ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ενσωμάτωση του Μοντέλου Καθυστέρησης CCS σε Εργαλείο Στατικής Ανάλυσης Χρονισμού Ολοκληρωμένων Κυκλωμάτων VLSI

Integration of CCS Timing Model into STA Tool

Διπλωματική Εργασία

Μηνάς Σπανόπουλος Καραλεξίδης

Επιβλέποντες Καθηγητές : Γεώργιος Σταμούλης
Καθηγητής

Νέστωρ Ευμορφόπουλος
Επίκουρος Καθηγητής

Βόλος, Οκτώβριος 2018

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ενσωμάτωση του Μοντέλου Καθυστέρησης CCS σε Εργαλείο Στατικής Ανάλυσης Χρονισμού Ολοκληρωμένων Κυκλωμάτων VLSI

Διπλωματική Εργασία

Μηνάς Σπανόπουλος Καραλεξίδης

Επιβλέποντες : Γεώργιος Σταμούλης
Καθηγητής

Νέστωρ Ευμορφόπουλος
Επίκουρος Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 11η Οκτωβρίου 2018

.....	.....
Γεώργιος Σταμούλης Καθηγητής	Ν. Ευμορφόπουλος Επίκουρος Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών, του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

.....

Μηνάς Σπανόπουλος Καραλεξίδης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Πανεπιστημίου Θεσσαλίας

To my family and friends

Στην οικογένειά μου και τους φίλους μου

Ευχαριστίες

Με την περάτωση της παρούσας διπλωματικής εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες καθηγητές κ. Γεώργιο Σταμούλη και κ. Νικόλαο Ευμορφόπουλο για την εμπιστοσύνη τους στο πρόσωπο μου, με την ανάθεση αυτού του θέματος, καθώς και τη συνεργασία τους και την έκαστη βοήθειά τους όποτε την χρειάστηκα.

Παράλληλα θα ήθελα να ευχαριστήσω τους φίλους και συνεργάτες του εργαστηρίου Ε5 και ειδικά τον διδακτορικό φοιτητή κ. Δημήτριο Γαρυφάλλου, καθώς χωρίς την υπερπολύτιμη και αμέριστη βοήθειά του, η περάτωση αυτής της εργασίας θα ήταν τρομερά δύσκολη.

Τέλος, δε θα ήταν εύλογο να παραλείψω τους βασικούς και μόνιμους υποστηρικτές μου σε κάθε βήμα που κάνω, πόσο μάλλον σε όλα αυτά τα χρόνια των σπουδών μου. Ένα τεράστιο ευχαριστώ στην οικογένεια μου.

Μηνάς Σπανόπουλος Καραλεξίδης

Βόλος, 2018

Contents

List of Tables

List of Figures

Περίληψη

Η ανάλυση χρονισμού είναι μία από τις πιο διαδεδομένες διαδικασίες επαλήθευσης ενός κυκλώματος. Υπάρχουν πάρα πολλές μέθοδοι επαλήθευσης ενός κυκλώματος, μία από τις οποίες είναι η Στατική Ανάλυση Χρονισμού (STA). Αυτή η μέθοδος όχι μόνο είναι μία από τις πιο γνωστές τεχνικές, αλλά επιταχύνει επίσης τη διαδικασία της προσομοίωσης καθώς δίνει το δικαίωμα να γίνει σε στοχευμένα μονοπάτια και όχι σε ολόκληρο το κύκλωμα. Μόλις τα τελευταία χρόνια αναπτύχθηκε ο OpenTimer, ένα λογισμικό ανοιχτού κώδικα Στατικής Ανάλυση Χρονισμού. Ο OpenTimer είναι ένα σύγχρονο εργαλείο, που χρησιμοποιεί το Μη-Γραμμικό Μοντέλο Καθυστέρησης (NLDM) για να διεξάγει ανάλυση καθυστέρησης ολοκληρωμένων κυκλωμάτων.

Σκοπός αυτής της διπλωματικής είναι να επιτύχει καλύτερα αποτελέσματα, όσον αφορά την ακρίβεια του OpenTimer, με την ενσωμάτωση ενός ακριβέστερου μοντέλου, της Σύνθετης Πηγής Ρεύματος (CCS). Αυτό το μοντέλο αποτελείται από δύο επιμέρους μέρη-μοντέλα· το ένα είναι το μοντέλο οδηγού και το άλλο το μοντέλο αποδέκτη. Το μοντέλο οδηγού χαρακτηρίζεται από την καταγραφή της κυματομορφής ρεύματος στους πυκνωτές φορτίου εξόδου των πυλών. Η αντίστοιχη κυμματομορφή ρεύματος, εξαρτάται από τον χρόνο μετάβασης της εισόδου, το φορτίο εξόδου και τις καταστάσεις των υπόλοιπων εισόδων. Το μοντέλο αποδέκτη είναι σχεδόν αντίστοιχο με αυτό του NLDM, με τη διαφορά ότι προσθέτει περισσότερη ανάλυση για να αντικαροπτρίσει ευαισθησίες όπως το φαινόμενο του Μίλλερ. Για να λάβει υπόψη με ακρίβεια το φαινόμενο Miller στην χωρητικότητα εισόδου και στην καθυστέρηση καλωδίων, χωρίζει την χωρητικότητα που οδηγεί η πύλη σε δύο μέρη – τα C1 και C2.

Τα αποτελέσματα της παρούσας διπλωματικής δείχνουν πως η αντικατάσταση του μοντέλου καθυστέρησης NLDM με το CCS στον OpenTimer επιτυγχάνει 1.02% κατά μέσο όρο και μέχρι 1.7% το μέγιστο, ακριβέστερη εκτίμηση της καθυστέρησης των μονοπατιών του κυκλώματος συγκριτικά με τα αποτελέσματα που υπολογίστηκαν χρησιμοποιώντας ως σημείο αναφοράς την ανάλυση μέσω του εργαλείου PrimeTime της Synopsys.

Λέξεις Κλειδιά:
Στατική Ανάλυση Χρονισμού, Μοντέλα Καθυστέρησης, CCS, OpenTimer

Abstract

Timing simulation is one the most popular verification procedures of a design. There are many methods to put a circuit under testing, one of them is Static Timing Analysis (STA). This method not only is one of the well-known techniques, but also provides the ability to speed up the simulation by designating the simulation path, making that way possible to avoid simulation of the whole circuit. The first open-source STA tool called OpenTimer, was developed only recently. OpenTimer is a state-of-the-art timer, using the Non-Linear Delay Model (NLDM) to carry through with timing simulation.

In this thesis, the goal is to achieve better results, in the terms of accuracy, by integrating a more accurate timing model, Composite Current Course (CCS). This model consists of two aliquot model-parts; one is a driver model and the other is a receiver model. The driver one is characterized by capturing current waveform flowing into the load capacitor of the cell. It also has sensitivity to input transition time, output load and side input states. The receiver model is quite similar to the NLDM one, with the difference of adding additional granularity to reflect sensitivities such as miller capacitance. To accurately reveal the miller effect on input capacitance and net-delay, it is divided into two parts - C1 and C2.

The results of this thesis exhibit that the integration of CCS delay model into OpenTimer contributes to 1.02% on average and up to 1.7% accuracy enhancement, concerning the evaluation of the circuit's paths delays.

Keywords:
Static Timing Analysis, Delay Models, CCS, OpenTimer

Chapter 1
Introduction

1.1 Motivation

Timing simulation is one of the most commonly used methods in the industry in order to validate a circuit under evaluation. Static Timing Analysis (STA) comes to play a vital role in this time-consuming procedure, by using simplified timing models and by mostly ignoring logical interaction in circuits but yet, maintaining the accuracy and speedup in high levels. The need of higher accuracy, especially in sub-20nm technology nodes is rising exponentially, thus new models or even methods are necessary to be found. The opportunity to study graphs, build a parser and have a first touch with Static Timing Analysis was already from the beginning an inevitable offer. Furthermore, what could be better than the existence of a highly appreciated open-source timing tool, which allows experiment on any level someone wants and why not integrate their work, if it does worth. OpenTimer, this state-of-the-art tool, apart from the above, provides the opportunity to get used to employing advanced coding skills, as each new version utilizes many new features of C++.

1.2 Thesis Contribution

As for this thesis, its purpose was to study and attempt adding something extra into this timing tool, in the matter of STA. This addition refers to accuracy. OpenTimer already utilizes an accurate enough timing model named Non Linear Delay Model (NLDM). There is analysis on the model in a later chapter, as also for the timing model which has been chosen to replace it. This model is Composite Current Source (CCS) and it is going to be explained why it was chosen and what it offers. Another ambition is to encourage anyone who wants to experiment with OpenTimer and STA generally, regardless of any results.

1.3 Outline

Starting with Chapter 2, an adumbration of Static Timing analysis will take place, giving a general idea of what it is and why should be used. A description of how standard cells are shaped and how delay propagation is performed will be given, along with some STA tools. Moving on, a further analysis of the mentioned timing models will take place at Chapter 3. On Chapter 4, an introduction to OpenTimer will be given, depicting its structure and a part of its design flow. There will be also a representation of the NLDM timing model and its data structures in the tool. Finally, the 5th Chapter concerns the integration of CCS timing model into OpenTimer, describing how the corresponding data structures are formed and providing a sample of the timing engine routines along with our findings and results, in Chapter 6.

Chapter 2
Static Timing Analysis

2.1 What is Static Timing Analysis?

STA is one of the most commonly used techniques to validate the timing of a digital design, and for the purpose of this thesis of a Very-Large Scale Integration circuit (VLSI). Another timing verification method is the timing simulation which can not only verify the timing, but also the functionality of the design. With the term timing analysis, one can be referred to either of these methods. Thus, the use of the term alludes to timing issues of the design.

The STA is static, meaning that the analysis of the design does not depend upon the data values being applied at the input pins, while it is carried out statically. On the other hand, simulation-based timing analysis perform a loop, where a stimulus is applied on input signals and then the outcoming behavior is observed and verified, and finally time proceeds to the next step with new input stimulus and the procedure starts over.

The purpose of static timing analysis is to examine if the given design along with a set of input clock definitions and the definition of the external environment can operate at the rated speed. This validation measures how safely the design can operate at the specified frequencies of the clock without any timing violations. The basic functionality of static timing analysis is shown in figure 2.1. The design under analysis is called DUA. There are many timing checks, two of them are setup and hold checks. A setup check ensures that a flip-flop is provided with the data needed within the given clock period. A hold check ensures that a flip-flop captures the intended data correctly; meaning 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. Thus, these checks are performed in order to ensure that the proper data is ready and available for capture and latched in for the new state.

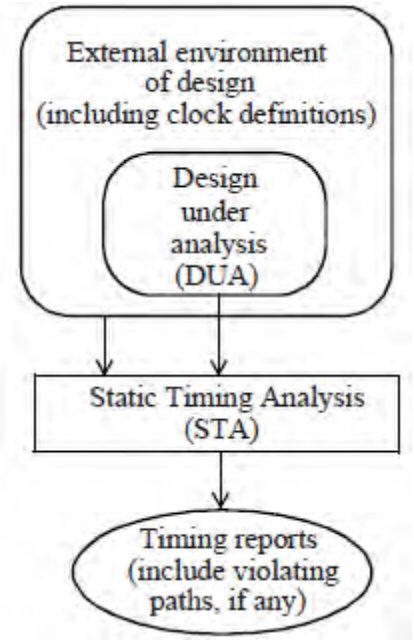

Figure 2.1: Static Timing Analysis

These required timing checks are performed for all possible paths and scenarios of the design, thus the entire design has to be analyzed. Having considered all these, it is easily understood that STA is a complete and exhaustive method for timing verification of a design.

The DUA is typically specified using a hardware description language such as VHDL or Verilog HDL. The external environment, including the clock definitions, is specified using SDC 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.

Given the opportunity, here is provided the first contact with STA for this thesis, which came from TAU 2016 Contest [1].

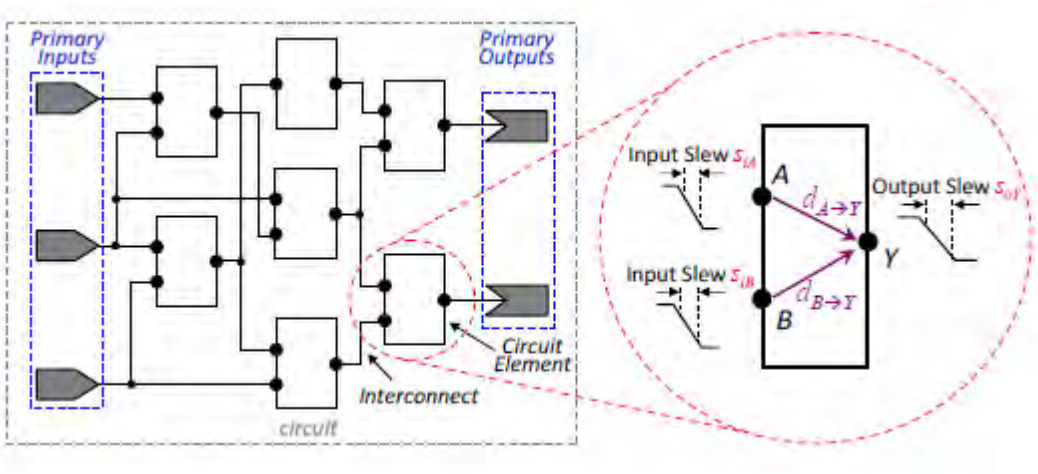

Figure 2.2: Generic circuit (left) and delay model representation of a circuit element (right).

Timing analysis computes the amount of time signals propagate in a circuit from its primary inputs (PIs) to its primary outputs (POs) through various circuit elements and interconnect. Signals at the output(s) of an element will be available via its inputs at some later time. This indicates a delay during signal propagation at each element. Furthermore, assume that signal transitions are characterized by their input slew and output slew, which is defined as the amount of time required for a signal too transition from hi-to-low and vice versa. For instance, as shown in Figure 2.2 (right), at this circuit element the delay from input A to output Y is labeled by $d_{A\to Y}$, the input slew at A by s_{iA}, and the output slew at Y by s_{oY}. Here, both the delay and the output slew are functions of input slew.

2.2 Why Static Timing Analysis?

As already said static timing analysis is a complete and exhaustive method of timing verification and all timing checks of a design. Simulation, which was previously introduced as an alternate analysis method, can only validate the portions of the design which get exercised by stimulus. Verification through timing simulation can be only as exhaustive as the test vectors used. Simulation and verification of all timing conditions of a design with 10-100 million gates is literally slow and the timing cannot be verified completely. Thus, exhaustive verification through timing simulation is very difficult and rare to happen.

On the other hand, static timing analysis as already said provides a quicker and simpler way of validating and analyzing all the possible timing paths in a design for any violations. Given the complexity of present day Application-Specific Integrated Circuit (ASICs), 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

Unfortunately, there is always an obstacle which needs to be overpassed; and one can be noise. Noise can limit the functionality and the performance of a design. It can occur due to crosstalk with other signals or due to noise on primary inputs or the power supply. Functional failures and frequency of operation limitation can occur due to noise. Thus, a design implementation has to be verified to be robust, meaning 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.

2.3 Standard Cells and Propagation Delay

2.3.1 Standard Cells

In a chip most of the complex functionality is usually designed utilizing basic building blocks, that implement simple logic functions such as and, or, not, nand, nor, and-or-invert, or-and-invert and flip-flop. These basic building blocks are pre-designed and referred to as standard cells [2]**.** These standard cells have an already pre-characterized functionality and timing which are available to the designer. Using the standard cells as the main building blocks, the designer then can implement the required functionality.

All digital CMOS cells are designed in such way that no current can be drawn from power supply (except for leakage) when the inputs are in a stable logic state. Therefore, the main reason of power dispersal is related to the activity in the design and is prompted by the charging and discharging of the inputs of CMOS cells in the design.

As for CMOS cells, logic-1 and logic-0 are considered with the following way; two values VIHmin and VILmax define the limits of the cell. Meaning that, any voltage value which surpasses VIHmin is considered as logic-1 and any voltage value below VILmax is considered as logic-0 respectively (Figure 2.3). Representative 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.

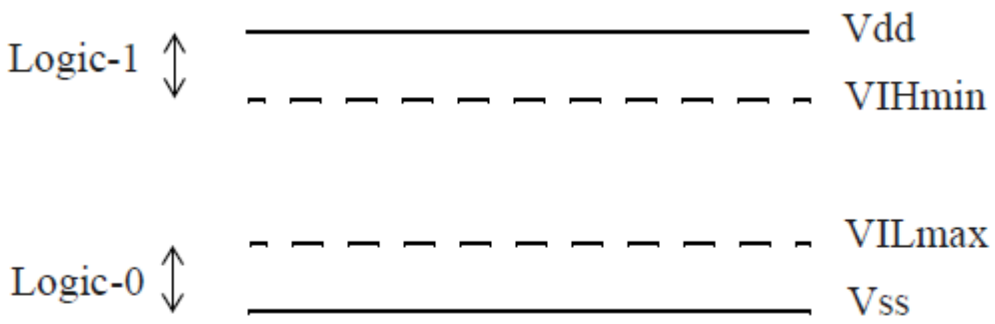

Figure 2.3: CMOS logic levels.

2.3.2 Propagation Delay

Considering a CMOS inverter cell as previously referred and its input and output waveforms, propagation delay of the cell is defined by using some measurement points on the switching waveforms. In order to define these points, the following four variables are used:

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

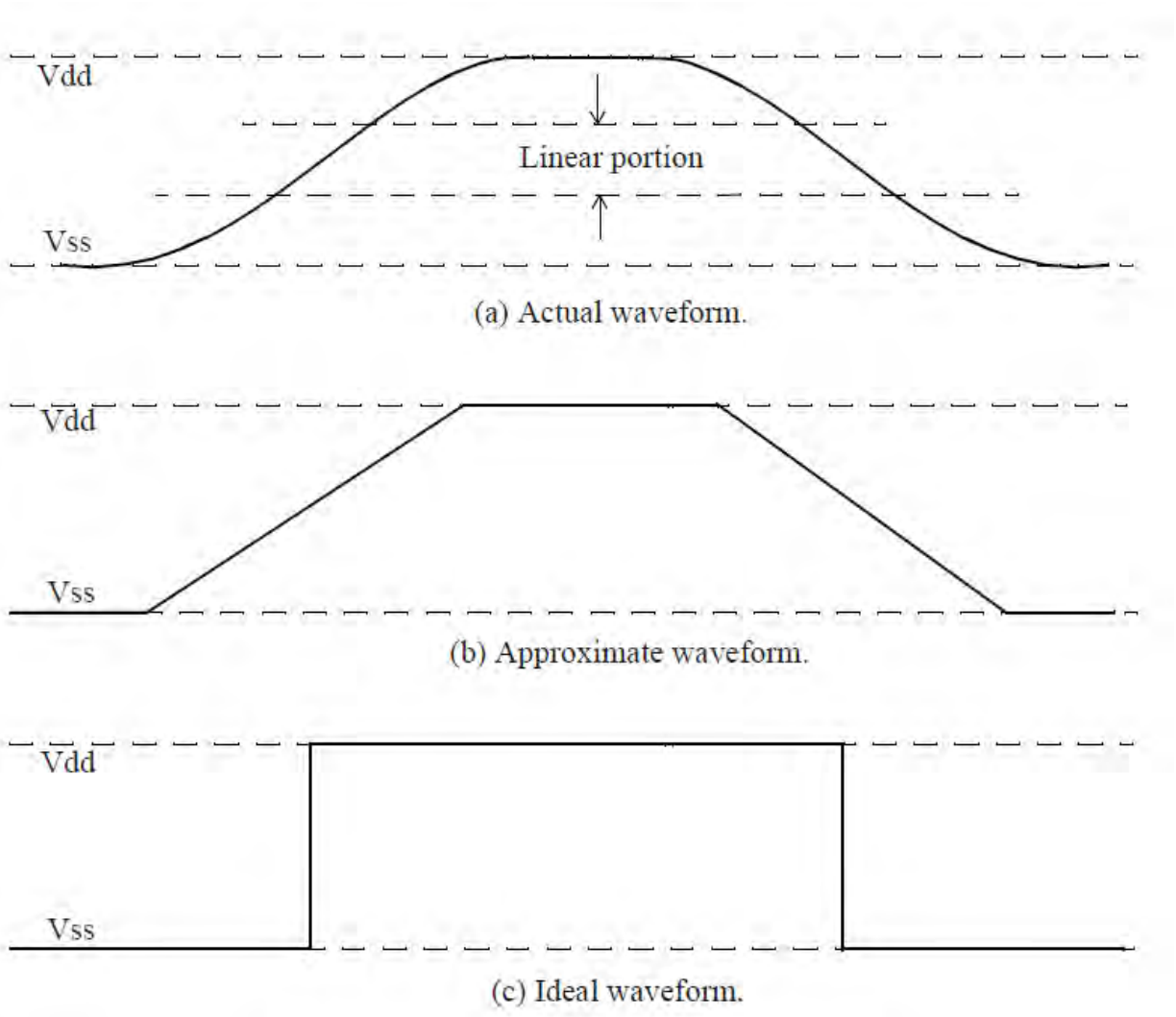

Figure 2.4: CMOS output waveforms.

These variables belong to a description command set of a cell library. When someone refers to these threshold specifications, they actually refer to the terms of percentage of Vdd, or the power supply. Usually, for most cell libraries, 50% threshold is used for delay measurement.

Rising edge is the transition from logic-0 to logic-1 and falling edge is the transition from logic-1 to logic-0 respectively.

Consider the example inverter cell and the waveforms at its pins shown in Figure 2.5. The propagation delays are represented as:

i. Output fall delay (Tf)
ii. Output rise delay (Tr)

Generally, these two values are different. Figure 2.5 illustrates how these two propagation delays are measured.

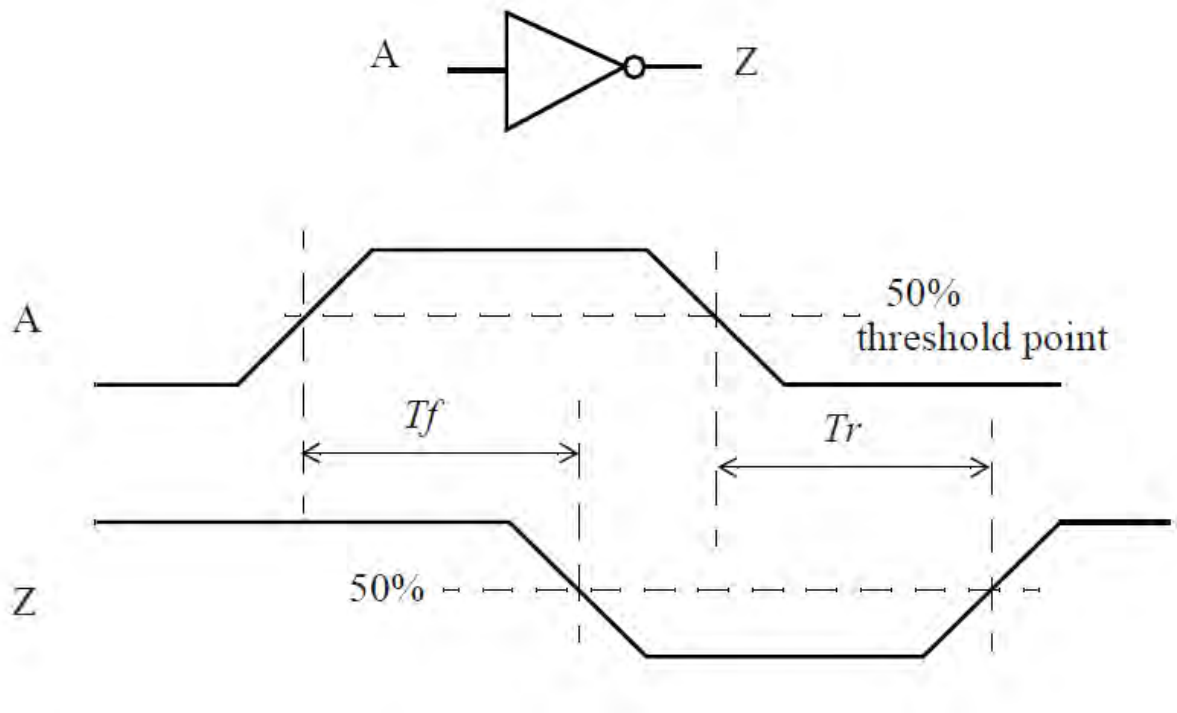

Figure 2.5: Propagation delays.

Ideally, propagation delay would simply be the delay between the two edges. This could happen only if we had ideal waveforms. A situation like this is shown if figure 2.6.

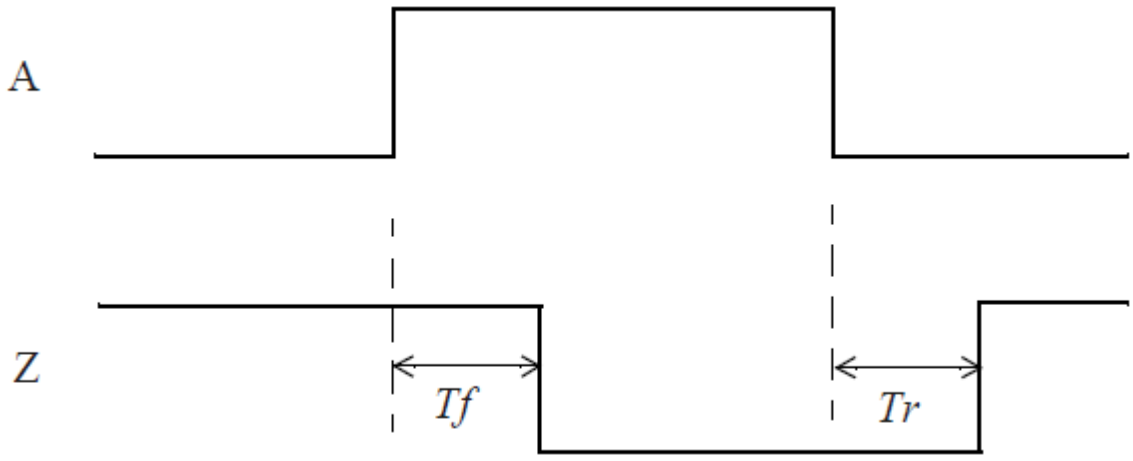

Figure 2.6: Propagation delay using ideal waveforms.

2.4 Static Timing Analysis Tools

This section of the chapter presents some of the well-known and commonly used STA tools. Two of them, that are going to be discussed, are Synopsys PrimeTime and Cadence Tempus. The tool which this thesis employs is OpenTimer, but it is going to be presented on another chapter.

2.4.1 PrimeTime

Synopsys PrimeTime is one of the famous tools and probably one of the most accurate. The simple version can perform core static timing analysis and multi-scenario analysis. Adding some extra versions like PrimeTime SI or PrimeTime PX one can carry out crosstalk delay and signal integrity analysis or even dynamic power analysis respectively and many other analyses (see Figure 2.7).

Some of its primary benefits are that provides accurate results and minimizes over-design; also the high capacity approach reduces hardware costs, the integrated design environment improves productivity and many more.

PrimeTime STA solution as Synopsys report on their datasheet, "provides designers with extensive timing analysis checks, on-chip variation analysis techniques, golden delay calculation, advanced modeling, unmatched productivity and ease-of-use" [3] and a user-friendly GUI.

PrimeTime's basic flow is the following:

1. Set library path (search and link path)
2. Read the design (Verilog)
3. Link library and the design
4. Add design constraints (Rise/Fall time, gate delay)
5. Add constant value to input port (for timing simulation)
6. Report (constraints and timing)

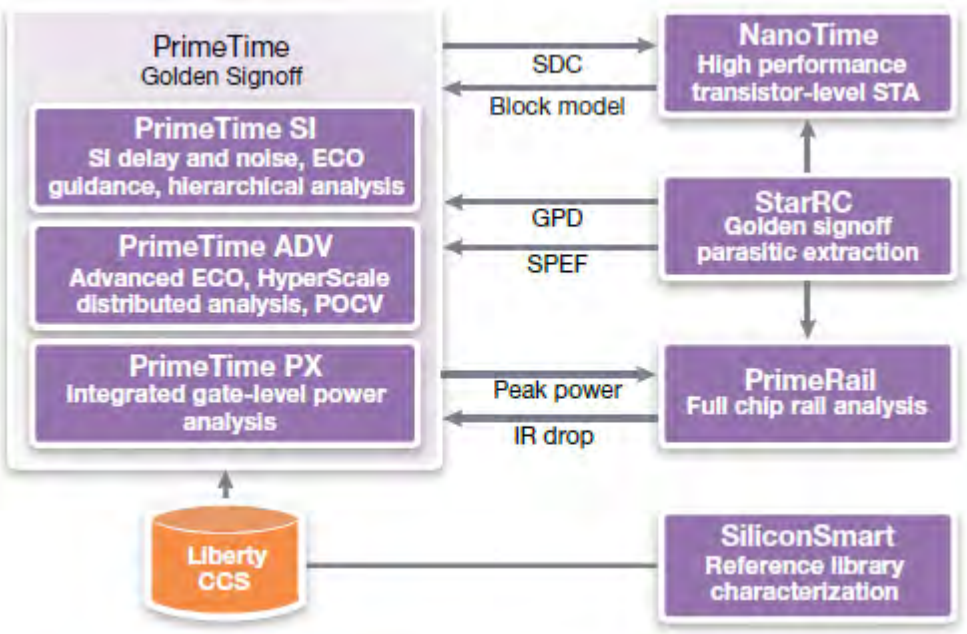

Figure 2.7: Galaxy Signoff Solutions.

2.4.2 Tempus

Cadence Tempus Timing Signoff Solution is also a well-known and accurate tool. It distributes silicon-accurate timing signoff and signal integrity analysis which ensures operational chips after tapeout. With the timing signoff environment combined with the implementation environment, the Tempus solution greatly decreases the time to design closure and improves timing convergence throughout the design flow [4].

The industry uses the ILMs (Interface Logic Models) [5] and with Tempus they provide a novel way of automatically breaking the design into semi-autonomous cones of logic each of which could be run on different threads (MTTA – multi-threaded timing analysis) and across multiple machines (DSTA – distributed static timing analysis). As part of this, methods for inter-client communications have been worked out which enabled the tool to pass vital information such as timing windows between associated cones of logic.

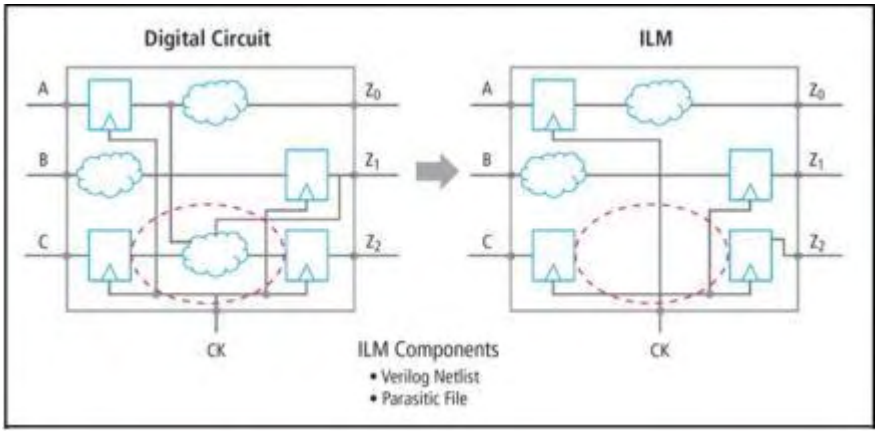

Figure 2.8: Interface Logic Model (ILM) concept.

Tempus speed is quite amazing and also allows someone to effectively run blocks of up to 40 million cells in a single client, distributing that it can handle huge designs. Furthermore, once Tempus could automatically identify cones of logic which were dependent upon each other for accurate timing analysis, it was also realized that the inverse was also true. Tempus can recognize which blocks can safely be ignored for any selected block that is to be timed; meaning that Tempus can automatically carve out just enough logic around a selected block to ensure an accurate analysis without having to time the entire netlist.

An example flow is firstly the building of the blocks, then the pass back to the top-level for assembly and routing. Once context is set, blocks could then be passed back down for final timing optimization. In conclusion, one of its primary benefits is that the same timing scripts, constraints and use-model for flat timing analysis can be used for top-level and block-level optimizations and scope-based analysis can be run in parallel either by multiple designers or through Tempus distributed processing.

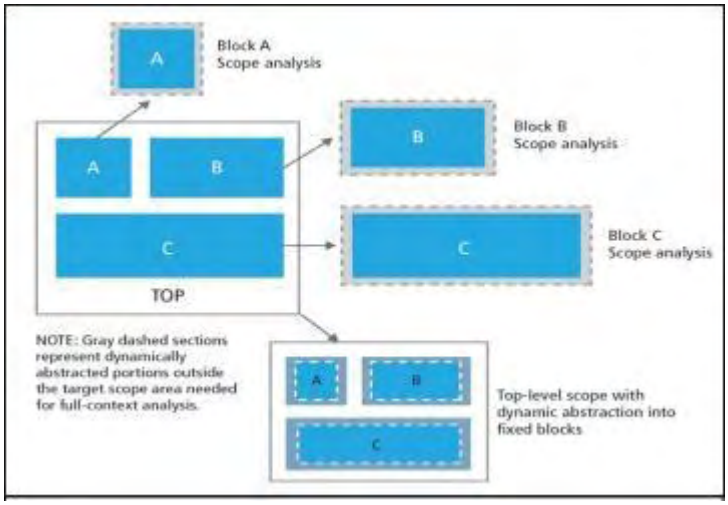

Figure 2.9: Timing scope can be used for individual blocks or top-level distributed STA.

Chapter 3
Timing Models

3.1 Gate and Interconnect Modeling

3.1.1 Interconnect Modeling

The word interconnect is used in order to describe wires used in a design, which basic instance is a net. Nets usually have an input pin which is called port and at least one output pin, called taps. Parasitic RC trees only contain grounded capacitors and floating resistors.

Delay. Electrical simulation can be an accurate tool for computation of port-to-tap delays. Nonetheless, Elmore delay model [6] will be assumed because of its simplicity and speed, where the delay is approximated by the symmetric of the value of the first moment of the impulse response. A summary of the topological method which is used in order to compute the delay of RC tree networks is provided in [7].

Consider any two nodes e and k in an RC network. Let C_k be the lumped capacitance at node k, and let $R_{k\to e}$ be the total resistance of the common path intermediate to the paths from Port to e and Port to k. For instance, the resistance between nodes 1 and T_2 ($R_{1\to T2}$), in Figure 2.1, is R_A, as that is the only common resistor among the paths Z to 1 and Z to T_2. The Elmore delay at node e is the following:

$$d_e = \sum_{k\in N} Rk \to eCk \tag{1}$$

where N is the set of all nodes in the RC network. For the example net illustrated in Figure 2.1 (right), the delay at node T_2 (tap) is (visiting in order nodes 1, T_1, 3, 2, T_2):

$$d_T2 = R_AC_1 + R_AC_3 + R_AC_4 + (R_A + R_B)C_2 + (R_A + R_B + R_E)C_5 \tag{2}$$

$$= R_A(C_1 + C_3 + C_4) + (R_A + R_B)C_2 + (R_A + R_B + R_E)C_5$$

Output slew. The value of the output slew (s_o) can be approximated on any given tap node T by a two-step process. The first step is to calculate the output slew of the impulse response on T, which was observed [6],[8] to be well-approximated by:

$$\hat{S}_{oT} \approx \sqrt{2\beta T - dT2} \tag{3}$$

where β_T is the second moment of the input response at node T and d_T is the corresponding Elmore delay from equation 1. The second step is to calculate the slew of the response to the input ramp by the expression given in [9]:

$$s_{oT} \approx \sqrt{si2 + \hat{s}2oT} \tag{4}$$

where s_i is the input slew.

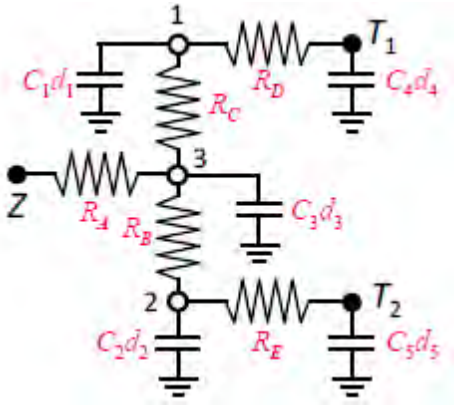

Figure 3.1: Modified RC network for output slew calculation.

The efficient path-tracing algorithm for moment computation suggested in [10], which is a generalization of the algorithm proposed in [6], can be the expedient to compute the value of β_T. To calculate β_T, two steps are required; the first step is to replace all capacitance values C_k in the RC network by $C_k d_k$, where d_k is the Elmore delay from Equation (1) (see Figure 3.1). The second step, is to follow the identical procedure as before in order to find β_T:

$$\beta_T = \sum_{k \in N} R_{k \to T} C_k d_k \tag{5}$$

Following the example in Figure 3.1, at node T_2

$$\beta_{T2} = R_A(C_1d_1 + C_3d_3 + C_4d_4) + (R_A + R_B)C_2d_2 + (R_A + R_B + R_E)C_5d_5 \tag{6}$$

3.1.2 Circuit Element Modeling

As for delay and output slew calculations between two pins, the information will be given as two-dimensional tables in the **.lib** file. Extrapolation or interpolation will be essential in order to find the equivalent timing information.

If the table contains a single value, i.e., a 1x1 table (Figure 3.2 left), no interpolation is needed. In other word, regardless of input x and y, the corresponding value is constant. If the table is one-dimensional, i.e., a 1xn table or a mx1 table (Figure 3.2 center), then the value will depend only on the non-scalar dimension. For instance, consider the 1x4 table in Figure 4. The methodology is the following. If $y < y_1$, then the corresponding output z value will be the linear extrapolation between z_1 and z_2. If $y_2 \leq y \leq y_3$, then z will be the linear interpolation between z_2 and z_3. If $y_4 < y$, then z will be the linear extrapolation between z_3 and z_4.

$$z_1 - (y_1 - y)\frac{z2-z1}{y2-y1} \text{ if } y < y_1 \tag{7}$$

$$z_1 \text{ if } y = y_1 \tag{8}$$

$$z_1 + (y - y_1)\frac{z2-z1}{y2-y1} \text{ if } y_1 < y < y_2 \tag{9}$$

$$z_2 \text{ if } y = y_2 \tag{10}$$

$$z_2 + (y - y_2)\frac{z3-z2}{y3-y2} \text{ if } y_2 < y < y_3 \tag{11}$$

$$z_3 \text{ if } y = y_3 \tag{12}$$

$$z_3 + (y - y_3)\frac{z4-z3}{y4-y3} \text{ if } y_3 < y < y_4 \tag{13}$$

$$z_4 \text{ if } y = y_4 \tag{14}$$

$$z_4 + (y - y_4)\frac{z4-z3}{y4-y3} \text{ if } y > y_4 \tag{15}$$

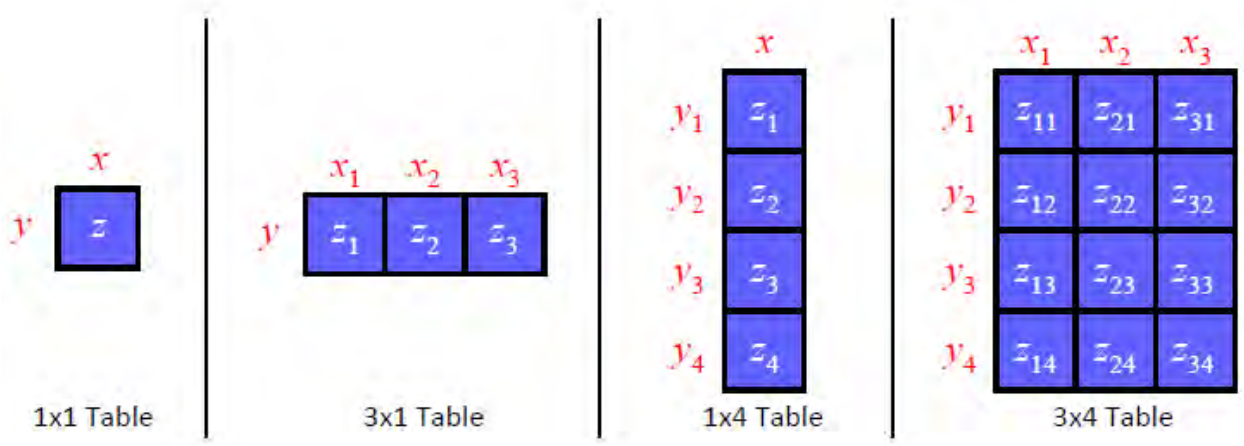

Figure 3.2: Illustration of different tables: scalar, one-dimensional and two-dimensional.

In case that the table is two-dimensional, perform linear interpolation on the x values first, and then perform interpolation on y values. For example again, take into account the 3x4 table in Figure 4. If $x_2 < x < x_3$ and $y_2 < y < y_3$, then (*i*) determine z_{first} by linear interpolation on z_{22} and z_{32}, (*ii*) determine z_{second} by linear interpolation on z_{23} and z_{33}, and then (*iii*) determine z by linear interpolation using z_{first} and z_{second}.

Combinational elements. Given any combinational cell, e.g., OR gate, let the delay d and output slew s_o for a input/output pin-pair (see Figure 3.3) be calculated by NLDM model interpolation/extrapolation. These delay and output slew are referenced by the input slew (x) and driving load (y) and are stored in the **.lib**. C_L denotes the equivalent downstream capacitance seen from the output pin of the cell. A few intellectually complex models have been suggested for computing C_L. For simplicity, the application of a simpler model is adopted in terms of this thesis. C_L is assumed to be the sum of all the capacitances in the parasitic RC trees, containing the cell pin capacitances at the taps of the interconnect network.

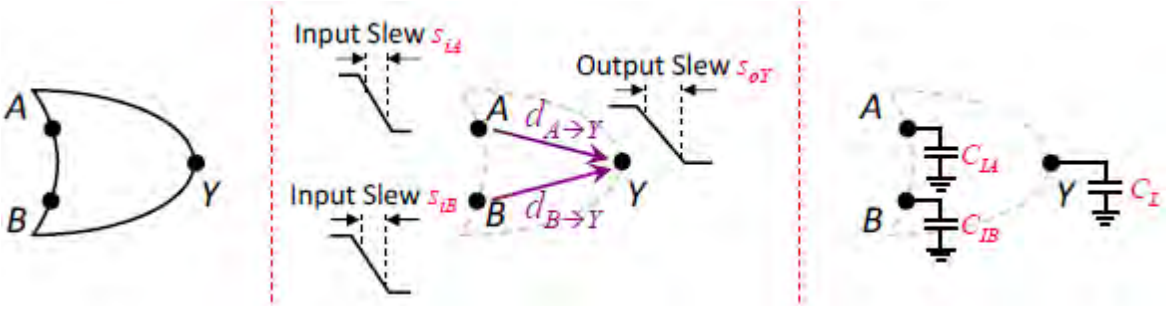

Figure 3.3: Combinational OR gate (left), its timing model (center) and capacitances (right).

Sequential elements. Sequential circuits consist of combinational blocks interleaved by *registers*, most frequently implemented with flip-flops (FFs). Usually, sequential circuits are composed of several stages, where a register captures data from the outputs of a combinational block from a previous stage, and injects it into the inputs of the combinational block in the next stage. *Clock signals* generated by one or multiple clock sources are used to achieve register operation synchronization. Clock signals that reach distinct flip-flops, e.g., sinks in the clock tree, are delayed from the clock source by a *clock latency l*.

A (D) flip-flop is a storage element that captures a given logic value at its input data pin *D*, when a given clock edge is detected at its clock pin *CK*, and subsequently presents the captured value and its complement at the output pins *Q* and $\acute{Q}$. The flip-flop also enables asynchronous preset (set) and clear (reset) of the output pins through the respective *S* and *R* input pins.

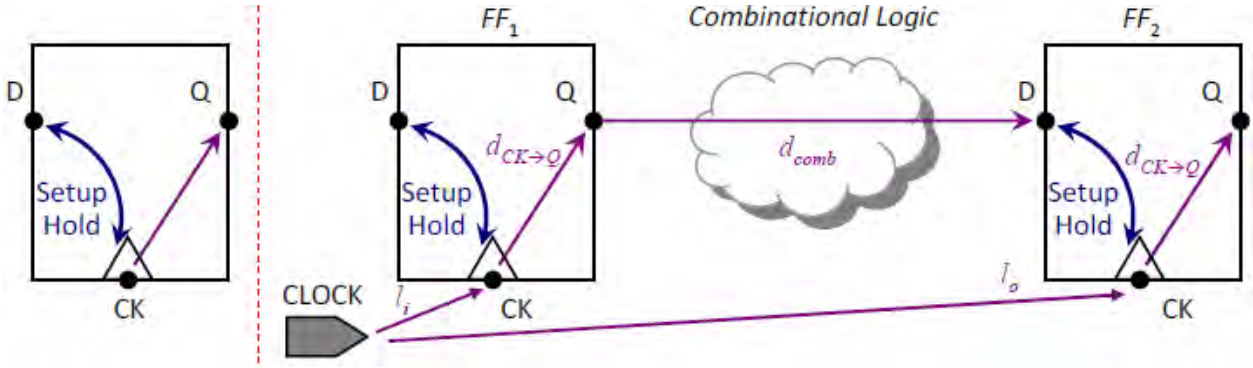

Figure 3.4: Generic D flip-flop and its timing model (left), and two FFs in series and their timing models (right).

Setup and hold constraints. The logic value of the input data pin is required to be stable for a specific period of time *before* the capturing clock edge in order to achieve proper operation of a flip-flop. This period of time is designated by the *setup time* t_{setup}. Furthermore, the logic value of the input data pin must also be stable for a specific period of time *after* the capturing clock edge. This period of time is designated by the *hold time* t_{hold}. The flip-flop timing models are depicted in Figure 3.4 (left). The test time are given in the **.lib** as two-dimensional tables, and are referenced by the clock-side input slew (*x*) and the data-side input slew (*y*).

Signal propagation. Figure 3.4 (right) illustrates how the standard signal transition between two flip-flops is performed. Supposing that the clock edge is generated at the source at time 0, it will reach the injecting (launching) flip-flop FF_1 at time l_i, making the data available at the input of the combinational block $d_{CK\to Q}$ time late. Assuming that the propagation delay in the combinational block is d_{comb}, then the data will be available at the input of the capturing flip-flop FF_2 at time $l_i + d_{CK\to Q} + d_{comb}$. Let the clock period to be a

constant T. Then the next clock edge will reach FF_2 at time $T + l_o$. For correct operation, the data have to be available at the input pin D of FF_2 t_{setup} time before the next clock edge. As a consequence, at the data input pin D of FF_2 applies the following:

$$at_D^{late} = l_i^{late} + d_{CK \to Q} + d_{comb}^{late} \tag{16}$$

$$rat_{setup} = rat_D^{late} = T + l_o^{early} - t_{setup} \tag{17}$$

A similar state can be derived for ensuring that the hold time is respected. The data input pin D of FF_2 must remain stable for at least t_{hold} time after the clock edge reaches the corresponding CK pin. Therefore, the following applies to the data input pin D of FF_2:

$$at_D^{early} = l_i^{early} + d_{CK \to Q} + d_{comb}^{early} \tag{18}$$

$$rat_{hold} = rat_D^{early} = l_o^{late} + t_{hold} \tag{19}$$

Note that when computing the required arrival times in equations (17) and (19) the value l_o is exact to Figure 3.4. In the general case, l_o should be replaced with at_C. The previous arrival times and required arrival times induce setup and hold slacks, which can be computed from the following equations

$$slack^{early} = at^{early} - rat^{early} \tag{20}$$

$$slack^{late} = rat^{late} - at^{late} \tag{21}$$

For the clock pins of the flip-flop, the required arrival time is derived from the test slack. For early mode, the slack at the clock pin is the setup or late test slack, and for late mode, the slack at the clock pin is the hold or early test slack. From the corresponding test slack and arrival time, the clock required arrival time can be derived, and suitably propagated.

3.2 Non-Linear Delay Model (NLDM)

Table models are included by most of the cell libraries in order 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.). The table models are referred to as NLDM 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 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. An example of such a table for a typical inverter cell is following:

```
pin (OUT) {
    max_trinsition : 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");
        }
    }
}
```

The delays of the output pin *OUT* are described in the above example. This portion of the cell description contains the rising and falling delay models for the timing arc from pin *INP1* to pin *OUT*, in addition to the *max_transition* allowed time at pin *OUT*. The labels *cell_rise* and *cell_fall* describe the separate models for the rise and fall delays (for the output pin) respectively. The type of indices and the order of table lookup indices are described in the lookup table (LUT) 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 LUT 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 of this LUT template, 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 template as a function of two variables, transition time and capacitance, is called the NLDM, 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 counterbalancing outputs, *Q* and *QN*.

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. Therefore, there are distinct 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*. The transition times are measured based on the specific slew thresholds, usually 10%-90% of the power supply.

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

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

Assuming a cell with these characteristics and given the input time and output capacitance, as illustrated in Figure 3.5, the rise delay is acquired from the *cell_rise* table for 15ps input transition time (falling) and 10fF load, and the fall delay is acquired from the *cell_fall* table for 20ps input transition time (rising) and 10fF load.

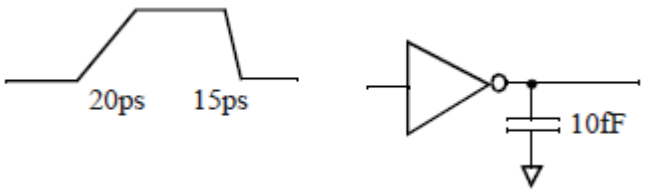

Figure 3.5: Transition time and capacitance for computing cell delays.

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

3.2.1 Example of Non-Linear Delay Model Lookup

In this section an illustration of the lookup of the table models takes place 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 matches directly to the value in the table. The example below corresponds to a general case where the lookup does not match to any of the entries available in the table. In such cases, two-dimensional interpolation is utilized to supply the resulting timing value. For the table interpolation, the two nearest table indices are chosen. 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 *intex_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:

$$T_{00} = 0.75 * 0.25 * 0.1937 + 0.75 * 0.75 * 0.7280 + 0.25 * 0.25 * 0.2327 + 0.25 * 0.72 * 0.7676 = 0.6043$$

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:

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

3.3 Composite Current Source Timing Model

3.3.1 Introduction

Delay calculation is performed for one stage at a time, where a stage consists of the driving cell arc, the output RC network and the capacitance of the network load pins. The goal is to compute the response at the driver output and at the network load pins, given an input slew or waveform at the driver input, as shown in Figure 3.6. The computed responses are then used to determine the cell delay of the driver and the input slews at the load pins.

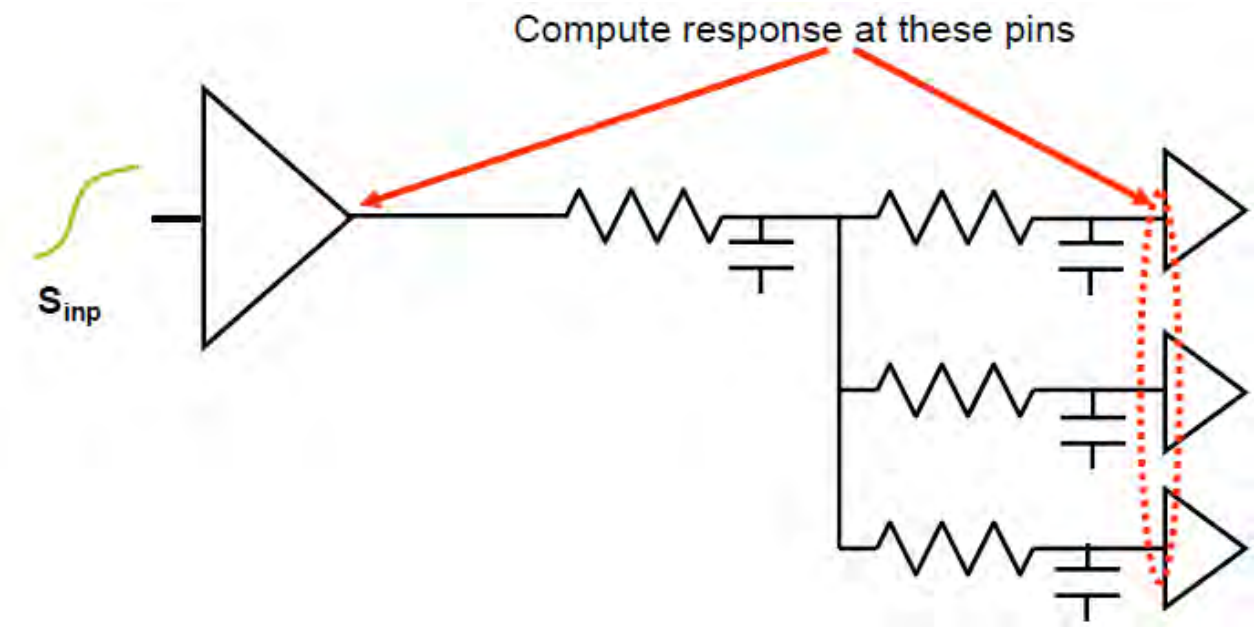

Figure 3.6: Stage Delay Calculation.

To perform stage delay calculation efficiently, three models are created:

- The driving cell arc is replaced by a driver model
- The interconnect RC network is replaced by a reduced order model (such as Block Arnoldi)
- The load pins are replaced by a receiver model

These models are depicted in Figure 3.7.

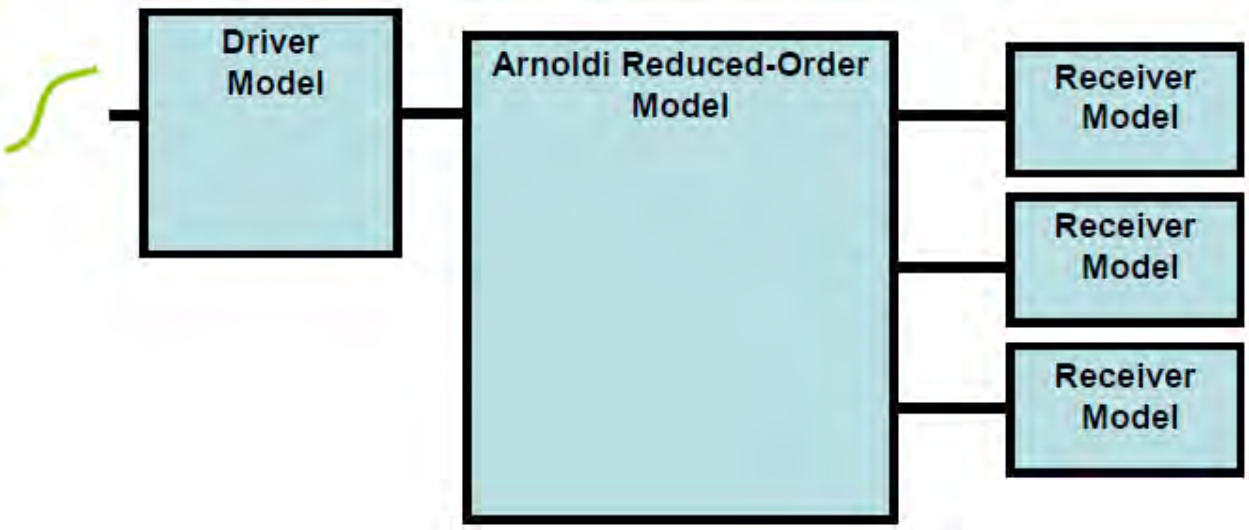

Figure 3.7: Stage represented by driver model, ROM and receiver models.

Note that the receiver model must represent the complex input capacitance of a cell input pin. The transistors do not present a constant input capacitance to a driver. The equivalent capacitive load (from I = C * dv/dt) can vary depending on the rise/fall direction of the transition, the input slew at the pin, the output load, and the state of the cell. In addition, this capacitance can change during the transition. The receiver model must be able to represent all these effects.

3.3.2 Previous Approaches

Thevenin and Norton Models

Previous driver models used either a time-dependent voltage source in a series with a resistor (Thevenin model) or a time-dependent current source in parallel with a resistor (Norton model). The resistor in those models is typically referred to as the "drive resistor" and is used to express the timing arc's sensitivity to output capacitance, whereas the waveform shape itself is primarily expressed by the voltage or current source.

Refinements to these models to account for complex aspects of transistor behavior have typically dealt with making the time-dependent nature of the voltage/current source more complex. Other approaches have dealt with multiple drive resistances and arbitrary dynamic impedances.

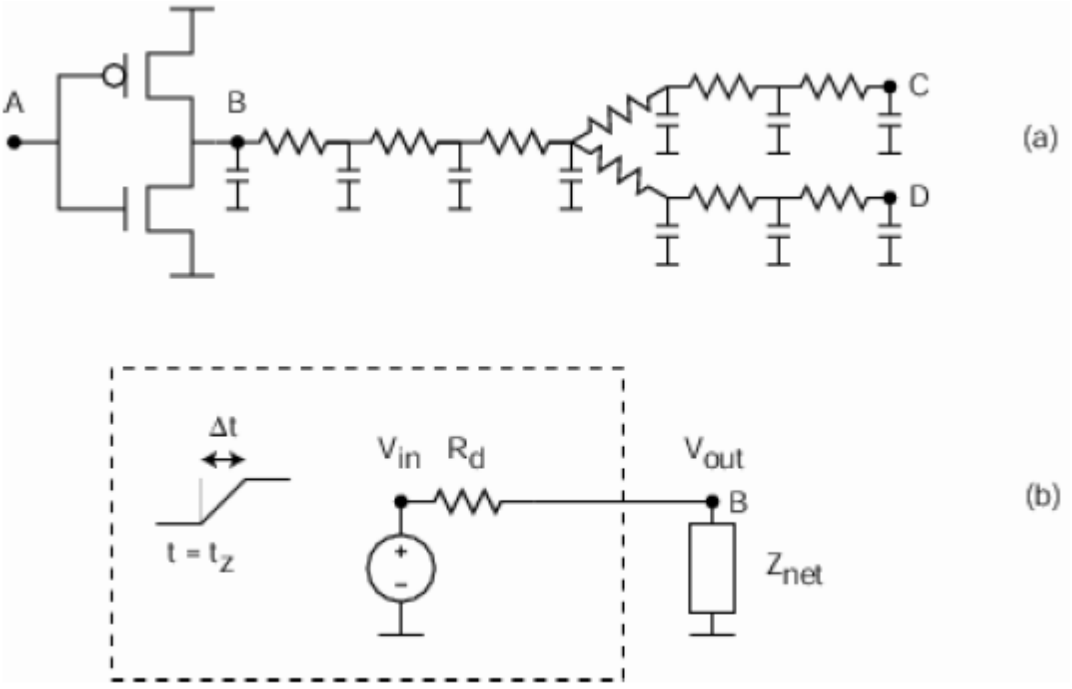

Figure 3.8: : The Rd << Znet problem. (a) shows a transistor circuit driving a detailed parasitic network at node 'B'. (b) The network presents an impedance Znet to the Thevenin driver model. When Rd << Znet, Vout approaches Vin and the driver model can lose accuracy.

Unfortunately, a major limitation occurs when conventional models are used to drive an interconnect network with an impedance Z_{net} much greater than the drive resistance R_d. Consider the Thevenin model driving a detailed parasitic network as shown in Figure 3.8. Note that the circuit forms a voltage divider with

$$\frac{Vout}{Vin} = \frac{Znet}{Rd+Znet}$$

which approaches unity when $R_d << Z_{net}$.This points out that a driver model based upon a drive resistance (or arbitrary impedance) that is set independent of the network load will be ineffective in this regime.

Since the transistor behavior deviates from the Thevenin voltage source nearest the power rails, this situation is usually worst when the network delay is greater than the output transition time.

Previous Receiver Models

The traditional receiver model is a single value of capacitance for an input pin. More recently, separate values have been allowed for rising vs. falling transitions, and a min/max range has been introduced that can bind the complex capacitance effects, but which leads to pessimism during analysis.
Using a single capacitance value for the entire transition results in inaccuracy for single-stage cells where the Miller effect is significant, affecting the calculation of both cell delay and slew. Figure 3.9 shows that the voltage waveform for the input of a single-stage cell, such as an inverter, cannot be approximated well by any single capacitance value.

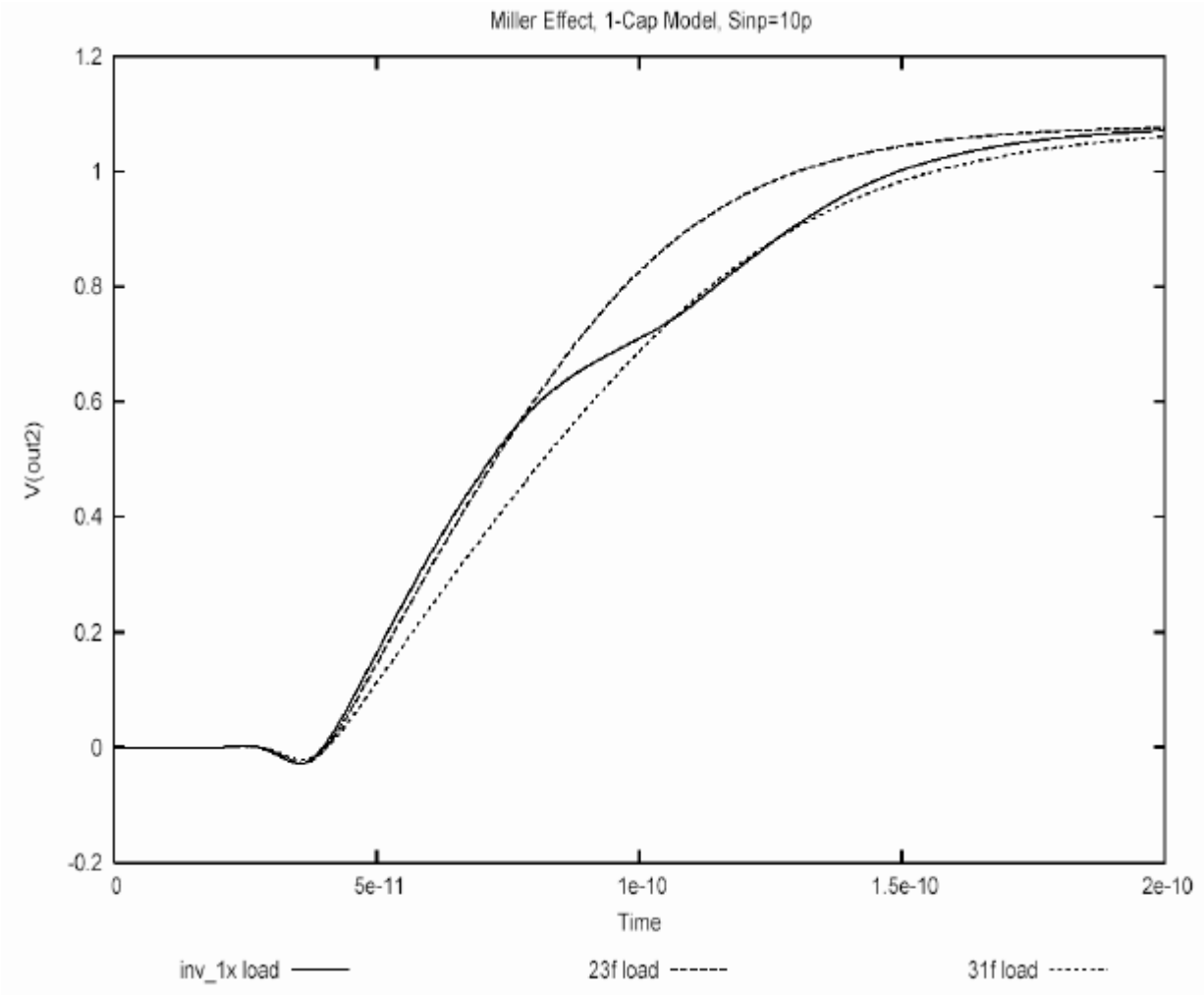

Figure 3.9: A single capacitance value is insufficient when Miller effect is large.

3.3.3 The CCS Timing Solution

CCS Timing consists of a driver model and a receiver model. The driver model describes how a timing arc propagates a transition from input to output, and how it can drive arbitrary RC networks. The receiver model describes the capacitance that an input pin presents to driving cells.

The CCS Timing driver model is a time and voltage dependent current source with an essentially infinite drive-resistance, which provides high accuracy even when R_d is much less than Z_{net}. The model achieves this accuracy by mapping the arbitrary transistor behavior for lumped loads to the behavior for an arbitrary detailed parasitic network, instead of modeling the transistor behavior. The following figure illustrates how the mapping algorithm basically works.

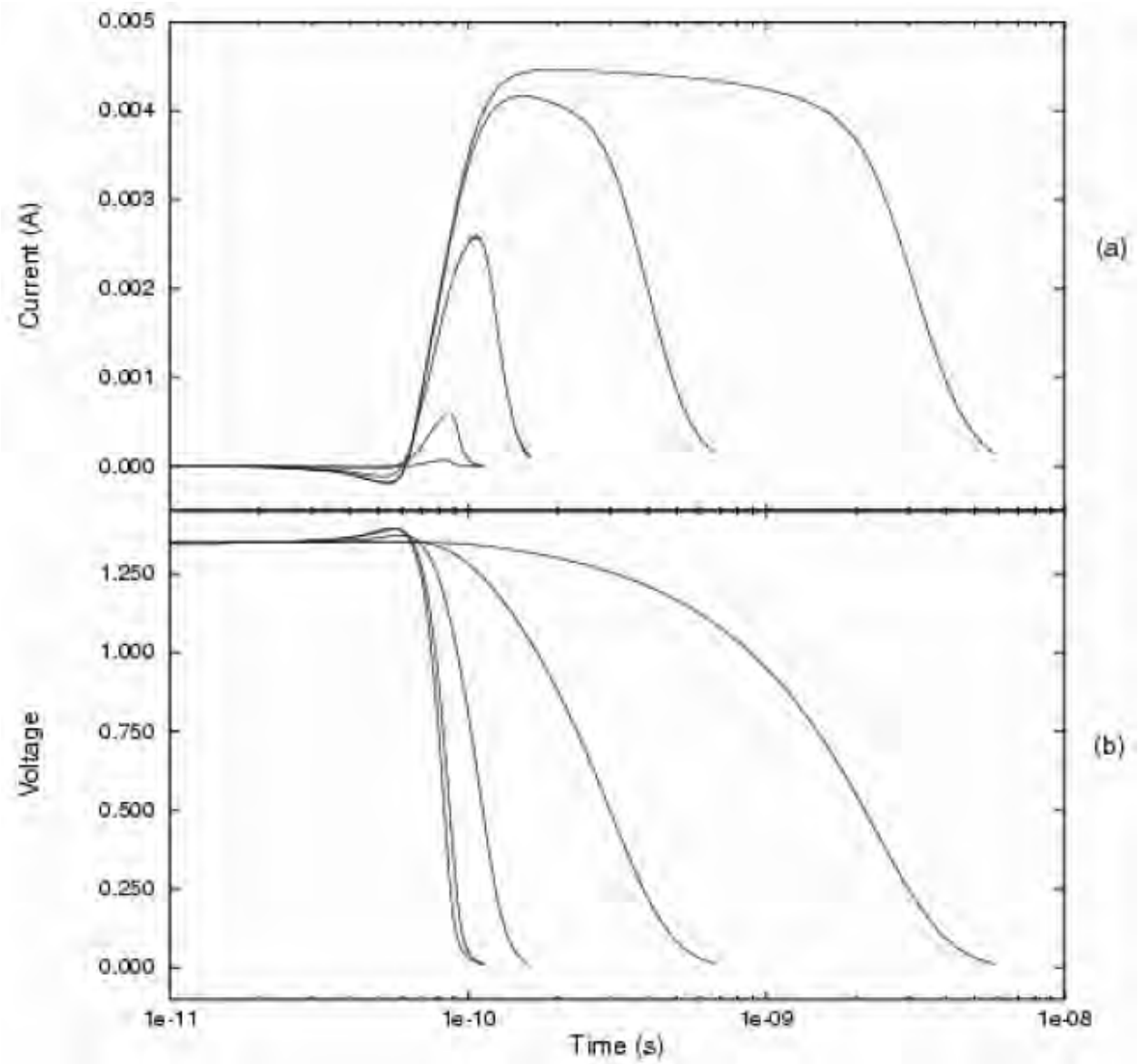

Figure 3.10: Output current and voltage responses for a timing arc. Transition-level simulation results are shown for different values of load capacitance (1fF, 10fF, 100fF, 1pF, 10pF). (a) Inverted current responses. (b) Voltage responses.

Consider a set of pre-characterization measurements of the output current as a function of time for a specific input slew and a set of output capacitances (Figure 3.10). When these currents are applied to their respective capacitances, the voltage waveforms can be reconstructed. If an output capacitance is presented, which was not pre-characterized with, then interpolation can be performed between the currents to predict the resulting waveform. Similarly, if an input slew is presented, that was not used for pre-characterizing, interpolation can also take place.

Now consider driving a detailed parasitic network. The output currents from pre-characterization can be applied to the network at a given timestep. There will be a unique

current that will elicit the same voltage on both a lumped capacitance and the network at the given timestep. This current is the chosen value for the given timestep, and this procedure is reapplied at every subsequent timestep. In other words, there is nothing more than application of $I_{out}(V_{out})(t)$.

CCS Timing delay calculation uses advanced interpolation technology to determine a current waveform when the input slew and/or output load values do not match those used during cell characterization. Additionally, interpolation is used for intermediate values of V_{dd} and temperature by using data from multiple libraries.

3.3.4 Characterization for CCS Timing

Characterizing a cell timing for CCS Timing is very similar to characterization for NLDM: an input stimulus is chosen to produce a specific input slew time (S_{inp}); a load capacitance (C_{out}) is connected to the output pin; and a circuit simulation is run in the same way as NLDM. But instead of measuring voltage thresholds at the output pin, current is measured through the load capacitor and into the input pin. The current through C_{out} is used for the driver model, and the current into the input pin is used to determine the receiver model.

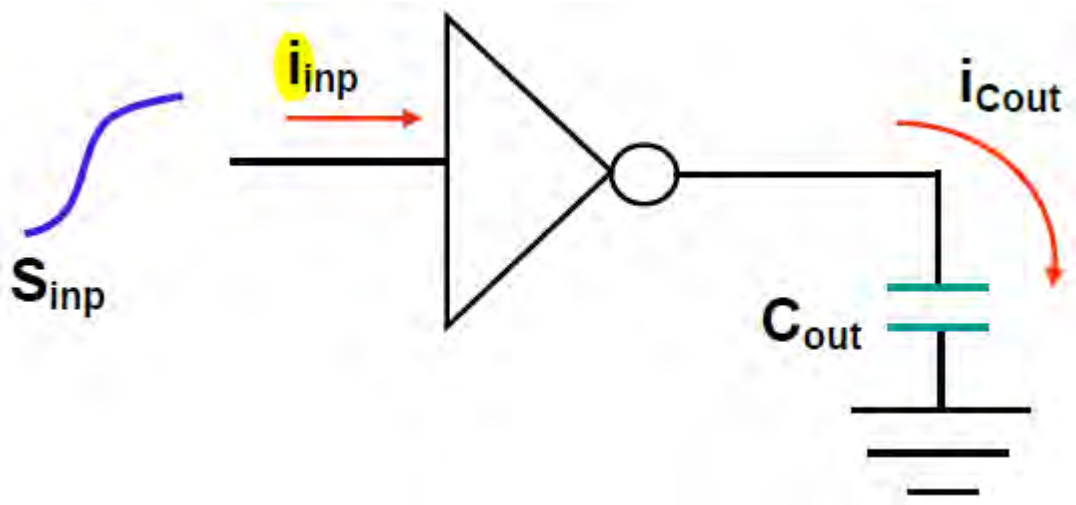

Figure 3.11: CCS Timing characterization measurements.

These characterization experiments are repeated for a table of different S_{inp} and C_{out} combinations. The current through C_{out} is saved for every circuit simulation timestep and then reduced to a much smaller set of current and time (i, t) points. The points are chosen such that $V_{out}(t)$ can be accurately reproduced for every timestep during the transition. Figure 3.12 provides an example of the complete $i(t)$ waveform and a reduced set of points.

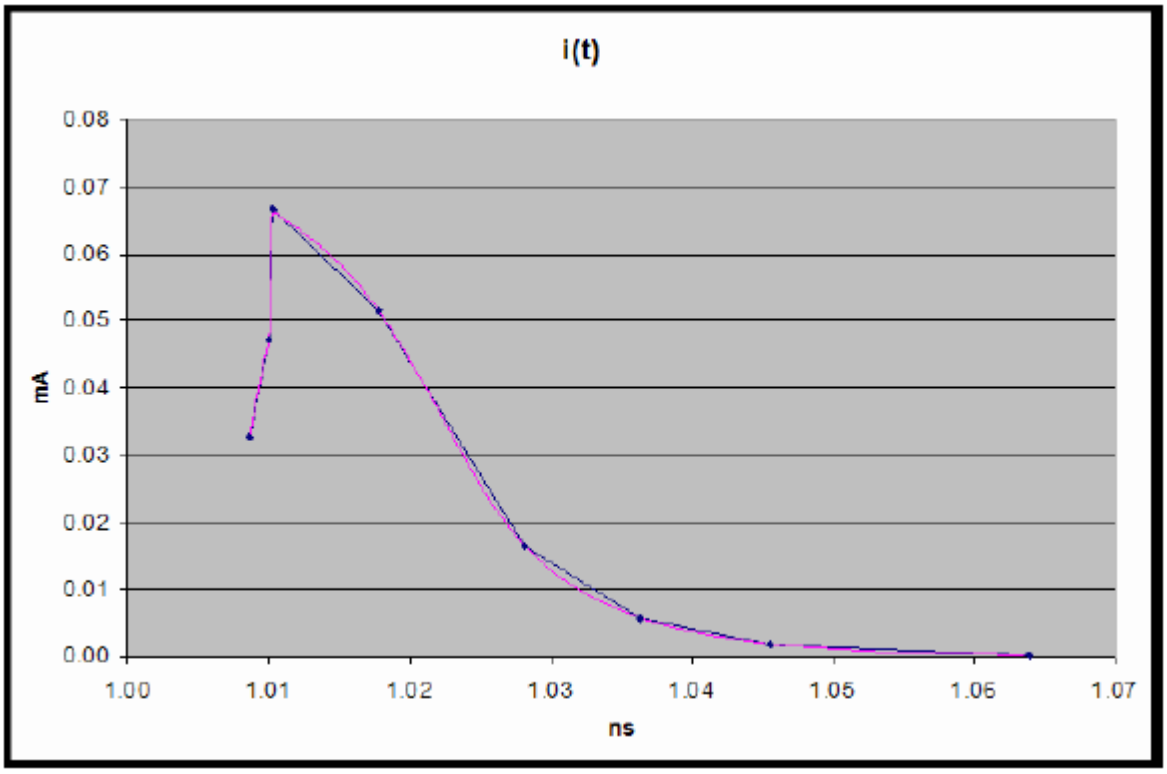

Figure 3.12: Current waveform from circuit simulation, and reduced current points.

The current and voltage at the input pin are saved and then used to determine *C1* and *C2* values such that gate-level delay calculation can closely match times to the delay threshold and to the second slew threshold at the input pin.

An additional piece of information, input reference time, is necessary in order to calculate cell delays. The reference time is the simulation time at which the waveform at the input pin crosses the rising or falling delay threshold (often this is 50% of V_{dd}).

Chapter 4
OpenTimer: A High-Performance Timing Analysis Tool

4.1 Introduction

The lack of accurate and fast algorithms for high-performance timing analysis tool with incremental capability has been recently pointed out as a major weakness of existing timing optimization flows [11]. In deep submicron era, timing-driven operations are imperative for the success of optimization flows. Optimization transforms change the design and therefore have the potential to significantly affect timing information. The timer must reflect such changes and update timing information incrementally and accurately in order to ensure slack integrity as well as reasonable turnaround time and performance. However, such process requires extremely high complexity especially when path-based analysis is configured. A high-quality incremental timer capable of path-based analysis is definitely advantageous in speeding up the timing closure.

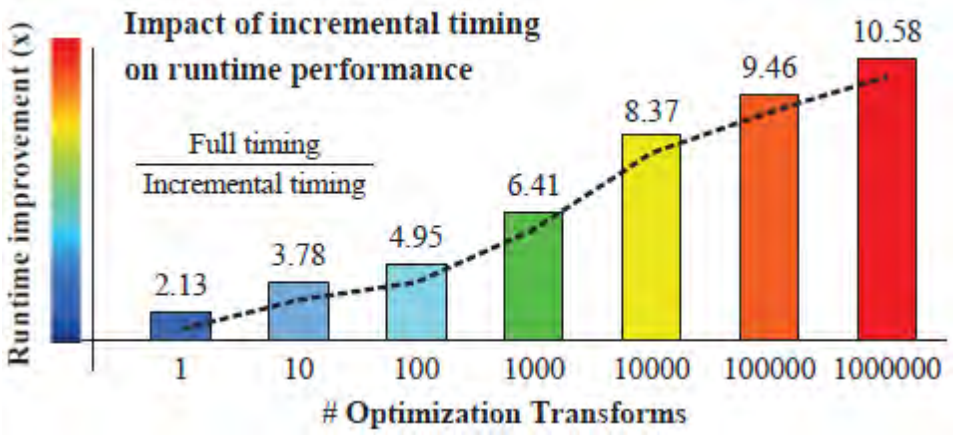

Figure 4.1: Performance improvement of incremental timing to full timing on one benchmark from.

The significance of incremental timing is demonstrated in Figure 4.1. It is observed that the runtime improvement keeps growing as the number of optimization transforms increases. One obvious reason is that once the critical paths in a design have been reported, the optimization tool would optimize the logic (e.g., gate sizing, buffer insertion) so as to overcome the timing violations. This subtle change can affect up to the majority of a circuit, whereas in reality, depending on the trace of critical paths, the timing update may only involve a small portion of the circuit. Since an optimization tool can perform millions of logic transformations, it is important that the timing profile is kept up-to-date in an

incremental fashion. Otherwise, optimization tools cannot support fast turnaround for timing-specific improvement, which dramatically degrades the productivity.

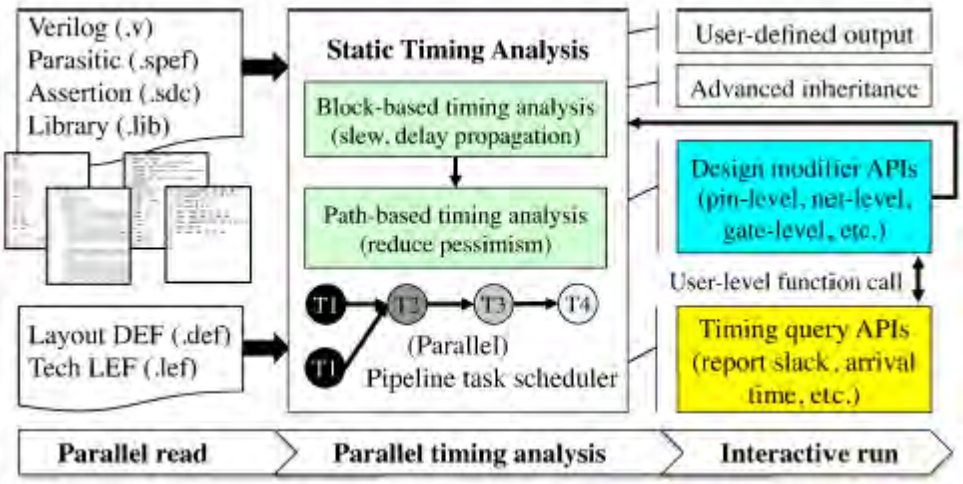

Figure 4.2: Program flow of OpenTimer.

Besides being incremental, one important feature of a practical timer is the capability of common path pessimism removal (CPPR). CPPR is a path-specific timing update that intends to remove redundant pessimism incurred by common segments between data paths and clock paths. Unwanted pessimism might force designers and optimization tools to waste an unnecessary yet significant amount of efforts on fixing paths that meet the intended clock frequency. This problem becomes even more critical when design comes to deep submicron era where data paths are shorter, clocks are faster, and clock networks are longer to accommodate larger and complex chips. However, the real problem is the amount of pessimism that needs to be removed is path-specific. Computational complexity and space requirements for CPPR typically grows exponentially as the design size increases, not to mention the challenge in conjunction with incremental timing analysis. Consequently, an open-source high-performance timing analysis tool, OpenTimer, is presented here and an overview of it is shown in Figure 4.2. Three key features of OpenTimer are highlighted below:

- **Parallel framework.** OpenTimer applies a pipeline task scheduler as the central engine. Critical tasks such as timing propagation and endpoint slack calculation are scheduled into the pipeline so as to overlap their runtimes.
- **Incremental capability.** OpenTimer precisely and minimally captures the features that are key to incremental timing. With lazy evaluation, one can be able to keep computation as minimum as necessary.
- **Path-based analysis.** OpenTimer represents the path implicitly using efficient and compact data structure, yielding a significant saving in both search space and search time for CPPR.

4.2 Incremental Timing Analysis and CPPR

Various stages of the design flow such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis. During these stages, local operations such as gate sizing, buffer insertion, or net rerouting can modify small fractions of the design and significantly change both local and global timing landscape. As the example shown if Figure 4.3, a change on gate *B3* has the potential to affect up to the majority of the circuit (downstream timing). Nevertheless, depending on the trace of critical paths, only a small portion of the timing would need to be updated. For instance, if such a change does not affect the arrival time at *I1:o*, then every downstream timing after *I1:o* is unaffected.

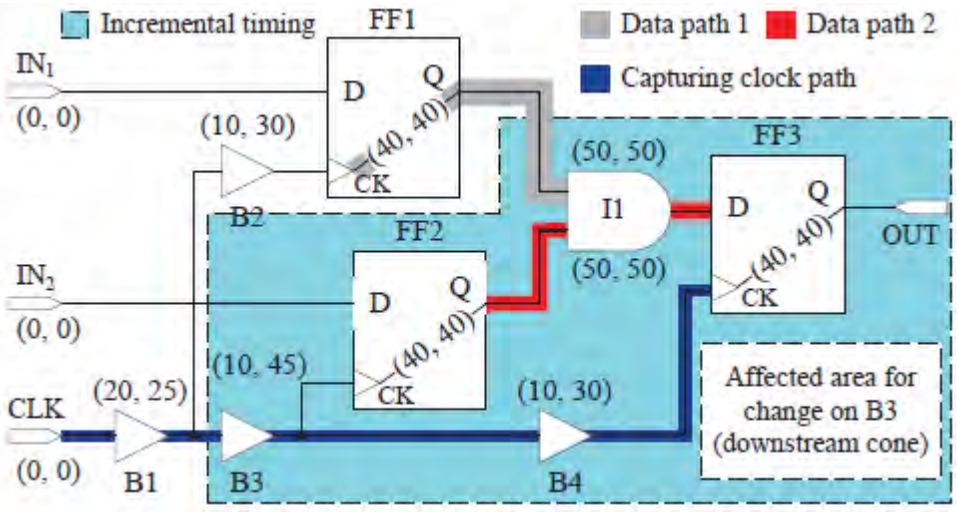

Figure 4.3: An example of sequential circuit network.

In addition to incremental processing, the capability of CPPR is another important component for modern timing analysis tools. Optimization transforms on the data network have no impact on CPPR credit (or CPPR adjustment) for any given launch-capture flip-flop (FF) pairs. Because the clock paths are not changed, any cached value for CPPR credit can be reused. However, in reality many optimization transforms are applied to the clock network, such as resizing a buffer or adding or deleting buffers on the clock tree in order to meet slack or skew targets. These changes can potentially affect a large number of data paths and slacks, and these data points must be recomputed with updated CPPR credits. Further, in some cases, changes on the clock network may not even impact CPPR for any data paths at all. As the example shows in Figure 4.3, the change on *B3* can impact the CPPER credit for the launch-capture *FF* pair *FF2* and *FF3*, while a change on *B4* does not affect the CPPR credit for any *FF* pair. Therefore, the challenge of incremental CPPR is correctly identifying what data points are affected by which changes in an incremental manner.

4.3 NLDM Representation and Data Structures

In this chapter, a presentation of how OpenTimer utilizes the NLDM model, along with the respective data structures, will take place. It will illustrate how graphs are constructed, and analyze briefly the structure of its members.

- First of all, comes the template of the LUT. It consists of two **variables,** variable_1 and variable_2, along with two **indexes**, index_1 and index_2. The valid types of the two variables are **input_net_transition** and **total_output_net_capacitance** and each of them can be assigned either on the first or the second variable. Index values have to be floats considering the corresponding value of the variable. Such a table template would be like this:

  lu_table_template (a_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");
  }

  and the LUT in OpenTimer is stored in a struct with optional enum variables and vectors of floats indices like this:

  struct LutTemplate {
  std::optional<LutVar> **variable1;**
  std::optional<LutVar> **variable2;**

  std::vector<float> **indices1;**
  std::vector<float> **indices2;**
  }

- Moving on to graphs, an introduction to cells will take place. After reading the LUT templates, the first instance the parser reads from the liberty file is a cell. Every early and late library has an equal number of cells with each other, in order to undergo a valid timing simulation. Cells "hold" the information referring to every pin of the gate described. These pins are members of an std::unorderd_map.

- Pins now, can be input, output, or inout. Thus, they have optional members, mostly float, which describe the min and max capacitance and the max transition they can supply. Furthermore, most importantly pins have a vector of timings, each of them characterizing timing information for a certain timing arc.

- This timing information is nothing more than **delay** and **transition** values concerning a respective arc. So, a NLDM timing example should be like this:

```
timing () {
        related_pin : "INP1";
        timing_sense : negative_unate;
        cell_rise : (a_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 (a_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 (a_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 (a_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");
        }
        . . .
}
```

Similarly for setup/hold constraint. These are stored in a timing struct with the following optional members:

struct Timing {

std::optional<TimingLut> **cell_rise**;
std::optional<TimingLut> **cell_fall**;
std::optional<TimingLut> **rise_transition**;
std::optional<TimingLut> **fall_transition**;
std::optional<TimingLut> **rise_constraint**;
std::optional<TimingLut> **fall_constraint**;

}

Where TimingLut being another struct, which holds the name of the LUT template, as also the indices and values with the form of vectors of floats.

For further explanation an illustration of class hierarchy and how the timing information is stored is given in Figure 4.4. As it can be seen, a celllib contains all cells characterized in a liberty file, which are stored in an unordered map using as key their name.

Each cell, contains a group of pins, called cellpins, that characterize the input inner or output pins of the cell. These pins are also stored in an unordered map.

A cellpin now, contains every timing member utilized by any timing arc which contains this pin. Timings are stored in a vector.

Moving on, a timing includes any necessary information for the STA engine, such as rise and fall delays, transitions and constraints. As seen in the figure, these members are optional, meaning that a timing may or may not contain any of them. That is, a timing characterized in the liberty file can be read without any of these members without producing an error.

Finally, each of these timing members is a TimingLut, a LUT model containing a pointer to the LUT template, the value of the two indexes; index1 and index2 which can either be input net transition or total output capacitance and the table which keeps the values. Indexes are stored in the indices, which along with the table are vector types. Of course these are not the only members of each instance, as shown be the three dots at the end of each one.

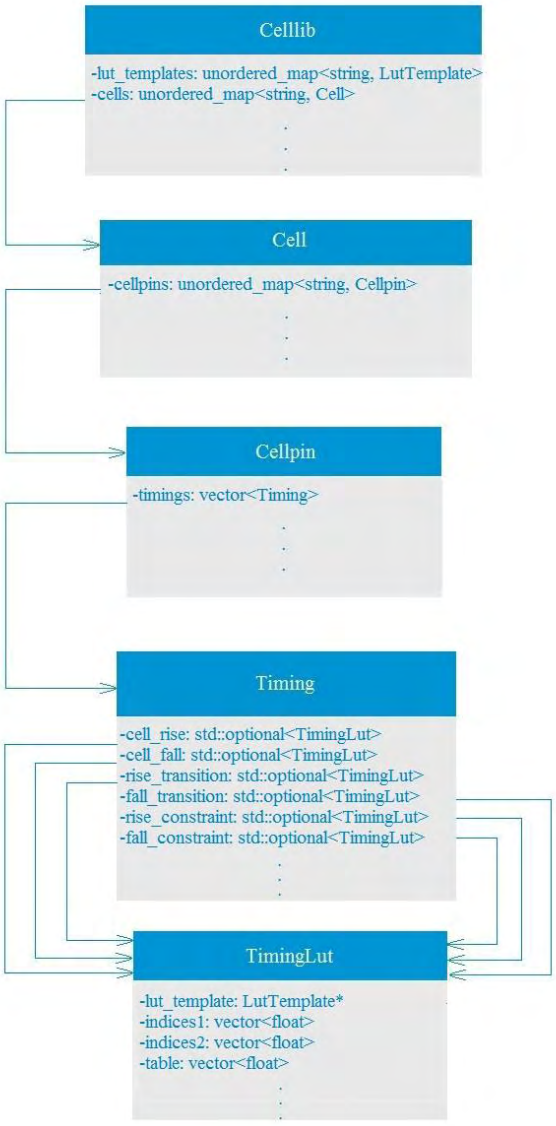

Figure 4.4: Liberty timing information storage.

It is essential to illustrate how graphs are constructed. A depiction of it is given in Figure 4.5. Graphs are constructed through a Verilog (.v) input file, starting with a gate.

A gate apart from its name includes the main parts of the graph. These are the gate's pins and the arcs between them. It also contains a CellView instance, where CellView is of type std::array<const Cell*, MAX_SPLIT>, meaning it is a pointer to a Cell depending on transition enum value (RISE/FALL).

Pins can relate to primary inputs, primary outputs or Cellpins of the corresponding cell of the early/late CellLib. This is described by the member handle which is a variant of type

<PrimaryInput*, PrimaryOutput*, CellpinView>, where CellpinView is a similar to the Cell array, std::array<const Cellpin*, MAX_SPLIT>. Pins also contain two lists, one for their input arcs and another for their output ones.

Arcs finally, apart from members, contain another handle variant. This is of type variant<Net*, TimingView>, meaning it can either be a net arc or a cell arc. TimingView is another array similar to CellView and CellpinView ones', which is of type array<const Timing*, MAX_SPLIT>. The use of it will be explained shortly.

Each timing has a correspondence with an arc, either early or late. This relation has to be unique, meaning one arc indicates to an exact timing and vice versa. An arc can be a net arc, or a cell arc, having difference between them in calculating the output delay and slew respectively.

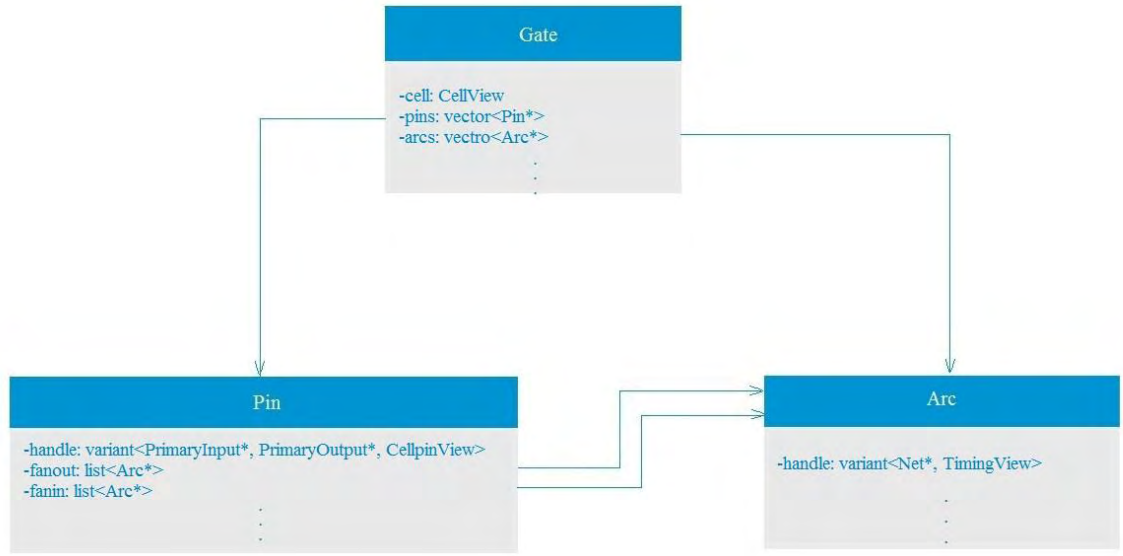

Figure 4.5: Graph construction from Verilog file.

An introduction of how graph construction and liberty information relate is shown in Figure 4.6. A Gate points to a Cell, a Pin relates to a Cellpin only if the variant has the corresponding value. Finally, if an arc is considered as a Cell arc, then it utilizes the relevant timing in order to perform any timing procedure, such as slew or delay propagation, using the respective LUT instances.

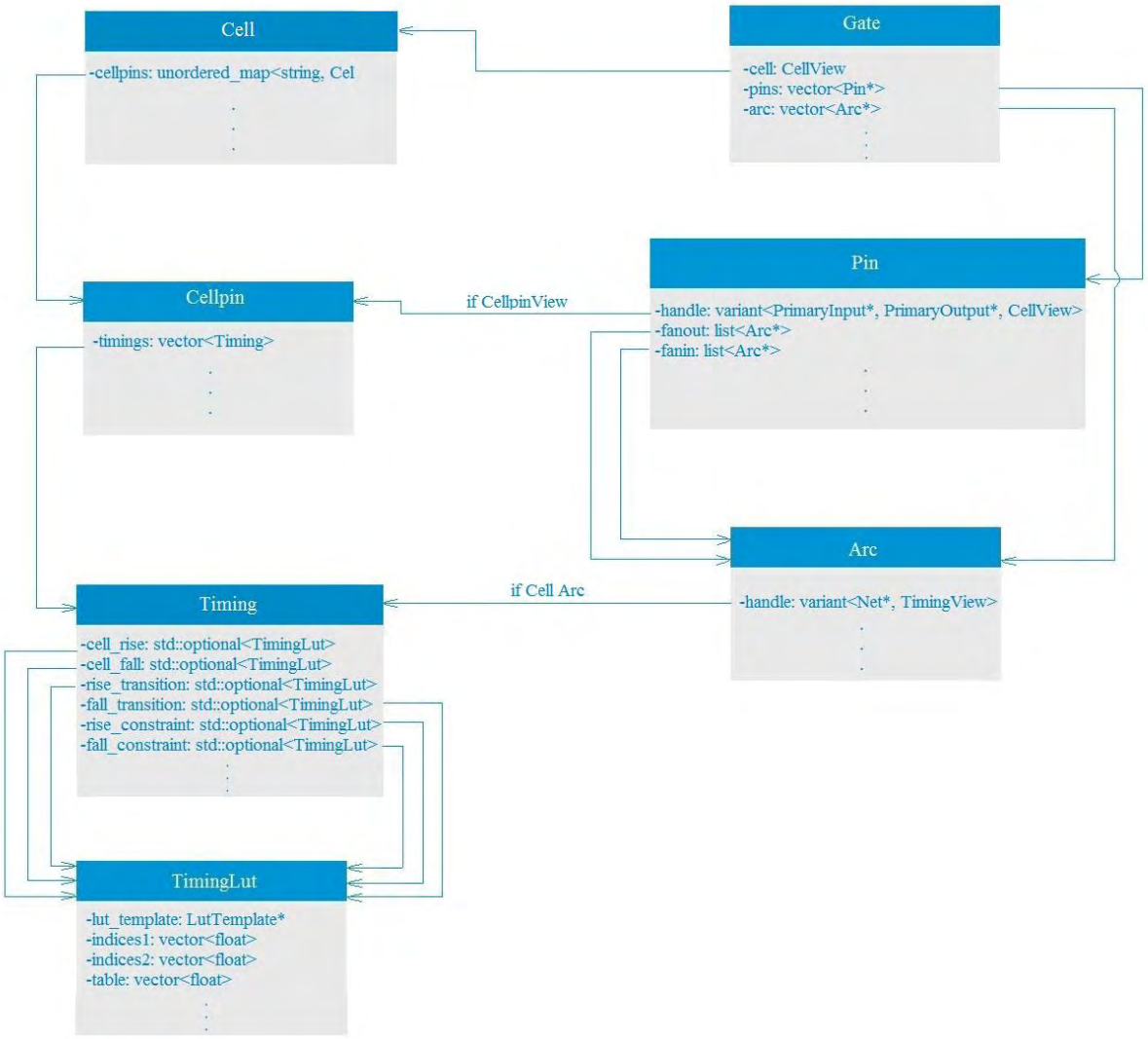

Figure 4.6: Liberty information and graph construction correlation.

Conclusively, a peek on how STA engine is working will be given briefly. While the circuits' graph is traversed, in a pipelined way, any timing information of each node is propagated through Arcs. If an arc is considered a cell arc, then timing information is used to calculate the arrival time, arc delay and slew at pins, based on the NLDM model. On the other hand, if this arc is a Net arc, then the Elmore Delay model is used.

Chapter 5
Integration of CCS Timing Model into OpenTimer

5.1 The Need for Better Timing Accuracy

The necessity of higher accuracy in modern industry has already been mentioned. Efficiency and speed have been constantly rising, along with power reduction.

In this thesis the goal is to provide a state-of-the-art timing tool, OpenTimer, with an even more accurate delay model; and this is nothing else than CCS.

5.2 CCS Timing Information and Data Structures

The idea was to keep timing and NLDM data structures intact and just extend the timing structures with the CCS information and the corresponding timing (slew/delay and C1/C2 capacitance) calculation methods. As shown on a previous chapter CCS timing model is described by two major parts; the **driver** model which replaces the input driving cell arc and the output **receiver** model which replaces the output load pins.

In this section, a demonstration will be given of how the driver and receiver CCS timing information is stored and used inside the STA engine.

Starting with the LUT templates, an introduction to a new variable takes place, meaning there is now a **variable_3**, which can only have one valid value; time. Time is essential because along with the values of the LUT, correspond to the CCS current waveforms (which are used to calculate output slew, cell delay and C1/C2 receiver capacitances).

These time values are float values and are stored in an **indices_3**, which has to come along with the third variable. Thus, the CCS LUT templates (output current templates) must comply to the following structure:

output_curret_template (ccs_template) {
    variable_1 : input_net_transition;
    variable_2 : total_output_capacitance;
    variable_3 : time;
}

To achieve that in OpenTimer, there has been an addition to the enum class LutVar of a new **TIME** value. Therefore, struct LutTemplate now having this form:

```
struct_LutTemplate {
        std::optional<LutVar> variable1;
        std::optional<LutVar> variable2;
        std::optional<LutVar> variable3;

        std::vector<float> indices1;
        std::vector<float> indices2;
        std::vector<float> indices3;
}
```

Driver Model

Timing arcs, as already mentioned have a 1-1 relation with a unique timing, kept hold in a vector of timings by the referenced pin. To support the CCS driver model, timing has to be supplied with output current information with its reference time, indicated by the respective transition and capacitance values.

To achieve this, the solution was found in a form of a map, actually map of maps, with the first key being **transition** and points to the inner map. As for the inner map, the key is **capacitance** and the value is a **pointer to a struct.** To be more precise, two of these maps were added, one for rise transition and one for fall, exactly like the all the other members of the timing struct. This is the CCS form of timing struct:

```
struct Timing {
        std::optional<TimingLut> cell_rise;
        std::optional<TimingLut> cell_fall;
        std::optional<TimingLut> rise_transition;
        std::optional<TimingLut> fall_transition;
        std::optional<TimingLut> rise_constraint;
        std::optional<TimingLut> fall_constraint;

        current_map output_current_rise;
        current_map output_current_fall;
}
```

Because of lack of space, where current_map :
std::map<float, std::map<float, OutputCurrentWaveform*>>

OutputCurrentWaveform is another struct, which contains the mentioned **reference time** as a float value and a **TimingLut** member, where TimingLut has also an indices3 now. This is the struct:

```
struct OutputCurrentWaveform {
        float reference_time;
        TimingLut output_current;
}
```

The corresponding information from the liberty file is member of timing related to a pin:

```
timing {
    related_pin : "INP";
        .
        .
        .
    output_current_rise () {
        vector (output_current_template) {
            reference_time : float;
            index_1 (float);
            index_2 (float);
            index_3 ("float, . . ., float");
            values ("float, . . ., float");
        }
    }
}
```

Notice that index1 and index2 are one dimensional and the matrix containing the values has the size of index3. This affects the corresponding indices.

Furthermore, when the parcer reads the keyword **vector**, a new element inserts to the map, containing the information of the vector.

Receiver Model

Moving on to receiver model, in order to represent the capacitance that an input pin presents to driving cells, four TimingLut objects where added considering the respective transition:

- std::optional<TimingLut> **receiver_capacitance1_rise**;
- std::optional<TimingLut> **receiver_capacitance1_fall**;
- std::optional<TimingLut> **receiver_capacitance2_rise**;
- std::optional<TimingLut> **receiver_capacitance2_fall**;

This capacitance information can either be characterized as a two-dimensional LUT in a timing arc or as a one-dimensional LUT in an input pin. If the characterization takes place in timing the LUT will have two-dimensional values and two indexes, while otherwise one-dimensional and one index.

An example of each case will be given:

- **characterization in timing-level**

timing {
. . .
receiver_capacitance1_fall (template_name) {
index_1 (“*float, . . ., float”*);
index_2 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance1_rise (template_name) {
index_1 (“*float, . . ., float”*);
index_2 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance2_fall (template_name) {
index_1 (“*float, . . ., float”*);
index_2 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance2_rise (template_name) {
index_1 (“*float, . . ., float”*);
index_2 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
}

- **characterization in pin-level**

pin (PinName) {
direction : input; /* or “inout” */
receiver_capacitance1_fall (template_name) {
index_1 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance1_rise (template_name) {
index_1 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance2_fall (template_name) {
index_1 (“*float, . . ., float”*);
values (“*float, . . ., float”*);
}
receiver_capacitance2_rise (template_name) {
index_1 (“*float, . . ., float”*);

values (“*float, . . ., float”*);
}
}

The data structs which have to be changed in order to support the CCS timing information are the following:

Figure 5.1: CCS information added to Cellpin.

1. The Cellpin struct (Figure 5.1): must contain the 1-dimension receiver capacitances.
2. The Timing struct (Figure 5.2): must include the 2-dimension receiver capacitances and the output current maps.

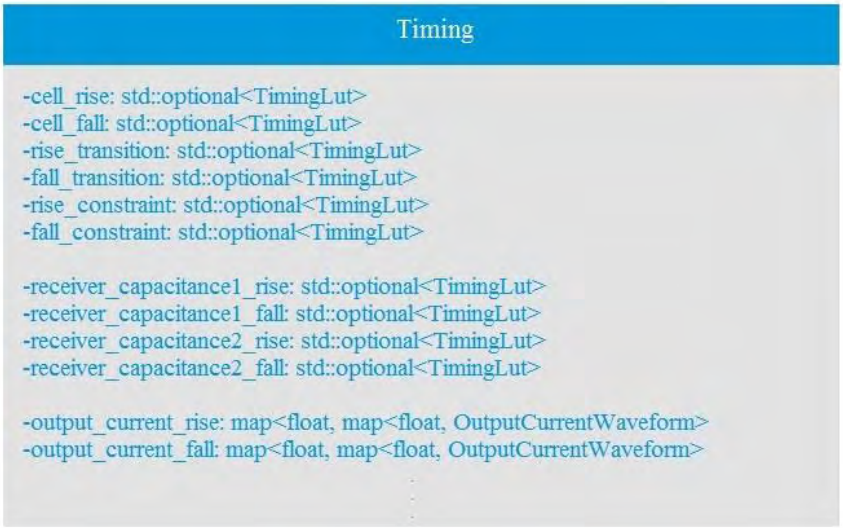

Figure 5.2: CCS information added into Timing.

It is essential to mention that the STA engine procedure, timing information storage and graph construction remain the same as the initial NLDM-based version.

However, new methods had to be implemented in order to calculate slew, delay and C1/C2 capacitance values based on the CCS Timing models. Those methods include:

- Methods to find the closer CCS Timing currents (from Liberty) for any (transition, capacitance) breakpoints during the STA algorithm.
- Transformation of CCS Timing current waveforms to Voltage waveforms (in order to compute slew and delay).
- Advanced interpolation techniques for interpolating on closer waveforms.
- An iterative algorithm for computing C1/C2 receiver capacitances that lead to the worst (min or max slew/arrival time) on each pin during the STA algorithm.

Concerning the STA engine, the only thing that changes is that the calculation of C1/C2 capacitances is added, while the routines calculate the worst arrival time, slew and delay at pins using our new CCS timing methods.

5.3 Results

In order to evaluate the methodology we followed to integrate CCS in OpenTimer, we tried to test its results compared to the results of the golden STA timer, Synopsys PrimeTime. They were both tested on the same circuits and the reported results were on the same critical path. The goal was to compare the reported Arrival Times (AT) on the endpoint of the critical path and figure whether they were close enough or not. Both NLDM and CCS models were used for each of the two tools so that we could come to an overall deduction.

Tables 5.1 and 5.2 show the computed NLDM/CCS AT at the endpoint of the critical path for both OpenTimer and Synopsys PrimeTime for a subset of ISCAS benchmarks. Each of these circuit were synthesized using the NANGate 45nm [12].

Unfortunately, the full path cannot be given, because its width is prohibitive for the table size. Therefore, only the startpoint and endpoint of each circuit are given along with their respective transitions and the path's split.

Circuit	Critical Path	Endpoint AT (ns)			
		With SPEF Parasitics			
		NLDM		CCS	
		OpenTimer	PrimeTime	OpenTimer	PrimeTime
c17	Split: late Startpoint: nx6 fall Endpoint: nx22 rise	0.0649447	0.06421547	0.0653241	0.06488016
c2670	Split: late Startpoint: n2104 rise Endpoint: n329 rise	0.692842	0.67869568	0.701103	0.70696133
c7552	Split: late Startpoint: n18 fall Endpoint: n338 fall	1.02616	0.98675621	1.03168	1.08759308
s27	Split: early Startpoint: G3 fall Endpoint: G17 rise	0.0588331	0.05837075	0.0593287	0.05886924
s1196	Split: late Startpoint: blif_clk_net rise Endpoint: G550 rise	0.724291	0.70778012	0.733295	0.71854591
s1494	Split: late Startpoint: blif_clk_net rise Endpoint: v13_D_24 rise	0.662193	0.65124935	0.673248	0.70559676

Table 5.1: OpenTimer and PrimeTime STA results including SPEF parasitics.

Circuit	Critical Path	Endpoint AT (ns)			
		Without SPEF Paracitics			
		NLDM		CCS	
		OpenTimer	PrimeTime	OpenTimer	PrimeTime
c17	Split: late Startpoint: nx6 fall Endpoint: nx22 rise	0.0592147	0.05849564	0.059541	0.06233476
c2670	Split: late Startpoint: n2104 rise Endpoint: n329 rise	0.627649	0.61069620	0.634376	0.66361535
c7552	Split: late Startpoint: n18 fall Endpoint: n338 fall	0.828644	0.82024121	0.833932	0.92549127
s27	Split: early Startpoint: G3 fall Endpoint: G17 rise	0.0545482	0.05405520	0.0550113	0.05700241
s1196	Split: late Startpoint: blif_clk_net rise Endpoint: G550 rise	0.618879	0.61076224	0.625704	0.63556525
s1494	Split: late Startpoint: blif_clk_net rise Endpoint: v13_D_24 rise	0.569857	0.56639723	0.578867	0.61416802

Table 5.2: OpenTimer and PrimeTime STA results without SPEF parasitics.

Experimental results illustrate that by embedding CCS Timing model into OpenTimer we can achieve a better timing accuracy, especially when not using SPEF parasitics. As we can observe, for every circuit the AT using the OpenTimer's CCS model is closer to the golden CCS AT of PrimeTime, than the one computed using the OpenTimer's NLDM model.

Note that for the above results, C1/C2 capacitances are computed using the NLDM model because we were confronted with a small issue using the iterative algorithm for computing C1/C2 capacitances based on the CCS model. Thus, for the case where SPEF is used, CCS ATs computed by OpenTimer and PrimeTime are not fairly comparable. The accuracy would be even better if the C1/C2 were computed based on the CCS model. Nonetheless, the results seem to have a similar behavior to the PrimeTime ones', meaning that both increase/decrease from NLDM to CCS in a same manner.

Also, we have to mention that for the case that no SPEF is given for the current circuit, PrimeTime by default uses NLDM C1/C2 capacitances if CCS Timing model is chosen for STA. This means that for this case OpenTimer and PrimeTime results are obtained using the same methodology and the results are directly comparable.

Another reason to enforce us not to use SPEF parasitcs is the difference on how the two timers model the interconnect timing. OpenTimer utilizes the Elmore delay model while PrimeTime on the other hand, uses advanced techniques in modeling the interconnect delay.

On the last table (5.3), we provide the accuracy improvement between OpenTimer's NLDM and CCS results in regard to the golden STA result, coming from PrimeTime's CCS model. The resulting accuracies are derived from the percentage error calculation method and refer to the results without SPEF parasitics.

Circuit	Critical Path	Accuracy Error (%)	
		OT NLDM / PT CCS	OT CCS / PT CCS
c17	Split: late Startpoint: nx6 fall Endpoint: nx22 rise	5.3	4.7
c2670	Split: late Startpoint: n2104 rise Endpoint: n329 rise	5.7	4.7
c7552	Split: late Startpoint: n18 fall Endpoint: n338 fall	11.7	11
s27	Split: early Startpoint: G3 fall Endpoint: G17 rise	4.5	3.6
s1196	Split: late Startpoint: blif_clk_net rise Endpoint: G550 rise	2.7	1.5

s1494	Split: late Startpoint: blif_clk_net rise Endpoint: v13_D_24 rise	7.8	6.1

Table 5.3: Accuracy error between OpenTimer NLDM-CCS and the golden CCS PrimeTime.

As it can be easily seen, OpenTimer's CCS is much closer to the golden PrimeTime's CCS, being 1.02%, on average and up to 1.7% more accurate than OpenTimer's NLDM. Furthermore, we should keep in mind that the accuracy improvement would be even more prominent for large industrial designs where the critical path consists of thousands or even millions of gates. Finally, even better results are expected for smaller process geometries, since for sub-20nm technologies the NLDM model is insufficient.

Chapter 6
Conclusion

In conclusion, Static Timing Analysis is a commonly used simulation method in order to compute the expected timing of a digital circuit without requiring to simulate the full circuit, using a transistor level simulator. The attempts to achieve higher accuracy, more efficiency, faster models and better results generally have been continuously and consistently going on.

In the matter of accuracy, NLDM is an efficient one, but CCS has even better results, as this thesis managed to prove, with the successful integration of CCS timing model to OpenTimer. As shown in the results, higher accuracies can be accomplished with the use of it. Using circuits without SPEF parasitics, it can produce 1.02% on average and up to 1.7% more accurate results than NLDM model.

6.1 Future Work

An interesting addition to this project could probably be the integration of a more accurate wire delay model, rather than Elmore delay, thus providing even better results to the final timing simulation of the circuit.

References

[1] TAU Contest 2016. [2016].
Available: https://sites.google.com/site/taucontest2016/

[2] J. Bashker, Rakesh Chadha, Static Timing Analysis for Nanometer Designs a Practical Approach, 2009.

[3] Synopsys PrimeTime Golden Signoff Solution.
Available:
https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

[4] Cadence Tempus Timing Signoff Solution.
Available:
https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/silicon-signoff/tempus-timing-signoff-solution.html

[5] VLSI Physical Design: Interface Logic Models – VLSI Junction. [2015].
Availble: http://www.vlsijunction.com/2015/11/interface-logic-models.html

[6] W. C. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wide-band Amplifiers", Journal of Applied Physics, 19(1)(1948), pp. 55-63.

[7] P. Penfield Jr. and J. Rubinstein, "Signal Delay in RC Tree Networks", Proc. Design Automation Conference, 1981, pp. 613-617.

[8] R. Gupta, B. Tutuianu and L. T. Pileggi, "The Elmore Delay as a Bound for RC Trees with Generalized Input Signals", IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, 16(1)(1997), pp. 95-104.

[9] C. V. Kashyap, C. J. Alpert, F. Liu and A. Devgan, "Closed-form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs for RC Trees", IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, 23(4)(2004), pp. 509-516.

[10] C. L. Ratzlaff and L. T. Pillage, "RICE: Rapid Interconnect Circuit Evaluation Using AWE", IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,13(6)(1994), pp. 763-776.

[11] T. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool, "2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2015.

[12] NanGate FreePDK45 Open Cell Library. [2008].
Available: http://www.nangate.com/?page_id=2325