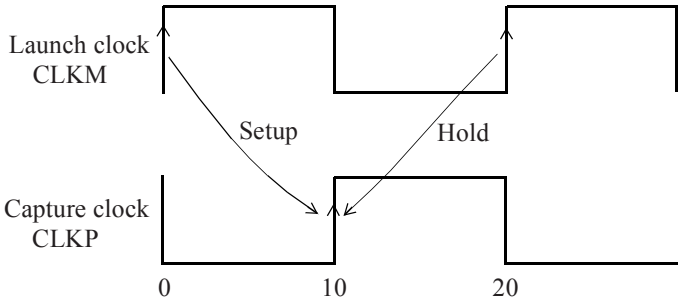


Figure 8-29 *Clock waveforms for half-cycle path - case 1.*

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
```

```
  clock source latency                      0.00       0.00
  CLKM (in)                                 0.00       0.00 r
  UCKBUF0/C (CKB  )                         0.06       0.06 r
  UCKBUF1/C (CKB  )                         0.06       0.11 r
  UFF0/CK (DFF )                            0.00       0.11 r
  UFF0/Q (DFF ) <-                          0.14       0.26 f
  UNAND0/ZN (ND2  )                         0.03       0.29 r
  UFF3/D (DFF )                             0.00       0.29 r
  data arrival time                                    0.29

  clock CLKP (rise edge)                   10.00      10.00
  clock source latency                      0.00      10.00
  CLKP (in)                                 0.00      10.00 r
  UCKBUF4/C (CKB  )                         0.07      10.07 r
  UFF3/CK (DFF )                            0.00      10.07 r
  clock uncertainty                        -0.30       9.77
  library setup time                       -0.04       9.72
  data required time                                   9.72
  ---------------------------------------------------------
  data required time                                   9.72
  data arrival time                                   -0.29
  ---------------------------------------------------------
  slack (MET)                                          9.44
```

Here is the hold path report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                     Incr       Path
  ---------------------------------------------------------
  clock CLKM (rise edge)                   20.00      20.00
  clock source latency                      0.00      20.00
  CLKM (in)                                 0.00      20.00 r
  UCKBUF0/C (CKB  )                         0.06      20.06 r
  UCKBUF1/C (CKB  )                         0.06      20.11 r
  UFF0/CK (DFF )                            0.00      20.11 r
  UFF0/Q (DFF ) <-                          0.14      20.26 r
  UNAND0/ZN (ND2  )                         0.03      20.29 f
```

```
UFF3/D (DFF )                              0.00      20.29 f
data arrival time                                    20.29

clock CLKP (rise edge)                    10.00      10.00
clock source latency                       0.00      10.00
CLKP (in)                                  0.00      10.00 r
UCKBUF4/C (CKB  )                          0.07      10.07 r
UFF3/CK (DFF )                             0.00      10.07 r
clock uncertainty                          0.05      10.12
library hold time                          0.02      10.13
data required time                                   10.13
-------------------------------------------------------------
data required time                                   10.13
data arrival time                                   -20.29
-------------------------------------------------------------
slack (MET)                                          10.16
```

Half-cycle Path - Case 2

This example is similar to case 1 and the launch and capture clocks are of opposite phase. The launch clock is shifted in time. Here are the clock specifications; the waveforms are shown in Figure 8-30.

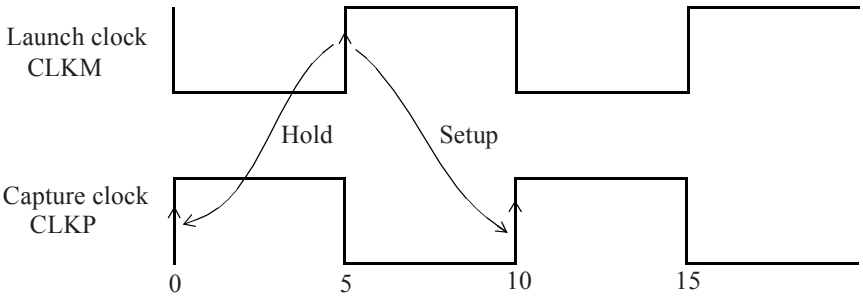


Figure 8-30 *Clock waveforms for half-cycle path - case 2.*

```
create_clock -name CLKM \
  -period 10 -waveform {5 10} [get_ports CLKM]
```

```
create_clock -name CLKP \
  -period 10 -waveform {0 5} [get_ports CLKP]
```

The setup check is from the launch clock edge at 5ns to the next capture clock edge at 10ns. The hold check is from the launch edge at 5ns to the capture edge at 0ns. Here is the setup path report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  CLKM (in)                                0.00       5.00 r
  UCKBUF0/C (CKB  )                        0.06       5.06 r
  UCKBUF1/C (CKB  )                        0.06       5.11 r
  UFF0/CK (DFF )                           0.00       5.11 r
  UFF0/Q (DFF ) <-                         0.14       5.26 f
  UNAND0/ZN (ND2  )                        0.03       5.29 r
  UFF3/D (DFF )                            0.00       5.29 r
  data arrival time                                   5.29

  clock CLKP (rise edge)                  10.00      10.00
  clock source latency                     0.00      10.00
  CLKP (in)                                0.00      10.00 r
  UCKBUF4/C (CKB  )                        0.07      10.07 r
  UFF3/CK (DFF )                           0.00      10.07 r
  clock uncertainty                       -0.30       9.77
  library setup time                      -0.04       9.72
  data required time                                  9.72
  -----------------------------------------------------------
  data required time                                  9.72
  data arrival time                                  -5.29
  -----------------------------------------------------------
  slack (MET)                                         4.44
```

Here is the hold path timing report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  CLKM (in)                                0.00       5.00 r
  UCKBUF0/C (CKB  )                        0.06       5.06 r
  UCKBUF1/C (CKB  )                        0.06       5.11 r
  UFF0/CK (DFF )                           0.00       5.11 r
  UFF0/Q (DFF ) <-                         0.14       5.26 r
  UNAND0/ZN (ND2  )                        0.03       5.29 f
  UFF3/D (DFF )                            0.00       5.29 f
  data arrival time                                   5.29

  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UFF3/CK (DFF )                           0.00       0.07 r
  clock uncertainty                        0.05       0.12
  library hold time                        0.02       0.13
  data required time                                  0.13
  -----------------------------------------------------------
  data required time                                  0.13
  data arrival time                                  -5.29
  -----------------------------------------------------------
  slack (MET)                                         5.16
```

Fast to Slow Clock Domain

In this example, the capture clock is a divide-by-2 of the launch clock. Here are the clock specifications.

```
create_clock -name CLKM \
  -period 10 -waveform {0 5} [get_ports CLKM]
create_clock -name CLKP \
  -period 20 -waveform {0 10} [get_ports CLKP]
```

The waveforms are shown in Figure 8-31. The setup check is from the launch edge at 10ns to the capture edge at 20ns. The hold check is from the launch edge at 0ns to the capture edge at 0ns. Here is the setup path report.

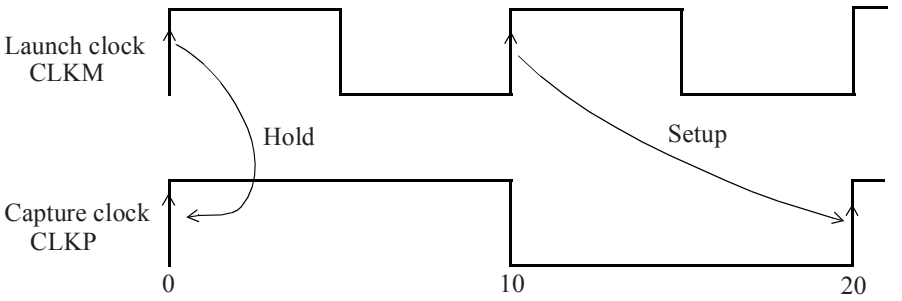


Figure 8-31 *Fast to slow clock domain example clocks.*

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                  10.00      10.00
  clock source latency                     0.00      10.00
```

```
  CLKM (in)                                0.00       10.00 r
  UCKBUF0/C (CKB  )                        0.06       10.06 r
  UCKBUF1/C (CKB  )                        0.06       10.11 r
  UFF0/CK (DFF )                           0.00       10.11 r
  UFF0/Q (DFF ) <-                         0.14       10.26 f
  UNAND0/ZN (ND2  )                        0.03       10.29 r
  UFF3/D (DFF )                            0.00       10.29 r
  data arrival time                                   10.29

  clock CLKP (rise edge)                  20.00       20.00
  clock source latency                     0.00       20.00
  CLKP (in)                                0.00       20.00 r
  UCKBUF4/C (CKB  )                        0.07       20.07 r
  UFF3/CK (DFF )                           0.00       20.07 r
  clock uncertainty                       -0.30       19.77
  library setup time                      -0.04       19.72
  data required time                                  19.72
  ------------------------------------------------------------
  data required time                                  19.72
  data arrival time                                  -10.29
  ------------------------------------------------------------
  slack (MET)                                          9.44
```

Here is the hold path timing report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UNAND0/ZN (ND2  )                        0.03       0.29 f
  UFF3/D (DFF )                            0.00       0.29 f
```

```
data arrival time                                    0.29

clock CLKP (rise edge)                    0.00       0.00
clock source latency                      0.00       0.00
CLKP (in)                                 0.00       0.00 r
UCKBUF4/C (CKB  )                         0.07       0.07 r
UFF3/CK (DFF )                            0.00       0.07 r
clock uncertainty                         0.05       0.12
library hold time                         0.02       0.13
data required time                                   0.13
-------------------------------------------------------------
data required time                                   0.13
data arrival time                                   -0.29
-------------------------------------------------------------
slack (MET)                                          0.16
```

Slow to Fast Clock Domain

In this example, the capture clock is at 2x the speed of the launch clock. Figure 8-32 shows the clock edge corresponding to the setup and hold checks. Setup check is done from the launch edge, at 0ns, to the next capture edge, at 5ns. The hold check is done with the capture edge one cycle prior to the setup capture edge, that is, both launch and capture edges are at 0ns.

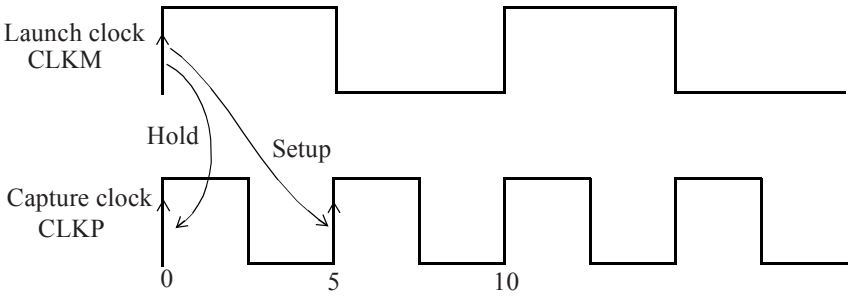


Figure 8-32 *Slow to fast clock domain example clocks.*

Here is the setup path report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 f
  UNAND0/ZN (ND2  )                        0.03       0.29 r
  UFF3/D (DFF )                            0.00       0.29 r
  data arrival time                                   0.29

  clock CLKP (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  CLKP (in)                                0.00       5.00 r
  UCKBUF4/C (CKB  )                        0.07       5.07 r
  UFF3/CK (DFF )                           0.00       5.07 r
  clock uncertainty                       -0.30       4.77
  library setup time                      -0.04       4.72
  data required time                                  4.72
  -----------------------------------------------------------
  data required time                                  4.72
  data arrival time                                  -0.29
  -----------------------------------------------------------
  slack (MET)                                         4.44
```

Here is the hold timing report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min
```

```
Point                                          Incr       Path
-------------------------------------------------------------
clock CLKM (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKM (in)                                      0.00       0.00 r
UCKBUF0/C (CKB  )                              0.06       0.06 r
UCKBUF1/C (CKB  )                              0.06       0.11 r
UFF0/CK (DFF )                                 0.00       0.11 r
UFF0/Q (DFF ) <-                               0.14       0.26 r
UNAND0/ZN (ND2  )                              0.03       0.29 f
UFF3/D (DFF )                                  0.00       0.29 f
data arrival time                                         0.29

clock CLKP (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
CLKP (in)                                      0.00       0.00 r
UCKBUF4/C (CKB  )                              0.07       0.07 r
UFF3/CK (DFF )                                 0.00       0.07 r
clock uncertainty                              0.05       0.12
library hold time                              0.02       0.13
data required time                                        0.13
-------------------------------------------------------------
data required time                                        0.13
data arrival time                                        -0.29
-------------------------------------------------------------
slack (MET)                                               0.16
```

8.10 Multiple Clocks

8.10.1 Integer Multiples

Often there are multiple clocks defined in a design with frequencies that are simple (or integer) multiples of each other. In such cases, STA is performed by computing a common base period among all related clocks (two clocks are **related** if they have a data path between their domains). The common base period is established so that all clocks are synchronized.

Here is an example that shows four related clocks:

```
create_clock -name CLKM \
  -period 20 -waveform {0 10} [get_ports CLKM]
create_clock -name CLKQ -period 10 -waveform {0 5}
create_clock -name CLKP \
  -period 5 -waveform {0 2.5} [get_ports CLKP]
```

A common base period of 20ns, as shown in Figure 8-33, is used when analyzing paths between the *CLKP* and *CLKM* clock domains.

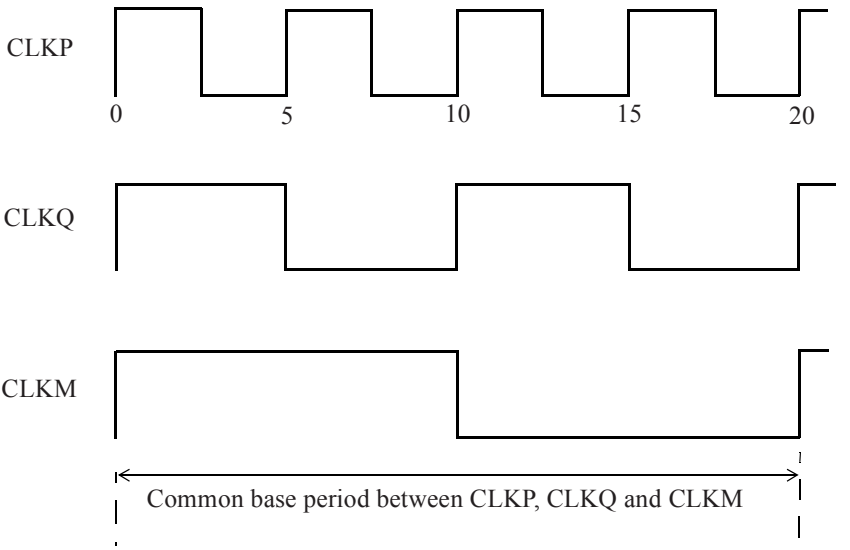


Figure 8-33 *Integer multiple clocks.*

```
Expanding clock 'CLKP' to base period of 20.00 (old period was
5.00, added 6 edges).
Expanding clock 'CLKQ' to base period of 20.00 (old period was
10.00, added 2 edges).
```

Here is a setup timing report for a path that goes from the faster clock to the slower clock.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                  15.00      15.00
  clock source latency                     0.00      15.00
  CLKP (in)                                0.00      15.00 r
  UCKBUF4/C (CKB  )                        0.07      15.07 r
  UFF3/CK (DFF )                           0.00      15.07 r
  UFF3/Q (DFF ) <-                         0.15      15.22 f
  UNOR0/ZN (NR2  )                         0.05      15.27 r
  UBUF4/Z (BUFF  )                         0.05      15.32 r
  UFF1/D (DFF )                            0.00      15.32 r
  data arrival time                                  15.32

  clock CLKM (rise edge)                  20.00      20.00
  clock source latency                     0.00      20.00
  CLKM (in)                                0.00      20.00 r
  UCKBUF0/C (CKB  )                        0.06      20.06 r
  UCKBUF2/C (CKB  )                        0.07      20.12 r
  UFF1/CK (DFF )                           0.00      20.12 r
  clock uncertainty                       -0.30      19.82
  library setup time                      -0.04      19.78
  data required time                                 19.78
  -----------------------------------------------------------
  data required time                                 19.78
  data arrival time                                 -15.32
  -----------------------------------------------------------
  slack (MET)                                         4.46
```

Here is the corresponding hold path report.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
```

```
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: min

Point                                       Incr       Path
-----------------------------------------------------------
clock CLKP (rise edge)                      0.00       0.00
clock source latency                        0.00       0.00
CLKP (in)                                   0.00       0.00 r
UCKBUF4/C (CKB  )                           0.07       0.07 r
UFF3/CK (DFF )                              0.00       0.07 r
UFF3/Q (DFF ) <-                            0.16       0.22 r
UNOR0/ZN (NR2  )                            0.02       0.25 f
UBUF4/Z (BUFF  )                            0.06       0.30 f
UFF1/D (DFF )                               0.00       0.30 f
data arrival time                                      0.30

clock CLKM (rise edge)                      0.00       0.00
clock source latency                        0.00       0.00
CLKM (in)                                   0.00       0.00 r
UCKBUF0/C (CKB  )                           0.06       0.06 r
UCKBUF2/C (CKB  )                           0.07       0.12 r
UFF1/CK (DFF )                              0.00       0.12 r
clock uncertainty                           0.05       0.17
library hold time                           0.01       0.19
data required time                                     0.19
-----------------------------------------------------------
data required time                                     0.19
data arrival time                                     -0.30
-----------------------------------------------------------
slack (MET)                                            0.12
```

8.10.2 Non-Integer Multiples

Consider the case when there is a data path between two clock domains whose frequencies are not multiples of each other. For example, the launch clock is divide-by-8 of a common clock and the capture clock is divide-by-5 of the common clock as shown in Figure 8-34. This section describes how the setup and hold checks are performed in such a situation.

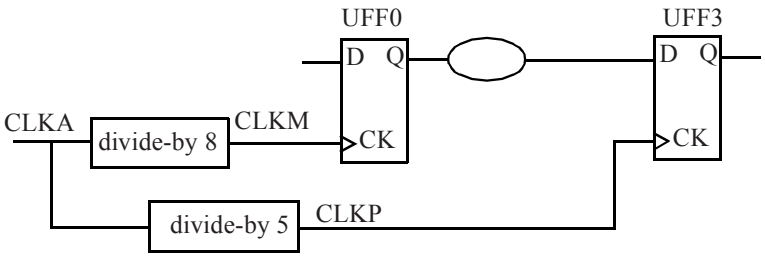


Figure 8-34 *Non-integer multiple clocks.*

Here are the clock definitions (waveforms are shown in Figure 8-35).

```
create_clock -name CLKM \
  -period 8 -waveform {0 4} [get_ports CLKM]
create_clock -name CLKQ -period 10 -waveform {0 5}
create_clock -name CLKP \
  -period 5 -waveform {0 2.5} [get_ports CLKP]
```

The timing analysis process computes a common period for the related clocks, and the clocks are then expanded to this base period. Note that the common period is found only for related clocks (that is, clocks that have timing paths between them). The common period for data paths between *CLKQ* and *CLKP* is expanded to a base period of 10ns only. The common period for data paths between *CLKM* and *CLKQ* is 40ns, and the common period for data paths between *CLKM* and *CLKP* is also 40ns.

Let us consider a data path from the *CLKM* clock domain to the *CLKP* clock domain. The common base period for timing analysis is 40ns.

```
Expanding clock 'CLKM' to base period of 40.00 (old period was
8.00, added 8 edges).
Expanding clock 'CLKP' to base period of 40.00 (old period was
5.00, added 14 edges).
```

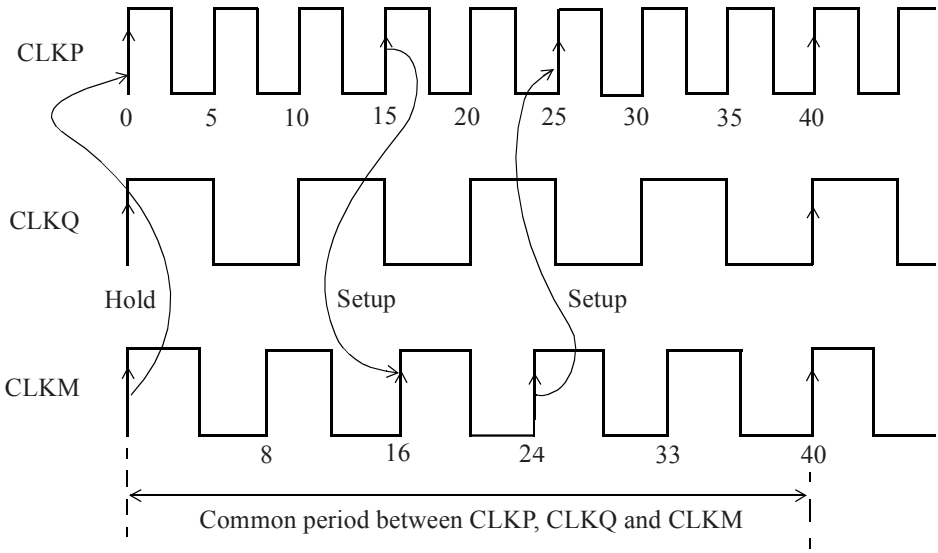


Figure 8-35 *Setup and hold checks for non-integer multiple clocks.*

The setup check occurs over the minimum time between the launch edge and the capture edge of the clock. In our example path from *CLKM* to *CLKP*, this would be a launch at 24ns of clock *CLKM* and a capture at 25ns of clock *CLKP*.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                  24.00      24.00
  clock source latency                     0.00      24.00
  CLKM (in)                                0.00      24.00 r
```

```
  UCKBUF0/C (CKB  )                        0.06       24.06 r
  UCKBUF1/C (CKB  )                        0.06       24.11 r
  UFF0/CK (DFF )                           0.00       24.11 r
  UFF0/Q (DFF ) <-                         0.14       24.26 f
  UNAND0/ZN (ND2  )                        0.03       24.29 r
  UFF3/D (DFF )                            0.00       24.29 r
  data arrival time                                   24.29

  clock CLKP (rise edge)                  25.00       25.00
  clock source latency                     0.00       25.00
  CLKP (in)                                0.00       25.00 r
  UCKBUF4/C (CKB  )                        0.07       25.07 r
  UFF3/CK (DFF )                           0.00       25.07 r
  clock uncertainty                       -0.30       24.77
  library setup time                      -0.04       24.72
  data required time                                  24.72
  ------------------------------------------------------------
  data required time                                  24.72
  data arrival time                                  -24.29
  ------------------------------------------------------------
  slack (MET)                                          0.44
```

Here is the hold timing path. The most restrictive hold path is from a launch at 0ns of *CLKM* to the capture edge of 0ns of *CLKP*.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UCKBUF0/C (CKB  )                        0.06       0.06 r
  UCKBUF1/C (CKB  )                        0.06       0.11 r
  UFF0/CK (DFF )                           0.00       0.11 r
  UFF0/Q (DFF ) <-                         0.14       0.26 r
  UNAND0/ZN (ND2  )                        0.03       0.29 f
```

```
  UFF3/D (DFF )                              0.00       0.29 f
  data arrival time                                     0.29

  clock CLKP (rise edge)                     0.00       0.00
  clock source latency                       0.00       0.00
  CLKP (in)                                  0.00       0.00 r
  UCKBUF4/C (CKB  )                          0.07       0.07 r
  UFF3/CK (DFF )                             0.00       0.07 r
  clock uncertainty                          0.05       0.12
  library hold time                          0.02       0.13
  data required time                                    0.13
  -----------------------------------------------------------
  data required time                                    0.13
  data arrival time                                    -0.29
  -----------------------------------------------------------
  slack (MET)                                           0.16
```

Now we examine the setup path from the *CLKP* clock domain to the *CLKM* clock domain. In this case, the most restrictive setup path is from a launch edge at 15ns of clock *CLKP* to the capture edge at 16ns of clock *CLKM*.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max

  Point                                      Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                    15.00      15.00
  clock source latency                       0.00      15.00
  CLKP (in)                                  0.00      15.00 r
  UCKBUF4/C (CKB  )                          0.07      15.07 r
  UFF3/CK (DFF )                             0.00      15.07 r
  UFF3/Q (DFF ) <-                           0.15      15.22 f
  UNOR0/ZN (NR2  )                           0.05      15.27 r
  UBUF4/Z (BUFF  )                           0.05      15.32 r
  UFF1/D (DFF )                              0.00      15.32 r
  data arrival time                                    15.32

  clock CLKM (rise edge)                    16.00      16.00
```

```
  clock source latency                    0.00      16.00
  CLKM (in)                               0.00      16.00 r
  UCKBUF0/C (CKB  )                       0.06      16.06 r
  UCKBUF2/C (CKB  )                       0.07      16.12 r
  UFF1/CK (DFF )                          0.00      16.12 r
  clock uncertainty                      -0.30      15.82
  library setup time                     -0.04      15.78
  data required time                                15.78
  ------------------------------------------------------------
  data required time                                15.78
  data arrival time                                -15.32
  ------------------------------------------------------------
  slack (MET)                                       0.46
```

Here is the hold path report, again the most restrictive one is the one at 0ns.

```
Startpoint: UFF3 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min

  Point                                   Incr       Path
  ------------------------------------------------------------
  clock CLKP (rise edge)                  0.00       0.00
  clock source latency                    0.00       0.00
  CLKP (in)                               0.00       0.00 r
  UCKBUF4/C (CKB  )                       0.07       0.07 r
  UFF3/CK (DFF )                          0.00       0.07 r
  UFF3/Q (DFF ) <-                        0.16       0.22 r
  UNOR0/ZN (NR2  )                        0.02       0.25 f
  UBUF4/Z (BUFF  )                        0.06       0.30 f
  UFF1/D (DFF )                           0.00       0.30 f
  data arrival time                                  0.30

  clock CLKM (rise edge)                  0.00       0.00
  clock source latency                    0.00       0.00
  CLKM (in)                               0.00       0.00 r
  UCKBUF0/C (CKB  )                       0.06       0.06 r
  UCKBUF2/C (CKB  )                       0.07       0.12 r
  UFF1/CK (DFF )                          0.00       0.12 r
  clock uncertainty                       0.05       0.17
```

```
  library hold time                          0.01       0.19
  data required time                                    0.19
  -----------------------------------------------------------
  data required time                                    0.19
  data arrival time                                    -0.30
  -----------------------------------------------------------
  slack (MET)                                           0.12
```

8.10.3 Phase Shifted

Here is an example where two clocks are ninety degrees phase-shifted with respect to each other.

```
create_clock -period 2.0 -waveform {0 1.0} [get_ports CKM]
create_clock -period 2.0 -waveform {0.5 1.5} \
  [get_ports CKM90]
```

Figure 8-36 shows an example with these clocks. The setup path timing report is as follows.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CKM)
Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CKM90)
  Path Group: CKM90
  Path Type: max

  Point                                      Incr       Path
  -----------------------------------------------------------
  clock CKM  (rise edge)                     0.00       0.00
  clock source latency                       0.00       0.00
  CKM  (in)                                  0.00       0.00 r
  UCKBUF0/C (CKB  )                          0.06       0.06 r
  UCKBUF1/C (CKB  )                          0.06       0.11 r
  UFF0/CK (DF  )                             0.00       0.11 r
  UFF0/Q (DF  ) <-                           0.14       0.26 f
  UNAND0/ZN (ND2  )                          0.03       0.29 r
  UFF3/D (DF  )                              0.00       0.29 r
  data arrival time                                     0.29

  clock CKM90(rise edge)                     0.50       0.50
  clock source latency                       0.00       0.50
```

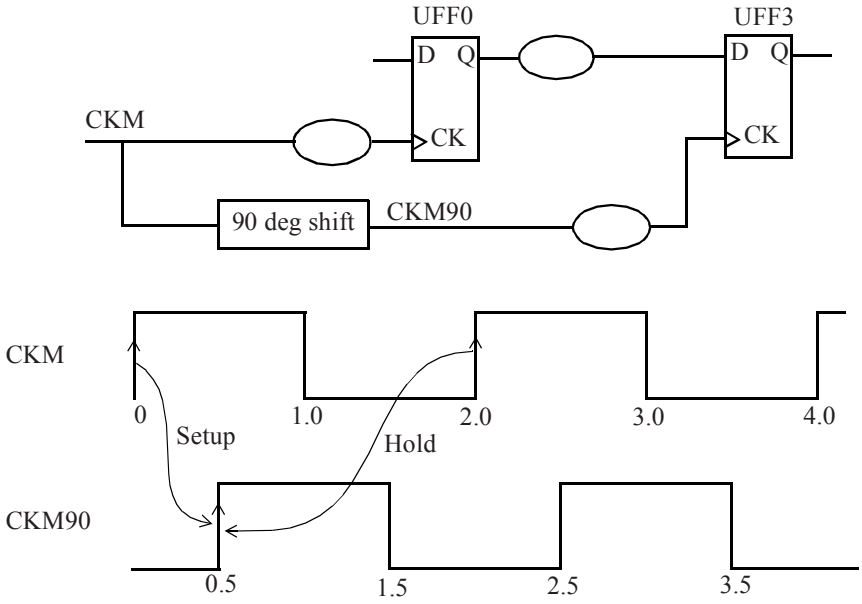


Figure 8-36 *Phase-shifted clocks.*

```
CKM90(in)                                  0.00       0.50 r
UCKBUF4/C (CKB  )                          0.07       0.57 r
UFF3/CK (DF  )                             0.00       0.57 r
clock uncertainty                         -0.30       0.27
library setup time                        -0.04       0.22
data required time                                    0.22
-------------------------------------------------------------
data required time                                    0.22
data arrival time                                    -0.29
-------------------------------------------------------------
slack (VIOLATED)                                     -0.06
```

The first rising edge of *CKM90* at 0.5ns is the capture edge. The hold check is for one cycle before the setup capture edge. For the launch edge at 2ns,

the setup capture edge is at 2.5ns. Thus the hold check is at the previous capture edge which is at 0.5ns. The hold path timing report is as follows.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CKM)
Endpoint: UFF3 (rising edge-triggered flip-flop clocked by CKM90)
  Path Group: CKM90
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CKM  (rise edge)                   2.00       2.00
  clock source latency                     0.00       2.00
  CKM  (in)                                0.00       2.00 r
  UCKBUF0/C (CKB  )                        0.06       2.06 r
  UCKBUF1/C (CKB  )                        0.06       2.11 r
  UFF0/CK (DF  )                           0.00       2.11 r
  UFF0/Q (DF  ) <-                         0.14       2.26 r
  UNAND0/ZN (ND2  )                        0.03       2.29 f
  UFF3/D (DF  )                            0.00       2.29 f
  data arrival time                                   2.29

  clock CKM90(rise edge)                   0.50       0.50
  clock source latency                     0.00       0.50
  CLM90(in)                                0.00       0.50 r
  UCKBUF4/C (CKB  )                        0.07       0.57 r
  UFF3/CK (DF  )                           0.00       0.57 r
  clock uncertainty                        0.05       0.62
  library hold time                        0.02       0.63
  data required time                                  0.63
  -----------------------------------------------------------
  data required time                                  0.63
  data arrival time                                  -2.29
  -----------------------------------------------------------
  slack (MET)                                         1.66
```

Other timing checks such as data to data checks and clock gating checks are described in Chapter 10.

❑

CHAPTER

9

Interface Analysis

This chapter describes the timing analysis procedures for various types of input and output paths, and several commonly used interfaces. Timing analysis of special interfaces such as for SRAMs and the timing analysis of source synchronous interfaces such as those used for DDR SDRAMs are also described.

9.1 IO Interfaces

This section presents examples that illustrate how the constraints on input and output interfaces of the DUA are defined. Later sections provide examples of timing constraints for the SRAM and DDR SDRAM interfaces.

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_9,

9.1.1 Input Interface

There are broadly two alternate ways of specifying the timing of the inputs:

i. The waveforms at an input of the DUA are provided as AC specifications[1].

ii. The path delay of the external logic to an input is specified.

Waveform Specification at Inputs

Consider the input AC specification shown in Figure 9-1. The specification is that the input *CIN* is stable 4.3ns before the rising edge of clock *CLKP*, and that the value remains stable until 2ns after the rising edge of the clock.

Consider the 4.3ns specification first. Given the clock cycle of 8ns (as shown in Figure 9-1), this requirement maps into the delay from the virtual flip-flop (the flip-flop that is driving this input) to the input *CIN*. The delay from the virtual flip-flop clock to *CIN* must be at most 3.7ns (= 8.0 - 4.3), the maximum delay being at 3.7ns. This ensures that the data at input *CIN* arrives 4.3ns prior to the rising edge. Hence, this part of the AC specification can equivalently be specified as a max input delay of 3.7ns.

The AC specification also states that the input *CIN* is stable for 2ns after the rising edge of the clock. This specification can also be mapped into the delay from the virtual flip-flop, that is, the delay from the virtual flip-flop to input *CIN* must be at least 2.0ns. Hence the minimum input delay is specified as 2.0ns.

1. Specification of a digital device is provided in two parts: DC - the constant values (static), and AC - the changing waveforms (dynamic).

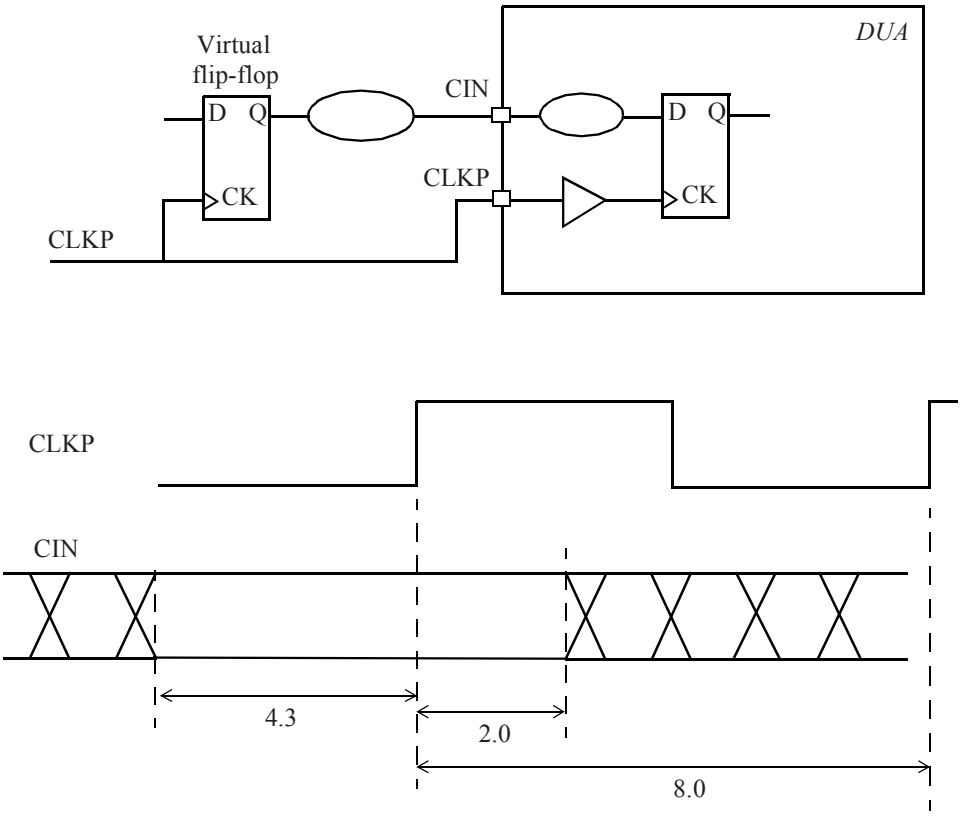


Figure 9-1 *AC specification for input port.*

Here are the input constraints.

```
create_clock -name CLKP -period 8 [get_ports CLKP]
set_input_delay -min 2.0 -clock CLKP [get_ports CIN]
set_input_delay -max 3.7 -clock CLKP [get_ports CIN]
```

Here are the path reports for the design under these input conditions. First is the setup report.

```
Startpoint: CIN (input port clocked by CLKP)
Endpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
Path Group: CLKP
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKP (rise edge)                   0.00       0.00
clock network delay (propagated)         0.00       0.00
input external delay                     3.70       3.70 f
CIN (in)                                 0.00       3.70 f
UBUF5/Z (BUFF  )                         0.05       3.75 f
UXOR1/Z (XOR2  )                         0.10       3.85 r
UFF4/D (DF  )                            0.00       3.85 r
data arrival time                                   3.85

clock CLKP (rise edge)                   8.00       8.00
clock source latency                     0.00       8.00
CLKP (in)                                0.00       8.00 r
UCKBUF4/C (CKB  )                        0.07       8.07 r
UCKBUF5/C (CKB  )                        0.06       8.13 r
UFF4/CP (DF  )                           0.00       8.13 r
library setup time                      -0.05       8.08
data required time                                  8.08
-----------------------------------------------------------
data required time                                  8.08
data arrival time                                  -3.85
-----------------------------------------------------------
slack (MET)                                         4.23
```

The max input delay specified (3.7ns) is added to the data path. The setup check ensures that delay inside the DUA is less than 4.3ns and the proper data can be latched. Next is the hold timing report.

```
Startpoint: CIN (input port clocked by CLKP)
Endpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
```

```
Path Group: CLKP
Path Type: min

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKP (rise edge)                   0.00       0.00
clock network delay (propagated)         0.00       0.00
input external delay                     2.00       2.00 r
CIN (in) <-                              0.00       2.00 r
UBUF5/Z (BUFF  )                         0.05       2.05 r
UXOR1/Z (XOR2  )                         0.07       2.12 r
UFF4/D (DF  )                            0.00       2.12 r
data arrival time                                   2.12

clock CLKP (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
CLKP (in)                                0.00       0.00 r
UCKBUF4/C (CKB  )                        0.07       0.07 r
UCKBUF5/C (CKB  )                        0.06       0.13 r
UFF4/CK (DF  )                           0.00       0.13 r
clock uncertainty                        0.05       0.18
library hold time                       -0.00       0.17
data required time                                  0.17
-----------------------------------------------------------
data required time                                  0.17
data arrival time                                  -2.12
-----------------------------------------------------------
slack (MET)                                         1.95
```

The min input delay is added to the data path in the hold check. The check ensures that the earliest data change at 2ns after the clock edge does not overwrite the previous data at the flip-flop.

Path Delay Specification to Inputs

When the path delays of the external logic connected to an input are known, specifying the input constraints is a straightforward task. Any delays along the external logic path to the input are added and the path delay is specified using the *set_input_delay* command.

Figure 9-2 shows an example of external logic path to input. The *Tck2q* and *Tc1* delays are added to obtain the external delay. Knowing *Tck2q* and *Tc1*, the input delay is directly obtained as *Tck2q* + *Tc1*.

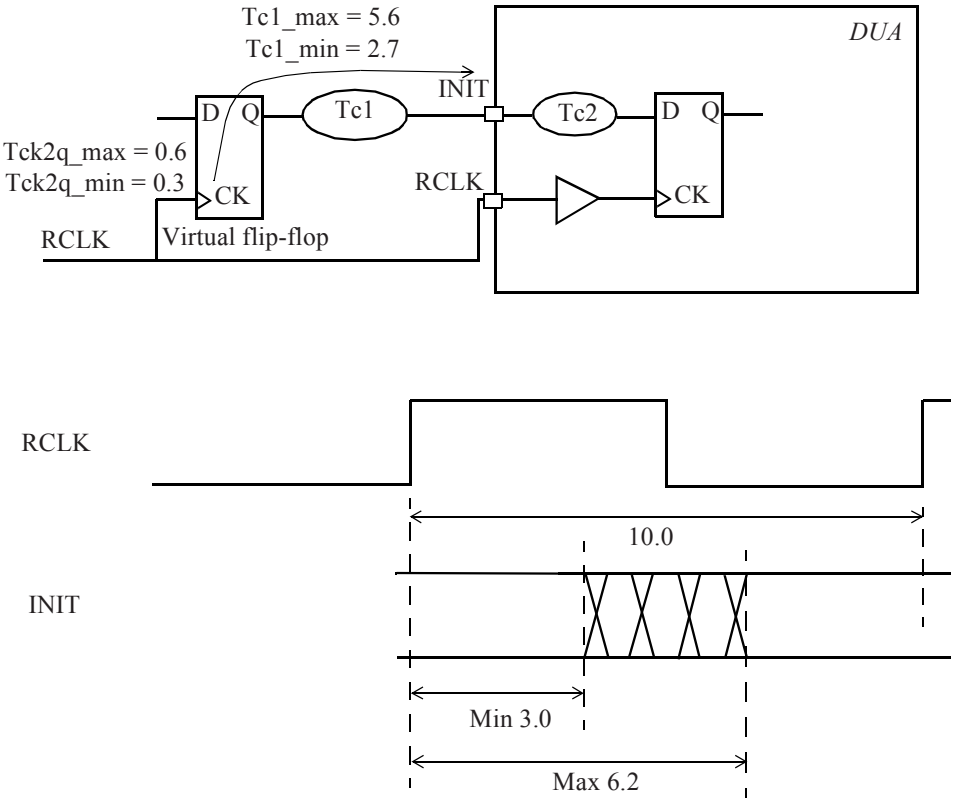


Figure 9-2 *Input path delay specifications.*

The external max and min path delays translate to the following input constraints.

```
create_clock -name RCLK -period 10 [get_ports RCLK]
set_input_delay -max 6.2 -clock RCLK [get_ports INIT]
set_input_delay -min 3.0 -clock RCLK [get_ports INIT]
```

The path reports for these are similar to the ones in Section 8.1 and Section 8.2.

Note that when computing the arrival time at the data pin of the flip-flop inside the design, the max and min input delay values get added to the data path delay depending on whether a *max path check* (setup) or a *min path check* (hold) is being performed.

9.1.2 Output Interface

Similar to the input case, there are broadly two alternate ways for specifying the output timing requirements:

i. The required waveforms at the output of the DUA are provided as AC specifications.

ii. The path delay of external logic is specified.

Output Waveform Specification

Consider the output AC specification shown in Figure 9-3. The output *QOUT* should be stable at the output 2ns prior to the rising edge of clock *CLKP*. Also, the output should not change until 1.5ns after the rising edge of the clock. These constraints are normally obtained from the setup and hold requirements of the external block that interfaces with *QOUT*.

Here are the constraints that expresses this requirement on the output.

```
create_clock -name CLKP -period 6 \
  -waveform {0 3} [get_ports CLKP}
```

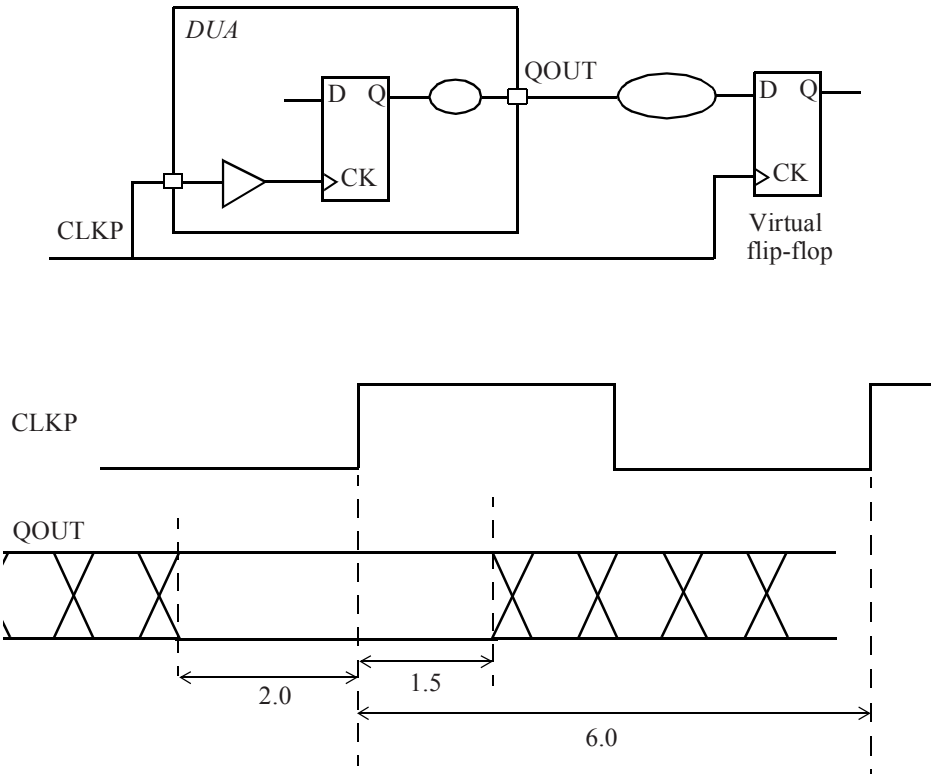


Figure 9-3 *Output AC specification.*

```
# Setup delay of virtual flip-flop:
set_output_delay -clock CLKP -max 2.0 [get_ports QOUT]
# Hold time for virtual flip-flop:
set_output_delay -clock CLKP -min -1.5 [get_ports QOUT]
```

The maximum output path delay is specified as 2.0ns. This will ensure that data *QOUT* changes ahead of the 2ns window before the clock edge. The minimum output path delay of -1.5ns specifies the requirement from the perspective of the virtual flip-flop, that is, to ensure the 1.5ns hold require-

ment at the output *QOUT*. A hold requirement of 1.5ns maps to a *set_output_delay* min of -1.5ns.

Here is the setup timing path report.

```
Startpoint: UFF6 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: QOUT (output port clocked by CLKP)
  Path Group: CLKP
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UCKBUF6/C (CKB  )                        0.07       0.14 r
  UCKBUF7/C (CKB  )                        0.05       0.19 r
  UFF6/CK (DFCN  )                         0.00       0.19 r
  UFF6/Q (DFCN  )                          0.16       0.35 r
  UAND1/Z (AN2  )                          1.31       1.66 r
  QOUT (out)                               0.00       1.66 r
  data arrival time                                   1.66

  clock CLKP (rise edge)                   6.00       6.00
  clock network delay (propagated)         0.00       6.00
  clock uncertainty                       -0.30       5.70
  output external delay                   -2.00       3.70
  data required time                                  3.70
  -----------------------------------------------------------
  data required time                                  3.70
  data arrival time                                  -1.66
  -----------------------------------------------------------
  slack (MET)                                         2.04
```

The max output delay is subtracted from the next clock edge to determine the required arrival time at the output of the DUA.

The hold timing check path report is next.

```
Startpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: QOUT (output port clocked by CLKP)
  Path Group: CLKP
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.07       0.07 r
  UCKBUF5/C (CKB  )                        0.06       0.13 r
  UFF4/CK (DF  )                           0.00       0.13 r
  UFF4/Q (DF  )                            0.14       0.27 f
  UAND1/Z (AN2  )                          0.75       1.02 f
  QOUT (out)                               0.00       1.02 f
  data arrival time                                   1.02

  clock CLKP (rise edge)                   0.00       0.00
  clock network delay (propagated)         0.00       0.00
  clock uncertainty                        0.05       0.05
  output external delay                    1.50       1.55
  data required time                                  1.55
  -----------------------------------------------------------
  data required time                                  1.55
  data arrival time                                  -1.02
  -----------------------------------------------------------
  slack (VIOLATED)                                   -0.53
```

The min output delay (of -1.5ns) is subtracted from the capture clock edge to determine the earliest arrival time at the output of the DUA that meets the hold requirement. It is common to have a negative min output delay requirement.

External Path Delays for Output

In this case, the path delay of the external logic is explicitly specified. See the example in Figure 9-4.

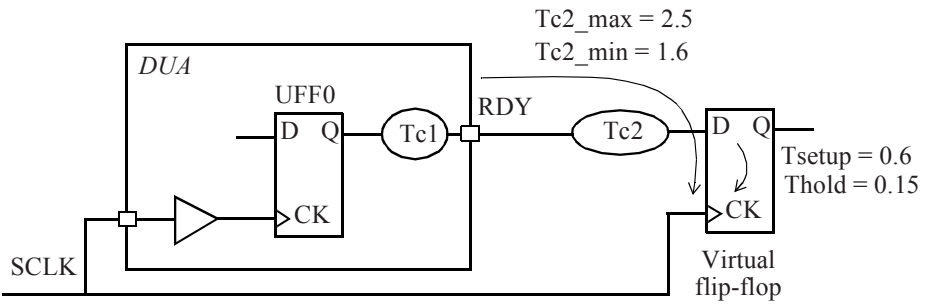


Figure 9-4 *Output path delay specifications.*

Let us examine the setup check first. A max output delay (*set_output_delay -max*) setting is obtained from the *Tc2_max* and *Tsetup*. To check the setup requirement for output paths between flip-flops inside DUA (such as *UFF0)* and the virtual flip-flop, the max output delay is specified as *Tc2_max* + *Tsetup*.

Next let us examine the hold check. A min output delay (*set_output_delay -min*) setting is obtained from *Tc2_min* and *Thold*. Since the hold of a capture flip-flop is added to the capture clock path, the min output delay is specified as (*Tc2_min - Thold*).

The constraints on the output translate to the following:

```
create_clock -name SCLK -period 5 [get_ports SCLK]
# Setup of the external logic (Tc2_max = 2.5,
# Tsetup = 0.6):
set_output_delay -max 3.1 -clock SCLK [get_ports RDY]
# Hold of the external logic (Tc2_min=1.6, Thold=0.15):
set_output_delay -min 1.45 -clock SCLK [get_ports RDY]
```

The path reports for these are similar to the ones in Section 8.1 and Section 8.2.

9.1.3 Output Change within Window

The *set_output_delay* specification can be used to specify maximum and minimum arrival times of an output signal with respect to a clock. This section considers the special case of specifying constraints that verify the scenario when the output can change only within a timing window relative to the clock edge. This requirement occurs quite often while verifying the timing of source synchronous interfaces.

In source synchronous interfaces, the clock also appears along with the data as an output. In such cases, there is normally a requirement for a timing relationship between the clock and the data. For example, the output data may be required to change only within a specific window around the rising edge of the clock.

An example requirement for a source synchronous interface is shown in Figure 9-5.

The requirement is that each bit of *DATAQ* can only change in the specified window 2ns prior to the clock rising edge and up to 1ns after the clock rising edge. This is quite different from the output delay specifications discussed in previous sections where the data pins are required to be stable in a specified timing window around the clock rising edge.

We create a generated clock on *CLK_STROBE* whose master clock is *CLKM*. This is to help specify timing constraints corresponding to the requirements of this interface.

```
create_clock -name CLKM -period 6 [get_ports CLKM}
create_generated_clock -name CLK_STROBE -source CLKM \
  -divide_by 1 [get_ports CLK_STROBE]
```

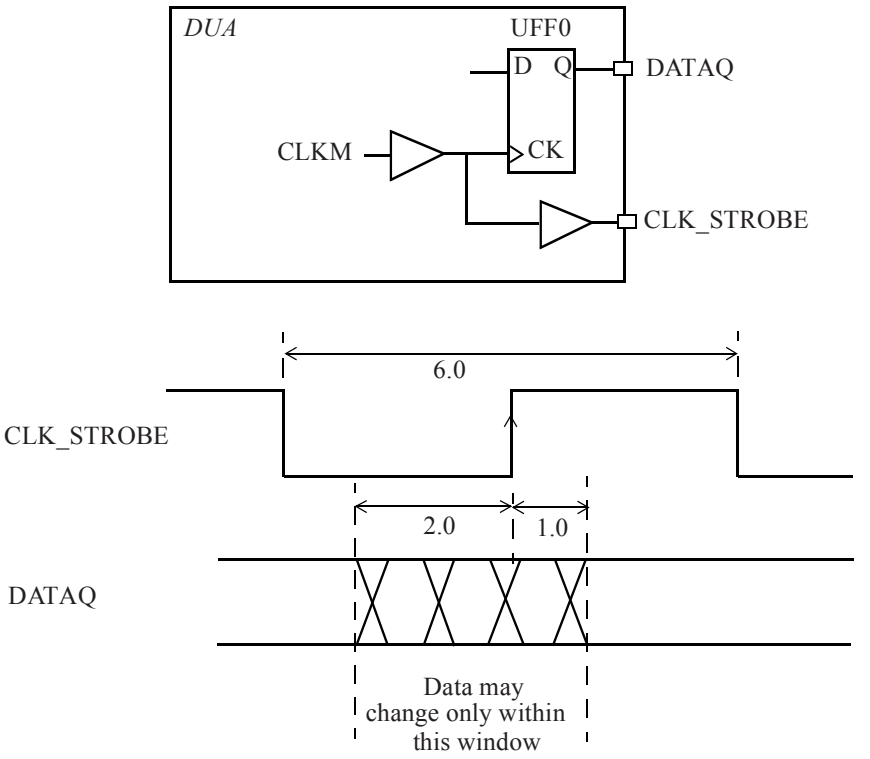


Figure 9-5 *Data change allowed only within a window around the clock.*

The window requirement is specified using a combination of setup and hold checks with multicycle path specifications. The timing requirement is mapped to a setup check that has to occur on a single rising edge (same edge for launch and capture). Thus, we specify a multicycle of 0 for setup.

```
set_multicycle_path 0 -setup -to [get_ports DATAQ]
```

In addition, the hold check has to occur on the same edge, thus we need to specify a multicycle of -1 (minus one) for hold check.

```
set_multicycle_path -1 -hold -to [get_ports DATAQ]
```

Now specify the timing constraints on the output with respect to the clock *CLK_STROBE*.

```
set_output_delay -max -1.0 -clock CLK_STROBE \
  [get_ports DATAQ]
set_output_delay -min +2.0 -clock CLK_STROBE \
  [get_ports DATAQ]
```

Notice that the *min* value of the output delay specification is larger than the *max* specification. This anomaly exists because, in this scenario, the output delay specification does not correspond to an actual logic block. Unlike the case of a typical output interface where the output delay specification corresponds to a logic block at the output, the *set_output_delay* specification in a source synchronous interface is just a mechanism to verify whether the outputs are constrained to switch within a specified window around the clock. Thus, we have the anomaly of the *min* output delay specification being larger than the *max* output delay specification.

Here is the setup timing check path report for the constraints specified.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: DATAQ (output port clocked by CLK_STROBE)
  Path Group: CLK_STROBE
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF0/C (CKB  )                        0.06       0.10 r
  UFF0/CK (DF  )                           0.00       0.10 r
  UFF0/Q (DF  )                            0.13       0.23 r
  UBUF1/Z (BUFF  )                         0.38       0.61 r
```

```
  DATAQ (out)                              0.00       0.61 r
  data arrival time                                   0.61

  clock CLK_STROBE (rise edge)             0.00       0.00
  clock CLKM (source latency)              0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF1/C (CKB  )                        0.05       0.09 r
  CLK_STROBE (out)                         0.00       0.09 r
  clock uncertainty                       -0.30      -0.21
  output external delay                    1.00       0.79
  data required time                                  0.79
  ------------------------------------------------------------
  data required time                                  0.79
  data arrival time                                  -0.61
  ------------------------------------------------------------
  slack (MET)                                        0.18
```

Notice that the launch and the capture edges are the same clock edge, which is at time 0. The report shows that *DATAQ* changes at 0.61ns while the *CLK_STROBE* changes at 0.09ns. Since *DATAQ* can change within 1ns of the *CLK_STROBE*, there is a slack of 0.18ns after accounting for the 0.3ns of clock uncertainty.

Here is the hold path report that checks for the bound on the other side of the clock.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: DATAQ (output port clocked by CLK_STROBE)
  Path Group: CLK_STROBE
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
```

```
UCKBUF0/C (CKB  )                          0.06       0.10 r
UFF0/CK (DF  )                             0.00       0.10 r
UFF0/Q (DF  )                              0.13       0.23 f
UBUF1/Z (BUFF  )                           0.25       0.48 f
DATAQ (out)                                0.00       0.48 f
data arrival time                                     0.48

clock CLK_STROBE (rise edge)               0.00       0.00
clock CLKM (source latency)                0.00       0.00
CLKM (in)                                  0.00       0.00 r
UINV0/ZN (INV  )                           0.01       0.01 f
UINV1/ZN (INV  )                           0.03       0.04 r
UCKBUF1/C (CKB  )                          0.05       0.09 r
CLK_STROBE (out)                           0.00       0.09 r
clock uncertainty                          0.05       0.14
output external delay                     -2.00      -1.86
data required time                                   -1.86
-------------------------------------------------------------
data required time                                   -1.86
data arrival time                                    -0.48
-------------------------------------------------------------
slack (MET)                                           2.35
```

With the min path analysis, *DATAQ* arrives at 0.48ns while the *CLK_STROBE* arrives at 0.09ns. Since the requirement is that data can change up to a 2ns limit before *CLK_STROBE*, we get a slack of 2.35ns after accounting for the clock uncertainty of 50ps.

Another example of a source synchronous interface is depicted in Figure 9-6. In this case, the output clock is a divide-by-2 of the main clock and is part of the synchronous interface with the data. The *POUT* is constrained to switch no earlier than 2ns before and no later than 1ns after the *QCLKOUT*.

Here are the constraints.

```
create_clock -name CLKM -period 6 [get_ports CLKM}
create_generated_clock -name QCLKOUT -source CLKM \
  -divide_by 2 [get_ports QCLKOUT]

set_multicycle_path 0 -setup -to [get_ports POUT]
```

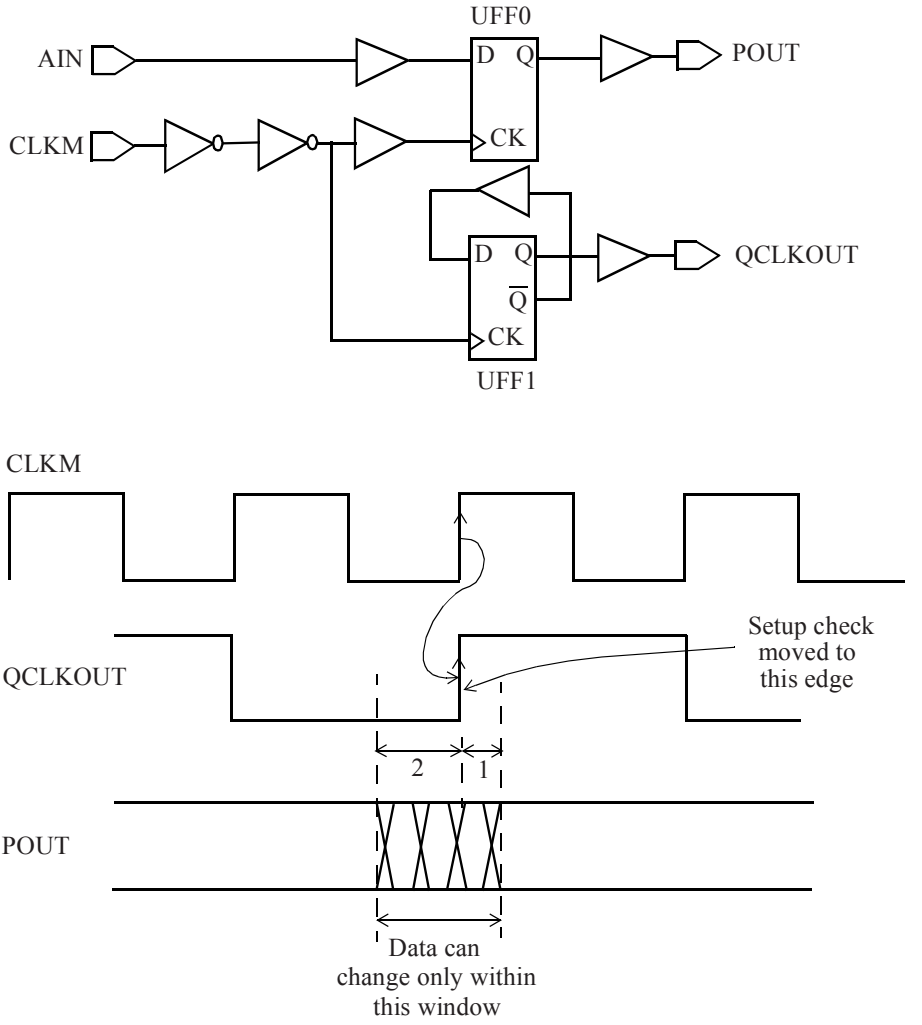


Figure 9-6 *Data change allowed only around divide-by-2 output clock.*

```
set_multicycle_path -1 -hold -to [get_ports POUT]

set_output_delay -max -1.0 -clock QCLKOUT [get_ports POUT]
set_output_delay -min +2.0 -clock QCLKOUT [get_ports POUT]
```

Here is the setup timing report.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: POUT (output port clocked by QCLKOUT)
  Path Group: QCLKOUT
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF0/C (CKB  )                        0.06       0.10 r
  UFF0/CK (DF  )                           0.00       0.10 r
  UFF0/Q (DF  )                            0.13       0.23 r
  UBUF1/Z (BUFF  )                         0.38       0.61 r
  POUT (out)                               0.00       0.61 r
  data arrival time                                   0.61

  clock QCLKOUT (rise edge)                0.00       0.00
  clock CLKM (source latency)              0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF1/C (CKB  )                        0.06       0.10 r
  UFF1/Q (DF  )                            0.13       0.23 r
  UBUF2/Z (BUFF  )                         0.04       0.27 r
  QCLKOUT (out)                            0.00       0.27 r
  clock uncertainty                       -0.30      -0.03
  output external delay                    1.00       0.97
  data required time                                  0.97
  -----------------------------------------------------------
  data required time                                  0.97
  data arrival time                                  -0.61
```

```
  ------------------------------------------------------------
  slack (MET)                                        0.36
```

Notice that the multicycle specification has moved the setup check one cycle back so that the check is performed on the same clock edge. Output *POUT* changes at 0.61ns while the clock *QCLKOUT* changes at 0.27ns. Given the requirement of changing within 1ns, and considering the clock uncertainty of 0.30ns, we get a slack of 0.36ns.

Here is the hold path report that checks for the other constraint on the switching window.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: POUT (output port clocked by QCLKOUT)
  Path Group: QCLKOUT
  Path Type: min

  Point                                    Incr       Path
  ------------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF0/C (CKB  )                        0.06       0.10 r
  UFF0/CK (DF  )                           0.00       0.10 r
  UFF0/Q (DF  )                            0.13       0.23 f
  UBUF1/Z (BUFF  )                         0.25       0.48 f
  POUT (out)                               0.00       0.48 f
  data arrival time                                   0.48

  clock QCLKOUT (rise edge)                0.00       0.00
  clock CLKM (source latency)              0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.03       0.04 r
  UCKBUF1/C (CKB  )                        0.06       0.10 r
  UFF1/Q (DF  )                            0.13       0.23 r
  UBUF2/Z (BUFF  )                         0.04       0.27 r
  QCLKOUT (out)                            0.00       0.27 r
```

```
clock uncertainty                          0.05       0.32
output external delay                     -2.00      -1.68
data required time                                   -1.68
-------------------------------------------------------------
data required time                                   -1.68
data arrival time                                    -0.48
-------------------------------------------------------------
slack (MET)                                           2.17
```

The path report shows that the data changes within the allowable window of 2ns before the *QCLKOUT* clock edge and there is a slack of 2.17ns.

9.2 SRAM Interface

All data transfers in an SRAM interface occur only on the active edge of the clock. All signals are latched by the SRAM or launched by the SRAM on the active clock edge only. The signals comprising the SRAM interface include the command, address and control output bus (CAC), the bidirectional data bus (DQ) and the clock. In the write cycle, the DUA writes data to the SRAM, the data and address go from the DUA to the SRAM, and are latched in the SRAM on the active clock edge. In the read cycle, the address signals still go from the DUA to the SRAM while the data signals output from the SRAM go to the DUA. The address and control are thus unidirectional and go from the DUA to the SRAM as shown in Figure 9-7. A DLL (delay-locked loop[1]) is typically placed in the clock path. The DLL allows the clock to be delayed, if necessary, to account for delay variations of the various signals across the interface due to PVT tolerances and other external variations. By accounting for such variations, there is a good timing margin for the data transfer for both the read cycle and the write cycle to and from the SRAM.

Figure 9-8 shows the AC characteristics of a typical SRAM interface. Note that the *Data in* and *Data out* in Figure 9-8 refer to the direction seen by the

1. See [BES07] in Bibliography.

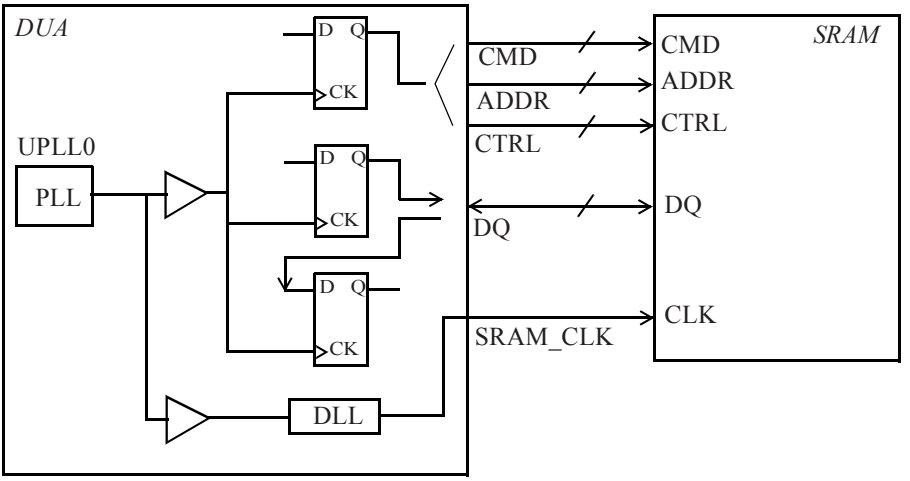


Figure 9-7 *SRAM interface.*

SRAM; the *Data out* from the SRAM is the input to the DUA and the *Data in* to the SRAM is the output of the DUA. These requirements translate to the following IO interface constraints for the DUA interfacing the SRAM.

```
# First define primary clock at the output of UPLL0:
create_clock -name PLL_CLK -period 5 [get_pins UPLL0/CLKOUT]
# Next define a generated clock at clock output pin of DUA:
create_generated_clock -name SRAM_CLK \
  -source [get_pins UPLL0/CLKOUT] -divide_by 1 \
  [get_ports SRAM_CLK]
# Constrain the address and control:
set_output_delay -max 1.5 -clock SRAM_CLK \
  [get_ports ADDR[0]]
set_output_delay -min -0.5 -clock SRAM_CLK \
  [get_ports ADDR[0]]
. . .
```

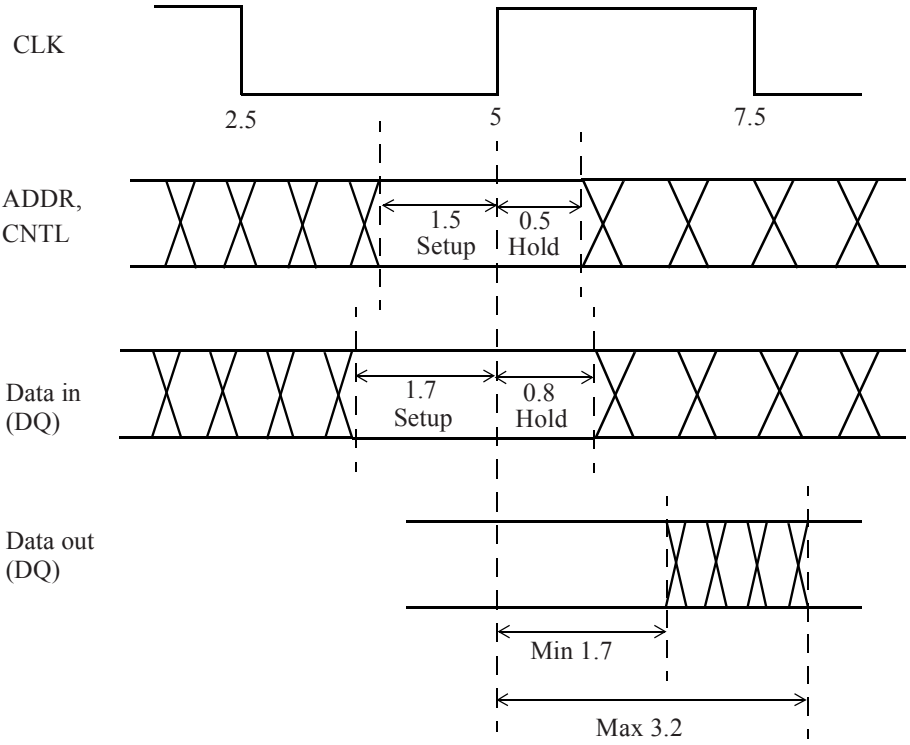


Figure 9-8 *SRAM AC characteristics.*

```
# Constrain the data going out of DUA:
set_output_delay -max 1.7 -clock SRAM_CLK [get_ports DQ[0]]
set_output_delay -min -0.8 -clock SRAM_CLK [get_ports DQ[0]]
# Constrain the data coming into the DUA:
set_input_delay -max 3.2 -clock SRAM_CLK [get_ports DQ[0]]
set_input_delay -min 1.7 -clock SRAM_CLK [get_ports DQ[0]]
. . .
```

Here is a representative setup path report for the address pin.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: ADDR (output port clocked by SRAM_CLK)
  Path Group: SRAM_CLK
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.04       0.05 r
  UCKBUF0/C (CKB  )                        0.06       0.11 r
  UFF0/CP (DF  )                           0.00       0.11 r
  UFF0/Q (DF  )                            0.13       0.24 f
  UBUF1/Z (BUFF  )                         0.05       0.29 f
  ADDR (out)                               0.00       0.29 f
  data arrival time                                   0.29

  clock SRAM_CLK (rise edge)              10.00      10.00
  clock CLKM (source latency)              0.00      10.00
  CLKM (in)                                0.00      10.00 r
  UINV0/ZN (INV  )                         0.01      10.01 f
  UINV1/ZN (INV  )                         0.04      10.05 r
  UCKBUF2/C (CKB  )                        0.05      10.10 r
  SRAM_CLK (out)                           0.00      10.10 r
  clock uncertainty                       -0.30       9.80
  output external delay                   -1.50       8.30
  data required time                                  8.30
  -----------------------------------------------------------
  data required time                                  8.30
  data arrival time                                  -0.29
  -----------------------------------------------------------
  slack (MET)                                         8.01
```

The setup check validates whether the address signal arrives at the memory 1.5ns (setup time of address pin of memory) prior to the *SRAM_CLK* edge.

Here is the hold timing path report for the same pin.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: ADDR (output port clocked by SRAM_CLK)
  Path Group: SRAM_CLK
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.04       0.05 r
  UCKBUF0/C (CKB  )                        0.06       0.11 r
  UFF0/CP (DF  )                           0.00       0.11 r
  UFF0/Q (DF  )                            0.13       0.24 r
  UBUF1/Z (BUFF  )                         0.04       0.28 r
  ADDR (out)                               0.00       0.28 r
  data arrival time                                   0.28

  clock SRAM_CLK (rise edge)               0.00       0.00
  clock CLKM (source latency)              0.00       0.00
  CLKM (in)                                0.00       0.00 r
  UINV0/ZN (INV  )                         0.01       0.01 f
  UINV1/ZN (INV  )                         0.04       0.05 r
  UCKBUF2/C (CKB  )                        0.05       0.10 r
  SRAM_CLK (out)                           0.00       0.10 r
  clock uncertainty                        0.05       0.15
  output external delay                    0.50       0.65
  data required time                                  0.65
  -----------------------------------------------------------
  data required time                                  0.65
  data arrival time                                  -0.28
  -----------------------------------------------------------
  slack (VIOLATED)                                   -0.37
```

The hold checks validate whether the address remains stable for 0.5ns after the clock edge.

9.3 DDR SDRAM Interface

The DDR SDRAM interface can be considered as an extension of the SRAM interface described in the previous section. Just like the SRAM interface, there are two main buses. Figure 9-9 illustrates the bus connectivity and the bus directions between the DUA and the SDRAM. The first bus, which consists of command, address, and control pins (often called CAC) uses the standard scheme of sending information out on one clock edge of a memory clock (or once per clock cycle). The two bidirectional buses consist of the *DQ*, the data bus, and *DQS*, the data strobe. The main differentiation of the DDR interface is the bidirectional data strobe *DQS*. A *DQS* strobe is provided for a group of data signals. This allows the data signals (one per byte or one per half-byte) to have tightly matched timing with the strobe; such a tight match may not be feasible with the clock signal if the clock is common for the entire data bus. The bidirectional strobe signal *DQS* is used for both the read and the write operations. The strobe is used to capture the data on both its edges (both falling and rising edges or Double Data Rate). The *DQ* bus is source synchronous to the data strobe *DQS* (instead of a memory clock) during the read mode of SDRAM, that is, the *DQ* and *DQS* are aligned with each other when they are sent out from the SDRAM. In the other direction, that is when the DUA is sending the data, the *DQS* is phase shifted by 90 degrees. Note that both the data *DQ* and the strobe *DQS* edges are derived from the memory clock inside the DUA.

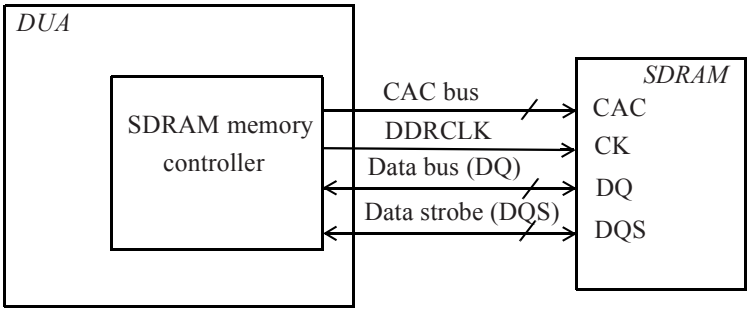


Figure 9-9 *DDR SDRAM interface.*

As described above, there is one data strobe *DQS* for a group of *DQ* signals (four or eight bits). This is done to make the skew balancing requirements between all bits of *DQ* and *DQS* easier. For example, with one *DQS* for a byte, a group of nine signals (eight *DQs* and one *DQS*) needs to be balanced, which is much easier than balancing a 72-bit data bus with the clock.

The above description is not a complete explanation of a DDR SDRAM interface, though sufficient to explain the timing requirements of such an interface.

Figure 9-10 shows the AC characteristics of the CAC bus (at the DUA) for a typical DDR SDRAM interface. These setup and hold requirements map into the following interface constraints for the CAC bus.

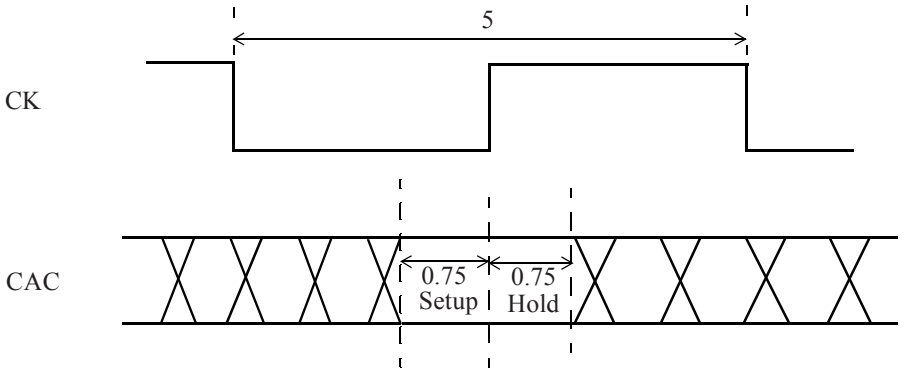


Figure 9-10 *AC characteristics of CAC signals for a typical DDR SDRAM interface.*

```
# DDRCLK is typically a generated clock of the PLL
# clock internal to DUA:
create_generated_clock -name DDRCLK \
  -source [get_pins UPLL0/CLKOUT]\
  -divide_by 1 [get_ports DDRCLK]
```

```
# Set output constraints for each bit of CAC:
set_output_delay -max 0.75 -clock DDRCLK [get_ports CAC]
set_output_delay -min -0.75 -clock DDRCLK [get_ports CAC]
```

The address bus may drive a much greater load than the clock for some scenarios especially when interfacing with unbuffered memory modules. In such cases, the address signals have a larger delay to the memory than the clock signals and this delay difference may result in different AC specifications than the ones depicted in Figure 9-10.

The alignment of *DQS* and *DQ* is different for read and write cycles. This is explored further in the following subsections.

9.3.1 Read Cycle

In a read cycle, the data output from the memory is edge-aligned to *DQS*. Figure 9-11 shows the waveforms; *DQ* and *DQS* in the figure represent the signals at the memory pins. Data (*DQ*) is sent out by the memory on each edge of *DQS* and *DQ* transitions are edge-aligned to the falling and rising edges of *DQS*.

Since the *DQS* strobe signal and the *DQ* data signals are nominally aligned with each other, a DLL (or any alternate method to achieve quarter-cycle delay) is typically used by the memory controller inside the DUA to delay the *DQS* and thus align the delayed *DQS* edge with the center of the data valid window.

Even though the *DQ* and *DQS* are nominally aligned at the memory, the *DQ* and *DQS* strobe signals may no longer be aligned at the memory controller inside the DUA. This can be due to differences in delays between IO buffers and due to factors such as differences in PCB interconnect traces.

Figure 9-12 shows the basic read schematic. The positive edge-triggered flip-flop captures the data *DQ* on the rising edge of *DQS_DLL*, while the negative edge-triggered flip-flop captures the data *DQ* on the falling edge of *DQS_DLL*. While a DLL is not depicted on the *DQ* path, some designs may contain a DLL on the data path also. This allows delaying of the sig-

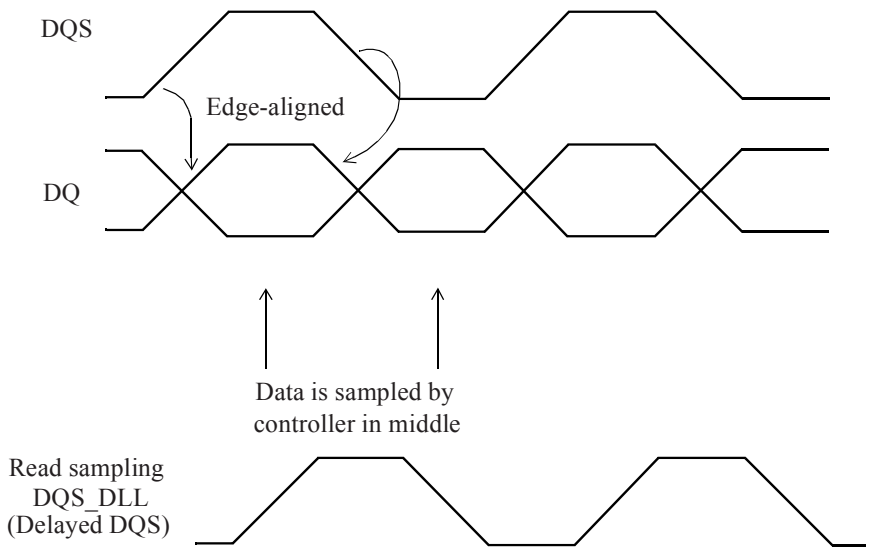


Figure 9-11 *DQS and DQ signals at the memory pins during DDR read cycle.*

nals (to account for variations due to PVT or interconnect trace length or other differences) so that the data can be sampled exactly in the middle of the data valid window.

To constrain the read interface on the controller, a clock is defined on *DQS*, and input delays are specified on the data with respect to the clock.

```
create_clock -period 5 -name DQS [get_ports DQS]
```

This assumes that the memory read interface operates at 200 MHz (equivalent to 400 Mbps as data is transferred on both clock edges), and corresponds to the *DQ* signals being sampled every 2.5ns. Since the data is being captured on both edges, input constraints need to be specified for each edge explicitly.

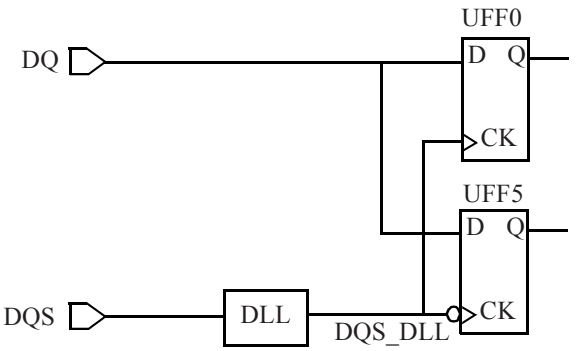


Figure 9-12 *Read capture logic in memory controller.*

```
# For rising clock edge:
set_input_delay 0.4 -max -clock DQS [get_ports DQ]
set_input_delay -0.4 -min -clock DQS [get_ports DQ]
# This is with respect to clock rising edge (default).

# Similarly for falling edge:
set_input_delay 0.35 -max -clock DQS -clock_fall \
  [get_ports DQ]
set_input_delay -0.35 -min -clock DQS -clock_fall \
  [get_ports DQ]
# The launch and capture are on the same edge:
set_multicycle_path 0 -setup -to UFF0/D
set_multicycle_path 0 -setup -to UFF5/D
```

The input delays represent the difference between the *DQ* and the *DQS* edges at the pins of the DUA. Even though these are normally launched together from the memory, there is a tolerance in the timing based upon the memory specification. Thus, the controller design within the DUA should consider that both signals can have skew with respect to each other. Here are the setup path reports to the two flip-flops. Assume that the setup requirement of the capturing flip-flop is 0.05ns and the hold requirement is

0.03ns. The DLL delay is assumed to be set to 1.25ns, a quarter of the cycle period.

```
Startpoint: DQ (input port clocked by DQS)
Endpoint: UFF0 (rising edge-triggered flip-flop clocked by DQS)
Path Group: DQS
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock DQS (rise edge)                          0.00       0.00
clock network delay (propagated)               0.00       0.00
input external delay                           0.40       0.40 f
DQ (in)                                        0.00       0.40 f
UFF0/D (DF  )                                  0.00       0.40 f
data arrival time                                         0.40

clock DQS (rise edge)                          0.00       0.00
clock source latency                           0.00       0.00
DQS (in)                                       0.00       0.00 r
UDLL0/Z (DLL   )                               1.25       1.25 r
UFF0/CP (DF  )                                 0.00       1.25 r
library setup time                            -0.05       1.20
data required time                                       1.20
-----------------------------------------------------------------
data required time                                       1.20
data arrival time                                       -0.40
-----------------------------------------------------------------
slack (MET)                                              0.80


Startpoint: DQ (input port clocked by DQS)
Endpoint: UFF5 (falling edge-triggered flip-flop clocked by DQS)
Path Group: DQS
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock DQS (fall edge)                          2.50       2.50
clock network delay (propagated)               0.00       2.50
input external delay                           0.35       2.85 r
DQ (in)                                        0.00       2.85 r
```

```
UFF5/D (DFN  )                          0.00       2.85 r
data arrival time                                  2.85

clock DQS (fall edge)                   2.50       2.50
clock source latency                    0.00       2.50
DQS (in)                                0.00       2.50 f
UDLL0/Z (DLL   )                        1.25       3.76 f
UFF5/CPN (DFN  )                        0.00       3.76 f
library setup time                     -0.05       3.71
data required time                                 3.71
-----------------------------------------------------------
data required time                                 3.71
data arrival time                                 -2.85
-----------------------------------------------------------
slack (MET)                                        0.86
```

Here are the hold timing reports.

```
Startpoint: DQ (input port clocked by DQS)
Endpoint: UFF0 (rising edge-triggered flip-flop clocked by DQS)
Path Group: DQS
Path Type: min

Point                                   Incr       Path
-----------------------------------------------------------
clock DQS (rise edge)                   5.00       5.00
clock network delay (propagated)        0.00       5.00
input external delay                   -0.40       4.60 r
DQ (in)                                 0.00       4.60 r
UFF0/D (DF  )                           0.00       4.60 r
data arrival time                                  4.60

clock DQS (fall edge)                   2.50       2.50
clock source latency                    0.00       2.50
DQS (in)                                0.00       2.50 f
UDLL0/Z (DLL   )                        1.25       3.75 f
UFF0/CP (DF  )                          0.00       3.75 f
library hold time                       0.03       3.78
data required time                                 3.78
-----------------------------------------------------------
data required time                                 3.78
```

```
data arrival time                                      -4.60
-------------------------------------------------------------
slack (MET)                                             0.82


Startpoint: DQ (input port clocked by DQS)
Endpoint: UFF5 (falling edge-triggered flip-flop clocked by DQS)
Path Group: DQS
Path Type: min

Point                                    Incr       Path
-------------------------------------------------------------
clock DQS (fall edge)                    2.50       2.50
clock network delay (propagated)         0.00       2.50
input external delay                    -0.35       2.15 f
DQ (in)                                  0.00       2.15 f
UFF5/D (DFN  )                           0.00       2.15 f
data arrival time                                   2.15

clock DQS (rise edge)                    0.00       0.00
clock source latency                     0.00       0.00
DQS (in)                                 0.00       0.00 r
UDLL0/Z (DLL   )                         1.25       1.25 r
UFF5/CPN (DFN  )                         0.00       1.25 r
library hold time                        0.03       1.28
data required time                                  1.28
-------------------------------------------------------------
data required time                                  1.28
data arrival time                                  -2.15
-------------------------------------------------------------
slack (MET)                                         0.87
```

9.3.2 Write Cycle

In a write cycle, the *DQS* edges are quarter-cycle offset from the *DQ* signals coming out of the memory controller within the DUA so that the *DQS* strobe can be used to capture the data at the memory.

Figure 9-13 shows the required waveforms at the memory pins. The *DQS* signal must be aligned to be at the center of the *DQ* window at the memory

pins. Note that aligning *DQ* and *DQS* to have the same property at the memory controller (inside DUA) is not enough for these signals to have the required alignment at the SDRAM pins due to mismatch in the IO buffer delays or variations in PCB interconnect traces. Thus, the DUA typically provides additional DLL controls in the write cycle to achieve the required quarter cycle offset between the *DQS* and the *DQ* signals.

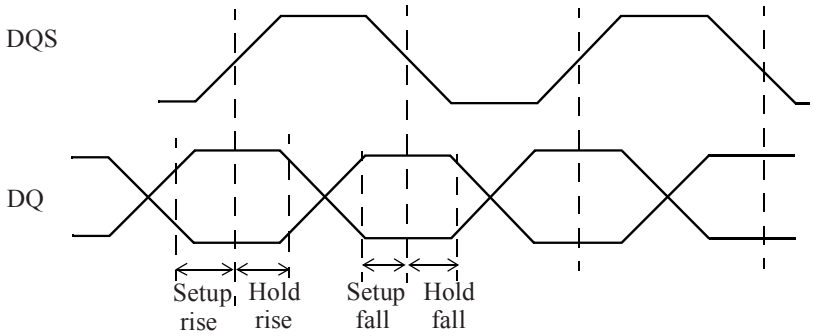


Figure 9-13 *DQ and DQS signals at memory pins during DDR write cycle.*

Constraining the outputs for this mode depends on how the clocks are generated in the controller. We consider two cases.

Case 1: Internal 2x Clock

If an internal clock that is double the frequency of the DDR clock is available, the output logic can be similar to the one shown in Figure 9-14. The DLL provides a mechanism to skew the *DQS* clock if necessary so that the setup and hold requirements at the memory pins are met. In some cases, the DLL may not be used - instead a negative edge flip-flop is used to get the 90 degree offset.

For the scenario shown in Figure 9-14, the outputs can be constrained as:

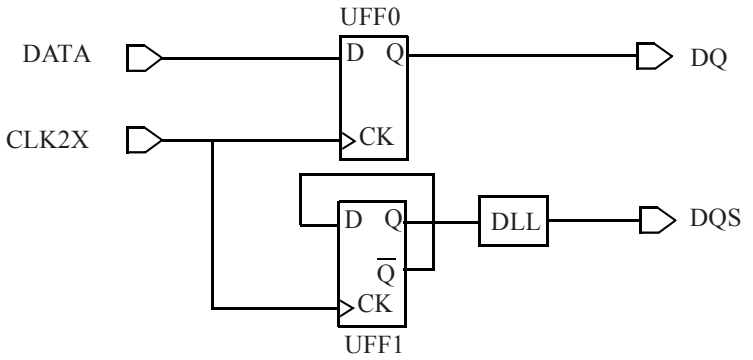


Figure 9-14 *DQS is a divide-by-2 of internal clock.*

```
# 166MHz (333Mbps) DDR; 2x clock is at 333MHz:
create_clock -period 3 [get_ports CLK2X]

# Define a 1x generated clock at the output of flip-flop:
create_generated_clock -name pre_DQS -source CLK2X \
  -divide_by 2 [get_pins UFF1/Q]
# Create the delayed version as DQS assuming 1.5ns DLL delay:
create_generated_clock -name DQS -source UFF1/Q \
  -edges {1 2 3} -edge_shift {1.5 1.5 1.5} [get_ports DQS]
```

The timing at the *DQ* output pins has to be constrained with respect to the generated clock *DQS*.

Assume that the setup requirements between the *DQ* and *DQS* pins at the DDR SDRAM are 0.25ns and 0.4ns for the rising edge and falling edge of *DQ* respectively. Similarly, a hold requirement of 0.15ns and 0.2ns is assumed for the rising and falling edges of *DQ* pins. The DLL delay at the *DQS* output has been set to a quarter-cycle, which is 1.5ns. The waveforms are shown in Figure 9-15.

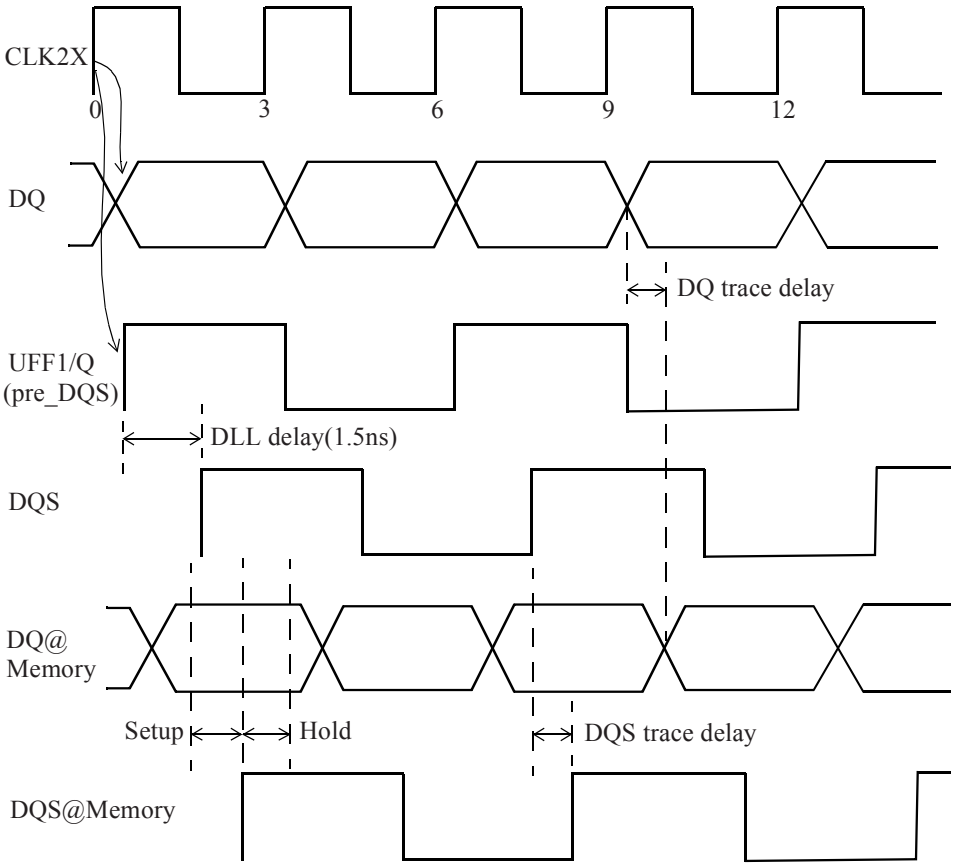


Figure 9-15 *DQS and DQ signals obtained from internal 2x clock.*

```
set_output_delay -clock DQS -max 0.25 -rise [get_ports DQ]
# Default above is rising clock.
set_output_delay -clock DQS -max 0.4 -fall [get_ports DQ]
# If setup requirements are different for falling edge of DQS,
# that can be specified by using the -clock_fall option.
```

```
set_output_delay -clock DQS -min -0.15 -rise DQ
set_output_delay -clock DQS -min -0.2 -fall DQ
```

Here is the setup report through the output *DQ*. The setup check is from the rise edge of *CLK2X* at 0ns which launches *DQ* to the rise edge of *DQS* at 1.5ns.

```
Startpoint: UFF0(rising edge-triggered flip-flop clocked by CLK2X)
  Endpoint: DQ (output port clocked by DQS)
  Path Group: DQS
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLK2X (rise edge)                  0.00       0.00
  clock source latency                     0.00       0.00
  CLK2X (in)                               0.00       0.00 r
  UFF0/CP (DFD1)                           0.00       0.00 r
  UFF0/Q (DFD1)                            0.12       0.12 f
  DQ (out)                                 0.00       0.12 f
  data arrival time                                   0.12

  clock DQS (rise edge)                    1.50       1.50
  clock CLK2X (source latency)             0.00       1.50
  CLK2X (in)                               0.00       1.50 r
  UFF1/Q (DFD1) (gclock source)            0.12       1.62 r
  UDLL0/Z (DLL   )                         0.00       1.62 r
  DQS (out)                                0.00       1.62 r
  output external delay                   -0.40       1.22
  data required time                                  1.22
  -----------------------------------------------------------
  data required time                                  1.22
  data arrival time                                  -0.12
  -----------------------------------------------------------
  slack (MET)                                         1.10
```

Note that the quarter-cycle delay in the above report appears in the first line with the *DQS* clock edge rather than on the line showing the DLL instance *UDLL0*. This is because the DLL delay has been modeled as part of

the generated clock definition for *DQS* instead of in the timing arc for the DLL.

Here is the hold report through output *DQ*. The hold check is done from the rising edge of clock *CLK2X* which launches *DQ* at 3ns to the previous rising edge of *DQS* at 1.5ns.

```
Startpoint: UFF0(rising edge-triggered flip-flop clocked by CLK2X)
  Endpoint: DQ (output port clocked by DQS)
  Path Group: DQS
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLK2X (rise edge)                  3.00       3.00
  clock source latency                     0.00       3.00
  CLK2X (in)                               0.00       3.00 r
  UFF0/CP (DFD1)                           0.00       3.00 r
  UFF0/Q (DFD1)                            0.12       3.12 f
  DQ (out)                                 0.00       3.12 f
  data arrival time                                   3.12

  clock DQS (rise edge)                    1.50       1.50
  clock CLK2X (source latency)             0.00       1.50
  CLK2X (in)                               0.00       1.50 r
  UFF1/Q (DFD1) (gclock source)            0.12       1.62 r
  UDLL0/Z (DLL   )                         0.00       1.62 r
  DQS (out)                                0.00       1.62 r
  output external delay                    0.20       1.82
  data required time                                  1.82
  -----------------------------------------------------------
  data required time                                  1.82
  data arrival time                                  -3.12
  -----------------------------------------------------------
  slack (MET)                                         1.30
```

Case 2: Internal 1x Clock

When only an internal 1x clock is available, the output circuitry may typically be similar to that shown in Figure 9-16.

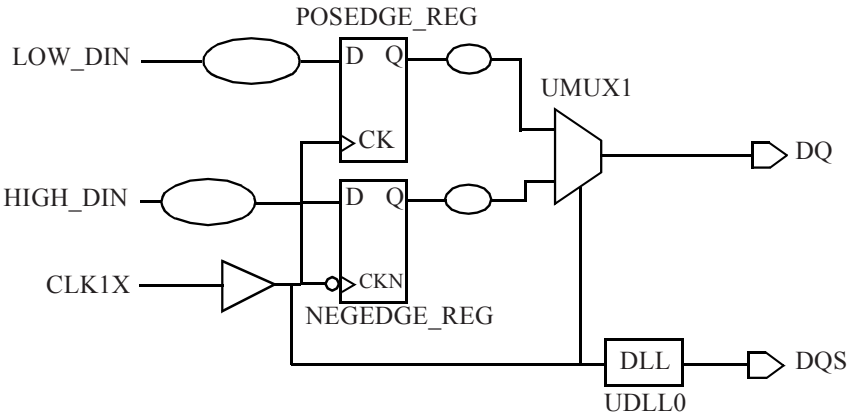


Figure 9-16 *Output logic in DUA using internal 1x clock.*

There are two flip-flops used to generate the *DQ* data. The first flip-flop *NEGEDGE_REG* is triggered by the negative edge of the clock *CLK1X*, and the second flip-flop *POSEDGE_REG* is triggered by the positive edge of the clock *CLK1X*. Each flip-flop latches the appropriate edge data and this data is then multiplexed out using the *CLK1X* as the multiplexer select. When *CLK1X* is high, the output of flip-flop *NEGEDGE_REG* is sent to *DQ*. When *CLK1X* is low, the output of flip-flop *POSEDGE_REG* is sent to *DQ*. Hence, data arrives at the output *DQ* on both edges of clock *CLK1X*. Notice that each flip-flop has half a cycle to propagate data to the input of the multiplexer so that the input data is ready at the multiplexer before it is selected by the *CLK1X* edge. The relevant waveforms are shown in Figure 9-17.

```
# Create the 1x clock:
create_clock -name CLK1X -period 6 [get_ports CLK1X]
```

```
# Define a generated clock at DQS. It is a divide-by-1 of
# CLK1X. Assume a quarter-cycle delay of 1.5ns on UDLL0:
create_generated_clock -name DQS -source CLK1X \
  -edges {1 2 3} -edge_shift {1.5 1.5 1.5} [get_ports DQS]

# Define a setup check of 0.25 and 0.3 between DQ and DQS
# pins on rising and falling edge of clock:
set_output_delay -max 0.25 -clock DQS [get_ports DQ]
set_output_delay -max 0.3 -clock DQS -clock_fall \
  [get_ports DQ]
set_output_delay -min -0.2 -clock DQS [get_ports DQ]
set_output_delay -min -0.27 -clock DQS -clock_fall \
  [get_ports DQ]
```

The setup and hold checks verify the timing from the multiplexer to the output. One of the setup checks is from the rising edge of *CLK1X* at the multiplexer input (which launches the *NEGEDGE_REG* data) to the rising edge of *DQS*. The other setup check is from the falling edge of *CLK1X* at the multiplexer input (which launches the *POSEDGE_REG* data) to the falling edge of *DQS*. Similarly, the hold checks are from the same *CLK1X* edges (as for setup checks) to the previous falling or rising edges of *DQS*.

Here is the setup timing check report through the port *DQ*. The check is between the rising edge of *CLK1X*, which selects the *NEGEDGE_REG* output, and the rising edge of the *DQS*.

```
Startpoint: CLK1X (clock source 'CLK1X')
Endpoint: DQ (output port clocked by DQS)
Path Group: DQS
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock CLK1X (rise edge)                        0.00       0.00
clock source latency                           0.00       0.00
CLK1X (in)                                     0.00       0.00 r
UMUX1/S (MUX2D1) <-                            0.00       0.00 r
UMUX1/Z (MUX2D1)                               0.07       0.07 r
```

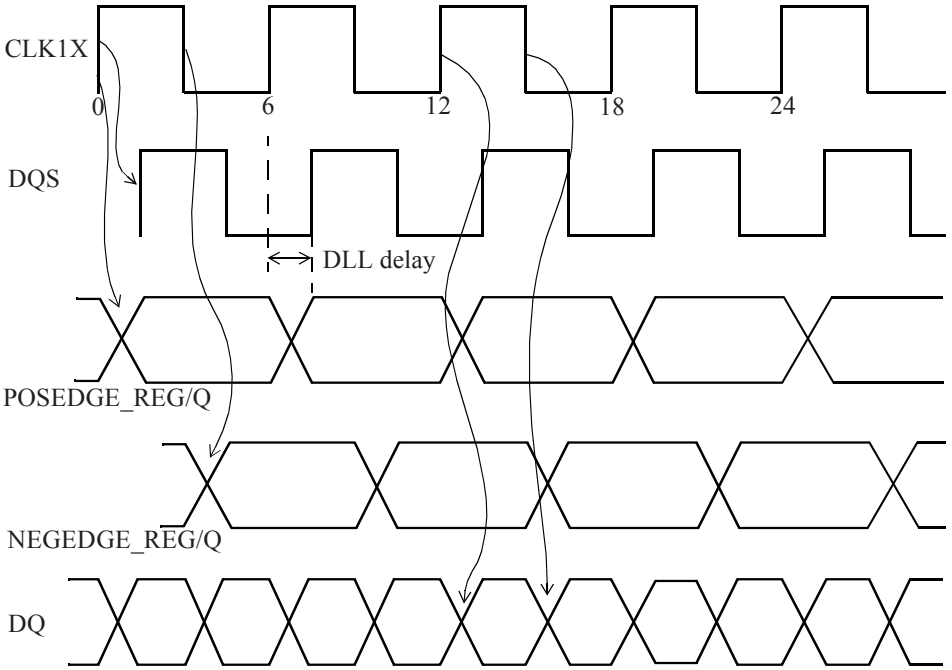


Figure 9-17 *DQS and DQ signals obtained using 1x internal clock.*

```
DQ (out)                                    0.00       0.07 r
data arrival time                                      0.07

clock DQS (rise edge)                       1.50       1.50
clock CLK1X (source latency)                0.00       1.50
CLK1X (in)                                  0.00       1.50 r
UBUF0/Z (BUFFD1)                            0.03       1.53 r
UDLL0/Z (DLL )                              0.00       1.53 r
DQS (out)                                   0.00       1.53 r
output external delay                      -0.25       1.28
data required time                                     1.28
-----------------------------------------------------------------
data required time                                     1.28
data arrival time                                     -0.07
-----------------------------------------------------------------
```

```
slack (MET)                                              1.21
```

Here is another setup timing check report through the port *DQ*. This setup check is between the falling edge of *CLK1X* which selects the *POSEDGE_REG* output and the falling edge of the *DQS*.

```
Startpoint: CLK1X (clock source 'CLK1X')
Endpoint: DQ (output port clocked by DQS)
Path Group: DQS
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock CLK1X (fall edge)                        3.00       3.00
clock source latency                           0.00       3.00
CLK1X (in)                                     0.00       3.00 f
UMUX1/S (MUX2D1) <-                            0.00       3.00 f
UMUX1/Z (MUX2D1)                               0.05       3.05 f
DQ (out)                                       0.00       3.05 f
data arrival time                                         3.05

clock DQS (fall edge)                          4.50       4.50
clock CLK1X (source latency)                   0.00       4.50
CLK1X (in)                                     0.00       4.50 f
UBUF0/Z (BUFFD1)                               0.04       4.54 f
UDLL0/Z (DLL   )                               0.00       4.54 f
DQS (out)                                      0.00       4.54 f
output external delay                         -0.30       4.24
data required time                                        4.24
-----------------------------------------------------------------
data required time                                        4.24
data arrival time                                        -3.05
-----------------------------------------------------------------
slack (MET)                                               1.19
```

Here is the hold timing check report through the port *DQ*. The check is between the rising edge of *CLK1X* and the previous falling edge of *DQS*.

```
Startpoint: CLK1X (clock source 'CLK1X')
Endpoint: DQ (output port clocked by DQS)
Path Group: DQS
Path Type: min

Point                                       Incr       Path
-----------------------------------------------------------
clock CLK1X (rise edge)                     6.00       6.00
clock source latency                        0.00       6.00
CLK1X (in)                                  0.00       6.00 r
UMUX1/S (MUX2D1) <-                         0.00       6.00 r
UMUX1/Z (MUX2D1)                            0.05       6.05 f
DQ (out)                                    0.00       6.05 f
data arrival time                                      6.05

clock DQS (fall edge)                       4.50       4.50
clock CLK1X (source latency)                0.00       4.50
CLK1X (in)                                  0.00       4.50 f
UBUF0/Z (BUFFD1)                            0.04       4.54 f
UDLL0/Z (DLL   )                            0.00       4.54 f
DQS (out)                                   0.00       4.54 f
output external delay                       0.27       4.81
data required time                                     4.81
-----------------------------------------------------------
data required time                                     4.81
data arrival time                                     -6.05
-----------------------------------------------------------
slack (MET)                                            1.24
```

Here is another hold timing check report through the port *DQ*. This check is between the falling edge of *CLK1X* and the previous rising edge of *DQS*.

```
Startpoint: CLK1X (clock source 'CLK1X')
Endpoint: DQ (output port clocked by DQS)
Path Group: DQS
Path Type: min
```

```
Point                                          Incr       Path
-------------------------------------------------------------
clock CLK1X (fall edge)                        3.00       3.00
clock source latency                           0.00       3.00
CLK1X (in)                                     0.00       3.00 f
UMUX1/S (MUX2D1) <-                            0.00       3.00 f
UMUX1/Z (MUX2D1)                               0.05       3.05 f
DQ (out)                                       0.00       3.05 f
data arrival time                                         3.05

clock DQS (rise edge)                          1.50       1.50
clock CLK1X (source latency)                   0.00       1.50
CLK1X (in)                                     0.00       1.50 r
UBUF0/Z (BUFFD1)                               0.03       1.53 r
UDLL0/Z (DLL   )                               0.00       1.53 r
DQS (out)                                      0.00       1.53 r
output external delay                          0.20       1.73
data required time                                        1.73
-------------------------------------------------------------
data required time                                        1.73
data arrival time                                        -3.05
-------------------------------------------------------------
slack (MET)                                               1.32
```

While the above interface timing analysis has ignored the effect of any loads on the outputs, additional load can be specified (using *set_load*) for more accuracy. However, STA can be supplemented with circuit simulation for achieving a robust DRAM timing as described below.

The *DQ* and the *DQS* signals for the DDR interface typically use ODT[1] in read and write modes to reduce any reflections due to impedance mismatch at the DRAM and at the DUA. The timing models used for STA are not able to provide adequate accuracy in presence of ODT termination. The designer may use an alternate mechanism such as detailed circuit level simulation to validate the signal integrity and the timing of the DRAM interface.

1. On-Die Termination.

9.4 Interface to a Video DAC

Consider Figure 9-18 which shows a typical DAC[1] interface where a high-speed clock is transferring data to the slow-speed clock interface of the DAC.

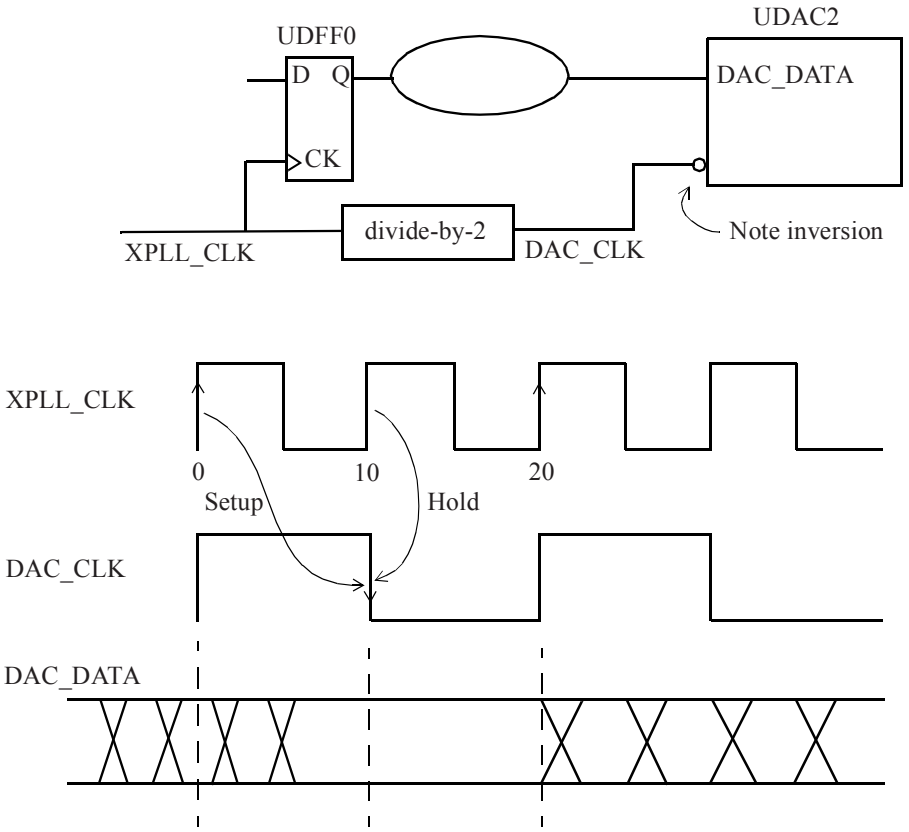


Figure 9-18 *Video DAC interface.*

1. Digital to Analog Converter.

Clock *DAC_CLK* is a divide-by-2 of the clock *XPLL_CLK*. The DAC setup and hold checks are with respect to the falling edge of *DAC_CLK*.

In this case, the setup time is considered as a single cycle (*XPLL_CLK*) path, even though the interface from a faster clock domain to a slower clock domain can be specified as a multicycle path if necessary. As shown in Figure 9-18, the rising edge of *XPLL_CLK* launches the data and the falling edge of *DAC_CLK* captures the data. Here is the setup report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by XPLL_CLK)
Endpoint: UDAC2
  (falling edge-triggered flip-flop clocked by DAC_CLK)
Path Group: DAC_CLK
Path Type: max

Point                                        Incr       Path
-------------------------------------------------------------
clock XPLL_CLK (rise edge)                   0.00       0.00
clock source latency                         0.00       0.00
XPLL_CLK (in)                                0.00       0.00 r
UDFF0/CK (DF  )                              0.00       0.00 r
UDFF0/Q (DF  )                               0.12       0.12 f
UBUF0/Z (BUFF  )                             0.06       0.18 f
UDFF2/D (DFN  )                              0.00       0.18 f
data arrival time                                       0.18

clock DAC_CLK (fall edge)                   10.00      10.00
clock XPLL_CLK (source latency)              0.00      10.00
XPLL_CLK (in)                                0.00      10.00 r
UDFF1/Q (DF  ) (gclock source)               0.13      10.13 f
UDAC2/CKN ( DAC )                            0.00      10.13 f
library setup time                          -0.04      10.08
data required time                                     10.08
-------------------------------------------------------------
data required time                                     10.08
data arrival time                                      -0.18
-------------------------------------------------------------
slack (MET)                                             9.90
```

Note that the interface is from a faster clock to a slower clock, thus it can be made into a two-cycle path if necessary.

Here is the hold timing report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by XPLL_CLK)
Endpoint: UDAC2
  (falling edge-triggered flip-flop clocked by DAC_CLK)
Path Group: DAC_CLK
Path Type: min

Point                                    Incr       Path
-------------------------------------------------------------
clock XPLL_CLK (rise edge)              10.00      10.00
clock source latency                     0.00      10.00
XPLL_CLK (in)                            0.00      10.00 r
UDFF0/CK (DF  )                          0.00      10.00 r
UDFF0/Q (DF  )                           0.12      10.12 r
UBUF0/Z (BUFF  )                         0.05      10.17 r
UDFF2/D (DFN  )                          0.00      10.17 r
data arrival time                                  10.17

clock DAC_CLK (fall edge)               10.00      10.00
clock XPLL_CLK (source latency)          0.00      10.00
XPLL_CLK (in)                            0.00      10.00 r
UDFF1/Q (DF  ) (gclock source)           0.13      10.13 f
UDAC2/CKN ( DAC )                        0.00      10.13 f
library hold time                        0.03      10.16
data required time                                 10.16
-------------------------------------------------------------
data required time                                 10.16
data arrival time                                 -10.17
-------------------------------------------------------------
slack (MET)                                         0.01
```

The hold check is done one cycle prior to the setup capture edge. In this case, the most critical hold check is the one where the setup and launch edges are the same, and this is shown in the hold timing report.

❑

CHAPTER

10

Robust Verification

This chapter describes special STA analyses such as time borrowing, clock gating and non-sequential timing checks. In addition, advanced STA concepts such as on-chip variations, statistical timing and trade-off between power and timing are also presented.

10.1 On-Chip Variations

In general, the process and environmental parameters may not be uniform across different portions of the die. Due to process variations, identical MOS transistors in different portions of the die may not have similar characteristics. These differences are due to process variations within the die. Note that the process parameter variations across multiple manufactured lots can cover the entire span of process models from *slow* to *fast* (Section

J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*,
DOI: 10.1007/978-0-387-93820-2_10, © Springer Science + Business Media, LLC 2009

2.10). In this section, we discuss the analysis of the process variations possible on one die (called *local* process variations) which are much smaller than the variations across multiple manufacturing lots (called *global* process variations).

Besides the variations in the process parameters, different portions of the design may also see different power supply voltage and temperature. It is therefore possible that two regions of the same chip are not at identical PVT conditions. These differences can arise due to many factors, including:

i. IR drop variation along the die area affecting the local power supply.

ii. Voltage threshold variation of the PMOS or the NMOS device.

iii. Channel length variation of the PMOS or the NMOS device.

iv. Temperature variations due to local hot spots.

v. Interconnect metal etch or thickness variations impacting the interconnect resistance or capacitance.

The PVT variations described above are referred to as **O**n-**C**hip **V**ariations (**OCV**) and these variations can affect the wire delays and cell delays in different portions of the chip. As discussed above, modeling of OCV is not intended to model the entire span of the PVT variations possible from wafer to wafer but to model the PVT variations that are possible locally within a single die. The OCV effect is typically more pronounced on clock paths as they travel longer distances in a chip. One way to account for the local PVT variations is to incorporate the OCV analysis during STA. The static timing analysis described in previous chapters obtains the timing at a specific timing corner and does not model the variations along the die. Since the clock and data paths can be affected differently by the OCV, the timing verification can model the OCV effect by making the PVT conditions for the launch and capture paths to be slightly different. The STA can include the OCV effect by derating the delays of specific paths, that is, by making those paths faster or slower and then validating the behavior of the design with these variations. The cell delays or wire delays or both can be derated to model the effect of OCV.

We now examine how the OCV derating is done for a setup check. Consider the logic shown in Figure 10-1 where the PVT conditions can vary along the chip. The worst condition for setup check occurs when the launch clock path and the data path have the OCV conditions which result in the largest delays, while the capture clock path has the OCV conditions which result in the smallest delays. Note that the smallest and largest here are due to local PVT variations on a die.

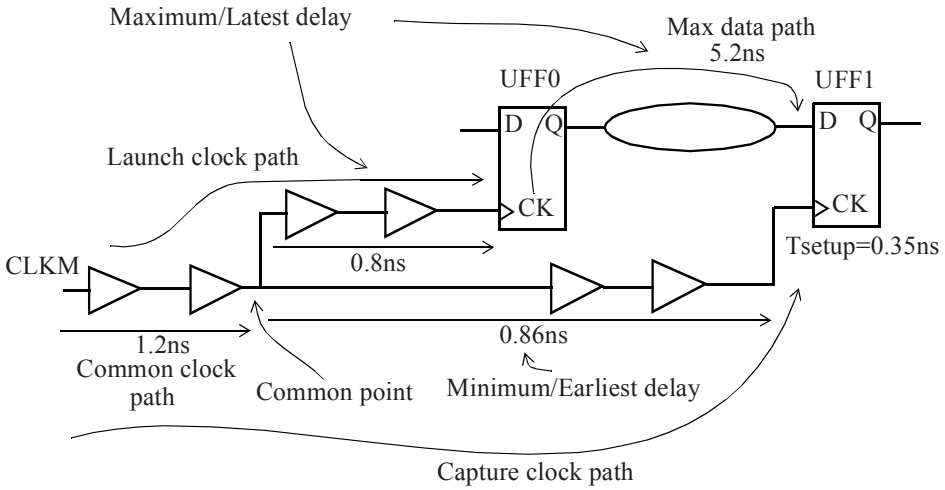


Figure 10-1 *Derating setup timing check for OCV.*

For this example, here is the setup timing check condition; this does not include any OCV setting for derating delays.

```
LaunchClockPath + MaxDataPath <= ClockPeriod +
 CaptureClockPath - Tsetup_UFF1

This implies that the minimum clock period = LaunchClockPath +
 MaxDataPath - CaptureClockPath + Tsetup_UFF1
```

```
From the figure,
  LaunchClockPath = 1.2 + 0.8 = 2.0
  MaxDataPath = 5.2
  CaptureClockPath = 1.2 + 0.86 = 2.06
  Tsetup_UFF1 = 0.35

This results in a minimum clock period of:
  2.0 + 5.2 – 2.06 + 0.35 = 5.49ns
```

The above path delays correspond to the delay values without any OCV derating. Cell and net delays can be derated using the **set_timing_derate** specification. For example, the commands:

```
set_timing_derate -early 0.8
set_timing_derate -late 1.1
```

derate the minimum/shortest/early paths by -20% and derate the maximum/longest/latest paths by +10%. Long path delays (for example, data paths and launch clock path for setup checks or capture clock paths for hold checks) are multiplied by the derate value specified using the *-late* option, and short path delays (for example, capture clock paths for setup checks or data paths and launch clock paths for hold checks) are multiplied by the derate values specified using the *-early* option. If no derating factors are specified, a value of 1.0 is assumed.

The derating factors apply uniformly to all net delays and cell delays. If an application scenario warrants different derating factors for cells and nets, the *-cell_delay* and the *-net_delay* options can be used in the *set_timing_derate* specification.

```
# Derate only the cell delays - early paths by -10%, and
# no derate on the late paths:
set_timing_derate -cell_delay -early 0.9
set_timing_derate -cell_delay -late 1.0
```

```
# Derate only the net delays - no derate on the early paths
# and derate the late paths by +20%:
set_timing_derate -net_delay -early 1.0
set_timing_derate -net_delay -late 1.2
```

Cell check delays, such as setup and hold of a cell, can be derated by using the *-cell_check* option. With this option, any output delay specified using *set_output_delay* is also derated as this specification is part of the setup requirement of that output. However, no such implicit derating is applied for the input delays specified using the *set_input_delay* specification.

```
# Derate the cell timing check values:
set_timing_derate -early 0.8 -cell_check
set_timing_derate -late 1.1 -cell_check

# Derate the early clock paths:
set_timing_derate -early 0.95 -clock
# Derate the late data paths:
set_timing_derate -late 1.05 -data
```

The *-clock* option (shown above) applies derating only to clock paths. Similarly, the *-data* option applies derating only to data paths.

We now apply the following derating to the example of Figure 10-1.

```
set_timing_derate -early 0.9
set_timing_derate -late 1.2
set_timing_derate -late 1.1 -cell_check

With these derating values, we get the following for setup
check:
 LaunchClockPath = 2.0 * 1.2 = 2.4
 MaxDataPath = 5.2 * 1.2 = 6.24
 CaptureClockPath = 2.06 * 0.9 = 1.854
 Tsetup_UFF1 = 0.35 * 1.1 = 0.385
```

```
This results in a minimum clock period of:
2.4 + 6.24 – 1.854 + 0.385 = 7.171ns
```

In the setup check above, there is a discrepancy since the *common clock path (*Figure 10-1) of the clock tree, with a delay of 1.2ns, is being derated differently for the launch clock and for the capture clock. This part of the clock tree is common to both the launch clock and the capture clock and should not be derated differently. Applying different derating for the launch and capture clock is overly pessimistic as in reality this part of the clock tree will really be at only one PVT condition, either as a maximum path or as a minimum path (or anything in between) but never both at the same time. The pessimism caused by different derating factors applied on the common part of the clock tree is called **Common Path Pessimism** (CPP) which should be removed during the analysis. CPPR, which stands for *Common Path Pessimism Removal*, is often listed as a separate item in a path report. It is also labeled as *Clock Reconvergence Pessimism Removal* (CRPR).

CPPR is the removal of artificially induced pessimism between the launch clock path and the capture clock path in timing analysis. If the same clock drives both the capture and the launch flip-flops, then the clock tree will likely share a common portion before branching. CPP itself is the delay difference along this common portion of the clock tree due to different deratings for launch and capture clock paths. The difference between the minimum and the maximum arrival times of the clock signal at the common point is the CPP. The *common point* is defined as the output pin of the last cell in the common portion of the clock tree.

```
CPP = LatestArrivalTime@CommonPoint –
  EarliestArrivalTime@CommonPoint
```

The *Latest* and *Earliest* times in the above analysis are in reference to the OCV derating at a specific timing corner - for example worst-case slow or best-case fast. For the example of Figure 10-1,

```
LatestArrivalTime@CommonPoint = 1.2 * 1.2 = 1.44
EarliestArrivalTime@CommonPoint = 1.2 * 0.9 = 1.08
```

```
This implies a CPP of: 1.44 - 1.08 = 0.36ns

With the CPP correction, this results in a
minimum clock period of: 7.171 - 0.36 = 6.811ns
```

Applying the OCV derating has increased the minimum clock period from 5.49ns to 6.811ns for this example design. This illustrates that the OCV variations modeled by these derating factors can reduce the maximum frequency of operation of the design.

Analysis with OCV at Worst PVT Condition

If the setup timing check is being performed at the worst-case PVT condition, no derating is necessary on the late paths as they are already the worst possible. However, derating can be applied to the early paths by making those paths faster by using a specific derating, for example, speeding up the early paths by 10%. A derate specification at the worst-case slow corner may be something like:

```
set_timing_derate -early 0.9
set_timing_derate -late 1.0
# Don't derate the late paths as they are already the slowest,
# but derate the early paths to make these faster by 10%.
```

The above derate settings are for max path (or setup) checks at the worst-case slow corner; thus the late path OCV derate setting is kept at 1.0 so as not to slow it beyond the worst-case slow corner.

An example of setup timing check at the worst-case slow corner is described next. The derate specification is specified for the capture clock path below:

```
# Derate the early clock paths:
set_timing_derate -early 0.8 -clock
```

Here is the setup timing check path report performed at the worst-case slow corner. The derating used by the late paths are reported as *Max Data*

Paths Derating Factor and as *Max Clock Paths Derating Factor*. The derating used for the early paths is reported as *Min Clock Paths Derating Factor*.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: max
  Max Data Paths Derating Factor  : 1.000
  Min Clock Paths Derating Factor : 0.800
  Max Clock Paths Derating Factor : 1.000

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                  0.000      0.000
  clock source latency                    0.000      0.000
  CLKM (in)                               0.000      0.000 r
  UCKBUF0/C (CKB  )                       0.056      0.056 r
  UCKBUF1/C (CKB  )                       0.058      0.114 r
  UFF0/CK (DF  )                          0.000      0.114 r
  UFF0/Q (DF  ) <-                        0.143      0.258 f
  UNOR0/ZN (NR2  )                        0.043      0.301 r
  UBUF4/Z (BUFF  )                        0.052      0.352 r
  UFF1/D (DF  )                           0.000      0.352 r
  data arrival time                                  0.352

  clock CLKM (rise edge)                 10.000     10.000
  clock source latency                    0.000     10.000
  CLKM (in)                               0.000     10.000 r
  UCKBUF0/C (CKB  )                       0.045     10.045 r
  UCKBUF2/C (CKB  )                       0.054     10.098 r
  UFF1/CK (DF  )                          0.000     10.098 r
  clock reconvergence pessimism           0.011     10.110
  clock uncertainty                      -0.300      9.810
  library setup time                     -0.044      9.765
  data required time                                 9.765
  -----------------------------------------------------------
  data required time                                 9.765
  data arrival time                                 -0.352
  -----------------------------------------------------------
  slack (MET)                                        9.413
```

Notice that the capture clock path is derated by 20%. See cell *UCKBUF0* in the timing report. In the launch path, it has a delay of 56ps, while it has a derated delay of 45ps in the capture path. The cell *UCKBUF0* is on the common clock path, that is, on both the capture clock path and the launch clock path. Since the common clock path cannot have a different derating, the difference in timing for this common path, 56ps - 45ps = 11ps, is corrected separately. This appears as the line *clock reconvergence pessimism* in the report. In summary, if one were to compare the reports of this path, with and without derating, one would notice that only the cell and net delays for the capture clock path have been derated.

OCV for Hold Checks

We now examine how the derating is done for a hold timing check. Consider the logic shown in Figure 10-2. If the PVT conditions are different along the chip, the worst condition for hold check occurs when the launch clock path and the data path have OCV conditions which result in the smallest delays, that is, when we have the earliest launch clock, and the capture clock path has the OCV conditions which result in the largest delays, that is, has the latest capture clock.

The hold timing check is specified in the following expression for this example.

```
LaunchClockPath + MinDataPath - CaptureClockPath -
  Thold_UFF1 >= 0
```

Applying the delay values in the Figure 10-2 to the expression, we get (without applying any derating):

```
LaunchClockPath = 0.25 + 0.6 = 0.85
MinDataPath = 1.7
CaptureClockPath = 0.25 + 0.75 = 1.00
Thold_UFF1 = 1.25
```

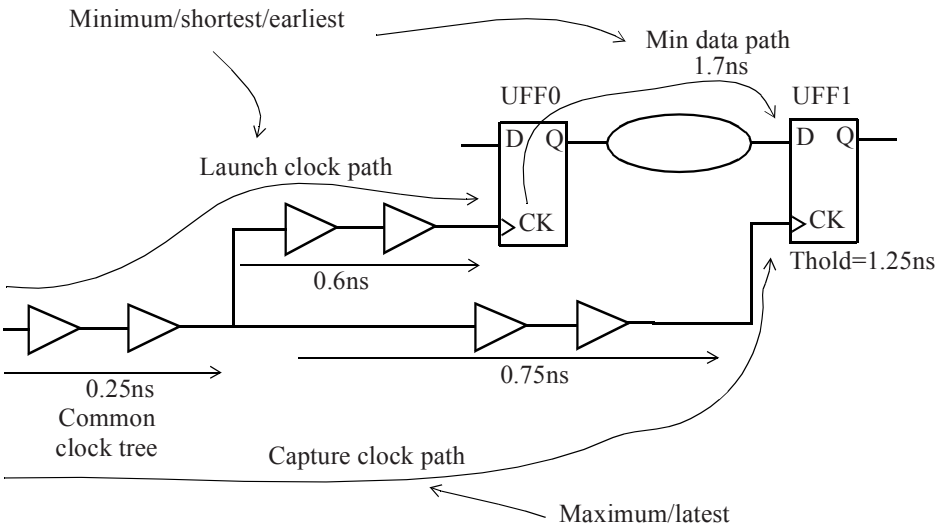


Figure 10-2 *Derating hold timing check for OCV.*

This implies that the condition is:

```
0.85 + 1.7 - 1.00 - 1.25 = 0.3n >=0
```

which is true, and thus no hold violation exists.

Applying the following derate specification:

```
set_timing_derate -early 0.9
set_timing_derate -late 1.2
set_timing_derate -early 0.95 -cell_check
```

we get:

```
LaunchClockPath = 0.85 * 0.9 = 0.765
MinDataPath = 1.7 * 0.9 = 1.53
```

```
CaptureClockPath = 1.00 * 1.2 = 1.2
Thold_UFF1 = 1.25 * 0.95 = 1.1875

Common clock path pessimism: 0.25 * (1.2 - 0.9) = 0.075
```

Common clock path pessimism created by applying derating on the common clock tree for both launch and capture clock paths is also removed for hold timing checks. The hold check condition then becomes:

```
0.765 + 1.53 – 1.2 - 1.1875 + 0.075 = -0.0175ns
```

which is less than 0, thus showing that there is a hold violation with the OCV derating factors applied to the early and late paths.

In general, the hold timing check is performed at the best-case fast PVT corner. In such a scenario, no derating is necessary on the early paths, as those paths are already the earliest possible. However, derating can be applied on the late paths by making these slower by a specific derating factor, for example, slowing the late paths by 20%. A derate specification at this corner would be something like:

```
set_timing_derate -early 1.0
set_timing_derate -late 1.2
# Don't derate the early paths as they are already the
# fastest, but derate the late paths slower by 20%.
```

In the example of Figure 10-2,

```
LatestArrivalTime@CommonPoint = 0.25 * 1.2 = 0.30
EarliestArrivalTime@CommonPoint = 0.25 * 1.0 = 0.25
```

This implies a common path pessimism of:

```
0.30 - 0.25 = 0.05ns
```

Here is a hold timing check path report for an example design that uses this derating.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLKM)
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
  Path Group: CLKM
  Path Type: min
  Min Data Paths Derating Factor  : 1.000
  Min Clock Paths Derating Factor : 1.000
  Max Clock Paths Derating Factor : 1.200

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLKM (rise edge)                  0.000      0.000
  clock source latency                    0.000      0.000
  CLKM (in)                               0.000      0.000 r
  UCKBUF0/C (CKB  )                       0.056      0.056 r
  UCKBUF1/C (CKB  )                       0.058      0.114 r
  UFF0/CK (DF  )                          0.000      0.114 r
  UFF0/Q (DF  ) <-                        0.144      0.258 r
  UNOR0/ZN (NR2  )                        0.021      0.279 f
  UBUF4/Z (BUFF  )                        0.055      0.334 f
  UFF1/D (DF  )                           0.000      0.334 f
  data arrival time                                  0.334

  clock CLKM (rise edge)                  0.000      0.000
  clock source latency                    0.000      0.000
  CLKM (in)                               0.000      0.000 r
  UCKBUF0/C (CKB  )                       0.067      0.067 r
  UCKBUF2/C (CKB  )                       0.080      0.148 r
  UFF1/CK (DF  )                          0.000      0.148 r
  clock reconvergence pessimism          -0.011      0.136
  clock uncertainty                       0.050      0.186
  library hold time                       0.015      0.201
  data required time                                 0.201
  -----------------------------------------------------------
  data required time                                 0.201
  data arrival time                                 -0.334
  -----------------------------------------------------------
  slack (MET)                                        0.133
```

Notice that the late paths are derated by +20% while the early paths are not derated. See cell *UCKBUF0*. Its delay on the launch path is 56ps while the delay on the capture path is 67ps - derated by +20%. *UCKBUF0* is the cell on the common clock tree and thus the pessimism introduced due to different derating on this common clock tree is, 67ps - 56ps = 11ps, which is accounted for separately on the line *clock reconvergence pessimism*.

10.2 Time Borrowing

The time borrowing technique, which is also called **cycle stealing**, occurs at a latch. In a latch, one edge of the clock makes the latch transparent, that is, it opens the latch so that output of the latch is the same as the data input; this clock edge is called the **opening edge.** The second edge of the clock closes the latch, that is, any change on the data input is no longer available at the output of the latch; this clock edge is called the **closing edge**.

Typically, the data should be ready at a latch input before the active edge of the clock. However, since a latch is transparent when the clock is active, the data can arrive later than the active clock edge, that is, it can borrow time from the next cycle. If such time is borrowed, the time available for the following stage (latch to another sequential cell) is reduced.

Figure 10-3 shows an example of time borrowing using an active rising edge. If data *DIN* is ready at time *A* prior to the latch opening on the rising edge of *CLK* at 10ns, the data flows to the output of the latch as it opens. If data arrives at time *B* as shown for *DIN (delayed)*, it borrows time *Tb*. However, this reduces the time available from the latch to the next flip-flop *UFF2* - instead of a complete clock cycle, only time *Ta* is available.

The first rule in timing to a latch is that if the data arrives before the opening edge of the latch, the behavior is modeled exactly like a flip-flop. The opening edge captures the data and the same clock edge launches the data as the start point for the next path.

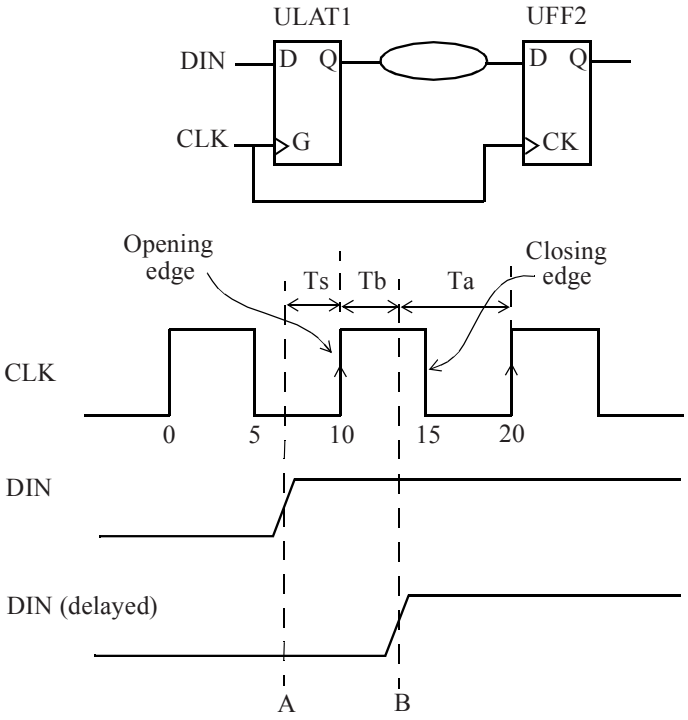


Figure 10-3 *Time borrowing.*

The second rule applies when the data signal arrives while the latch is transparent (between the opening and the closing edge). The output of the latch, rather than the clock pin, is used as the launch point for the next stage. The amount of time borrowed by the path ending at the latch determines the launch time for the next stage.

A data signal that arrives after the closing edge at the latch is a timing violation. Figure 10-4 shows the timing regions for data arrival for positive slack, zero slack, and negative slack (that is, when a violation occurs).

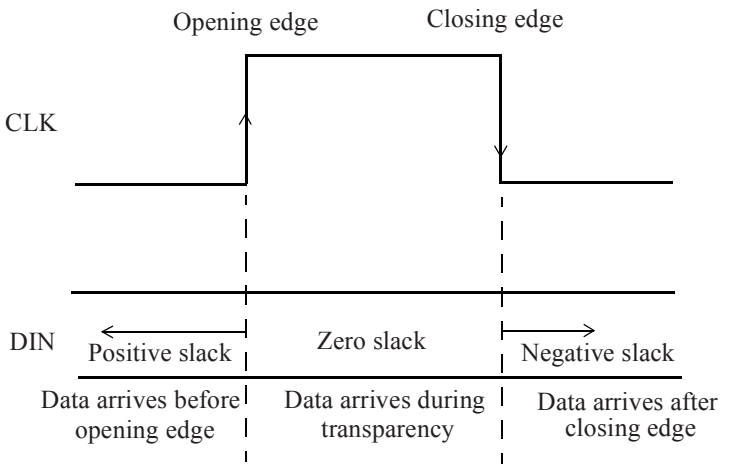


Figure 10-4 *Latch timing violation windows.*

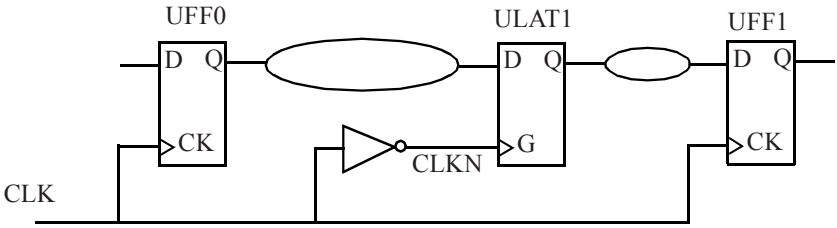
Figure 10-5(a) shows the use of a latch with a half-cycle path to the next stage flip-flop. Figure 10-5(b) depicts the waveforms for the scenario of time borrowing. The clock period is 10ns. The data is launched by *UFF0* at time 0, but the data path takes 7ns. The latch *ULAT1* opens up at 5ns. Thus 2ns is borrowed from the path *ULAT1* to *UFF1*. The time available for the *ULAT1* to *UFF1* path is only 3ns (5ns - 2ns).

We next describe three sets of timing reports for the latch example in Figure 10-5(a) to illustrate the different amounts of time borrowed from the next stage.

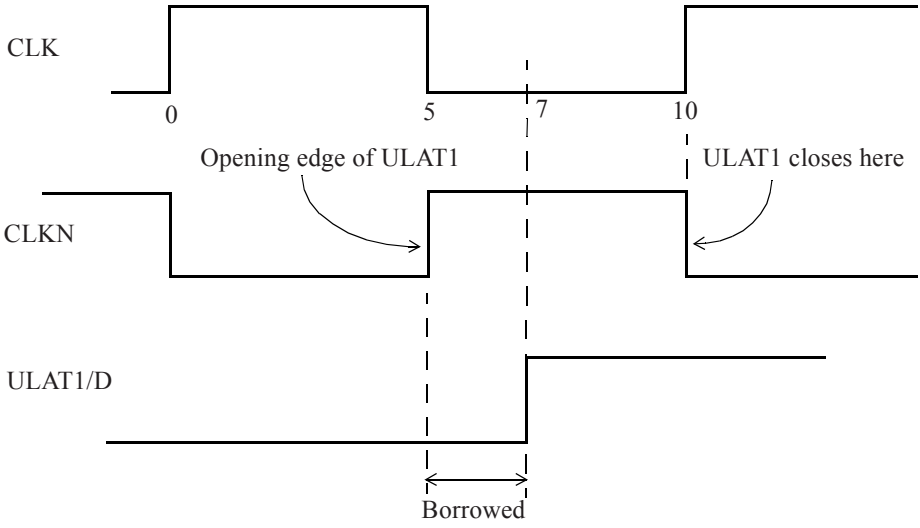
Example with No Time Borrowed

Here is the setup path report when the data path delay from the flip-flop *UFF0* to the latch *ULAT1* is less than 5ns.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: ULAT1 (positive level-sensitive latch clocked by CLK')
  Path Group: CLK
```



(a) Logic.



(b) Clock and data waveforms for 7ns data path.

Figure 10-5 *Time borrowing example.*

```
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock CLK (rise edge)                          0.00       0.00
clock source latency                           0.00       0.00
```

```
clk (in)                                  0.00       0.00 r
UFF0/CK (DF  )                            0.00       0.00 r
UFF0/Q (DF  )                             0.12       0.12 r
UBUF0/Z (BUFF  )                          2.01       2.13 r
UBUF1/Z (BUFF  )                          2.46       4.59 r
UBUF2/Z (BUFF  )                          0.07       4.65 r
ULAT1/D (LH  )                            0.00       4.65 r
data arrival time                                    4.65

clock CLK' (rise edge)                    5.00       5.00
clock source latency                      0.00       5.00
clk (in)                                  0.00       5.00 f
UINV0/ZN (INV  )                          0.02       5.02 r
ULAT1/G (LH  )                            0.00       5.02 r
time borrowed from endpoint               0.00       5.02
data required time                                   5.02
-------------------------------------------------------------
data required time                                   5.02
data arrival time                                   -4.65
-------------------------------------------------------------
slack (MET)                                          0.36

Time Borrowing Information
-------------------------------------------------
CLK' nominal pulse width                  5.00
clock latency difference                 -0.00
library setup time                       -0.01
-------------------------------------------------
max time borrow                           4.99
actual time borrow                        0.00
-------------------------------------------------
```

In this case, there is no need to borrow as data reaches the latch *ULAT1* in time before the latch opens.

Example with Time Borrowed

The path report below shows the case where the data path delay from the flip-flop *UFF0* to the latch *ULAT1* is greater than 5ns.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: ULAT1 (positive level-sensitive latch clocked by CLK')
  Path Group: CLK
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLK (rise edge)                    0.00       0.00
  clock source latency                     0.00       0.00
  clk (in)                                 0.00       0.00 r
  UFF0/CK (DF  )                           0.00       0.00 r
  UFF0/Q (DF  )                            0.12       0.12 r
  UBUF0/Z (BUFF  )                         3.50       3.62 r
  UBUF1/Z (BUFF  )                         3.14       6.76 r
  UBUF2/Z (BUFF  )                         0.07       6.83 r
  ULAT1/D (LH  )                           0.00       6.83 r
  data arrival time                                   6.83

  clock CLK' (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  clk (in)                                 0.00       5.00 f
  UINV0/ZN (INV  )                         0.02       5.02 r
  ULAT1/G (LH  )                           0.00       5.02 r
  time borrowed from endpoint              1.81       6.83
  data required time                                  6.83
  -----------------------------------------------------------
  data required time                                  6.83
  data arrival time                                  -6.83
  -----------------------------------------------------------
  slack (MET)                                         0.00

  Time Borrowing Information
  -------------------------------------------------
  CLK' nominal pulse width                 5.00
  clock latency difference                -0.00
  library setup time                      -0.01
  -------------------------------------------------
```

```
  max time borrow                          4.99
  actual time borrow                       1.81
  -------------------------------------------------
```

In this case, since the data becomes available while the latch is transparent, the required delay of 1.81ns is borrowed from the subsequent path and the timing is still met. Here is the path report of the subsequent path showing that 1.81ns was already borrowed by the previous path.

```
Startpoint: ULAT1 (positive level-sensitive latch clocked by CLK')
  Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLK)
  Path Group: CLK
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------------
  clock CLK' (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  clk (in)                                 0.00       5.00 f
  UINV0/ZN (INV  )                         0.02       5.02 r
  ULAT1/G (LH  )                           0.00       5.02 r
  time given to startpoint                 1.81       6.83
  ULAT1/QN (LH  )                          0.13       6.95 f
  UFF1/D (DF  )                            0.00       6.95 f
  data arrival time                                   6.95

  clock CLK (rise edge)                   10.00      10.00
  clock source latency                     0.00      10.00
  clk (in)                                 0.00      10.00 r
  UFF1/CK (DF  )                           0.00      10.00 r
  library setup time                      -0.04       9.96
  data required time                                  9.96
  -----------------------------------------------------------------
  data required time                                  9.96
  data arrival time                                  -6.95
  -----------------------------------------------------------------
  slack (MET)                                         3.01
```

Example with Timing Violation

In this case, the data path delay is much larger and data becomes available only after the latch closes. This is clearly a timing violation.

```
Startpoint: UFF0 (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: ULAT1 (positive level-sensitive latch clocked by CLK')
  Path Group: CLK
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock CLK (rise edge)                    0.00       0.00
  clock source latency                     0.00       0.00
  clk (in)                                 0.00       0.00 r
  UFF0/CK (DF  )                           0.00       0.00 r
  UFF0/Q (DF  )                            0.12       0.12 r
  UBUF0/Z (BUFF  )                         6.65       6.77 r
  UBUF1/Z (BUFF  )                         4.33      11.10 r
  UBUF2/Z (BUFF  )                         0.07      11.17 r
  ULAT1/D (LH  )                           0.00      11.17 r
  data arrival time                                  11.17

  clock CLK' (rise edge)                   5.00       5.00
  clock source latency                     0.00       5.00
  clk (in)                                 0.00       5.00 f
  UINV0/ZN (INV  )                         0.02       5.02 r
  ULAT1/G (LH  )                           0.00       5.02 r
  time borrowed from endpoint              4.99      10.00
  data required time                                 10.00
  -----------------------------------------------------------
  data required time                                 10.00
  data arrival time                                 -11.17
  -----------------------------------------------------------
  slack (VIOLATED)                                   -1.16

  Time Borrowing Information
  -------------------------------------------------
  CLK' nominal pulse width                 5.00
  clock latency difference                -0.00
  library setup time                      -0.01
```

```
-------------------------------------------------
max time borrow                          4.99
actual time borrow                       4.99
-------------------------------------------------
```

10.3 Data to Data Checks

Setup and hold checks can also be applied between any two arbitrary data pins, neither of which is a clock. One pin is the **constrained pin**, which acts like a data pin of a flip-flop, and the second pin is the **related pin**, which acts like a clock pin of a flip-flop. One important distinction with respect to the setup check of a flip-flop is that the data to data setup check is performed on the same edge as the launch edge (unlike a normal setup check of a flip-flop, where the capture clock edge is normally one cycle away from the launch clock edge). Thus, the data to data setup checks are also referred to as **zero-cycle checks** or **same-cycle checks**.

A data to data check is specified using the **set_data_check** constraint. Here are example SDC specifications.

```
set_data_check -from SDA -to SCTRL -setup 2.1
set_data_check -from SDA -to SCTRL -hold 1.5
```

See Figure 10-6. *SDA* is the related pin and *SCTRL* is the constrained pin. The setup data check implies that *SCTRL* should arrive at least 2.1ns prior to the edge of the related pin *SDA*. Otherwise it is a data to data setup check violation. The hold data check specifies that SCTRL should arrive at least 1.5ns after *SDA*. If the constrained signal arrives earlier than this specification, then it is a data to data hold check violation.

This check is useful in a custom-designed block where it may be necessary to provide specific arrival times of one signal with respect to another. One such common situation is that of a data signal gated by an enable signal and it is required to ensure that the enable signal is stable when the data signal arrives.

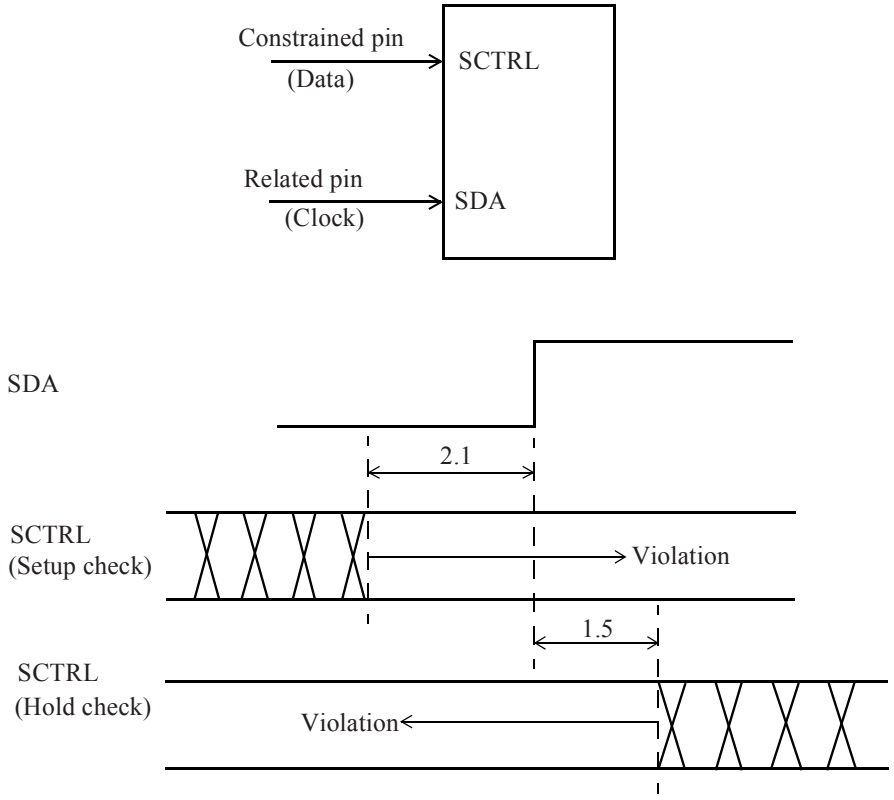


Figure 10-6 *Data to data checks.*

Consider the *and* cell shown in Figure 10-7. We assume the requirement is to ensure that *PNA* arrives 1.8ns before the rising edge of *PREAD* and that it should not change for 1.0ns after the rising edge of *PREAD*. In this example, *PNA* is the constrained pin and *PREAD* is the related pin. The required waveforms are shown in Figure 10-7.

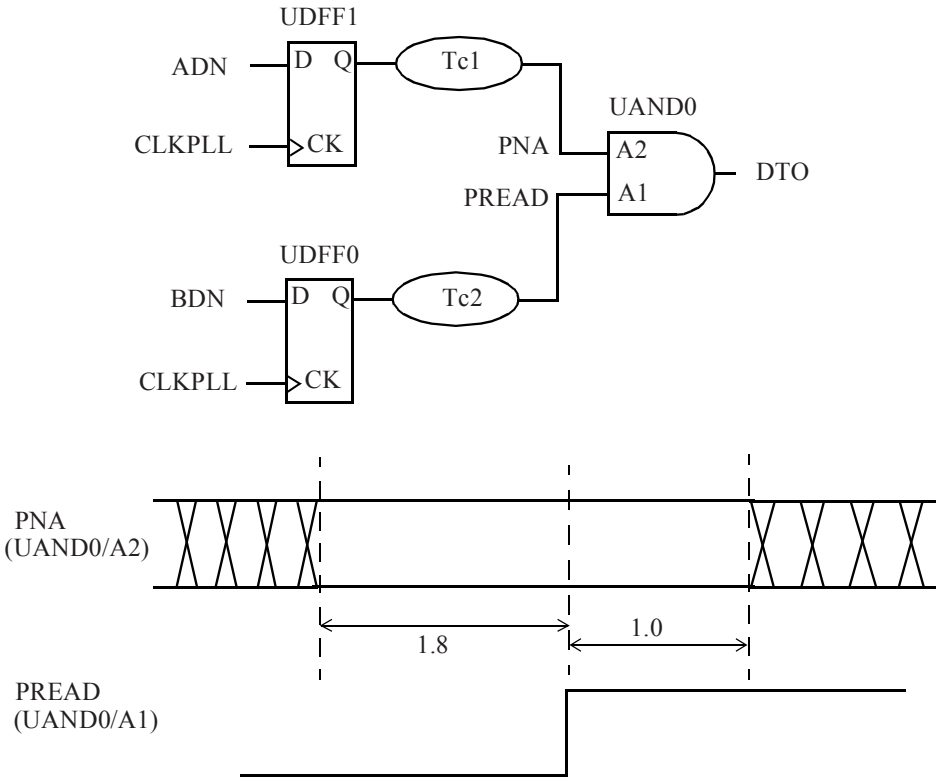


Figure 10-7 *Setup and hold timing checks between PNA and PREAD.*

Such a requirement can be specified using a data to data setup and hold check.

```
set_data_check -from UAND0/A1 -to UAND0/A2 -setup 1.8
set_data_check -from UAND0/A1 -to UAND0/A2 -hold 1.0
```

Here is the setup report.

```
Startpoint: UDFF1
 (rising edge-triggered flip-flop clocked by CLKPLL)
Endpoint: UAND0
 (rising edge-triggered data to data check clocked by CLKPLL)
Path Group: CLKPLL
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKPLL (rise edge)                 0.00       0.00
clock source latency                     0.00       0.00
CLKPLL (in)                              0.00       0.00 r
UDFF1/CK (DF  )                          0.00       0.00 r
UDFF1/Q (DF  )                           0.12       0.12 f
UBUF0/Z (BUFF  )                         0.06       0.18 f
UAND0/A2 (AN2  )                         0.00       0.18 f
data arrival time                                   0.18

clock CLKPLL (rise edge)                 0.00       0.00
clock source latency                     0.00       0.00
CLKPLL (in)                              0.00       0.00 r
UDFF0/CK (DF  )                          0.00       0.00 r
UDFF0/Q (DF  )                           0.12       0.12 r
UBUF1/Z (BUFF  )                         0.05       0.17 r
UBUF2/Z (BUFF  )                         0.05       0.21 r
UBUF3/Z (BUFF  )                         0.05       0.26 r
UAND0/A1 (AN2  )                         0.00       0.26 r
data check setup time                   -1.80      -1.54
data required time                                 -1.54
-----------------------------------------------------------
data required time                                 -1.54
data arrival time                                  -0.18
-----------------------------------------------------------
slack (VIOLATED)                                   -1.72
```

The setup time is specified as *data check setup time* in the report. The failing report indicates that the *PREAD* needs to be delayed by at least 1.72ns to ensure that *PENA* arrives 1.8ns before *PREAD* - which is our requirement.

One important aspect of a data to data setup check is that the clock edges that launch both the constrained pin and the related pin are from the same clock cycle (also referred to as *same-cycle checks*). Thus notice in the report that the starting time for the capture edge (*UDFF0/CK*) is at 0ns, not one cycle later as one would typically see in a setup report.

The zero-cycle setup check causes the hold timing check to be different from other hold check reports - the hold check is no longer on the same clock edge. Here is the clock specification for *CLKPLL* which is utilized for the hold path report below.

```
create_clock -name CLKPLL -period 10 -waveform {0 5} \
  [get_ports CLKPLL]
```

```
Startpoint: UDFF1
  (rising edge-triggered flip-flop clocked by CLKPLL)
Endpoint: UAND0
  (falling edge-triggered data to data check clocked by CLKPLL)
Path Group: CLKPLL
Path Type: min

Point                                        Incr       Path
-------------------------------------------------------------
clock CLKPLL (rise edge)                    10.00      10.00
clock source latency                         0.00      10.00
CLKPLL (in)                                  0.00      10.00 r
UDFF1/CK (DF  )                              0.00      10.00 r
UDFF1/Q (DF  ) <-                            0.12      10.12 r
UBUF0/Z (BUFF  )                             0.05      10.17 r
UAND0/A2 (AN2  )                             0.00      10.17 r
data arrival time                                      10.17

clock CLKPLL (rise edge)                     0.00       0.00
clock source latency                         0.00       0.00
CLKPLL (in)                                  0.00       0.00 r
UDFF0/CK (DF  )                              0.00       0.00 r
UDFF0/Q (DF  )                               0.12       0.12 f
UBUF1/Z (BUFF  )                             0.06       0.18 f
```

```
UBUF2/Z (BUFF  )                          0.05       0.23 f
UBUF3/Z (BUFF  )                          0.06       0.29 f
UAND0/A1 (AN2  )                          0.00       0.29 f
data check hold time                      1.00       1.29
data required time                                   1.29
-------------------------------------------------------------
data required time                                   1.29
data arrival time                                  -10.17
-------------------------------------------------------------
slack (MET)                                          8.88
```

Notice that the clock edge used to launch the related pin for the hold check is one cycle prior to the launch edge for the constrained pin. This is because by definition a hold check is normally performed one cycle prior to the setup capture edge. Since the clock edges for the constrained pin and the related pin are the same for a data to data setup check, the hold check is done one cycle prior to the launch edge.

In some scenarios, a designer may require the data to data hold check to be performed on the same clock cycle. The same cycle hold requirement implies that the clock edge used for the related pin be moved back to where the clock edge for the constrained pin is. This can be achieved by specifying a multicycle of -1.

```
set_multicycle_path -1 -hold -to UAND0/A2
```

Here is the hold timing report for the example above with this multicycle specification.

```
Startpoint: UDFF1
  (rising edge-triggered flip-flop clocked by CLKPLL)
Endpoint: UAND0
  (falling edge-triggered data to data check clocked by CLKPLL)
Path Group: CLKPLL
Path Type: min

Point                                     Incr       Path
-------------------------------------------------------------
```

```
clock CLKPLL (rise edge)                  0.00       0.00
clock source latency                      0.00       0.00
CLKPLL (in)                               0.00       0.00 r
UDFF1/CK (DF  )                           0.00       0.00 r
UDFF1/Q (DF  ) <-                         0.12       0.12 r
UBUF0/Z (BUFF  )                          0.05       0.17 r
UAND0/A2 (AN2  )                          0.00       0.17 r
data arrival time                                    0.17

clock CLKPLL (rise edge)                  0.00       0.00
clock source latency                      0.00       0.00
CLKPLL (in)                               0.00       0.00 r
UDFF0/CK (DF  )                           0.00       0.00 r
UDFF0/Q (DF  )                            0.12       0.12 f
UBUF1/Z (BUFF  )                          0.06       0.18 f
UBUF2/Z (BUFF  )                          0.05       0.23 f
UBUF3/Z (BUFF  )                          0.06       0.29 f
UAND0/A1 (AN2  )                          0.00       0.29 f
data check hold time                      1.00       1.29
data required time                                   1.29
-----------------------------------------------------------
data required time                                   1.29
data arrival time                                   -0.17
-----------------------------------------------------------
slack (VIOLATED)                                    -1.12
```

The hold check is now performed using the same clock edge for the constrained pin and the related pin. An alternate way of having the data to data hold check performed in the same cycle is to specify this as a data to data setup check between the pins in the reverse direction.

```
set_data_check -from UAND0/A2 -to UAND0/A1 -setup 1.0
```

The data to data check is also useful in defining a **no-change data check**. This is done by specifying a setup check on the rising edge and a hold check on the falling edge, such that a no-change window gets effectively defined. This is shown in Figure 10-8.

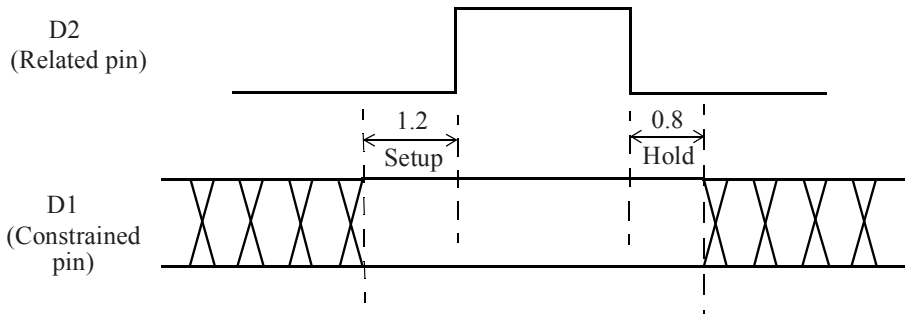


Figure 10-8 *A no-change data check achieved using setup and hold data checks.*

Here are the specifications for this scenario.

```
set_data_check -rise_from D2 -to D1 -setup 1.2
set_data_check -fall_from D2 -to D1 -hold 0.8
```

10.4 Non-Sequential Checks

A library file for a cell or a macro may specify a timing arc to be a non-sequential check, such as a timing arc between two data pins. A non-sequential check is a check between two pins, neither of which is a clock. One pin is the constrained pin that acts like data, while the second pin is the related pin and this acts like a clock. The check specifies how long the data on the constrained pin must be stable before and after the change on the related pin.

Note that this check is specified as part of the cell library specification and no explicit data to data check constraint is required. Here is how such a timing arc may appear in a cell library.

```
pin (WEN) {
  timing () {
    timing_type: non_seq_setup_rising;
```

```
    intrinsic_rise: 1.1;
    intrinsic_fall:1.15;
    related_pin: "D0";
  }
  timing () {
    timing_type: non_seq_hold_rising;
    intrinsic_rise: 0.6;
    intrinsic_fall:0.65;
    related_pin: "D0";
  }
}
```

The *setup_rising* refers to the rising edge of the related pin. The intrinsic rise and fall values refer to the rise and fall setup times for the constrained pin. Similar timing arcs can be defined for *hold_rising*, *setup_falling* and *hold_falling*.

A non-sequential check is similar to a data to data check described in Section 10.3, though there are two main differences. In a non-sequential check, the setup and hold values are obtained from the standard cell library, where the setup and hold timing models can be described using a NLDM table model or other advanced timing models. In a data to data check, only a single value can be specified for the data to data setup or hold check. The second difference is that a non-sequential check can only be applied to pins of a cell, whereas a data to data check can be applied to any two arbitrary pins in a design.

A **non-sequential setup check** specifies how early the constrained signal must arrive relative to the related pin. This is shown in Figure 10-9. The cell library contains the setup arc *D0->WEN* which is specified as a non-sequential arc. If the *WEN* signal occurs within the setup window, the non-sequential setup check fails.

A **non-sequential hold check** specifies how late the constrained signal must arrive relative to the related pin. See Figure 10-9. If *WEN* changes within the hold window, then the hold check fails.

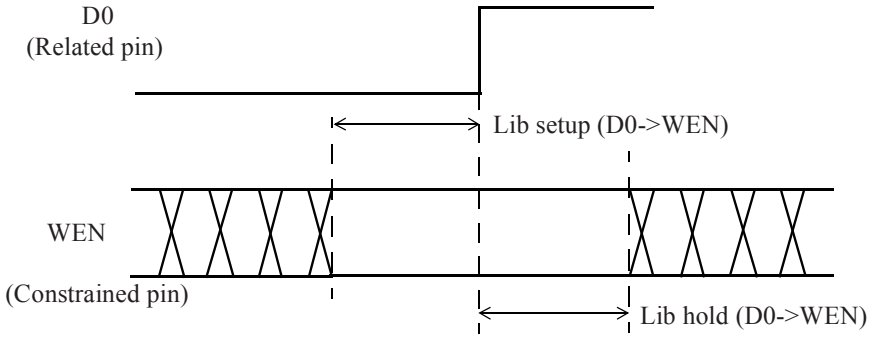


Figure 10-9 *A non-sequential setup and hold check.*

10.5 Clock Gating Checks

A clock gating check occurs when a *gating signal* can control the path of a *clock signal* at a logic cell. An example is shown in Figure 10-10. The pin of the logic cell connected to the clock is called the **clock pin** and the pin where the gating signal is connected to is the **gating pin**. The logic cell where the clock gating occurs is also referred to as the **gating cell**.

One condition for a clock gating check is that the clock that goes through the cell must be used as a clock downstream. The downstream clock usage can be either as a flip-flop clock or it can fanout to an output port or as a generated clock that refers to the output of the gating cell as its master. If the clock is not used as a clock after the gating cell, then no clock gating check is inferred.

Another condition for the clock gating check applies to the gating signal. The signal at the gating pin of the check should not be a clock or if it is a clock, it should not be used as a clock downstream (an example of a clock as a gating signal is included later in this section).

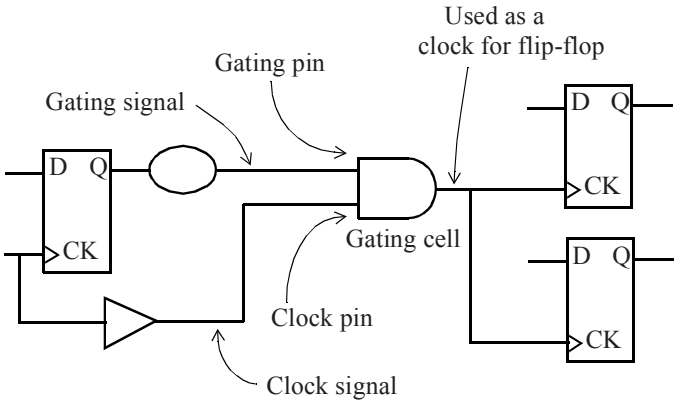


Figure 10-10 *A clock gating check.*

In a general scenario, the clock signal and the gating signal do not need to be connected to a single logic cell such as *and,* or *or*, but may be inputs to an arbitrary logic block. In such cases, for a clock gating check to be inferred, the clock pin of the check and the gating pin of the check must fan out to a common output pin.

There are two types of clock gating checks inferred:

- *Active-high clock gating check*: Occurs when the gating cell has an *and* or a *nand* function.
- *Active-low clock gating check*: Occurs when the gating cell has an *or* or a *nor* function.

The active-high and active-low refer to the logic state of the gating signal which activates the clock signal at the output of the gating cell. If the gating cell is a complex function where the gating relationship is not obvious, such as a multiplexer or an *xor* cell, STA output will typically provide a warning that no clock gating check is being inferred. However this can be changed by specifying a clock gating relationship for the gating cell explicitly by using the command **set_clock_gating_check**. In such cases, if the

set_clock_gating_check specification disagrees with the functionality of the gating cell, the STA will normally provide a warning. We present examples of such commands later in this section.

As specified earlier, a clock can be a gating signal only if it is not used as a clock downstream. Consider the example in Figure 10-11. *CLKB* is not used as a clock downstream due to the definition of the generated clock of *CLKA* - the path of *CLKB* is blocked by the generated clock definition. Hence a clock gating check for clock *CLKA* is inferred for the *and* cell.

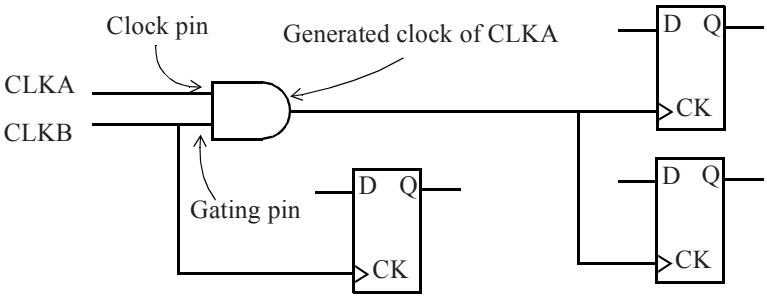


Figure 10-11 *Gating check inferred - clock at the gating pin not used as a clock downstream.*

Active-High Clock Gating

We now examine the timing relationship of an active-high clock gating check. This occurs at an *and* or a *nand* cell; an example using *and* is shown in Figure 10-12. Pin *B* of the gating cell is the clock signal, and pin *A* of the gating cell is the gating signal.

Let us assume that both clocks *CLKA* and *CLKB* have the same waveforms.

```
create_clock -name CLKA -period 10 -waveform {0 5} \
  [get_ports CLKA]
create_clock -name CLKB -period 10 -waveform {0 5} \
  [get_ports CLKB]
```

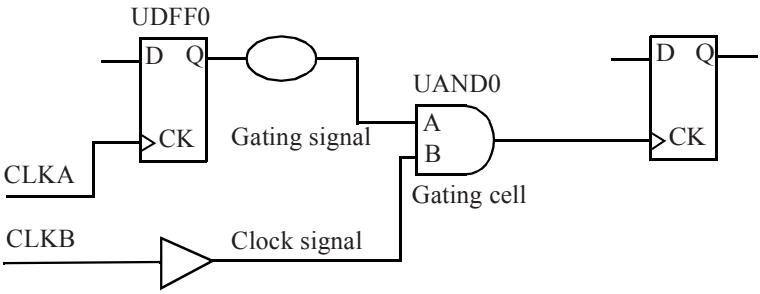


Figure 10-12 *Active high clock gating using an AND cell.*

Because it is an *and* cell, a high on gating signal *UAND0/A* opens up the gating cell and allows the clock to propagate through. The clock gating check is intended to validate that the gating pin transition does not create an active edge for the fanout clock. For positive edge-triggered logic, this implies that the rising edge of the gating signal occurs during the inactive period of the clock (when it is low). Similarly, for negative edge-triggered logic, the falling edge of the gating signal should occur only when the clock is low. Note that if the clock drives both positive and negative edge-triggered flip-flops, any transition of the gating signal (rising or falling edge) must occur only when the clock is low. Figure 10-13 shows an example of a gating signal transition during the active edge which needs to be delayed to pass the clock gating check.

The active-high clock gating setup check requires that the gating signal changes before the clock goes high. Here is the setup path report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by CLKA)
Endpoint: UAND0
  (rising clock gating-check end-point clocked by CLKB)
Path Group: **clock_gating_default**
Path Type: max

Point                                    Incr       Path
-------------------------------------------------------------
```

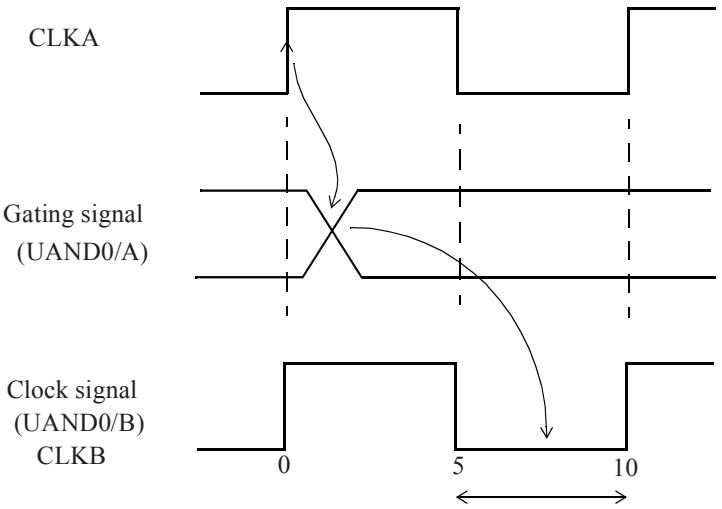


Figure 10-13 *Gating signal needs to be delayed.*

```
clock CLKA (rise edge)                    0.00       0.00
clock source latency                      0.00       0.00
CLKA (in)                                 0.00       0.00 r
UDFF0/CK (DF  )                           0.00       0.00 r
UDFF0/Q (DF  )                            0.13       0.13 f
UAND0/A1 (AN2  )                          0.00       0.13 f
data arrival time                                    0.13

clock CLKB (rise edge)                   10.00      10.00
clock source latency                      0.00      10.00
CLKB (in)                                 0.00      10.00 r
UAND0/A2 (AN2  )                          0.00      10.00 r
clock gating setup time                   0.00      10.00
data required time                                  10.00
-----------------------------------------------------------------
data required time                                  10.00
data arrival time                                   -0.13
-----------------------------------------------------------------
slack (MET)                                          9.87
```

Notice that the *Endpoint* indicates that it is a clock gating check. In addition, the path is in the *clock_gating_default* group of paths as specified in *Path Group*. The check validates that the gating signal changes before the next rising edge of clock *CLKB* at 10ns.

The active-high clock gating hold check requires that the gating signal changes only after the falling edge of the clock. Here is the hold path report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by CLKA)
Endpoint: UAND0
  (rising clock gating-check end-point clocked by CLKB)
Path Group: **clock_gating_default**
Path Type: min

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKA (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
CLKA (in)                                0.00       0.00 r
UDFF0/CK (DF  )                          0.00       0.00 r
UDFF0/Q (DF  )                           0.13       0.13 r
UAND0/A1 (AN2  )                         0.00       0.13 r
data arrival time                                   0.13

clock CLKB (fall edge)                   5.00       5.00
clock source latency                     0.00       5.00
CLKB (in)                                0.00       5.00 f
UAND0/A2 (AN2  )                         0.00       5.00 f
clock gating hold time                   0.00       5.00
data required time                                  5.00
-----------------------------------------------------------
data required time                                  5.00
data arrival time                                  -0.13
-----------------------------------------------------------
slack (VIOLATED)                                   -4.87
```

The hold gating check fails because the gating signal is changing too fast, before the falling edge of *CLKB* at 5ns. If a 5ns delay was added between *UDFF0/Q* and *UAND0/A1* pins, both setup and hold gating checks would pass validating that the gating signal changes only in the specified window.

One can see that the hold time requirement is quite large. This is caused by the fact that the sense of the gating signal and the flip-flops being gated are the same. This can be resolved by using a different type of launch flip-flop, say, a negative edge-triggered flip-flop to generate the gating signal. Such an example is shown next.

In Figure 10-14, the flip-flop *UFF0* is controlled by the negative edge of clock *CLKA*. Safe clock gating implies that the output of flip-flop *UFF0* must change during the inactive part of the gating clock, which is between 5ns and 10ns.

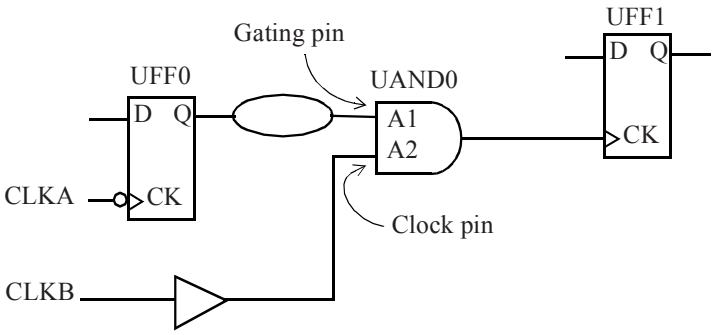


Figure 10-14 *Gating signal clocked on falling edge.*

The signal waveforms corresponding to the schematic in Figure 10-14 are depicted in Figure 10-15. Here is the clock gating setup report.

```
Startpoint: UFF0
  (falling edge-triggered flip-flop clocked by CLKA)
Endpoint: UAND0
  (rising clock gating-check end-point clocked by CLKB)
```

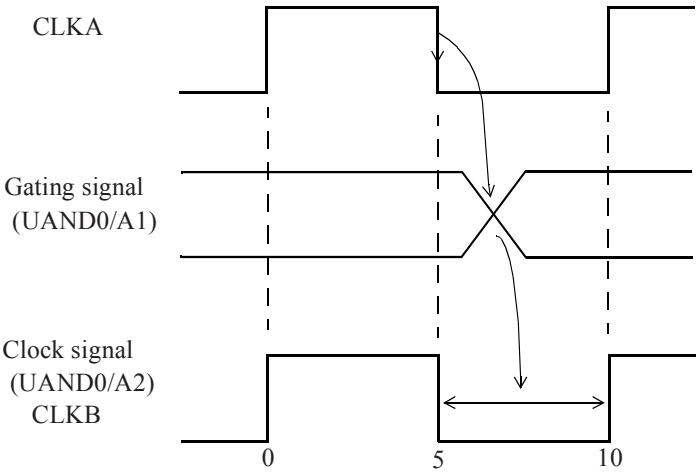


Figure 10-15 *Gating signal generated from negative edge flip-flop meets the gating checks.*

```
Path Group: **clock_gating_default**
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock CLKA (fall edge)                         5.00       5.00
clock source latency                           0.00       5.00
CLKA (in)                                      0.00       5.00 f
UFF0/CKN (DFN  )                               0.00       5.00 f
UFF0/Q (DFN  )                                 0.15       5.15 r
UAND0/A1 (AN2  )                               0.00       5.15 r
data arrival time                                         5.15

clock CLKB (rise edge)                        10.00      10.00
clock source latency                           0.00      10.00
CLKB (in)                                      0.00      10.00 r
UAND0/A2 (AN2  )                               0.00      10.00 r
clock gating setup time                        0.00      10.00
data required time                                       10.00
-----------------------------------------------------------------
data required time                                       10.00
```

```
data arrival time                                      -5.15
-------------------------------------------------------------
slack (MET)                                             4.85
```

Here is the clock gating hold report. Notice that the hold time check is much easier to meet with the new design.

```
Startpoint: UFF0
  (falling edge-triggered flip-flop clocked by CLKA)
Endpoint: UAND0
  (rising clock gating-check end-point clocked by CLKB)
Path Group: **clock_gating_default**
Path Type: min

Point                                    Incr       Path
-------------------------------------------------------------
clock CLKA (fall edge)                   5.00       5.00
clock source latency                     0.00       5.00
CLKA (in)                                0.00       5.00 f
UFF0/CKN (DFN  )                         0.00       5.00 f
UFF0/Q (DFN  )                           0.13       5.13 f
UAND0/A1 (AN2  )                         0.00       5.13 f
data arrival time                                   5.13

clock CLKB (fall edge)                   5.00       5.00
clock source latency                     0.00       5.00
CLKB (in)                                0.00       5.00 f
UAND0/A2 (AN2  )                         0.00       5.00 f
clock gating hold time                   0.00       5.00
data required time                                  5.00
-------------------------------------------------------------
data required time                                  5.00
data arrival time                                  -5.13
-------------------------------------------------------------
slack (MET)                                         0.13
```

Since the clock edge (negative edge) that launches the gating signal is opposite of the clock being gated (active-high), the setup and hold require-

ments are easy to meet. This is the most common structure used for gated clocks.

Active-Low Clock Gating

Figure 10-16 shows an example of an active-low clock gating check.

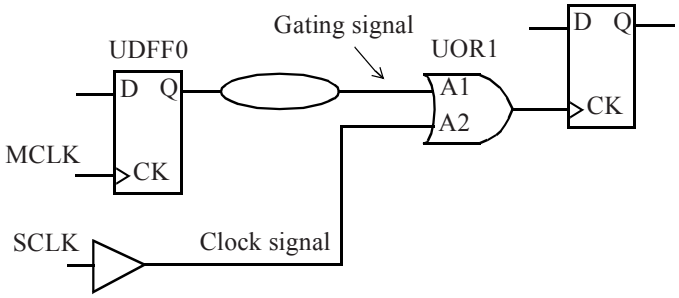


Figure 10-16 *Active-low clock gating check.*

```
create_clock -name MCLK -period 8 -waveform {0 4} \
  [get_ports MCLK]
create_clock -name SCLK -period 8 -waveform {0 4} \
  [get_ports SCLK]
```

Active-low clock gating check validates that the rising edge of the gating signal arrives at the active portion of the clock (when it is high) for positive edge-triggered logic. As described before, the key is that the gating signal should not cause an active edge for the output gated clock. When the gating signal is high, the clock cannot go through. Thus the gating signal should switch only when the clock is high as illustrated in Figure 10-17.

Here is the active-low clock gating setup timing report. This check ensures that the gating signal arrives before the clock edge becomes inactive, in this case, at 4ns.

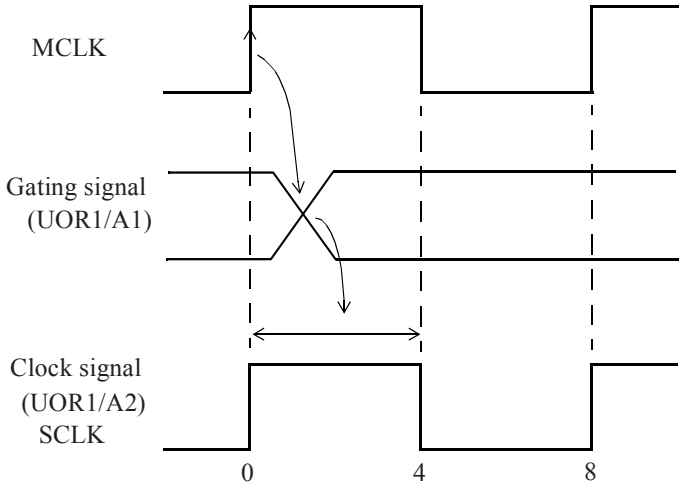


Figure 10-17 *Gating signal changes when clock is high.*

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by MCLK)
Endpoint: UOR1
  (falling clock gating-check end-point clocked by SCLK)
Path Group: **clock_gating_default**
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock MCLK (rise edge)                         0.00       0.00
clock source latency                           0.00       0.00
MCLK (in)                                      0.00       0.00 r
UDFF0/CK (DF  )                                0.00       0.00 r
UDFF0/Q (DF  )                                 0.13       0.13 f
UOR1/A1 (OR2  )                                0.00       0.13 f
data arrival time                                         0.13

clock SCLK (fall edge)                         4.00       4.00
clock source latency                           0.00       4.00
SCLK (in)                                      0.00       4.00 f
UOR1/A2 (OR2  )                                0.00       4.00 f
clock gating setup time                        0.00       4.00
```

```
data required time                                    4.00
-------------------------------------------------------------
data required time                                    4.00
data arrival time                                    -0.13
-------------------------------------------------------------
slack (MET)                                           3.87
```

Here is the clock gating hold timing report. This check ensures that the gating signal changes only after the rising edge of the clock signal, which in this case is at 0ns.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by MCLK)
Endpoint: UOR1
  (falling clock gating-check end-point clocked by SCLK)
Path Group: **clock_gating_default**
Path Type: min

Point                                    Incr       Path
-------------------------------------------------------------
clock MCLK (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
MCLK (in)                                0.00       0.00 r
UDFF0/CK (DF  )                          0.00       0.00 r
UDFF0/Q (DF  )                           0.13       0.13 r
UOR1/A1 (OR2  )                          0.00       0.13 r
data arrival time                                   0.13

clock SCLK (rise edge)                   0.00       0.00
clock source latency                     0.00       0.00
SCLK (in)                                0.00       0.00 r
UOR1/A2 (OR2  )                          0.00       0.00 r
clock gating hold time                   0.00       0.00
data required time                                  0.00
-------------------------------------------------------------
data required time                                  0.00
data arrival time                                  -0.13
-------------------------------------------------------------
slack (MET)                                         0.13
```

Clock Gating with a Multiplexer

Figure 10-18 shows an example of clock gating using a multiplexer cell. A clock gating check at the multiplexer inputs ensures that the multiplexer select signal arrives at the right time to cleanly switch between *MCLK* and *TCLK*. For this example, we are interested in switching to and from *MCLK* and assume that *TCLK* is low when the select signal switches. This implies that the select signal of the multiplexer should switch only when *MCLK* is low. This is similar to the active-high clock gating check.

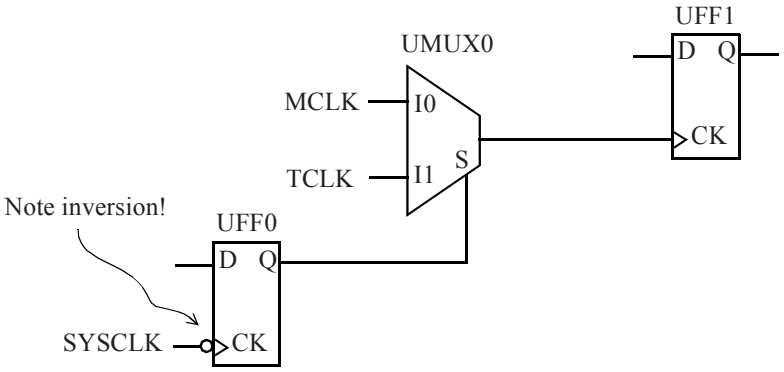


Figure 10-18 *Clock gating using a multiplexer.*

Figure 10-19 shows the timing relationships. The select signal for the multiplexer must arrive at the time *MCLK* is low. Also, assume *TCLK* will be low when select changes.

Since the gating cell is a multiplexer, the clock gating check is not inferred automatically, as evidenced in this message reported during STA.

```
Warning: No clock-gating check is inferred for clock MCLK at
pins UMUX0/S and UMUX0/I0 of cell UMUX0.
Warning: No clock-gating check is inferred for clock TCLK at
pins UMUX0/S and UMUX0/I1 of cell UMUX0.
```

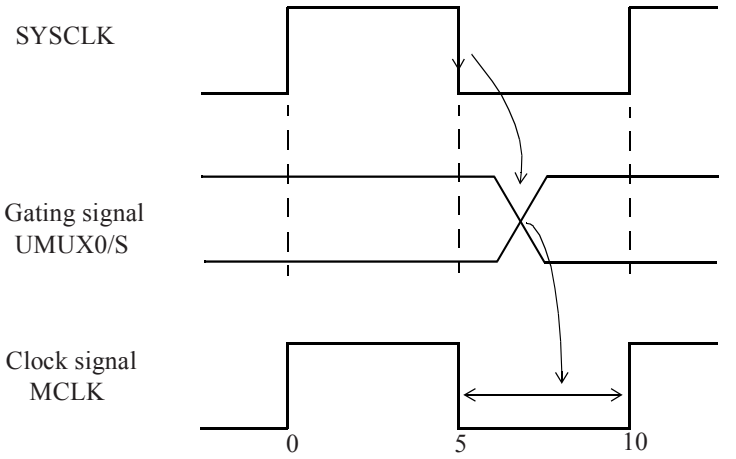


Figure 10-19 *Gating signal arrives when clock is low.*

However a clock gating check can be explicitly forced by providing a *set_clock_gating_check* specification.

```
set_clock_gating_check -high [get_cells UMUX0]
# The -high option indicates an active-high check.
set_disable_clock_gating_check UMUX0/I1
```

The disable check turns off the clock gating check on the specific pin, as we are not concerned with this pin. The clock gating check on the multiplexer has been specified to be an active-high clock gating check. Here is the setup timing path report.

```
Startpoint: UFF0
  (falling edge-triggered flip-flop clocked by SYSCLK)
Endpoint: UMUX0
  (rising clock gating-check end-point clocked by MCLK)
Path Group: **clock_gating_default**
Path Type: max
```

```
Point                                          Incr       Path
-------------------------------------------------------------------
clock SYSCLK (fall edge)                       5.00       5.00
clock source latency                           0.00       5.00
SYSCLK (in)                                    0.00       5.00 f
UFF0/CKN (DFN  )                               0.00       5.00 f
UFF0/Q (DFN  )                                 0.15       5.15 r
UMUX0/S (MUX2  )                               0.00       5.15 r
data arrival time                                         5.15

clock MCLK (rise edge)                        10.00      10.00
clock source latency                           0.00      10.00
MCLK (in)                                      0.00      10.00 r
UMUX0/I0 (MUX2  )                              0.00      10.00 r
clock gating setup time                        0.00      10.00
data required time                                       10.00
-------------------------------------------------------------------
data required time                                       10.00
data arrival time                                        -5.15
-------------------------------------------------------------------
slack (MET)                                               4.85
```

The clock gating hold timing report is next.

```
Startpoint: UFF0
  (falling edge-triggered flip-flop clocked by SYSCLK)
Endpoint: UMUX0
  (rising clock gating-check end-point clocked by MCLK)
Path Group: **clock_gating_default**
Path Type: min

Point                                          Incr       Path
-------------------------------------------------------------------
clock SYSCLK (fall edge)                       5.00       5.00
clock source latency                           0.00       5.00
SYSCLK (in)                                    0.00       5.00 f
UFF0/CKN (DFN  )                               0.00       5.00 f
UFF0/Q (DFN  )                                 0.13       5.13 f
UMUX0/S (MUX2  )                               0.00       5.13 f
data arrival time                                         5.13

clock MCLK (fall edge)                         5.00       5.00
```

```
clock source latency                    0.00       5.00
MCLK (in)                               0.00       5.00 f
UMUX0/I0 (MUX2  )                       0.00       5.00 f
clock gating hold time                  0.00       5.00
data required time                                 5.00
-----------------------------------------------------------
data required time                                 5.00
data arrival time                                 -5.13
-----------------------------------------------------------
slack (MET)                                        0.13
```

Clock Gating with Clock Inversion

Figure 10-20 shows another clock gating example where the clock to the flip-flop is inverted and the output of the flip-flop is the gating signal. Since the gating cell is an *and* cell, the gating signal must switch only when the clock signal at the *and* cell is low. This defines the setup and hold clock gating checks.

Here is the clock gating setup timing report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by MCLK')
Endpoint: UAND0
  (rising clock gating-check end-point clocked by MCLK')
Path Group: **clock_gating_default**
Path Type: max

Point                                   Incr       Path
-----------------------------------------------------------
clock MCLK' (rise edge)                 5.00       5.00
clock source latency                    0.00       5.00
MCLK (in)                               0.00       5.00 f
UINV0/ZN (INV  )                        0.02       5.02 r
UDFF0/CK (DF  )                         0.00       5.02 r
UDFF0/Q (DF  )                          0.13       5.15 f
UAND0/A1 (AN2  )                        0.00       5.15 f
data arrival time                                  5.15

clock MCLK' (rise edge)                15.00      15.00
```

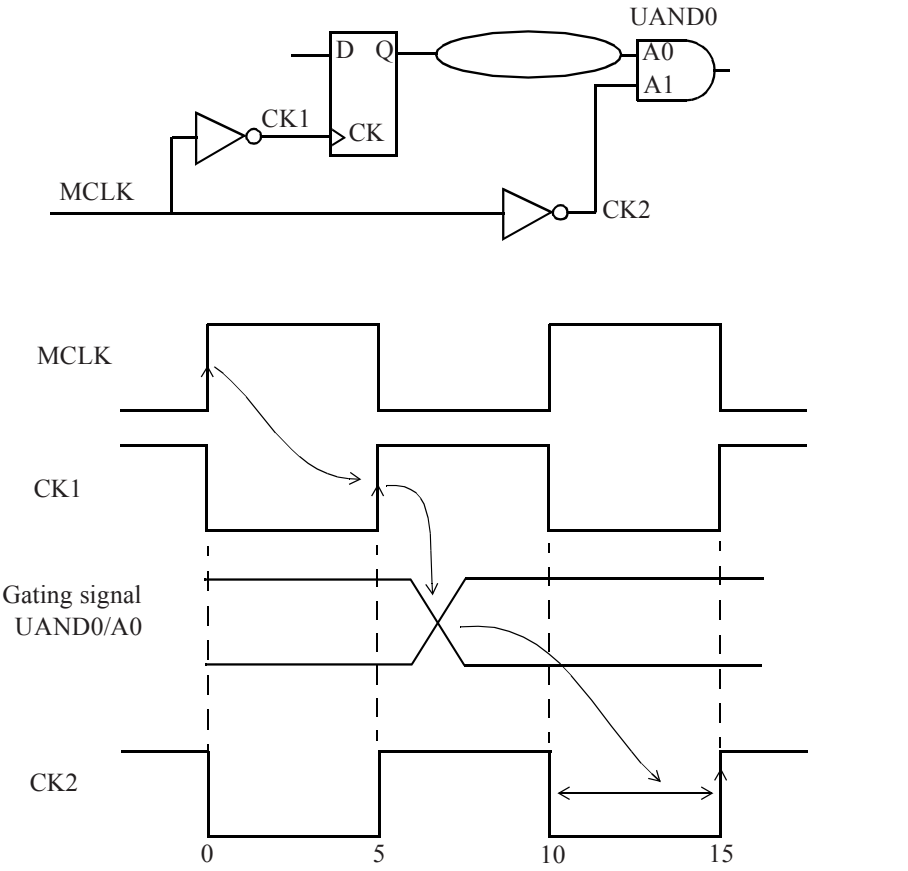


Figure 10-20 *Clock gating example with clock inversion.*

```
clock source latency                          0.00       15.00
MCLK (in)                                     0.00       15.00 f
UINV1/ZN (INV  )                              0.02       15.02 r
UAND0/A2 (AN2  )                              0.00       15.02 r
clock gating setup time                       0.00       15.02
data required time                                       15.02
-----------------------------------------------------------------
data required time                                       15.02
```

```
data arrival time                                      -5.15
-------------------------------------------------------------
slack (MET)                                             9.87
```

Notice that the setup check validates if the data changes before the edge on *MCLK* at time 15ns. Here is the clock gating hold timing report.

```
Startpoint: UDFF0
  (rising edge-triggered flip-flop clocked by MCLK')
Endpoint: UAND0
  (rising clock gating-check end-point clocked by MCLK')
Path Group: **clock_gating_default**
Path Type: min

Point                                    Incr       Path
-------------------------------------------------------------
clock MCLK' (rise edge)                  5.00       5.00
clock source latency                     0.00       5.00
MCLK (in)                                0.00       5.00 f
UINV0/ZN (INV  )                         0.02       5.02 r
UDFF0/CK (DF  )                          0.00       5.02 r
UDFF0/Q (DF  )                           0.13       5.15 r
UAND0/A1 (AN2  )                         0.00       5.15 r
data arrival time                                   5.15

clock MCLK' (fall edge)                 10.00      10.00
clock source latency                     0.00      10.00
MCLK (in)                                0.00      10.00 r
UINV1/ZN (INV  )                         0.01      10.01 f
UAND0/A2 (AN2  )                         0.00      10.01 f
clock gating hold time                   0.00      10.01
data required time                                 10.01
-------------------------------------------------------------
data required time                                 10.01
data arrival time                                  -5.15
-------------------------------------------------------------
slack (VIOLATED)                                   -4.86
```

The hold check validates whether the data (gating signal) changes before the falling edge of *MCLK* at time 10ns.

In the event that the gating cell is a complex cell and the setup and hold checks are not obvious, the *set_clock_gating_check* command can be used to specify a setup and hold check on the gating signal that gates a clock signal. The setup check validates that the gating signal is stable before the active edge of the clock signal. A setup failure can cause a glitch to appear at the gating cell output. The hold check validates that the gating signal is stable at the inactive edge of the clock signal. Here are some examples of the *set_clock_gating_check* specification.

```
set_clock_gating_check -setup 2.4 -hold 0.8 \
  [get_cells U0/UXOR1]
# Specifies the setup and hold time for the clock
# gating check at the specified cell.
set_clock_gating_check -high [get_cells UMUX5]
# Check is performed on high level of clock. Alternately, the
-low option can be used for an active-low clock gating check.
```

10.6 Power Management

Managing the power is an important aspect of any design and how it is implemented. During design implementation, a designer typically needs to evaluate different approaches for trade-off between speed, power and area of the design.

As described in Chapter 3, the power dissipated in the logic portion of the design is comprised of leakage power and the active power. In addition, the analog macros and the IO buffers (especially those with active termination) can dissipate power which is not activity dependent and is not leakage. In this section, we focus on the tradeoffs for power dissipated in the logic portion of the design.

In general, there are two considerations for managing the power contributions from the digital logic comprised of standard cells and memory macros:

- *To minimize the total active power of the design*. A designer would ensure that the total power dissipation stays within the available power limit. There may be different limits for different operating modes of the design. In addition, there can also be different limits from different power supplies used in the design.
- *To minimize the power dissipation of the design in standby mode*. This is an important consideration for battery operated devices (for example, cell phone) where the goal is to minimize the power dissipation in standby mode. The power dissipation in standby mode is leakage power plus any power dissipation for the logic that is active in standby mode. As discussed above, there may be other modes such as *sleep mode*, with different constraints on power.

This section describes various approaches for power management. Each of these approaches has its pros and cons which are described herein.

10.6.1 Clock Gating

As described in Chapter 3, clock activity at the flip-flops contributes to a significant component of the total power. A flip-flop dissipates power due to clock toggle even when the flip-flop output does not switch. Consider the example in Figure 10-21(a) where the flip-flop receives new data only when the enable signal *EN* is active otherwise it retains the previous state. During the time *EN* signal is inactive, the clock toggling at the flip-flop do not cause any output change though the clock activity still results in the power dissipated inside the flip-flop. The purpose of clock gating is to minimize this contribution by eliminating the clock activity at the flip-flop during clock cycles when the flip-flop input is not active. The logic restructuring through clock gating introduces gating of the clock at the flip-flop pin. An example of the transformation due to clock gating is illustrated in Figure 10-21.

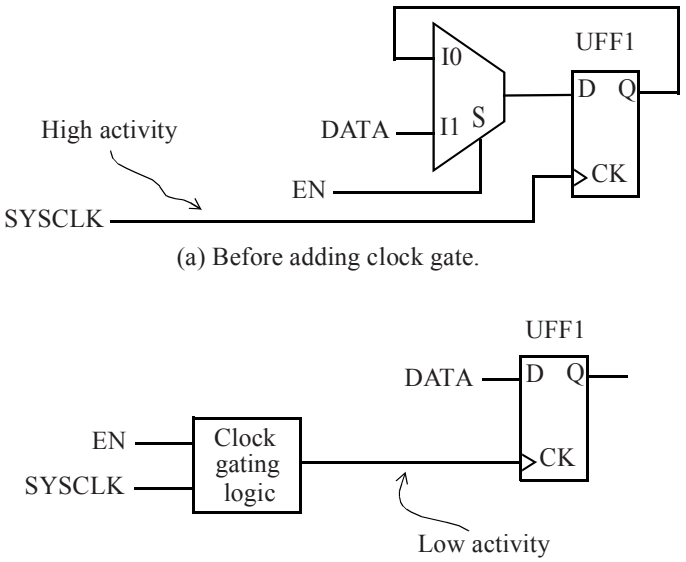


Figure 10-21 *Adding clock gate to save flip-flop power.*

The clock gating thus ensures that the clock pin of the flip-flop toggles only when new data is available at its data input.

10.6.2 Power Gating

Power gating involves gating off the power supply so that the power to the inactive blocks can be turned off. This procedure is illustrated in Figure 10-22, where a **footer** (or a **header**) MOS device is added in series with the power supply. The control signal *SLEEP* is configured so that the footer (or header) MOS device is *on* during normal operation of the block. Since the power gating MOS device (footer or header) is *on* during normal operation, the block is powered and it operates in normal functional mode. During inactive (or sleep) mode of the block, the gating MOS device (footer or header) is turned off which eliminates any active power dissipation in the logic block. The footer is a large NMOS device between the actual ground and

the ground net of the block which is controlled through power gating. The header is a large PMOS device between the actual power supply and the power supply net of the block which is controlled through power gating. During sleep mode, the only power dissipated in the block is the leakage through the footer (or header) device.

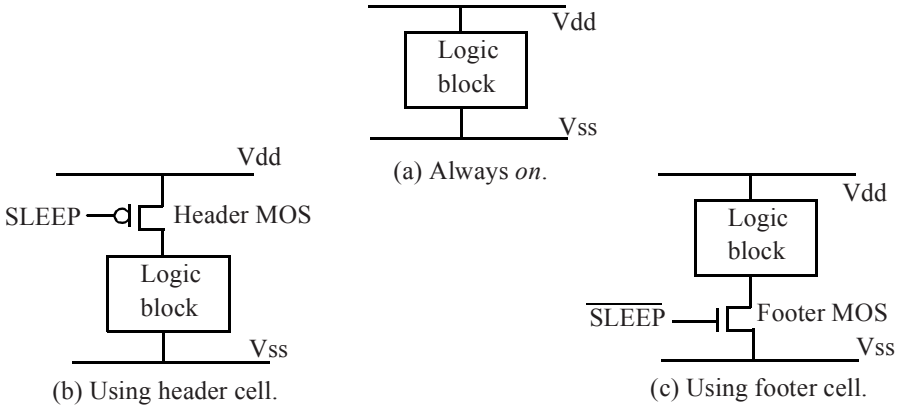


Figure 10-22 *Cutting off power to an inactive logic block using a header or a footer device.*

The footers or headers are normally implemented using multiple power gating cells which correspond to multiple MOS devices in parallel. The footer and header devices introduce a series *on* resistance to the power supply. If the value of the *on* resistance is not small, the IR drop through the gating MOS device can affect the timing of the cells in the logic block. While the primary criteria regarding the size of the gating devices is to ensure that the *on* resistance value is small, there is a trade-off as the power gating MOS devices determine the leakage in the inactive or sleep mode.

In summary, there should be adequate number of power gating cells in parallel to ensure minimal IR drop from the series *on* resistance in active mode. However, the leakage from the gating cells in the inactive or sleep mode is also a criteria in choosing the number of power gating cells in parallel.

10.6.3 Multi Vt Cells

As described in Chapter 3 (Section 3.8), the multi Vt cells are used to trade-off speed with leakage. The high Vt cells have less leakage, though these are slower than the standard Vt cells which are faster but have higher leakage. Similarly, the low Vt cells are faster than standard Vt cells but the leakage is also correspondingly higher.

In most designs, the goal is to minimize the total power while achieving the desired operational speed. Even though leakage can be a significant component of the total power, implementing a design with only high Vt cells to reduce leakage can increase the total power even though the leakage contribution may be reduced. This is because the resulting design implementation may require many more (or higher strength) high Vt cells to achieve the required performance. The increase in equivalent gate count can increase the active power much more than the reduction in leakage power due to use of high Vt cells. However, there are scenarios where the leakage is a dominant component of the total power; in such cases, a design with high Vt cells can result in reduction of the total power. The above trade-off between cells with different Vt in terms of their speed and leakage needs to be utilized suitably since it is dependent on the design and its switching activity profile. Two scenarios of a high performance block are illustrated below where the implementation approach can be different depending on whether the block is very active or has low switching activity.

High Performance Block with High Activity

This scenario is of a high performance block with high switching activity and the power is dominated by the active power. For such blocks, focussing only on reducing leakage power can cause the total power to increase even though the leakage contribution may be minimized. In such cases, the initial design implementation should use standard Vt (or low Vt) cells to meet the desired performance. After the required timing is achieved, the cells along the paths which have positive timing slack can be changed into high Vt cells so that the leakage contribution is reduced while still meeting the timing requirement. Thus, in the final implementation, the standard Vt (or low Vt) cells are used only along the critical or hard to achieve timing

paths, whereas cells along the non-critical timing paths can be high Vt cells.

High Performance Block with Low Activity

This scenario is of a high performance block with very low switching activity so that the leakage power is a significant component of the total power. Since the block has low activity, the active power is not a major component for the total power of the design. For such blocks, the initial implementation attempts to use only high Vt cells in the combinational logic and flip-flops. An exception is the clock tree which is always active and therefore is built with standard Vt (or low Vt) cells. After the initial implementation with only high Vt cells, there may be some timing paths where the required timing cannot be achieved. The cells along such paths are then replaced with standard Vt (or low Vt) cells to achieve the required timing performance.

10.6.4 Well Bias

The **well bias** refers to adding a small voltage bias to the P-well or N-well used for the NMOS and PMOS devices respectively. The bulk (or P-well) connection for the NMOS device shown in Figure 2-1 is normally connected to the ground. Similarly, the bulk (or N-well) connection for the PMOS device shown in Figure 2-1 is normally connected to the power (*Vdd*) rail.

The leakage power can be reduced significantly if the well connections have a slight negative bias. This means that the P-well for the NMOS devices is connected to a small negative voltage (such as -0.5V). Similarly, the N-well connection for the PMOS devices is connected to a voltage above the power rail (such as *Vdd* + 0.5V). By adding a well bias, the speed of the cell is impacted; however the leakage is reduced substantially. The timing in the cell libraries are generated by taking the well bias into account.

The drawback of using well bias is that it requires additional supply levels (such as -0.5V and *Vdd*+0.5V) for the P-well and N-well connections.

10.7 Backannotation

10.7.1 SPEF

How does STA know what the parasitics of the design are? Quite often, this information is extracted by using a parasitic extraction tool and this data is read in the form of SPEF by the STA tool. Detailed information and the format of the SPEF are described in Appendix C.

An STA engine inside a physical design layout tool also behaves similarly, except that the extraction information is written to an internal database.

10.7.2 SDF

In some cases, the delays of the cells and interconnect are computed by another tool and these are read in for STA via SDF. The advantage of using SDF is that the cell delays and interconnect delays no longer need to be computed - as these come from the SDF directly, and consequently STA can focus on the timing checks. However, the disadvantage of this delay annotation is that STA cannot perform crosstalk computation as the parasitic information is missing. SDF is the mechanism normally used to pass delay information to simulators.

Detailed information and the format of SDF are described in Appendix B.

10.8 Sign-off Methodology

STA can be run for many different scenarios. The three main variables that determine a scenario are:

- Parasitics corners (RC interconnect corners and operating conditions used for parasitic extraction)
- Operating mode
- PVT corner

Parasitic Interconnect Corners

Parasitics can be extracted at many corners. These are mostly governed by the variations in the metal width and metal etch in the manufacturing process. Some of these are:

- *Typical:* This refers to the nominal values for interconnect resistance and capacitance.
- *Max C*: This refers to the interconnect corner which results in maximum capacitance. The interconnect resistance is smaller than at *typical* corner. This corner results in largest delay for paths with short nets and can be used for max path analysis.
- *Min C*: This refers to the interconnect corner which results in minimum capacitance. The interconnect resistance is larger than at *typical* corner. This corner results in smallest delay for paths with short nets and can be used for min path analysis.
- *Max RC*: This refers to the interconnect corner which maximizes the interconnect RC product. This typically corresponds to larger etch which reduces the trace width. This results in largest resistance but corresponds to smaller than typical capacitance. Overall, this corner has the largest delay for paths with long interconnects and can be used for max path analysis.
- *Min RC*: This refers to the interconnect corner which minimizes the interconnect RC product. This typically corresponds to smaller etch which increases the trace width. This results in smallest resistance but corresponds to larger than typical capacitance. Overall, this corner has the smallest path delay for paths with long interconnects and can be used for min path analysis.

Based upon the interconnect R and C for various corners described above, an interconnect corner with larger C results in smaller R and a corner with smaller C results in larger R. Thus, the R partially compensates for the C across various interconnect corners. This implies that no single corner maps to an extreme value (worst-case or best-case) for path delay for all types of nets. The path delay using *Cworst* / *Cbest* corners is extreme only for short nets while *RCworst* / *RCbest* corners is extreme only for long nets. The *typical* interconnect corner is often the extreme in terms of path delay

for nets with average length. Thus, designers often choose to verify the timing at various interconnect corners described above. However, even the verification at each corner does not cover all possible scenarios since different metal layers can actually be at different interconnect corners independently - for example, *Max C* corner for *METAL2*, *Max RC* corner for *METAL1*, and so on. Statistical timing analysis described in Section 10.9 offers a mechanism for static timing analysis where different metal layers can be at different interconnect corners.

Operating Modes

The operating mode dictates the operation of the design. Various operating modes for a design can be:

- Functional mode 1 (for e.g. high-speed clocks)
- Functional mode 2 (for e.g. slow clocks)
- Functional mode 3 (for e.g. sleep mode)
- Functional mode 4 (for e.g. debug mode)
- Test mode 1 (for e.g. scan capture mode)
- Test mode 2 (for e.g. scan shift mode)
- Test mode 3 (for e.g. bist mode)
- Test mode 4 (for e.g. jtag mode).

PVT Corners

The PVT corners dictate at what conditions the STA analysis takes place. The most common PVT corners are:

- WCS (slow process, low power supply, high temperature)
- BCF (fast process, high power supply, low temperature)
- Typical (typical process, nominal power supply, nominal temperature)
- WCL (worst-case slow at cold - slow process, low power supply, low temperature)
- Or any other point in the PVT domain.

STA analysis can be performed for any scenario. A scenario here refers to a combination of the interconnect corner, operating mode, and PVT corner described above.

Multi-Mode Multi-Corner Analysis

Multi-mode multi-corner (MMMC) analysis refers to performing STA across multiple operating modes, PVT corners and parasitic interconnect corners at the same time. For example, consider a DUA that has four operating modes (*Normal*, *Sleep*, *Scan shift*, *Jtag*), and is being analyzed at three PVT corners (*WCS*, *BCF*, *WCL*) and three parasitic interconnect corners (*Typical*, *Min C*, *Min RC*) as shown in Table 10-23.

PVT corner/ Parasitic corner	*WCS*	*BCF*	*WCL*
Typical	1: Normal/Sleep/ Scan shift/ Jtag	2: Normal/ Sleep/ Scan shift	3: Normal/ Sleep
Min C	4: Not required	5: Normal/ Sleep	6: Not required
Min RC	7: Not required	8: Normal/ Sleep	9: Not required

Table 10-23 *Multiple modes and corners used during timing sign-off.*

There are a total of thirty six possible scenarios at which all timing checks, such as setup, hold, slew, and clock gating checks can be performed. Running STA for all thirty six scenarios at the same time can be prohibitive in terms of runtime depending upon the size of the design. It is possible that a scenario may not be necessary as it may be included within another scenario, or a scenario may not be required. For example, the designer may determine that scenarios 4, 6, 7 and 9 are not relevant and thus are not required. Also, it may not be necessary to run all modes in one corner, such as *Scan shift* or *Jtag* modes may not be needed in scenario 5. STA could be run on a single scenario or on multiple scenarios concurrently if multi-mode multi-corner capability is available.

The advantage of running multi-mode multi-corner STA is of savings in runtime and complexity in setting up the analysis scripts. Additional savings in an MMMC scenario is that the design and parasitics need to be loaded only once or twice as opposed to loading these individually multiple times for each mode or corner. Such a job is also more amenable to running them on an LSF farm. Multi-mode multi-corner has a bigger advantage in an optimization flow where the optimization is done across all scenarios such that fixing timing violations in one scenario does not introduce timing violations in another scenario.

For IO constraints, the *-add_delay* option can be used with multiple clock sources to analyze different modes in one run, such as scan or bist modes, or different operating modes in a PHY[1] corresponding to different speeds. Often each mode is analyzed in a separate run, but not always.

It is not unusual to find a design with a large number of clocks that requires tens of independent runs to cover every mode in max and min corners, and including the effect of crosstalk and noise.

10.9 Statistical Static Timing Analysis

The static timing analysis techniques described thus far are deterministic since the analysis is based upon fixed delays for all timing arcs in the design. The delay of each arc is computed based upon the operating conditions along with the process and interconnect models. While there may be multiple modes and multiple corners, the timing path delays for a given scenario are obtained deterministically.

In practice, the WCS or BCF for the process and operating corner conditions typically used during the STA correspond to the extreme 3σ corners[2]. The timing libraries are based upon the process corner models provided by the foundry and characterized with the operating conditions which result

1. Physical layer interface IP block such as a 10G PHY.
2. The σ here refers to standard deviation of an independent variable modeled statistically.

in the corresponding corner for the timing values of the cells. For example, the best-case fast library is characterized using fast process models, highest power supply and lowest temperature.

10.9.1 Process and Interconnect Variations

Global Process Variations

The global process variations, which are also called **inter-die device variations**, refer to the variations in the process parameters which impact all devices on a die (or wafer). See Figure 10-24. This depicts that all devices on a die are impacted similarly by these process variations - every device on a die will be *slow* or *fast* or anywhere in between. Thus, the variations modeled by the global process parameters are intended to capture the variations from die to die.

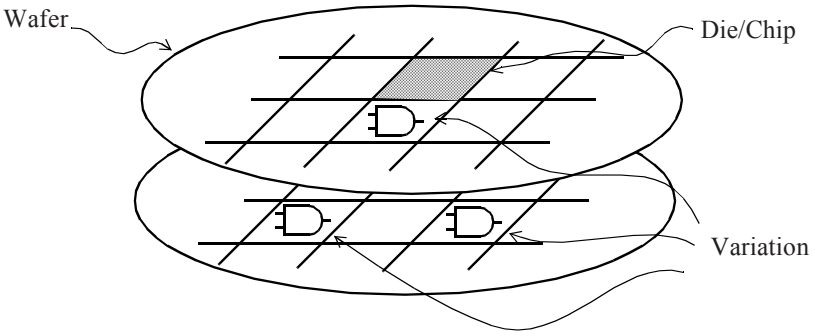


Figure 10-24 *Inter-die process variations.*

An illustration of the variations of a global parameter value (say *g_par1*) is shown in Figure 10-25. For example, the parameter *g_par1* may correspond to *IDSsat* (device saturation current) for a standard[1] NMOS device. Since this is a global parameter, all NMOS devices in all cell instances of a die will correspond to the same value of *g_par1*. This can alternately be stated

1. The standard device here means a device with fixed length and width.

as follows. The variations in *g_par1* for all cell instances are fully correlated or the variations in *g_par1* on a die track each other. Note that there would be other global parameters (*g_par2*, . . .) which may, for example, model the PMOS device saturation current and other relevant variables.

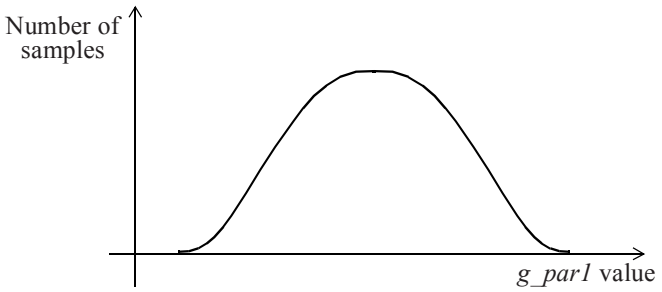


Figure 10-25 *Variations in a global parameter.*

Different global parameters (*g_par1, g_par2*, . . .) are uncorrelated. The variations in different global parameters do not track each other which means that the *g_par1* and *g_par2* parameters vary independently of each other; in a die *g_par1* may be at its maximum while *g_par2* may be at its minimum.

In the deterministic (that is, non-statistical) analysis, the slow process models may correspond to the $+3\sigma$ corner condition for the inter-die variations. Similarly, the fast process models may correspond to the -3σ corner condition for the inter-die variations.

Local Process Variations

The local process variations, which are also called **intra-die device variations**, refer to the variations in the process parameters which can affect the devices differently on a given die. See Figure 10-26. This implies that identical devices on a die placed side by side may have different behavior on the same die. The variations modeled by the local process variations are intended to capture the random process variations within the die.

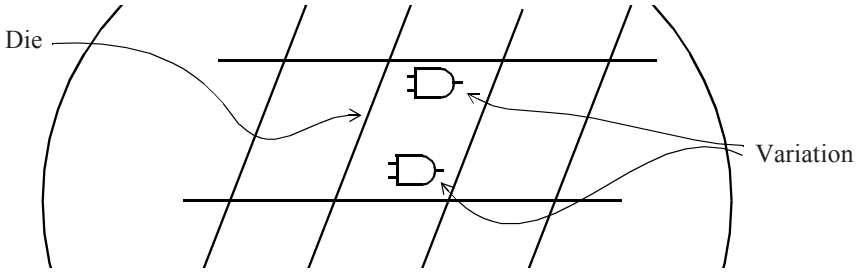


Figure 10-26 *Intra-die device variation.*

An illustration of the variations in a local process parameter is depicted in Figure 10-27. The local parameter variations on a die do not track each other and their variations from one cell instance to another cell instance are uncorrelated. This means that a local parameter may have different values for different devices on the same die. For example, different NAND2 cell instances on a die may see different local process parameter values. This can cause different instances of the same NAND2 cell to have different delay values even if other parameters such as input slew and output loading are identical.

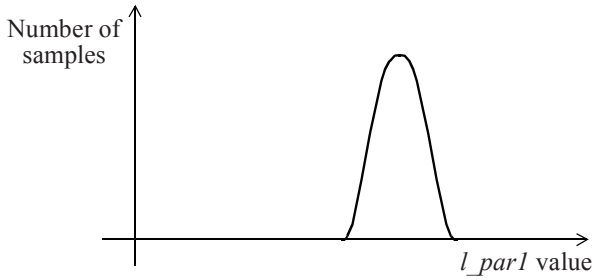


Figure 10-27 *Variations in local process parameter.*

An illustration of the variations in the NAND2 cell delay caused by global and local variations is depicted in Figure 10-28. The figure illustrates that the global parameter variations cause larger delay variation than the local parameter variations.

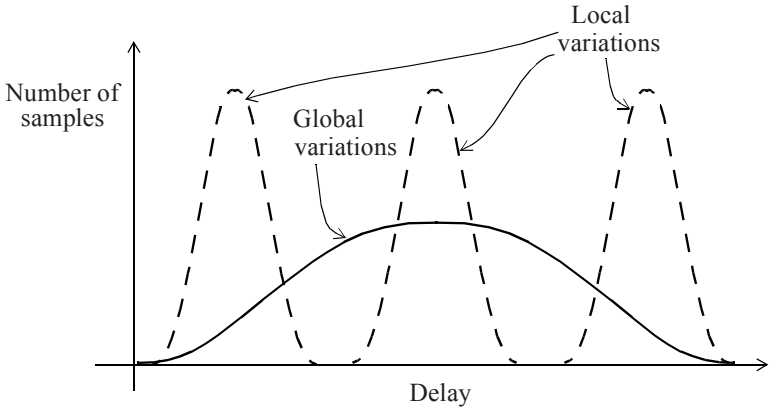


Figure 10-28 *Variation in cell delay due to global and local process variations.*

The local process variations are one of the variations intended to be captured in the analysis using OCV modeling, described in Section 10.1. Since statistical timing models normally include the local process variations, the OCV analysis using statistical timing models should not include the local process variation in the OCV setting.

Interconnect Variations

As described in Section 10.8, there are various interconnect corners which represent the parameter variations of each metal layer affecting the interconnect resistance and capacitance values. These parameter variations are generally the thicknesses of the metal and the dielectric, and the metal etch which affects the width and spacing of the metal traces in various metal layers. In general, the parameters affecting a metal impact the parasitics of all traces in that metal layer but have minimal or no effect on the parasitics of the traces in other metal layers.

The interconnect corners described in Section 10.8 model the interconnect variations so that all the metal layers map to the same interconnect corner. The interconnect variations when modeled statistically allow each metal

layer to vary independently. The statistical approach models all possible combinations of variations in the interconnect space and thus models variations which may not be captured by analyzing only at the specified interconnect corners. For example, it is possible that the launch path of a clock tree is in *METAL2*, whereas the capture path of the clock tree is in *METAL3*. Timing analysis at the traditional interconnect corners considers various corners which vary all metals together and thus cannot model the scenario where the *METAL2* is at a corner which results in max delay, and the *METAL3* is at a corner which results in min delay. Such a combination corresponds to the worst-case scenario for the setup paths and can only be captured by modeling the interconnect variations statistically.

10.9.2 Statistical Analysis

What is SSTA?

The modeling of variations described above is feasible if the cell timing models and the interconnect parasitics are modeled statistically. Apart from delay, the pin capacitance values at the inputs of the cells are also modeled statistically. This implies that the timing models are described in terms of mean and standard deviations with respect to process parameters (global and local). The interconnect resistances and capacitances are described in terms of mean and standard deviations with respect to interconnect parameters. The delay calculation procedures (described in Chapter 5) obtain the delays of each timing arc (cell as well as interconnect) which are then represented by mean and standard deviations with respect to various parameters. Thus, every delay is represented by a mean and N standard deviations (where N is the number of independent process and interconnect parameters modeled statistically).

Since the delays through individual timing arcs are expressed statistically, the statistical static timing analysis (SSTA) procedure combines the delays of the timing arcs to obtain the path delay which is also expressed statistically (with mean and standard deviations). The SSTA maps the standard deviations with respect to the independent process and interconnect parameters to obtain the overall standard deviation of the path delay. For example, consider the path delay comprised from two timing arcs as shown

in Figure 10-29. Since each delay component has its variations, the variations are combined differently depending upon whether these are correlated or uncorrelated. If the variations are from the same source (such as caused by *g_par1* which track each other), the σ of the path delay is simply equal to ($\sigma_1 + \sigma_2$). However, if the variations are uncorrelated (such as due to *l_par1*), the σ of the path delay is equal to *sqrt*($\sigma_1^2 + \sigma_2^2$), which is smaller than ($\sigma_1 + \sigma_2$). The phenomenon of smaller σ for the path delay when modeling local (uncorrelated) process variations is also referred to as statistical cancellation of the individual delay variations.

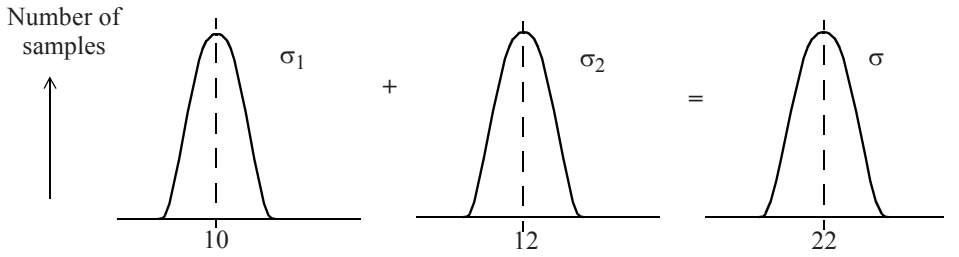


Figure 10-29 *Path delay comprised of variations in components.*

For a real design, both correlated as well as uncorrelated variations are modeled, and thus the contributions from both of these types of variations need to be combined appropriately.

The clock path delays for launch and capture clock are also expressed statistically in the same manner. Based upon the data and clock path delays, the slack is obtained as a statistical variable with its nominal value as well as standard deviation.

Assuming normal distribution, effective minimum and maximum values corresponding to (*mean* +/- 3σ) can be obtained. The (*mean* -/+ 3σ) corresponds to 0.135% and 99.865% quantile values of the normal distribution shown in Figure 10-30. The 0.135% quantile means that only 0.135% of the resulting distribution is smaller than this value (*mean* - 3σ); similarly 99.865% quantile means that 99.865% of the distribution is smaller than this value or only 0.135% (100% - 99.865%) of the distribution is larger than this

value (*mean* + 3σ). The effective lower and upper bounds are referred to as the quantiles in an SSTA report and the designer can select the quantile value used in the analysis, such as 0.5% or 99.5% which corresponds to (*mean* -/+ 2.576 σ).

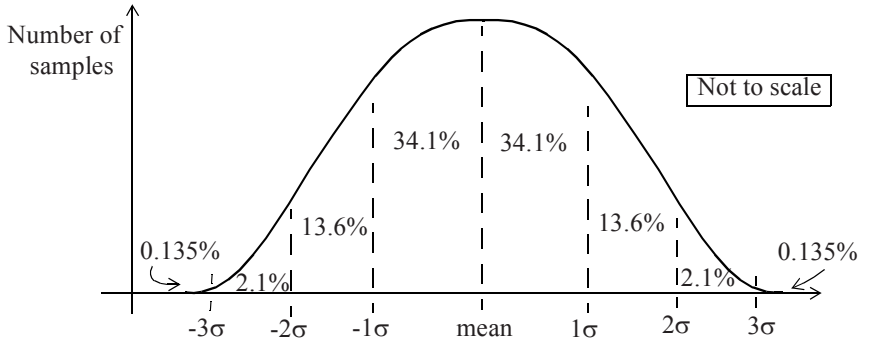


Figure 10-30 *Normal distribution.*

For noise and crosstalk analysis (Chapter 6), the path delays as well as timing windows used are modeled statistically with mean and standard deviations with respect to various parameters.

Based upon the path slack distribution, the SSTA reports the mean, standard deviation and the quantile values of slack for each path whereby the passing or failing can be determined based upon the required statistical confidence.

Statistical Timing Libraries

In the SSTA approach, the standard cell libraries (and libraries for other macros used in a design) provide timing models at various environmental conditions. For example, the analysis at min *Vdd* and high temperature corner utilizes libraries which are characterized at this condition but the process parameters are modeled statistically. The library includes timing

models for the nominal parameter values as well as with parameter variations. For *N* process parameters, a statistical timing library characterized at a power supply of 0.9V and 125C may include the following:

- Timing models with nominal process parameters, *plus the following with respect to each of the process parameters.*
- Timing models with respect to a parameter *i* at (*nominal* + 1σ), the other parameters being held at nominal value.
- Timing models with respect to a parameter *i* at (*nominal* - 1σ), the other parameters being held at nominal value.

For a simplified example scenario with two independent process parameters, the timing models are characterized with the nominal parameter values and also with the variations in parameter values as illustrated in Figure 10-31.

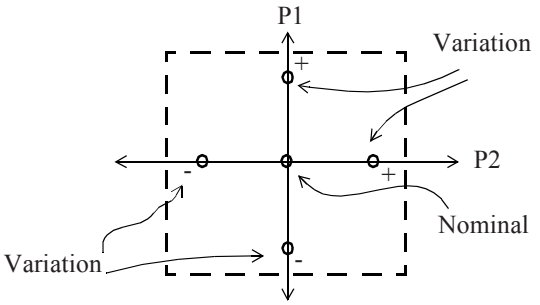


Figure 10-31 *N-dimensional process space.*

Statistical Interconnect Variations

There are three independent parameters for each metal layer:

- *Metal etch.* This controls the metal width as well as spacing to the neighboring conductor. A large etch in a metal layer reduces the width (which increases resistance) and increases the spacing to

the neighboring traces (which reduces the coupling capacitance to the neighboring traces). This parameter is expressed as a variation in the width of the conductor.

- *Metal thickness*. Thicker metal implies larger capacitance to the layers below. This is expressed as a variation in the thickness of the conductor.
- *IMD (Inter Metal Dielectric) thickness*. Larger IMD thickness reduces the coupling to the layers below. This parameter is expressed as a variation in the IMD thickness.

SSTA Results

The output results in the statistical analysis provide path slack in terms of its mean and effective corner values. An example of the SSTA report for a setup check (max path analysis) is shown below.

```
Path startpoint endpoint     quantile  sensitiv  mean   stddev
-------------------------------------------------------------------------
0    DBUS[7]   PDAT[5]       -0.43     50.00     0.86   0.43


Path attribute          quantile    sensitiv      mean     stddev
-----------------------------------------------------------------
arrival                    6.74      7.88         5.45     0.43
slack                      -0.43     50.00        0.86     0.43
required                   6.31      0.00         6.31     0.00
startpoint_clock_latency   0.25      0.00         0.25     0.00
endpoint_clock_latency     0.33      0.00         0.33     0.00


Point arrival             quantile    sensitiv  mean   stddev incr
-------------------------------------------------------------
DBUS[7]/CP (SDFQD2)       0.25        0.00      0.25   0.00
DBUS[7]/Q (SDFQD2)        0.66        4.04      0.61   0.02   0.02
U1/ZN (INVD2)             0.80        4.09      0.72   0.03   0.01
. . .
U22/ZN (NR3D4)            2.20        5.82      1.89   0.11   0.03
U23/Z (AN2D3)             2.41        6.40      2.03   0.13   0.02
U24/ZN (CKBD3)            2.53        7.10      2.10   0.15   0.02
```

```
U25/Z (AO23D2)              2.89        8.65        2.31   0.20   0.05
U26/ZN (IND2D4)             2.98        8.84        2.36   0.21   0.01
U27/Z (MUX3D4)              3.26        8.89        2.58   0.23   0.02
. . .
U51/ZN (ND2D4)              6.74        7.88        5.45   0.43   0.02
PDAT[5]/D (SDFQD1)          6.74        7.88        5.45   0.43   0.00
```

The above report shows that while the mean of the timing path meets the requirement, the 0.135% quantile value has a violation by 0.43ns - path slack quantile is -0.43ns. The path slack has a mean value of +0.86ns with 0.43ns standard deviation. This implies that +/- 2σ of the distribution meets the requirement. Since 95.5% of the distribution falls within 2σ variation, this implies that only 2.275% of the manufactured parts will have a timing violation (the remaining 2.275% of the distribution has large positive path slack). A 2.275% quantile setting will thus show a slack of 0 or no timing violation. The arrival time and the path slack distribution is depicted in Figure 10-32.

Note that the above report is for the setup path and thus the quantile column provides the upper bound quantile (for example +3σ value for path delay) - the hold path would specify the equivalent lower bound quantile (for example -3σ value). The *sensitiv* column in the report refers to the sensitivity which is the ratio of the standard deviation to the mean (expressed as a percentage). In terms of slack, smaller sensitivity is desired which means that a path passing at mean value continues to pass even with variations. The *incr* column specifies the incremental standard deviation for that line in the report.

With the statistical models for the cells and interconnect, the statistical timing approach analyzes the design at corner environment conditions and explores the space due to *process* and interconnect parameter variations. For example, a statistical analysis at worst-case VT (Voltage and Temperature) would explore the entire global *process* and interconnect space. Another statistical analysis at the best-case VT (Voltage and Temperature) would also explore the entire *process* and interconnect space. These analyses can be contrasted with the traditional corner analysis at the worst-case or the best-case PVT, each of which explores only a single point of process and interconnect space.

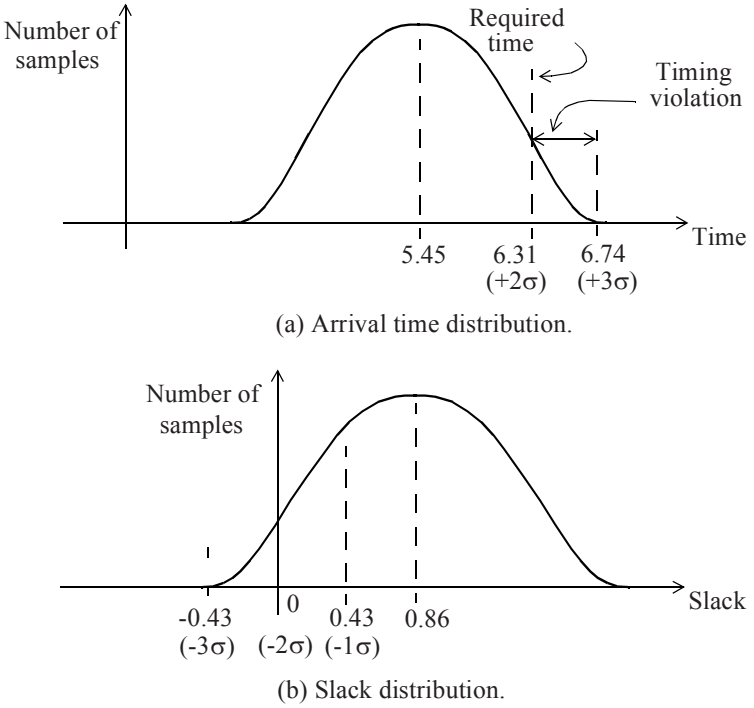


Figure 10-32 *Path delay and slack distribution.*

10.10 Paths Failing Timing?

In this section, we provide examples that highlight the critical aspects that a designer needs to focus on during debugging of STA results. Several of these examples contain only the relevant excerpts from the STA reports.

No Path Found

What if one is trying to obtain a path report and the STA reports that no path is found, or it provides a path report but the slack is infinite? In both of these cases, the situation likely occurs because:

i. the timing path is broken, or

ii. the path does not exist, or

iii. there is a false path.

In each of these cases, careful debugging of the constraints is required to identify what constraint causes the path to be blocked. One brute force option is to remove all false path settings and timing breaks and then see if the path can be timed. (A timing break is the removal of a timing arc from STA and is achieved by using the **set_disable_timing** specification as described in Section 7.10.)

Clock Crossing Domain

Here is a header of a path report.

```
Startpoint: IP_IO_RSTHN[0](input port clocked by SYS_IN_CLK)
Endpoint: X_WR_PTR_GEN/Q_REG
          (recovery check against rising-edge clock PX9_CLK)
Path Group: **async_default**
Path Type: max

Point                                        Incr       Path
-------------------------------------------------------------
clock SYS_IN_CLK (rise edge)                 6.00       6.00
. . .
IO_IO_RSTHN[0] (in)                          0.00       6.00 r
. . .
X_WR_PTR_GEN/Q_REG/CDN (DFCN  )              0.00       6.31 r
. . .

clock PX9_CLK (rise edge)                    12.00      12.00
. . .
```

```
library recovery time                    0.122      15.98
. . .
```

The first thing to notice is that this path starts from an input port and ends at the clear pin of a flip-flop, and a recovery check (see *library recovery time*) on the clear pin is being validated. The next thing to notice is that the path goes across two different clock domains, *SYS_IN_CLK*, the clock that launches the input, and *PX9_CLK*, the clock at the flip-flop whose recovery timing is being checked. Even though it is not apparent from the timing report, but from the design knowledge, one can examine if the two clocks are fully asynchronous and whether any paths between these two clock domains should be treated as false.

Lesson: Verify if the launch clock and capture clock and the paths between the two are valid.

Inverted Generated Clocks

When creating generated clocks, the *-invert* option needs to be used carefully. If a generated clock is specified using the *-invert* option, STA assumes that the generated clock at the specified point is of the type specified. However based upon the logic, it is possible that such a waveform cannot occur in the design. STA would normally provide an error or a warning message indicating that the generated clock is not realizable, however it will continue with the analysis and report the timing paths.

Consider Figure 10-33. Let us define a generated clock with *-invert* on the output of the cell *UCKBUF0*.

```
create_clock -name CLKM -period 10 -waveform {0 5} \
  [get_ports CLKM]
create_generated_clock -name CLKGEN -divide_by 1 -invert \
  -source [get_ports CLKM] [get_pins UCKBUF0/C]
```

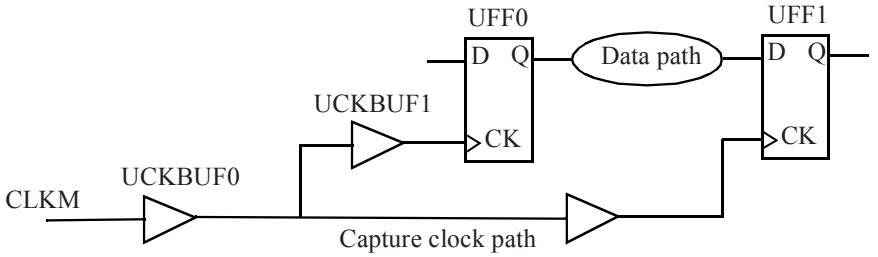


Figure 10-33 *Example of a generated clock.*

Here is the setup timing report based upon the above specifications.

```
Startpoint: UFF0
  (rising edge-triggered flip-flop clocked by CLKGEN)
Endpoint: UFF1
  (rising edge-triggered flip-flop clocked by CLKGEN)
Path Group: CLKGEN
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock CLKGEN (rise edge)                 5.00       5.00
clock network delay (ideal)              0.00       5.00
UFF0/CK (DF  )                           0.00       5.00 r
UFF0/Q (DF  ) <-                         0.14       5.14 f
UNOR0/ZN (NR2  )                         0.04       5.18 r
UBUF4/Z (BUFF  )                         0.05       5.23 r
UFF1/D (DF  )                            0.00       5.23 r
data arrival time                                   5.23

clock CLKGEN (rise edge)                15.00      15.00
clock network delay (ideal)              0.00      15.00
UFF1/CK (DF  )                                     15.00 r
library setup time                      -0.05      14.95
data required time                                 14.95
-----------------------------------------------------------
data required time                                 14.95
data arrival time                                  -5.23
```

```
-------------------------------------------------------------
slack (MET)                                          9.72
```

Notice that the STA faithfully assumes that the waveform at the output of cell *UCKBUF0* is the inverted clock of clock *CLKM*. Thus, the rise edge is at 5ns and the capture setup clock edge is at 15ns. Other than the fact that the rising edge of the clock is at 5ns instead of 0ns, it is not apparent from the timing report that something is wrong. It should be noted that since the error is on the common portion of both the launch and the capture clock paths, the setup and hold timing checks are indeed performed correctly. The warnings and the errors produced by STA need to be carefully analyzed and understood.

The important point to note is that STA will create the generated clock as specified whether it is realizable or not.

Now let us try to move the generated clock with the *-invert* option to the output of the cell *UCKBUF1* and see what happens.

```
create_clock -name CLKM -period 10 -waveform {0 5} \
  [get_ports CLKM]
create_generated_clock -name CLKGEN -divide_by 1 -invert \
  -source [get_ports CLKM] [get_pins UCKBUF1/C]
```

Here is the setup report.

```
Startpoint: UFF0
  (rising edge-triggered flip-flop clocked by CLKGEN)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max

Point                                    Incr       Path
-------------------------------------------------------------
clock CLKGEN (rise edge)                 5.00       5.00
clock network delay (ideal)              0.00       5.00
UFF0/CK (DF  )                           0.00       5.00 r
```

```
UFF0/Q (DF  ) <-                          0.14       5.14 f
UNOR0/ZN (NR2  )                          0.04       5.18 r
UBUF4/Z (BUFF  )                          0.05       5.23 r
UFF1/D (DF  )                             0.00       5.23 r
data arrival time                                    5.23

clock CLKM (rise edge)                   10.00      10.00
clock source latency                      0.00      10.00
CLKM (in)                                 0.00      10.00 r
UCKBUF0/C (CKB  )                         0.06      10.06 r
UCKBUF2/C (CKB  )                         0.07      10.12 r
UFF1/CK (DF  )                            0.00      10.12 r
clock uncertainty                        -0.30       9.82
library setup time                       -0.04       9.78
data required time                                   9.78
-----------------------------------------------------------
data required time                                   9.78
data arrival time                                   -5.23
-----------------------------------------------------------
slack (MET)                                          4.55
```

The path looks like a half-cycle path, but this is incorrect since there is no inversion on the clock path in the actual logic. Once again, STA assumes that the clock at the *UCKBUF1/C* pin is the one as specified in the *create_generated_clock* command. Hence the rising edge occurs at 5ns. The capture clock edge is running off clock *CLKM*, whose next rising edge occurs at 10ns. The hold path report below also contains a similar discrepancy as the setup path.

```
Startpoint: UFF0
  (rising edge-triggered flip-flop clocked by CLKGEN)
Endpoint: UFF1 (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: min

Point                                     Incr       Path
-----------------------------------------------------------
clock CLKGEN (rise edge)                  5.00       5.00
clock network delay (ideal)               0.00       5.00
UFF0/CK (DF  )                            0.00       5.00 r
```

```
UFF0/Q (DF  ) <-                          0.14       5.14 r
UNOR0/ZN (NR2  )                          0.02       5.16 f
UBUF4/Z (BUFF  )                          0.06       5.21 f
UFF1/D (DF  )                             0.00       5.21 f
data arrival time                                    5.21

clock CLKM (rise edge)                    0.00       0.00
clock source latency                      0.00       0.00
CLKM (in)                                 0.00       0.00 r
UCKBUF0/C (CKB  )                         0.06       0.06 r
UCKBUF2/C (CKB  )                         0.07       0.12 r
UFF1/CK (DF  )                            0.00       0.12 r
clock uncertainty                         0.05       0.17
library hold time                         0.01       0.19
data required time                                   0.19
-------------------------------------------------------------
data required time                                   0.19
data arrival time                                   -5.21
-------------------------------------------------------------
slack (MET)                                          5.03
```

Typically, the STA output will include an error or warning indicating that the generated clock is not realizable. The best way to debug these kind of improper paths is to actually draw the clock waveforms at the capture flip-flop and at the launch flip-flop and try to understand if the edges being shown are indeed valid.

Lesson: Check the edges of the capture and launch clocks to see if they are indeed what they should be.

Missing Virtual Clock Latency

Consider the following path report.

```
Startpoint: RESET_L (input port clocked by VCLKM)
Endpoint: NPIWRAP/REG_25
  (rising edge-triggered flip-flop clocked by CLKM)
Path Group: CLKM
Path Type: max
```

```
Point                                          Incr       Path
-------------------------------------------------------------------
clock VCLKM (rise edge)                        0.00       0.00
clock network delay                            0.00       0.00
input external delay                           2.55       2.55 f
RESET_L (in) <-                                0.00       2.55 f
. . .
NPIWRAP/REG_25/D (DFF )                        0.00       2.65 f
data arrival time                                         2.65

clock CLKM (rise edge)                        10.00      10.00
. . .
```

It is a path that starts from an input pin. Notice that the starting arrival time is listed as 0. This indicates that there was no latency specified on the clock *VCLKM* - the clock used to define the input arrival time on the input pin *RESET_L*; most probably this is a virtual clock, and that is why the arrival time is missing.

Lesson: When using virtual clocks, make sure latencies on virtual clocks are specified or are accounted for in the *set_input_delay* and *set_output_delay* constraints.

Large I/O Delays

When input or output paths have timing violations, the first thing to check is the latency on the clock used as reference to specify the input arrival time or the output required time. This is also applicable for the previous example.

The second thing to check is the input or output delays, that is, the input arrival time on an input path or the output required time on an output path. Quite often, one may find that these numbers are unrealistic for the target frequency. The input arrival time is usually the first value in the data path of the report, while the output required time is usually the last value in the data path of the report.

```
  . . .
  Point                                    Incr       Path
  -------------------------------------------------------------
  clock VIRTUAL_CLKM (rise edge)           0.00       0.00
  clock network delay                      0.00       0.00
  input external delay                    14.00      14.00 f
  PORT_NIP (in) <-                         0.00      14.00 f
  UINV1/ZN (INV  )                         0.34      14.34 r
  UAND0/Z (AN2  )                          0.61      14.95 r
  UINV2/ZN (INV  )                         0.82      15.77 f
  . . .
```

In this data path of an input failing path, notice the input arrival time of 14ns. In this particular case, there was an error in the input arrival time specification in that it was too large.

Lesson: When reviewing input or output paths, check if the external delay specified is reasonable.

Incorrect I/O Buffer Delay

When a path goes through an input or an output buffer, it is possible for an incorrect specification to cause large delay values for the input or output buffer delays. In the case shown below, notice the large output buffer delay of 18ns; this is caused by a large load value specified on the output pin.

```
Startpoint: UFF4 (rising edge-triggered flip-flop clocked by CLKP)
  Endpoint: ROUT (output port clocked by VIRTUAL_CLKP)
  Path Group: VIRTUAL_CLKP
  Path Type: max

  Point                                    Incr       Path
  -------------------------------------------------------------
  clock CLKP (rise edge)                   0.00       0.00
  clock source latency                     0.00       0.00
  CLKP (in)                                0.00       0.00 r
  UCKBUF4/C (CKB  )                        0.06       0.06 r
  UCKBUF5/C (CKB  )                        0.06       0.12 r
```

```
UFF4/CK (DFF )                          0.00       0.12 r
UFF4/Q (DFF )                           0.13       0.25 r
UBUF3/Z (BUFF  )                        0.09       0.33 r
IO_1/PAD:OUT (DDRII )                  18.00      18.33
ROUT (out)                              0.00      18.33 r
data arrival time                                 18.33
. . .
```

Lesson: Watch out for large delays on buffers caused by incorrect load specifications.

Incorrect Latency Numbers

When a timing path fails, one thing to check is if the latencies of the launch clock and the capture clock are reasonable, that is, ensure that the skew between these clocks is within acceptable limits. Either an incorrect latency specification or incorrect clock balancing during clock construction can cause large skew in the launch and capture clock paths leading to timing violations.

Lesson: Check if clock skew is within reasonable limits.

Half-cycle Path

As mentioned in an earlier example, one needs to check the clock domains of the failing path. Along with this, one may need to check the edges at which the launch and capture flip-flops are being clocked. In some cases, one may find a half-cycle path - a rise to fall path or a fall to rise path - and it may be unrealistic to meet timing with a half-cycle path, or maybe the half-cycle path is not real.

Lesson: Ensure that data path has sufficient time to propagate.

Large Delays and Transition Times

One key item is to check for unusually large values for the delays or transition times along the data path. Some of these can be due to:

- *High-fanout nets*: Nets which are not buffered properly.
- *Long nets*: Nets which need buffer insertion in between.
- *Low strength cells*: Cells which may not have been replaced because these are labeled as don't touch in the design.
- *Memory paths*: Paths that typically fail due to large setup times on memory inputs and large output delays on memory outputs.

Missing Multicycle Hold

For a multicycle *N* setup specification, it is common to see the corresponding multicycle *N-1* hold specification missing. Consequently, this can cause a large number of unnecessary delay cells to get inserted when a tool is fixing the hold violations.

Lesson: Always audit the hold violations *before* fixing to ensure that the hold violations that are being fixed are real.

Path Not Optimized

STA violations may be present on a path that has not been optimized yet. One can determine this situation by examining the data path. Are there cells with large delays? Can one manually improve the timing on the data path? Maybe the data path needs to be optimized more. It is possible that the tool is working on other worse violating paths.

Path Still Not Meeting Timing

If the data path appears to have good strong cells and if the path is still failing timing, one needs to examine the pins where the routing delay and wireload is high. This can be the next source of improvement. Maybe the cells can be moved closer and consequently the wireload and the wire routing delay can be decreased.

What if Timing Still Cannot be Met

One can utilize **useful skew** to help close the timing. Useful skew is where one purposely imbalances the clock trees, especially the launch and capture clock paths of a failing path so that the timing passes on that path. It typically means that the capture clock can be delayed so that the clock at the capture flip-flop arrives later when the data is ready. This does assume that there is enough slack on the succeeding data paths, that is, the data path for the next stage of flip-flop to flip-flop paths.

The reverse can also be attempted, that is, the launch clock path can be made shorter so that the data from the launch flip-flop is launched earlier to help meet the setup timing. Once again this can only be done if the preceding stage of flip-flop to flip-flop paths have the extra slack to give away.

Useful skew techniques can be used to fix both setup and hold violations. One disadvantage of this technique is that if the design has multiple modes of operation, then useful skew can potentially cause a problem in another mode.

10.11 Validating Timing Constraints

As chip size grows, there is more and more dependence on signing off timing with static timing analysis. The risk of relying only upon STA is that the STA is dependent on how good the timing constraints are. Therefore, validation of timing constraints becomes an important consideration.

Checking Path Exceptions

There are tools available that check the validity of false paths and multicycle paths based on the structure (netlist) of the design. These tools determine whether a given false path or multicycle path specification is valid. In addition, these tools may also be able to generate missing false path and multicycle path specifications based upon the structure of the design. However, some of the path exceptions generated by the tools may not be valid. This is because these tools determine the proof of a false path or a

multicycle path by the structure of the logic, typically using formal verification techniques, whereas a designer has a more in-depth knowledge of the functional behavior of the design. Thus the path exceptions generated by the tools need to be reviewed by the designer before accepting these and using them in STA. There may also be additional path exceptions that are based upon the semantic behavior of the design that have to be defined by the designer if the tool is unable to extract such exceptions.

The biggest risk in timing constraints are the path exceptions. Thus, false paths and multicycle paths should be determined after a careful analysis. In general, it is preferable to use a multicycle path as opposed to a false path. This ensures that the path in question is at least constrained by some amount. If a signal is sampled at a known or a predictable time, no matter how far out, use a multicycle path so that static timing analysis has some constraints to work with. False paths have the danger of causing timing optimization tools to completely ignore such paths, whereas in reality, they may indeed be getting sampled after some large number of clock cycles.

Checking Clock Domain Crossing

Tools are available to ensure that all clock domain crossings in a design are valid. These tools may also have the capability to automatically generate the necessary false path specifications. Such tools may also be able to identify illegal clock domain crossing, that is, cases where data is crossing two different clock domains without any clock synchronization logic. In such cases, the tools may provide the capability to automatically insert suitable clock synchronization logic where required. Note that not all asynchronous clock domain crossings require clock synchronizers. The requirement depends upon the nature of the data and whether it needs to be captured on the next cycle or a few cycles later.

An alternate way of checking asynchronous clock crossings using STA is to set a large clock uncertainty that is equal to the period of the sampling clock. This ensures that there will at least be some violations based upon which one can determine the appropriate path exceptions, or add the clock synchronization logic to the design.

Validating IO and Clock Constraints

Validating IO and clock constraints are still a challenge. Quite often timing simulations are performed to check the validity of all clocks in the design. System timing simulations are performed to validate the IO timing to ensure that the chip can communicate with its peripherals without any timing issues.

❑

APPENDIX

A

SDC

This appendix describes the SDC[1] format version 1.7. This format is primarily used to specify the timing constraints of a design. It does not contain any commands for any specific tool such as link and compile. It is a text file. It can be handwritten or created by a program and read in by a program. Some SDC commands are applicable to implementation or synthesis only. However all SDC commands are listed here.

SDC syntax is a TCL-based format, that is, all commands follow the TCL syntax. An SDC file contains the SDC version number at the beginning of the file. This is followed by the design constraints. Optional comments (a comment starts with the # character and ends at the end of the line) can be present in a SDC file interspersed with the design constraints. Long lines in

1. Synopsys Design Constraints. Reproduced here with permission from Synopsys, Inc.

design constraints can be split up across multiple lines using the backslash (\) character.

A.1 Basic Commands

These are the basic commands in SDC.

```
current_instance [instance_pathname]
# Sets the current instance of design. This allows other
# commands to set or get attributes from that instance.
# If no argument is supplied, then the current instance
# becomes the top-level.
Examples:
 current_instance /core/U2/UPLL
 current_instance ..    # Go up one hierarchy.
 current_instance       # Set to top.

expr arg1 arg2 . . . argn

list arg1 arg2 . . . argn

set variable_name value

set_hierarchy_separator separator
# Specifies the default hierarchy separator used within
# the SDC file. This can be overridden by using the -hsc
# option in the individual SDC commands where allowed.
Examples:
 set_hierarchy_separator /
 set_hierarchy_separator .

set_units [-capacitance cap_unit] [-resistance res_units]
  [-time time_unit] [-voltage voltage_unit]
  [-current current_unit] [-power power_unit]
```

```
# Specifies the units used in the SDC file.
Examples:
 set_units -capacitance pf -time ps
```

A.2 Object Access Commands

These commands specify how to access objects in a design instance.

```
all_clocks
# Returns a collection of all clocks.
Examples:
 foreach_in_collection clkvar [all_clocks] {
   . . .
 }
 set_clock_transition 0.150 [all_clocks]

all_inputs [-level_sensitive] [-edge_triggered]
  [-clock clock_name]
# Returns a collection of all input ports in the design.
Example:
 set_input_delay -clock VCLK 0.6 -min [all_inputs]

all_outputs [-level_sensitive] [-edge_triggered]
  [-clock clock_name]
# Returns a collection of all output ports in the design.
Example:
 set_load 0.5 [all_outputs]

all_registers [-no_hierarchy] [-clock clock_name]
  [-rise_clock clock_name] [-fall_clock clock_name]
  [-cells] [-data_pins] [-clock_pins] [-slave_clock_pins]
  [-async_pins] [-output_pins] [-level_sensitive]
  [-edge_triggered] [-master_slave]
# Returns the set of registers with the properties
```

```
# as specified, if any.
Examples:
 all_registers -clock DAC_CLK
 # Returns all registers clocked by clock DAC_CLK.

current_design [design_name]
# Returns the name of the current design. If specified with
# an argument, it sets the current design to the one
# specified.
Examples:
 current_design FADD   # Sets the current context to FADD.
 current_design        # Returns the current context.

get_cells [-hierarchical] [-hsc separator] [-regexp]
 [-nocase] [-of_objects objects] patterns
# Returns a collection of cells in the design that match the
# specified pattern. Wildcard can be used to match
# multiple cells.
Examples:
 get_cells RegEdge* # Returns all cells that
                    # match pattern.
 foreach_in_collection cvar [get_cells -hierarchical *] {
   . . .
 }  # Returns all cells in design by searching
    # recursively down the hierarchy.

get_clocks [-regexp] [-nocase] patterns
# Returns a collection of clocks in the design that match
# the specified pattern. When used in context such as -from
# or -to, it returns a collection of all flip-flops driven
# by the specified clocks.
Examples:
 set_propagated_clock [get_clocks SYS_CLK]
 set_multicycle_path -to [get_clocks jtag*]
```

```
get_lib_cells [-hsc separator] [-regexp] [-nocase]
  patterns
# Creates a collection of library cells that are currently
# loaded and those that match the specified pattern.
Example:
  get_lib_cells cmos13lv/AOI3*

get_lib_pins [-hsc separator] [-regexp] [-nocase]
  patterns
# Returns a collection of library cell pins that match the
# specified pattern.

get_libs [-regexp] [-nocase] patterns
# Returns a collection of libraries that are currently
# loaded in the design.

get_nets [-hierarchical] [-hsc separator] [-regexp]
  [-nocase] [-of_objects objects] patterns
# Returns a collection of nets that match the specified
# pattern.
Examples:
  get_nets -hierarchical *    # Returns list of all nets in
    # design by searching recursively down the hierarchy.
  get_nets FIFO_patt*

get_pins [-hierarchical] [-hsc separator] [-regexp]
  [-nocase] [-of_objects objects] patterns
# Returns a collection of pin names that match the
# specified pattern.
Examples:
  get_pins *
  get_pins U1/U2/U3/UAND/Z

get_ports [-regexp] [-nocase] patterns
# Returns a collection of port names (inputs and outputs)
# of design that match the specified pattern.
```

```
Example:
  foreach_in_collection port_name [get_ports clk*] {
    # For all ports that start with "clk".
    . . .
  }
```

Can an object such as a port be referenced without "getting" the object? When there is only one object with that name in the design, there really isn't any difference. However, when multiple objects share the same name, then using the **get_*** commands become more important. It avoids any possible confusion over which type of object is being referred to. Imagine a case where there is a net called *BIST_N1* and a port called *BIST_N1*. Consider the SDC command:

```
set_load 0.05 BIST_N1
```

The question is which *BIST_N1* is being referred to? The net or the port? It is best in most cases to explicitly qualify the type of object, such as in:

```
set_load 0.05 [get_nets BIST_N1]
```

Consider another example of a clock *MCLK* and a port called *MCLK*, and the following SDC command:

```
set_propagated_clock MCLK
```

Does the object refer to the port called *MCLK*, or to the clock called *MCLK*? In this particular case, it refers to the clock since that is what the precedence rules of *set_propagated_clock* command will select. However, to be clear, it is better to qualify the object explicitly, either as:

```
set_propagated_clock [get_clocks MCLK]
```

or as:

```
set_propagated_clock [get_ports MCLK]
```

With this explicit qualification, there is no need to depend on the precedence rules, and the SDC is clear.

A.3 Timing Constraints

This section describes the SDC commands that are related to timing specifications.

```
create_clock -period period_value [-name clock_name]
  [-waveform edge_list] [-add] [source_objects]
# Defines a clock.
# When clock_name is not specified, the clock name is the
# name of the first source object.
# The -period option specifies the clock period.
# The -add option is used to create a clock at a pin that
# already has an existing clock definition. Else if this
# option is not used, this clock definition overrides any
# other existing clock definition at that node.
# The -waveform option specifies the rising edge and
# falling edge (duty cycle) of the clock. The default
# is (0, period/2). If a clock definition is on a path
# after another clock, then it blocks the previous clock
# from that point onwards.
Examples:
  create_clock -period 20 -waveform {0 6} -name SYS_CLK \
    [get_ports SYS_CLK]      # Creates a clock of period
    # 20ns with rising edge at 0ns and the falling edge
    # at 6ns.
  create_clock -name CPU_CLK -period 2.33 \
    -add [get_ports CPU_CLK]    # Adds the clock definition
```

```
    # to the port without overriding any existing
    # clock definitions.

create_generated_clock [-name clock_name]
  -source master_pin [-edges edge_list]
  [-divide_by factor] [-multiply_by factor]
  [-duty_cycle percent] [-invert]
  [-edge_shift shift_list] [-add] [-master_clock clock]
  [-combinational]
  source_objects
# Defines an internally generated clock.
# If no -name is specified, the clock name is that of the
# first source object.
# The source of the generated clock, specified by -source,
# is a pin or port in the design.
# If more than one clock feeds the source node,
# the -master_clock option must be used to specify which of
# these clocks to use as the source of the generated clock.
# The -divide_by option can be used to specify the clock
# division factor; similarly for -multiply_by.
# The -duty_cycle can be used to specify the new duty cycle
# for clock multiplication.
# The -invert option can be specified if the phase of the
# clock has been inverted.
# Instead of using clock multiplication or division, clock
# generation can also be specified using -edges
# and -edge_shift options. The -edges option specifies a
# list of three numbers specifying the edges of the master
# clock edges to use for the first rising edge, the next
# falling edge, and the next rising edge. For example, a
# clock divider can be specified as -divide_by 2 or
# as -edges {1 3 5}.
# The -edge_shift option can be used in conjunction with
# the -edges option to specify an amount to shift for each
# of the three edges.
Examples:
  create_generated_clock -divide_by 2 -source \
```

```
    [get_ports sys_clk] -name gen_sys_clk [get_pins UFF/Q]
  create_generated_clock -add -invert -edges {1 2 8} \
    -source [get_ports mclk] -name gen_clk_div
  create_generated_clock -multiply_by 3 -source \
    [get_ports ref_clk] -master_clock clk10MHz \
    [get_pins UPLL/CLKOUT] -name gen_pll_clk

group_path [-name group_name] [-default]
  [-weight weight_value] [-from from_list]
  [-rise_from from_list] [-fall_from from_list]
  [-to to_list] [-rise_to to_list] [-fall_to to_list]
  [-through through_list] [-rise_through through_list]
  [-fall_through through_list]
# Gives a name to the specified group of paths.

set_clock_gating_check [-setup setup_value]
  [-hold hold_value] [-rise] [-fall] [-high] [-low]
  [object_list]
# Provides the ability to specify a clock gating check on
# any object.
# Clock gating checks are performed only on gates that get
# a clock signal.
# By default, the setup and hold values are 0.
Examples:
  set_clock_gating_check -setup 0.15 -hold 0.05 \
    [get_clocks ck20m]
  set_clock_gating_check -hold 0.3 \
    [get_cells U0/clk_divider/UAND1]

set_clock_groups [-name name] [-logically_exclusive]
  [-physically_exclusive] [-asynchronous] [-allow_paths]
  -group clock_list
# Specifies a group of clocks with the specific
# property and assigns a name to the group.
```

```
set_clock_latency [-rise] [-fall] [-min] [-max]
  [-source] [-late] [-early] [-clock clock_list] delay
  object_list
# Specifies the clock latency for a given clock.
# There are two types of latency: network and source.
# Source latency is the clock network delay between the
# clock definition pin and its source, while network
# latency is the clock network delay between the clock
# definition pin and the flip-flop clock pins.
Examples:
  set_clock_latency 1.86 [get_clocks clk250]
  set_clock_latency -source -late -rise 2.5 \
    [get_clocks MCLK]
  set_clock_latency -source -late -fall 2.3 \
    [get_clocks MCLK]

set_clock_sense [-positive] [-negative] [-pulse pulse]
  [-stop_propagation] [-clock clock_list] pin_list
# Set clock property on pin.

set_clock_transition [-rise] [-fall] [-min] [-max]
  transition clock_list
# Specifies the clock transition at the clock
# definition point.
Examples:
  set_clock_transition -min 0.5 [get_clocks SERDES_CLK]
  set_clock_transition -max 1.5 [get_clocks SERDES_CLK]

set_clock_uncertainty [-from from_clock]
  [-rise_from rise_from_clock]
  [-fall_from fall_from_clock] [-to to_clock]
  [-rise_to rise_to_clock] [-fall_to fall_to_clock]
  [-rise] [-fall] [-setup] [-hold]
  uncertainty [object_list]
# Specifies the clock uncertainty for clocks or for
# clock-to-clock transfers.
```

```
# The setup uncertainty is subtracted from the data
# required time for a path, and the hold uncertainty is
# added to the data required time for each path.
Examples:
 set_clock_uncertainty -setup -rise -fall 0.2 \
   [get_clocks CLK2]
 set_clock_uncertainty -from [get_clocks HSCLK] -to \
   [get_clocks SYSCLK] -hold 0.35

set_data_check [-from from_object] [-to to_object]
  [-rise_from from_object] [-fall_from from_object]
  [-rise_to to_object] [-fall_to to_object]
  [-setup] [-hold] [-clock clock_object] value
# Performs the specified check between the two pins.
Example:
 set_data_check -from [get_pins UBLK/EN] \
   -to [get_pins UBLK/D] -setup 0.2

set_disable_timing [-from from_pin_name]
  [-to to_pin_name] cell_pin_list
# Disables a timing arc/edge inside the specified cell.
Example:
 set_disable_timing -from A -to ZN [get_cells U1]

set_false_path [-setup] [-hold] [-rise] [-fall]
  [-from from_list] [-to to_list] [-through through_list]
  [-rise_from rise_from_list] [-rise_to rise_to_list]
  [-rise_through -rise_through_list]
  [-fall_from fall_from_list] [-fall_to fall_to_list]
  [-fall_through fall_through_list]
# Specifies a path exception that is not to be considered
# for STA.
Examples:
 set_false_path -from [get_clocks jtag_clk] \
   -to [get_clocks sys_clk]
 set_false_path -through U1/A -through U4/ZN
```

```
set_ideal_latency [-rise] [-fall] [-min] [-max]
  delay object_list
# Sets ideal latency to specific objects.

set_ideal_network [-no_propagate] object_list
# Identifies points in design that are sources of an
# ideal network.

set_ideal_transition [-rise] [-fall] [-min] [-max]
  transition_time object_list
# Specifies the transition time for the ideal networks
# and ideal nets.

set_input_delay [-clock clock_name] [-clock_fall]
  [-rise] [-fall] [-max] [-min] [-add_delay]
  [-network_latency_included] [-source_latency_included]
  delay_value port_pin_list
# Specifies the data arrival times at the specified input
# ports relative to the clock specified.
# The default is the rising edge of clock.
# The -add_delay option allows the capability to add more
# than one constraint to that particular pin or port.
# Multiple input delays with respect to different clocks
# can be specified using this -add_delay option.
# By default, the clock source latency of the launch clock
# is added to the input delay value, but when
# the -source_latency_included option is specified, the
# source network latency is not added because it is
# assumed to be factored into the input delay value.
# The -max delay is used for clock setup checks and recovery
# checks, while the -min delay is used for hold and removal
# checks. If only -min or -max or neither is specified,
# the same value is used for both.
Examples:
  set_input_delay -clock SYSCLK 1.1 [get_ports MDIO*]
  set_input_delay -clock virtual_mclk 2.5 [all_inputs]
```

```
set_max_delay [-rise] [-fall]
  [-from from_list] [-to to_list] [-through through_list]
  [-rise_from rise_from_list] [-rise_to rise_to_list]
  [-rise_through rise_through_list]
  [-fall_from fall_from_list] [-fall_to fall_to_list]
  [-fall_through fall_through_list]
  delay_value
# Sets the maximum delay on the specified path.
# This is used to specify delay between two arbitrary pins
# instead of from a flip-flop to another flip-flop.
Examples:
  set_max_delay -from [get_clocks FIFOCLK] \
    -to [get_clocks MAINCLK] 3.5
  set_max_delay -from [all_inputs] \
    -to [get_cells UCKDIV/UFF1/D] 2.66

set_max_time_borrow delay_value object_list
# Sets the max time that can be borrowed when analyzing
# a path to a latch.
Example:
  set_max_time_borrow 0.6 [get_pins CORE/CNT_LATCH/D]

set_min_delay [-rise] [-fall]
  [-from from_list] [-to to_list] [-through through_list]
  [-rise_from rise_from_list] [-rise_to rise_to_list]
  [-rise_through rise_through_list]
  [-fall_from fall_from_list] [-fall_to fall_to_list]
  [-fall_through fall_through_list]
  delay_value
# Sets the min delay for the specified path, which can
# be between any two arbitrary pins.
Examples:
  set_min_delay -from U1/S -to U2/A 0.6
  set_min_delay -from [get_clocks PCLK] \
    -to [get_pins UFF/*/S]
```

```
set_multicycle_path [-setup] [-hold] [-rise] [-fall]
  [-start] [-end] [-from from_list] [-to to_list]
  [-through through_list] [-rise_from rise_from_list]
  [-rise_to rise_to_list]
  [-rise_through rise_through_list]
  [-fall_from fall_from_list] [-fall_to fall_to_list]
  [-fall_through fall_through_list] path_multiplier
# Specifies a path as a multicycle path. Multiple -through
# can also be specified.
# Use the -setup option if the multicycle path is just for
# setup. Use the -hold option if the multicycle path is
# for hold.
# If neither -setup nor -hold is specified, the default
# is -setup and the default hold multiplier is 0.
# The -start refers to the path multiplier being applied to
# the launch clock, while -end refers to the path
# multiplier being applied to the capture clock.
# Default is -start.
# The value of the -hold multiplier represents the number
# of clock edges away from the default hold multicycle
# value which is 0.
Examples:
  set_multicycle_path -start -setup \
    -from [get_clocks PCLK] -to [get_clocks MCLK] 4
  set_multicycle_path -hold -from UFF1/Q -to UCNTFF/D 2
  set_multicycle_path -setup -to [get_pins UEDGEFF*] 4

set_output_delay [-clock clock_name] [-clock_fall]
  [-level_sensitive]
  [-rise] [-fall] [-max] [-min] [-add_delay]
  [-network_delay_included] [-source_latency_included]
  delay_value port_pin_list
# Specifies the required time of the output relative
# to the clock. The rising edge is default.
# By default, the clock source latency is added to the
# output delay value but when the -source_latency_included
# option is specified, the clock latency value is not added
```

```
# as it is assumed to be included in the output delay value.
# The -add_delay option can be used to specify multiple
# set_output_delay on a pin/port.

set_propagated_clock object_list
# Specifies that clock latency needs to be computed,
# that is, it is not ideal.
Example:
 set_propagated_clock [all_clocks]
```

A.4 Environment Commands

This section describes the commands that are used to setup the environment of the design under analysis.

```
set_case_analysis value port_or_pin_list
# Specifies the port or pin that is set to
# the constant value.
Examples:
 set_case_analysis 0 [get_pins UDFT/MODE_SEL]
 set_case_analysis 1 [get_ports SCAN_ENABLE]

set_drive [-rise] [-fall] [-min] [-max]
 resistance port_list
# Is used to specify the drive strength of the input port.
# It specifies the external drive resistance to the port.
# A value of 0 signifies highest drive strength.
Example:
 set_drive 0 {CLK RST}

set_driving_cell [-lib_cell lib_cell_name] [-rise]
 [-fall] [-library lib_name] [-pin pin_name]
 [-from_pin from_pin_name] [-multiply_by factor]
 [-dont_scale] [-no_design_rule]
```

```
  [-input_transition_rise rise_time]
  [-input_transition_fall fall_time] [-min] [-max]
  [-clock clock_name] [-clock_fall] port_list
# Is used to model the drive resistance of the cell
# driving the input port.
Example:
  set_driving_cell -lib_cell BUFX4 -pin ZN [all_inputs]

set_fanout_load value port_list
# Sets the specified fanout load on the output ports.
Example:
  set_fanout_load 5 [all_outputs]

set_input_transition [-rise] [-fall] [-min] [-max]
  [-clock clock_name] [-clock_fall]
  transition port_list
# Specifies the transition time on an input pin.
Examples:
  set_input_transition 0.2 \
    [get_ports SD_DIN*]
  set_input_transition -rise 0.5 \
    [get_ports GPIO*]

set_load [-min] [-max] [-subtract_pin_load] [-pin_load]
  [-wire_load] value objects
# Set the value of capacitive load on pin or net in design.
# The -subtract_pin_load option specifies to subtract the
# pin cap from the indicated load.
Examples:
  set_load 50 [all_outputs]
  set_load 0.1 [get_pins UFF0/Q]     # On an internal pin.
  set_load -subtract_pin_load 0.025 \
    [get_nets UCNT0/NET5]            # On a net.

set_logic_dc port_list
set_logic_one port_list
```

```
set_logic_zero port_list
# Sets the specified ports to be a don't care value,
# a logic one or a logic zero.
Examples:
 set_logic_dc SE
 set_logic_one TEST
 set_logic_zero [get_pins USB0/USYNC_FF1/Q]

set_max_area area_value
# Sets the max area limit for current design.
Example:
 set_max_area 20000.0

set_max_capacitance value object_list
# Specifies the max capacitance for ports or on a design.
# If for a design, it specifies the max capacitance for all
# pins in the design.
Examples:
 set_max_capacitance 0.2 [current_design]
 set_max_capacitance 1 [all_outputs]

set_max_fanout value object_list
# Specifies the max fanout value for ports or on a design.
# If for a design, it specifies the max fanout for all
# output pins in the design.
Examples:
 set_max_fanout 16 [get_pins UDFT0/JTAG/ZN]
 set_max_fanout 50 [current_design]

set_max_transition  [-clock_path]
  [-data_path] [-rise] [-fall] value object_list
# Specifies the max transition time on a port or
# on a design. If for a design, it specifies the max
# transition on all pins in a design.
Example:
 set_max_transition 0.2 UCLKDIV0/QN
```

```
set_min_capacitance value object_list
# Specifies a minimum capacitance value for a port
# or on pins in design.
Example:
  set_min_capacitance 0.05 UPHY0/UCNTR/B1

set_operating_conditions [-library lib_name]
  [-analysis_type type] [-max max_condition]
  [-min min_condition] [-max_library max_lib]
  [-min_library min_lib] [-object_list objects]
  [condition]
# Sets the specified operating condition for timing
# analysis. Analysis type can be single, bc_wc, or
# on_chip_variation. Operating conditions are defined in
# libraries using the operating_conditions command.
Examples:
  set_operating_conditions -analysis_type bc_wc
  set_operating_conditions WCCOM
  set_operating_conditions -analysis_type \
    on_chip_variation

set_port_fanout_number value port_list
# Sets maximum fanout of a port.
Example:
  set_port_fanout_number 10 [get_ports GPIO*]

set_resistance [-min] [-max] value list_of_nets
# Sets the resistance on the specified nets.
Examples:
  set_resistance 10 -min U0/U1/NETA
  set_resistance 50 -max U0/U1/NETA

set_timing_derate [-cell_delay] [-cell_check]
  [-net_delay] [-data] [-clock] [-early] [-late]
  derate_value [object_list]
# Specifies derating values.
```

```
set_wire_load_min_block_size size
# Specifies the minimum block size to be used when the
# wire load mode is set to enclosed.
Example:
 set_wire_load_min_block_size 5000

set_wire_load_mode mode_name
# Defines the mechanism of how a wire load model is to be
# used for nets in a hierarchical design.
# The mode_name can be top, enclosed, or segmented.
# The top mode causes the wire load model defined in the
# top-level of the hierarchy to be used at all lower levels.
# The enclosed mode causes the wire load model of the block
# that fully encloses that net to be used for that net.
# The segmented mode causes net segment in the block to use
# the block's wire load model.
Example:
 set_wire_load_mode enclosed

set_wire_load_model -name model_name [-library lib_name]
  [-min] [-max] [object_list]
# Defines the wire load model to be used for the current
# design or for the specified nets.
Example:
 set_wire_load_model -name "eSiliconLightWLM"

set_wire_load_selection_group [-library lib_name]
  [-min] [-max] group_name [object_list]
# Sets the wire load selection group for a design when
# determining wire load model based on cell area of the
# blocks. Selection groups are typically defined in
# technology libraries.
```

A.5 Multi-Voltage Commands

These commands apply when multi-voltage islands are present in a design.

```
create_voltage_area -name name
  [-coordinate coordinate_list] [-guard_band_x float]
  [-guard_band_y float] cell_list

set_level_shifter_strategy [-rule rule_type]

set_level_shifter_threshold [-voltage float]
  [-percent float]

set_max_dynamic_power power [unit]
# Specify max dynamic power.
Example:
  set_max_dynamic_power 0 mw

set_max_leakage_power power [unit]
# Specify max leakage power.
Example:
  set_max_leakage_power 12 mw
```

❑

APPENDIX

B

Standard Delay Format (SDF)

This appendix describes the standard delay annotation format and explains how backannotation is performed in simulation. The delay format describes cell delays and interconnect delays of a design netlist and is independent of the language the design may be described in, may it be VHDL or Verilog HDL, the two dominant standard hardware description languages.

While this chapter describes backannotation for simulation, backannotation for STA is a more simple and straightforward process in which the timing arcs in a DUA are annotated with the specified delays from the SDF.

B.1 What is it?

SDF stands for Standard Delay Format. It is an IEEE standard - IEEE Std 1497. It is an ASCII text file. It describes timing information and constraints. Its purpose is to serve as a textual timing exchange medium between various tools. It can also be used to describe timing data for tools that require it. Since it is an IEEE standard, timing information generated by one tool can be consumed by a number of other tools that support such a standard. The data is represented in a tool-independent and language-independent way and it includes specification of interconnect delays, device delays and timing checks.

Since SDF is an ASCII file, it is human-readable, though these files tend to be rather large for real designs. However, it is meant as an exchange medium between tools. Quite often when exchanging information, one could potentially run into a problem where a tool generates an SDF file but the other tool that reads SDF does not read the SDF properly. The tool reader could either generate an error or a warning reading the SDF or it might interpret the values in the SDF incorrectly. In that case, one may have to look into the file and see what went wrong. This chapter explains the basics of the SDF file and provides necessary and sufficient information to help understand and debug any annotation problems.

Figure B-1 shows a typical flow of how an SDF file is used. A timing calculator tool typically generates the timing information that is stored in an SDF file. This information is then backannotated into the design by the tool that reads the SDF. Note that the complete design information is not captured in an SDF file, but only the delay values are stored. For example, instance names and pin names of instances are captured in the SDF as they are necessary to specify instance-specific or pin-specific delays. Therefore, it is imperative that the same design be presented to both the SDF generation tool and the SDF reader tool.

One design can have multiple SDF files associated with it. One SDF file can be created for one design. In a hierarchical design, multiple SDFs may be created for each block in a hierarchy. During annotation, each SDF is ap-

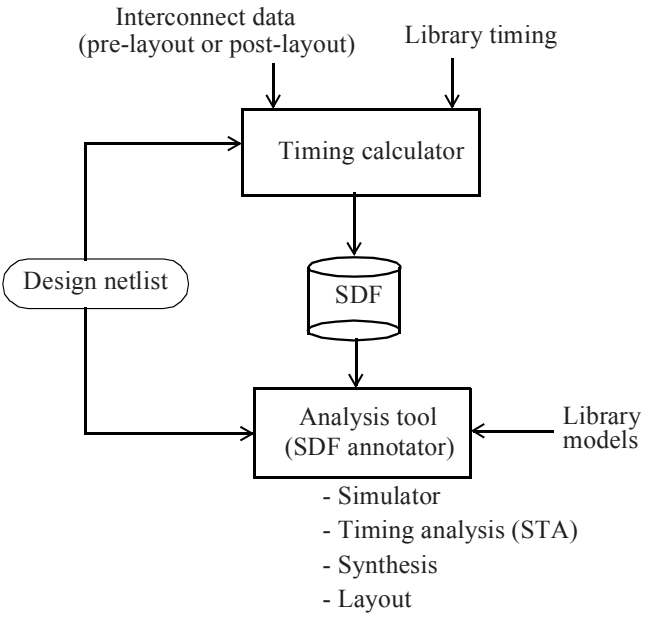


Figure B-1 *The SDF flow.*

plied to the appropriate hierarchical instance. Figure B-2 shows this figuratively.

An SDF file contains computed timing data for backannotation and for forward-annotation. More specifically, it contains:

i. Cell delays

ii. Pulse propagation

iii. Timing checks

iv. Interconnect delays

v. Timing environment

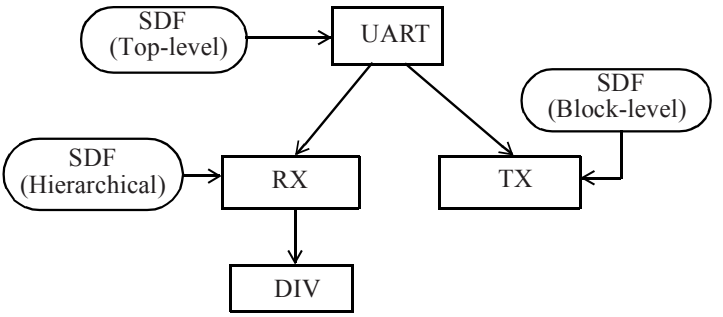


Figure B-2 *Multiple SDFs in a hierarchical design.*

Both pin-to-pin delay and distributed delay can be modeled for cell delays. Pin-to-pin delays are represented using the IOPATH construct. These constructs define input to output path delays for each cell. The COND construct can additionally be used to specify conditional pin-to-pin delays. State-dependent path delays can be specified using the COND construct as well. Distributed delay modeling is specified using the DEVICE construct.

The pulse propagation constructs, PATHPULSE and PATHPULSEPERCENT, can be used to specify the size of glitches that are allowed to propagate to the output of a cell using the pin-to-pin delay model.

The range of timing checks that can be specified in SDF includes:

i. Setup: SETUP, SETUPHOLD

ii. Hold: HOLD, SETUPHOLD

iii. Recovery: RECOVERY, RECREM

iv. Removal: REMOVAL, RECREM

v. Maximum skew: SKEW, BIDIRECTSKEW

vi. Minimum pulse width: WIDTH

vii. Minimum period: PERIOD

viii. No change: NOCHANGE

Conditions may be present on signals in timing checks. Negative values are allowed in timing checks, though tools that don't support negative values can choose to replace it with zero.

There are three styles of interconnect modeling that are supported in an SDF description. The INTERCONNECT construct is the most general and often used and can be used to specify point-to-point delay (from source to sink). Thus a single net can have multiple INTERCONNECT constructs. The PORT construct can be used to specify nets delays at only the load ports - it assumes that there is only one source for the net. The NETDELAY construct can be used to specify the delay of an entire net without regard to the sources or its sinks and therefore is the least specific way of specifying delays on a net.

The timing environment provides information under which the design operates. Such information includes the ARRIVAL, DEPARTURE, SLACK and WAVEFORM constructs. These constructs are mainly used for forward-annotation, such as for synthesis.

B.2 The Format

An SDF file contains a header section followed by one or more cells. Each cell represents a region or scope in a design. It can be a library primitive or a user-defined black box.

```
(DELAYFILE
 <header_section>
 (CELL
  <cell_section>
 )
 (CELL
  <cell_section>
 )
```

```
    ... <other cells>
)
```

The header section contains general information and does not affect the semantics of the SDF file, except for the hierarchy separator, timescale and the SDF version number. The hierarchy separator, DIVIDER, by default is the dot (`'.'`) character. It can be replaced with the `'/'` character by specifying:

```
(DIVIDER /)
```

If no timescale information is present in the header, the default is 1ns. Otherwise a timescale, TIMESCALE, can be explicitly specified using:

```
(TIMESCALE 10ps)
```

which says to multiply all delay values specified in the SDF file by 10ps.

The SDF version, SDFVERSION, is required and is used by the consumer of SDF to ensure that the file conforms to the specified SDF version. Other information that may be present in the header section, which is a general information category, includes date, program name, version and operating condition.

```
(DESIGN "BCM")
(DATE "Tuesday, May 24, 2004")
(PROGRAM "Star Galaxy Automation Inc., TimingTool")
(VERSION "V2004.1")
(VOLTAGE 1.65:1.65:1.65)
(PROCESS "1.000:1.000:1.000")
(TEMPERATURE 0.00:0.00:0.00)
```

Following the header section is a description of one or more cells. Each cell represents one or more instances (using wildcard) in the design. A cell may either be a library primitive or a hierarchical block.

```
(CELL
 (CELLTYPE <cell_type>)
 (INSTANCE <hierarchical_instance_name>)
 (DELAY
  <path_delay_section>
 )
 (TIMINGCHECK
  <timing_check_section>
 )
 (TIMINGENV
  <timing_environment_section>
 )
 (LABEL
  <label_section>
 )
)
. . . <other cells>
```

The order of cells is important as data is processed top to bottom. A later cell description may override timing information specified by an earlier cell description (usually it is not common to have timing information of the same cell instance defined twice). In addition, timing information can be annotated either as an absolute value or as an increment. If timing is incrementally applied, it adds the new value to the existing value; if the timing is absolute, it overwrites any previously specified timing information.

The cell instance can be a hierarchical instance name. The separator used for hierarchy separator must conform to the one specified in the header section. The cell instance name can optionally be the '*' character referring to a wildcard character, which means all cell instances of the specified type.

```
(CELL
 (CELLTYPE "NAND2")
 (INSTANCE *)
 // Refers to all instances of NAND2.
 . . .
```

There are four types of timing specifications that can be described in a cell:

i. DELAY: Used to describe delays.

ii. TIMINGCHECK: Used to describe timing checks.

iii. TIMINGENV: Used to describe the timing environment.

iv. LABEL: Declares timing model variables that can be used to describe delays.

Here are some examples.

```
// An absolute path delay specification:
(DELAY
  (ABSOLUTE
    (IOPATH A Y (0.147))
  )
)

// A setup and hold timing check specification:
(TIMINGCHECK
  (SETUPHOLD (posedge Q) (negedge CK) (0.448) (0.412))
)

// A timing constraint between two points:
(TIMINGENV
  (PATHCONSTRAINT UART/ENA UART/TX/CTRL (2.1) (1.5))
)

// A label that overrides the value of a Verilog HDL
// specparam:
(LABEL
  (ABSOLUTE
    (t$CLK$Q (0.480:0.512:0.578) (0.356:0.399:0.401))
    (tsetup$D$CLK (0.112))
  )
)
```

There are four types of DELAY timing specifications:

i. ABSOLUTE: Replaces existing delay values for cell instance during backannotation.

ii. INCREMENT: Adds the new delay data to any existing delay values of the cell instance.

iii. PATHPULSE: Specifies pulse propagation limit between an input and output of the design. This limit is used to decide whether to propagate a pulse appearing on the input to the output, or to be marked with an `'X'`, or to get filtered out.

iv. PATHPULSEPERCENT: This is exactly identical to PATHPULSE except that the values are described as percents.

Here are some examples.

```
// Absolute port delay:
(DELAY
  (ABSOLUTE
    (PORT UART.DIN (0.170))
    (PORT UART.RX.XMIT (0.645))
  )
)

// Adds IO path delay to existing delays of cell:
(DELAY
  (INCREMENT
    (IOPATH (negedge SE) Q (1.1:1.22:1.35))
  )
)

// Pathpulse delay:
(DELAY
  (PATHPULSE RN Q (3) (7))
)
// The ports RN and Q are input and output of the
```

```
// cell. The first value, 3, is the pulse rejection
// limit, called r-limit; it defines the narrowest pulse
// that can appear on output. Any pulse narrower than
// this is rejected, that is, it will not appear on
// output. The second value, 7, if present, is the
// error limit - also called e-limit. Any pulse smaller
// than e-limit causes the output to be an X.
// The e-limit must be greater than r-limit. See
// Figure B-3. When a pulse that is less than 3 (r-limit)
```

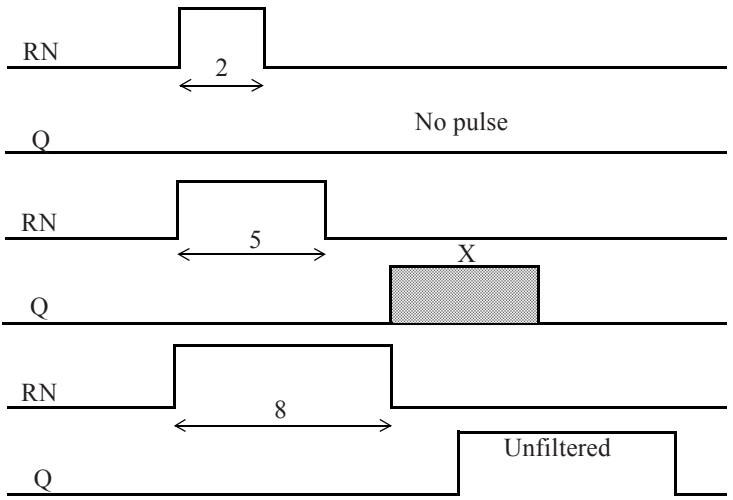


Figure B-3 *Error limit and rejection limit.*

```
// occurs, the pulse does not propagate to the output.
// When the pulse width is between the 3 (r-limit) and
// 7 (e-limit), the output is an X. When the pulse width
// is larger than 7 (e-limit), pulse propagates to output
// without any filtering.
```

```
// Pathpulsepercent delay type:
(DELAY
  (PATHPULSEPERCENT CIN SUM (30) (50))
)
// The r-limit is specified as 30% of the delay time from
// CIN to SUM and the e-limit is specified as 50% of
// this delay.
```

There are eight types of delay definitions that can be described with either ABSOLUTE or INCREMENT:

i. IOPATH: Input-output path delays.

ii. RETAIN: Retain definition. Specifies the time for which an output shall retain its previous value after a change on its related input port.

iii. COND: Conditional path delay. Can be used to specify state-dependent input-to-output path delays.

iv. CONDELSE: Default path delay. Specifies default value to use for conditional paths.

v. PORT: Port delays. Specifies interconnect delays that are modeled as delay at input ports.

vi. INTERCONNECT: Interconnect delays. Specifies the propagation delay across a net from a source to its sink.

vii. NETDELAY: Net delays. Specifies the propagation delay from all sources to all sinks of a net.

viii. DEVICE: Device delay. Primarily used to describe a distributed timing model. Specifies propagation delay of all paths through a cell to the output port.

Here are some examples.

```
// IO path delay between posedge of CK and Q:
(DELAY
```

```
  (ABSOLUTE
    (IOPATH (posedge CK) Q (2) (3))
  )
)
// 2 is the propagation rise delay and 3 is the
// propagation fall delay.

// Retain delay in an IO path:
(DELAY
  (ABSOLUTE
    (IOPATH A Y
      (RETAIN (0.05:0.05:0.05) (0.04:0.04:0.04))
      (0.101:0.101:0.101) (0.09:0.09:0.09))
  )
)
// Y shall retain its previous value for 50ps (40ps for
// a low value) after a change of value on input A.
// 50ps is the retain high value, 40ps is the retain
// low value, 101ps is the propagate rise delay and
// 90ps is the propagate fall delay. See Figure B-4.
```

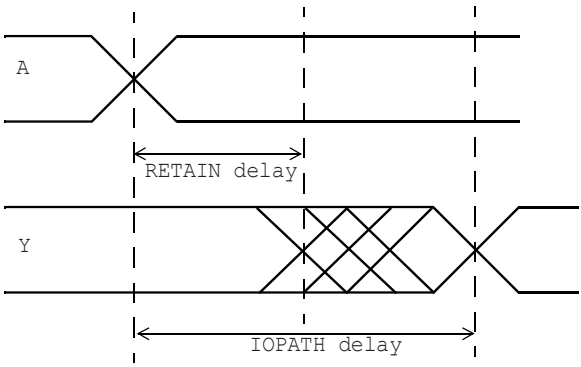


Figure B-4 *RETAIN delay.*

```
// Conditional path delay:
(DELAY
  (ABSOLUTE
    (COND SE == 1'b1 (IOPATH (posedge CK) Q (0.661)))
  )
)

// Default conditional path delay:
(DELAY
  (ABSOLUTE
    (CONDELSE (IOPATH ADDR[7] COUNT[0] (0.870) (0.766)))
  )
)

// Port delay on input FRM_CNT[0]:
(DELAY
  (ABSOLUTE
    (PORT UART/RX/FRM_CNT[0] (0.439))
  )
)

// Interconnect delay:
(DELAY
  (ABSOLUTE
    (INTERCONNECT O1/Y O2/B (0.209:0.209:0.209))
  )
)

// Net delay:
(DELAY
  (ABSOLUTE
    (NETDELAY A3/B (0.566))
  )
)
```

Delays

So far we have seen many different forms of delays. There are additional forms of delay specification. In general, delays can be specified as a set of one, two, three, six or twelve tokens that can be used to describe the following transition delays: `0->1`, `1->0`, `0->Z`, `Z->1`, `1->Z`, `Z->0`, `0->X`, `X->1`, `1->X`, `X->0`, `X->Z`, `Z->X`. The following table shows how fewer than twelve delay tokens are used to represent the twelve transitions.

Transition	2-values (v1 v2)	3-values (v1 v2 v3)	6-values (v1 v2 v3 v4 v5 v6)	12-values (v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12)
0->1	v1	v1	v1	v1
1->0	v2	v2	v2	v2
0->Z	v1	v3	v3	v3
Z->1	v1	v1	v4	v4
1->Z	v2	v3	v5	v5
Z->0	v2	v2	v6	v6
0->X	v1	min(v1,v3)	min(v1,v3)	v7
X->1	v1	v1	max(v1,v4)	v8
1->X	v2	min(v2,v3)	min(v2,v5)	v9
X->0	v2	v2	max(v2,v6)	v10
X->Z	max(v1,v2)	v3	max(v3,v5)	v11
Z->X	min(v1,v2)	min(v1,v2)	min(v6,v4)	v12

Table B-5 Mapping to twelve transition delays.

Here are some examples of these delays.

```
(DELAY
  (ABSOLUTE
```

```
    // 1-value delay:
    (IOPATH A Y (0.989))
    // 2-value delay:
    (IOPATH B Y (0.989) (0.891))
    // 6-value delay:
    (IOPATH CTRL Y (0.121) (0.119) (0.129)
                   (0.131) (0.112) (0.124))
    // 12-value delay:
    (COND RN == 1'b0
     (IOPATH C Y (0.330) (0.312) (0.330) (0.311) (0.328)
                 (0.321) (0.328) (0.320) (0.320)
                 (0.318) (0.318) (0.316)
     )
    )
    // In this 2-value delay, the first one is null
    // implying the annotator is not to change its value.
    (IOPATH RN Q () (0.129))
  )
)
```

Each delay token can, in turn, be written as one, two or three values as shown in the following examples.

```
(DELAY
 (ABSOLUTE
  // One value in a delay token:
  (IOPATH A Y (0.117))
  // The delay value, the pulse rejection limit
  // (r-limit) and X filter limit (e-limit) are same.

  // Two values in a delay (note no colon):
  (IOPATH (posedge CK) Q (0.12 0.15))
  // 0.12 is the delay value and 0.15 is the r-limit
  // and e-limit.
```

```
      // Three values in a delay:
      (IOPATH F1/Y AND1/A (0.339 0.1 0.15))
      // Path delay is 0.339, r-limit is 0.1 and
      // e-limit is 0.15.
    )
  )
```

Delay values in a single SDF file can be written using signed real numbers or as triplets of form:

```
(8.0 : 3.6 : 9.8)
```

to denote minimum, typical, maximum delays that represent the three process operating conditions of the design. The choice of which value is selected is made by the annotator typically based on a user-provided option. The values in the triplet form are optional, though it should have at least one. For example, the following are legal.

```
(::0.22)
(1.001: :0.998)
```

Values that are not specified are simply not annotated.

Timing Checks

Timing check limits are specified in the section that starts with the TIMINGCHECK keyword. In any of these checks, a COND construct can be used to specify conditional timing checks. In some cases, two additional conditional checks can be specified, SCOND and CCOND, that are associated with the *stamp event* and the *check event*.

Following are the set of checks:

i. SETUP: Setup timing check

ii. HOLD: Hold timing check

iii. SETUPHOLD: Setup and hold timing check

iv. RECOVERY: Recovery timing check

v. REMOVAL: Removal timing check

vi. RECREM: Recovery and removal timing check

vii. SKEW: Unidirectional skew timing check

viii. BIDIRECTSKEW: Bidirectional skew timing check

ix. WIDTH: Width timing check

x. PERIOD: Period timing check

xi. NOCHANGE: No-change timing check

Here are some examples.

```
(TIMINGCHECK
 // Setup check limit:
 (SETUP din (posedge clk) (2))

 // Hold check limit:
 (HOLD din (negedge clk) (0.445:0.445:0.445))

 // Conditional hold check limit :
 (HOLD (COND RST==1'b1 D) (posedge CLK) (1.15))
 // Hold check between D and positive edge of CLK, but
 // only when RST is 1.

 // Setup and hold check limit:
 (SETUPHOLD J CLK (1.2) (0.99))
 // 1.2 is the setup limit and 0.99 is the hold limit.

 // Conditional setup and hold limit:
 (SETUPHOLD D CLK (0.809) (0.591) (CCOND ~SE))
 // Condition applies with CLK for setup and
 // with D for hold.
```

```
    // Conditional setup and hold check limit:
    (SETUPHOLD (COND ~RST D) (posedge CLK) (1.452) (1.11))
    // Setup and hold check between D and positive edge
    // of CLK, but only when RST is low.

    // RECOVERY check limit:
    (RECOVERY SE (negedge CLK) (0.671))

    // Conditional removal check limit:
    (REMOVAL (COND ~LOAD CLEAR) CLK (2.001:2.1:2.145))
    // Removal check between CLEAR and CLK but only
    // when LOAD is low.

    // Recovery and removal check limit:
    (RECREM RST (negedge CLK) (1.1) (0.701))
    // 1.1 is the recovery limit and 0.701 is the
    // removal limit.

    // Skew conditional check limit:
    (SKEW (COND MMODE==1'b1 GNT) (posedge REQ) (3.2))

    // Bidirectional skew check limit:
    (BIDIRECTSKEW (posedge CLOCK1) (negedge TCK) (1.409))

    // Width check limit:
    (WIDTH (negedge RST) (12))

    // Period check limit:
    (PERIOD (posedge JTCLK) (13.33))

    // Nochange check limit:
    (NOCHANGE (posedge REQ) (negedge GNT) (2.5) (3.12))
  )
```

Labels

Labels are used to specify values for VHDL generics or Verilog HDL specify parameters.

```
(LABEL
  (ABSOLUTE
    (thold$d$clk (0.809))
    (tph$A$Y (0.553))
  )
)
```

Timing Environment

There are a number of constructs available that can be used to describe the timing environment of a design. However, these constructs are used for forward-annotation rather than backward-annotation, such as in logic synthesis tools. These are not described in this text.

B.2.1 Examples

We provide complete SDFs for two designs.

Full-adder

Here is the Verilog HDL netlist for a full-adder circuit.

```
module FA_STR (A, B, CIN, SUM, COUT);
  input A, B, CIN;
  output SUM, COUT;
  wire S1, S2, S3, S4, S5;

  XOR2X1 X1 (.Y(S1), .A(A), .B(B));
  XOR2X1 X2 (.Y(SUM), .A(S1), .B(CIN));
```

```
  AND2X1 A1 (.Y(S2), .A(A), .B(B));
  AND2X1 A2 (.Y(S3), .A(B), .B(CIN));
  AND2X1 A3 (.Y(S4), .A(A), .B(CIN));

  OR2X1 O1 (.Y(S5), .A(S2), .B(S3));
  OR2X1 O2 (.Y(COUT), .A(S4), .B(S5));
endmodule
```

Here is the complete corresponding SDF file produced by a timing analysis tool.

```
(DELAYFILE
  (SDFVERSION "OVI 2.1")
  (DESIGN "FA_STR")
  (DATE "Mon May 24 13:56:43 2004")
  (VENDOR "slow")
  (PROGRAM "CompanyName ToolName")
  (VERSION "V2.3")
  (DIVIDER /)
  // OPERATING CONDITION "slow"
  (VOLTAGE 1.35:1.35:1.35)
  (PROCESS "1.000:1.000:1.000")
  (TEMPERATURE 125.00:125.00:125.00)
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "FA_STR")
    (INSTANCE)
    (DELAY
      (ABSOLUTE
        (INTERCONNECT A A3/A (0.000:0.000:0.000))
        (INTERCONNECT A A1/A (0.000:0.000:0.000))
        (INTERCONNECT A X1/A (0.000:0.000:0.000))
        (INTERCONNECT B A2/A (0.000:0.000:0.000))
        (INTERCONNECT B A1/B (0.000:0.000:0.000))
        (INTERCONNECT B X1/B (0.000:0.000:0.000))
        (INTERCONNECT CIN A3/B (0.000:0.000:0.000))
```

```
      (INTERCONNECT CIN A2/B (0.000:0.000:0.000))
      (INTERCONNECT CIN X2/B (0.000:0.000:0.000))
      (INTERCONNECT X2/Y SUM (0.000:0.000:0.000))
      (INTERCONNECT O2/Y COUT (0.000:0.000:0.000))
      (INTERCONNECT X1/Y X2/A (0.000:0.000:0.000))
      (INTERCONNECT A1/Y O1/A (0.000:0.000:0.000))
      (INTERCONNECT A2/Y O1/B (0.000:0.000:0.000))
      (INTERCONNECT A3/Y O2/A (0.000:0.000:0.000))
      (INTERCONNECT O1/Y O2/B (0.000:0.000:0.000))
    )
  )
)
(CELL
  (CELLTYPE "XOR2X1")
  (INSTANCE X1)
  (DELAY
    (ABSOLUTE
      (IOPATH A Y (0.197:0.197:0.197)
                  (0.190:0.190:0.190))
      (IOPATH B Y (0.209:0.209:0.209)
                  (0.227:0.227:0.227))
      (COND B==1'b1 (IOPATH A Y (0.197:0.197:0.197)
                                (0.190:0.190:0.190)))
      (COND A==1'b1 (IOPATH B Y (0.209:0.209:0.209)
                                (0.227:0.227:0.227)))
      (COND B==1'b0 (IOPATH A Y (0.134:0.134:0.134)
                                (0.137:0.137:0.137)))
      (COND A==1'b0 (IOPATH B Y (0.150:0.150:0.150)
                                (0.163:0.163:0.163)))
    )
  )
)
(CELL
  (CELLTYPE "XOR2X1")
  (INSTANCE X2)
  (DELAY
    (ABSOLUTE
```

```
      (IOPATH (posedge A) Y (0.204:0.204:0.204)
                            (0.196:0.196:0.196))
      (IOPATH (negedge A) Y (0.198:0.198:0.198)
                            (0.190:0.190:0.190))
      (IOPATH B Y (0.181:0.181:0.181)
                  (0.201:0.201:0.201))
      (COND B==1'b1 (IOPATH A Y (0.198:0.198:0.198)
                                (0.196:0.196:0.196)))
      (COND A==1'b1 (IOPATH B Y (0.181:0.181:0.181)
                                (0.201:0.201:0.201)))
      (COND B==1'b0 (IOPATH A Y (0.135:0.135:0.135)
                                (0.140:0.140:0.140)))
      (COND A==1'b0 (IOPATH B Y (0.122:0.122:0.122)
                                (0.139:0.139:0.139)))
    )
   )
  )

  (CELL
   (CELLTYPE "AND2X1")
   (INSTANCE A1)
   (DELAY
    (ABSOLUTE
     (IOPATH A Y (0.147:0.147:0.147)
                 (0.157:0.157:0.157))
     (IOPATH B Y (0.159:0.159:0.159)
                 (0.173:0.173:0.173))
    )
   )
  )
  (CELL
   (CELLTYPE "AND2X1")
   (INSTANCE A2)
   (DELAY
    (ABSOLUTE
     (IOPATH A Y (0.148:0.148:0.148)
                 (0.157:0.157:0.157))
```

```
      (IOPATH B Y (0.160:0.160:0.160)
                  (0.174:0.174:0.174))
    )
   )
  )
  (CELL
   (CELLTYPE "AND2X1")
   (INSTANCE A3)
   (DELAY
    (ABSOLUTE
     (IOPATH A Y (0.147:0.147:0.147)
                 (0.157:0.157:0.157))
     (IOPATH B Y (0.159:0.159:0.159)
                 (0.173:0.173:0.173))
    )
   )
  )
  (CELL
   (CELLTYPE "OR2X1")
   (INSTANCE O1)
   (DELAY
    (ABSOLUTE
     (IOPATH A Y (0.138:0.138:0.138)
                 (0.203:0.203:0.203))
     (IOPATH B Y (0.151:0.151:0.151)
                 (0.223:0.223:0.223))
    )
   )
  )
  (CELL
   (CELLTYPE "OR2X1")
   (INSTANCE O2)
   (DELAY
    (ABSOLUTE
     (IOPATH A Y (0.126:0.126:0.126)
                 (0.191:0.191:0.191))
     (IOPATH B Y (0.136:0.136:0.136)
```

```
                    (0.212:0.212:0.212))
      )
    )
  )
)
```

All delays in the INTERCONNECTs are 0 as this is pre-layout data and ideal interconnects are modeled.

Decade Counter

Here is the Verilog HDL model for a decade counter.

```
module DECADE_CTR (COUNT, Z);
  input COUNT;
  output [0:3] Z;
  wire S1, S2;

  AND2X1 a1 (.Y(S1), .A(Z[2]), .B(Z[1]));

  JKFFX1
    JK1 (.J(1'b1), .K(1'b1), .CK(COUNT),
         .Q(Z[0]), .QN()),
    JK2 (.J(S2), .K(1'b1), .CK(Z[0]), .Q(Z[1]), .QN()),
    JK3 (.J(1'b1), .K(1'b1), .CK(Z[1]),
         .Q(Z[2]), .QN()),
    JK4 (.J(S1), .K(1'b1), .CK(Z[0]),
         .Q(Z[3]), .QN(S2));
endmodule
```

The complete corresponding SDF follows.

```
(DELAYFILE
  (SDFVERSION "OVI 2.1")
  (DESIGN "DECADE_CTR")
  (DATE "Mon May 24 14:30:17 2004")
```

```
(VENDOR "Star Galaxy Automation, Inc.")
(PROGRAM "MyCompanyName ToolTime")
(VERSION "V2.3")
(DIVIDER /)
// OPERATING CONDITION "slow"
(VOLTAGE 1.35:1.35:1.35)
(PROCESS "1.000:1.000:1.000")
(TEMPERATURE 125.00:125.00:125.00)
(TIMESCALE 1ns)
(CELL
  (CELLTYPE "DECADE_CTR")
  (INSTANCE)
  (DELAY
    (ABSOLUTE
      (INTERCONNECT COUNT JK1/CK (0.191:0.191:0.191))
      (INTERCONNECT JK1/Q Z\[0\] (0.252:0.252:0.252))
      (INTERCONNECT JK2/Q Z\[1\] (0.186:0.186:0.186))
      (INTERCONNECT JK3/Q Z\[2\] (0.18:0.18:0.18))
      (INTERCONNECT JK4/Q Z\[3\] (0.195:0.195:0.195))
      (INTERCONNECT JK3/Q a1/A (0.175:0.175:0.175))
      (INTERCONNECT JK2/Q a1/B (0.207:0.207:0.207))
      (INTERCONNECT JK4/QN JK2/J (0.22:0.22:0.22))
      (INTERCONNECT JK1/Q JK2/CK (0.181:0.181:0.181))
      (INTERCONNECT JK2/Q JK3/CK (0.193:0.193:0.193))
      (INTERCONNECT a1/Y JK4/J (0.224:0.224:0.224))
      (INTERCONNECT JK1/Q JK4/CK (0.218:0.218:0.218))
    )
  )
)
(CELL
  (CELLTYPE "AND2X1")
  (INSTANCE a1)
  (DELAY
    (ABSOLUTE
      (IOPATH A Y (0.179:0.179:0.179)
                  (0.186:0.186:0.186))
      (IOPATH B Y (0.190:0.190:0.190)
```

```
                    (0.210:0.210:0.210))
      )
    )
  )
  (CELL
    (CELLTYPE "JKFFX1")
    (INSTANCE JK1)
    (DELAY
      (ABSOLUTE
        (IOPATH (posedge CK) Q (0.369:0.369:0.369)
                               (0.470:0.470:0.470))
        (IOPATH (posedge CK) QN (0.280:0.280:0.280)
                                (0.178:0.178:0.178))
      )
    )
    (TIMINGCHECK
      (SETUP (posedge J) (posedge CK)
        (0.362:0.362:0.362))
      (SETUP (negedge J) (posedge CK)
        (0.220:0.220:0.220))
      (HOLD (posedge J) (posedge CK)
        (-0.272:-0.272:-0.272))
      (HOLD (negedge J) (posedge CK)
        (-0.200:-0.200:-0.200))
      (SETUP (posedge K) (posedge CK)
        (0.170:0.170:0.170))
      (SETUP (negedge K) (posedge CK)
        (0.478:0.478:0.478))
      (HOLD (posedge K) (posedge CK)
        (-0.158:-0.158:-0.158))
      (HOLD (negedge K) (posedge CK)
        (-0.417:-0.417:-0.417))
      (WIDTH (negedge CK)
        (0.337:0.337:0.337))
      (WIDTH (posedge CK) (0.148:0.148:0.148))
    )
  )
```

```
(CELL
  (CELLTYPE "JKFFX1")
  (INSTANCE JK2)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge CK) Q (0.409:0.409:0.409)
                          (0.512:0.512:0.512))
      (IOPATH (posedge CK) QN (0.326:0.326:0.326)
                           (0.222:0.222:0.222))
    )
  )
  (TIMINGCHECK
    (SETUP (posedge J) (posedge CK)
      (0.348:0.348:0.348))
    (SETUP (negedge J) (posedge CK)
      (0.227:0.227:0.227))
    (HOLD (posedge J) (posedge CK)
      (-0.257:-0.257:-0.257))
    (HOLD (negedge J) (posedge CK)
      (-0.209:-0.209:-0.209))
    (SETUP (posedge K) (posedge CK)
      (0.163:0.163:0.163))
    (SETUP (negedge K) (posedge CK)
      (0.448:0.448:0.448))
    (HOLD (posedge K) (posedge CK)
      (-0.151:-0.151:-0.151))
    (HOLD (negedge K) (posedge CK)
      (-0.392:-0.392:-0.392))
    (WIDTH (negedge CK) (0.337:0.337:0.337))
    (WIDTH (posedge CK) (0.148:0.148:0.148))
  )
)
(CELL
  (CELLTYPE "JKFFX1")
  (INSTANCE JK3)
  (DELAY
    (ABSOLUTE
```

```
      (IOPATH (posedge CK) Q (0.378:0.378:0.378)
                             (0.485:0.485:0.485))
      (IOPATH (posedge CK) QN (0.324:0.324:0.324)
                              (0.221:0.221:0.221))
    )
   )
   (TIMINGCHECK
    (SETUP (posedge J) (posedge CK)
     (0.339:0.339:0.339))
    (SETUP (negedge J) (posedge CK)
     (0.211:0.211:0.211))
    (HOLD (posedge J) (posedge CK)
     (-0.249:-0.249:-0.249))
    (HOLD (negedge J) (posedge CK)
     (-0.192:-0.192:-0.192))
    (SETUP (posedge K) (posedge CK)
     (0.163:0.163:0.163))
    (SETUP (negedge K) (posedge CK)
     (0.449:0.449:0.449))
    (HOLD (posedge K) (posedge CK)
     (-0.152:-0.152:-0.152))
    (HOLD (negedge K) (posedge CK)
     (-0.393:-0.393:-0.393))
    (WIDTH (negedge CK) (0.337:0.337:0.337))
    (WIDTH (posedge CK) (0.148:0.148:0.148))
   )
  )
  (CELL
   (CELLTYPE "JKFFX1")
   (INSTANCE JK4)
   (DELAY
    (ABSOLUTE
     (IOPATH (posedge CK) Q (0.354:0.354:0.354)
                            (0.464:0.464:0.464))
     (IOPATH (posedge CK) QN (0.364:0.364:0.364)
                             (0.256:0.256:0.256))
    )
```

```
    )
    (TIMINGCHECK
      (SETUP (posedge J) (posedge CK)
        (0.347:0.347:0.347))
      (SETUP (negedge J) (posedge CK)
        (0.226:0.226:0.226))
      (HOLD (posedge J) (posedge CK)
        (-0.256:-0.256:-0.256))
      (HOLD (negedge J) (posedge CK)
        (-0.208:-0.208:-0.208))
      (SETUP (posedge K) (posedge CK)
        (0.163:0.163:0.163))
      (SETUP (negedge K) (posedge CK)
        (0.448:0.448:0.448))
      (HOLD (posedge K) (posedge CK)
        (-0.151:-0.151:-0.151))
      (HOLD (negedge K) (posedge CK)
        (-0.392:-0.392:-0.392))
      (WIDTH (negedge CK) (0.337:0.337:0.337))
      (WIDTH (posedge CK) (0.148:0.148:0.148))
    )
  )
)
```

B.3 The Annotation Process

In this section, we describe how the annotation of the SDF occurs to an HDL description. SDF annotation can be performed by a number of tools, such as logic synthesis, simulation and static timing analysis; the SDF annotator is the component of these tools that reads the SDF, interprets and annotates the timing values to the design. It is assumed that the SDF file is created using information that is consistent with the HDL model and that the same HDL model is used during backannotation. Additionally, it is the responsibility of the SDF annotator to ensure that the timing values in the SDF are interpreted correctly.

The SDF annotator annotates the backannotation timing generics and parameters. It reports any errors if there is any noncompliance to the standard, either in syntax or in the mapping process. If certain SDF constructs are not supported by an SDF annotator, no errors are produced - the annotator simply ignores these.

If the SDF annotator fails to modify a backannotation timing generic, then the value of the generic is not modified during the backannotation process, that is, it is left unchanged.

In a simulation tool, backannotation typically occurs just following the elaboration phase and directly preceding negative constraint delay calculation.

B.3.1 Verilog HDL

In Verilog HDL, the primary mechanism for annotation is the specify block. A specify block can specify path delays and timing checks. Actual delay values and timing check limit values are specified via the SDF file. The mapping is an industry standard and is defined in IEEE Std 1364.

Specify path delays, specparam values, timing check constraint limits and interconnect delays are among the information obtained from an SDF file and annotated in a specify block of a Verilog HDL module. Other constructs in an SDF file are ignored when annotating to a Verilog HDL model. The LABEL section in SDF defines specparam values. Backannotation is done by matching SDF constructs to corresponding Verilog HDL declarations and then replacing the existing timing values with those in the SDF file.

Here is a table that shows how SDF delay values are mapped to Verilog HDL delay values.

Verilog transition	1-value (v1)	2-values (v1 v2)	3-values (v1 v2 v3)	6-values (v1 v2 v3 v4 v5 v6)	12-values (v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12)
0->1	v1	v1	v1	v1	v1
1->0	v1	v2	v2	v2	v2
0->z	v1	v1	v3	v3	v3
z->1	v1	v1	v1	v4	v4
1->z	v1	v2	v3	v5	v5
z->0	v1	v2	v2	v6	v6
0->x	v1	v1	min(v1 v3)	min(v1 v3)	v7
x->1	v1	v1	v1	max(v1 v4)	v8
1->x	v1	v2	min (v2 v3)	min(v2 v5)	v9
x->0	v1	v2	v2	max(v2 v6)	v10
x->z	v1	max(v1 v2)	v3	max(v3 v5)	v11
z->x	v1	min(v1 v2)	min(v1 v2)	min(v4 v6)	v12

Table B-6 Mapping SDF delays to Verilog HDL delays.

The following table describes the mapping of SDF constructs to Verilog HDL constructs.

Kinds	SDF construct	Verilog HDL
Propagation delay	IOPATH	Specify paths
Input setup time	SETUP	**$setup**, **$setuphold**
Input hold time	HOLD	**$hold**, **$setuphold**
Input setup and hold	SETUPHOLD	**$setup**, **$hold**, **$setuphold**
Input recovery time	RECOVERY	**$recovery**
Input removal time	REMOVAL	**$removal**
Recovery and removal	RECREM	**$recovery**, **$removal**, **$recrem**
Period	PERIOD	**$period**
Pulse width	WIDTH	**$width**
Input skew time	SKEW	**$skew**
No-change time	NOCHANGE	**$nochange**
Port delay	PORT	Interconnect delay
Net delay	NETDELAY	Interconnect delay
Interconnect delay	INTERCONNECT	Interconnect delay
Device delay	DEVICE_DELAY	Specify paths
Path pulse limit	PATHPULSE	Specify path pulse limit
Path pulse limit	PATHPULSEPERCENT	Specify path pulse limit

Table B-7 Mapping of SDF to Verilog HDL.

See later section for examples.

B.3.2 VHDL

Annotation of SDF to VHDL is an industry standard. It is defined in the IEEE standard for VITAL ASIC Modeling Specification, IEEE Std 1076.4; one of the components of this standard describes the annotation of SDF delays into ASIC libraries. Here, we present only the relevant part of the VITAL standard as it relates to SDF mapping.

SDF is used to modify backannotation timing generics in a VITAL-compliant model directly. Timing data can be specified only for a VITAL-compliant model using SDF. There are two ways to pass timing data into a VHDL model: via configurations, or directly into simulation. The SDF annotation process consists of mapping SDF constructs and corresponding generics in a VITAL-compliant model during simulation.

In a VITAL-compliant model, there are rules on how generics are to be named and declared that ensures that a mapping can be established between the timing generics of a model and the corresponding SDF timing information.

A timing generic is made up of a generic name and its type. The name specifies the kind of timing information and the type of the generic specifies the kind of timing value. If the name of generic does not follow the VITAL standard, then it is not a timing generic and does not get annotated.

Here is the table showing how SDF delay values are mapped to VHDL delays.

VHDL transition	1-value (v1)	2-values (v1 v2)	3-values (v1 v2 v3)	6-values (v1 v2 v3 v4 v5 v6)	12-values (v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12)
0->1	v1	v1	v1	v1	v1
1->0	v1	v2	v2	v2	v2
0->z	v1	v1	v3	v3	v3
z->1	v1	v1	v1	v4	v4
1->z	v1	v2	v3	v5	v5
z->0	v1	v2	v2	v6	v6
0->x	-	-	-	-	v7
x->1	-	-	-	-	v8
1->x	-	-	-	-	v9
x->0	-	-	-	-	v10
x->z	-	-	-	-	v11
z->x	-	-	-	-	v12

Table B-8 Mapping SDF delays to VHDL delays.

In VHDL, timing information is backannotated via generics. Generic names follow a certain convention so as to be consistent or derived from SDF constructs. With each of the timing generic names, an optional suffix of a conditioned edge can be specified. The edge specifies an edge associated with the timing information.

Table B-9 shows the different kinds of timing generic names.

Kinds	SDF construct	VHDL generic
Propagation delay	IOPATH	**tpd**_*InputPort_OutputPort* [_*condition*]
Input setup time	SETUP	**tsetup**_*TestPort_RefPort* [_*condition*]
Input hold time	HOLD	**thold**_*TestPort_RefPort* [_*condition*]
Input recovery time	RECOVERY	**trecovery**_*TestPort_RefPort* [_*condition*]
Input removal time	REMOVAL	**tremoval**_*TestPort_RefPort* [_*condition*]
Period	PERIOD	**tperiod**_*InputPort* [_*condition*]
Pulse width	WIDTH	**tpw**_*InputPort* [_*condition*]
Input skew time	SKEW	**tskew**_*FirstPort_SecondPort* [_*condition*]
No-change time	NOCHANGE	**tncsetup**_*TestPort_RefPort* [_*condition*] **tnchold**_*TestPort_RefPort* [_*condition*]
Interconnect path delay	PORT	**tipd**_*InputPort*
Device delay	DEVICE	**tdevice**_*InstanceName* [_*OutputPort*]
Internal signal delay		**tisd**_*InputPort_ClockPort*
Biased propagation delay		**tbpd**_*InputPort_OutputPort_ClockPort* [_*condition*]
Internal clock delay		**ticd**_*ClockPort*

Table B-9 Mapping of SDF to VHDL generics.

B.4 Mapping Examples

Here are examples of mapping SDF constructs to VHDL generics and Verilog HDL declarations.

Propagation Delay

- Propagation delay from input port A to output port Y with a rise time of 0.406 and a fall of 0.339.

```
// SDF:
(IOPATH A Y (0.406) (0.339))

-- VHDL generic:
tpd_A_Y : VitalDelayType01;

// Verilog HDL specify path:
(A *> Y) = (tplh$A$Y, tphl$A$Y);
```

- Propagation delay from input port OE to output port Y with a rise time of 0.441 and a fall of 0.409. The minimum, nominal and maximum delays are identical.

```
// SDF:
(IOPATH OE Y (0.441:0.441:0.441) (0.409:0.409:0.409))

-- VHDL generic:
tpd_OE_Y : VitalDelayType01Z;

// Verilog HDL specify path:
(OE *> Y) = (tplh$OE$Y, tphl$OE$Y);
```

- Conditional propagation delay from input port S0 to output port Y.

```
// SDF:
(COND A==0 && B==1 && S1==0
  (IOPATH S0 Y (0.062:0.062:0.062) (0.048:0.048:0.048)
  )
)
```

```
-- VHDL generic:
tpd_S0_Y_A_EQ_0_AN_B_EQ_1_AN_S1_EQ_0 :
  VitalDelayType01;

// Verilog HDL specify path:
if ((A == 1'b0) && (B == 1'b1) && (S1 == 1'b0))
  (S0 *> Y) = (tplh$S0$Y, tphl$S0$Y);
```

- Conditional propagation delay from input port A to output port Y.

```
// SDF:
(COND B == 0
  (IOPATH A Y (0.130) (0.098)
  )
)

-- VHDL generic:
tpd_A_Y_B_EQ_0 : VitalDelayType01;

// Verilog HDL specify path:
if (B == 1'b0)
  (A *> Y) = 0;
```

- Propagation delay from input port CK to output port Q.

```
// SDF:
(IOPATH CK Q (0.100:0.100:0.100) (0.118:0.118:0.118))

-- VHDL generic:
tpd_CK_Q : VitalDelayType01;

// Verilog HDL specify path:
(CK *> Q) = (tplh$CK$Q, tphl$CK$Q);
```

- Conditional propagation delay from input port A to output port Y.

```
// SDF:
(COND B == 1
  (IOPATH A Y (0.062:0.062:0.062) (0.048:0.048:0.048)
  )
)

-- VHDL generic:
tpd_A_Y_B_EQ_1 : VitalDelayType01;

// Verilog HDL specify path:
if (B == 1'b1)
  (A *> Y) = (tplh$A$Y, tphl$A$Y);
```

- Propagation delay from input port CK to output port ECK.

```
// SDF:
(IOPATH CK ECK (0.097:0.097:0.097))

-- VHDL generic:
tpd_CK_ECK : VitalDelayType01;

// Verilog HDL specify path:
(CK *> ECK) = (tplh$CK$ECK, tphl$CK$ECK);
```

- Conditional propagation delay from input port CI to output port S.

```
// SDF:
(COND (A == 0 && B == 0) || (A == 1 && B == 1)
  (IOPATH CI S (0.511) (0.389)
```

```
  )
)

-- VHDL generic:
tpd_CI_S_OP_A_EQ_0_AN_B_EQ_0_CP_OR_OP_A_EQ_1_AN_B_EQ_1_CP:
  VitalDelayType01;

// Verilog HDL specify path:
if ((A == 1'b0 && B == 1'b0) || (A == 1'b1 && B == 1'b1))
  (CI *> S)  = (tplh$CI$S, tphl$CI$S);
```

- Conditional propagation delay from input port CS to output port S.

```
// SDF:
(COND (A == 1 ^ B == 1 ^ CI1 == 1) &&
     !(A == 1 ^ B == 1 ^ CI0 == 1)
  (IOPATH CS S (0.110) (0.120) (0.120)
               (0.110) (0.119) (0.120)
  )
)

-- VHDL generic:
tpd_CS_S_OP_A_EQ_1_XOB_B_EQ_1_XOB_CI1_EQ_1_CP_AN_NT_
OP_A_EQ_1_XOB_B_EQ_1_XOB_CI0_EQ_1_CP:
  VitalDelayType01;

// Verilog HDL specify path:
if ((A == 1'b1 ^ B == 1'b1 ^ CI1N == 1'b0) &&
    !(A == 1'b1 ^ B == 1'b1 ^ CI0N == 1'b0))
  (CS  *> S)  = (tplh$CS$S, tphl$CS$S);
```

- Conditional propagation delay from input port A to output port ICO.

```
// SDF:
(COND B == 1 (IOPATH A ICO (0.690)))

-- VHDL generic:
tpd_A_ICO_B_EQ_1 : VitalDelayType01;

// Verilog HDL specify path:
if (B == 1'b1)
  (A  *> ICO) = (tplh$A$ICO,  tphl$A$ICO);
```

- Conditional propagation delay from input port A to output port CO.

```
// SDF:
(COND (B == 1 ^ C == 1) && (D == 1 ^ ICI == 1)
  (IOPATH A CO (0.263)
  )
)

-- VHDL generic:
tpd_A_CO_OP_B_EQ_1_XOB_C_EQ_1_CP_AN_OP_D_EQ_1_XOB_ICI_E
Q_1_CP: VitalDelayType01;

// Verilog HDL specify path:
if ((B == 1'b1 ^ C == 1'b1) && (D == 1'b1 ^ ICI == 1'b1))
  (A *> CO) = (tplh$A$CO,  tphl$A$CO);
```

- Delay from positive edge of CK to Q.

```
// SDF:
(IOPATH (posedge CK) Q (0.410:0.410:0.410)
  (0.290:0.290:0.290))
```

```
-- VHDL generic:
tpd_CK_Q_posedge_noedge : VitalDelayType01;

// Verilog HDL specify path:
(posedge CK *> Q) = (tplh$CK$Q, tphl$CK$Q);
```

Input Setup Time

- Setup time between posedge of D and posedge of CK.

```
// SDF:
(SETUP (posedge D) (posedge CK) (0.157:0.157:0.157))

-- VHDL generic:
tsetup_D_CK_posedge_posedge: VitalDelayType;

// Verilog HDL timing check task:
$setup(posedge CK, posedge D, tsetup$D$CK, notifier);
```

- Setup between negedge of D and posedge of CK.

```
// SDF:
(SETUP (negedge D) (posedge CK) (0.240))

-- VHDL generic:
tsetup_D_CK_negedge_posedge: VitalDelayType;

// Verilog HDL timing check task:
$setup(posedge CK, negedge D, tsetup$D$CK, notifier);
```

- Setup time between posedge of input E with posedge of reference CK.

```
// SDF:
(SETUP (posedge E) (posedge CK) (-0.043:-0.043:-0.043))

-- VHDL generic:
tsetup_E_CK_posedge_posedge : VitalDelayType;

// Verilog HDL timing check task:
$setup(posedge CK, posedge E, tsetup$E$CK, notifier);
```

- Setup time between negedge of input E and posedge of reference CK.

```
// SDF:
(SETUP (negedge E) (posedge CK) (0.101) (0.098))

-- VHDL generic:
tsetup_E_CK_negedge_posedge : VitalDelayType;

// Verilog HDL timing check task:
$setup(posedge CK, negedge E, tsetup$E$CK, notifier);
```

- Conditional setup time between SE and CK.

```
// SDF:
(SETUP (cond E != 1 SE) (posedge CK) (0.155) (0.135))

-- VHDL generic:
tsetup_SE_CK_E_NE_1_noedge_posedge : VitalDelayType;

// Verilog HDL timing check task:
$setup(posedge CK &&& (E != 1'b1), SE, tsetup$SE$CK,
      notifier);
```

Input Hold Time

- Hold time between posedge of D and posedge of CK.

```
// SDF:
(HOLD (posedge D) (posedge CK) (-0.166:-0.166:-0.166))

-- VHDL generic:
thold_D_CK_posedge_posedge: VitalDelayType;

// Verilog HDL timing check task:
$hold (posedge CK, posedge D, thold$D$CK, notifier);
```

- Hold time between RN and SN.

```
// SDF:
(HOLD (posedge RN) (posedge SN) (-0.261:-0.261:-0.261))

-- VHDL generic:
thold_RN_SN_posedge_posedge: VitalDelayType;

// Verilog HDL timing check task:
$hold (posedge SN, posedge RN, thold$RN$SN, notifier);
```

- Hold time between input port SI and reference port CK.

```
// SDF:
(HOLD (negedge SI) (posedge CK) (-0.110:-0.110:-0.110))

-- VHDL generic:
thold_SI_CK_negedge_posedge: VitalDelayType;

// Verilog HDL timing check task:
$hold (posedge CK, negedge SI, thold$SI$CK, notifier);
```

- Conditional hold time between E and posedge of CK.

```
// SDF:
(HOLD (COND SE ^ RN == 0 E) (posedge CK))

-- VHDL generic:
thold_E_CK_SE_XOB_RN_EQ_0_noedge_posedge:
 VitalDelayType;

// Verilog HDL timing check task:
$hold (posedge CK &&& (SE ^ RN == 0), posedge E,
      thold$E$CK, NOTIFIER);
```

Input Setup and Hold Time

- Setup and hold timing check between D and CLK. It is a conditional check. The first delay value is the setup time and the second delay value is the hold time.

```
// SDF:
(SETUPHOLD (COND SE ^ RN == 0 D) (posedge CLK)
 (0.69) (0.32))

-- VHDL generic (split up into separate setup and hold):
tsetup_D_CK_SE_XOB_RN_EQ_0_noedge_posedge:
 VitalDelayType;
thold_D_CK_SE_XOB_RN_EQ_0_noedge_posedge:
 VitalDelayType;

-- Verilog HDL timing check (it can either be split up or
-- kept as one construct depending on what appears in the
-- Verilog HDL model):
$setuphold(posedge CK &&& (SE ^ RN == 1'b0)), posedge D,
          tsetup$D$CK, thold$D$CK, notifier);
-- Or as:
```

```
$setup(posedge CK &&& (SE ^ RN == 1'b0)), posedge D,
      tsetup$D$CK, notifier);
$hold(posedge CK &&& (SE ^ RN == 1'b0)), posedge D,
     thold$D$CK, notifier);
```

Input Recovery Time

- Recovery time between CLKA and CLKB.

```
// SDF:
(RECOVERY (posedge CLKA) (posedge CLKB)
  (1.119:1.119:1.119))

-- VHDL generic:
trecovery_CLKA_CLKB_posedge_posedge: VitalDelayType;

// Verilog timing check task:
$recovery (posedge CLKB, posedge CLKA,
          trecovery$CLKB$CLKA, notifier);
```

- Conditional recovery time between posedge of CLKA and posedge of CLKB.

```
// SDF:
(RECOVERY (posedge CLKB)
  (COND ENCLKBCLKArec  (posedge CLKA)) (0.55:0.55:0.55)
)

-- VHDL generic:
trecovery_CLKB_CLKA_ENCLKBCLKArec_EQ_1_posedge_
posedge: VitalDelayType;

// Verilog timing check task:
$recovery (posedge CLKA && ENCLKBCLKArec, posedge CLKB,
           trecovery$CLKA$CLKB, notifier);
```

- Recovery time between SE and CK.

```
// SDF:
(RECOVERY SE (posedge CK) (1.901))

-- VHDL generic:
trecovery_SE_CK_noedge_posedge: VitalDelayType;

// Verilog timing check task:
$recovery (posedge CK, SE, trecovery$SE$CK, notifier);
```

- Recovery time between RN and CK.

```
// SDF:
(RECOVERY (COND D == 0 (posedge RN)) (posedge CK) (0.8))

-- VHDL generic:
trecovery_RN_CK_D_EQ_0_posedge_posedge:
 VitalDelayType;

// Verilog timing check task:
$recovery (posedge CK && (D == 0), posedge RN,
          trecovery$RN$CK, notifier);
```

Input Removal Time

- Removal time between posedge of E and negedge of CK.

```
// SDF:
(REMOVAL (posedge E) (negedge CK) (0.4:0.4:0.4))

-- VHDL generic:
tremoval_E_CK_posedge_negedge: VitalDelayType;
```

```
// Verilog timing check task:
$removal (negedge CK, posedge E, tremoval$E$CK,
         notifier);
```

- Conditional removal time between posedge of CK and SN.

```
// SDF:
(REMOVAL (COND D != 1'b1 SN) (posedge CK) (1.512))

-- VHDL generic:
tremoval_SN_CK_D_NE_1_noedge_posedge : VitalDelayType;

// Verilog timing check task:
$removal (posedge CK &&& (D != 1'b1), SN,
         tremoval$SN$CK, notifier);
```

Period

- Period of input CLKB.

```
// SDF:
(PERIOD CLKB (0.803:0.803:0.803))

-- VHDL generic:
tperiod_CLKB: VitalDelayType;

// Verilog timing check task:
$period (CLKB, tperiod$CLKB);
```

- Period of input port EN.

```
// SDF:
(PERIOD EN (1.002:1.002:1.002))
```

```
-- VHDL generic:
tperiod_EN : VitalDelayType;

// Verilog timing check task:
$period (EN, tperiod$EN);
```

- Period of input port TCK.

```
// SDF:
(PERIOD (posedge TCK) (0.220))

-- VHDL generic:
tperiod_TCK_posedge: VitalDelayType;

// Verilog timing check task:
$period (posedge TCK, tperiod$TCK);
```

Pulse Width

- Pulse width of high pulse of CK.

```
// SDF:
(WIDTH (posedge CK) (0.103:0.103:0.103))

-- VHDL generic:
tpw_CK_posedge: VitalDelayType;

// Verilog timing check task:
$width (posedge CK, tminpwh$CK, 0, notifier);
```

- Pulse width for a low pulse CK.

```
// SDF:
(WIDTH (negedge CK) (0.113:0.113:0.113))
```

```
-- VHDL generic:
tpw_CK_negedge: VitalDelayType;

// Verilog timing check task:
$width (negedge CK, tminpwl$CK, 0, notifier);
```

- Pulse width for a high pulse on RN.

```
// SDF:
(WIDTH (posedge RN) (0.122))

-- VHDL generic:
tpw_RN_posedge: VitalDelayType;

// Verilog timing check task:
$width (posedge RN, tminpwh$RN, 0, notifier);
```

Input Skew Time

- Skew between CK and TCK.

```
// SDF:
(SKEW (negedge CK) (posedge TCK) (0.121))

-- VHDL generic:
tskew_CK_TCK_negedge_posedge: VitalDelayType;

// Verilog timing check task:
$skew (posedge TCK, negedge CK, tskew$TCK$CK, notifier);
```

- Skew between SE and negedge of CK.

```
// SDF:
(SKEW SE (negedge CK) (0.386:0.386:0.386))
```

```
-- VHDL generic:
tskew_SE_CK_noedge_negedge: VitalDelayType;

// Verilog HDL timing check task:
$skew (negedge CK, SE, tskew$SE$CK, notifier);
```

No-change Setup Time

The SDF NOCHANGE construct maps to both *tncsetup* and *tnchold* VHDL generics.

- No-change setup time between D and negedge CK.

```
// SDF:
(NOCHANGE D (negedge CK) (0.343:0.343:0.343))

-- VHDL generic:
tncsetup_D_CK_noedge_negedge: VitalDelayType;
tnchold_D_CK_noedge_negedge: VitalDelayType;

// Verilog HDL timing check task:
$nochange (negedge CK, D, tnochange$D$CK, notifier);
```

No-change Hold Time

The SDF NOCHANGE construct maps to both *tncsetup* and *tnchold* VHDL generics.

- Conditional no-change hold time between E and CLKA.

```
// SDF:
(NOCHANGE (COND RST == 1'b1 (posedge E)) (posedge CLKA)
  (0.312))
```

```
-- VHDL generic:
tnchold_E_CLKA_RST_EQ_1_posedge_posedge:
 VitalDelayType;
tncsetup_E_CLKA_RST_EQ_1_posedge_posedge:
 VitalDelayType;

// Verilog HDL timing check task:
$nochange (posedge CLKA &&& (RST == 1'b1), posedge E,
         tnochange$E$CLKA, notifier);
```

Port Delay

- Delay to port OE.

```
// SDF:
(PORT OE (0.266))

-- VHDL generic:
tipd_OE: VitalDelayType01;

// Verilog HDL:
No explicit Verilog declaration.
```

- Delay to port RN.

```
// SDF:
(PORT RN (0.201:0.205:0.209))

-- VHDL generic:
tipd_RN : VitalDelayType01;

// Verilog HDL:
No explicit Verilog declaration.
```

Net Delay

- Delay on net connected to port CKA.

```
// SDF:
(NETDELAY CKA (0.134))

-- VHDL generic:
tipd_CKA: VitalDelayType01;

// Verilog HDL:
No explicit Verilog declaration.
```

Interconnect Path Delay

- Interconnect path delay from port Y to port D.

```
// SDF:
(INTERCONNECT bcm/credit_manager/U304/Y
 bcm/credit_manager/frame_in/PORT0_DOUT_Q_reg_26_/D
 (0.002:0.002:0.002) (0.002:0.002:0.002))

-- VHDL generic of instance
-- bcm/credit_manager/frame_in/PORT0_DOUT_Q_reg_26_:
tipd_D: VitalDelayType01;
-- The "from" port does not contribute to the timing
-- generic name.

// Verilog HDL:
No explicit Verilog declaration.
```

Device Delay

- Device delay of output SM of instance uP.

```
// SDF:
(INSTANCE uP) . . . (DEVICE SM . . .

-- VHDL generic:
tdevice_uP_SM

// Verilog specify paths:
// All specify paths to output SM.
```

B.5 Complete Syntax

Here is the complete syntax[1] for SDF shown using the BNF form. Terminal names are in uppercase, keywords are in bold uppercase but are case insensitive. The start terminal is `delay_file`.

```
absolute_deltype ::= ( ABSOLUTE del_def { del_def } )

alphanumeric ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p |
      q | r | s | t | u | v | w | x | y | z
  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
      Q | R | S | T | U | V | W | X | Y | Z
  | _ | $
  | decimal_digit

any_character ::=
    character
  | special_character
  | \"
```

1. The syntax is reprinted with permission from IEEE Std. 1497-2001, Copyright 2001, by IEEE. All rights reserved.

```
arrival_env ::=
  ( ARRIVAL [ port_edge ] port_instance rvalue rvalue
    rvalue rvalue )

bidirectskew_timing_check ::=
  ( BIDIRECTSKEW port_tchk port_tchk value value )

binary_operator ::=
    +
  | -
  | *
  | /
  | %
  | ==
  | !=
  | ===
  | !==
  | &&
  | ||
  | <
  | <=
  | >
  | >=
  | &
  | |
  | ^
  | ^~
  | ~^
  | >>
  | <<

bus_net ::= hierarchical_identifier [ integer : integer ]

bus_port ::= hierarchical_identifier [ integer : integer ]

ccond ::= ( CCOND [ qstring ] timing_check_condition )

cell ::= ( CELL celltype cell_instance { timing_spec } )

celltype ::= ( CELLTYPE qstring )
```

```
cell_instance ::=
   ( INSTANCE [ hierarchical_identifier ] )
  | ( INSTANCE * )

character ::=
   alphanumeric
  | escaped_character

cns_def ::=
   path_constraint
  | period_constraint
  | sum_constraint
  | diff_constraint
  | skew_constraint

concat_expression ::= , simple_expression

condelse_def ::= ( CONDELSE iopath_def )

conditional_port_expr ::=
   simple_expression
  | ( conditional_port_expr )
  | unary_operator ( conditional_port_expr )
  | conditional_port_expr binary_operator
     conditional_port_expr

cond_def ::=
  ( COND [ qstring ] conditional_port_expr iopath_def )

constraint_path ::= ( port_instance port_instance )

date ::= ( DATE qstring )

decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

delay_file ::= ( DELAYFILE sdf_header cell { cell } )

deltype ::=
   absolute_deltype
  | increment_deltype
  | pathpulse_deltype
  | pathpulsepercent_deltype
```

```
delval ::=
   rvalue
  | ( rvalue rvalue )
  | ( rvalue rvalue rvalue )

delval_list ::=
   delval
  | delval delval
  | delval delval delval
  | delval delval delval delval [ delval ] [ delval ]
  | delval delval delval delval delval delval delval
     [ delval ] [ delval ] [ delval ] [ delval ] [ delval ]

del_def ::=
   iopath_def
  | cond_def
  | condelse_def
  | port_def
  | interconnect_def
  | netdelay_def
  | device_def

del_spec ::= ( DELAY deltype { deltype } )

departure_env ::=
  ( DEPARTURE [ port_edge ] port_instance rvalue rvalue
    rvalue rvalue )

design_name ::= ( DESIGN qstring )

device_def ::= ( DEVICE [ port_instance ] delval_list )

diff_constraint ::=
  ( DIFF constraint_path constraint_path value [ value ] )

edge_identifier ::=
   posedge
  | negedge
  | 01
  | 10
  | 0z
  | z1
```

```
  | 1z
  | z0

edge_list ::=
    pos_pair { pos_pair }
  | neg_pair { neg_pair }

equality_operator ::=
    ==
  | !=
  | ===
  | !==

escaped_character ::=
    \ character
  | \ special_character
  | \"

exception ::= ( EXCEPTION cell_instance { cell_instance } )

hchar := . | /

hierarchical_identifier ::= identifier { hchar identifier }

hierarchy_divider ::= ( DIVIDER hchar )

hold_timing_check ::= ( HOLD port_tchk port_tchk value )

identifier ::= character { character }

increment_deltype ::= ( INCREMENT del_def { del_def } )

input_output_path ::= port_instance port_instance

integer ::= decimal_digit { decimal_digit }

interconnect_def ::=
  ( INTERCONNECT port_instance port_instance delval_list )

inversion_operator ::=
    !
  | ~
```

```
iopath_def ::=
  ( IOPATH port_spec port_instance { retain_def } delval_list )

lbl_def ::= ( identifier delval_list )

lbl_spec ::= ( LABEL lbl_type { lbl_type } )

lbl_type :=
    ( INCREMENT lbl_def { lbl_def } )
  | ( ABSOLUTE lbl_def { lbl_def } )

name ::= ( NAME qstring )

neg_pair ::=
  ( negedge signed_real_number [ signed_real_number ] )
  ( posedge signed_real_number [ signed_real_number ] )

net ::=
    scalar_net
  | bus_net

netdelay_def ::= ( NETDELAY net_spec delval_list )

net_instance ::=
    net
  | hierarchical_identifier hier_divider_char net

net_spec ::=
    port_instance
  | net_instance

nochange_timing_check ::=
  ( NOCHANGE port_tchk port_tchk rvalue rvalue )

pathpulsepercent_deltype ::=
  ( PATHPULSEPERCENT [ input_output_path ] value [ value ] )

pathpulse_deltype ::=
  ( PATHPULSE [ input_output_path ] value [ value ] )

path_constraint ::=
  ( PATHCONSTRAINT [ name ] port_instance port_instance
    { port_instance } rvalue rvalue )
```

```
period_constraint ::=
  ( PERIODCONSTRAINT port_instance value [ exception ] )

period_timing_check ::= ( PERIOD port_tchk value )

port ::=
    scalar_port
  | bus_port

port_def ::= ( PORT port_instance delval_list )

port_edge ::= ( edge_identifier port_instance )

port_instance ::=
    port
  | hierarchical_identifier hchar port

port_spec ::=
    port_instance
  | port_edge

port_tchk ::=
    port_spec
  | ( COND [ qstring ] timing_check_condition port_spec )

pos_pair ::=
  ( posedge signed_real_number [ signed_real_number ] )
  ( negedge signed_real_number [ signed_real_number ] )

process ::= ( PROCESS qstring )

program_name ::= ( PROGRAM qstring )

program_version ::= ( VERSION qstring )

qstring ::= " { any_character } "

real_number ::=
    integer
  | integer [ . integer ]
  | integer [ . integer ] e [ sign ] integer
```

```
recovery_timing_check ::=
  ( RECOVERY port_tchk port_tchk value )

recrem_timing_check ::=
    ( RECREM port_tchk port_tchk rvalue rvalue )
  | ( RECREM port_spec port_spec rvalue rvalue
      [ scond ] [ ccond ] )

removal_timing_check ::= ( REMOVAL port_tchk port_tchk value )

retain_def ::= ( RETAIN retval_list )

retval_list ::=
    delval
  | delval delval
  | delval delval delval

rtriple ::=
    signed_real_number : [ signed_real_number ] :
      [ signed_real_number ]
  | [ signed_real_number ] : signed_real_number :
      [ signed_real_number ]
  | [ signed_real_number ] : [ signed_real_number ] :
      signed_real_number

rvalue ::=
    ( [ signed_real_number ] )
  | ( [ rtriple ] )

scalar_constant ::=
    0
  | 'b0
  | 'B0
  | 1'b0
  | 1'B0
  | 1
  | 'b1
  | 'B1
  | 1'b1
  | 1'B1
```

```
scalar_net ::=
    hierarchical_identifier
  | hierarchical_identifier [ integer ]

scalar_node ::=
    scalar_port
  | hierarchical_identifier

scalar_port ::=
    hierarchical_identifier
  | hierarchical_identifier [ integer ]

scond ::= ( SCOND [ qstring ] timing_check_condition )

sdf_header ::=
  sdf_version [ design_name ] [ date ] [ vendor ]
  [ program_name ] [ program_version ] [ hierarchy_divider ]
  [ voltage ] [ process ] [ temperature ] [ time_scale ]

sdf_version ::= ( SDFVERSION qstring )

setuphold_timing_check ::=
    ( SETUPHOLD port_tchk port_tchk rvalue rvalue )
  | ( SETUPHOLD port_spec port_spec rvalue rvalue
      [ scond ] [ ccond ] )

setup_timing_check ::= ( SETUP port_tchk port_tchk value )

sign ::= + | -

signed_real_number ::= [ sign ] real_number

simple_expression ::=
    ( simple_expression )
  | unary_operator ( simple_expression )
  | port
  | unary_operator port
  | scalar_constant
  | unary_operator scalar_constant
  | simple_expression ? simple_expression : simple_expression
  | { simple_expression [ concat_expression ] }
  | { simple_expression { simple_expression
      [ concat_expression ] } }
```

```
skew_constraint ::= ( SKEWCONSTRAINT port_spec value )

skew_timing_check ::= ( SKEW port_tchk port_tchk rvalue )

slack_env ::=
  ( SLACK port_instance rvalue rvalue rvalue rvalue
    [ real_number ] )

special_character ::=
  ! | # | % | & | ‘ | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? |
    @ | [ | \ | ] | ^ | ` | { | | | } | ~

sum_constraint ::=
  ( SUM constraint_path constraint_path { constraint_path }
    rvalue [ rvalue ] )

tchk_def ::=
    setup_timing_check
  | hold_timing_check
  | setuphold_timing_check
  | recovery_timing_check
  | removal_timing_check
  | recrem_timing_check
  | skew_timing_check
  | bidirectskew_timing_check
  | width_timing_check
  | period_timing_check
  | nochange_timing_check

tc_spec ::= ( TIMINGCHECK tchk_def { tchk_def } )

temperature ::=
    ( TEMPERATURE rtriple )
  | ( TEMPERATURE signed_real_number )

tenv_def ::=
    arrival_env
  | departure_env
  | slack_env
  | waveform_env
```

```
te_def ::=
    cns_def
  | tenv_def

te_spec ::= ( TIMINGENV te_def { te_def } )

timescale_number ::= 1 | 10 | 100 | 1.0 | 10.0 | 100.0

timescale_unit ::= s | ms | us | ns | ps | fs

time_scale ::= ( TIMESCALE timescale_number timescale_unit )

timing_check_condition ::=
    scalar_node
  | inversion_operator scalar_node
  | scalar_node equality_operator scalar_constant

timing_spec ::=
    del_spec
  | tc_spec
  | lbl_spec
  | te_spec

triple ::=
    real_number : [ real_number ] : [ real_number ]
  | [ real_number ] : real_number : [ real_number ]
  | [ real_number ] : [ real_number ] : real_number

unary_operator ::=
    +
  | -
  | !
  | ~
  | &
  | ~&
  | |
  | ~|
  | ^
  | ^~
  | ~^
```

```
value ::=
    ( [ real_number ] )
  | ( [ triple ] )

vendor ::= ( VENDOR qstring )

voltage ::=
    ( VOLTAGE rtriple )
  | ( VOLTAGE signed_real_number )

waveform_env ::=
  ( WAVEFORM port_instance real_number edge_list )

width_timing_check ::= ( WIDTH port_tchk value )
```

❑

APPENDIX

C

Standard Parasitic Extraction Format (SPEF)

This appendix describes the Standard Parasitic Extraction Format (SPEF). It is part of the IEEE Std 1481.

C.1 Basics

SPEF allows the description of parasitic information of a design (R, L and C) in an ASCII exchange format. A user can read and check values in a SPEF file, though the user would never create this file manually. It is mainly

used to pass parasitic information from one tool to another. Figure C-1 shows that SPEF can be generated by tools such as a place-and-route tool or a parasitic extraction tool, and then used by a timing analysis tool, in circuit simulation or to perform crosstalk analysis.

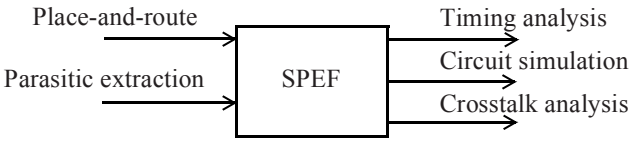


Figure C-1 *SPEF is a tool exchange medium.*

Parasitics can be represented at many different levels. SPEF supports the distributed net model, the reduced net model and the lumped capacitance model. In the distributed net model (`D_NET`), each segment of a net route has its own `R` and `C`. In a reduced net model (`R_NET`), only a single reduced `R` and `C` is considered on the load pins of the net and a pie model (`C-R-C`) is considered on the driver pin of the net. In a lumped capacitance model, only a single capacitance is specified for the entire net. Figure C-2 shows an example of a physical net route. Figure C-3 shows the distributed net model. Figure C-4 shows the reduced net model and Figure C-5 shows the lumped capacitance model.

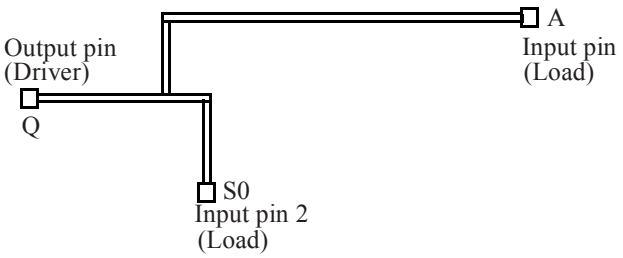


Figure C-2 *A layout of a net.*

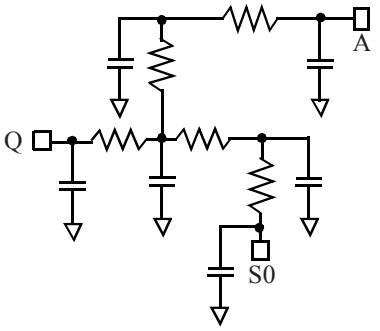


Figure C-3 *Distributed net (D_NET) model.*

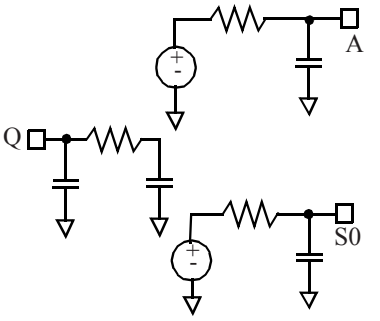


Figure C-4 *Reduced net (R_NET) model.*

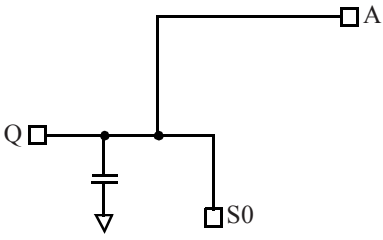


Figure C-5 *Lumped capacitance model.*

Interconnect parasitics depends on process. SPEF supports the specification of best-case, typical, and worst-case values. Such triplets are allowed for R, L and C values, port slews and loads.

By providing a name map consisting of a map of net names and instance names to indices, the SPEF file size is made effectively smaller, and more importantly, all long names appear in only one place.

A SPEF file for a design can be split across multiple files and can also be hierarchical.

C.2 Format

The format of a SPEF file is as follows.

```
header_definition
[ name_map ]
[ power_definition ]
[ external_definition ]
[ define_definition ]
internal_definition
```

The *header definition* contains basic information such as the SPEF version number, design name and units for R, L and C. The *name map* specifies the mapping of net names and instance names to indices. The *power definition* declares the power nets and ground nets. The *external definition* defines the ports of the design. The *define definition* identifies instances, whose SPEF is described in additional files. The *internal definition* contains the guts of the file, which are the parasitics of the design.

Figure C-6 shows an example of a header definition.

```
*SPEF "IEEE 1481-1998"
*DESIGN "ddrphy"
*DATE "Thu Oct 21 00:49:32 2004"
*VENDOR "SGP Design Automation"
*PROGRAM "Galaxy-RCXT"
*VERSION "V2000.06 "
*DESIGN_FLOW "PIN_CAP NONE" "NAME_SCOPE
LOCAL"
*DIVIDER /
*DELIMITER :
*BUS_DELIMITER [ ]
*T_UNIT 1.00000 NS
*C_UNIT 1.00000 FF
*R_UNIT 1.00000 OHM
*L_UNIT 1.00000 HENRY

// A comment starts with the two characters "//".

// TCAD_GRD_FILE /cad/13lv/galaxy-rcxt/
t013s6ml_fsg.nxtgrd
// TCAD_TIME_STAMP Tue May 14 22:19:36 2002
```

Figure C-6 *A header definition.*

`*SPEF name`

specifies the SPEF version.

`*DESIGN name`

specifies the design name.

`*DATE string`

specifies the time stamp when the file was created.

`*VENDOR string`

specifies the vendor tool that was used to create the SPEF.

*PROGRAM string

specifies the program that was used to generate the SPEF.

*VERSION string

specifies the version number of the program that was used to create the SPEF.

*DESIGN_FLOW string string string . . .

specifies at what stage the SPEF file was created. It describes information about the SPEF file that cannot be derived by reading the file. The predefined string values are:

- **EXTERNAL_LOADS** : External loads are fully specified in the SPEF file.
- **EXTERNAL_SLEWS** : External slews are fully specified in the SPEF file.
- **FULL_CONNECTIVITY** : Logical netlist connectivity is present in the SPEF.
- **MISSING_NETS** : Some logical nets may be missing from the SPEF file.
- **NETLIST_TYPE_VERILOG** : Uses Verilog HDL type naming conventions.
- **NETLIST_TYPE_VHDL87** : Uses VHDL87 naming convention.
- **NETLIST_TYPE_VHDL93** : Uses VHDL93 netlist naming convention.
- **NETLIST_TYPE_EDIF** : Uses EDIF type naming convention.
- **ROUTING_CONFIDENCE** positive_integer : Default routing confidence number for all nets, basically the level of accuracy of the parasitics.

- **ROUTING_CONFIDENCE_ENTRY** positive_integer string : Supplements the routing confidence values.
- **NAME_SCOPE LOCAL** | **FLAT** : Specifies whether paths in the SPEF file are relative to file or to top of design.
- **SLEW_THRESHOLDS** low_input_threshold_percent high_input_threshold_percent : Specifies the default input slew threshold for the design.
- **PIN_CAP NONE** | **INPUT_OUTPUT** | **INPUT_ONLY** : Specifies what type of pin capacitances are included as part of total capacitance. The default is **INPUT_OUTPUT**.

The line in the header definition:

*DIVIDER /

specifies the hierarchy delimiter. Other characters that can be used are ., :, and /.

*DELIMITER :

specifies the delimiter between an instance and its pin. Other possible characters that can be used are ., /, :, or |.

*BUS_DELIMITER [ ]

specifies the prefix and suffix that are used to identify a bit of a bus. Other possible characters that can be used for prefix and suffix are {, (, <, :, . and },), >.

*T_UNIT positive_integer **NS** | **PS**

specifies the time unit.

*C_UNIT positive_integer **PF** | **FF**

specifies the capacitance unit.

```
*R_UNIT positive_integer OHM | KOHM
```

specifies the resistance unit.

```
*L_UNIT positive_integer HENRY | MH | UH
```

specifies the inductance unit.

A *comment* in a SPEF file can appear in two forms.

```
// Comment - until end of line.

/* This comment can
extend across multiple
lines */
```

Figure C-7 shows an example of a *name map*. It is of the form:

```
*NAME_MAP
*positive_integer name
*positive_integer name
. . .
```

The name map specifies the mapping of names to unique integer values (their indices). The name map helps in reducing the file size by making all future references of the name by the index. A name can be a net name or an instance name. Given the name map in Figure C-7, the names can later be referenced in the SPEF file by using their index, such as:

```
*364:D          // D pin of instance
       // mcdll_write_data/write19/d_out_2x_reg_19
*11172:Y        // Y pin of instance
                // Tie_VSSQ_assign_buf_318_N_1
```

```
*NAME_MAP
*1 memclk
*2 memclk_2x
*3 reset_
*4 refresh
*5 resync
*6 int_d_out[63]
*7 int_d_out[62]
*8 int_d_out[61]
*9 int_d_out[60]
*10 int_d_out[59]
*11 int_d_out[58]
*12 int_d_out[57]
. . .
*364 mcdll_write_data/write19/d_out_2x_reg_19
*366 mcdll_write_data/write20/d_out_2x_reg_20
*368 mcdll_write_data/write21/d_out_2x_reg_21
. . .
*5423 mcdll_read_data/read21/capture_data[53]
. . .
*5426 mcdll_read_data/read21/capture_pos_0[21]
. . .
*11172 Tie_VSSQ_assign_buf_318_N_1
. . .
*14954 test_se_15_S0
*14955 wr_sdly_course_enc[0]_L0
*14956 wr_sdly_course_enc[0]_L0_1
*14957 wr_sdly_course_enc[0]_S0
```

Figure C-7 *A name map.*

```
*5426:116      // Internal node of net
      // mcdll_read_data/read21/capture_pos_0[21]
*5426:10278    // Internal node of net *5426
*12            // The net int_d_out[57]
```

The name map thus avoids repeating long names and their paths by using their unique integer representation.

The *power definition* section defines the power and ground nets.

```
*POWER_NETS net_name net_name . . .
*GROUND_NETS net_name net_name . . .
```

Here are some examples.

```
*POWER_NETS VDDQ
*GROUND_NETS VSSQ
```

The *external definition* contains the definition of the logical and physical ports of the design. Figure C-8 shows an example of logical ports. Logical ports are described in the form:

```
*PORTS
port_name direction { conn_attribute }
port_name direction { conn_attribute }
. . .
```

where a *port_name* can be the port index of form `*positive_integer`. The *direction* is `I` for input, `O` for output and `B` for bidirectional. *Connection attributes* are optional, and can be the following:

- ***C** `number number` : Coordinates of the port.
- ***L** `par_value` : Capacitive load of the port.
- ***S** `par_value par_value` : Defines the shape of the waveform on the port.
- ***D** `cell_type` : Defines the driving cell of the port.

Physical ports in a SPEF file are defined using:

```
*PHYSICAL_PORTS
pport_name direction { conn_attribute }
pport_name direction { conn_attribute }
. . .
```

```
*PORTS
*1 I
*2 I
*3 I
*4 I
*5 I
*6 I
*7 I
*8 I
*9 I
*10 I
*11 I
...
*450 O
*451 O
*452 O
*453 O
*454 O
*455 O
*456 O
```

Figure C-8 *An external definition.*

The *define definition* section defines entity instances that are referenced in the current SPEF file but whose parasitics are described in additional SPEF files.

```
*DEFINE instance_name { instance_name } entity_name
*PDEFINE physical_instance entity_name
```

The `*PDEFINE` is used when the entity instance is a physical partition (instead of a logical hierarchy). Here are some examples.

```
*DEFINE core/u1ddrphy core/u2ddrphy "ddrphy"
```

This implies that there would be another SPEF file with a `*DESIGN` value of `ddrphy` - this file would contain the parasitics for the design `ddrphy`. It is

possible to have physical and logical hierarchy. Any nets that cross the hierarchical boundaries have to be described as distributed nets (D_NET).

The *internal definition* forms the guts of the SPEF file - it describes the parasitics for the nets in the design. There are basically two forms: the *distributed net*, D_NET, and the *reduced net*, R_NET. Figure C-9 shows an example of a distributed net definition.

```
*D_NET *5426 0.899466

*CONN
*I *14212:D I *C 21.7150 79.2300
*I *14214:Q O *C 21.4950 76.6000 *D DFFQX1

*CAP
1 *5426:10278 *5290:8775 0.217446
2 *5426:10278 *16:3754 0.0105401
3 *5426:10278 *5266:9481 0.0278254
4 *5426:10278 *5116:9922 0.113918
5 *5426:10278 0.529736

*RES
1 *5426:10278 *14212:D 0.340000
2 *5426:10278 *5426:10142 0.916273
3 *5426:10142 *14214:Q 0.340000
*END
```

Figure C-9 *Distributed net parasitics for net *5426.*

In the first line,

```
*D_NET *5426 0.899466
```

*5426 is the net index (see name map for the net name) and 0.899466 is the total capacitance value on the net. The capacitance value is the sum of all capacitances on the net including cross-coupling capacitances that are assumed to be grounded, and including load capacitances. It may or may not

include pin capacitances depending on the setting of PIN_CAP in the *DESIGN_FLOW definition.

The *connectivity section* describes the drivers and loads for the net. In:

```
*CONN
*I *14212:D I *C 21.7150 79.2300
*I *14214:Q O *C 21.4950 76.6000 *D DFFQX1
```

*I refers to an internal pin (*P is used for a port), *14212:D refers to the D pin of instance *14212 which is an index (see name map for actual name). "I" says that it is a load (input pin) on the net. "O" says that it is a driver (output pin) on the net. *C and *D are as defined earlier in connection attributes - *C defines the coordinates of the pin and *D defines the driving cell of the pin.

The *capacitance section* describes the capacitances of the distributed net. The capacitance unit is as specified earlier with *C_UNIT.

```
*CAP
1 *5426:10278 *5290:8775 0.217446
2 *5426:10278 *16:3754 0.0105401
3 *5426:10278 *5266:9481 0.0278254
4 *5426:10278 *5116:9922 0.113918
5 *5426:10278 0.529736
```

The first number is the capacitance identifier. There are two forms of capacitance specification; the first through fourth are of one form and the fifth is of the second form. The first form (first through fourth) specifies the cross-coupling capacitances between two nets, while the second form (with id 5) specifies the capacitance to ground. So in capacitance id 1, the cross-coupling capacitance between nets *5426 and *5290 is 0.217446. And in capacitance id 5, the capacitance to ground is 0.529736. Notice that the first node name is necessarily the net name for the D_NET that is being described. The positive integer following the net index (10278 in *5426:10278) specifies an internal node or junction point. So capacitance id 4 states that there

is a coupling capacitance between net *5426 with internal node 10278 and net *5116 with internal node 9922, and the value of this coupling capacitance is 0.113918.

The *resistance section* describes the resistances of the distributed net. The resistance unit is as specified with ***R_UNIT**.

```
*RES
1 *5426:10278 *14212:D 0.340000
2 *5426:10278 *5426:10142 0.916273
3 *5426:10142 *14214:Q 0.340000
```

The first field is the resistance identifier. So there are three resistance components for this net. The first one is between the internal node *5426:10278 to the D pin on *14212 and the resistance value is 0.34. The capacitance and resistance section can be better understood with the RC network shown pictorially in Figure C-10.

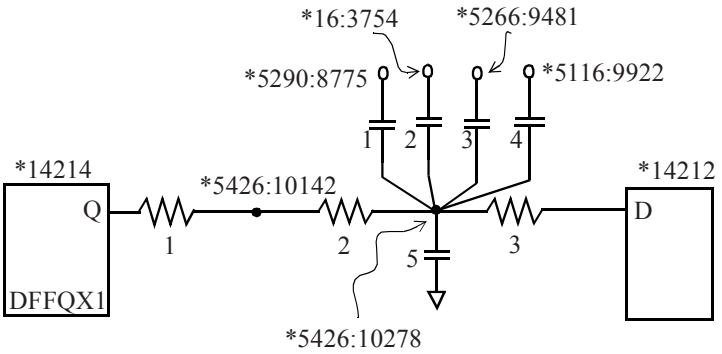


Figure C-10 *RC for net *5426.*

Figure C-11 shows another example of a distributed net. This net has one driver and two loads and the total capacitance on the net is 2.69358. Figure C-12 shows the RC network that corresponds to the distributed net specification.

```
*D_NET *5423 2.69358

*CONN
*I *14207:D I *C 21.7450 94.3150
*I *14205:D I *C 21.7450 90.4900
*I *14211:Q O *C 21.4900 83.8800 *D DFFQX1

*CAP
1 *5423:10107 *547:12722 0.202686
2 *5423:10107 *5116:10594 0.104195
3 *5423:10107 *5233:9552 0.208867
4 *5423:10107 *5265:9483 0.0225810
5 *5423:10107 *267:9668 0.0443454
6 *5423:10107 *5314:7853 0.120589
7 *5423:10212 *2109:996 0.0293744
8 *5423:10212 *5187:7411 0.526945
9 *5423:14640 *6577:10075 0.126929
10 *5423:10213 1.30707

*RES
1 *5423:10107 *5423:10212 2.07195
2 *5423:10107 *5423:10106 0.340000
3 *5423:10212 *5423:10211 0.340000
4 *5423:10212 *5423:14640 1.17257
5 *5423:14640 *5423:10213 0.340000
6 *5423:10213 *14207:D 0.0806953
7 *5423:10211 *14205:D 0.210835
8 *5423:10106 *14211:Q 0.0932139
*END
```

Figure C-11 *Another example of a distributed net *5423.*

In general, an internal definition can comprise of the following specifications:

- **D_NET**: Distributed RC network form of a logical net.
- **R_NET**: Reduced RC network form of a logical net.
- **D_PNET**: Distributed form of a physical net.
- **R_PNET**: Reduced form of a physical net.

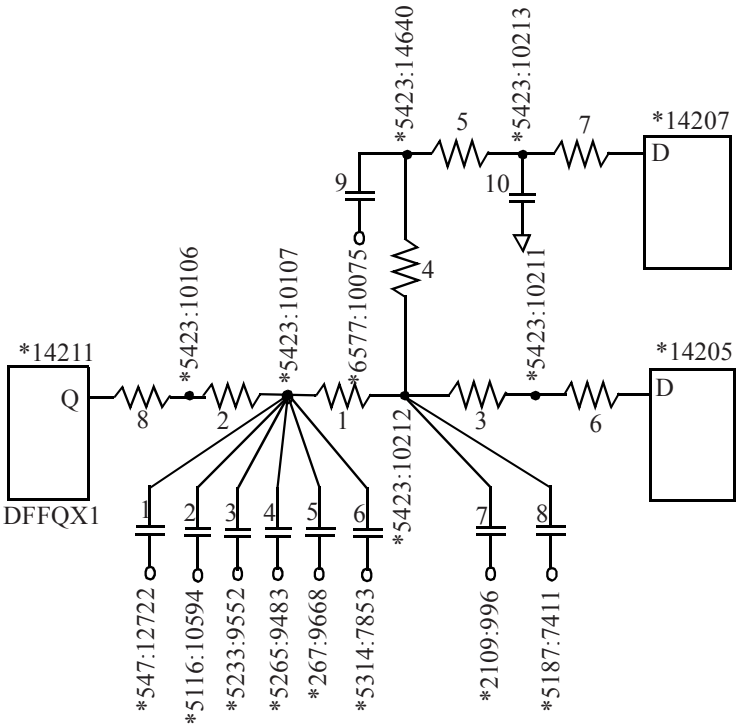


Figure C-12 *RC network for D_NET *5423.*

Here is the syntax.

```
*D_NET net_index total_cap [*V routing_confidence ]
  [ conn_section ]
  [ cap_section ]
  [ res_section ]
  [ inductance_section ]
*END

*R_NET net_index total_cap [ *V routing_confidence ]
  [ driver_reduction ]
```

```
*END

*D_PNET pnet_index total_cap [*V routing_confidence ]
  [ pconn_section ]
  [ pcap_section ]
  [ pres_section ]
  [ pinduc_section ]
*END

*R_PNET pnet_index total_cap [*V routing_confidence ]
  [ pdriver_reduction ]
*END
```

The *inductance section* is used to specify inductances and the format is similar to the resistance section. The ***V** is used to specify the accuracy of the parasitics of the net. These can be specified individually with a net or can be specified globally using the ***DESIGN_FLOW** statement with the `ROUTING_CONFIDENCE` value, such as:

```
*DESIGN_FLOW "ROUTING_CONFIDENCE 100"
```

which specifies that the parasitics were extracted after final cell placement and final route and `3d` extraction was used. Other possible values of routing confidence are:

- 10: Statistical wireload model
- 20: Physical wireload model
- 30: Physical partitions with locations, and no cell placement
- 40: Estimated cell placement with steiner tree based route
- 50: Estimated cell placement with global route
- 60: Final cell placement with steiner route
- 70: Final cell placement with global route
- 80: Final cell placement, final route, `2d` extraction
- 90: Final cell placement, final route, `2.5d` extraction

- 100: Final cell placement, final route, `3d` extraction

A *reduced net* is a net that has been reduced from a distributed net form. There is one driver reduction section for each driver on a net. The driver reduction section is of the form:

```
*DRIVER pin_name
*CELL cell_type
// Driver reduction: one such section for each driver
// of net:
*C2_R1_C1 cap_value res_value cap_value
*LOADS   // One following set for each load on net:
*RC pin_name rc_value
*RC pin_name rc_value
. . .
```

The ***C2_R1_C1** shows the parasitics for the pie model on the driver pin of the net. The *rc_value* in the ***RC** construct is the Elmore delay (`R*C`). Figure C-13 shows an example of a reduced net SPEF and Figure C-14 shows the

```
*R_NET *1200 2.995
*DRIVER *1201:Q
*CELL SEDFFX1
*C2_R1_C1 0.511 2.922 0.106
*LOADS
*RC *1202:A 1.135
*RC *1203:A 0.946
*END
```

Figure C-13 *Reduced net example.*

RC network pictorially.

A *lumped capacitance* model is described using either a ***D_NET** or a ***R_NET** construct with just the total capacitance and with no other information. Here are examples of lumped capacitance declarations.

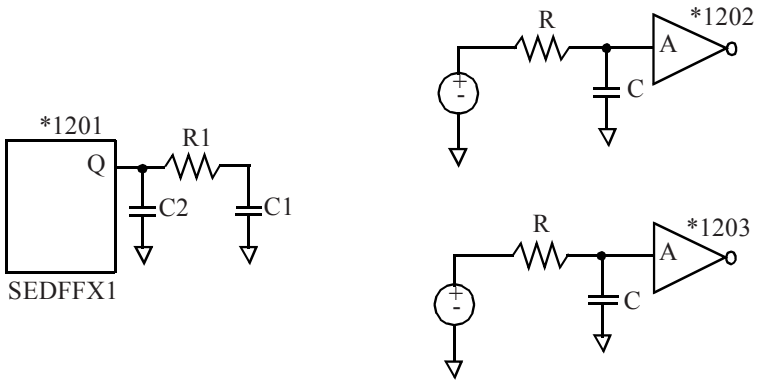


Figure C-14 *Reduced net model.*

```
*D_NET *1 80.2096
*CONN
*I *2:Y O *L 0 *D CLKMX2X2
*P *1 O *L 0
*END

*R_NET *17 58.5204
*END
```

Values in a SPEF file can be in a triplet form that represents the process variations, such as:

```
0.243:0.269:0.300
```

`0.243` is the best-case value, `0.269` is the typical value and `0.300` is the worst-case value.

C.3 Complete Syntax

This section describes the complete syntax[1] of a SPEF file.

A character can be escaped by preceding with a backslash (\). Comments come in two forms: // starts a comment until end of line, while /* . . .*/ is a multi-line comment.

In the following syntax, bold characters such as **(**, **[** are part of the syntax. All constructs are arranged alphabetically and the start symbol is SPEF_file.

```
alpha ::= upper | lower

bit_identifier ::=
    identifier
  | <identifier><prefix_bus_delim><digit>{<digit>}
    [ <suffix_bus_delim> ]

bus_delim_def ::=
  *BUS_DELIMITER prefix_bus_delim [ suffix_bus_delim ]

cap_elem ::=
    cap_id node_name par_value
  | cap_id node_name node_name2 par_value

cap_id ::= pos_integer

cap_load ::= *L par_value

cap_scale ::= *C_UNIT pos_number cap_unit

cap_sec ::= *CAP cap_elem { cap_elem }

cap_unit ::= PF | FF

cell_type ::= index | name
```

1. Syntax is reprinted here with permission from IEEE Std. 1481-1999, Copyright 1999, by IEEE. All rights reserved.

```
cnumber ::= ( real_component imaginary_component )

complex_par_value ::=
    cnumber
  | number
  | cnumber:cnumber:cnumber
  | number:number:number

conf ::= pos_integer

conn_attr ::= coordinates | cap_load | slews | driving_cell

conn_def ::=
  *P external_connection direction { conn_attr }
| *I internal_connection direction { conn_attr }

conn_sec ::=
  *CONN conn_def { conn_def } { internal_node_coord }

coordinates ::= *C number number

date ::= *DATE qstring

decimal ::= [sign]<digit>{<digit>}.{<digit>}

define_def ::= define_entry { define_entry }

define_entry ::=
    *DEFINE inst_name { inst_name } entity
  | *PDEFINE physical_inst entity

design_flow ::= *DESIGN_FLOW qstring [ qstring ]

design_name ::= *DESIGN qstring

digit ::= 0 - 9

direction ::= I | B | O

driver_cell ::= *CELL cell_type

driver_pair ::= *DRIVER pin_name
```

```
driver_reduc ::= driver_pair driver_cell pie_model load_desc

driving_cell ::= *D cell_type

d_net ::=
  *D_NET net_ref total_cap [ routing_conf ]
  [ conn_sec ]
  [ cap_sec ]
  [ res_sec ]
  [ induc_sec ]
  *END

d_pnet ::=
  *D_PNET pnet_ref total_cap [ routing_conf ]
  [ pconn_sec ]
  [ pcap_sec ]
  [ pres_sec ]
  [ pinduc_sec ]
  *END

entity ::= qstring

escaped_char ::= \<escaped_char_set>

escaped_char_set ::= <special_char> | “

exp ::= <radix><exp_char><integer>

exp_char ::= E | e

external_connection ::= port_name | pport_name

external_def ::=
   port_def [ physical_port_def ]
  | physical_port_def

float ::=
   decimal
  | fraction
  | exp

fraction ::= [ sign ].<digit>{<digit>}
```

```
ground_net_def ::= *GROUND_NETS net_name { net_name }

hchar ::= . | / | : | |

header_def ::=
  SPEF_version
  design_name
  date
  vendor
  program_name
  program_version
  design_flow
  hierarchy_div_def
  pin_delim_def
  bus_delim_def
  unit_def

hierarchy_div_def ::= *DIVIDER hier_delim

hier_delim ::= hchar

identifier ::= <identifier_char>{<identifier_char>}

identifier_char ::=
    <escaped_char>
  | <alpha>
  | <digit>
  | _

imaginary_component ::= number

index ::= *<pos_integer>

induc_elem ::= induc_id node_name node_name par_value

induc_id ::= pos_integer

induc_scale ::= *L_UNIT pos_number induc_unit

induc_sec ::= *INDUC induc_elem { induc_elem }

induc_unit ::= HENRY | MH | UH
```

```
inst_name ::= index | path

integer ::= [ sign ]<digit>{<digit>}

internal_connection ::= pin_name | pnode_ref

internal_def ::= nets { nets }

internal_node_coord ::= *N internal_node_name coordinates

internal_node_name ::= <net_ref><pin_delim><pos_integer>

internal_pnode_coord ::= *N internal_pnode_name coordinates

internal_pnode_name ::= <pnet_ref><pin_delim><pos_integer>

load_desc ::= *LOADS rc_desc { rc_desc }

lower ::= a - z

mapped_item ::=
    identifier
  | bit_identifier
  | path
  | name
  | physical_ref

name ::= qstring | identifier

name_map ::= *NAME_MAP name_map_entry { name_map_entry }

name_map_entry ::= index mapped_item

neg_sign ::= -

nets ::= d_net | r_net | d_pnet | r_pnet

net_name ::= net_ref | pnet_ref

net_ref ::= index | path

net_ref2 ::= net_ref
```

```
node_name ::=
    external_connection
  | internal_connection
  | internal_node_name
  | pnode_ref

node_name2 ::=
    node_name
  | <pnet_ref><pin_delim><pos_integer>
  | <net_ref2><pin_delim><pos_integer>

number ::= integer | float

partial_path ::= <hier_delim><bit_identifier>

partial_physical_ref ::= <hier_delim><physical_name>

par_value ::= float | <float>:<float>:<float>

path ::=
  [<hier_delim>]<bit_identifier>{<partial_path>}
  [<hier_delim>]

pcap_elem ::=
    cap_id pnode_name par_value
  | cap_id pnode_name pnode_name2 par_value

pcap_sec ::= *CAP pcap_elem { pcap_elem }

pconn_def ::=
    *P pexternal_connection direction { conn_attr }
  | *I internal_connection direction { conn_attr }

pconn_sec ::=
  *CONN pconn_def { pconn_def } { internal_pnode_coord }

pdriver_pair ::= *DRIVER internal_connection

pdriver_reduc ::= pdriver_pair driver_cell pie_model load_desc

pexternal_connection ::= pport_name

physical_inst ::= index | physical_ref
```

```
physical_name ::= name

physical_port_def ::=
  *PHYSICAL_PORTS pport_entry { pport_entry }

physical_ref ::= <physical_name>{<partial_physical_ref>}

pie_model ::=
  *C2_R1_C1 par_value par_value par_value

pin ::= index | bit_identifier

pinduc_elem ::= induc_id pnode_name pnode_name par_value

pinduc_sec ::=
  *INDUC
  pinduc_elem
  { pinduc_elem }

pin_delim ::= hchar

pin_delim_def ::= *DELIMITER pin_delim

pin_name ::= <inst_name><pin_delim><pin>

pnet_ref ::= index | physical_ref

pnet_ref2 ::= pnet_ref

pnode ::= index | name

pnode_name ::=
    pexternal_connection
  | internal_connection
  | internal_pnode_name
  | pnode_ref

pnode_name2 ::=
    pnode_name
  | <net_ref><pin_delim><pos_integer>
  | <pnet_ref2><pin_delim><pos_integer>

pnode_ref ::= <physical_inst><pin_delim><pnode>
```

```
pole ::= complex_par_value

pole_desc ::= *Q pos_integer pole { pole }

pole_residue_desc ::= pole_desc residue_desc

port_def ::=
  *PORTS
  port_entry
  { port_entry }

pos_decimal ::= <digit>{<digit>}.{<digit>}

port ::= index | bit_identifier

port_entry ::= port_name direction { conn_attr }

port_name ::= [<inst_name><pin_delim>]<port>

pos_exp ::= pos_radix exp_char integer

pos_float ::= pos_decimal | pos_fraction | pos_exp

pos_fraction ::= .<digit>{<digit>}

pos_integer ::= <digit>{<digit>}

pos_number ::= pos_integer | pos_float

pos_radix ::= pos_integer | pos_decimal | pos_fraction

pos_sign ::= +

power_def ::=
    power_net_def [ ground_net_def ]
  | ground_net_def

power_net_def ::= *POWER_NETS net_name { net_name }

pport ::= index | name

pport_entry ::= pport_name direction { conn_attr }
```

```
pport_name ::= [<physical_inst><pin_delim>]<pport>

prefix_bus_delim ::= { | [ | ( | < | : | .

pres_elem ::= res_id pnode_name pnode_name par_value

pres_sec ::=
  *RES
  pres_elem
  { pres_elem }

program_name ::= *PROGRAM qstring

program_version ::= *VERSION qstring

qstring ::= "{qstring_char}"

qstring_char ::= special_char | alpha | digit | white_space | _

radix ::= decimal | fraction

rc_desc ::= *RC pin_name par_value [ pole_residue_desc ]

real_component ::= number

residue ::= complex_par_value

residue_desc := *K pos_integer residue { residue }

res_elem ::= res_id node_name node_name par_value

res_id ::= pos_integer

res_scale ::= *R_UNIT pos_number res_unit

res_sec ::=
  *RES
  res_elem
  { res_elem }

res_unit ::= OHM | KOHM

routing_conf ::= *V conf
```

```
r_net ::=
  *R_NET net_ref total_cap [ routing_conf ]
  { driver_reduc }
  *END

r_pnet ::=
  *R_PNET pnet_ref total_cap [ routing_conf ]
  { pdriver_reduc }
  *END

sign ::= pos_sign | neg_sign

slews ::= *S par_value par_value [ threshold threshold ]

special_char ::=
   ! | # | $ | % | & | ` | ( | ) | * | + | , | - | . | / | : | ; | < | =| >
  | ? | @ | [ | \ | ] | ^ | ‘ | { | | | } | ~

SPEF_file ::=
  header_def
  [ name_map ]
  [ power_def ]
  [ external_def ]
  [ define_def ]
  internal_def

SPEF_version ::= *SPEF qstring

suffix_bus_delim ::= ] | } | ) | >

threshold ::=
   pos_fraction
  | <pos_fraction>:<pos_fraction>:<pos_fraction>

time_scale ::= *T_UNIT pos_number time_unit

time_unit ::= NS | PS

total_cap ::= par_value

unit_def ::= time_scale cap_scale res_scale induc_scale

upper ::= A - Z
```

```
vendor ::= *VENDOR qstring

white_space ::= space | tab
```

❑

Bibliography

1. [ARN51] Arnoldi, W.E., *The principle of minimized iteration in the solution of the matrix eigenvalue problem*, Quarterly of Applied Mathematics, Volume 9, pages 17–25, 1951.

2. [BES07] Best, Roland E., *Phase Locked Loops: Design, Simulation and Applications*, McGraw-Hill Professional, 2007.

3. [BHA99] Bhasker, J., *A VHDL Primer, 3rd edition*, Prentice Hall, 1999.

4. [BHA05] Bhasker, J., *A Verilog HDL Primer, 3rd edition*, Star Galaxy Publishing, 2005.

5. [CEL02] Celik, M., Larry Pileggi and Altan Odabasioglu, *IC Interconnect Analysis*, Springer, 2002.

6. [DAL08] Dally, William J., and John Poulton, *Digital Systems Engineering*, Cambridge University Press, 2008.

7. [ELG05] Elgamel, Mohamed A. and Magdy A. Bayoumi, *Interconnect Noise Optimization in Nanometer Technologies*, Springer, 2005.

8. [KAN03] Kang, S.M. and Yusuf Leblebici, *CMOS Digital Integrated Circuits Analysis and Design, 3rd Edition,* New York: McGraw Hill, 2003.

9. [LIB] *Liberty Users Guide,* available at "http://www.opensourceliberty.org".

10. [MON51] Monroe, M.E., *Theory of Probability,* New York: McGraw Hill, 1951.

11. [MUK86] Mukherjee, A., *Introduction to nMOS & CMOS VLSI Systems Design*, Prentice Hall, 1986.

12. [NAG75] Nagel, Laurence W., *SPICE2: A computer program to simulate semiconductor circuits,* Memorandum No. ERL-M520, University of California, Berkeley, May 1975.

13. [QIA94] Qian, J., S. Pullela and L. Pillegi, *Modeling the "Effective Capacitance" for the RC Interconnect of CMOS Gates*, IEEE Transaction on CAD of ICS, Vol 13, No 12, Dec 94.

14. [RUB83] Rubenstein, J., P. Penfield, Jr., and M. A. Horowitz, *Signal delay in RC tree networks*, IEEE Trans. Computer-Aided Design, Vol. CAD-2, pp. 202-211, 1983.

15. [SDC07] *Using the Synopsys Design Constraints Format: Application Note*, Version 1.7, Synopsys Inc., March 2007.

16. [SRI05] Srivastava, A., D. Sylvester, D. Blaauw, *Statistical Analysis and Optimization for VLSI: Timing and Power*, Springer, 2005.

❑

Index

❑