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

“ Βελτιστοποίηση Ταχύτητας και Κατανάλωσης Ισχύος Ψηφιακών Κυκλωμάτων ”

“ Power of Timing Optimization of Digital Circuits ”

Διπλωματική Εργασία

Αντωνιάδης Τσατσάιας Νικόλαος

Επιβλέποντες καθηγητές :

Σταμούλης Γεώργιος

Ευμορφόπουλος Νέστωρ

Βόλος, Σεπτέμβριος 2016

Διπλωματική Εργασία για την απόκτηση του 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.

Ευχαριστίες

Με την παρούσα διπλωματική εργασία θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες καθηγητές μου για τη συνεργασία και την εμπιστοσύνη που μου επέδειξαν καθώς και τους φίλους/συνεργάτες του εργαστηρίου Ε5 και ιδιαιτέρως το διδακτορικό φοιτητή Γαρυφάλλου Δημήτριο για την καθοδήγηση και τις ουσιώδεις υποδείξεις τους. Τέλος ένα μεγάλο ευχαριστώ στην οικογένεια μου για την ανεκτίμητη βοήθεια και υποστήριξη που μου παρείχαν κατά τη διάρκεια των σπουδών μου.

Contents

Chapter 1

1.1 Static Timing Analysis Method

Static timing analysis (STA) is a simulation method of computing the expected timing of a digital circuit without requiring a simulation of the full circuit.

High-performance integrated circuits have traditionally been characterized by the clock frequency at which they operate. Gauging the ability of a circuit to operate at the specified speed requires an ability to measure, during the design process, its delay at numerous steps. Moreover, delay calculation must be incorporated into the inner loop of timing optimizers at various phases of design, such as logic synthesis, layout (placement and routing), and in in-place optimizations performed late in the design cycle. While such timing measurements can theoretically be performed using a rigorous circuit simulation, such an approach is liable to be too slow to be practical. Static timing analysis plays a vital role in facilitating the fast and reasonably accurate measurement of circuit timing. The speedup comes from the use of simplified timing models and by mostly ignoring logical interactions in circuits. It has become a mainstay of design over the last few decades.

One of the earliest descriptions of a static timing approach was based on the Program Evaluation and Review Technique (PERT), in 1966[1]. More modern versions and algorithms appeared in the early 1980s.

1.2 Purpose and Definitions

In a synchronous digital system, data is supposed to move in lockstep, advancing one stage on each tick of the clock signal. This is enforced by synchronizing elements such as flip-flops or latches, which copy their input to their output when instructed to do so by the clock. Only two kinds of timing errors are possible in such a system:

- A **setup time violation**, when a signal arrives too late, and misses the time when it should advance;
- A **hold time violation**, when an input signal changes too soon after the clock's active transition.

The time when a signal arrives can vary due to many reasons - the input data may vary, the circuit may perform different operations, the temperature and voltage may change, and there are manufacturing differences in the exact construction of each part. The main goal of static timing analysis is to verify that despite these possible variations, all signals will arrive neither too early nor too late, and hence proper circuit operation can be assured.

Since STA is capable of verifying every path, it can detect other problems like glitches, slow paths and clock skew.

- The **critical path** is defined as the path between an input and an output with the maximum delay. Once the circuit timing has been computed by one of the techniques below, the critical path can easily be found by using a **traceback method**.
- The **arrival time** of a signal is the time elapsed for a signal to arrive at a certain point. The reference, or time 0.0, is often taken as the arrival time of a clock signal. To calculate the arrival time, delay calculation of all the components in the path will be required. Arrival times, and indeed almost all times in timing analysis, are normally kept as a pair of values - the earliest possible time at which a signal can change, and the latest.
- Another useful concept is **required time**. This is the latest time at which a signal can arrive without making the clock cycle longer than desired. The computation of the required time proceeds as follows: at each primary output, the required times for rise/fall are set according to the specifications provided to the circuit. Next, a backward topological traversal is carried out, processing each gate when the required times at all of its fanouts are known.
- The **slack** associated with each connection is the difference between the required time and the arrival time. A *positive slack* **s** at some node implies that the arrival time at that node may be increased by **s**, without affecting the overall delay of the circuit. Conversely, *negative slack* implies that a path is too slow, and the path must sped up (or the reference signal delayed) if the whole circuit is to work at the desired speed[2].

1.3 Method Analysis

Timing analysis computes the amount of time signals propagate in a circuit from its p r i mary 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. Further-more, assume that 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.

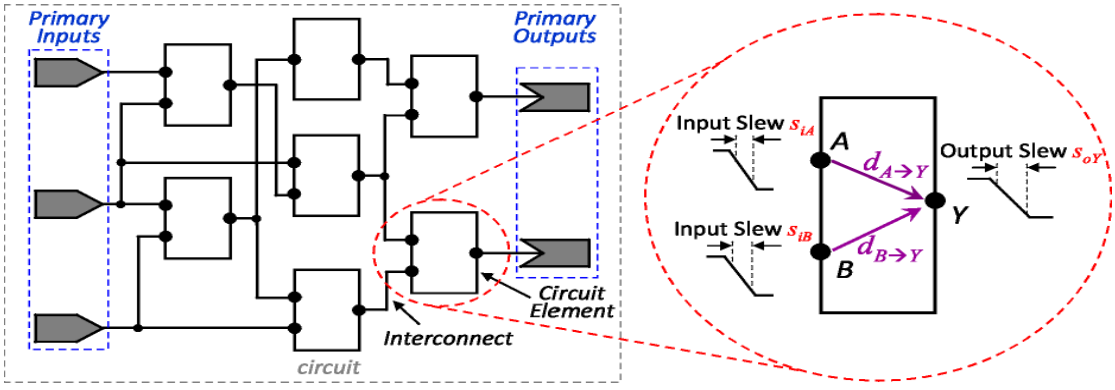

Figure 1 - Slews and delays in a circuit element.

In the figure, the delay across the circuit element from input A to output Y is designated by $d_{A\to Y}$, the input slew at A by $S_{iA,}$ and the output slew at Y by S_{oY} . Here, both the delay and the output slew are functions of input slew.

1.4 Timing Propagation

Starting from the primary input(s), we quantify the instant that a signal reaches an input or output of a circuit element as the arrival time (at). Similarly, starting from the primary output(s), we quantify the limits imposed for each arrival time to ensure proper circuit operation as the required arrival time (rat). Given an arrival time and a required arrival time, we define the slack at a circuit node as a measurement of how well timing constraints are met. That is, a positive slack means the required time is satisfied, and a negative slack means the required time is in violation.

To account for multiple sources of within-chip variation, such as manufacturing variations, temperature fluctuation, voltage drops, and electromigration, timing analysis is typically done using an early/late split, where each circuit node has an early (lower) bound and a late (upper) bound on its time. By convention, if the early or late mode is not explicitly stated, both modes will be need to be considered. For example, a generic output slew so that is a function of input slew s_i implies that the early mode s_o^{early} is a function of early mode s_i^{early} , and the late mode s_o^{late} is a function of late mode s_i^{late}.

Actual arrival time. Starting from the primary inputs, arrival times (at) 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 convergence point. That is, 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 in Figure(). Then the early mode arrival time at the output pin Y will be

$$at_Y^{early} = min(at_A^{early} + d^{early}_{A\to Y,}\ at_B^{early} + d^{early}_{B\to Y})$$

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 in Figure 1 (right), the late mode arrival time at Y will be

$$at_Y^{late} = max(at_A^{late} + d^{late}_{A\to Y,}\ at_B^{late} + d^{late}_{B\to Y})$$

Required arrival time. Starting from the primary outputs, required arrival times (rat) are computed by subtracting the delays across a path, and performing the maximum (in early mode) or minimum (in late mode) of such accumulated times at a convergence point. That is, in early mode, we are concerned with computing the earliest time instant that a signal transition must reach any circuit node. For example, in Figure 2 (left), the early mode required arrival time at the input pin Z will be

$$rat_z^{early} = \max(rat_{T1}^{early} - d^{early}_{Z\rightarrow T1}, rat_{T2}^{early} - d^{early}_{Z\rightarrow T2})$$

Conversely, in late mode, we are concerned with computing the latest time instant that a signal transition must reach any given circuit node. Following the same example in Figure 2 (left), the late mode required arrival time at the input pin Z will be

$$rat_z^{late} = \min(rat_{T1}^{late} - d^{late}_{Z\rightarrow T1}, rat_{T2}^{late} - d^{late}_{Z\rightarrow T2})$$

Slacks. For proper circuit operation, the following conditions must hold:

$$at^{early} \geq rat^{early}$$

$$at^{late} \leq rat^{late}$$

To quantify how well timing constraints are met at each circuit node, slacks (slack) can be computed based on equations for *at and rat* . That is, slacks are positive when the required times are met, and negative otherwise.

$$Slack^{early} = at^{early} - rat^{early}$$

$$Slack^{late} = rat^{late} - at^{late}$$

Slew propagation. As circuit element delays and interconnect delays are a function of the input slew (si), the subsequent output slew (so) must be propagated. In this contest, we will assume worst-slew propagation, where we propagate the smallest (largest) slew in early (late) mode. Following the example in Figure 1 (right), the early mode and late output slew at output pin Y are, respectively:

$$s_{oY}^{early} = \min(s_{oAY}^{early}(s_{iA}^{early}), s_{oBY}^{early}(s_{iB}^{early}))$$

$$s_{oY}^{late} = \max(s_{oAY}^{late}(s_{iA}^{late}), s_{oBY}^{late}(s_{iB}^{late}))$$

Transitions. For each timing arc, delay and output slew values will propagate only for transitions that exist. For example, suppose there two timing arcs in serial, where the first timing arc propagates rise-to-rise (R→R) and fall-to-fall (F→F), and the second timing arc propagates fall-to-rise (F→R). After timing analysis, the only valid output transition at the second timing arc will be rise (R). The delay through both the timing arcs is the

sum of the delay for the F→F transition in first arc plus the delay for the second arc for the F→R transition in the second arc. Note that the delay for the R→R delay from the first arc is not used, and the fall arrival time for the second arc is undefined. For this contest, an undefined early (late) arrival time is set as 987654 (-987654), and an undefined early (late) required arrival time is set as (-987654) (987654).

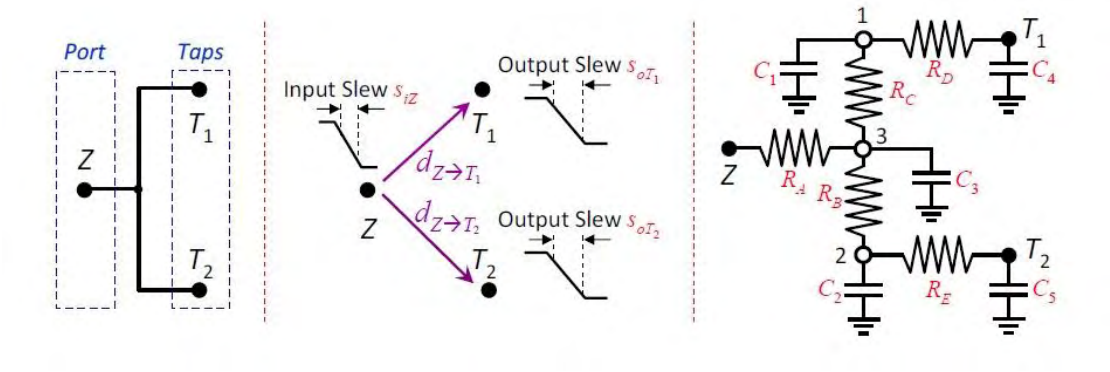

Figure 2 : Generic interconnect (left), its timing model (center) and RC network (right).

1.5 Interconnect Modeling

The basic instance of interconnect (wire) is a net, which is assumed to have an input pin (Port) and one or more output pins (Taps), as illustrated in Figure 2 (left). Parasitic RC trees only contain grounded capacitors and floating resistors (we will not include the discussion of coupling capacitors or grounded resistors).

Delay. The computation of port-to-tap delays can be accurately performed through electrical simulation. However, and for the sake of simplicity (and speed), we will assume the simpler Elmore delay model [3], where the delay is approximated by the symmetric of the value of the first moment of the impulse response. To compute the delay of RC tree networks, we summarize the topological method [4].

In an RC network, consider any two nodes e and k. Let Ck be the lumped capacitance at node k, and let $R_{k->e}$ be the total resistance of the common path between the paths from Port to e and Port to k. For example, in Figure 2 (right), the resistance between nodes 1 and T2 (R1->T2) is RA, as that is the only common resistor between the paths Z to 1 and Z to T2. The

Elmore delay at node e is

$$d_e = \sum_{k \in N} R_{k \to e} \; C_k$$

where N is the set of all nodes in the RC network. For the example net illustrated in Figure (right), the delay at node T2 (tap) is (visiting in order nodes 1, T1, 3, 2, T2):

$d_{T2} = R_A C_1 + R_A C_3 + R_A C_4 + (R_A + R_B) C_2 + (R_A + R_B + R_E) C_5 = R_A (C_1 + C_3 + C_4) + (R_A + R_B) C_2 + (R_A + R_B + R_E) C_5$

Output slew. The value of the output slew (s_o) on any given tap node T can be approximated by a two-step process. First, compute the output slew of the impulse response on T, which was observed to be well-approximated by

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

where β_T is the second moment of the input response at node T, and d_T is the corresponding Elmore delay from Equation . Second, compute the slew of the response to the input ramp by the expression given :

$$\hat{S}_{oT} \approx \sqrt{s_i{}^2 - \hat{s}_{oT}{}^2}$$

where s_i is the input slew.

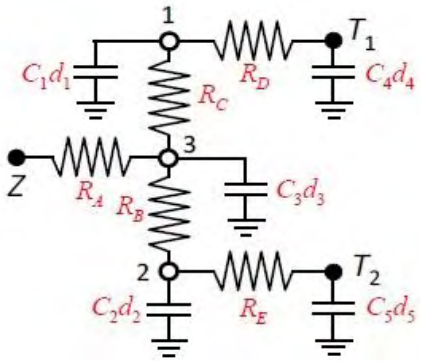

Figure 3 . RC tree

Modified RC network for output slew calculation

The value of β_T can be computed through the efficient path-tracing algorithm for moment computation proposed in [5], which is a generalization of the algorithm proposed in [1]. To calculate β_T, first replace all capacitance values C_k in the RC network by $C_k d_k$ where d_k is the Elmore delay. Second, follow the same procedure as before for finding β_T

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

At node T_2 we have :

$\beta_{T2} = R_A (C_1 d_1 + C_3 d_3 + C_4 d_4) + (R_A + R_B) C_2 d_2 + (R_A + R_B + R_E) C_5 d_5$

1.6 Circuit Element Modeling

For delay and output slew calculations between two pins, the information will be given in the .lib file as two-dimensional tables. To find the corresponding timing information, extrapolation or interpolation will be necessary.

If the table contains a single value, i.e., a 1x1 table (Figure 4 left), no interpolation is necessary. That is, regardless of input x and y, the corresponding value is constant. If the table is one-dimensional, i.e., a 1x*n* table or a *m*x1 table (Figure 4 center), then the value will depend only on the non-scalar dimension. For example, consider the 1x4 table in Figure 4. If $y < y1$, then the corresponding output z value will be the linear extrapolation between $z1$ and $z2$. If $y2 \le y \le y3$, then z will be the linear interpolation between $z2$ and $z3$.

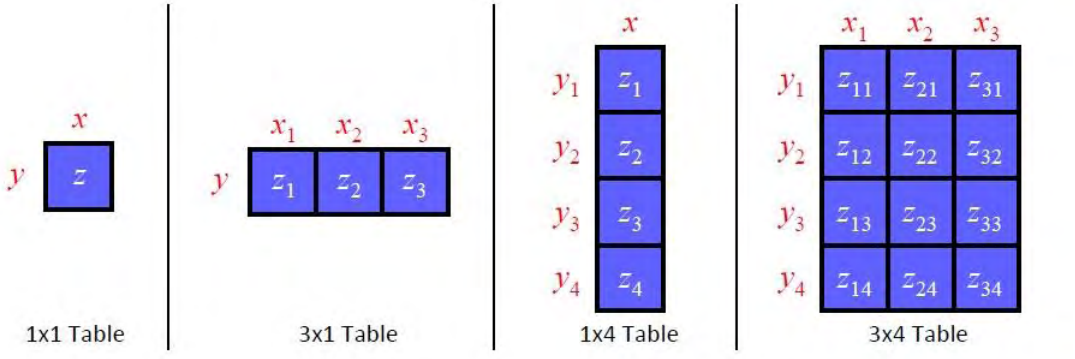

Table 1.Illustration of different tables: scalar, one-dimensional, and two-dimensional.

If $y4 < y$, then z will be the linear extrapolation between $z3$ and $z4$.

$$
\begin{aligned}
z_1 - (y_1 - y)\frac{z_2 - z_1}{y_2 - y_1} &\quad \text{if} \quad y < y_1 \\
z_1 &\quad \text{if} \quad y = y_1 \\
z_1 + (y - y_1)\frac{z_2 - z_1}{y_2 - y_1} &\quad \text{if} \quad y_1 < y < y_2 \\
z_2 &\quad \text{if} \quad y = y_2 \\
z_2 + (y - y_2)\frac{z_3 - z_2}{y_3 - y_2} &\quad \text{if} \quad y_2 < y < y_3 \\
z_3 &\quad \text{if} \quad y = y_3 \\
z_3 + (y - y_3)\frac{z_4 - z_3}{y_4 - y_3} &\quad \text{if} \quad y_3 < y < y_4 \\
z_4 &\quad \text{if} \quad y = y_4 \\
z_4 + (y - y_4)\frac{z_4 - z_3}{y_4 - y_3} &\quad \text{if} \quad y > y_4
\end{aligned}
$$

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

Combinational elements. For a given combinational cell, e.g., OR gate, let the delay d and output slew s_o for a input/output pin-pair (see Figure) be calculated by non-linear delay model interpolation/extrapolation. These delay and output slew tables are stored in the .lib, and are referenced by the input slew (x) and driving load (y). C_L denotes the equivalent downstream capacitance seen from the output pin of the cell. Several sophisticated models have been proposed for computing C_L. For simplicity, the application of such models is considered to be out of the scope of the present contest, and a simple model is adopted. C_L is assumed to be the sum of all the capacitances in the parasitic RC tree, including the cell pin capacitances at the taps of the interconnect network.

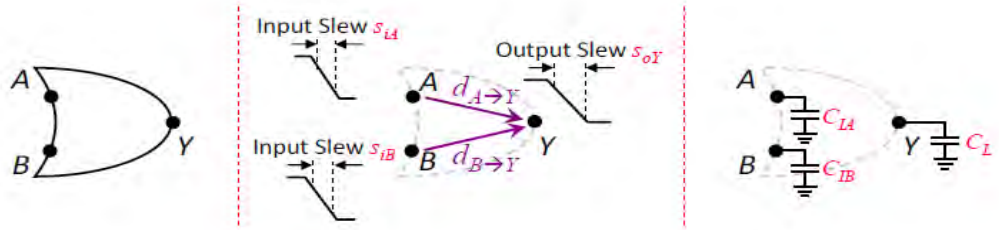

Figure 4: Combinational OR gate (left), its timing model (center) and capacitances (right).

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

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

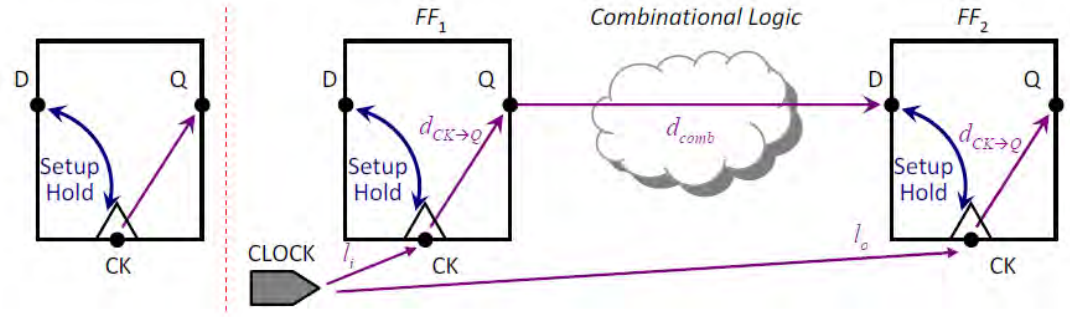

Figure 5:Generic D flip-flop and its timing model (left), and two FFs in series and their timing models (right).

Setup and hold constraints. Proper operation of a flip-flop requires the logic value of the input data pin to be stable for a specific period of time before the capturing clock edge. This period of time is designated by the *setup time* t_{setup} .Additionally, the logic value of the input data pin must also be stable for a specific period of time after the capturing clock edge. This period of time is designated by the *hold time* t_{hold}.The flip-flop timing models are depicted in

---

The complement, preset and clear signals are stated here for completeness. For the purposes of the contest, their behavior will be ignored.

Figure 5 (left). The test time are given in the **.lib** as two-dimensional tables, and are referenced by the clock-side input slew (*x*) and the data-side input slew (*y*).

Signal propagation. Consider the standard signal transition between two flip-flops as illustrated in Figure 5 (right). Assuming that the clock edge is generated at the source at time 0, it will reach the injecting (launching) flip-flop F F $_1$ at time l_i, making the data available at the input of the combinational block $d_{CK\to Q}$ time later. If the propagation delay in the combinational block is d_{comb}, then the data will be available at the input of the capturing flip-flop F F $_2$ at time $l_i + d_{CK\to Q} + d_{comb}$. Let the clock period to be a constant *T*. Then the next clock edge will reach FF $_2$ at time $T + l_o$. For correct operation, the data must be be available at the input pin *D* of FF_2 t_{setup} time before the next clock edge. Therefore, at the data input pin *D* of FF_2, we have the following :

$$at_D^{late} = l_i^{late} + d_{CK\to Q} + d_{comb}^{late}$$
$$rat_{setup} = rat_D^{late} = T + l_o^{early} - t_{setup}$$

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

$$at_D^{early} = l_i^{early} + d_{CK\to Q} + d_{comb}^{early}$$
$$rat_{hold} = rat_D^{early} = l_o^{late} - t_{hold}$$

Note that when computing the required arrival times in Equations 27 and 29, the value l_o is specific to Figure 6. In the general case, l_o should be replaced with at_C. The previous arrival times and required arrival times induce setup and hold slacks, which can be computed from Equations 7 and 8. For the clock pins of the flip-flop, the required arrival time is derived from the test slack. For early mode, the slack at the clock pin is the setup or late test slack, and for late mode, the slack at the clock pin is the hold or early test slack. From the corresponding test slack and arrival time, the clock required arrival time can be derived, and appropriately propagated.

Chapter 2

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) [6], 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 6). The technique has since been adopted as a basis for numerous CAD tools, for the sake of its simplicity.

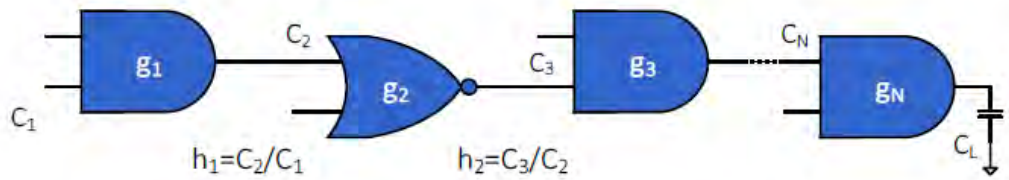

Figure 6 - Logical effort optimization for gates without wires is based on equal stage efforts, g1h1=g2h2 etc.

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 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:

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

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:

$$d = f + p$$

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:

$$f = g \times h$$

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

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

where $\boldsymbol{Cout}$ is the capacitance that loads the output of the logic gate and $\boldsymbol{Cin}$ 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 $\boldsymbol{\tau}$:

$$d = g \times h + p$$

This equation shows that logical effort $\boldsymbol{g}$ and electrical effort h both contribute to delay in the same way. This formulation separates $\boldsymbol{\tau}$, , $\boldsymbol{h}$, and $\boldsymbol{p}$, the four contributions to delay. The process parameter $\boldsymbol{\tau}$ represents the speed of the basic transistors. The parasitic delay **p** 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, $\boldsymbol{h}$, combines the effects of external load, which establishes $\boldsymbol{Cout}$, with the sizes of the transistors in the logic gate, which establish $\boldsymbol{Cin}$. The logical effort $\boldsymbol{g}$ 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 2 - 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 2. 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.

---

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.

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 2 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. $G=\Pi gi$

$$G = \prod g_i$$

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. $H= C_{out}/C_{in}$

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

In this case, $\boldsymbol{Cin}$ and $\boldsymbol{Cout}$ 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

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

where $Con-path$ is the load capacitance along the path we are analyzing and $Coff-path$ 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.

$$B = \prod b_i$$

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:

$$F = G \times B \times H$$

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

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

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 , we shall distinguish the path effort delay D_F and the path parasitic delay P:

$$D = \sum d_i = D_F + P$$

The path effort delay is simply:

$$D_F = \sum g_i \times h_i$$

and the path parasitic delay is:

$$P = \sum p_i$$

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:

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

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:

$$\widehat{D} = N \times F^{1/N} + P$$

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

$$\widehat{h_i} = \frac{F^{1/N}}{g_i}$$

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:

$$C_{in_i} = \frac{g_i \times C_{out_i}}{\hat{f}}$$

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 7). This behavior is described by the authors of the LE method as “one of the most dissatisfying limitations of logical effort”.

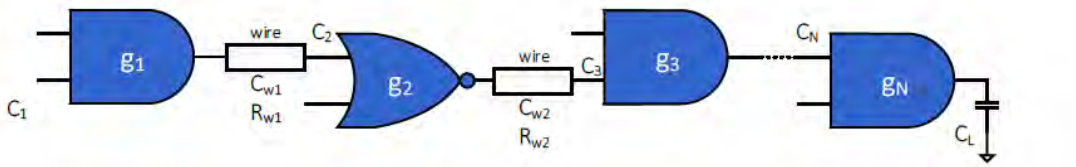

Figure 7 – In the case of gates with wires, the rule of equal effort breaks down because of fixed wire parameters.

2.2.2 Delay Model of Logic Gates with Wires

The logical effort model is modified to include the interconnect delay [7]. 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 delay model the delay of a circuit comprising logic gates and wires (see Figure 8) can be easily calculated

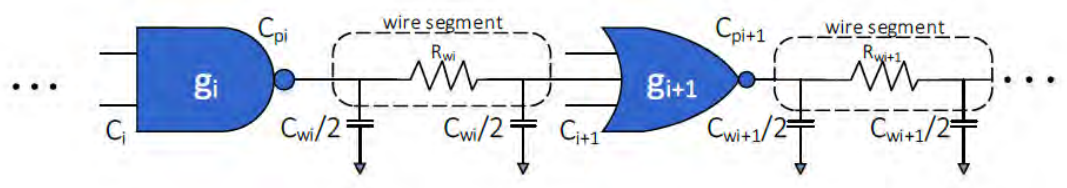

Figure 8 - Cascaded logic gates with resistive-capacitive interconnect.

The total combined delay expression is:

$Di = Ri \times (Cpi + Cwi + Ci{+}1) + Rwi \times (0.5 \times Cwi + Ci{+}1)$

where $\boldsymbol{R_i}$ is the effective output resistance of the gate $\boldsymbol{i}$, $\boldsymbol{Cp_i}$ is the parasitic output capacitance of gate $\boldsymbol{i}$, $\boldsymbol{Cw_i}$ and $\boldsymbol{Rw_i}$ are, respectively, the wire capacitance and resistance of segment $\boldsymbol{i}$, and $\boldsymbol{C_{i+1}}$ is the input capacitance of gate $\boldsymbol{i}$**+1** .

This expression can be rewritten similar with the function of the delay of a minimum sized inverter $\boldsymbol{\tau}$ **=***R0C0*, where **R0** and **C0** are the output resistance and input capacitance of a minimum sized inverter:

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

The delay $\boldsymbol{d_i}$ normalized with respect to a minimum sized inverter delay $\boldsymbol{\tau}$ is defined by:

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

Where,

$gi{=}(Ri{\times}Ci)/(R0{\times}\ C0)$ is the logical effort ,
$hi{=}Ci{+}1/Ci$ is the electrical effort,
$pi{=}(Ri{\times}Cpi)/(R0{\times}\ C0)$ is the parasitic delay.

The capacitive interconnect effort $\boldsymbol{h_w}$ and the resistive interconnect effort $\boldsymbol{p_w}$ are, respectively:

$$h_{wi} = \frac{C_{wi}}{C_i}$$

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

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:

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

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

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

Note that in the case of short wires, the resistance Rw of the wire may be neglected, eliminating pw and leaving only the capacitive interconnect effort hw 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 , the resulting optimum condition is:

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

The meaning of the optimum size of gate i+1 is achieved when the delay component $(\boldsymbol{R_i + R_{wi}}) \times \boldsymbol{C_{i+1}}$ due to the gate capacitance is equal to the delay component $\boldsymbol{R_{i+1}} \times (\boldsymbol{C_{i+2}} + \boldsymbol{C_{w_{i+1}}})$ due to the effective resistance of the gate. A schematic model describing the related delay components is shown in Figure 9.

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

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

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

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.

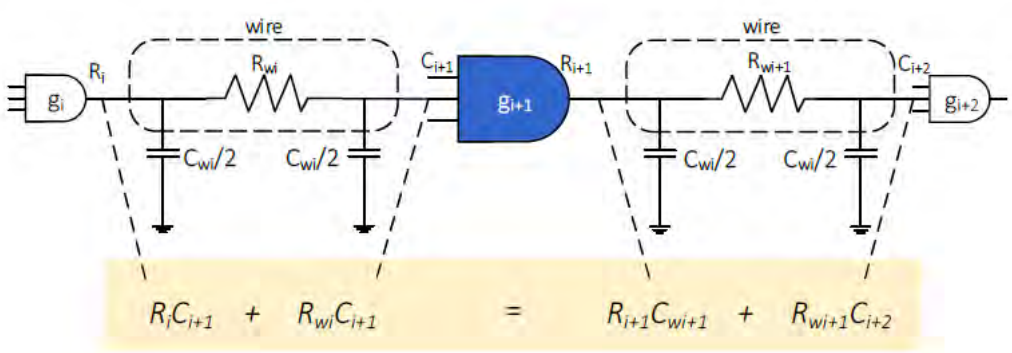

Figure 9 : Delay components in characterizing ULE for long wires.

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:

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

where $\tau = R0 \times C0$ 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:

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

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

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:

$$C_{BF} = \sum_{1}^{n} C_{b_n} + \sum_{1}^{m} C_{f_m}$$

where $\boldsymbol{n}$ and $\boldsymbol{m}$ 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 :

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

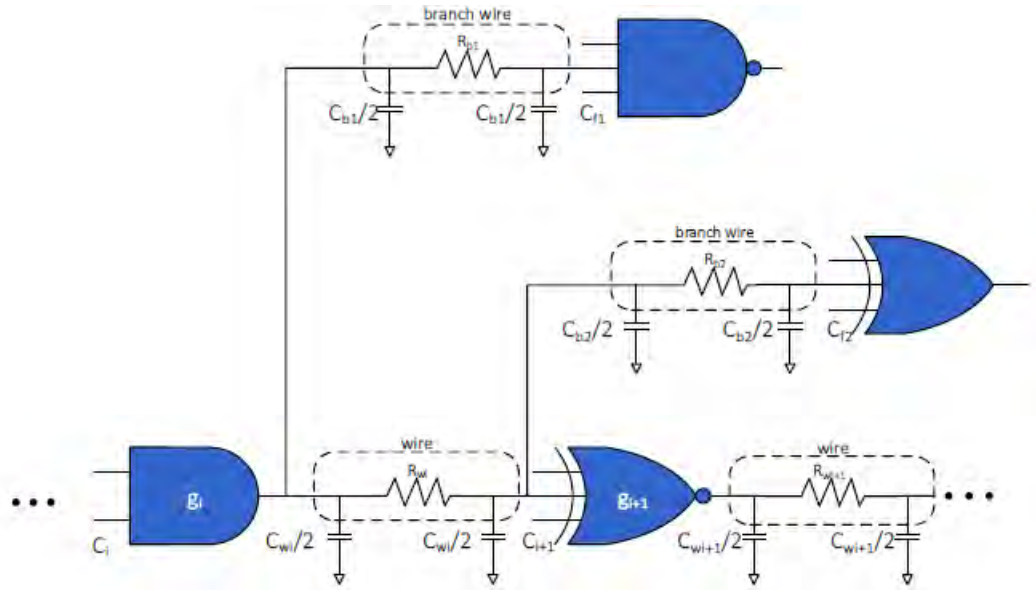

Figure 10 : A logic path segment including RC interconnect and two branches.

In the case of a more complex parasitic tree (see Figure 11), 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,

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

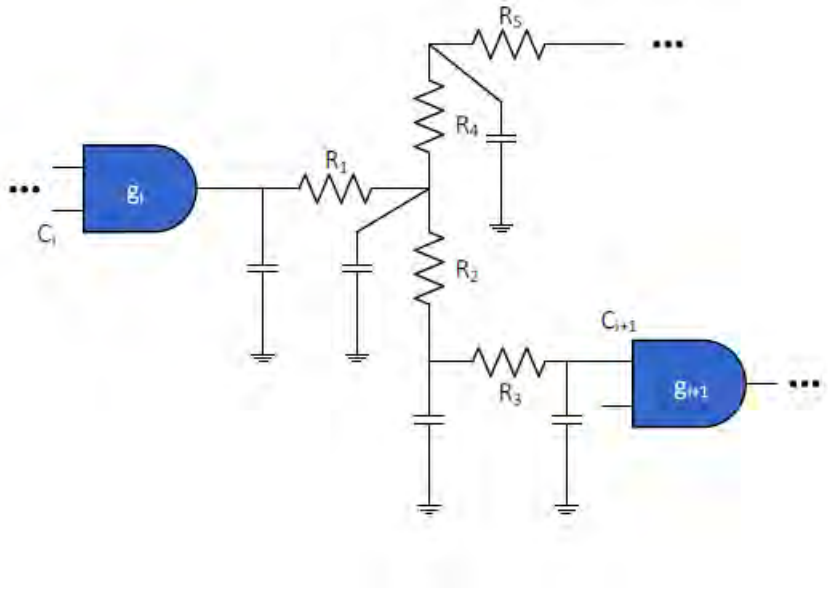

Figure 11 - $R_{wi} = R_1 + R_2 + R_3$.

In order to simplify the solution, a relaxation method is proposed in [8]. 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.

The following chapter introduce the input files that needed for the resizing tools in order to perform the implementation of the static timing analysis and resizing methods.

Chapter 3

3.1 Input Files

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 a module declaration, defining the interface with of the module with name <circuit name>. The inputs and output pins are explicitly declared; the internal 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 12 .

```
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 22 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.

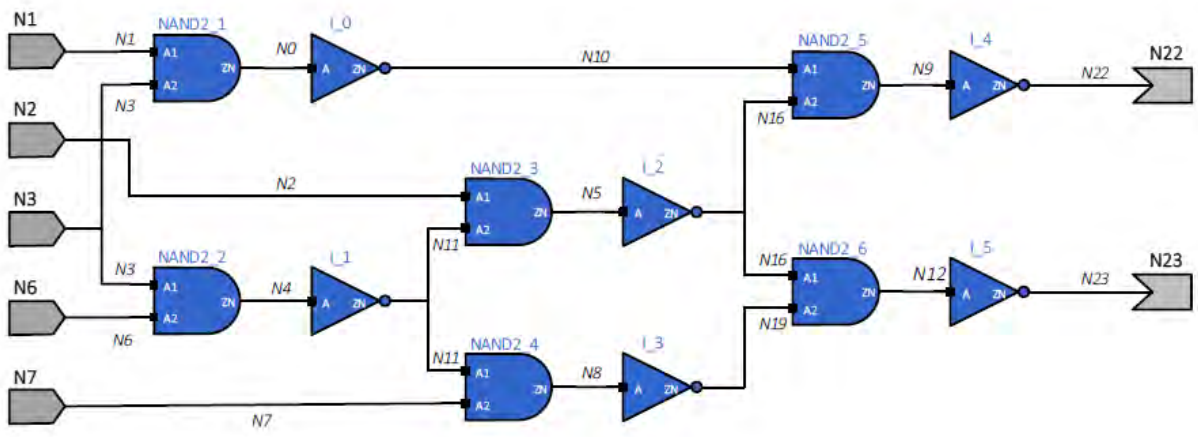

Figure 12 - Implementation of c17.v.

```
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 ;
14. DFF_X1 q_reg(.CK (clk), .D (d), .Q (q), .QN ())q;
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).

3.2 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.

---

1. *SPEF "IEEE 1481-1998"
2. *DESIGN "c17"
3. *DATE "Thu Sep 25 17:47:29 2014"
4. *VENDOR "Cadence Design Systems, Inc."
5. *PROGRAM "Encounter"
6. *VERSION "13.13-s017_1"
7. *DESIGN_FLOW "PIN_CAP NONE" "NAME_SCOPE LOCAL"
8. *DIVIDER /
9. *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:

```
1.  // MMMC spef file for corner 'typ'
2.
3.  *NAME_MAP
4.  *1 N1
5.  *2 N2
6.  *3 N3
7.  *4 N6
8.  *5 N7
9.  *6 N22
10. *7 N23
11. *8 N0
12. *9 N4
13. *10 N5
14. *11 N8
15. *12 N9
16. *13 N12
17. *14 N10
18. *15 N11
19. *16 N16
20. *17 N19
21. *18 I_5
22. *19 NAND2_6
```

23. *20 I_4
24. *21 NAND2_5
25. *22 I_3
26. *23 NAND2_4
27. *24 I_2
28. *25 NAND2_3
29. *26 I_1
30. *27 NAND2_2
31. *28 I_0
32. *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>

* CO N N

<pin type> <pin name> <pin direction>

// more pin definitions

*CAP

<integer label> <pin or node name> <pin or node capacitance>

// more capacitor definitions

* R ES

<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. * CO N N
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. * R ES
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 design 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.

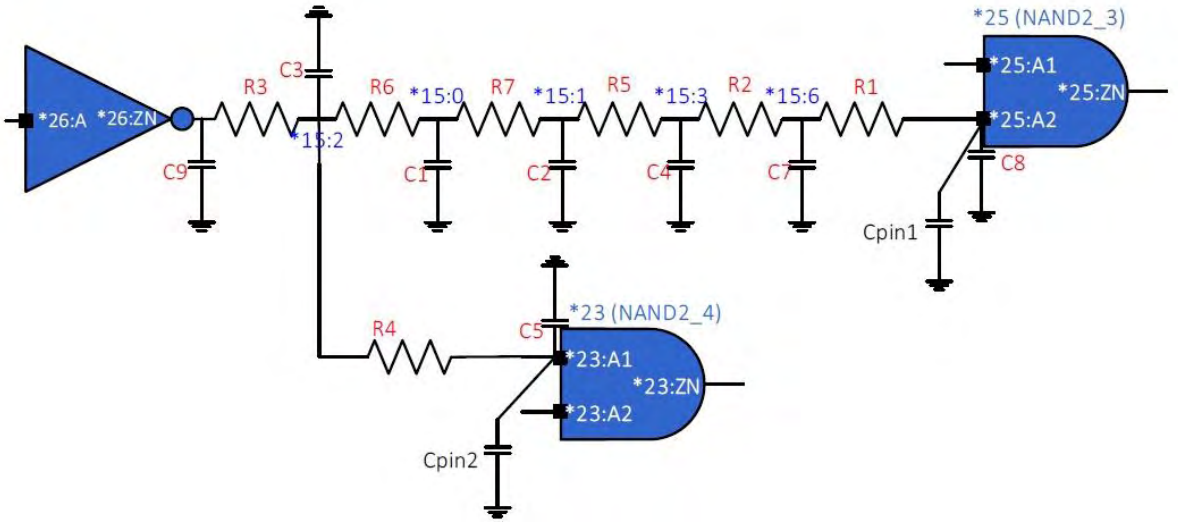

Figure 13 - Parasitics of net *15 (N11). The R (C) labels refer to resistors (capacitors).

3.3 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. ll 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:

```
1. lu_table_template (delay_template_3x3) {
2. variable_1 : input_net_transition;
3. variable_2 : total_output_net_capacitance;
4. index_1 ("1000,1001,1002");
5. index_2 ("1000,1001,1002");
6. }
```

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:

```
1. lu_table_template(delay_template_3x3) {
2. variable_1 : input_net_transition;
3. va ria ble_2 : total_output_net_capacitance;
4. index_1 ("0.1, 0.3, 0.7");
5. index_2 ("0.16, 0.35, 1.43");
6. }
```

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

```
1.  cell(OR2_X2) {
2.     pin ("o") {
3.      direction : output ;
4.      capacitance : 2.00 ;
5.       timing() {
6.         related_pin : "a";
7.          timing_sense : positive_u nate;
```

```
8.          timing_type : combinational;
9.          cell_fall (scalar) {
10.           values ("40.00");
11.         }
12.         fall_transition (delay_slew_load_6x1) {
13.           index_1 ("1.050, 2.000, 5.000, 5.500, 9.000, 20.00");
14. index_2 ("1.0000");
15.           values ( \
16.             "1.050000", \
17.             "2.000000", \
18.             "5.000000", \
19.             "5.500000", \
20.             "9.000000", \
21.            "20.000000" \
22.           );
23.         }
24.      }
25.    }
26. }
```

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.

3.4 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>

Chapter 4

4.1 OpenTimer : Timing Analysis Tool

4.1.1 Introduction

OpenTimer is a high-performance academic timing analysis tool developed by Tsung-Wei Huang and Prof. Martin D. F. Wong in the University of Illinois at Urbana-Champaign (UIUC), IL, USA. Evolving from its previous generation "UI-Timer", OpenTimer works on industry formats (.v, .spef, .lib, .sdc, .lef, .def), and supports important features such as block-based analysis, path-based analysis, cppr, incremental timing, and multi-threading. OpenTimer is extremely fast by its effective data structure and algorithm which can efficiently and accurately analyze large-scale designs. To further facilitate seamless integration between timing and other electronic design automation (EDA) applications such as timing-driven placement and routing, OpenTimer provides user-friendly application programming inteface (API) for interactive analysis. Most importantly, OpenTimer is open-source [9].

Experimental results on industry benchmarks released from TAU 2015 timing analysis contest have demonstrated remarkable results achieved by OpenTimer, especially in its order-of-magnitude speedup over existing timers.

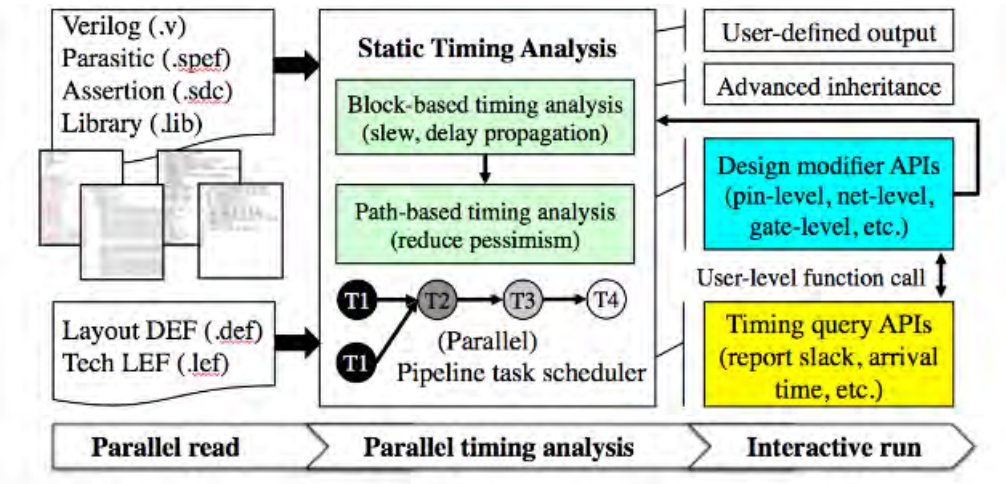

Figure 14 : Program flowchart of OpenTimer.

In deep submicron era, timing-driven operations are imperative for the success of optimization flows. Optimization transforms change the design and therefore have the potential to significantly affect timing information. The timer must reflect such changes and update timing information incrementally and accurately in order to ensure slack integrity as well as reasonable turnaround time and performance.

However, such process requires extremely high complexity especially when path-based analysis is configured. A high-quality incremental timer capable of path-based analysis is definitely advantageous in speeding up the timing closure.

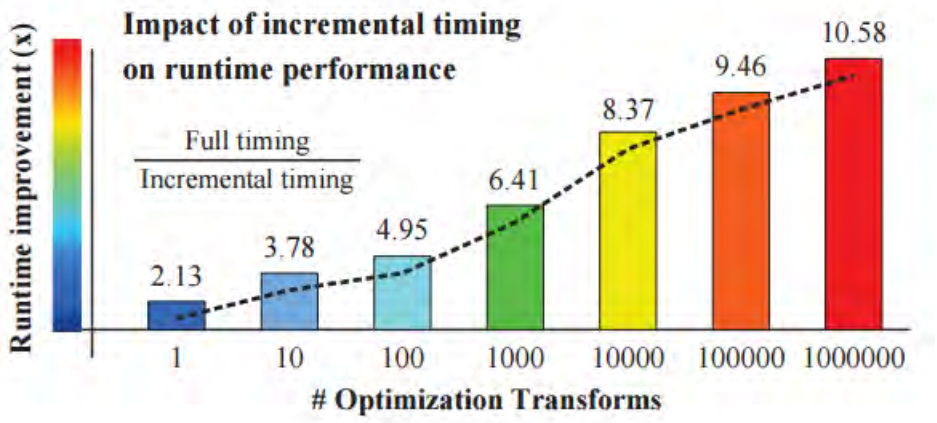

Figure 15. Performance improvement of incremental timing to full timing

The significance of incremental timing is demonstrated in Figure 1. It is observed that the runtime improvement keeps growing as the number of optimization transforms increases. One obvious reason is that once the critical paths in a design have been reported, the optimization tool would optimize the logic (e.g., gate sizing, buffer insertion) so as to overcome the timing violations. This subtle change can affect up to the majority of a circuit, whereas in reality, depending on the trace of critical paths, the timing update may only involve a small portion of the circuit. Since an optimization tool can perform millions of logic transformations, it is important that the timing profile is kept up-to-date in an incremental fashion. Otherwise, optimization tools cannot support fast turnaround for timing-specific improvement, which dramatically degrades the productivity.

Three main key features of OpenTimer are:

• Parallel framework. OpenTimer applies a pipeline task scheduler as the central engine. Critical tasks such as timing propagation and endpoint slack calculation are scheduled into the pipeline so as to overlap their runtimes.

• Incremental capability. OpenTimer precisely and minimally captures the features that are key to incremental timing. With lazy evaluation, we are able to keep computation as minimum as necessary.

• Path-based analysis. OpenTimer represents the path implicitly using efficient and compact data structure, yielding a significant saving in both search space and search time for CPPR.

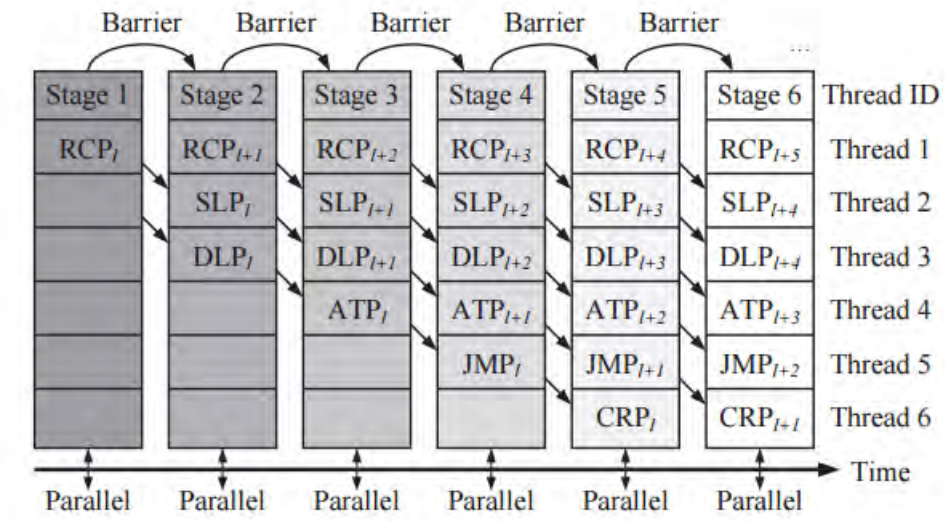

Figure 16. Parallel forward timing propagation using pipeline

The effectiveness and efficiency of our timer have been evaluated on a set of industry benchmarks released from TAU 2015 CAD contest. Compared to the top performers in TAU 2015 CAD contest, OpenTimer confers a high degree of differential in nearly all aspects. The source code of OpenTimer has been released to the public domain for promoting further research [10].

4.1.2 Purpose

The purpose of this thesis is to extend the Open Timer timing analysis tool in order to get critical paths with positive slacks so we can perform the Unified Logical Effort (ULE) and resizing method. For this purpose it is necessary to parse the minimum scale factor (**min_scf.scf**) for every cell, to set a unit inverter and calculate inverter's values in order to proceed to Logical Effort's parameters extraction for every cell of our **.lib** file.

4.1.3 Find critical paths with positive slacks

First we declare the (Path*) object *critical_path :*

```
private:

  Path* critical_path = new Path();
```

Then we iterate the endpoint vector in order to get the nodes of the path. We perform backward tracing by checking the unateness of the node that we point every time in order to get the correct previous one, until we reach a primary input:

```
53 void resizer::findCriticalPaths( endpoint_pt v, Path* path ) {
54
55     int tmp_slack=(v)->slack();
56     Node* n = v->constrained_pin_ptr()->node_ptr() ;
57
58
59     if (n != NULL && tmp_slack <= (v)->slack()) {
60
61         path->nodes.push_back(n);
62         cout << "pin_name : " << n->pin_ptr()->name() <<  endl;
63     }
64
65     int state = (v)->rf();
66     int state_tmp;
67
68
69     switch ((v)->el()) {
70
71     case EARLY:
72             while (n != NULL) {
73
74                 if(state  == 0 ) {
75
76                     if (n->invRise_e() == false) {
77                         state_tmp = 0;
78                     }
79
80                     else if (n->invRise_e() == true) {
81                         state_tmp = 1;
82                     }
83                     n = n->preRise_e();
84                 }
```

```
87                 else {
88
89                  if (n->invFall_e() == false) {
90                     state_tmp = 1;
91                 }
92
93
94                 else if (n->invFall_e() == true) {
95                     state_tmp = 0;
96                 }
97                 n = n->preFall_e();
98             }
99
100            if (n != NULL) {
101
102                if (tmp_slack <= (v)->slack()) {
103                path->nodes.push_back(n);
104                }
105                cout << "pin_name : " << n->pin_ptr()->name()  << endl;
106                state=state_tmp;
107
108            }
109
110        }
111
112    case LATE:
113            while (n != NULL) {
114
115                if(state  == 0 ) {
116
117                    if (n->invRise_l() == false) {
118                        state_tmp = 0;
119                    }
120                    else if (n->invRise_l() == true) {
121                        state_tmp = 1;
122                    }
123                    n = n->preRise_l();
124
125                }
126
127
128                else {
129
130                     if (n->invFall_l() == false) {
131                        state_tmp = 1;
132                    }
133
134
135                    else if (n->invFall_l() == true) {
136                        state_tmp = 0;
137                    }
138                     n = n->preFall_l();
139
140                }
141
142                if (n != NULL) {
143                    if (tmp_slack <= (v)->slack()) {
144                        path->nodes.push_back(n);
145                    }
146                    cout << "pin_name : " << n->pin_ptr()->name()  << endl;
147                    state=state_tmp;
148
149                }
150            }
151    }
152 }
```

4.1.4 Minimum scale factor file parser

```
384 void_t Timer::set_min_scf(char* arg) {
385
386     cout << "\nParsing the minimum_scf file...\n\n";
387
388         ifstream file;
389         string buf;
390         double min_scf;
391
392
393         LibraryCellIter *cell_it = new LibraryCellIter(_celllib_uptr[0].get());
394
395         file.open( arg );
396         if (!file.good())
397         {
398             cout << "The file  was not found.\n";
399             cout << "Terminating.\n";
400             exit( -1 );
401         }
402
```

```
402
403          while (true)
404          {
405              getline( file, buf );
406
407              if (file.eof())
408                  break;
409
410              typedef tokenizer<char_separator<char> > tokenizer;
411              char_separator<char> sep( " \t" );
412              tokenizer tokens( buf, sep );
413
414              tokenizer::iterator tok_iter = tokens.begin();
415
416              if (tok_iter != tokens.end())
417              {
```

```
418
419                 cell_it->begin();
420                 while(cell_it->operator ()()) {
421
422                 /* If it  exist, assign its min_scf */
423                 if ( cell_it->cell_ptr()->name().compare(*tok_iter)==0) {
424
425                     min_scf = atof((*(++tok_iter)).c_str());
426                     cell_it->cell_ptr()->set_min_scf(min_scf);
427
428                     }
429
430                 cell_it->operator ++() ;
431                 }
432
433
434             }
435
436         }
437
438         cell_it->begin();
439         while (cell_it->operator ()()) {
440             cout << "CELL : " << cell_it->cell_ptr()->name() <<" SCF  ::  " << cell_it->cell_ptr()->ret_min_scf() << endl;
441
442             cell_it->operator ++();
443         }
444
445         file.close();
446
447         return;
448 }
```

4.1.5 Setting the unit inverter

In order to proceed, we have to set our unit inverter, which is the inverter "*INV_X1*", in order to calculate it's values :

```
public:

  typedef struct _unit_inverter
              {
                Cell* cell;
                double a_inv_f; // Parasitic delay fall
                double a_inv_r; // Parasitic delay rise
                double b_inv_f; // Logical Effort fall
                double b_inv_r; // Logical Effort rise
                double C0;
                double tau;
              } unit_inverter;

              unit_inverter  inv;
```

Setting the unit inverter which is defined from the **.conf** file:

```
334 void_t Shell::_feed_set_unit_inverter(int argc, char **argv, timer_pt timer) {
335     if (_is_argc_valid(argc, 2)) {
336
337         cout << "UNIT INVERTER HAS BEEN DEFINED " << endl;
338         res->set_inv(timer->cell_ptr(1,argv[1]));
```

```
156 void resizer::set_inv(Cell* cell_p) {
157
158     inv.cell=cell_p;
159
160 }
```

4.1.6 Unit inverter's values

Next step is to calculate inverter's **parasitic delay** (rise/fall), **logical effort** (rise/fall) the **.C0** and **.tau** value. For this function we need the timing look up tables (rise/fall) in order to perform the inter-extra polation.

```
171  void resizer::calcUnitInverterVals() {
172
173
174      cout << inv.cell->cellpin_dict_ptr()->begin()->second->name() << endl;
175      cellpin_pt pin_ptr = inv.cell->cellpin_ptr(inv.cell->cellpin_dict_ptr()->begin()->second->name());
176      cell_pt cell_pt = pin_ptr->cell_ptr();
177      timing_lut_pt lut_pt_r;
178      timing_lut_pt lut_pt_f;
179
180
181      double dr = -numeric_limits<double>::max(), df = -numeric_limits<double>::max();
182      inv.C0 = pin_ptr->capacitance();
183
184
185      CellpinIter *pin_it = new CellpinIter(cell_pt);
186
187      while(pin_it->operator ()()) {
188
189          cout << "in cell pin iteration" << endl;
190          if (pin_it->cellpin_ptr()->direction() == 3) {
191              //cout << "In output pin " << endl;
192              //cout <<"cellrise_ptr addr: " <<pin_it->cellpin_ptr()->timingset_ptr()->begin().operator *()->cell_rise_ptr() << endl;
193              lut_pt_r = pin_it->cellpin_ptr()->timingset_ptr()->begin().operator *()->cell_rise_ptr();
194              lut_pt_f = pin_it->cellpin_ptr()->timingset_ptr()->begin().operator *()->cell_fall_ptr();
195          }
196
197          pin_it->operator ++();
198
199      }
200
201
202       CHECK(lut_pt_r != nullptr);
```

```
203
204         float_t temp = lut_pt_r->lut_polation(inv.C0, 0);
205         dr = temp > dr ? temp : dr;
206        cout << "delay rise : " << dr << endl;
207
208
209         CHECK(lut_pt_f != nullptr);
210
211         float_t temp2 = lut_pt_f->lut_polation(inv.C0, 0);
212         df = temp2 > df ? temp2 : df;
213         cout << "delay fall : " << df << endl;
214
215
216
217         if (df != -numeric_limits<double>::max() && dr != -numeric_limits<double>::max())
218            {
219                 inv.tau = (df + dr) / 2;
220            }
221         else if (df == -numeric_limits<double>::max() && dr != -numeric_limits<double>::max())
222            {
223                inv.tau = dr;
224            }
225         else if (dr == -numeric_limits<double>::max() && df != -numeric_limits<double>::max())
226            {
227                inv.tau = df;
228            }
229         else
230            {
231                inv.tau = 0;
232            }
233
234         cout << "inv.tau : " <<inv.tau << endl;
235         cout << "inv.C0  : " << inv.C0 << endl;
236
```

```
224             }
225          else if (dr == -numeric_limits<double>::max() && df != -numeric_limits<double>::max())
226             {
227                 inv.tau = df;
228             }
229          else
230             {
231                 inv.tau = 0;
232             }
233
234          cout << "inv.tau : " <<inv.tau << endl;
235          cout << "inv.C0  : " << inv.C0 << endl;
236
237
238
239          /******************NLDM_to_LDM_conv*****************************/
240
241         double af, ar, br, bf;
242
243         NLDM_to_LDM_conv(pin_ptr ,af, ar, br, bf,inv.cell);
244
245         inv.a_inv_f = af;
246         inv.a_inv_r = ar;
247         inv.b_inv_f = bf;
248         inv.b_inv_r = br;
249
250
251         inv.cell->cellpin_ptr(inv.cell->cellpin_dict_ptr()->begin()->second->name())->set_Pfall(1.0);
252         inv.cell->cellpin_ptr(inv.cell->cellpin_dict_ptr()->begin()->second->name())->set_Prise(1.0);
253         inv.cell->cellpin_ptr(inv.cell->cellpin_dict_ptr()->begin()->second->name())->set_Gfall(1.0);
254         inv.cell->cellpin_ptr(inv.cell->cellpin_dict_ptr()->begin()->second->name())->set_Grise(1.0);
255
256     }
```

In order to calculate the parasitic delay values (rise/fall) and logical effort values (rise/fall) we need to call the *NLDM_to_LDM_conv* function, the Non-Linear-Delay-Model to Linear Delay Model conversion. **a** stands for the parasitic delay, **b** stands for the logical effort delay (ps).

```
261     void resizer::NLDM_to_LDM_conv(cellpin_pt related_pin,double& af, double& ar, double& br, double& bf,cell_pt cell) {
262
263         timing_lut_pt lut_pt_r;
264         timing_lut_pt lut_pt_f;
265
266         double af_s_o, ar_s_o, br_s_o, bf_s_o,num_outs;
267         af_s_o = ar_s_o = br_s_o = bf_s_o = num_outs = 0;
268
269
270         /************* outPut iteration *****************/
271         CellpinIter *it  = new CellpinIter(cell);
272
273         while(it->operator ()()) {
274
275             if(it->cellpin_ptr()->direction()== OUTPUT_CELLPIN_DIRECTION) {
276
277 //              //TimingArcIter *timing_arc_it = new TimingArcIter(it->cellpin_ptr(),it->cellpin_ptr());
278 //              timingset_iter_t timing_pt = it->cellpin_ptr()->timingset_ptr()->begin();
279 //
280 //              while (timing_pt.operator *() != NULL ) {
281
282                  for(auto& timing_pt : *(it->cellpin_ptr()->timingset_ptr())) {
283
284                     if(timing_pt->from_cellpin_name().compare(related_pin->name()) == 0) {
285
286                         double fall_cap = related_pin->capacitance();
287                         double rise_cap = fall_cap;
288
289                         cout << "fall_cap : " << fall_cap << endl;
290                         cout << "rise_cap : " << rise_cap << endl;
291
```

```
292                 vector<double> list_of_af;
293                 vector<double> list_of_ar;
294                 vector<double> list_of_bf;
295                 vector<double> list_of_br;
296
297
298                 /******* ITERATE ON TIMING SETS?    *******/
299                 lut_pt_r = it->cellpin_ptr()->timingset_ptr()->begin().operator *()->cell_rise_ptr();
300                 lut_pt_f = it->cellpin_ptr()->timingset_ptr()->begin().operator *()->cell_fall_ptr();
301
302                 double af_k = 0, ar_k = 0, bf_k = 0, br_k = 0;
303                 double cov00 = 0, cov01 = 0, cov11 = 0, sumsq = 0;
304
305
306                 cout << "***********FALL**********************" << endl;
307
308                if (lut_pt_f) {
309
310                    double *x_fall_val = new double[lut_pt_f->table()[1].size()];
311                    double *y_val = new double[lut_pt_f->table()[1].size()];
312
313                      for (unsigned int p = 0; p < lut_pt_f->table()[1].size() ; ++p)
314                      {
315                          x_fall_val[p] = lut_pt_f->indices2()[p]/fall_cap;
316                          cout << "x_fall_val[p]" << x_fall_val[p] << endl;
317                      }
318
319
320                      for (unsigned int l = 0; l < lut_pt_f->table()[1].size() ; ++l)
321                      {
322                          double sum = 0.0;
323
324
325
```

```
325
326                 for (unsigned int m = 0; m < lut_pt_f->table().size() ; ++m) //gia grammes
327                 {
328                     sum +=  lut_pt_f->table()[m][l];
329                 }
330
331                 y_val[l] = sum /lut_pt_f->table().size();
332                 cout << "y_val[l]" << y_val[l] << endl;
333                 }
334
335
336
337                 gsl_fit_linear( x_fall_val, 1, y_val, 1, lut_pt_f->table()[1].size(), &af_k, &bf_k, &cov00, &cov01, &cov11, &sumsq );
338
339                 cout << "af_k : " << af_k <<endl;
340                 cout << "bf_k :" << bf_k << endl;
341                 list_of_af.push_back( af_k );
342                 list_of_bf.push_back( bf_k );
343
344
345                 delete[] x_fall_val;
346                 delete[] y_val;
347
348               }
349
350             cout << "**************RISE*********************" << endl;
351
352               if (lut_pt_r) {
353
354                   double *x_rise_val = new double[lut_pt_r->table()[1].size()];
355                   double *y_val = new double[lut_pt_r->table()[1].size()];
356
```

```
357
358                     for (unsigned int p = 0; p < lut_pt_r->table()[1].size() ; ++p)
359                         {
360                             x_rise_val[p] = lut_pt_r->indices2()[p]/rise_cap;
361                             cout << "x_rise_val[p]" << x_rise_val[p] << endl;
362                         }
363
364
365                     for (unsigned int l = 0; l < lut_pt_r->table()[1].size() ; ++l)
366                     {
367                         double sum = 0.0;
368
369                         for (unsigned int m = 0; m < lut_pt_r->table().size() ; ++m) //gia grammes
370                         {
371                             sum +=  lut_pt_r->table()[m][l];
372                         }
373
374                         y_val[l] = sum /lut_pt_r->table().size();
375                         cout << "y_val[l]" << y_val[l] << endl;
376                     }
377
378
379                    gsl_fit_linear( x_rise_val, 1, y_val, 1, lut_pt_r->table()[1].size(), &ar_k, &br_k, &cov00, &cov01, &cov11, &sumsq );
380
381                    cout << "ar_k : " << ar_k <<endl;
382                    cout << "br_k :" << br_k << endl;
383
384                     list_of_ar.push_back( ar_k );
385                     list_of_br.push_back( br_k );
386
387                     delete[] x_rise_val;
388                     delete[] y_val;
389
390
```

```
390
391                      }
392
393                      /* Mean value of parasitic delays b*/
394                     double sum = 0;
395                     for (unsigned int k = 0; k < list_of_ar.size(); ++k)
396                     {
397                         sum += list_of_ar[k];
398                     }
399                     ar_s_o += sum / list_of_ar.size();
400
401                     sum = 0;
402                     for (unsigned int k = 0; k < list_of_af.size(); ++k)
403                     {
404                         sum += list_of_af[k];
405                     }
406
407                     af_s_o += sum / list_of_af.size();
408
409                     /* Mean value of logical effort parameter */
410                     sum = 0;
411                     for (unsigned int k = 0; k < list_of_br.size(); ++k){
412
413                         sum += list_of_br[k];
414                     }
415                     br_s_o += sum / list_of_br.size();
416
417                     sum = 0;
418
419                     for (unsigned int k = 0; k < list_of_bf.size(); ++k)
420                     {
421                         sum += list_of_bf[k];
422                     }
423                     bf_s_o += sum / list_of_bf.size();
```

```
415                         br_s_o += sum / list_of_br.size();
416
417                         sum = 0;
418
419                         for (unsigned int k = 0; k < list_of_bf.size(); ++k)
420                         {
421                             sum += list_of_bf[k];
422                         }
423                         bf_s_o += sum / list_of_bf.size();
424                         num_outs = num_outs + 1 ;
425
426                         cout << "af_s_o   :" << af_s_o << endl;
427                         cout << "ar_s_o   :" << ar_s_o << endl;
428                         cout << "br_s_o   :" << br_s_o << endl;
429                         cout << "bf_s_o   :" << bf_s_o << endl;
430
431
432                     } //if compare
433                 }   //timing arc
434             }       // if output
435
436         it->operator ++();
437
438     }   //pin iter
439
440          af = af_s_o/num_outs ;
441          ar = ar_s_o/num_outs ;
442          br = br_s_o/num_outs ;
443          bf = bf_s_o/num_outs ;
444 }
445
```

4.1.7 Logical Effort values extraction

For the purpose of this function we need to iterate every cell and call the *LExtraction :*

```
348         cell_it->begin();
349         while(cell_it->operator ()()) {
350
351             cell_pt cell_p = cell_it->cell_ptr();
352
353             double a_inv_f,a_inv_r,b_inv_f,b_inv_r;
354              a_inv_f = res->inv.a_inv_f;
355              a_inv_r = res->inv.a_inv_r;
356              b_inv_f = res->inv.b_inv_f;
357              b_inv_r = res->inv.b_inv_r;
358
359             cout << "LExtraction started" << endl;
360             resizer::LExtraction(cell_p,a_inv_f, a_inv_r,b_inv_f,b_inv_r);
361             cout << "LExtraction finished" << endl;
362
363             cell_it->operator ++();
364
365         }
```

```
449    void resizer::LExtraction(cell_pt cell_p,double a_inv_f,double a_inv_r,double b_inv_f,double b_inv_r) {
450
451        CellpinIter *it  = new CellpinIter(cell_p);
452
453        while(it->operator ()()) {
454
455            if (it->cellpin_ptr()->direction() == INPUT_CELLPIN_DIRECTION ) {
456
457
458                double af, ar, br, bf;
459
460                NLDM_to_LDM_conv(it->cellpin_ptr(), af, ar, br, bf,cell_p);
461                cout << "LExtraction for: " << cell_p->name() << endl;
462
463                it->cellpin_ptr()->set_Gfall(bf/b_inv_f);
464                it->cellpin_ptr()->set_Grise( br/b_inv_r );
465
466                it->cellpin_ptr()->set_Pfall( af/a_inv_f);
467                it->cellpin_ptr()->set_Prise( ar/a_inv_r );
468
469        }
470
471    it->operator ++();
472
473    }
474 }
```

4.2 Conclusion

We have checked and compared our results and values from OpenTimer with the ones that resulting from the CCSOpt , a continuous gate-level resizing tool that produce valid and credible values for parasitc delay and logical effort.

For example both tools produce the following values for the input pins of the gate NOR4_Y20 :

G_fall : 21.2632	(*logical effort*)
G_rise: 21.2632	
P_fall: 0.997727	(*parasitic delay*)
P_rise: 0.997727	

That comparison is verified for all the cells of our .lib file, so we end up that we have settled all the necessary tools and parameters in order to implement the resizing method for the critical paths with positive slacks.

Bibliography

[1] Kirkpatrick, TI & Clark, NR (1966). "PERT as an aid to logic design". IBM Journal of Research and Development.

[2] McWilliams, T.M. (1980). "Verification of timing constraints on large digital systems" (PDF). *Design Automation, 1980. 17th Conference on*. IEEE.

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

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

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

[6]. "Logical Effort:designing for speed on the back of an envelope," in *IEEE Advanced Research in VLSI*, 1991.

[7]. 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.

[8] 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*.

[9] Tsung-Wei Huang and Martin D. F. Wong “OpenTimer: An Open-Source High-Performance Timing Analysis Tool**”**

[10] Tsung-Wei Huang and Martin D. F. Wong “Special Session Paper: Incremental Timing and CPPR Analysis”, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA