ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Ανάλυση και Ανάπτυξη Αλγορίθμων για τη Βελτιστοποίηση Ψηφιακών Κυκλωμάτων σε Όρους Ταχύτητας & Κατανάλωσης Ισχύος

Analysis and Development of Digital Circuits Optimization Algorithms in Terms of Speed and Power Dissipation

Μεταπτυχιακή Διατριβή

Χρήστος Ν. Καλονάκης Δήμος Π. Ντιούδης

Επιβλέποντες Καθηγητές :

Σταμούλης Γεώργιος
Ευμορφόπουλος Νέστωρ
Μποζάνης Παναγιώτης

Βόλος, Ιούνιος 2015

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Ανάλυση και Ανάπτυξη Αλγορίθμων για τη Βελτιστοποίηση Ψηφιακών Κυκλωμάτων σε Όρους Ταχύτητας & Κατανάλωσης Ισχύος

Μεταπτυχιακή Διατριβή

Χρήστος Ν. Καλονάκης Δήμος Π. Ντιούδης

Επιβλέποντες :

Σταμούλης Γεώργιος
Ευμορφόπουλος Νέστωρ
Μποζάνης Παναγιώτης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Ιουνίου 2015

....

Διπλωματική Εργασία για την απόκτηση του Μεταπτυχιακού Διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας.

…………………………….. ……………………………..

Χρήστος Ν. Καλονάκης Δήμος Π. Ντιούδης

Διπλωματούχοι Μηχανικοί Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και

Δικτύων Πανεπιστημίου Θεσσαλίας

To our friends and families

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα θέλαμε να ευχαριστήσουμε θερμά τους επιβλέποντες της διπλωματικής εργασίας για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μας, την άριστη συνεργασία, την συνεχή καθοδήγηση και τις ουσιώδεις υποδείξεις και παρεμβάσεις, που διευκόλυναν την εκπόνηση της μεταπτυχιακής αυτής διατριβής. Επίσης, θα θέλαμε να ευχαριστήσουμε τους φίλους και συνεργάτες του Εργαστηρίου Ε5 για την υποστήριξη και την δημιουργία ενός ευχάριστου και δημιουργικού κλίματος, για τις εύστοχες υποδείξεις τους και την συνεχή στήριξή τους. Τέλος, οφείλουμε ένα μεγάλο ευχαριστώ στις οικογένειές μας και στους φίλους μας για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μας παρείχαν τόσο κατά την διάρκεια των σπουδών μας όσο και κατά την εκπόνηση της μεταπτυχιακής αυτής διατριβής.

Καλονάκης Χρήστος,
Ντιούδης Δήμος
Βόλος, 2015

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

EDA	Electronic Design Automation
CCSopt	Continuous Cell Size Optimizer
IC	Integrated Circuit
CMOS	Complementary Metal Oxide Semiconductor
FINFET	Fin shaped Field Effect Transistor
RTL	Register Transfer Level
VHDL	Very High Speed Integrated Circuit Hardware Description Language
DEF	Design Exchange Format
SPEF	Standard Parasitic Exchange Format
LE	Logical Effort
VLSI	Very Large Scale Integration
ULE	Unified Logical Effort
CCS	Composite Current Source
NLDM	Non Linear Delay Model
STA	Static Timing Analysis

Abstract

Optimal continuous transistor/device sizing has been a holy grail in the EDA community. However, efforts to this end have been hampered by the sheer size of the optimization problem (millions of variables and constraints), modeling issues especially in the timing domain. This research work proposes Continuous Cell Size Optimizer (CCSopt), a continuous transistor sizing tool taking into consideration power and timing constraints, in order to arrive at solutions that are reliably implemented in silicon, and easily integrated into mainstream design flows. CCSopt comprises a hybrid heuristic approach and state-of-the-art algorithms for finding the optimal transistor sizes. In addition, CCSopt can exploit the computational power of parallel architectures in order to decrease execution time and enable analysis of very large-scale integrated circuits.

Keywords:

Transistor sizing, power optimization, delay optimization, logical effort, static timing analysis

Περίληψη

Η εύρεση των βέλτιστων μεγεθών των τρανζίστορ, υπήρξε το άγιο δισκοπότηρο που απασχόλησε την κοινότητα του EDA. Παρόλα αυτά, όποιες προσπάθειες πραγματοποιήθηκαν, για αυτόν το σκοπό, παρεμποδιστήκαν από το τεράστιο μέγεθος του προβλήματος (εκατομμύρια μεταβλητές και περιορισμοί) και από ζητήματα μοντελοποίησης, κυρίως στον τομέα του χρονισμού. Η συγκεκριμένη ερευνητική προσπάθεια προτείνει το Continuous Cell Size Optimizer (CCSopt), ένα εργαλείο κλιμάκωσης των μεγεθών των τρανζίστορ, με συνεχή τρόπο, το οποίο λαμβάνει υπόψη περιορισμούς ισχύος και χρονισμού, με σκοπό να προσεγγίσει λύσεις οι οποίες μπορούν να υλοποιηθούν στο πυρίτιο με αξιόπιστο τρόπο, ενώ μπορεί εύκολα να ενοποιηθεί στις επικρατούσες ροές σχεδιασμού. Το CCSopt περιλαμβάνει μια υβριδική ευρεστική προσέγγιση, σε συνδυασμό με εξειδικευμένους αλγορίθμους με σκοπό την εύρεση των βέλτιστων μεγεθών των τρανζίστορ. Επίσης, το CCSopt εκμεταλλεύεται την υπολογιστική δύναμη των παράλληλων αρχιτεκτονικών ώστε να μειώσει το χρόνο εκτέλεσης και να καταστήσει ικανή την ανάλυση ολοκληρωμένων κυκλωμάτων πολύ μεγάλης κλίμακας.

Λέξεις Κλειδιά:

Κλιμάκωση Τρανζίστορ, βελτιστοποίηση ισχύος, βελτιστοποίηση καθυστέρησης, logical effort, στατική ανάλυση χρονισμού

Chapter 1

Introduction

1.1 Problem Description

This thesis is part of the research project Nanotrim, a work supported by EU and the Greek State through ESPA 2017-2013, Action SYNERGASIA 2011, Project Code: 11SYN 5 719. The project aims to make significant advances in Electronic Design Automation (EDA) technology, to develop key enablers for the performance optimization of nanoscale integrated circuits (ICs) and allow for significantly more power-efficient chips. This is targeted by means of innovative methods and algorithms for physical (i.e. gate- and transistor-level) synthesis, which are pioneered by the partners. The project shall build on top of existing technology and knowledge owned by the partners, proven through several years of Silicon Valley expertise and a distinguished commercial track record.

The proposed activity is building on the learnings from both academic and industrial attempts to tackle a difficult yet attractive design problem. The approach taken is to perform continuous sizing optimization but in a constrained mode, in order to arrive at solutions that are reliably implemented in silicon, and easily integrated into mainstream design flows. The project's team brings together experience in all areas required to not only provide a world-class solution to the continuous sizing problem but also to eventually, successfully incorporate this solution into a viable product, addressing all issues that have prevented previous academic and industrial efforts from arriving at this goal.

The major research vectors are:

1. Development of a continuous transistor sizing EDA tool, which takes into account timing and power constraints.
2. Development of a layout (physical design) manipulation tool, which will implement in the physical layout, the sizes that are calculated by the continuous transistor sizing tool.
3. Special, parallelized algorithms and distributed data models that will deal with the massive data requirements of this undertaking.
4. Development of nano-device models (for CMOS transistors under 45nm, with a focus on FINFETs at 22nm and beyond), which will express both timing performance and dissipated power of basic circuit cells.

This thesis deals with the implementation of the 1st aforementioned research vector.

1.2 Related Work on Transistor Sizing

Transistor sizing tools have been around since the publication on TILOS [1] [2]. Initially the sizing effort was focused on transistor sizes only, for which a number of approaches have been developed (posynomial sizing [3], logical [4] [5], AMPS [6]). Further strains of the aforementioned basic approaches have been proposed initially for timing and area optimization, and, later for multi-objective cost functions involving mainly power [7] [8]. Most of the sizing tools are path based, meaning that they treat the transistors of gates along a path as an optimization sub-problem, which can cause serious conflicts especially in similarly timed reconvergent fanout paths.

A major undertaking has been to observe design constraints while performing transistor sizing. Minimum and maximum slope constraints have been the most difficult to implement as they are not very compatible with any of the sizing methods that have been proposed thus far. Minimum and maximum transistor sizes, maximum delay constraints, and fixed relative transistor sizing are more straightforward to implement. Recent academic papers and patents incorporate interconnect capacitance and, in some cases resistance, to account for the delay in interconnect lines [9].

Our hybrid heuristic approach takes into account all the aforementioned constraints as well as interconnect capacitance and resistance. It also takes into account reconvergent fan-outs and arrives at a stable solution in all cases without the possibility of divergence, which has plagued even commercial tools like AMPS.

1.3 Design Flow

This section presents the top-down design flow and its steps. As shown in Figure 1 the first step is the creation of synthesizable VHDL[1] Register Transfer Level (RTL) models. Such models describe the structure and the behavior of the design at a relatively high level of abstraction. VHDL RTL models can be then verified using logic simulation. VHDL test-bench models define a relevant set of stimulus to be applied to the design under test and verification procedures on the output signals of the design under test. Logic simulation uses abstract logic signals and event-driven behaviors to achieve fast simulation times. The simulation of VHDL RTL models essentially checks the design functionality. No timings are considered yet at that stage.

The RTL (or logic) synthesis step infers a possible gate-level realization of the input RTL description that meets user-defined constraints such as area, timings or power consumption. The design constraints are defined outside the VHDL models by means of tool-specific

[1] Very High Speed Integrated Circuit Hardware Description Language

commands. The targeted logic gates, also known as standard cells, belong to a library that is provided by a foundry or an IP company as part of a so-called design kit. Typical gate libraries include a few hundreds of combinational gates (e.g., inverter, NAND or MUX gates) and sequential logic gates (e.g., flip-flops, latches). Each logic function is implemented in several gates to accommodate several fanout capabilities or drive strengths. The gate library is described in a tool-specific format that defines, for each gate, its function, its area, its timing and power characteristics and its environmental constraints. The synthesis step generates several outputs one of which is a Verilog gate-level netlist. The Verilog netlist can be used as input to our CCSopt tool, as well as the input to the place & route step.

The Place & Route (P & R) step infers a geometric realization of the gate-level netlist, so-called a layout. The standard cell design style places logic gates in rows of equal heights. As a consequence, all standard cells from the library have the same height, but may have different widths. Each cell has a power rail at its top and a ground rail at its bottom. The interconnections (routing) between gates are done over the cells since current processes allow several metal layers (i.e. up to 9 metal layers for the UMC 90nm CMOS process). As a consequence, the rows may be abutted and flipped so power and ground rails are shared between successive rows. Placement and routing can consider timing constraints, usually the same as the ones defined for the RTL synthesis step. Special nets such as power/ground wires and clocks are usually routed separately to meet specific constraints such as, respectively, voltage drop and electro migration, or clock skew.

The P & R step generates a geometric description (layout) in DEF format which can used as input to the parasitic extraction step as well as the input to the layout sizing step which is the 2nd major vector of the Nanotrim project.

This parasitic extraction step calculates the parasitic effects in both the designed devices and the required wiring interconnects of an electronic circuit. The major purpose of parasitic extraction is to create an accurate analog model of the circuit, so that detailed simulations can emulate actual digital and analog circuit responses. Digital circuit responses are often used to populate databases for signal delay and loading calculation such as: timing analysis; circuit simulation; and signal integrity analysis. Analog circuits are often run in detailed test benches to indicate if the extra extracted parasitics will still allow the designed circuit to function. The parasitic extraction step generates a net parasitic in spef[2] format which is used as the second input to our tool.

[2] Standard Parasitic Exchange Format

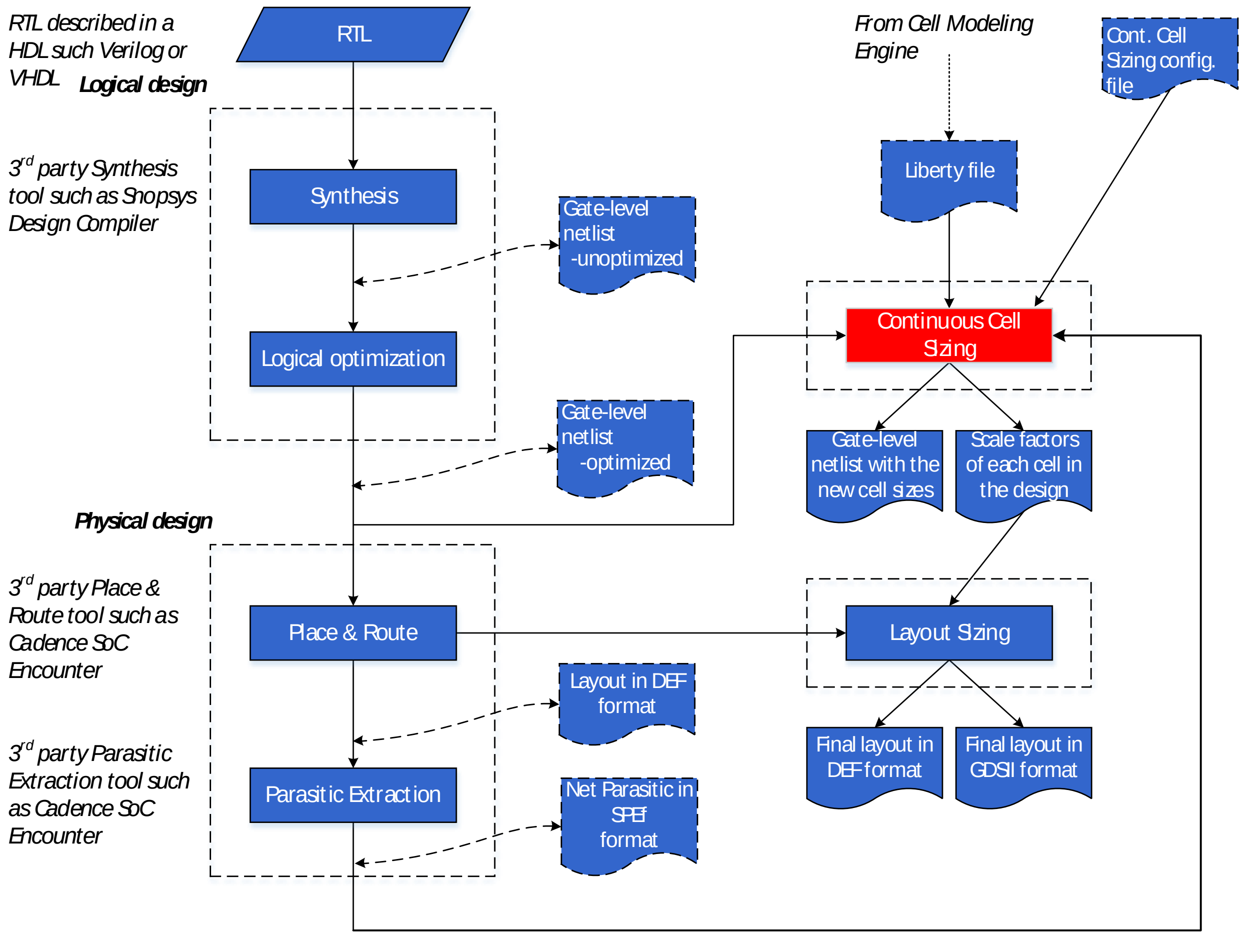

RTL described in a HDL such Verilog or VHDL
RTL
Logical design
3rd party Synthesis tool such as Snopsys Design Compiler
Synthesis
Gate-level netlist -unoptimized
Logical optimization
Gate-level netlist -optimized
Physical design
3rd party Place & Route tool such as Cadence SoC Encounter
Place & Route
Layout in DEF format
3rd party Parasitic Extraction tool such as Cadence SoC Encounter
Parasitic Extraction
Net Parasitic in SPEF format
From Cell Modeling Engine
Cont. Cell Sizing config. file
Liberty file
Continuous Cell Sizing
Gate-level netlist with the new cell sizes
Scale factors of each cell in the design
Layout Sizing
Final layout in DEF format
Final layout in GDSII format

Figure 1 - Design Flow.

The rest of this thesis is organized as follows. In section 2 we give background material on certain useful fragments of cell resizing. In section 3 we present the implementation steps of the CCSopt tool. Section 4 presents experimental results on several benchmarks. Section 5 provides ideas for future work.

Chapter 2

Cell-Resizing Methods

2.1 Logical Effort

2.1.1 Introduction

Timing modeling and optimization are two of the primary issues in high complexity circuit design. The method of Logical Effort (LE) [10], a term invented by I. Sutherland and B. Sproull in 1991, is a straightforward technique for fast evaluation and optimization of delay in logic paths (see Figure 2). The technique has since been adopted as a basis for numerous CAD tools, for the sake of its simplicity.

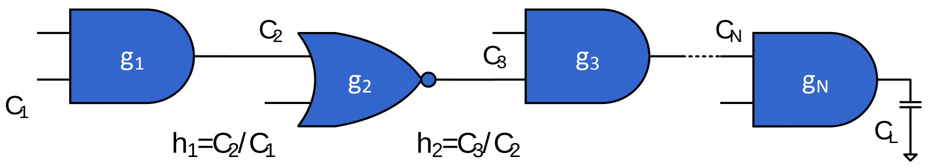


Figure 2 - Logical effort optimization for gates without wires is based on equal stage efforts, $g_1h_1=g_2h_2$ etc.

2.1.2 Delay in a Logic Gate

The LE method is founded on a simple model of delay [4] through a single MOS logic gate. The model describes delays caused by the capacitive load that the logic gate drives and by the topology of the logic gate. Clearly, as the load increases, the delay increases, but delay also depends on the logic function of the gate. Inverters, the simplest logic gates, drive loads best and are often used as amplifiers to drive large capacitances. Logic gates that compute other functions require more transistors, some of which are connected in series, making them poorer than inverters at driving current. Thus a NAND gate has more delay than an inverter with similar transistor sizes that drives the same load. The method of logical effort quantifies these effects to simplify delay analysis for individual logic gates and multistage logic networks.

As a first step, delay is expressed in terms of a basic delay unit, τ[3], which is the delay of an inverter driving an identical inverter with no parasitic capacitance. The unit-less number associated with this is known as the normalized delay. The absolute delay is then simply defined as the product of the normalized delay of the gate, d, and τ:

$$d_{abs} = d \times \tau \tag{1}$$

The delay incurred by a logic gate is comprised of two components, a fixed part called the parasitic delay p and a part that is proportional to the load on the gate's output, called the effort delay or stage effort f. The total delay, measured in units of τ, is the sum of the effort and parasitic delays:

$$d = f + p \tag{2}$$

The effort delay depends on the load and on properties of the logic gate driving the load. We introduce two related terms for these effects: the logical effort g captures properties of the logic gate, while the electrical effort h characterizes the load. The effort delay of the logic gate is the product of these two factors:

$$f = g \times h \tag{3}$$

The logical effort g captures the effect of the logic gate's topology on its ability to produce output current. It is independent of the size of the transistors in the circuit. The electrical effort h describes how the electrical environment of the logic gate affects performance and how the size of the transistors in the gate determines its load-driving capability. The electrical effort is defined by,

$$h = \frac{C_{out}}{C_{in}} \tag{4}$$

where C_{out} is the capacitance that loads the output of the logic gate and C_{in} is the capacitance presented by the input terminal of the logic gate. Electrical effort is also called fanout by many CMOS designers.

Combining the last two equations, we obtain the basic equation that models the delay through a single logic gate, in units of τ:

$$d = g \times h + p \tag{5}$$

This equation shows that logical effort g and electrical effort h both contribute to delay in the same way. This formulation separates τ, g, h, and p, the four contributions to delay. The process parameter τ represents the speed of the basic transistors. The parasitic delay p

[3] In a typical 600-nm process τ is about 50 ps. For a 250-nm process, τ is about 20 ps. In modern 45 nm processes the delay is approximately 4 to 5 ps.

expresses the intrinsic delay of the gate due to its own internal capacitance, which is largely independent of the size of the transistors in the logic gate. The electrical effort, h, combines the effects of external load, which establishes C_{out}, with the sizes of the transistors in the logic gate, which establish C_{in}. The logical effort g expresses the effects of circuit topology on the delay free of considerations of loading or transistor size. Logical effort is useful because it depends only on circuit topology.

	Number of inputs					
Gate type	1	2	3	4	5	n
Inverter	1					
NOR		$5/3$	$7/3$	$9/3$	$11/3$	$(2n+1)/3$
NAND		$4/3$	$5/3$	$6/3$	$7/3$	$(n+2)/3$

Table 1 - Logical effort for inputs of static CMOS gates, assuming γ=2. γ is the ratio of an inverter's pull-up transistor width to pull-down transistor width.

Logical effort values for a few CMOS logic gates are shown in Table 1. Logical effort is defined so that an inverter has a logical effort of 1. An inverter driving an exact copy of itself experiences an electrical effort of 1. Therefore, an inverter driving an exact copy of itself will have an effort delay of 1, according to third equation.

The logical effort of a logic gate tells how much worse it is at producing output current than is an inverter, given that each of its inputs may present only the same input capacitance as the inverter. Reduced output current means slower operation, and thus the logical effort number for a logic gate tells how much more slowly it will drive a load than would an inverter. Equivalently, logical effort is how much more input capacitance a gate must present in order to deliver the same output current as an inverter.

It is interesting but not surprising to note from Table 1 that more complex logic functions have larger logical effort. Moreover, the logical effort of most logic gates grows with the number of inputs to the gate. Larger or more complex logic gates will thus exhibit greater delay. These properties make it worthwhile to contrast different choices of logical structure.

2.1.3 Multistage Logic Networks

The method of logical effort reveals the best number of stages in a multistage network and how to obtain the least overall delay by balancing the delay among the stages. The notions of logical and electrical effort generalize easily from individual gates to multistage paths.

The logical effort along a path compounds by multiplying the logical efforts of all the logic gates along the path. We use the uppercase symbol G to denote the path logical effort, so that it is distinguished from g, the logical effort of a single gate in the path. The subscript i indexes the logic stages along the path.

$$G = \prod g_i \tag{6}$$

The electrical effort along a path through a network is simply the ratio of the capacitance that loads the last logic gate in the path to the input capacitance of the first gate in the path. We use an uppercase symbol H to indicate the electrical effort along a path.

$$H = \frac{C_{out}}{C_{in}} \tag{7}$$

In this case, C_{in} and C_{out} refer to the input and output capacitances of the path as a whole, as may be inferred from context. We need to introduce a new kind of effort, named branching effort, to account for fanout within a network. So far we have treated fanout as a form of electrical effort: when a logic gate drives several loads, we sum their capacitances, to obtain an electrical effort. Treating fanout as a form of electrical effort is easy when the fanout occurs at the final output of a network. This method is less suitable when the fanout occurs within a logic network because we know that the electrical effort for the network depends only on the ratio of its output capacitance to its input capacitance. When fanout occurs within a logic network, some of the available drive current is directed along the path we are analyzing, and some is directed off that path. We define the branching effort b at the output of a logic gate to be

$$b = \frac{C_{on-path} + C_{off-path}}{C_{on-path}} = \frac{C_{total}}{C_{useful}} \tag{8}$$

where $C_{on-path}$ is the load capacitance along the path we are analyzing and $C_{off-path}$ is the capacitance of connections that lead off the path. Note that if the path does not branch, the branching effort is one. The branching effort along an entire path B is the product of the branching effort at each of the stages along the path.

$$B = \prod b_i \tag{9}$$

Armed with definitions of logical, electrical, and branching effort along a path, we can define the path effort F. Again, we use an uppercase symbol to distinguish the path effort from the stage effort f associated with a single logic stage. The equation that defines path effort is reminiscent of the third equation, which defines the effort for a single logic gate:

$$F = G \times B \times H \tag{10}$$

Note that the path branching and electrical efforts are related to the electrical effort of each stage:

$$B \times H = \frac{C_{out}}{C_{in}} \prod b_i = \prod h_i \tag{11}$$

Although it is not a direct measure of delay along the path, the path effort holds the key to minimizing the delay. Observe that the path effort depends only on the circuit topology and loading and not upon the sizes of the transistors used in logic gates embedded within the network. Moreover, the effort is unchanged if inverters are added to or removed from the path, because the logical effort of an inverter is one. The path effort is related to the minimum achievable delay along the path, and permits us to calculate that delay easily. Only a little more work yields the best number of stages and the proper transistor sizes to realize the minimum delay.

The path delay D is the sum of the delays of each of the stages of logic in the path. As in the expression for delay in a single stage (equation 5), we shall distinguish the path effort delay D_F and the path parasitic delay P:

$$D = \sum d_i = D_F + P \tag{12}$$

The path effort delay is simply:

$$D_F = \sum g_i \times h_i \tag{13}$$

and the path parasitic delay is:

$$P = \sum p_i \tag{14}$$

Optimizing the design of an N-stage logic network proceeds from a very simple result: The path delay is least when each stage in the path bears the same stage effort. This minimum delay is achieved when the stage effort is:

$$\hat{f} = g_i \times h_i = F^{1/N} \tag{15}$$

We use a hat over a symbol to indicate an expression that achieves minimum delay.

Combining these equations, we obtain the principal result of the method of logical effort, which is an expression for the minimum delay achievable along a path:

$$\hat{D} = N \times F^{1/N} + P \tag{16}$$

To equalize the effort borne by each stage on a path, and therefore achieve the minimum delay along the path, we must choose appropriate transistor sizes for each stage of logic along the path. Equation 15 shows that each logic stage should be designed with electrical effort

$$\widehat{h_i} = \frac{F^{1/N}}{g_i} \qquad (17)$$

From this relationship, we can determine the transistor sizes of gates along a path. Start at the end of the path and work backward, applying the capacitance transformation:

$$C_{in_i} = \frac{g_i \times C_{out_i}}{\hat{f}} \qquad (18)$$

This determines the input capacitance of each gate, which can then be distributed appropriately among the transistors connected to the input.

2.2 Unified Logical Effort

2.2.1 Introduction

The LE method benefits from an uncomplicated and intuitive delay model and closed-form optimization conditions. The optimization rule of logical effort, however, only addresses logic gates and does not consider on-chip wires. As VLSI circuits continue to scale, the contribution of wires to the delay increases and cannot be neglected. This characteristic occurs not only with respect to long wires connecting separate modules but also to the interconnect within logic modules where the delays introduced by the wires connecting closely coupled gates approach and can exceed the gate delays. The useful LE rule that the path delay is minimum when the effort of each stage is equal breaks down, because interconnect has fixed capacitances which do not correlate with the characteristics of the gates (see Figure 3). This behavior is described by the authors of the LE method as “one of the most dissatisfying limitations of logical effort”.

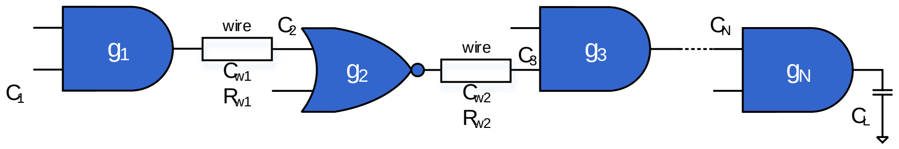


Figure 3 - In the case of gates with wires, the rule of equal effort breaks down because of fixed wire parameters.

2.2.2 Delay Model of Logic Gates with Wires

The logical effort model is modified to include the interconnect delay [3]. This change is achieved by extending the gate logical effort delay by the wire delay, establishing a Unified Logical Effort (ULE) model. Thanks to the Elmore [12] delay model the delay of a circuit comprising logic gates and wires (see Figure 4) can be easily calculated.

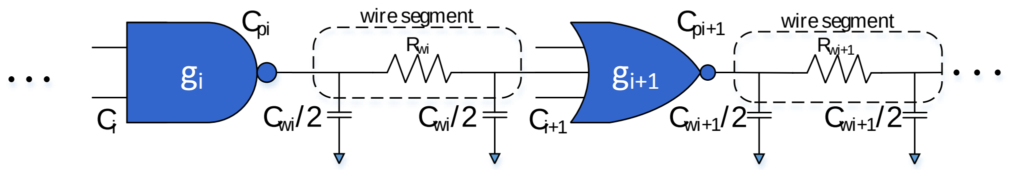


Figure 4 - Cascaded logic gates with resistive-capacitive interconnect.

The total combined delay expression is [13]:

$$D_i = R_i \times (C_{pi} + C_{wi} + C_{i+1}) + R_{wi} \times (0.5 \times C_{wi} + C_{i+1}) \qquad (19)$$

where R_i is the effective output resistance of the gate i , C_{pi} is the parasitic output capacitance of gate i , C_{wi} and R_{wi} are, respectively, the wire capacitance and resistance of segment i , and C_{i+1} is the input capacitance of gate $i+1$.

This expression can be rewritten similar to [14], [15] and [16] in function of the delay of a minimum sized inverter $\tau = R_0 C_0$, where R$_0$ and C$_0$ are the output resistance and input capacitance of a minimum sized inverter:

$$D_i = \tau \times d_i = \tau \times [\frac{R_i}{R_0} \times \frac{C_{wi} + C_{i+1} + C_{pi}}{C_0} + \frac{R_{wi}}{R_0 \times C_0} \times (0.5 \times C_{wi} + C_{i+1}) \qquad (20)$$

The delay d_i normalized with respect to a minimum sized inverter delay τ is defined by:

$$d_i = g_i \times \left(h_i + \frac{C_{wi}}{C_i}\right) + \frac{R_{wi} \times (0.5 \times C_{wi} \times C_{i+1})}{\tau} + p_i \qquad (21)$$

where,

$$g_i = (R_i \times C_i)/(R_0 \times C_0) \quad \textit{is the logical effort ,} \qquad (22)$$
$$h_i = C_{i+1}/C_i \quad \textit{is the electrical effort,} \qquad (23)$$
$$p_i = \left(R_i \times C_{pi}\right)/(R_0 \times C_0) \quad \textit{is the parasitic delay.} \qquad (24)$$

The capacitive interconnect effort h_w and the resistive interconnect effort p_w are, respectively:

$$h_{wi} = \frac{C_{wi}}{C_i} \qquad (25)$$

$$p_{wi} = \frac{R_{wi} \times (0.5 \times C_{wi} + C_{i+1})}{\tau} \qquad (26)$$

The wire influences the electrical effort of the logic gate with h_w and contributes more delay to the total delay with p_w. The final expression of the ULE delay of a single logic gate considering the interconnect is:

$$d = g \times (h + h_w) + (p + p_w) \qquad (27)$$

For an N stage logic path with interconnect the ULE delay is the sum of each delay of the single stage:

$$d = \sum_{i=1}^{N} g_i \times (h_i + h_{wi}) + (p_i + p_{wi}) \qquad (28)$$

Note that in the case of short wires, the resistance R_w of the wire may be neglected, eliminating p_w and leaving only the capacitive interconnect effort h_w in the expression. When the wire impedance along the logic path is negligible, the extended delay expression reduces to the standard LE delay equation.

2.2.3 Delay Minimization using Unified Logical Effort

As a first step in the path delay optimization process, consider a two-stage portion of a logic path with wires (as shown in Figure 4). The condition for optimal gate sizing is determined by equating the derivative of the delay with respect to the gate size to zero. As proven [11] the resulting optimum condition is:

$$(R_i + R_{w_i}) \times C_{i+1} = R_{i+1} \times (C_{i+2} + C_{w_{i+1}}) \qquad (29)$$

The meaning of the optimum size of gate $i+1$ is achieved when the delay component $(R_i + R_{w_i}) \times C_{i+1}$ due to the gate capacitance is equal to the delay component $R_{i+1} \times (C_{i+2} + C_{w_{i+1}})$ due to the effective resistance of the gate. A schematic model describing the related delay components is shown in Figure 5.

After solving the differential equations that occur in the optimization problem [3], we get the expression for the optimum input capacitance of each gate based on the ULE model:

$$C_{i_{opt}} = \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \times C_{i-1}}{R_0 \times C_0}} \times C_{i-1} \times (C_{i+1} + C_{w_i})}$$
$$= \sqrt{C_{i-1} \times C_{i+1}} \times \sqrt{(1 + \frac{C_{w_i}}{C_{i+1}})} \times \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \times C_{i-1}}{R_0 \times C_0}}} \qquad (30)$$

The first part of the resulting expression is similar to the condition described by the LE model for a path of identical gates. The second component expresses the influence of the interconnect capacitance. The last component is related to the resistance of the wire and the difference among the individual logical efforts (types of logic gates) along the path. This expression illustrates the quadratic relationship between the sizes of the neighboring gates. The gate size based on ULE can be determined by solving a set of N polynomial expressions for the N gates along the path.

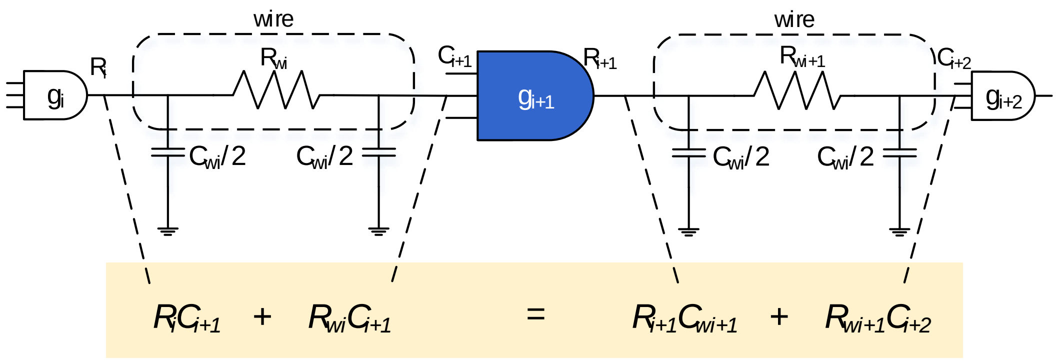


Figure 5 - Delay components in characterizing ULE for long wires.

Later in this thesis we will show how this expression can be further extended in order to include fixed side branches and multiple fan-outs. In order to simplify the solution, a relaxation method has been used. The technique is based on an iterative calculation along the path while applying the optimum conditions. Each capacitance along the path is iteratively replaced by the capacitance determined from applying the optimum expression of the capacitance to two neighboring logic gates.

2.2.4 ULE Optimization in Paths with Branches

As we mentioned earlier, the expression of the optimum input capacitance of each gate based on the ULE model can be further extended to address the general design case where the logic path may include branches or gates with multiple fanout. For instance, consider the circuit shown in Figure 6. The circuit shows the general structure containing a side branch with RC interconnect and/or a fanout load with arbitrary capacitance where R_b and C_b are the resistance and capacitance of branch wires, respectively, and C_f is the fanout load capacitance.

The ULE expression of the total delay of stages i and $i+1$ containing branches and fanout can be written as:

$$\begin{aligned} d &= g_i \times \left[h_i + h_{w_i} + \frac{C_{b1_i} + C_{f1_i}}{C_i} + \frac{C_{b2_i} + C_{f2_i}}{C_i}\right] \\ &+ \frac{R_{w_i}}{\tau} \times \left[0.5 \times C_{w_i} + h_i \times C_i + C_{b2_i} + C_{f2_i}\right] \\ &+ g_{i+1} \times \left[\frac{C_{w_{i+1}} + C_{i+2} + C_{b1_{i+1}} + C_{f1_{i+1}} + C_{b2_{i+1}} + C_{f2_{i+1}}}{h_i \times C_i}\right] \\ &+ \frac{R_{w_{i+1}}}{\tau} \times \left[0.5 \times C_{w_{i+1}} + C_{i+2} + C_{b2_{i+1}} + C_{f2_{i+1}}\right] \end{aligned} \tag{31}$$

where $\tau = R_0 \times C_0$ is the minimum inverter delay. Following the same procedure as in the case with no branches and fan-outs, we equate the derivative of the delay with respect to the gate size to zero, and the optimum expression for the input capacitance of each gate can be written as:

$$C_i = \sqrt{\frac{g_i \times C_{i-1} \times (C_{w_i} + C_{i+1} + C_{b1_i} + C_{f1_i} + C_{b2_i} + C_{f2_i})}{g_{i-1} + \frac{R_{w_{i-1}} \times C_{i-1}}{\tau}}}$$
$$= \sqrt{C_{i-1} \times C_{i+1}} \times \sqrt{1 + \frac{C_{w_i}}{C_{i+1}} + \frac{(C_{b1_i} + C_{f1_i} + C_{b2_i} + C_{f2_i})}{C_{i+1}}} \times \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \times C_{i-1}}{\tau}}} \quad (32)$$

This ULE optimum expression can be generalized for any combination of side branch wires and fanout gates by determining the total effective capacitance of the fanout branches for each stage of the path:

$$C_{BF} = \sum_{1}^{n} C_{b_n} + \sum_{1}^{m} C_{f_m} \quad (33)$$

where n and m are the number of branch wires and fanout gates in a path, respectively. Taking into consideration the last equation, the general ULE optimum expression for the input capacitance is determined [11]:

$$C_i = \sqrt{C_{i-1} \times C_{i+1}} \times \sqrt{1 + \frac{C_{w_i}}{C_{i+1}} + \frac{C_{BF_i}}{C_{i+1}}} \times \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \times C_{i-1}}{\tau}}} \quad (34)$$

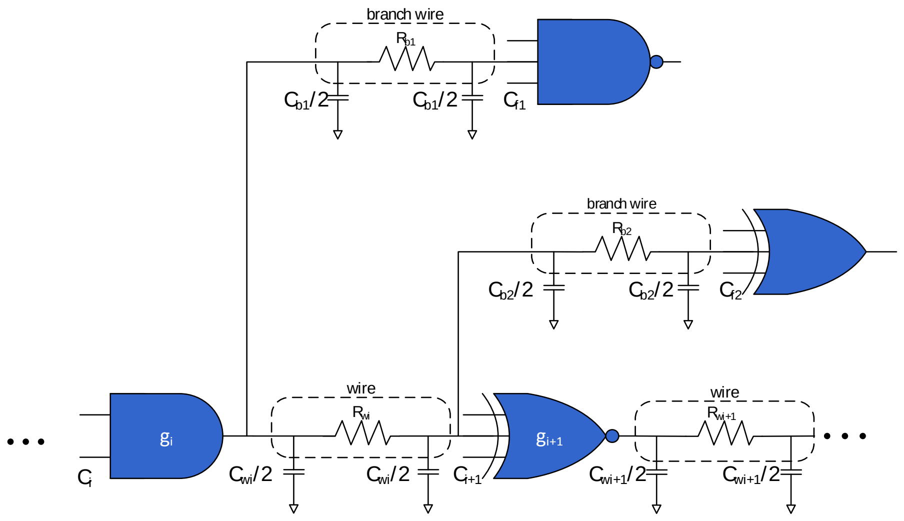


Figure 6 - A logic path segment including RC interconnect and two branches.

In the case of a more complex parasitic tree (see Figure 7), the resistance of a wire, between two adjacent cells, is defined as the sum of all the resistances in the path between the adjacent cells,

$$R_{wi} = \sum R_{i \to i+1} \tag{35}$$

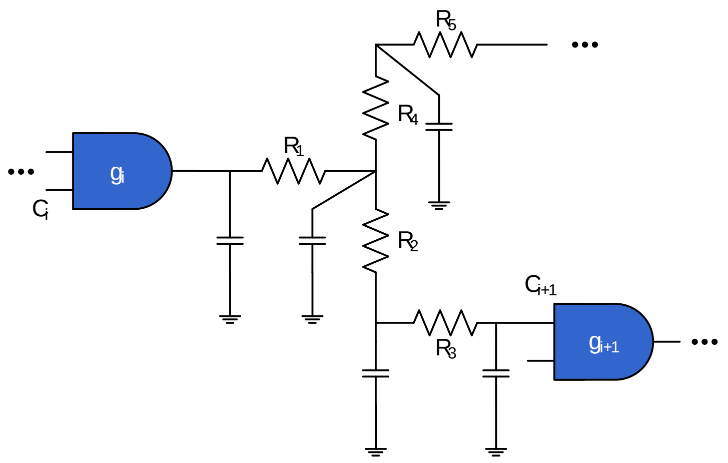


Figure 7 - $R_{wi} = R_1 + R_2 + R_3$.

In order to simplify the solution, a relaxation method is proposed in [11]. The technique is based on an iterative calculation along the path while applying the optimum conditions. Each capacitance along the path is iteratively replaced by the capacitance determined from applying the optimum expressions to two neighboring logic gates. The technique consists of the following steps:

a) *(Initialization)* Set the gate capacitances along the path to arbitrary values (only the first and last values are given).

b) *(Iteration)* Replace each capacitance by the value determined from applying the optimum expressions on two neighboring logic gates

c) *(Stop check)* If any of the new values differ by more than a given precision from the previous value, reiterate step b

The application of the algorithm generally produces the optimal size, converging to 5% accuracy after three iterations. The gates in the last few stages of the path are the first to converge, since the accuracy increases while propagating along the path from the leaf to the root of the path. Consequently, fewer calculations are performed in each successive iteration.

2.2.5 Conclusion

Delay minimization in logic paths with wires is an important issue in the high complexity IC design process. The interconnect is a dominant factor in performance-driven circuits and must be explicitly considered throughout the design process. The characteristics of the wires are not correlated with those of the gates, thereby not permitting the use of the standard logical effort model. In fact, gate sizing in the presence of interconnect does not correspond to equal effort of all of the stages along a path. The ULE method is proposed for delay evaluation and minimization of logic paths with general gates and RC wires. The ULE method provides conditions to achieve minimum delay. Optimal gate sizing in logic paths with wires is achieved when the delay component due to the gate capacitance is equal to the delay component due to the effective resistance of the gate. The ULE method converges to the standard Logical Effort when wire resistance and capacitance are negligible. Gate sizing determined by the proposed ULE method makes ULE suitable for both manual calculations and integration into existing EDA tools.

Chapter 3

Software Architecture

3.1 Introduction

CCSopt is a stand-alone tool, which was built on top of the ULE method in order to achieve high convergence rate to the optimal cell sizes solution. The core of algorithm consists of fast Timing Analysis Incremental engine which evaluates the design's timing information throughout the execution of the algorithm. The inputs of the tool consists of the design's external topology or cell connectivity information (.v file), the cell's internal information such as internal connectivity and delay (.lib file), the parasitic information derived after placing and routing (.spef file), along with a set of instructions for the algorithm (.cfg file). The outputs of the tool consists of the transformed design (.v file) along with the new cells scale factors (.scf file). CCSopt was developed in C++, using OpenMP for multithreading.

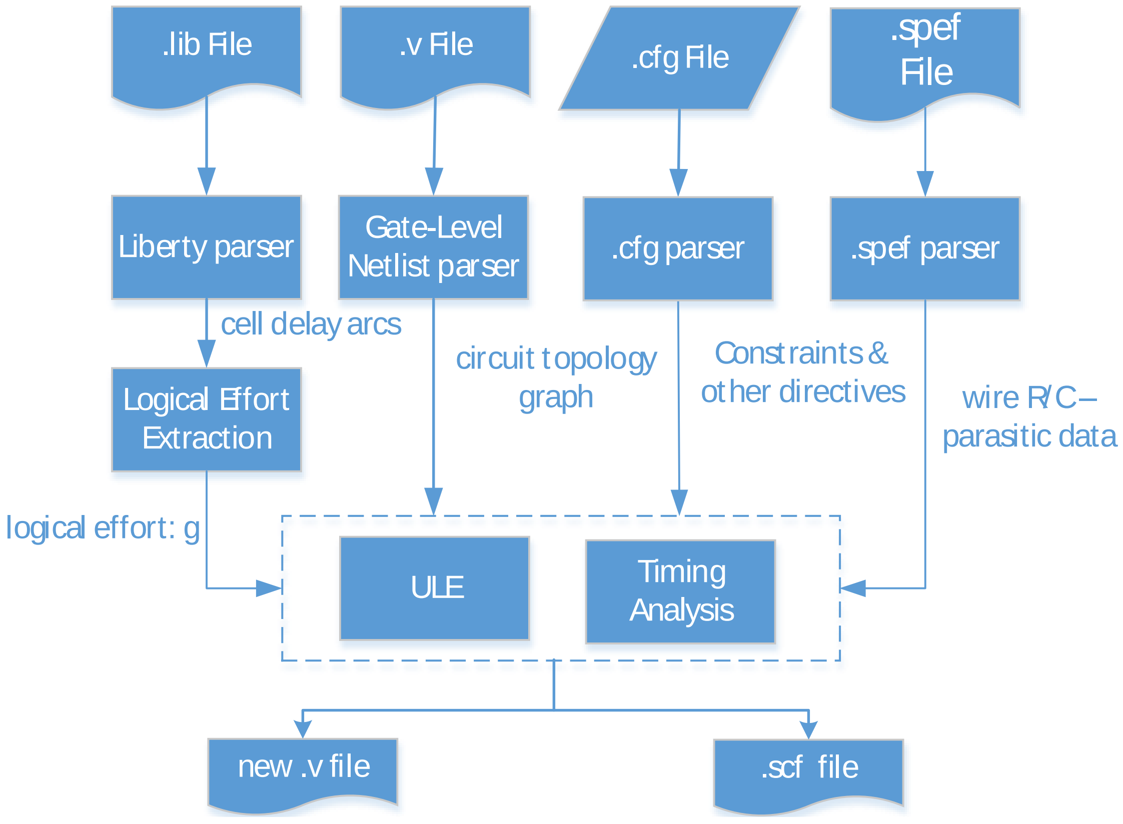


Figure 8 - The software architecture.

3.2 File Formats

In this section, the files, imported and produced, by the tool are described [17]. The .cfg file is available in Appendix A.

3.2.1 Input Verilog (.v)

The Verilog file specifies the top-level hierarchy of the design. For this thesis, we will be using a small set of keywords with the Verilog language. Our Verilog parser supports the set of keywords found within the simple.v file (reproduced below for clarity). It also supports comments that start with '//'. The expected syntax is:

```
module <circuit name> (
        <input 1>,
        …,
        <input n>,
        <output 1>,
        …,
        <output m> );

        input <input 1>;
        …
        input <input n>;
        output <output 1>;
        …
        output <output m>;

        // begin wire definitions
        wire <wire 1>;
        // end wire definitions

        // begin cell definitions
        <cell type> <cell instance name> (.<pin name> (<net name) );
        // end cell definitions
endmodule
```

The expected structure of the Verilog file is to start with the module declaration, defining the interface of the module with name <circuit name>. The inputs and output pins are explicitly declared; the initial wires are optionally declared with the keyword wire. For each cell definition, every <cell type> (.<pin name>) should be a specified cell type (pin) in the library file, and every <cell instance name> and <net name> should be found in the design specification. Each field is considered a string. The following example is from c17.v; its corresponding implementation is shown in Figure 9.

```
01. module c17 (
02.      N1, N2, N3, N6, N7,
03.      N22, N23
04.      );
05.
06.      // Start PIs
07.      input N1, N2, N3, N6, N7;
08.
09.      // Start POs
10.      output N22, N23;
11.
12.      // Start wires
13.      wire N0, N4, N5, N8, N9, N12, N10, N11, N16, N19;
14.
15.      // Start cells
16.      INV_X2 I_5 ( .A(N12), .ZN(N23) );
17.      AND2_X2 NAND2_6 ( .A1(N16), .A2(N19), .ZN(N12) );
18.      INV_X2 I_4 ( .A(N9), .ZN(N22) );
19.      AND2_X2 NAND2_5 ( .A1(N10), .A2(N16), .ZN(N9) );
20.      INV_X2 I_3 ( .A(N8), .ZN(N19) );
21.      AND2_X2 NAND2_4 ( .A1(N11), .A2(N7), .ZN(N8) );
22.      INV_X2 I_2 ( .A(N5), .ZN(N16) );
23.      AND2_X2 NAND2_3 ( .A1(N2), .A2(N11), .ZN(N5) );
24.      INV_X2 I_1 ( .A(N4), .ZN(N11) );
25.      AND2_X2 NAND2_2 ( .A1(N3), .A2(N6), .ZN(N4) );
26.      INV_X2 I_0 ( .A(N0), .ZN(N10) );
27.      AND2_X2 NAND2_1 ( .A1(N1), .A2(N3), .ZN(N0) );
28.
29. endmodule
```

Lines 01 and 29 define the start and end of the specified design with the keywords module and endmodule. Lines 01-04 specify the input and output connection names of the module (note that the direction is not specified here). Line 07 specifies the primary inputs (PIs) of the module with the keyword input. These names must match the ones started with module (lines 01-04). Line 10 specifies the primary output (PO) of the module with the keyword output. This name must match the one stated with the module (lines 01-04). Line 13 specifies the connections or nets within the module with the keyword wire. These connections specify both the external PIs and POs as well as the internal connections between gates (explained further after lines 16-27). Lines 17-27 specify the cells used in the design, as well as how the cells are connected. For example, on line 16, an INV_X2-type cell instance of I_5 is specified, it's A pin is fed by primary input N12, and its ZN pin feeds the primary output N23. On line 27, N1 feeds the A1 pin of the AND2_X2-type cell instance NAND2_1. Line 29 terminates the module definition.

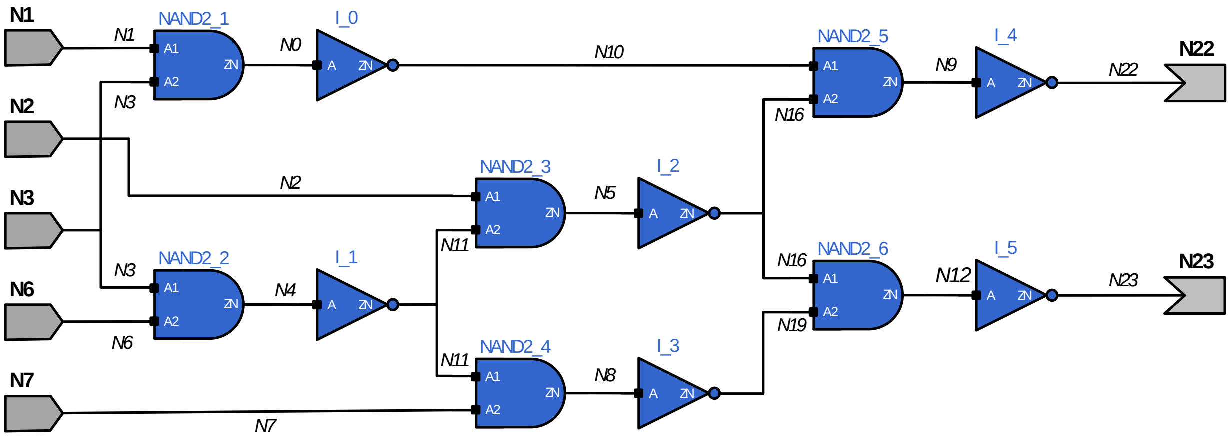


Figure 9 - Implementation of c17.v.

The developed tool can handle multiple Verilog files, in the case that the design is not flat and contains a hierarchy of modules, which may be scattered across different files. For instance a file containing 4 modules, including the top module, s27, is presented,

```
01.  module dff_d(clk, q, d);
02.        input clk, d;
03.        output q;
04.        wire clk, d;
05.        wire q;

06.        DFF_X1 q_reg(.CK (clk), .D (d), .Q (q), .QN ());
07. endmodule
08.
09. module dff_d_4(clk, q, d);
10.       input clk, d;
11.       output q;
12.       wire clk, d;
13.       wire q;
14.       DFF_X1 q_reg(.CK (clk), .D (d), .Q (q), .QN ());
15. endmodule
16.
```

```
17. module dff_d_3(clk, q, d);
18.     input clk, d;
19.     output q;
20.     wire clk, d;
21.     wire q;
22.     DFF_X1 q_reg(.CK(clk), .D (d), .Q (q), .QN ());
23. endmodule
24.
25. module s27(CK, G0, G1, G17, G2, G3,);
26.     input CK, G0, G1, G2, G3;
27.     output G17;
28.
29.     wire CK, G0, G1, G2, G3;
30.     wire G17;
31.     wire G5, G6, G7, G10, G11, G13, n_0, n_1;
32.     wire n_2;
33.
34.     dff_d DFF_0(.d (G10), .clk (CK), .q (G5));
35.     dff_d_4 DFF_1(CK, G6, G11);
36.     dff_d_3 DFF_2(CK, G7, G13);
37.
38.     INV_X32 p1579A(.A (G11), .ZN (G17));
39.     NOR2_X1 p5988A(.A1 (G11), .A2 (n_0), .ZN (G10));
40.     NOR2_X1 p2151D(.A1 (n_2), .A2 (G5), .ZN (G11));
41.     AOI22_X1 p2104A(.A1 (n_1), .A2 (G3), .B1 (n_0), .B2 (G6), .ZN (n_2));
42.     NOR2_X1 p6096A(.A1 (n_1), .A2 (G2), .ZN (G13));
43.     NOR2_X1 p2096A(.A1 (G1), .A2 (G7), .ZN (n_1));
44.     INV_X1 Fp2096A(.A (G0), .ZN (n_0));
45. endmodule
```

Line 34 instantiates the module dff_d, and the arguments are passed in explicit format, where in line 35 the module dff_d_4 is instantiated in implicit format.

The keyword ***assign*** can also be handled along the constants ***1'b0***, ***1'b1***, where the later can be used as wires.

```
assign <wire_name_a> = <wire_name_b>
```

Designs containing busses only in the top level module can also be partial handled (bus operations are not supported). See Appendix A for a more detailed example.

3.2.2 Input Standard Parasitic Exchange Format (.spef)

This file contains the parasitics of a set of nets as a resistive-capacitive (RC) network. If a (e.g. gate-to-gate) connection does not have parasitics, then that connection has 0 delay and the output slew is equivalent to the input slew. Our SPEF parser supports the format specified in simple.spef (see Appendix A) (portions reproduced for clarity). It also supports comments beginning with ‘//’. The format is:

```
// begin header
*SPEF <string>
*DESIGN <string>
*DATE <string>
*VENDOR <string>
*PROGRAM <string>
*VERSION <string>
*DESIGN_FLOW <string>
*DIVIDER <string>
*DELIMITER <string>
*BUS_DELIMITER <string>
*T_UNIT <int> <string>
*C_UNIT <int> <string>
*R_UNIT <int> <string>
*L_UNIT <int> <string>
// end header

// begin nets
// …
// end nets
```

The header describes the general set of units for the file. In this thesis, the DELIMITER field will be set to ‘:’ , the C_UNIT field will be set to one picoFarad (1 PF), and the R_UNIT field will be set to one Ohm (1 OHM). All other fields in the header will not be used. Below shows an example header.

```
01. *SPEF "IEEE 1481-1998"
02. *DESIGN "c17"
03. *DATE "Thu Sep 25 17:47:29 2014"
04. *VENDOR "Cadence Design Systems, Inc."
05. *PROGRAM "Encounter"
06. *VERSION "13.13-s017_1"
07. *DESIGN_FLOW "PIN_CAP NONE" "NAME_SCOPE LOCAL"
08. *DIVIDER /
```

```
09. *DELIMITER :
10. *BUS_DELIMITER []
11. *T_UNIT 1 NS
12. *C_UNIT 1 PF
13. *R_UNIT 1 OHM
14. *L_UNIT 1 HENRY
```

Line 01 specifies the SPEF format date. Line 02 specifies the design name. Line 03 specifies the date at which the file was generated. Line 04 specifies the consumer of this file. Line 05 specifies the tool used to generate the file. Line 06 specifies the version of this file. Line 07 specifies the format in which this file is used. Line 08 specifies the hierarchy divider character. Line 09 specifies the pin divider character. Line 10 specifies the bus delimiter characters. Line 11 specifies the time units for the design. Line 12 specifies the capacitance units for the design. Line 13 specifies the resistance units for the design. Line 14 specifies the inductance units for the design. To reduce file size, SPEF allows long names to be mapped (optional) to shorter numbers preceded by a *. This mapping is defined in the name map section. For example:

```
01. // MMMC spef file for corner 'typ'
02.
03. *NAME_MAP
04.
05. *1 N1
06. *2 N2
07. *3 N3
08. *4 N6
09. *5 N7
10. *6 N22
11. *7 N23
12. *8 N0
13. *9 N4
14. *10 N5
15. *11 N8
16. *12 N9
17. *13 N12
18. *14 N10
19. *15 N11
20. *16 N16
21. *17 N19
22. *18 I_5
23. *19 NAND2_6
24. *20 I_4
25. *21 NAND2_5
26. *22 I_3
27. *23 NAND2_4
```

```
*24 I_2
*25 NAND2_3
*26 I_1
*27 NAND2_2
*28 I_0
*29 NAND2_1
```

Later in the file, N1 can be referred to by its name or by *1. Name mapping in SPEF is not required. Also, mapped and non-mapped names can appear in the same file. Typically, short names such as a pin named A will not be mapped as mapping would not reduce file size. One can write a script that will map the numbers back into names. This will make SPEF easier to read, but greatly increase file size.

After the name map section, each net's parasitics will be defined by the following format:

```
*D_NET <net name> <total net capacitance>
*CONN
<pin type> <pin name> <pin direction>
// more pin definitions
*CAP
<integer label> <pin or node name> <pin or node capacitance>
// more capacitor definitions
*RES
<integer label> <pin or node name> <pin or node name> <pin or node resistance>
// more resistor definitions
*END
```

Each net's definition begins with the keyword *D NET followed by its name and the sum of all the capacitors of the net. The <net name> will be unique for each net. The <total net capacitance> will be a decimal value, and is the sum of all the capacitors defined in the *CAP section. The *CONN keyword describes the set of pins attached to the net. The <pin type> field will either be of type port (*P), which is a primary input or output pin, or internal (*I), which is an internal pin in the design. In this section, only design pins will be referenced – no intermediate SPEF-specific node will be listed. The <pin name> field will be either a primary input, a primary output, have the syntax <cell name>:<cell pin name>, e.g., NAND2_1:A1, or have the syntax <net name>:<int>, e.g., N1:1. The <pin direction> field refers to the pin directional type (not the net), and will be either input (I) or output (O).

The *CAP keyword describes the set of *grounded* capacitors that are in the net. Namely, each capacitor will be connected to a specified node and GND. For each capacitor, the <integer label> is a unique integer that identifies the capacitor *for this net*. The <pin or node name> is a string, and can be a primary input, primary output, a design pin with the syntax <cell name>:<cell pin name>, or an intermediate SPEF-specific node with the syntax <net

name>:<integer>. The <pin or node capacitance> will be a decimal value specifying the capacitance attached to the node. The actual capacitance will be this value multiplied by the C_UNIT value specified in the header. For example, if C_UNIT is 1 PF and <pin or node capacitance> is 1.2, the capacitance is 1.2 *pF*.

The *RES keyword describes the set of resistors in the net. Each resistor connects two pins or nodes (whose format is identical to the *CAP field), and similarly has a unique <integer label>. The <pin or node resistance> is a decimal value; the actual resistance value is this field multiplied by the R_UNIT value specified in the header. For example, if R_UNIT is 1 OHM and <pin or node resistance> is 3.4, then the resistance is 3.4 Ω. The *END keyword indicates the end of the net parasitics. An example net definition is shown below:

```
01.  *D_NET *15 0.000332396
02.  *CONN
03.  *I *23:A1 I *C 4 3 *L 0.00166 *D AND2_X2
04.  *I *26:ZN O *C 4 3 *L 0 *D INV_X2
05.  *I *25:A2 I *C 4 6 *L 0.00173 *D AND2_X2
06.  *CAP
07.  1 *15:0 0.000117155
08.  2 *15:1 0.000134821
09.  3 *15:2 1.83593e-05
10.  4 *15:3 3.06835e-05
11.  5 *23:A1 9.17966e-06
12.  6 *26:ZN 9.17966e-06
13.  7 *15:6 1.30172e-05
14.  *RES
15.  1 *15:6 *25:A2 4
16.  2 *15:3 *15:6 1
17.  3 *15:2 *26:ZN 1.03143
18.  4 *15:2 *23:A1 1.03143
19.  5 *15:1 *15:3 1.35714
20.  6 *15:0 *15:2 4
21.  7 *15:0 *15:1 9
22.  *END
```

Let *R_UNIT and *C_UNIT be the same values as in the header above, i.e., *R_UNIT is 1 OHM and *C_UNIT is 1 PF. Line 01 defines the net *15 (or N11 before name mapping) with a total lumped capacitance of 0.000332396 *pF*. Lines 02-05 define the connectivity of the net *15. Line 03 specifies the internal deisgn pin *23:A1 is an input type. Line 04 specifies the internal design pin *26:ZN in an output type. Line 05 specifies the internal design pin *25:A2 is an input type. Lines 06-13 define the set of capacitors for the net *15. Line 07 specifies capacitor 1 between the SPEF-specific intermediate node *15:0 and GND with a value 0.000117155 *pF*. Line 08 specifies capacitor 2 between the SPEF-specific intermediate node *15:1 and GND with a value 0.000134821 *pF.* Line 09 specifies capacitor 3 between the SPEF-specific intermediate node *15:2 and GND with a value 1.83593e-05 *pF.* Line 10 specifies capacitor 4 between the SPEF-specific intermediate node *15:3 and GND with a value 3.06835e-05 *pF.* Line 11 specifies capacitor 5 between the SPEF-specific intermediate node *23:A1 and GND with a value 9.17966e-06 *pF.* Line 12 specifies capacitor 6 between the SPEF-specific intermediate node *26:ZN and GND with a value 9.17966e-06 *pF.* Line 13 specifies capacitor 7 between the SPEF-specific intermediate node *15:6 and GND with a value 1.30172e-05 *pF.* Lines 14-21 defines the set of resistors of net *15. Line 15 specifies resistor 1 between the SPEF-specific intermediate nodes *15:6 and *25:A2 with a value of 4 Ω. Line 15 specifies resistor 1 between the SPEF-specific intermediate nodes *15:6 and *25:A2 with a value of 4 Ω. Line 16 specifies resistor 2 between the SPEF-specific intermediate nodes *15:3 and *15:6 with a value of 1 Ω. Line 17 specifies resistor 3 between the SPEF-specific intermediate nodes *15:2 and *26:ZN with a value of 1.03143 Ω. Line 18 specifies resistor 4 between the SPEF-specific intermediate nodes *15:2 and *23:A1 with a value of 1.03143 Ω. Line 19 specifies resistor 5 between the SPEF-specific intermediate nodes *15:1 and *15:3 with a value of 4 Ω. Line 20 specifies resistor 6 between the SPEF-specific intermediate nodes *15:0 and *15:2 with a value of 4 Ω. Line 21 specifies resistor 7 between the SPEF-specific intermediate nodes *15:0 and *15:1 with a value of 9 Ω. Line 22 ends the net definition. Figure 10 illustrates the parasitics described above for net *15.

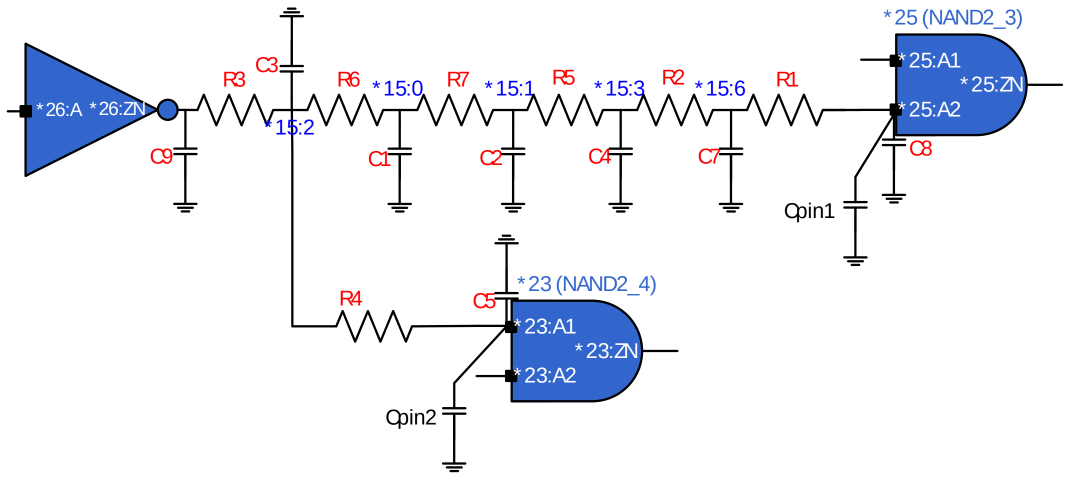


Figure 10 - Parasitics of net *15 (N11). The R (C) labels refer to resistors (capacitors).

3.2.3 Input Liberty (.lib)

This file contains the set of all cells or gates that are available to the design. All cell instances found in the .v file will have corresponding cell type that is located in this file. Gate-level delay and output slew calculations will use the relevant timing information found for the appropriate cell type. For this thesis, we will be using the NanGate 45nm Open Cell Library and the Open Source Liberty parser. The parser supports the full logical (.lib) set of constructs including Composite Current Source (CCS) Modeling Technology, and noise, plus syntax, and common semantic checks.

The relevant portions of the .lib file are explained below. The library consists of (i) a header, (ii) a set of lookup-table definitions, and (iii) a set of cell definitions, where a cell will be a combinational element (e.g., NAND2) or a sequential element (e.g., flip-flop DFF). While there are many keywords available, this thesis will only use the following set. For readability, each syntax set is discussed in separate subsections below.

HEADER. The header sets the general information about the library, and is defined in the NanGate 45nm Open Cell Library with the following format:

```
01. /* Documentation Attributes */
02. date                          : "Thu 10 Feb 2011, 18:11:20";
03. revision                      : "revision 1.0";
04. comment                       : "Copyright (c) 2004-2011 Nangate Inc. All Rights
Reserved.";
05.
06. /* General Attributes */
07. technology                     (cmos);
08. delay_model                   : table_lookup;
09. in_place_swap_mode                      : match_footprint;
10. library_features               report_delay_calculation,report_power_calculation);
11.
12. /* Units Attributes */
13. time_unit                     : "1ns";
14. leakage_power_unit            : "1nW";
15. voltage_unit                  : "1V";
16. current_unit                  : "1mA";
17. pulling_resistance_unit       : "1kohm";
18. capacitive_load_unit           (1,ff);
19.
20. /* Operation Conditions */
21. nom_process                   : 1.00;
22. nom_temperature               : 25.00;
```

```
23. nom_voltage                          : 1.10;
24.
25. voltage_map (VDD,1.10);
26. voltage_map (VSS,0.00);
27.
28. define(process_corner, operating_conditions, string);
29. operating_conditions (typical) {
30.   process_corner          : "TypTyp";
31.   process         : 1.00;
32.   voltage         : 1.10;
33.   temperature             : 25.00;
34.   tree_type       : balanced_tree;
35. }
36. default_operating_conditions : typical;
```

Line 08 specifies the delay model used. Lines 13-18 specify the units in which the values in the .lib file are referenced. Lines 21-23 specify the nominal process, temperature, and voltage at which the library is characterized at. Lines 29-35 specify a set of operating conditions for the "typical" profile. Line 24 sets the default operating conditions of the library. All other lines are being ignored.

LOOKUP TABLES. Most of the cell libraries include table models to specify the delays and timing checks for various timing arcs of the cell. The table models are referred to as NLDM (Non-Linear Delay Model) 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. The lookup table templates are defined as follows:

```
lu_table_template (<table label>) {
  variable_1 : <variable name> ;
  index_1 (<string of data points for variable_1>);
  variable_2 : <variable name> ;
  index_2 (<string of data points for variable_2>);
  ...
}
```

The <table label > and <variable name> fields are considered to be strings, and may or may not be enclosed in "" and "". The string of data points will be a set of integer or double values indicating the index values of the table. The variable and index definition lines can be in any order, e.g., all variable definitions can come before all index definitions. Each <table label> can be referenced in the cell definitions. An example table template looks like:

```
01. lu_table_template (delay_template_3x3) {
02.    variable_1 : input_net_transition;
03.    variable_2 : total_output_net_capacitance;
04.    index_1 ("1000,1001,1002");
05.    index_2 ("1000,1001,1002");
06. }
```

Line 01 and 06 define the table template with label “delay_template_3x3”. Line 02 specifies that variable_1 is the input transition time. Line 03 specifies that variable_2 is the output capacitance. The table values are specified like a nested loop with the first index_1 (line 04) being the outer (or least varying) variable and the second index_2 (line 05) 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:

```
01. lu_table_template(delay_template_3x3) {
02.    variable_1 : input_net_transition;
03.    variable_2 : total_output_net_capacitance;
04.    index_1 ("0.1, 0.3, 0.7");
05.    index_2 ("0.16, 0.35, 1.43");
06. }
```

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 (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 complementary 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. Thus, there are separate two-dimensional tables for computing the output rise and fall transition times of a cell.

CELL DEFINITIONS. A cell specifies a gate that could be used as part of a design, e.g., combinational
gate NAND2 and flip-flop DFF. Its relevant specified syntax in the .lib format is:

```
cell (<cell type>) {
  pin(<pin name>) {
    direction           : <direction> ;
    capacitance                 : <double> ;
    max_capacitance   : <double> ;
    min_capacitance   : <double> ;
    timing() {
      related_pin       : <pin name> ;
      /* combinational or sequential definitions */
    }
    /* other timing() definitions */
  }
  /* other pin definitions */
}
```

In a cell, multiple pins can be defined, e.g., a standard NAND2 will have 3 pins – two inputs and one output. For each pin, the direction field indicates the type of pin: (i) input, (ii) output, or (iii) internal. The capacitance, max capacitance, and min capacitance fields specify the respective pin capacitance, maximum and minimum expected pin loads. A timing() definition creates a timing arc (directed pin-to-pin) inside a cell. The specific syntax is different for a combinational and sequential connection (discussed below). Combinational timing arcs. Combinational arcs propagate delay and output slew from a source pin to a sink pin. They are found in common combinational logic gates, e.g., NAND2 or as a clock-trigger segment in flip-flops. A propagate segment's timing() syntax is:

```
timing() {
  related_pin           : <pin name> ;
  timing_sense                  : <timing sense> ;
  timing_type           : <timing type> ;
  cell_<transition> (<table label>) {
    <table instance> /* omitted for space */
  }
  <transition>_transition(<table label>) {
    <table instance> /* omitted for space */
  }
  /* other cell transition table definitions */
}
```

The related pin is the source of the segment, and the pin (from the pin definition) is the sink of the segment. The timing sense field specifies the transition mode: (i) positive unate, where the source and sink transitions are the same (e.g., rise-to-rise), (ii) negative unate, where the source and sink transitions are opposite (e.g., rise-to-fall), and (iii) non unate, where the source transition has no relation to the sink transition. The timing type field specifies if the arc is combinational, where the unateness is be defined as either positive unate or negative unate, or

<timing type edge> edge, where the unateness is defined as non unate and <timing type edge> is either rising or falling, and refers to the source. The cell <transition> table refers to delay; the <transition> transition table refers to output slew. In both tables, the <transition> refers to the sink of the arc, and is either rise or fall. Note that in the case of (i) positive unate and (ii) negative unate, the direction of the source-to-sink transition is implicitly defined by knowing the unateness and the <transition> transition. For instance, if the arc is negative unate and there exists a table with fall transition, the arc described is a rise-to-fall transition. In the case of non unate, both <timing sense> and <transition> transition must be used, where the former describes the source edge, and the latter describes the sink edge. For example, if <timing sense> is rising edge and there exists a table with fall transition, the arc described is a rise-to-fall transition. The <table label> will be a string that corresponds either (i) to a previously-declared lookup-table template or (ii) be the keyword scalar, indicating that the value stored is a single element (i.e., a 1x1 table). A sample gate is shown below:

```
cell(OR2_X2) {
  pin ("o") {
    direction : output ;
    capacitance : 2.00 ;
    timing() {
      related_pin : "a";
      timing_sense : positive_unate;
      timing_type : combinational;
      cell_fall (scalar) {
        values ("40.00");
      }
      fall_transition (delay_slew_load_6x1) {
        index_1 ("1.050, 2.000, 5.000, 5.500, 9.000, 20.00");
        index_2 ("1.0000");
        values ( \
          "1.050000", \
          "2.000000", \
          "5.000000", \
          "5.500000", \
          "9.000000", \
          "20.000000" \
        );
      }
    }
  }
}
```

Lines 01-26 define the cell OR2 X2. Lines 02-25 define the pin o inside cell OR X2. Line 03 specifies that o is an output pin. Line 04 specifies that the pin capacitance of the cell (for both rise and fall) is 2fF. Lines 05-24 specify a timing arc between source pin a (line 06) and sink pin o. Line 07 specifies that this timing arc is of type positive unate, which propagates the incoming transition to the output transition (i.e., rise-to-rise and fall-to-fall). Lines 09-11 specify that the arc contains a fall transition at the output with a fixed (scalar) delay value of 40ps. Due to the cell fall definition and the positive unate type, this arc is implicitly a fall-to-fall transition. Lines 12-23 specify the output slew table using lookup-table template delay slew load 6x1, with lines 13-22 matching the corresponding table syntax.

3.2.4 Output Files (.v .scf)

The produced files comprise of a verilog file, as described in a previous section, containing the new cell names, after the resizing has taken place, and a file containing the scale factors of the new cells. The output Verilog file will be flatten, which means that if the input Verilog files contained a hierarchy of modules, the output file will contain only the top module which will include all the instantiated cells and nets of the hierarchical modules.

The .scf file defines the scale of the new cells compared to the cell sizes contained in the original design, and the format is defined as,

```
<instance_name_1> <scale_factor_1>
<instance_name_2> <scale_factor_2>
…
<instance_name_n> <scale_factor_n>
```

3.3 Internal Representation

The Verilog files describe the circuit connectivity, in the abstract level of cells and nets. In order to apply the resizing algorithm, that connectivity information must be interpreted and mapped into a graph, which is performed in the parsing module. For example visualizing the circuit c17.v (see Figure 11), in the abstract model of cells and nets will yield the following connectivity map.

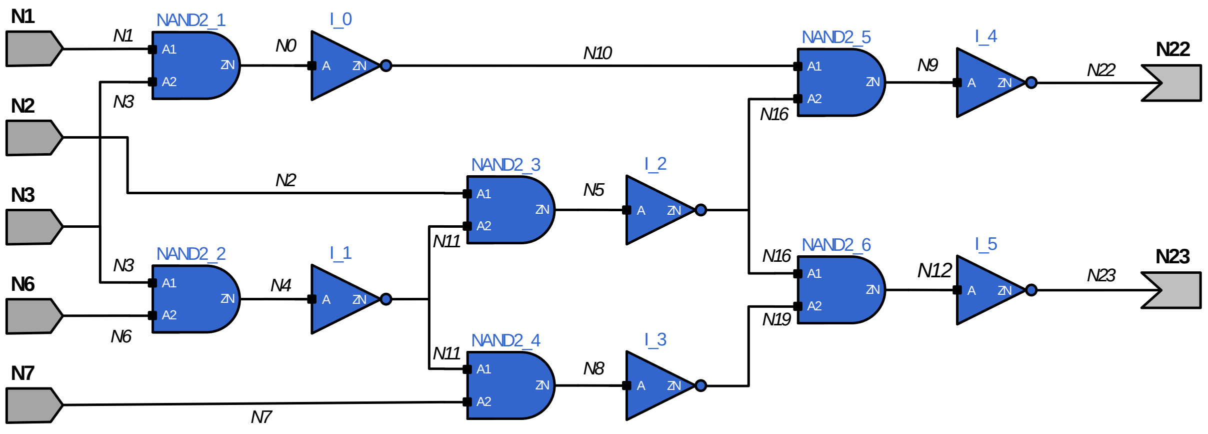


Figure 11 - Implementation of c17.v

Many of the algorithms used in the industry, consists of graph explorations or different graph operations, which means that the connectivity map must be transformed into a simple directed graph representation (nodes/edges).

The nodes of the graph will be the pins of each cell (see Figure 12), and the ports (Primary inputs/Primary outputs). The names of each node must be unique, which means that the name of the pins cannot uniquely define a node. In that case an enhanced name must be used, which uses as a prefix the instance name of the cell, which is already unique. The ports are already unique, but for simplicity reasons, have a prefix of input/output.

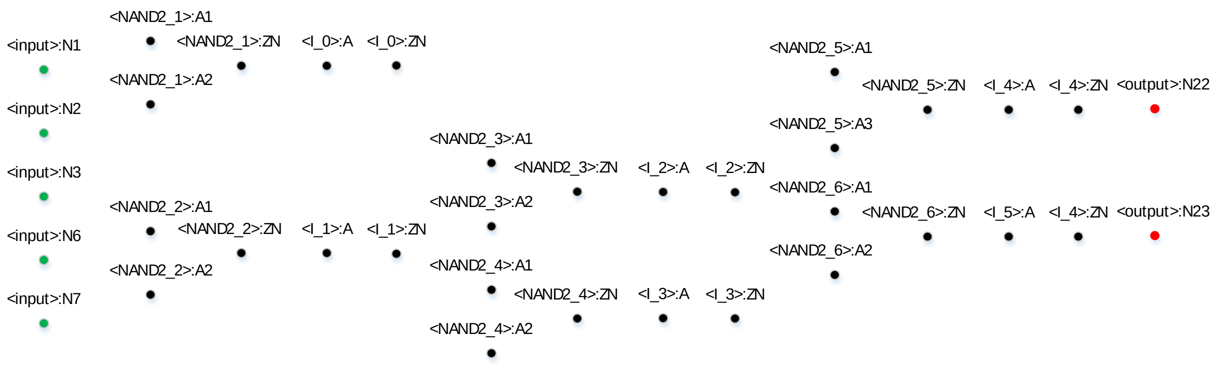


Figure 12 - The nodes of a pin based graph.

The edges of the graph, that describe the connectivity between cells, will be the nets of the circuit (see Figure 13). There is no need for naming the edges, but for simplification the actual name of the nets are used. These edges are described as net edges.

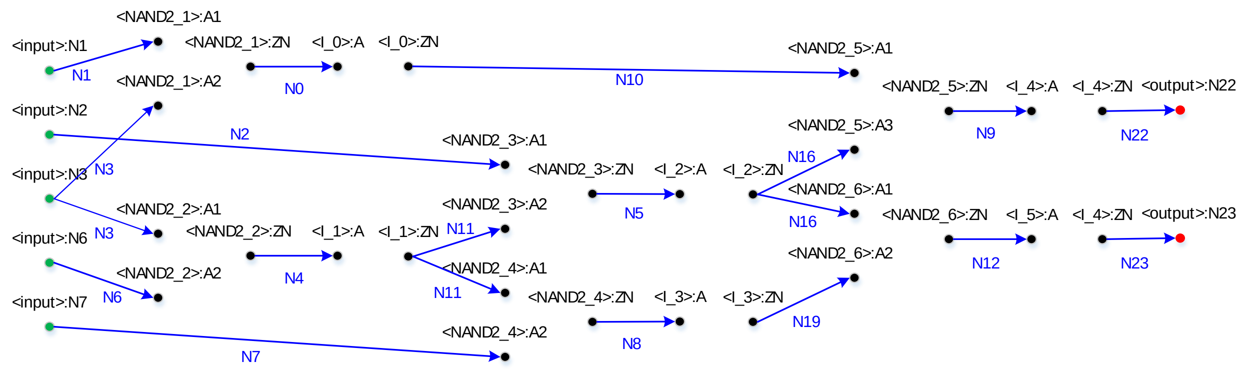


Figure 13 - The graph including the cell connectivity.

The tool cannot extract the total graph of the circuit by only using the Verilog files. The connectivity information inside its cell, is described in the .lib file. Each output of a cell has a set of input related pins, for which the timing information is specified. Those relations specify the internal connectivity of a cell and the edges are described as cell edges (see Figure 14).

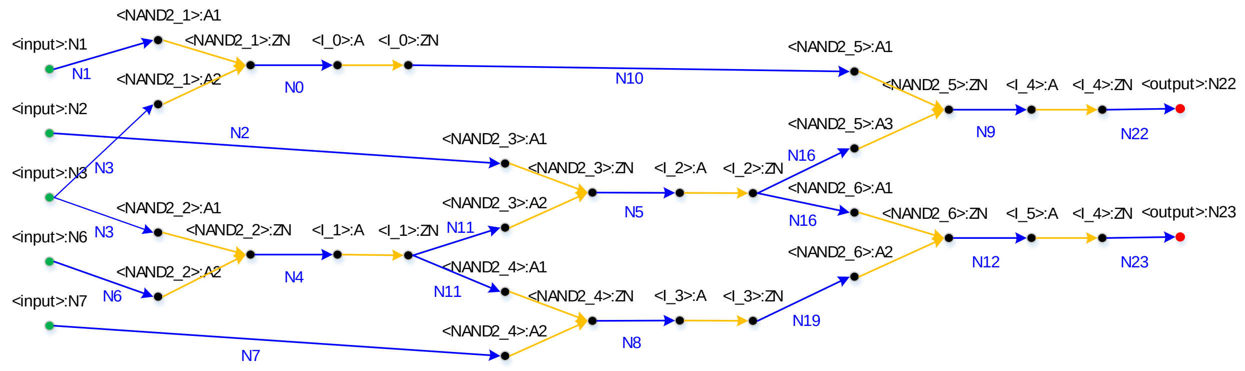


Figure 14 - The graph including the cell internal connectivity.

The internal representation is further enhanced by grouping all the associated pins in the same data structure, which is the cell instance (see Figure 15). The pins are also categorized by its type, input or output.

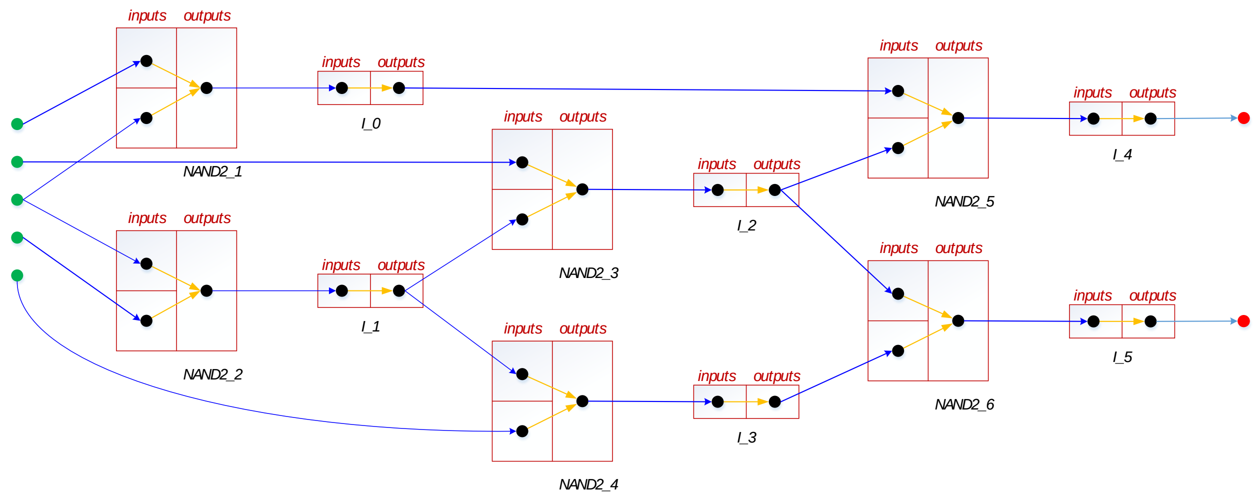


Figure 15 - The final internal representation, including the cell instances.

The nodes of the circuit store the transition and arrival times. The net edges store the wire capacitance, the resistance, and the calculated delay. The cell edges store the timing tables and calculated delay.

3.4 Logical Effort Parameters

3.4.1 Logical effort parameter extraction

There are multiple methods for extracting the logical effort parameter. A good standard cell library (.lib file) contains multiple sizes of each common gate. The sizes are typically labeled with their drive strength [18] . For example, a unit inverter may be called *inv_x1*. An inverter of eight times unit size is called *inv_x8*. A 2-input NAND that delivers the same current as the inverter is called *nand2_x1*. It is often more intuitive to characterize gates by their drive strength, x, rather than their input capacitance. If we redefine a unit inverter to have one unit of input capacitance, then the drive strength of an arbitrary gate is:

$$x = \frac{C_{in}}{g} \qquad (36)$$

In order to redefine the unit inverter, every input capacitance must be scaled by the capacitance of the unit inverter

$$x = \frac{\frac{C_{in}}{C_o}}{g} \qquad (37)$$

Since the drive strength of cell is defined in a standard cell library, the logical effort can be extracted by,

$$g = \frac{\frac{C_{in}}{C_o}}{x} \tag{38}$$

In particular, since every input pin has different capacitance, according to its state (falling, rising), there will exist two logical effort: g values for every input pin (g_{fall}, g_{rise}).

This extraction methodology is used in the resizing algorithm.

An alternative method, when using a standard cell library, is to the extract logical effort of gates directly from the delay timing tables.

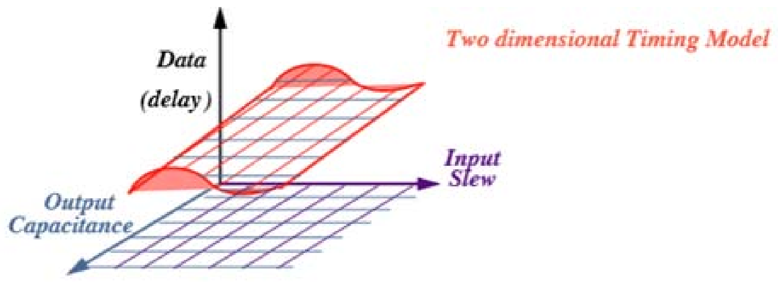
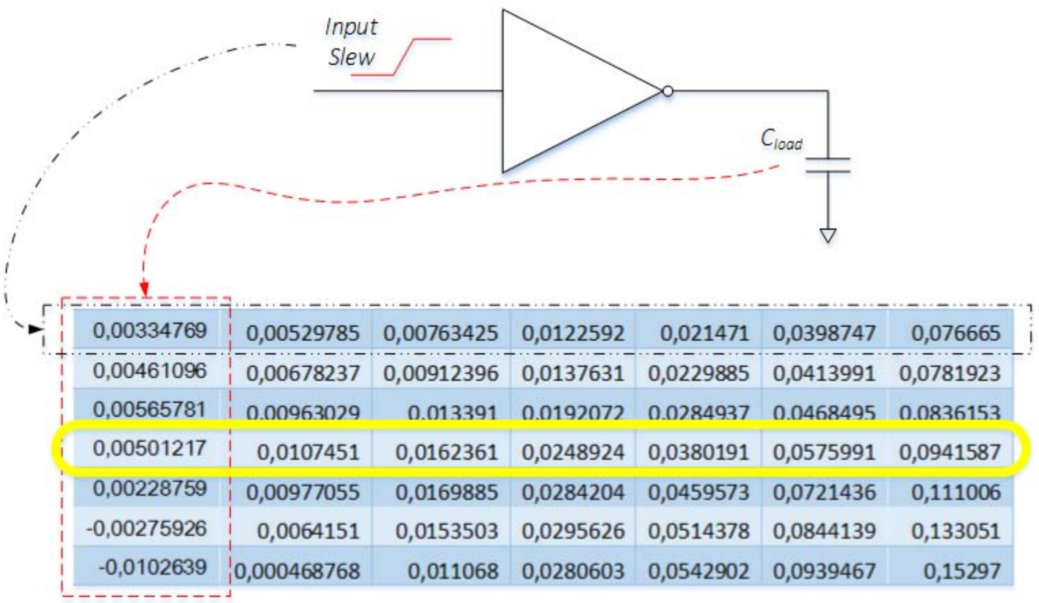


Figure 16 - Three dimensional plot of the timing data.

Using the logical effort delay expression, $d = g \times h + p$, where $h = \frac{C_{load}}{C_{in}}$, linear regression can be performed in order to extract g and p, in the NLDM tables.

For example linear regression on cell INV_X1 can be performed as,



0,00334769	0,00529785	0,00763425	0,0122592	0,021471	0,0398747	0,076665
0,00461096	0,00678237	0,00912396	0,0137631	0,0229885	0,0413991	0,0781923
0,00565781	0,00963029	0,013391	0,0192072	0,0284937	0,0468495	0,0836153
0,00501217	0,0107451	0,0162361	0,0248924	0,0380191	0,0575991	0,0941587
0,00228759	0,00977055	0,0169885	0,0284204	0,0459573	0,0721436	0,111006
-0,00275926	0,0064151	0,0153503	0,0295626	0,0514378	0,0844139	0,133051
-0,0102639	0,000468768	0,011068	0,0280603	0,0542902	0,0939467	0,15297

Figure 17 - NanGate's x1 inverter.

Where the input transitions are,

0,00117378	0,00472397	0,0171859	0,0409838	0,0780596	0,130081	0,198535

And the capacitance loads are,

0,365616	1,89781	3,79562	7,59125	15,1825	30,365	60,73

Since there exist seven input transition values, the regression will produce seven lines (seven different pairs of g, p). By simply averaging all the g, and all the p, the algorithm produces the final line.

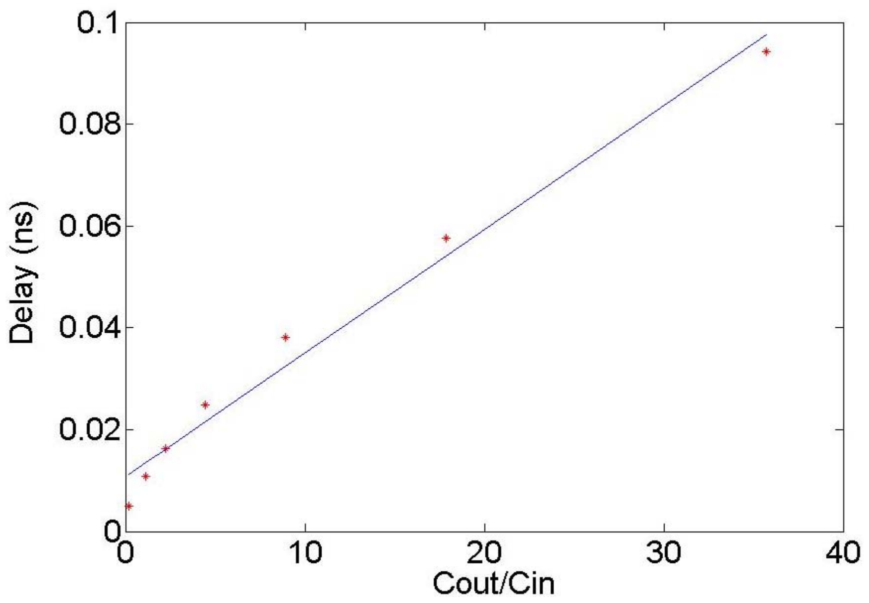


Figure 18 - The linear regression output for the highlighted yellow data.

3.4.2 Unit Inverter

The unit inverter is a cell, which belongs to the standard cell library and it is usually an inverter. That cell in needed in order to normalize the input capacitance of every cell, for the logical effort parameter. The unit inverter capacitance is defined as,

$$C_o = \frac{C_{fall}^{un\_inv} + C_{rise}^{un\_inv}}{2} \tag{39}$$

Where C_{fall}, and C_{rise} are the input capacitance values as described in the .lib file.

The ULE method requires the calculation of another constant, the unit inverter delay (τ), which is defined as,

$$\tau = R_o \times C_o \tag{40}$$

The lib file does not provide any information regarding the resistance of cell, so in order to calculate the unit inverter delay, the delay timing tables must be used. Assuming zero slew as the input of the unit inverter, and another unit inverter as the driving cell, bilinear interpolation can be performed in order to acquire the delay. The interpolation method will be described further, in the cell delay calculation part of the Static Timing Analysis (STA) chapter.

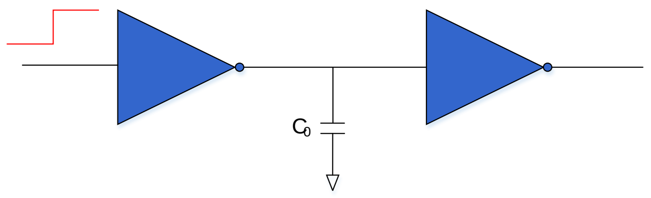


Figure 19 - A unit inverter driving another unit inverter for zero slew.

In the same context τ is defined as,

$$\tau = \frac{d_{fall} + d_{rise}}{2} \tag{41}$$

Where d_{fall}, and d_{rise} are the delays, as calculated using bilinear interpolation.

3.5 Timing Analysis

3.5.1 Introduction

The resizing algorithm performs the ULE method for delay evaluation and minimization in paths composed of CMOS logic gates. Each path, that will be evaluated, can be chosen arbitrarily from a pool of paths, but this technique will not yield the best results. A choosing criterion must be used in order to select the most suitable path every time. That criterion would ideally be a combination of metrics, which is power consumption and delay. The developed algorithm only uses the delay as a filter when examining paths.

Timing analysis must be performed in order to sort the available paths according to their criticality, that is extracting the paths with the worst or best delay (Late / Early timing analysis). Since the resizing algorithm is very conservative, the timing analysis will only be performed for the worst delay paths. There are different approaches for timing analysis, such as static timing analysis, dynamic timing analysis, statistical timing analysis, and each of them uses a different methodology to analyze the circuit.

Static timing analysis verifies circuit timing by "adding up propagation delays along paths between clocked elements" in a circuit. It checks the delays along each path against the specified timing constraints for each circuit path and reports any existing timing violations.

Dynamic timing analysis verifies circuit timing by applying test vectors to the circuit. This approach is an extension of simulation and ensures that circuit timing is tested in its functional context. This method reports timing errors that functionally exist in the circuit and avoids reporting errors that occur in unused circuit paths.

Statistical timing analysis is a variation of the static timing analysis which replaces the normal deterministic timing of gates and interconnects with probability distributions, and gives a distribution of possible circuit outcomes rather than a single outcome.

Every method has its own advantages and disadvantages, but for the purposes of the resizing algorithm, static timing analysis was selected, to provide an estimate of the worst paths, with the least algorithm complexity.

3.5.2 Static Timing Analysis

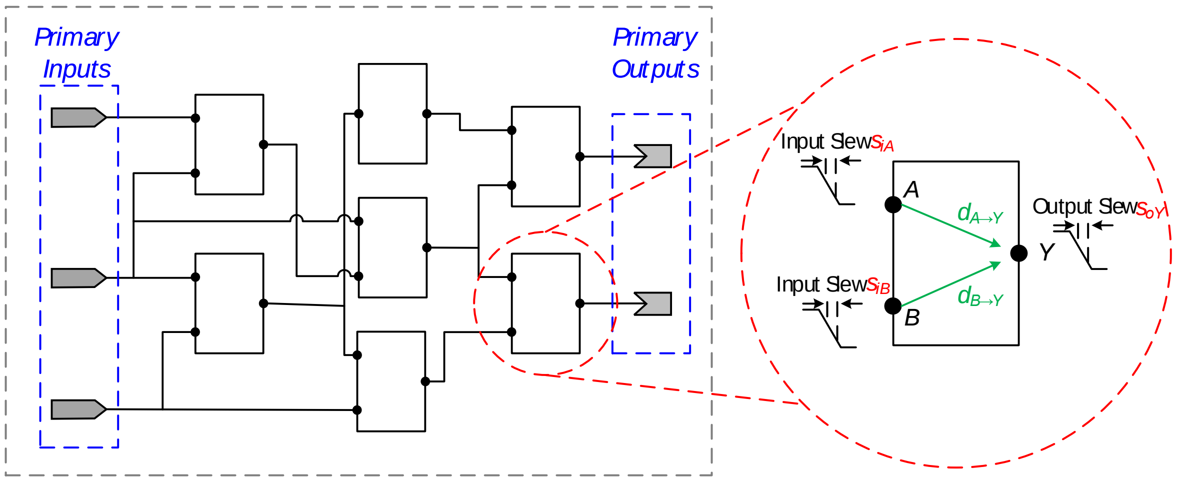


Figure 20 - Slews and delays in a circuit element.

Static timing analysis [18] [19] computes the amount of time required for signals to propagate in a circuit from its **primary inputs (PIs)** to its **primary outputs (POs)** through various circuit elements and interconnect. Signals arriving at an input of an element will be available at its output(s) at some later time; each element therefore introduces a delay during signal propagation. Furthermore, signal transitions are characterized by their input slew and their output slew, which is defined as the amount of time required for a signal to transition from *high to low* or *low to high*. For example, the delay across the circuit element from input A to output Y is designated be $d_{A \to Y}$, the input slew at A by S_{iA}, and the output slew at Y by S_{oY} (see Figure 20). Both the delay and the output slew are functions of input slew.

Starting from the primary inputs, we quantify the instant that a signal reaches an input or output of a circuit element as the *arrival time* (at). To account for multiple sources of within-chip variation, such as manufacturing variations, temperature fluctuation, voltage drops, and electro-migration, the timing analysis is typically performed using an early / late split, which each circuit node has an early (lower) bound and a late (upper) bound on its time. The early / late mode is applied in both the arrival times and slews.

The arrival time propagation is performed by starting from the primary inputs, and the arrival times are computed by adding delays across a path, and performing the minimum (in early mode) or maximum (in late mode) of such accumulated times at a merge point, which is the output of a circuit element.

In early mode, we are concerned with computing the earliest time instant that a signal transition can reach any given circuit node. For example, let at_A^{early} and at_B^{early} to be the early arrival times at pins A and B. Then the early mode arrival time at the output pin Y will be

$$at_Y^{early} = min(at_A^{early} + d_{A \to Y}^{early}\ , at_B^{early} + d_{B \to Y}^{early}\ ) \tag{42}$$

Conversely, in late mode, we are concerned with computing the latest time instant that a signal transition can reach any given circuit node. Following the same example, the late mode arrival time at output pin Y will be:

$$at_Y^{late} = max(at_A^{late} + d_{A \to Y}^{late}\ , at_B^{late} + d_{B \to Y}^{late}\ ) \tag{43}$$

The delays that are accumulated in a path, fall into two categories; delays introduced by the circuit elements and delays introduced by the interconnect.

The slew propagation is performed by starting from the primary inputs, but in this time, the values are not accumulated, but instead they are changed based on some function. Similar to the arrival time propagation, when reaching at a merge point, the minimum (in early mode) or maximum (in late mode) must be performed, in order to propagate the best/worst slew. Following the same example, the early mode and late mode slew at output pin Y are, respectively:

$$S_{OY}^{early} = min(f_{A \to Y}(S_{iA}^{early}), f_{B \to Y}(S_{iB}^{early})) \tag{44}$$

$$S_{OY}^{late} = max(f_{A\to Y}(S_{iA}^{late}), f_{B\to Y}(S_{iB}^{late})) \tag{45}$$

Typically the function of the slew is different in early and late mode, but in our case, the developed algorithm uses the same function.

The functions of slew fall into two categories; functions of slew for the circuit elements, and functions of slew for the interconnect.

The values of slew and arrival times at the primary inputs can be initialized to arbitrary values. If the design contains constants (1'b0, 1'b1), which are handled as virtual primary input wires, their slews and arrival times will be initialized to 0.

3.5.3 Functions of delay and slew for cells

The .lib file provides the functions for delay and slew for each circuit element, in fall and rise state, that can be used for a valid design. Each cell has a function for every timing arc, and sometimes has multiple functions for even the same timing arc. The functions have two variables, which are input slew and total output capacitance, and usually are constructed as two-dimensional tables. The values on the tables are either the slew on the output of a pin, or the delay for the given timing arc. The tables basically contain some corner cases, which were extracted over simulations, and the model used is the NLDM. In order to obtain the value of the function, given the input slew and the total output capacitance, as inputs, bilinear interpolation needs to be performed when the inputs belong in the domain of the variables. When any of inputs lie outside the domain of the function, bilinear extrapolation needs to be performed.

If a table lookup is required for (x_0, y_0), the lookup value is obtained by interpolation, which is given by:

$$T_{00} = x_{20} * y_{20} * T_{11} + x_{20} * y_{01} * T_{12} + x_{01} * y_{20} * T_{21} + x_{01} * y_{01} * T_{22} \tag{46}$$

where,

$$x_{01} = \frac{x_0 - x_1}{x_2 - x_1} \quad x_{20} = \frac{x_2 - x_0}{x_2 - x_1} \quad y_{01} = \frac{y_0 - y_1}{y_2 - y_1} \quad y_{20} = \frac{y_2 - y_0}{y_2 - y_1} \tag{47}$$

x_1, x_2, y_1, y_2 belong to the domain of variables (input transition, total output capacitance), with $x_1 < x_0 < x_2$ and $y_1 < y_0 < y_2$. T_{ij} are the corresponding values in the two-dimensional array.

	y_1	y_2	y_3	…	y_{n-2}	y_{n-1}	y_n
x_1	$T_{1,1}$	$T_{1,2}$	$T_{1,3}$	…	$T_{1,n-2}$	$T_{1,n-1}$	$T_{1,n}$
x_2	$T_{2,1}$	$T_{2,2}$	$T_{2,3}$	…	$T_{2,n-2}$	$T_{2,n-1}$	$T_{2,n}$
x_3	$T_{3,1}$	$T_{3,2}$	$T_{3,3}$	…	$T_{3,n-2}$	$T_{3,n-1}$	$T_{3,n}$
…	…	…	…	…	…	…	…
x_{m-2}	$T_{m-2,1}$	$T_{m-2,2}$	$T_{m-2,3}$	…	$T_{m-2,n-2}$	$T_{m-2,n-1}$	$T_{m-2,n}$
x_{m-1}	$T_{m-1,1}$	$T_{m-1,2}$	$T_{m-1,3}$	…	$T_{m-1,n-2}$	$T_{m-1,n-1}$	$T_{m-1,n}$
x_m	$T_{m,1}$	$T_{m,2}$	$T_{m,3}$	…	$T_{m,n-2}$	$T_{m,n-1}$	$T_{m,n}$

Table 2 - Two-dimensional NLDM table

The above equation is valid both for interpolation and extrapolation.

In the case of multiple tables per timing arc in the same cell, the algorithm calculates the worst case which is the maximum (in late mode) and the minimum (in early mode).

Consider the table lookup for fall transition 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"
                   . . .
```

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

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

Total output capacitance

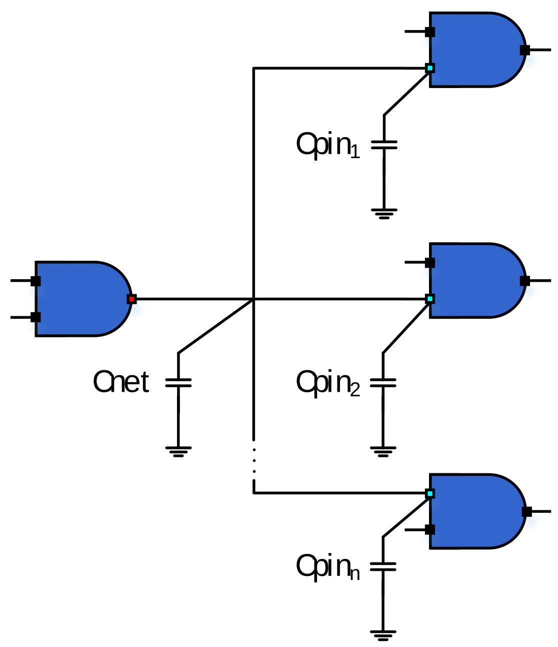


Figure 21 - Output capacitance.

The total output capacitance of a cell is defined as:

$$C_{total} = C_{net} + \sum_{i=1}^{n} C_{pin_i} \tag{48}$$

The net capacitance is the total capacitance for a given net, as defined in the .spef file.

Timing arcs and unateness

Every cell has multiple timing arcs. For example, a combinational logic cell, such as AND, OR, NAND, NOR cell, has timing arcs from each input to each output of the cell. 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) (see Figure 22). 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 falling transition on an input causes the output to have a rising transition (or not to change) (see Figure 23). 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 a xor cell (exclusive-or) are non-unate. In the case of non-unate three sub cases are specified, when considering the timing type field in the lib file. In the case of rising edge and falling edge, the output transition can be specified by the input transition (see Figure 24, Figure 25). In the case of combinational edge, every combination of input transition must be taken into account, so the output transition will be computed as the worst/best combination (maximum for late mode and minimum for early mode) (see Figure 26).

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.

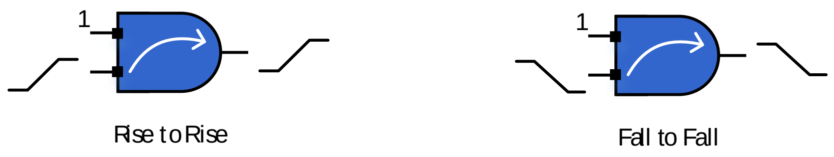


Figure 22 - Positive Unate.

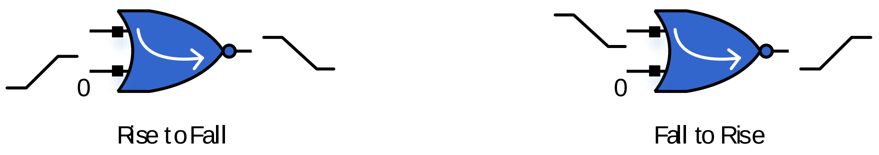


Figure 23 - Negative Unate.

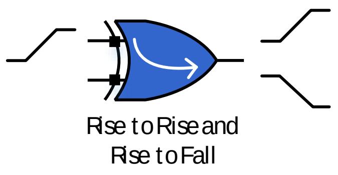


Figure 24 - Non Unate, (Rising Edge).

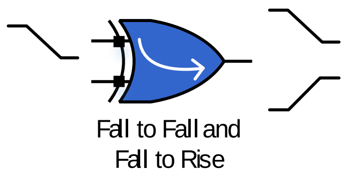


Figure 25 - Non Unate, (Falling Edge).

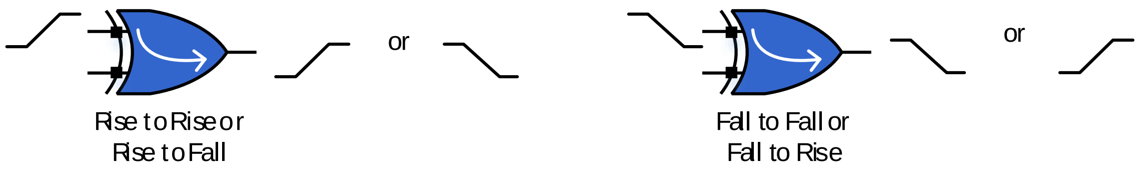


Figure 26 - Non Unate, (Combinational Edge).

In that context, the interpolation for the output slew of a circuit element will be performed as,

```
if timing_sense is POSITIVE_UNATE
{
    O_tr_r = interpolate (transition_rise, I_tr_r, C_out_r)
    O_tr_f = interpolate (transition_fall, I_tr_f, C_out_f)
}
else if timing_sense is NEGATIVE_UNATE
{
    O_tr_r = interpolate (transition_rise, I_tr_f, C_out_r)
    O_tr_f = interpolate (transition_fall, I_tr_r, C_out_f)
}
else if timing_sense is NON_UNATE
{
    if timing_type is FALLING_EDGE
    {
        O_tr_r = interpolate (transition_rise, I_tr_f, C_out_r)
        O_tr_f = interpolate (transition_fall, I_tr_f, C_out_f)
    }
    else if timing_type is RISING_EDGE
        O_tr_r = interpolate (transition_rise, I_tr_r, C_out_r)
        O_tr_f = interpolate (transition_fall, I_tr_r, C_out_f)
    }
    else if timing_type is COMBINATIONAL
    {
        tr_r_1 = interpolate (transition_rise, I_tr_r, C_out_r)
        tr_r_2 = interpolate (transition_rise, I_tr_f, C_out_r)
        O_tr_r = compare (tr_r_1, tr_r_2)

        tr_f_1 = interpolate (transition_fall, I_tr_r, C_out_f)
        tr_f_2 = interpolate (transition_fall, I_tr_f, C_out_f)
        O_tr_f = compare (tr_f_1, tr_f_2)
    }
}
```

Algorithm 1 - Output slew calculation.

where I_tr_r, I_tr_f are the slews at the input of a cell, O_tr_r, O_tr_f are the slews at the output of a cell, and C_out_f, C_out_r are the total output capacitances at the output of a cell. In the case of early and late mode every slew will be calculated twice. For instance at an input of cell there will exist 4 slew values, I_tr_r_early, I_tr_f_early, I_tr_r_late, I_tr_f_late. The delay for each timing arc is calculated exactly the same way as slew, with the difference that the cell_rise/cell_fall tables are used in the interpolation instead of transition_rise/transition_fall.

3.5.4 Functions of delay and slew for interconnect

The .spef file provides the resistances and capacitances, in segments, for each net found in the circuit, so the delay of each net (port to taps) can be calculated once using the elmore delay model. In that context the function of delay is just a constant value. The function of slew at any given tap T can be approximated by a two-step process. First, the output slew of the impulse response is computed, which can be well-approximated by,

$$\hat{S}_{oT} \approx \sqrt{2\beta_T - d_T^2} \tag{49}$$

Where β_T is the second moment of the input response at node T, and d_T is the corresponding elmore delay. Second, the slew of the response to the input ramp is computed by the expression:

$$S_{oT} \approx \sqrt{s_i^2 + \hat{S}_{oT}^2} \tag{50}$$

where s_i is the input slew (see Figure 27).

The unateness of the interconnect is defined as positive unate, that means if a signal is rising/falling, it will not change through the interconnect.

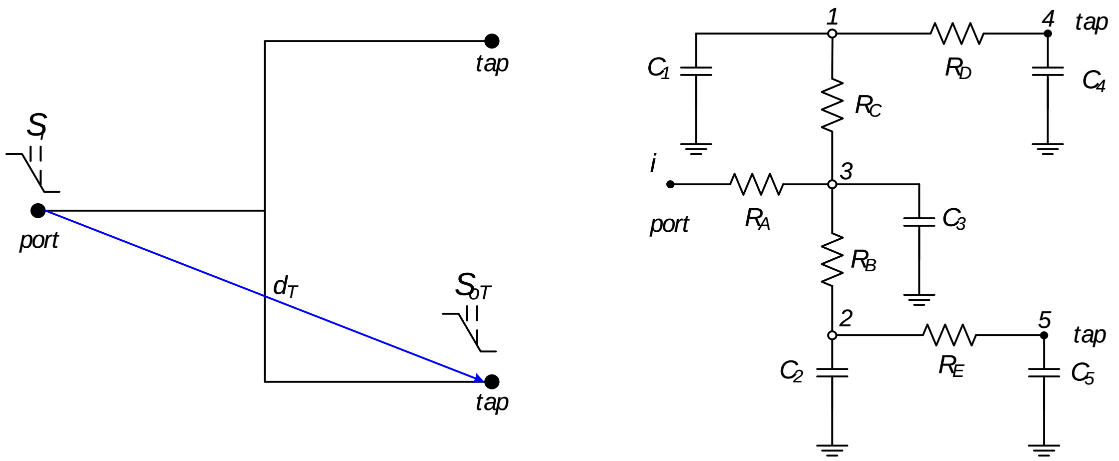


Figure 27 - Interconnect.

Elmore delay model

Elmore delays are applicable for RC trees, which meet the following three conditions. They have a single input (port) node, they do not contain any resistive loops, every capacitance is connected between a node and ground (lumped capacitance). 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.

The downstream capacitance is defined as sum of the enclosed capacitances, under the same color, including the capacitance of the node with the same color (see Figure 28).

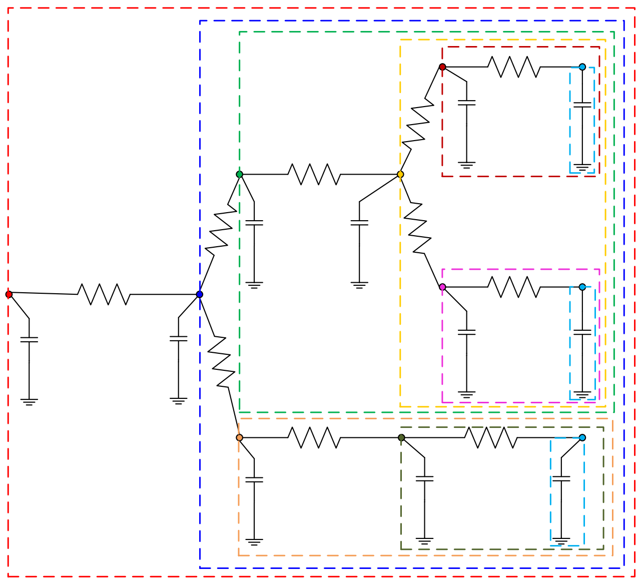


Figure 28 - Downstream capacitance.

The Elmore delay at node e is defined as,

$$d_e = \sum_{k \in N} R_{k \to e} C_k \qquad (51)$$

where N is the set of all nodes in the RC network.

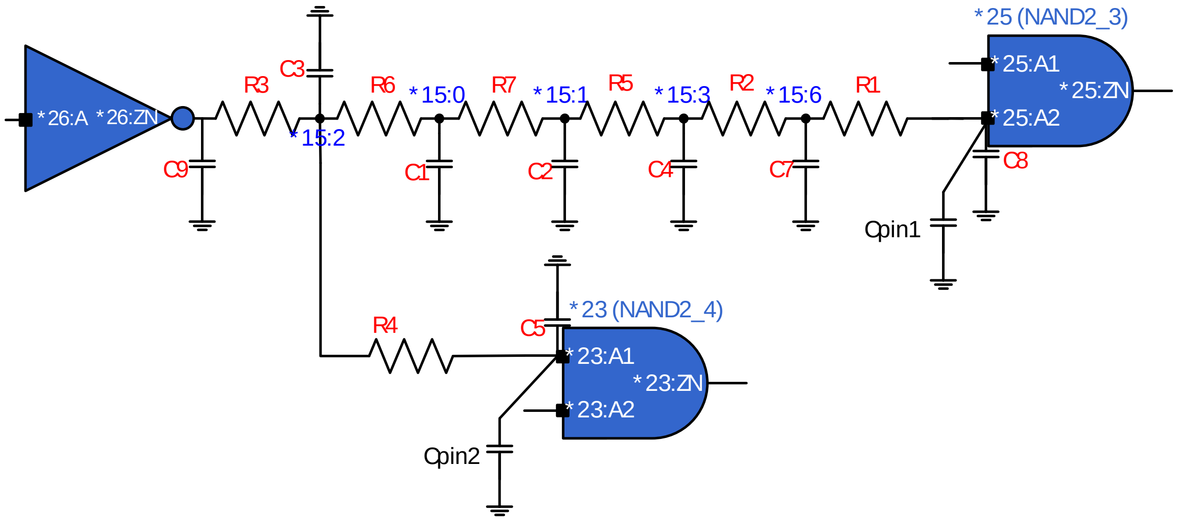


Figure 29 - Parasitics of the net *15 (N11).

For instance the delays at taps (see Figure 29) *25:A2 and *23:A1 of the net *15 (N11) can be calculated as,

$$\begin{aligned} d_{*25:A2} = {} & R_3 * (C_3 + C_1 + C_2 + C_4 + C_7 + C_8 + C_{pin1} + C_5 + C_{pin2}) \\ & + R_6 * (C_1 + C_2 + C_4 + C_7 + C_8 + C_{pin1}) \\ & + R_7 * (C_2 + C_4 + C_7 + C_8 + C_{pin1}) \\ & + R_5 * (C_4 + C_7 + C_8 + C_{pin1}) \\ & + R_2 * (C_7 + C_8 + C_{pin1}) \\ & + R_1 * (C_8 + C_{pin1}) \end{aligned}$$

$$\begin{aligned} d_{*23:A1} = {} & R_3 * (C_3 + C_1 + C_2 + C_4 + C_7 + C_8 + C_{pin1} + C_5 + C_{pin2}) \\ & + R_4 * (C_5 + C_{pin2}) \end{aligned}$$

The second moment of the input response (β_T) at node n can be calculated as:

$$\beta_T = \sum_{k \in N} R_{k \to T} C_k d_k \tag{52}$$

where d_k is the Elmore delay.

For instance the beta factors at taps *25:A2 and *23:A1 of the net *15 (N11) can be calculated as,

$$\begin{aligned}\beta_{*25:A2} = {} & R_3 * (C_3 * d_3 + C_1 * d_1 + C_2 * d_2 + C_4 * d_4 + C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2} + [C_5 \\ & + C_{pin2}] * d_{*23:A1}) \\ & + R_6 * (C_1 * d_1 + C_2 * d_2 + C_4 * d_4 + C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2}) \\ & + R_7 * (C_2 * d_2 + C_4 * d_4 + C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2}) \\ & + R_5 * (C_4 * d_4 + C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2}) \\ & + R_2 * (C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2}) \\ & + R_1 * ([C_8 + C_{pin1}] * d_{*25:A2})\end{aligned}$$

$$\begin{aligned}\beta_{*23:A1} = {} & R_3 * (C_3 * d_3 + C_1 * d_1 + C_2 * d_2 + C_4 * d_4 + C_7 * d_7 + [C_8 + C_{pin1}] * d_{*25:A2} + [C_5 + \\ & C_{pin2}] * d_{*23:A1}) \\ & + R_4 * ([C_5 + C_{pin2}] * d_{*23:A1})\end{aligned}$$

The developed algorithm in the resizing tool, traverses the tree in post order (see Appendix B), in order to accumulate the downstream capacitance at every node, and then traverses the tree using depth first traversal (see Appendix B) in order to calculate the elmore delay at each node. During the depth first traversal, the sum of $C_i * d_i$ is also accumulated at each node. The process is finalized with a second depth first traversal, which calculates the beta factors at every tap.

Parasitics with coupling capacitances and resistive loops

Ideally all the nets of a design must be trees, but sometimes that is not the case. Spef can contain capacitances between two nodes (coupling capacitances) along with lumped capacitances. The resistance network for a net can be very complex. Spef can contain resistor loops or seemingly ridiculously huge resistors even if the layout is a simple point to point route (see Figure 30). This is due how the extraction tool cuts nets into tiny pieces for extraction and then mathematically stitches them back together when writing spef.

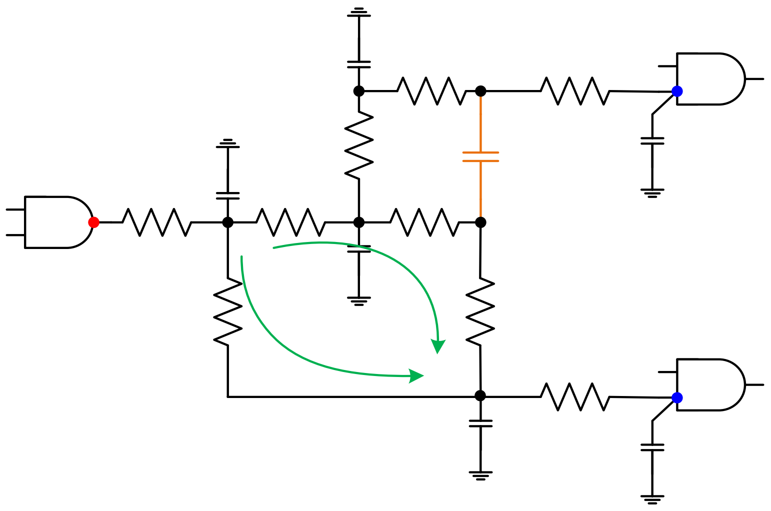


Figure 30 - Parasitics containing resistor loop and coupling capacitance.

The elimination of coupling capacitances is performed while parsing the .spef file, but the elimination of resistor loops requires the creation of the spanning tree.

First a loop detection algorithm takes places during the first post order traversal, and if a loop is discovered the elmore delay calculation must be stopped. At this point a spanning tree algorithm needs to be performed in order to reduce the graph into a tree. Since the whole analysis is pessimistic, only the largest resistances need to be part of the final tree. Kruskal's maximum spanning tree algorithm is deployed, which sorts all the resistances of a given net into descending order, and then every resistance is inserted into the final tree if it does not create a loop. The loop detection step in Kruskal's algorithm is implemented with disjoint-set data structure (union–find data structure). After the spanning tree step, the elmore delay algorithm needs to be perform again from the beginning. (see Appendix B).

3.5.5 Path delay

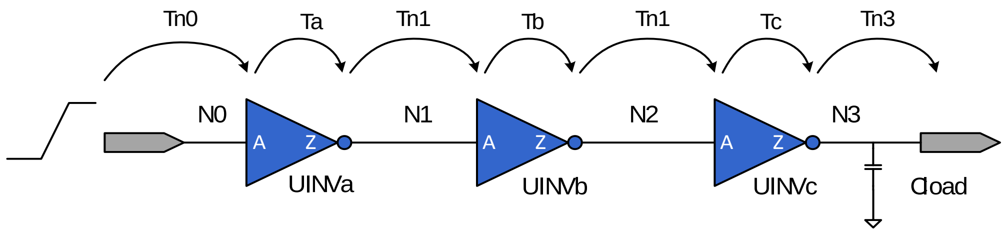


Figure 31 - Delays across a path.

Consider the three inverters in series (see Figure 31); every inverter has negative unate. 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 elmore delay model as specified in the previous section. The same delay model is also used in determining the delay, Tn0, for net N0. The total capacitance at the output UINVa/Z is obtained based upon the net load at the output of UINVa, and the capacitance of the pin UINVb/A. The transition time at input UINVa/A and the total load at output UINVa/Z is then used to obtain the cell output fall delay. The elmore delay model at pin UINVa/Z is used to determine the transition time at pin UINVb/A and 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 total 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 the case of asynchronous timing arcs, like preset or clear arcs, which appear in some sequential elements, the arrival time propagation can be disabled through those aforementioned arcs, if static timing analysis is done in a synchronous mode.

3.5.6 Static Timing Analysis Implementation

The basic requirement to correctly calculate the slew at the output of a cell and the delay of a cell is, that every cell needs to be processed when the slew has arrived in all of its inputs, and not earlier. In order to achieve this requirement, the circuit must be ordered based on distance from the primary [20].

Levelization

The distance of a cell from the primary inputs is defined as its *level* and it can be obtained as,

$$Level(Cell_i) = max_j(Level(Cell_j) + 1) \text{ and } Cell_j \rightarrow Cell_i$$

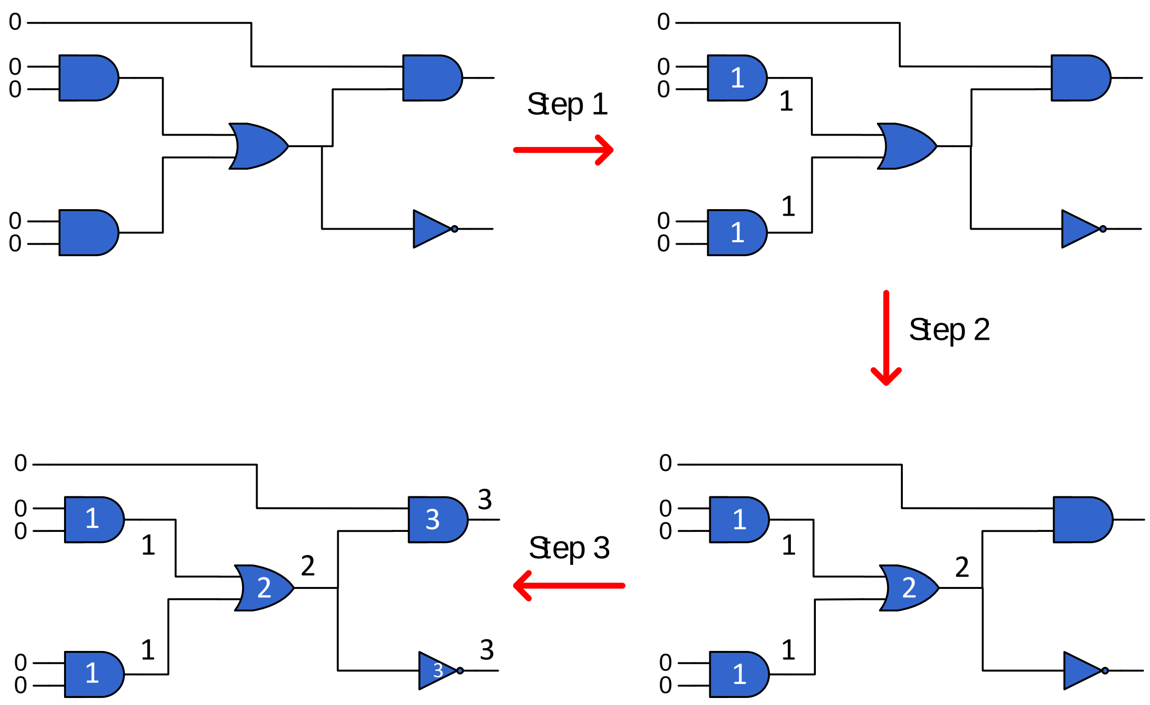


Figure 32 - Levelization example.

Starting from the primary inputs (see Figure 32), a breadth first traversal (see Appendix B) is deployed to propagate the levels through the circuit. So cells that are connected directly to the primary inputs should be level 1, and so on. The levelization algorithm exhibits a problem when sequential elements (flip flops or latches) are found in our design, since these elements introduce loops in the cell connectivity graph. In order to resolve this issue the breadth first algorithm does not propagate the level through a register pin of a sequential element. If loops are present in the design (see Figure 33), the levelization algorithm will fail to proceed the design.

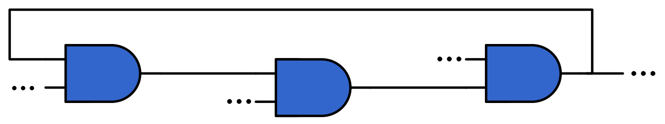


Figure 33 - An unlevelized circuit.

After the levelization has taken place, a histogram is created, which indicates which cells are in which level.

Static timing analysis algorithm

The pseudocode of the STA algorithm is presented,

```
for every net
{
    calculate elmore delay
    calculate beta factors
}

for every PI
{
    propagate (slew, arrival time) at level 1 cells
}

for every level i
{
    for all the cells at level i
    {
        calculate (slew, arrival time) at the output of the cell
        propagate (slew, arrival time) at driving cells
    }
}
```

Algorithm 2 - STA.

The calculation of the slew and arrival time at the output (merge point) of a cell, and the propagation through the interconnect, is done as specified in the previous sections. The STA algorithm only propagates the late slews and arrival times, since the resizing algorithm needs only the worst paths of the circuit.

3.5.7 Critical path extraction

After the propagation of slew and arrival time, every node on the circuit has the worst possible arrival time from all the paths that converge to that node. Tracing back from every node will produce the worst possible sub-path for each node.

The resizing algorithm uses a set of terminal nodes, from which will select the node with the worst arrival time. Every node has two arrival times, one indicating the fall state and one indicating the rise state. So in order to find the worst arrival time from all the nodes the algorithm uses,

$$max_{at} = max(max_i(at_{fall}^i), max_i(at_{rise}^i)) \text{ where } i \epsilon \; terminal \; nodes$$

At this point starting from the node with the worst arrival time, in addition with the state of that arrival time, the algorithm extracts the path in backward traversal, until it reaches a primary input.

Terminal nodes

The set of terminal nodes, is initialized as the leaves of the graph, which was produced from the node representation of the circuit. That leaves can be categorized as the primary outputs, the register pins, and every pin of a cell that is not connected to any net. The initialization of the set can be configured by the user.

The set of terminal nodes, is updated by the resizing algorithm, so the whole circuit can be examined. Nodes will be removed from the set, while new nodes will be inserted during the execution of the resizing algorithm.

3.5.8 Optimizations

Exploitation of the benefits that current multi-core architecture have to offer is of high importance for compute intensive problems. The resizing algorithm can be compute intensive for large designs, because the timing information needs to be reevaluated after a change occurs in the design. For this reason a series of algorithmic optimizations, which target multi-core architectures or reduce the problem space are deployed in order to reduce the required execution time.

Multi-threading optimization

The presented pseudocode for STA has two embarrassingly parallel regions, that do not share any data, and there is not any need for a locking scheme to be deployed.

The first region corresponds to the elmore delay and beta factor calculation, because every net is completely independent from each other. The computation that needs to be performed in this region, requires path tracing to be deployed in the interconnect graph (one post order graph traversal, two depth first graph traversals), and on top of that, in case of resistor loops the maximum spanning tree algorithm needs to be deployed to alleviate the problem. In very large designs with millions of nets, which can be very complex, an optimization at this region might have a big impact on the execution time.

The second region corresponds the calculation and propagation of the slew and arrival time, for all the cells that are in the same level, because all the cells at that level do not share any common points in the graph of the circuit. This region of STA is the main hotspot of the resizing algorithm, since the timings of the design need to be revaluated over and over again. In very large designs, the STA algorithm will have to be performed thousands of times, which means that an optimization in this region will have a huge impact on the execution time.

The pseudocode of the STA algorithm, including parallelization, is presented,

```
partition the (nets) into chunks
assign the chunks to the threads
thread j:
{
    for every net in the chunk
    {
        calculate elmore delay
        calculate beta factors
    }
}

for every PI
{
    propagate (slew, arrival time) at level 1 cells
}

for every level i
{
    partition the (cells at level i) into chunks
    assign the chunks to the threads
    thread j:
    {
        for all the cells in the chunk
        {
            calculate (slew, arrival time) at the output of the cell
            propagate (slew, arrival time) at driving cells
        }
    }
}
```

Algorithm 3 - Parallel STA.

Incremental optimization

STA needs to be performed every time a set of changes has occurred in the design, which is the resizing of some cells within a path. The initial STA algorithm propagates the timing information in the design, starting from its primary inputs. The same approach can be used throughout the execution of the resizing algorithm. Although this is a valid way of performing the STA, it does not take into account that some portion of the design will stay unchanged, in terms of timing information, even if a change has taken place. A better approach would be to first find points that will be affected by the change, and then deploy the STA from those points.

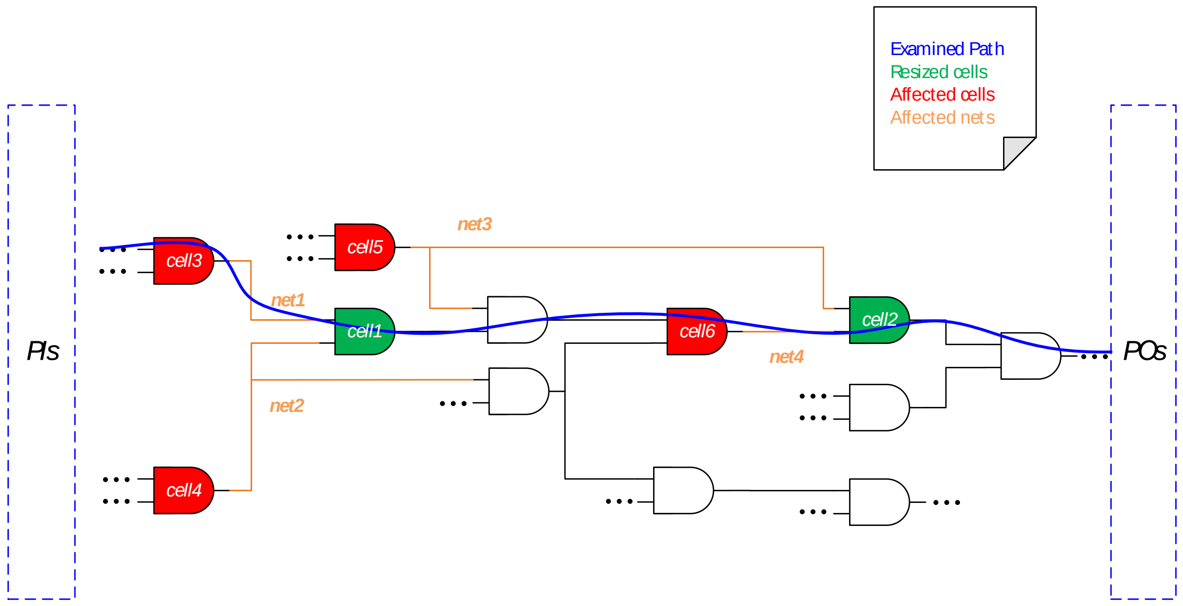


Figure 34 - Affected cells and net after resizing.

Consider the case that two cells have been resized in the examined path (see Figure 34), cells cell1 and cell2. The only elements, cells or interconnect, that are directly affected by the resized cells, are cell3, cell4, net1, net2 for the cell cell1 and cell5, cell6, net3, net4 for the cell cell2.

For all the affected nets, the STA algorithm needs to recalculate the elmore delay and the beta factors, since the input pins capacitance has changed for their driving cells.

In the case of cells, every affected cell will be tagged for examination, but that is not enough for the STA algorithm. The minimum affected cell level needs to be found, so that the STA can initialize the propagation from this level. During the stage of the STA, a cell at a given level will be examined only if it was tagged earlier during the sta. Each examined cell will tag for examination its driving cells. In this way the slews and arrival times will be propagated through the circuit, reaching the primary outputs.

The pseudocode of the incremental STA, including parallelization, is presented,

```
for every resized cell
{
    find the affected nets
    find the affected cells
}

partition the (affected nets) into chunks
assign the chunks to the threads
thread j:
{
    for every net in the chunk
    {
        calculate elmore delay
        calculate beta factors
    }
}

for every affected cell
{
    tag the cell for examination
}

find the min level from the affected cells

if min level is 0
{
    propagate (slew, arrival time) at level 1 cells
    set min level to 1
}

starting from min level for every level i
{
    partition the (cells at level i) into chunks
    assign the chunks to the threads
    thread j:
    {
        for all the cells in the chunk
        {
            if the cell is tagged
            {
                calculate (slew, arrival time) at the output of the cell
                propagate (slew, arrival time) at driving cells
                tag the driving cells for examination
            }
        }
    }
}
```

Algorithm 4 - Minimum affected cell level optimization.

Fanout cone and dominant edges

The fanout cone is defined as a set of cells that were tagged by the incremental sta. Each of the tagged cells may or may not have its values for slew or arrival time changed. This counter intuitive problem, that is a cell will not be updated even if its predecessor in the fanout cone was updated, arises from edges that originate outside the fanout cone and connect to a cell inside the cone.

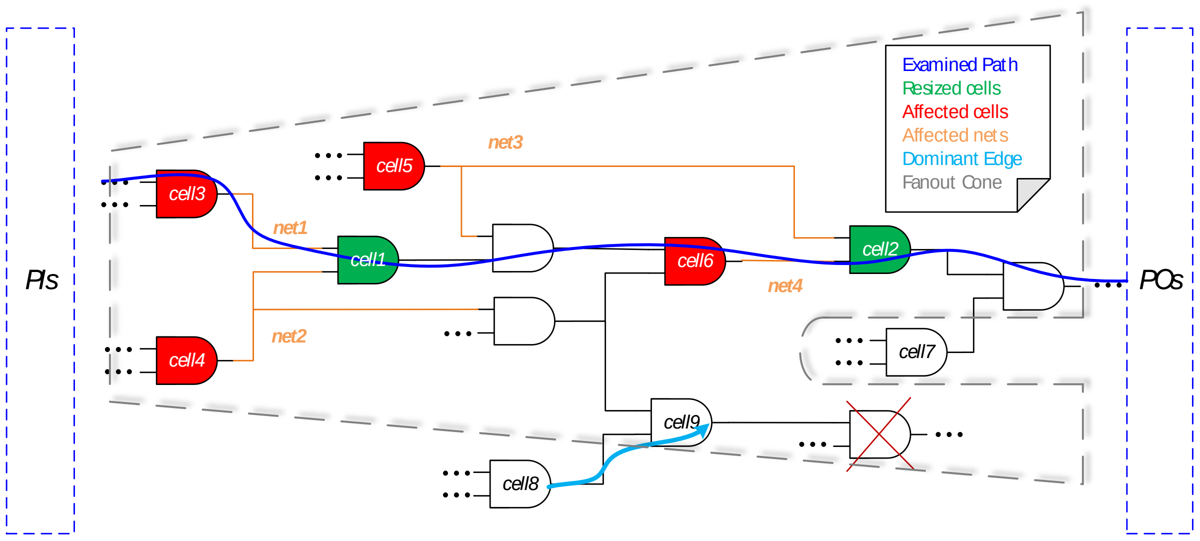


Figure 35 - Fanout cone.

For instance the cells, cell7 and cell8, originate outside the cone and they are connected to cells inside the cone (see Figure 35). Let’s assume that the slew and arrival time, originating from cell8, that were propagated in an earlier STA execution, dominates the slew and arrival time at the output of cell9. This means that there is no need to further examine the fanout cone of cell9, since all the slews and arrival times of that cone are already calculated from an earlier STA step. In this way, all the driving cells of cell9 will not be tagged by the STA, which leads into further reduction in the execution time.

The pseudocode of the incremental STA with dominant edges, including parallelization, is presented,

```
for every resized cell
{
    find the affected nets
    find the affected cells
}

partition the (affected nets) into chunks
assign the chunks to the threads
thread j:
{
    for every net in the chunk
    {
        calculate elmore delay
        calculate beta factors
    }
}

for every affected cell
{
    tag the cell for examination
}

find the min level from the affected cells

if min level is 0
{
    propagate (slew, arrival time) at level 1 cells
    set min level to 1
}

starting from min level for every level i
{
    partition the (cells at level i) into chunks
    assign the chunks to the threads
    thread j:
    {
        for all the cells in the chunk
        {
            if the cell is tagged
            {
                calculate (slew, arrival time) at the output of the cell
                if (slew, arrival time) differs from the old values
                {
                    propagate (slew, arrival time) at driving cells
                    tag the driving cells for examination
                }
            }
        }
    }
}
```

Algorithm 5 - Dominant edges optimization.

Maximum propagation level

The increment STA produces the same results, in terms of slews and arrival times at each node, in comparison to the non-incremental version of the algorithm, with the least possible execution time. Every value will be propagated through the circuit all the way until they reach the primary outputs. A further optimization step can be applied, when the points in the circuit, in which the arrival times will be requested, are known beforehand. Knowing this information, the STA algorithm can terminate the propagation at the furthest point.

The resizing algorithm requires the critical path, within a set of points in the circuit, which is the set of terminal nodes. The terminal nodes can be part of a cell, or primary outputs, which means that they can have a corresponding level assigned to them.

If the terminal node is a primary output, its level is derived as,

$$level = 1 \text{ if } PI \rightarrow \mathrm{PO}$$
$$level = Level(Cell_j) + 1 \text{ if } Cell_j \rightarrow \mathrm{PO}$$

If the terminal node is a register pin of a sequential cell, its level is derived as,

$$level = 1 \text{ if } PI \rightarrow Sequential\ Cell$$
$$level = Level(Cell_j) + 1 \text{ if } Cell_j \rightarrow \text{Sequential Cell}$$

If the terminal node is an output pin of a cell, its level is derived as,

$$level = Level(Cell) + 1$$

At this point the maximum level from all the terminal nodes must be found, so that the STA can be performed up until that level (see Figure 36).

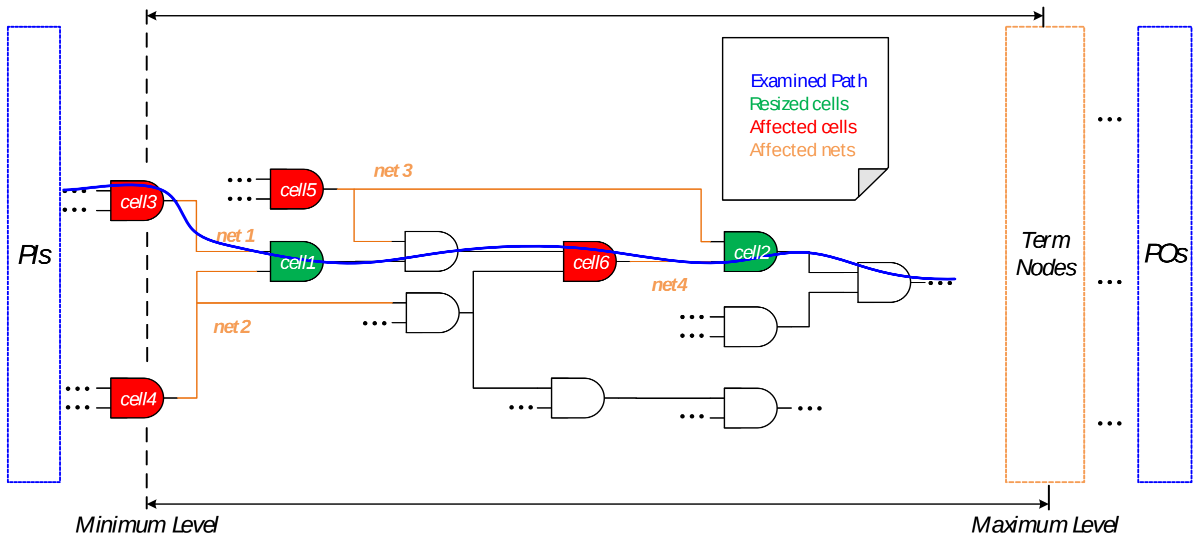


Figure 36 - Min and Max levels.

The pseudocode of the incremental STA with dominant edges, until the maximum level, including parallelization, is presented,

```
for every resized cell
{
    find the affected nets
    find the affected cells
}


partition the (affected nets) into chunks
assign the chunks to the threads
thread j:
{
    for every net in the chunk
    {
        calculate elmore delay
        calculate beta factors
    }
}

for every affected cell
{
    tag the cell for examination
}

find the min level from the affected cells

if min level is 0
```

```
{
    propagate (slew, arrival time) at level 1 cells
    set min level to 1
}

find the max level from the terminal nodes

starting from min level for every level i until the max level
{
    partition the (cells at level i) into chunks
    assign the chunks to the threads
    thread j:
    {
        for all the cells in the chunk
        {
            if the cell is tagged
            {
                calculate (slew, arrival time) at the output of the cell
                if (slew, arrival time) differs from the old values
                {
                    propagate (slew, arrival time) at driving cells
                    tag the driving cells for examination
                }
            }
        }
    }
}
```

Algorithm 6 - Maximum terminal node level optimization.

3.6 Resizing Algorithm

3.6.1 Introduction

The core of the resizing engine is the ULE method, which uses an iterative approach, in order to converge into the optimal cell input capacitances for a given path, and therefore the optimal cell sizes. The method, as mentioned before, takes into account branches and wire load (Resistances/Capacitances), along with the slope at every pin in the path (Rise/Fall). The path is evaluated, using a backward traversal, for a number of iterations until the values of all the input capacitances have not changed, in comparison to a defined error threshold, from the values in the previous iteration. At the initialization step, instead of assigning arbitrary capacitance values to the cells in the examined path, the original capacitance values of the current cells are used.

A continuous sizing algorithm will then use the new capacitance values to generate the lib file data for the new version of cells, which meet those input capacitance restrictions. An alternate approach was used, since the modeling algorithm that would produce the new cells was not in its final stage in order to be included into the resizing tool (was under development by another group of researchers). For that reason, the input capacitance values must be approximated, by the corresponding type cell in the lib file, which has the closest capacitance value to the optimal value. In order to achieve better results with this approach, the lib file, which contains the discrete versions of the cells (x1, x2, x4…), must be changed in order to include intermediate cells sizes (x0.5, x1, x1.5…). Creating a lib file with a small discretization step will lead to better approximations, which could be done by the modeling algorithm.

The final step is to substitute every cell in the path, excluding the first cell, according to ULE input capacitance values in the case of continuous cells resizing, or the approximation in the case of discrete cell resizing. All the cells in the path, including the first cells in the off-path branches, excluding the off-path branch cells of the first cell (since the first cell cannot be changed there is no need to mark its off-path branch cells), will be marked as examined (see Figure 37). The marked cells will not be changed again throughout the execution of the tool. Additionally two lists will be created, containing the cell instances that were changed, and the nets that were affected by the changes, which will be processed by the STA algorithm in order to calculate the new arrival times.

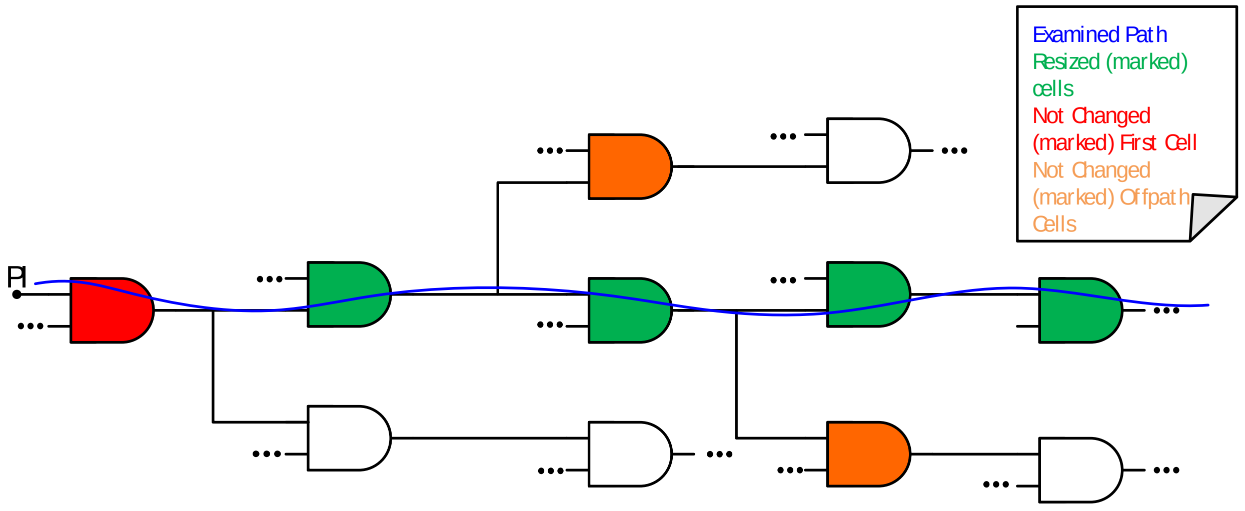

Examined Path
Resized (marked) cells
Not Changed (marked) First Cell
Not Changed (marked) Offpath Cells
PI

Figure 37 - An instance of a resized path.

The unified logical effort method optimizes a path in terms of performance (delay), so in order to achieve a power-performance optimization, an optional criterion was implemented, in the resizing algorithm, which will not allow the upsizing of the cells. The ULE method will converge to input capacitance values, which are greater than the original capacitance values for a given cell. Although the method would lead to an optimal performance path, the upsizing of a cell would increase the overall power consumption. If a case of upsizing arises, no cell will be changed in the examined path, and the first cell that created the upsizing exception will be marked as examined (see Figure 38).

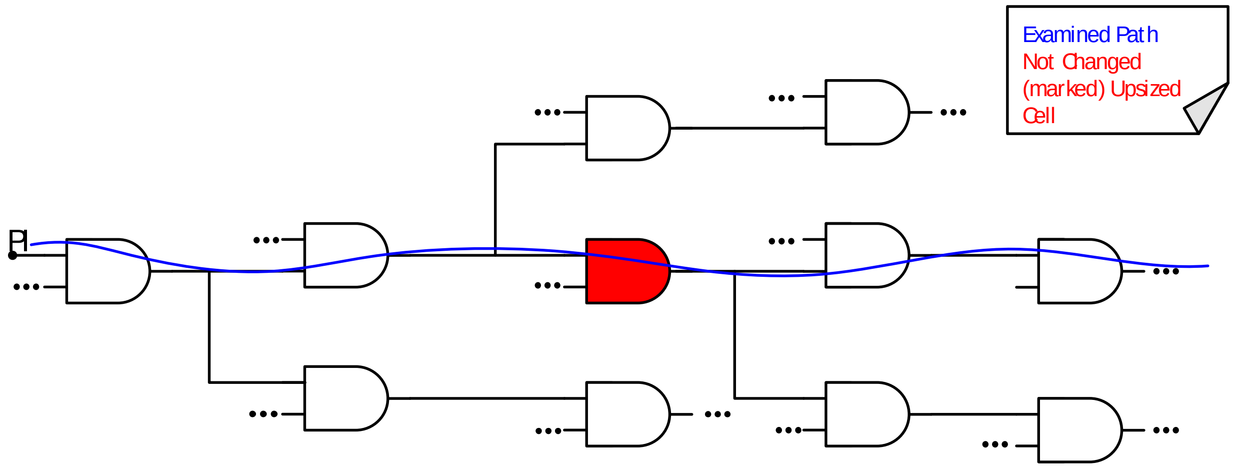


Figure 38 - An instance of up-sizing.

3.6.2 Path Preprocessing

The ULE method, evaluates a given path, which was not re-examined before by the resizing algorithm. In order to filter the paths into examined (Partially) or not, a sub-path extraction algorithm was implemented. This algorithm takes as input a path from the STA engine, and specifically from the critical path extraction algorithm. The path is then tokenized into smaller, not examined portions of the path, if any, and each sub path is then processed by the ULE method (see Figure 39).

In the case of disallowing upsizing, the ULE method fails to process the given path, which will lead to even further preprocessing of the path.

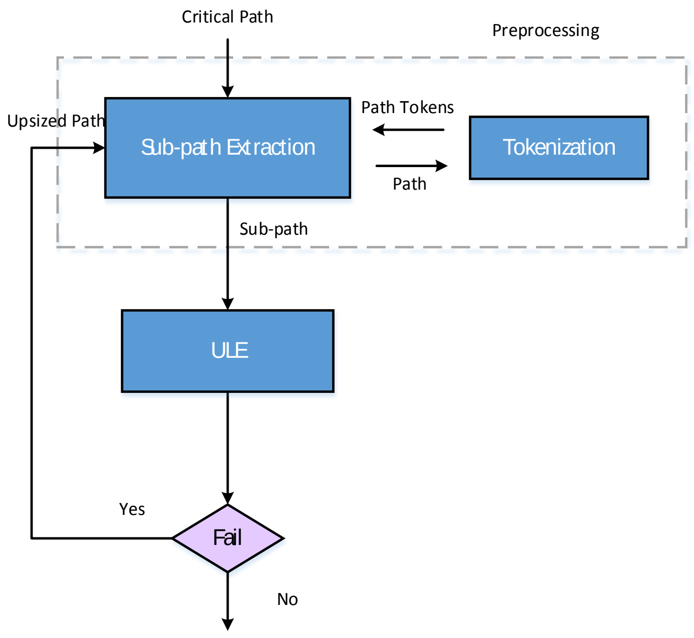


Figure 39 - Path preprocessing.

Path Tokenization

The path tokenization algorithm uses the information, in each cell instance, which define if a cell was examined before throughout the execution of the resizing algorithm, in order to create smaller tokens of the path, given as input. In that case a simple string tokenization approach is used, where the delimiter is the examined information. A simple modification can be used in the tokenization algorithm, which takes into account that the ULE method will never change the first cell in a path. In that case a token is defined as a set of not examined consecutive cells, which may or may not have the first cell in the sequence re-examined before (see Figure 40). Including this extra cell in the path token will yield better results in the ULE method.

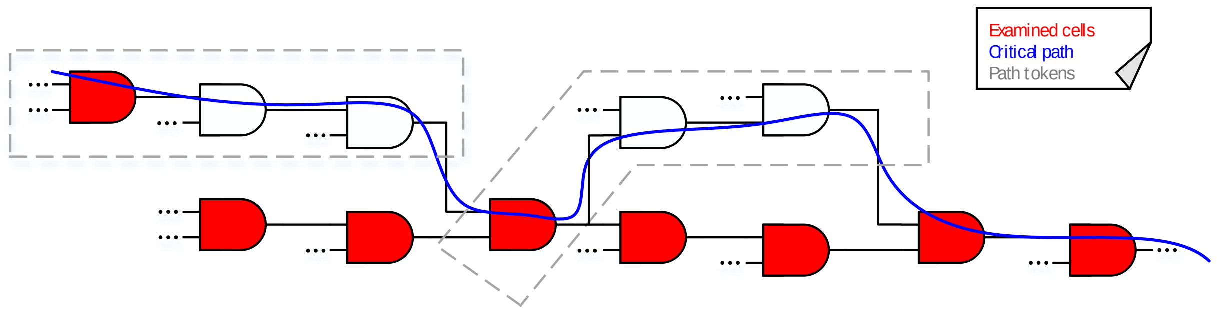


Figure 40 - Path tokenization.

3.6.3 Terminal Nodes Re-evaluation

The main purpose of keeping a set of the terminal nodes, ensures that the whole circuit will be examined by the resizing algorithm, which means that the set must be updated throughout the execution of the algorithm. The ULE method populates a list of changed cells, along with the affected nets. The cells are defined as changed, only if they were replaced by a downsized version. The ULE method, often cannot replace any cell in the examined path, which means that the resizing algorithm reached a dead-end. This issue arises when all the cells in the current path were already marked as examined by the algorithm. In that case the algorithm must continue the examination of the rest of the circuit, by finding the next in line critical path. Since no cell was changed during the last path examination, there is no need to execute the STA algorithm. The problem that arises, is the critical path extraction algorithm, which is searching for a path that ends in the terminal nodes set, will simply produce the same path as before. In order to alleviate that problem, the terminal node that produced the dead-end must be removed from the set. The total examination is ensured by adding, as terminal nodes, all the cells that feed into the problematic path, and specifically the outputs of those cells will be added into the terminal nodes set, if they were encountered for the first time (see Figure 41).

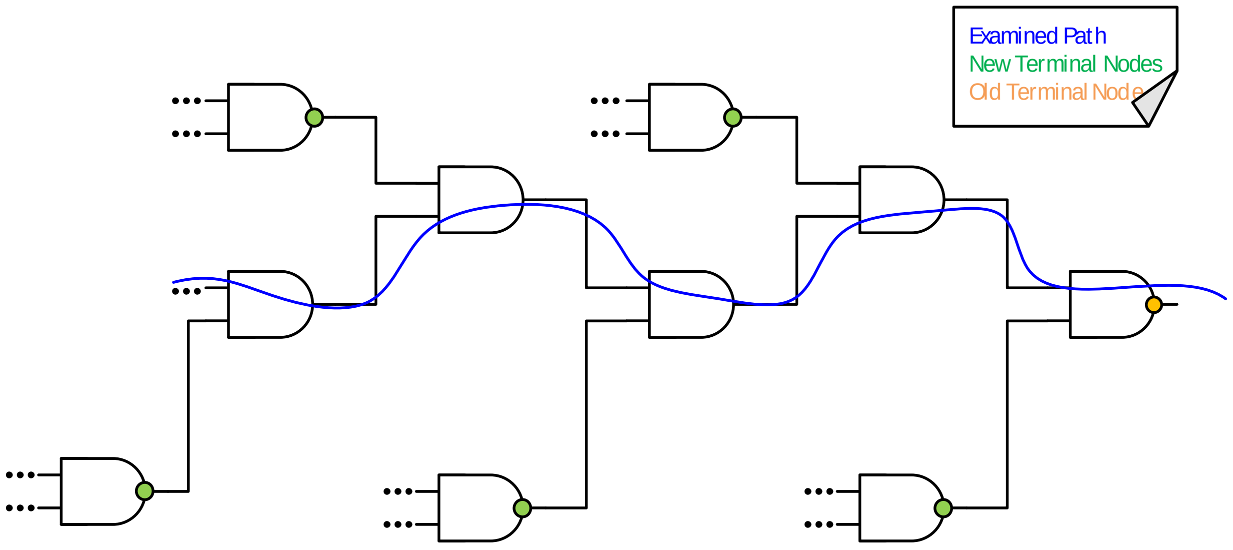


Figure 41 - Terminal nodes re-examination.

The new terminal nodes, added in the set, can only be outputs of cells, which means that the level of that terminal node is defined as,

$$level = Level(Cell) + 1$$

The levels of the new terminal nodes, added to the set, are always lower than the old terminal node, which leads to the effectiveness of the maximum propagation level STA optimization, as described earlier in the STA chapter. This property also ensures that the arrival times and slews from earlier executions will be valid for a new execution of the algorithm, since an earlier execution would have propagated the values in a level greater or equal to the new maximum terminal node level.

3.6.4 Sequential cells handling

The resizing tool uses a different methodology for sequential cells, if any, in the design. The two available options are: No change, which means that all the sequential cells will stay the same throughout the execution of the algorithm. Minimum size, which means that every sequential cell will be replaced by the lowest driving strength cell candidate. In both cases, the cells will be marked as examined. The tokenizing algorithm, as described before, will produce sub paths, if the examined path contains a sequential cell (see Figure 42).

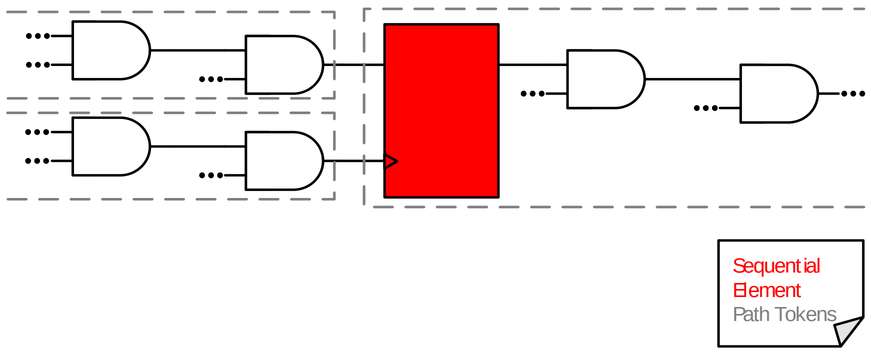


Figure 42 - Sequential cell handling.

3.6.5 Algorithm Overview

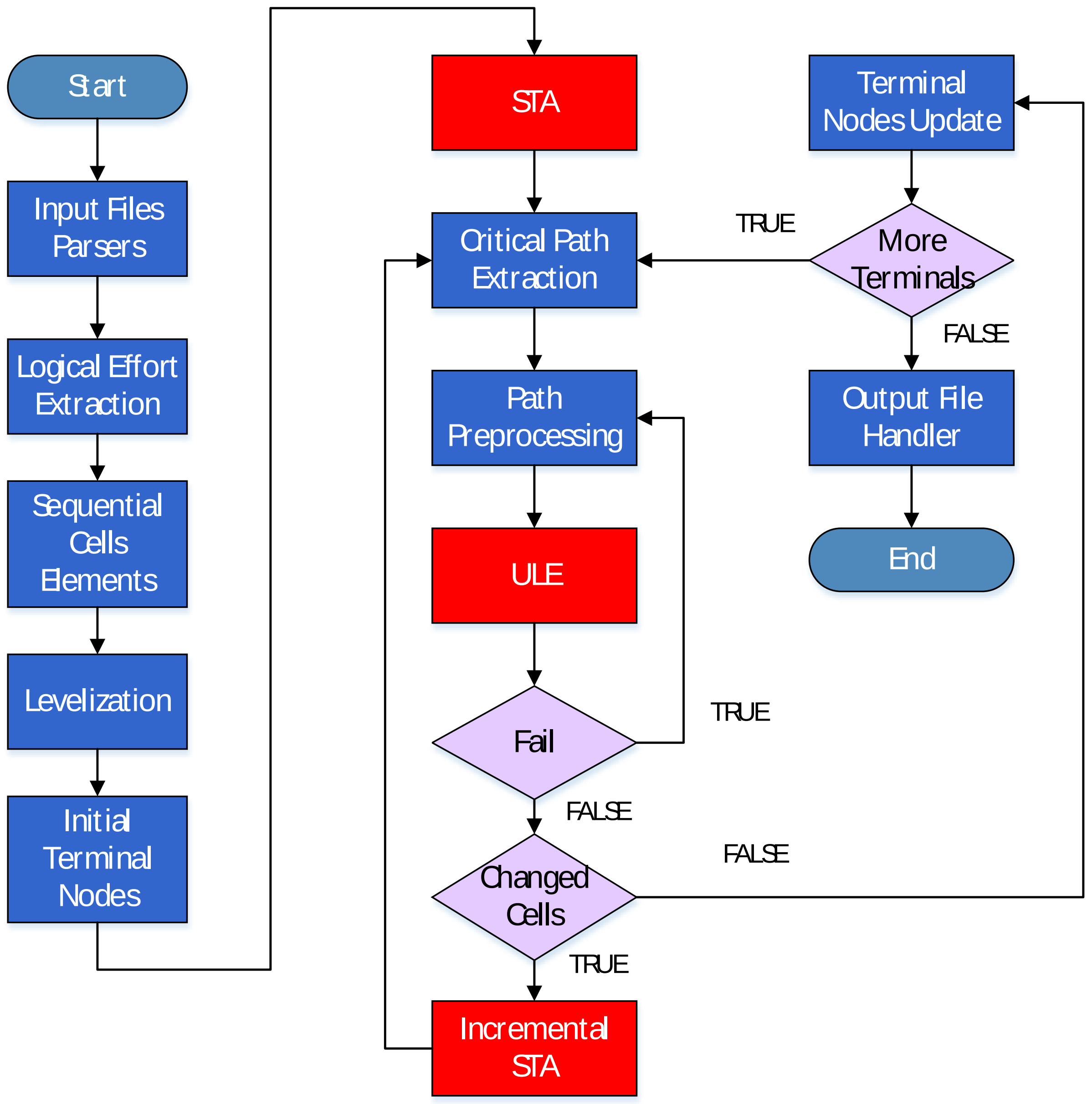


Figure 43 - CCSopt algorithm overview.

ULE

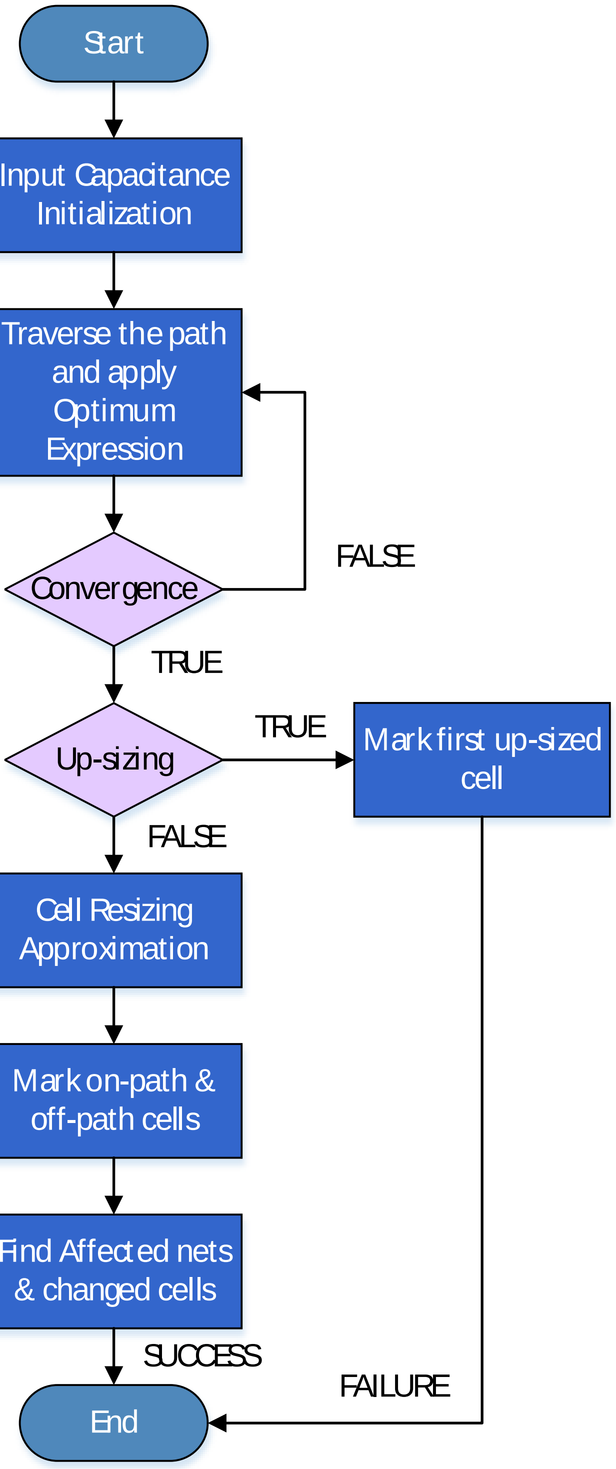

Start
Input Capacitance Initialization
Traverse the path and apply Optimum Expression
Convergence
FALSE
TRUE
Up-sizing
TRUE
Mark first up-sized cell
FALSE
Cell Resizing Approximation
Mark on-path & off-path cells
Find Affected nets & changed cells
SUCCESS
FAILURE
End

Figure 44 - ULE method overview.

STA

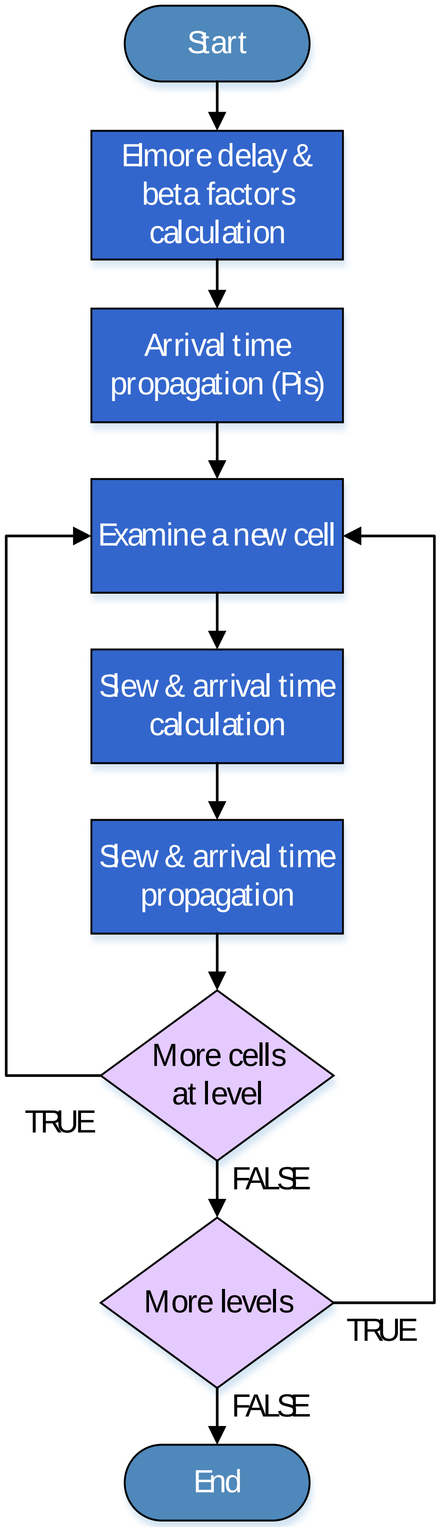

Start
Elmore delay & beta factors calculation
Arrival time propagation (Pis)
Examine a new cell
Slew & arrival time calculation
Slew & arrival time propagation
More cells at level
TRUE
FALSE
More levels
TRUE
FALSE
End

Figure 45 - STA algorithm overview.

STA Incremental

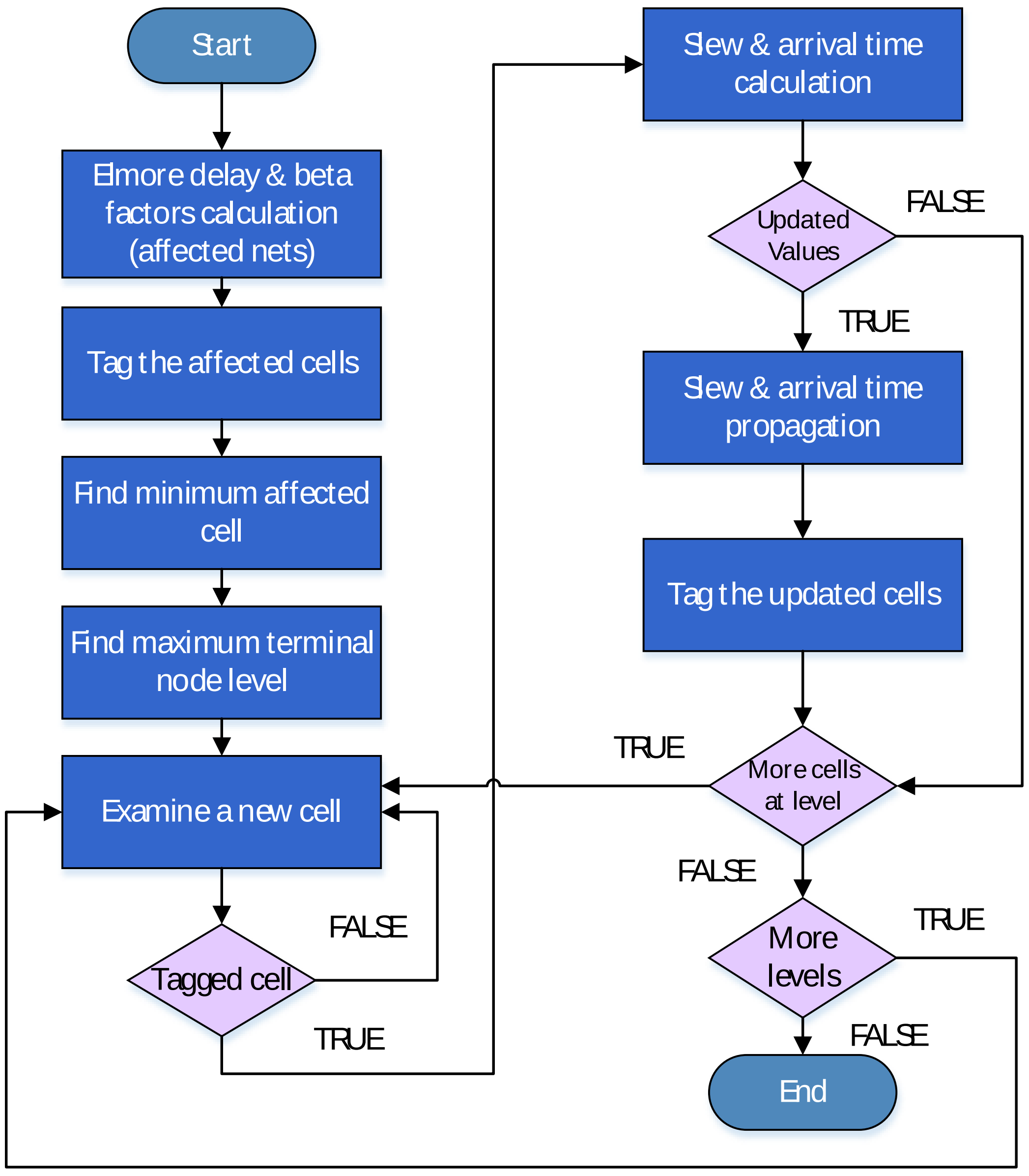

Start
Elmore delay & beta factors calculation (affected nets)
Tag the affected cells
Find minimum affected cell
Find maximum terminal node level
Examine a new cell
Tagged cell
FALSE
TRUE
Slew & arrival time calculation
Updated Values
FALSE
TRUE
Slew & arrival time propagation
Tag the updated cells
More cells at level
TRUE
FALSE
More levels
TRUE
FALSE
End

Figure 45 - Incremental STA algorithm overview.

Chapter 4

Results

In this section, a number of benchmarks is presented, indicating the number of the resized cells, using the discrete resizing algorithm, along with the potential number of resized cells when using the continuous algorithm. The suite of larger benchmarks where found in TAU 2015 timing contest.

Iscas Benchmarks

Benchmark	**# Cells**	**# Primary Inputs**	**# Primary Outputs**	**# Resized (Discrete)**	**# Resized (Continuous)**	**Execution Time (s)**
c432	168	36	7	9	24	0.491
c499	210	41	32	6	17	0.547
c880	383	60	26	16	39	0.743
c1355	554	41	32	19	37	0.913
c1908	932	33	25	30	59	1.178
c2670	1278	233	140	41	63	1.649
c3540	1719	50	22	18	106	2.034
c5315	2332	178	123	62	174	2.773
c6288	2416	32	32	22	171	2.836
c7552	3573	207	108	101	156	3.811
s298	136	4	6	10	25	0.449
s344	175	10	11	8	17	0.462
s382	183	4	6	16	25	0.493
s400	188	5	6	9	23	0.498
s420	237	19	1	15	50	0.523
s641	398	36	24	21	72	0.631
s5378	2970	36	49	78	335	2.984
s38417	23835	29	106	346	2574	19.361

Table 3 - Iscas Benchmarks

Larger Benchmarks

Benchmark	**# Cells**	**# Primary Inputs**	**# Primary Outputs**	**# Resized (Discrete)**	**# Resized (Continuous)**	**Execution Time (s)**
ac97_ctrl	14341	84	48	1350	3603	15.545
aes_core	22938	260	129	1728	2965	29.054
des_perf	105371	235	64	7695	16665	161.352
mem_ctrl	10531	115	152	1027	2186	13.710
pci_bridge32	19057	162	207	1170	4234	25.073
systemcaes	6484	260	129	573	1236	9.055
systemcdes	3441	132	65	219	469	3.776
tv80	5285	14	32	307	773	6.492
usb_funct	15743	128	121	1576	3593	18.424
wb_dma	4195	217	215	326	891	5.052
vga_lcd	139529	80	109	17073	34146	290.549
cordic_ispd	45359	34	64	11992	20118	52.970
des_perf_ispd	138878	234	140	52814	83548	198.536
edit_dist_ispd	147650	2562	12	73386	88152	251.985
fft_ispd	38158	1026	1984	6833	13824	45.411
matrix_mult_ispd	164040	3202	1600	41485	68545	261.750
pci_bridge32_ispd	40790	160	201	18366	21701	44.429
usb_phy_ispd	923	15	19	356	551	1.044
netcard_iccad	1496719	1836	10	287608	369598	17942.690

Table 4 - Larger Benchmarks

Chapter 5

Conclusion & Future Work

A lot of effort has been put, over the years, in arriving at the optimum transistor sizes both in respect of delay and power. The proposed method builds on the learnings from previous attempts at continuous transistor sizing by resolving all the issues that have prevented such attempts from arriving to a viable solution. Our hybrid heuristic approach takes into account interconnect capacitance and resistance. It also considers reconvergent fanouts and arrives at a stable solution in all cases without the possibility of divergence, which has plagued even commercial tools. The CCSopt tool can be easily incorporated into a standard cell based digital IC design flow.

A number of possible extensions and changes should be revisited, that will allow CCSopt to have better quality and performance. Those extensions can be summarized as:

- Implementing the cell characterization models that will allow the tool to behave as a continuous sizing tool.
- Reducing the number of cells that do not get resized, such as the first cells of the off-path branches.
- Using a better criterion, in addition to only allowing down-sizing, to further reduce the power consumption.
- Changing the delay calculation method, to use the CCS model as described in the lib file, to better approximate the delays of a cell, since the NLDM is not so accurate in the sub-nanometer regime.
- Altering the incremental STA, in order to only propagate the slews and arrival times through cells that lead to active terminal nodes, which will improve the overall performance.
- Changing the busses handling, in order to provide a better functionality.
- Introducing a multi-threaded approach for parsing the .spef file, since the parsing overhead for large designs is significant.

Bibliography

[1] J. Fishburn and A. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1985.

[2] A. E. Dunlop, "Transistor Sizing for integrates Circuit". U.S. Patent No.4827428.

[3] S. S. Sapatnekar, B. V. Rao, P. M. Vaidya and S. M. Kang, "An exact Solution to the Transistor Sizing Problem for CMOS Circuits using Convex Optimization," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1993.

[4] I. Sutherland, D. Harris and B. Sproull, Logical Effort - Designing Fast CMOS Circuits, Morgan Kaufmann Publishers, 1999.

[5] L. Sutherland, "Determining transistor widths using the theory of logical effort". U.S. Patent No.6629301.

[6] H. H. F. Jyu, "Minimization of circuit delay and power through transistor sizing". U.S. Patent No.6209122.

[7] L. G. Jones, "Method and apparatus for designing an integrated circuit". U.S. Patent No.5666288.

[8] R. F. Leimbach, "Method of optimizing signal timing delays and power consumption in LSI circuits". U.S. Patent No.4698760.

[9] A. Morgenshtein, "Logic circuit delay optimization". U.S. Patent No.12292931.

[10] F. R. Sproull and E. S. Ivan, "Logical Effort:designing for speed on the back of an envelope," in *IEEE Advanced Research in VLSI*, 1991.

[11] A. Morgenshtein, E. Friedman, R. Ginosar and A. Kolodny, "Unified logical effort - a method for delay evaluation and minimization in logic paths with RC interconnect.," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

[12] C. W. Elmore, The trasient response of damped linear networks with particular regard to wide band amplifiers, vol. 19, J. Appl. Phys., 1948, pp. 55-63.

[13] "Closed form solution to silmutaneous buffer insertion/sizing and wire sizing," in *ACM Trans. Design Autom. Electron. Syst.*, 2001.

[14] K. Venkat, "Generalized delay optimization of resistive interconnections through an extension of logicla effort," in *Proc. IEEE Int. Symp Circuits Syst.*, May 1993.

[15] M. Moreinis, A. Morgenshtein, I. Wagner and A. Kolodny, "Logic Gates as repeaters (LRG) for area-efficient timing optimization," in *IEEE Trans Very Large Scale Integr. (VLSI) Syst.*, Nov. 2006.

[16] A. Cao, R. Lu and C. K. Koh, "Post-Layout logic duplication for synthesis of domino circuits with complex gates," in *Proc. ASP-DAC*, Jan. 2005.

[17] TAU Workshop, "tauworkshop," 9 2 2015. [Online]. Available: https://sites.google.com/site/taucontest2015/resources/documents/contest_file_formats.pdf?attredirects=0.

[18] H. E. N. Weste and M. D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 4th ed., Addison-Wesley Publishing Company, 2010.

[19] J. Basker and C. Rakesh, Static Timing Analysis for Nanometer Design: A Practical Approach, New York: Springer, 2009.

[20] TAU Workshop, "tauworkshop," 19 1 2015. [Online]. Available: https://sites.google.com/site/taucontest2015/resources/documents/contest_education.pdf?attredirects=0.

[21] A. Mittas, "Timing Analysis of Integrated Circuits".

Appendix A

Config File

The resizer uses a simple interface in order to initialize the performed operations. The list of the available commands

output_capacitance <state> <output_node_name> <value>

<state>: [**-fall** | **-rise**]

<output_node_name>: [**-all** | node_name]

This command sets the output capacitance, either for rise or fall state, on the specified output. If the -all specifier is set, then the given capacitance value is set on every output node of the circuit. The capacitance value must be given in the liberty file capacitance unit.

input_slew <state> <mode> <input_node_name> <value>

<state>: [**-fall** | **-rise**]

<mode>: [**-late**]

<input_node_name> : [**-all** | node_name]

This command sets the input slew, either for rise or fall state, in late mode, on the specified input. If the -all specifier is set, then the given slew value is set on every input node of the circuit. The slew value must be given in the liberty file time unit.

input_at <state> <mode> <input_node_name> <value>

<state>: [**-fall** | **-rise**]

<mode>: [**-late**]

<input_node_name>: [**-all** | node_name]

This command sets the input arrival time, either for rise or fall state, in late mode, on the specified input. If the -all specifier is set, then the given arrival time value is set on every input node of the circuit. The arrival time value must be given in the liberty file time unit.

resize_sequential <mode>

<mode>: [**-no** | **-min**]

This command sets the resizing mode for the sequential cells. If -no is specified, the resizer will not resize any sequential cells. If -min is specified, the resizer will resize the sequential cells, with the corresponding minimum drive strength sequential cells.

path_type <type>

<type>: [**-all** | **-out** | **-reg**]

This command sets the path type examined by the resizer. The paths are categorized into: Paths starting from primary inputs and end at every node in the circuit without any fanout, if -all is specified. Paths starting from primary inputs and end on primary outputs, if -out is specified. Paths starting from primary inputs and end on register pins, if -reg is specified.

preset_clear_arcs <enable>

<enable>: [**-on** | **-off**]

This command enables the asynchronous timing arcs on sequential elements.

upsizing <enable>

<enable>: [**-on** | **-off**]

This command enables upsizing on cells by the resizer. If -off is specified the resizer will not allow upsizing on cells.

unit_inverter <cell_name>

This command sets the unit inverter. The unit inverter specified must be included in the lib file.

c_unit_lib <value> <unit>

<value>: [**1** | **10** | **100**]

<unit>: [**PF** | **FF**]

This command specifies the capacitance unit that will be used by the resizer. The unit must match the unit defined in the lib file. For instance if the value is 10 and the unit is FF, then every capacitance value is scaled as 10 femtofarad.

t_unit_lib <value> <unit>

<value>: [**1** | **10** | **100**]

<unit>: [**NS**| **PS**]

This command specifies the time unit that will be used by the resizer. The unit must match the unit defined in the lib file. For instance if the value is 10 and the unit is PS, then every delay value is scaled as 10 picoseconds.

max_iterations <value>

This command sets the maximum iterations that will be used by the unified logical effort algorithm.

tolerance <value>

This command sets the tolerance that will be used by the unified logical effort algorithm.

cell_delimiters <characters>

This command sets the delimiter characters in the name of a cell. For instance if a cell is named *AND2_X1*, then the delimiter should be the character X. If the lib file contains cell names with different delimiters, every delimiter must be included. For instance if the lib file contains the cells *AND2_X1* and *NOR2_Y1,* both characters X and Y must be included as delimiters.

report_timing <value>

<value>: [**-all** | **numerical_value**]

This command reports the worst paths, sorted by their arrival time. The reported paths are filtered by path_type. Warning this command requires the examination of every path in the design, which leads to exponential complexity. It is only designed for debugging purposes.

Extra Supported Verilog Files

Explicit version

```
module adder_explicit (
          result      , // Output of the adder
          carry       , // Carry output of adder
          r1          , // first input
          r2          , // second input
          ci           // carry input

          );

          // Input Port Declarations
          input   [3:0]   r1      ;
          input   [3:0]   r2      ;
          input           ci      ;

          // Output Port Declarations
          output  [3:0]  result    ;
          output         carry     ;

          // Port Wires
          wire    [3:0]   r1      ;
          wire    [3:0]   r2      ;
          wire            ci      ;
          wire    [3:0]   result  ;
          wire            carry   ;

          // Internal variables
          wire            c1      ;
          wire            c2      ;
          wire            c3      ;

          // Code Starts Here
          addbit u0 (
          .a        (r1[0])      ,
          .b        (r2[0])      ,
          .ci       (ci)         ,
          .sum        (result[0])   ,
          .co        (c1)
          );
```

```
            addbit u1 (
            .a          (r1[1])       ,
            .b          (r2[1])       ,
            .ci         (c1)          ,
            .sum        (result[1])   ,
            .co         (c2)
            );

            addbit u2 (
            .a          (r1[2])       ,
            .b          (r2[2])       ,
            .ci         (c2)          ,
            .sum        (result[2])   ,
            .co         (c3)
            );

            addbit u3 (
            .a          (r1[3])       ,
            .b          (r2[3])       ,
            .ci         (c3)          ,
            .sum        (result[3])   ,
            .co         (carry)
            );

endmodule // End Of Module adder
```

Implicit version

```
module adder_implicit (
            result      , // Output of the adder
            carry       , // Carry output of adder
            r1          , // first input
            r2          , // second input
            ci            // carry input
            );

            // Input Port Declarations
            input   [3:0]   r1      ;
            input   [3:0]   r2      ;
            input           ci      ;
```

```
// Output Port Declarations
output   [3:0]  result     ;
output          carry      ;

// Port Wires
wire    [3:0]    r1       ;
wire    [3:0]    r2       ;
wire             ci       ;
wire    [3:0]    result    ;
wire             carry    ;

// Internal variables
wire             c1       ;
wire             c2       ;
wire             c3       ;

// Code Starts Here
addbit u0 (
r1[0]          ,
r2[0]          ,
ci             ,
result[0]      ,
c1
);

addbit u1 (
r1[1]         ,
r2[1]         ,
c1            ,
result[1]     ,
c2
);

addbit u2 (
r1[2]         ,
r2[2]         ,
c2            ,
result[2]     ,
c3
);

addbit u3 (
r1[3]         ,
r2[3]         ,
            c3          ,
            result[3]     ,
            carry
            );

endmodule // End Of Module adder
```

Instantiated module

```
module addbit (
            a     , // first input
            b     , // Second input
            ci    , // Carry input
            sum   , // sum output
            co      // carry output
            );
            //Input declaration
            input a;
            input b;
            input ci;
            //Ouput declaration
            output sum;
            output co;
            //Port Data types
            wire  a;
            wire  b;
            wire  ci;
            wire  sum;
            wire  co;

            wire xor1_wire, and1_wire, and2_wire;
            //Code starts here

            XOR2_X1 xor1(.A (a), .B (b), .Z (xor1_wire));
            XOR2_X1 xor2(.A (xor1_wire), .B (ci), .Z (sum));
            AND2_X1 and1(.A1 (ci), .A2 (xor1_wire), .ZN (and1_wire));
            AND2_X1 and2(.A1 (a), .A2 (b), .ZN (and2_wire));
            OR2_X1 or1(.A1 (and1_wire), .A2 (and2_wire), .ZN (co));

endmodule // End of Module addbit
```

Simple spef

```
*SPEF "IEEE 1481-1998"
*DESIGN "simple"
*DATE "Tue Sep 25 11:51:50 2012"
*VENDOR "TAU 2015 Contest"
*PROGRAM "Benchmark Parasitic Generator"
*VERSION "0.0"
*DESIGN_FLOW "NETLIST_TYPE_VERILOG"
*DIVIDER /
*DELIMITER :
*BUS_DELIMITER [ ]
*T_UNIT 1 PS
*C_UNIT 1 FF
*R_UNIT 1 KOHM
*L_UNIT 1 UH

*D_NET inp1 5.4
*CONN
*P inp1 I
*I u1:a I
*CAP
1 inp1 1.2
2 inp1:1 1.3
3 inp1:2 1.4
4 u1:a 1.5
*RES
1 inp1 inp1:1 3.4
2 inp1:1 inp1:2 3.5
3 inp1:2 u1:a 3.6
*END

*D_NET inp2 2.0
*CONN
*P inp2 I
*I u1:b I
*CAP
1 inp2 0.2
```

```
2 inp2:1 0.5
3 inp2:2 0.4
4 u1:b 0.9
*RES
1 inp2 inp2:1 1.4
2 inp2:1 inp2:2 1.5
3 inp2:2 u1:b 1.6
*END

*D_NET out 0.7
*CONN
*I u3:o O
*P out O
*CAP
1 u3:o 0.2
2 out 0.5
*RES
1 u3:o out 1.4
*END

*D_NET n1 1.0
*CONN
*I u1:o O
*I u4:a I
*CAP
1 u1:o 0.2
1 n1:1 0.3
2 u4:a 0.5
*RES
1 u1:o n1:1 1.1
2 n1:1 u4:a 1.0
*END

*D_NET n2 1.2
*CONN
*I u4:o O
*I f1:d I
*CAP
1 u4:o 0.7
2 f1:d 0.5
*RES
```

```
1 u4:o f1:d 2.1
*END

*D_NET n3 23.4
*CONN
*I f1:q O
*I u2:a I
*I u4:b I
*CAP
1 n3:1 6.7
2 n3:2 7.8
3 n3:3 8.9
*RES
1 f1:q n3:3 1.2
2 n3:3 n3:1 2.3
3 n3:1 u2:a 3.4
4 n3:3 n3:2 4.5
5 n3:2 u4:b 5.6
*END
```

Appendix B

Algorithms

Depth First Traversal

```
1.  procedure DFS(G,v):
2.     let S be a stack
3.     S.push(v)
4.     while S is not empty
5.         v ← S.pop()
6.         if v is not labeled as discovered
7.             label v as discovered
8.             for every edge from v to w in G.adjacentEdges(v) do
9.                 S.push(w)
```

Breadth First Traversal

```
1.  procedure BFS(G,v):
2.     create a queue Q
3.     enqueue v onto Q
4.     while Q is not empty
5.        t ← Q.dequeue()
6.       for every edge e in G.adjacentEdges(t) do
7.         u ← G.adjacentVertex(t,e)
8.         if u is not in visited
9.             set v as visited
10.              Q.enqueue(u)
```

Post-order Traversal

```
1. procedure  postorder(node):
2.    if node == null then return
3.    for every child of the node
4.        postorder(node.child)
```

Kruskal’s Spanning Tree

```
procedure KRUSKAL(G):
A = $\emptyset$
for every v in G.V:
  MAKE-SET(v)
for every (u, v) ordered by weight(u, v), decreasing:
  if FIND-SET(u) $\neq$ FIND-SET(v):
    A = A $\cup$ {(u, v)}
    UNION(u, v)
return A
```