UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Static Timing Analysis with Clock Domain Crossings Support

Integrated Master's Thesis

Giorgos Stanimeropoulos

Supervisor: Christos Sotiriou

Volos, June 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Στατική Χρονική Ανάλυση με Υποστήριξη Διασταυρώσεων Πεδίων Ρολογιών

Διπλωματική Εργασία

Γιώργος Στανιμερόπουλος

Επιβλέπων: Χρήστος Σωτηρίου

Acknowledgements

I would like to thank Prof. Christos Sotiriou for all the help he provided since my first years at the university, as well as Nikolaos Sketopoulos for his support during my first steps in my academic career. I would also like to express my sincere gratitude to Stavros Simoglou for his constant support, help and guidance he provided, both as a mentor and a friend. Finally, I'd like to thank all my friends and family, nothing would be possible without them.

Success is the product of a great team. I could not ask for anything better.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Giorgos Stanimeropoulos

Integrated Master's Thesis

Static Timing Analysis with Clock Domain Crossings Support

Giorgos Stanimeropoulos

Abstract

Modern Integrated Circuits (ICs) have advanced to the point where they consist of separate connected blocks, each one with a unique functionality to them, forming a System on Chip (SoC). The blocks of an SoC have their own specifications in terms of performance, power and area (PPA), thus they can operate under different clock domains with different frequencies. Depending on the architecture, paths starting from different blocks can converge into a point, otherwise known as Clock Domain Crossing (CDC) points. Such points may produce inaccurate results during Static Timing Analysis (STA), a methodology used in the semiconductor industry to determine whether the SoC is able to operate under the specified clock frequencies. Inaccurate STA can lead to devastating results, effectively destroying the SoC.

This thesis is focused on expanding the single-clock STA methodology to support analysis between connected design blocks that operate under different frequencies. An already established STA engine is used for the conventional STA methodology and is expanded in order to support CDCs and accurately determine whether the specified clock frequencies cause timing and data violations. The updated STA engine is then compared to the standard industry tool, showcasing a high correlation between the two.

Διπλωματική Εργασία

Στατική Χρονική Ανάλυση με Υποστήριξη Διασταυρώσεων Πεδίων Ρολογιών

Γιώργος Στανιμερόπουλος

Περίληψη

Τα μοντέρνα ολοκληρωμένα κυκλώματα έχουν εξελιχθεί στο σημείο που πλέον αποτελούνται από ξεχωριστά συνδεδεμένα μικρότερα κυκλώματα, το καθένα με τη δική του ξεχωριστή λειτουργία, σχηματίζοντας έτσι ένα Σύστημα πάνω σε Ολοκληρωμένο Κύκλωμα (Σ.Ο.Κ). Τα μικρότερα κυκλώματα ενός ΣΟΚ έχουν ξεχωριστές προδιαγραφές όσον αφορά την απόδοση, κατανάλωση και εμβαδόν (Α.Κ.Ε) και ως εκ τούτου μπορούν να λειτουργήσουν έχοντας το καθένα ξεχωριστό πεδίο ρολογιού με ξεχωριστή συχνότητα το καθένα. Αναλόγως την αρχιτεκτονική κάθε κυκλώματος, μονοπάτια που ξεκινούν από διαφορετικά υποκυκλώματα μπορούν να συγκλίνουν σε ένα σημείο, γνωστό ως σημείο Διασταύρωσης Πεδίων Ρολογιών (Δ.Π.Ρ). Τέτοια σημεία μπορούν να παράξουν ανακριβή αποτελέσματα κατά τη διάρκεια της Στατικής Χρονικής Ανάλυσης (Σ.Χ.Α), μια μεθοδολογίας που χρησιμοποιείται στη βιομηχανία ημιαγωγών με σκοπό να ελεγχθεί εάν οι συχνότητες των ρολογιών υπό τις οποίες το Ολοκληρωμένο Κύκλωμα ορίστηκε να λειτουργήσει είναι εφικτές. Ανακριβής Στατική Χρονική Ανάλυση μπορεί να οδηγήσει σε καταστροφικά αποτελέσματα, διαλύοντας ουσιαστικά το Ολοκληρωμένο Κύκλωμα.

Η παρούσα διπλωματική εργασία επικεντρώνεται στο να επεκτείνει την απλή μεθοδολογία της Σ.Χ.Α ώστε να μπορεί να υποστηρίξει ανάλυση μεταξύ κυκλωμάτων με διαφορετική συχνότητα λειτουργίας. Ένα προϋπάρχον εργαλείο Σ.Χ.Α χρησιμοποιείται για την απλή μεθοδολογία ανάλυσης και επεκτείνεται ώστε να μπορεί να υποστηρίξει Δ.Π.Ρ και να βρει με ακρίβεια εάν είναι δυνατό το Σ.Ο.Κ να λειτουργήσει υπό τις επιθυμητές συχνότητες. Το ανανεωμένο εργαλείο Σ.Χ.Α συγκρίνεται με τα βιομηχανικά στάνταρ, επιδεικνύοντας μεγάλη ακρίβεια μετρήσεων μεταξύ των δύο.

Table of contents

List of figures

List of tables

Listings

Abbreviations

ΣΧΑ	Στατική Χρονική Ανάλυση
ΣΟΚ	Σύστημα πάνω σε Ολοκληρωμένο Κύκλωμα
ΑΚΕ	Απόδοση Κατανάλωση Εμβαδόν
ΔΠΡ	Διασταύρωση Πεδίων Ρολογιών

IC	Integrated Circuit
STA	Static Timing Analysis
SoC	System on Chip
PPA	Performance Power Area
CDC	Clock Domain Crossings
VLSI	Very Large Scale Integration
IP	Intelectual Property

Chapter 1

Introduction

As Very Large Scale Integration (VLSI) systems get more and more dense, hardware designers need to come up with new methodologies to manage their inherent complexity. One such methodology are System-on-Chips (SoCs), circuits that emerge by combining in-house or third-patry pre-designed Intelectual Property (IP) blocks. SoCs are widely used in mobile devices, embedded systems and other cases where size, power consumption and manufacturing cost are of great importance. A typical SoC is comprised of different IC blocks, each one with a completely different and unique functionality, as shown in figure 1.1.

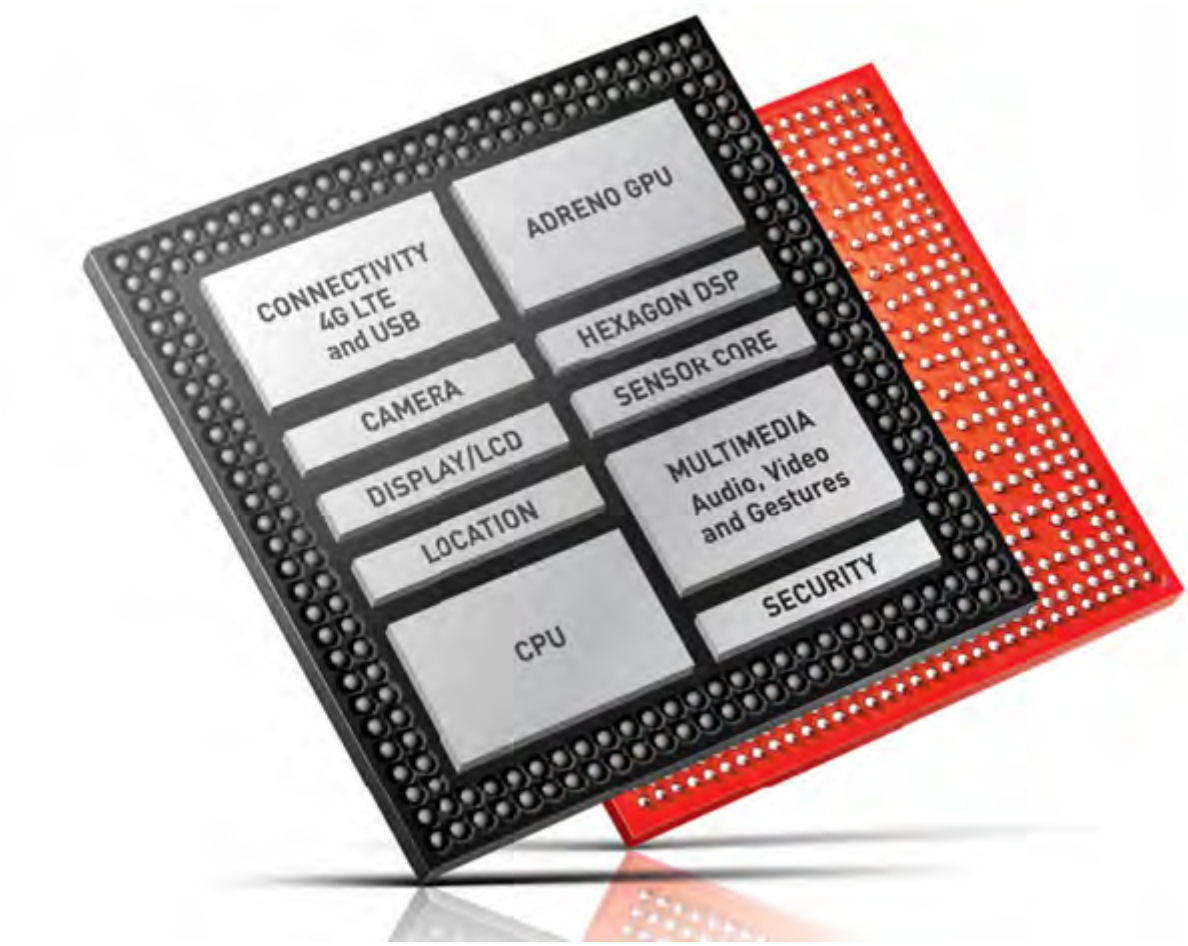

Figure 1.1: System on Chip example. Each block has a unique function

Year	Technology	Chip Complexity	ASIC Frequency
1997	250 nm	50M Tr.	100MHz
1999	180 nm	150M Tr.	200MHz
2002	130 nm	250M Tr.	400MHz
2004	90 nm	500M Tr.	600MHz

Figure 1.2: SoC scaling over the years

A simpler description of an SoC can be shown in figure 1.3. Here, each block may have different requirements as far as Performance, Power and Area (PPA) are concerned. Our focus is on Performance i.e how fast a block's clock can operate without compromising the Quality of Results (QoR). In this example, the clock frequencies for each block are shown in table 1.1. Each block belongs to its own *clock domain* and since all blocks connect to each other, there may be certain convergence points known as *Clock Domain Crossing points*.

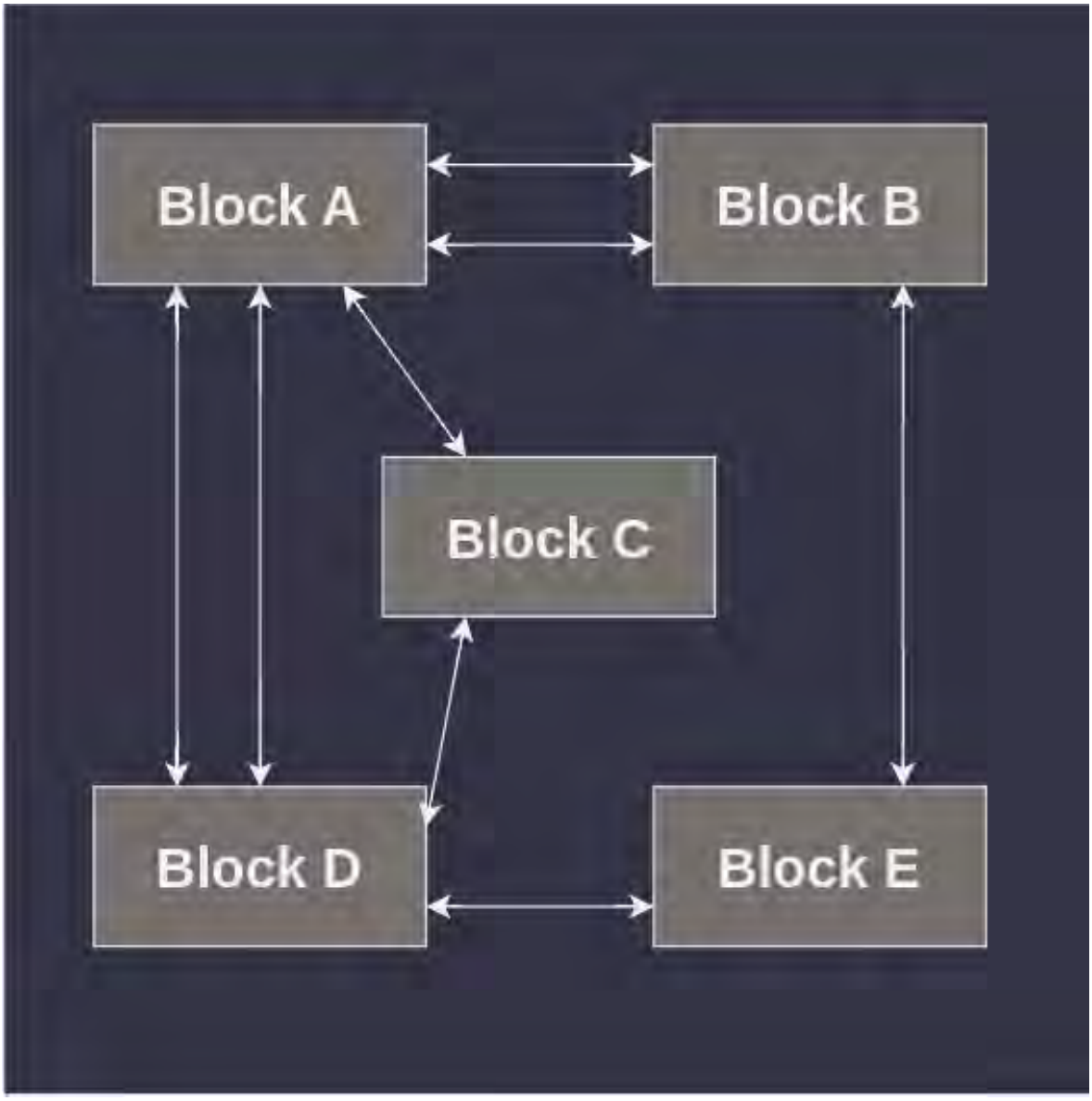

Figure 1.3: Simple SoC description. Blocks with different characteristcs are connected to each other

Block Name	Block Type	Block Hz
A	CPU	3,300
B	GPU	4,000
C	Modem	1,000
D	Video enc.	5,000
E	Compressor	2,500

Table 1.1: Block clock frequencies example

In order to ensure that the SoC can operate under the required clock frequencies, the industry uses a methodology known as Static Timing Analysis (STA). This methodology considers the SoC as a graph of nodes connected to each other via edges and calculates the delay of each node to determine if data will be transmitted reliably from one node to another under the given clock frequency the user specifies. However, STA is not able to produce accurate timing results when Clock Domain Crossings are introduced to the SoC, since it assumes both launch and capture clocks will fire at the same time. Such inaccurate results can have devestating concequences which can vary from ineffective circuits to total breakdowns.

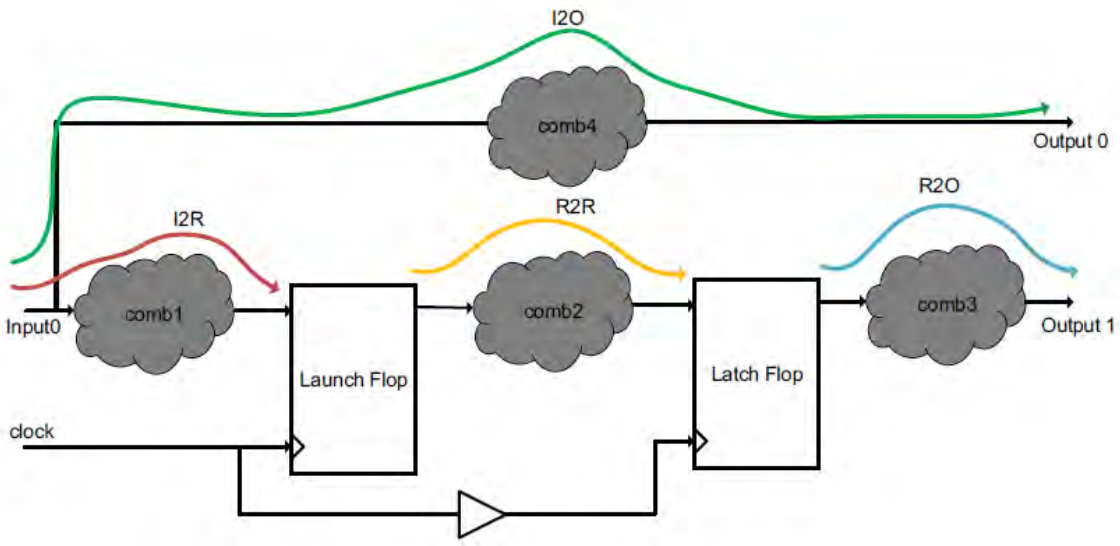

Figure 1.4: STA computes the delay of the circuit and determines whether it can operate under the specified clock frequency

1.1 Thesis Goal

The central theme of this thesis revolves around an in-depth analysis and comprehension of the established Static Timing Analysis (STA) methodology. Furthermore, the study aims to unravel the intricacies surrounding the Clock Domain Crossing (CDC) corner case, showing the limitations encountered by the single-clock STA approach when addressing such scenarios. The steps that enable STA to properly detect and handle CDCs are explained afterwards. Finally, this methodology is used in the single-clock STA engine explained in [1] and the effectiveness of this refined approach is compared to cutting-edge industrial timing tools with high accuracy.

1.1.1 Contribution

The contribution of this thesis can be summarized as follows:

1. Single-clock Static Timing Analysis was studied and tested under various conditions
2. An algorithm for computing nearest edges between two clock waveforms was developed
3. Single-clock STA algorithm was expanded to support Clock Domain Crossings analysis
4. The expanded STA engine was compared with the industrial tool showcasing high accuracy

1.2 Chapter Organization

The single-clock STA methodology is explained in Chapter 2. Chapter 3 focuses on the CDC corner cases where STA fails to produce quality results and explains the steps taken to expand the methodology. Finally, Chapter 4 shows the experimental flow and the expanded STA engine results compared to the industry standard tool.

Chapter 2

Timing Analysis

A digital design (or digital circuit) is an electrical circuit that is able to perform one or more boolean functions. A typical digital design consists of:

- Input/Output ports, to get electrical signal from the environment and return the result of its operations back to it
- Combinational gates: Circuit components that can perform a boolean function (e.g AND gate, OR gate, inverter etc)
- Sequential gates: Circuit components that are able to hold a specific state (e.g latches, flip-flops etc)
- Wires: Metal traces used to connect one gate to another

Digital designs can be *synchronous* or *asynchronous*. Synchronous designs rely on an outer *clock signal* to transmit their data from one flip-flop to another whereas asynchronous circuits rely on a combination of ”handshakes” between certain gates to ensure data have reached a part of the circuit.

When designing a digital circuit, the gates and wires essentially create a graph of nodes and vertices, with nodes being boolean gates of a certain technology and vertices being the traces used to connect them. The graph is usually both directed and acyclic, although cyclic graphs can also be met in digital designs [2]. Each one of these nodes and vertices introduce a certain delay to the path from one startpoint (flip-flop output or circuit input) to an endpoint (flip-flop input or circuit output). In order for the digital design to function reliably, engineers need to ensure that data from one startpoint will reach the endpoint within a proper timing

window so that the signal is captured and re-transimitted without any issue. This kind of analysis is known as *timing analysis* and can be broken down into three major categories:

1. Electrical simulations

2. Static Timing Analysis (STA)

3. Dynamic Timing Analysis (DTA)

2.1 Electrical Simulations

Electrical simulations are a method used to represent the behaviour of the circuit using mathematical models. Every boolean gate is built using transistors with different characteristics. This kind of simulation will take an electric signal as the input of a gate and use various analytical methods to calculate the signal that will be produced in the output. Since every gate is connected to another through wires which can also be mathematically modeled, the output signal can be propagated to the next gate, keeping the simulation going until it reaches a circuit endpoint. Since an electric signal is essentially pairs of (time – voltage value), we can calculate the delay from a startpoint to an endpoint, whether the signal reached its destination reliably etc. Electrical simulations are the most accurate method of timing analysis, thus, they are often the golden model every algorithm is compared to. However, they are *extremely* time consuming to be performed. That is why they are used in small portions of the circuit, simulating only a couple of dozen to a few hundred gates at a time. Finally, simulation is the most effective way of measuring *interconnect delay* [3]. As technology nodes advance, interconnect between two gates has an increasing impact on circuit performance. Although several models have been developed to measure interconnect delay fast and accurately, [4, 5, 6, 7], simulation remains the golden standard, since none of these approaches can match its accuracy.

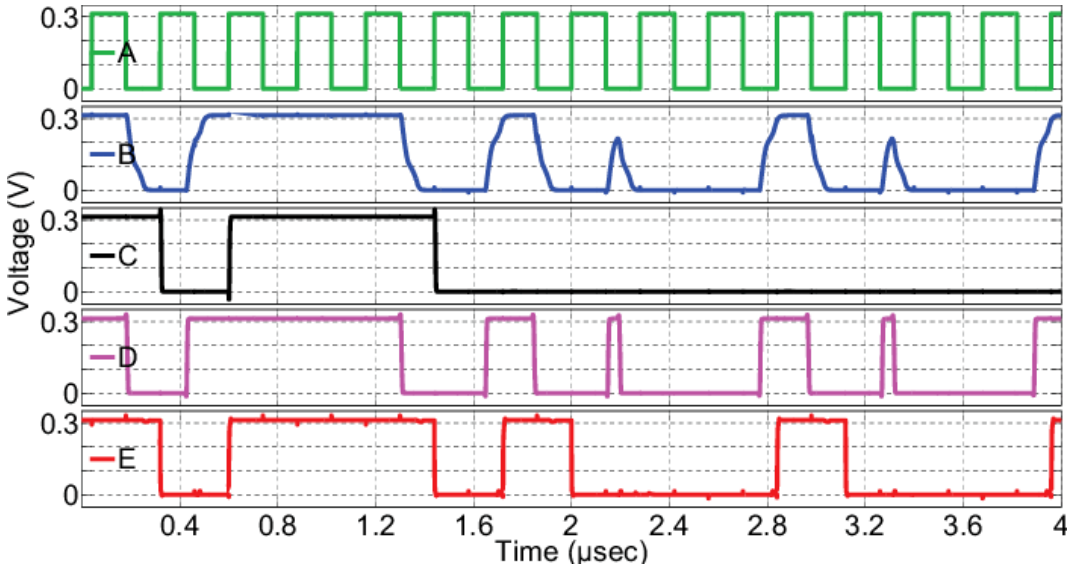

Figure 2.1: Electrical Simulation; Signals are modeled in the circuit input. Mathematical models are used to determine the behaviour of the signal in the circuit

2.2 Static Timing Analysis

Static Timing Analysis (STA) is a method that, contrary to Electrical Simulation, does not rely on any kind of simulation to perform timing analysis. Instead, it sets delay values to the circuit inputs and then traverses the graph, annotating the delay on each node until an endpoint is reached. There, the annotated delay is compared against the corresponding clock signal to verify whether a timing violation has occurred or not. This method is considered to be the fastest as, depending on the implementation, the complexity of traversing a DAG can be $O(V+E)$ where V is the number of vertices and E is the number of edges. However, Static Timing Analysis is typically the most pessimistic one as well, as its calculations are based on a worst-case analysis of the design which in most cases won't take place. This pessimism can be reduced using several methods such as false path specification, i.e paths that will never be sensitized due to circuit's functionality [8], multiple operating corner analysis [9], On-Chip-Variation (OCV) analysis [10] etc.

2.3 Dynamic Timing Analysis

Dynamic Timing Analysis (DTA) [11] is essentially a combination of STA and electrical simulations; Using specific input signals, it traverses the graph, sensitizing the nodes and

calculating the longest path in the process. One of its advantages over STA is that, since it uses specific input patterns, it tests the design in its actual functional context and therefore, it does not consider any false paths during the analysis. However, it is more inefficient as far as speed is concerned, since it uses a simulation-like analysis and its result are heavily depended on the input signals that are used for the analysis; Insufficient input signals will produce insufficient results.

This thesis will focus on STA and how this method can be expanded to support multiple clock domains and CDCs.

2.4 Timing Analysis Terminology

The STA algorithm takes the following input:

- Circuit's graph, usually represented as a Verilog netlist
- Timing library, describing timing and power information for each of the Boolean gates used in the verilog netlist

The timing library is technology depended and provides various timing and power information for the technology's gates. This thesis utilizes the following:

- Gate's pins' transition times (slew)
- Gate delay
- Wire models for computing interconnect delay

2.4.1 Gate pins transition times

Transition times (otherwise referred to as *slew*) describe the amount of time a gate pin needs to transition from 0 -> VDD (rise slew) or from VDD -> 0 (fall slew). More precisely, it is computed as

$$|t_{V=10\%*VDD} - t_{V=90\%*VDD}|$$

where $t_{V=10\%*VDD}$ is the time the gate reaches the 10% of its maximum voltage and $t_{V=90\%*VDD}$ is the time the gate reaches the 90% of its maximum voltage, although some libraries may use 20 and 80% of VDD as slew's thresholds.

Figure 2.2 shows the rise slew of an inverter cell.

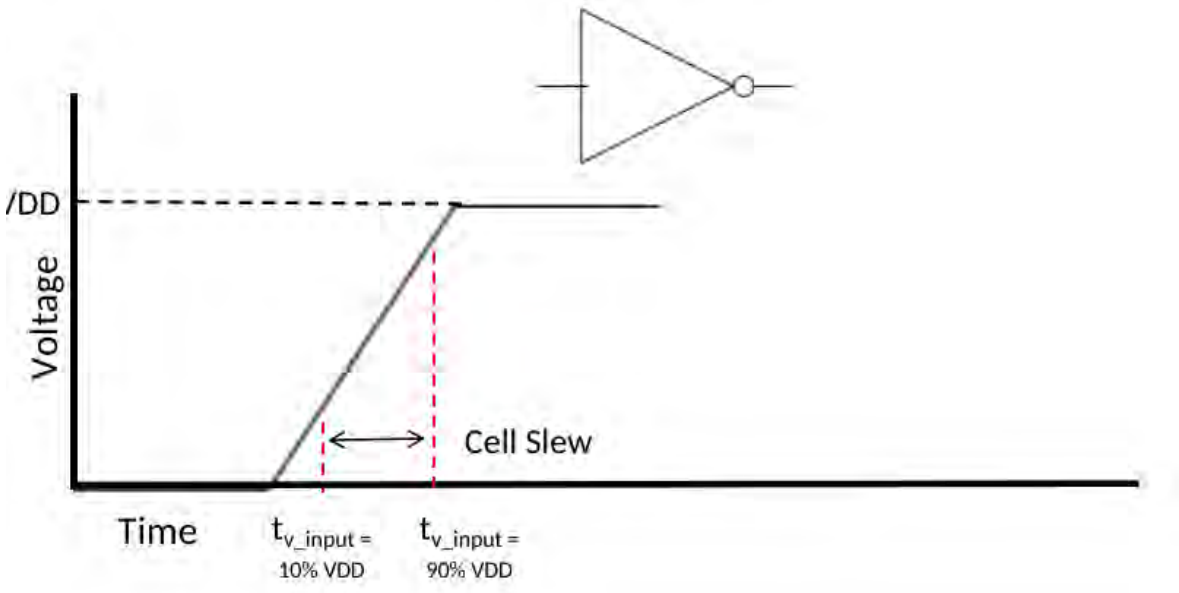

Figure 2.2: Inverter rise slew

2.4.2 Gate Delay

Gate delay refers to the internal delay a boolean gate has, depending on how it was physically constructed. It describes the gap between the times an input and output pin reach 50% of their transition. Figure 2.3 shows the delay of an inverter cell; As the input rises from 0 -> VDD, the output changes from VDD -> 0. The difference between the times the two pins each reach 50% of VDD represents the inverter's internal delay. As it will be explained later, delay on an output depends on the related input pin slew and the capacitance the output pin drives.

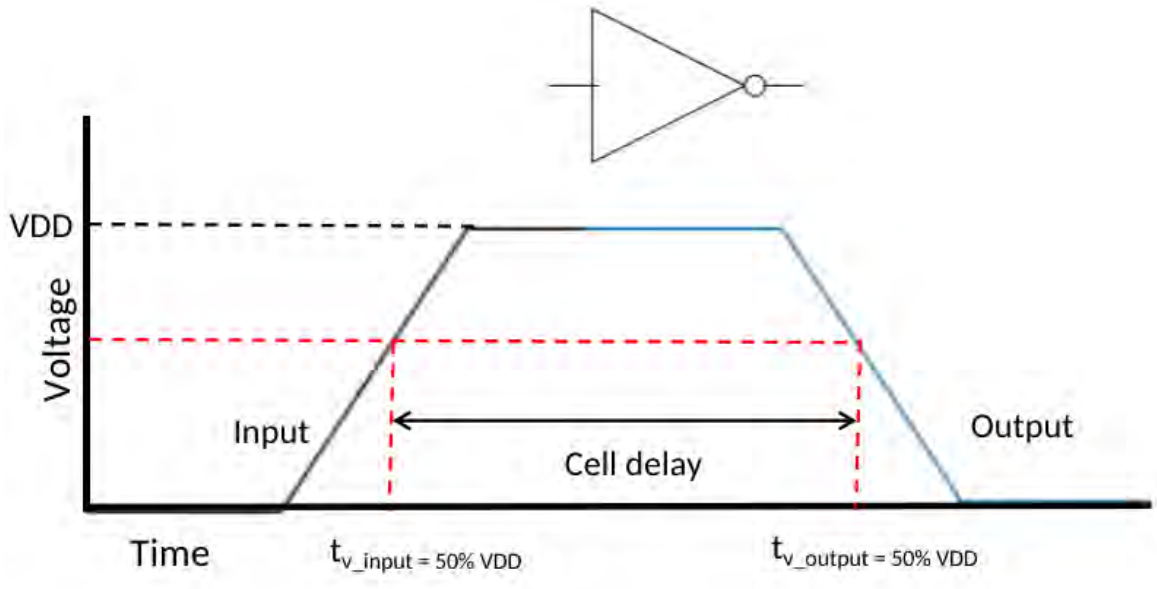

Figure 2.3: Gate delay of an inverter

2.4.3 Setup/Hold Slack

Setup time is defined on flip-flops and specifies the minimum amount of time the data input signal must be stable *before* the clock signal arrives to ensure data is captured correctly and to avoid metastability. Figures 2.4 and 2.5 show the difference in the flip-flop's output if the input signal changes before and after setup time. In the latter, the output may not be able to reach 0 or VDD, or transition at all, or breaking down entirely, compromising the circuit's functionality. Setup time is calculated between two different clock edges, one clock period apart.

Likewise, hold time defines the minimum time a data input signal must be stable *after* the clock signal arrives to ensure data is sent correctly. If a signal transitions before hold time, metastability issues are inevitable. Figures 2.6 and 2.7 show the different outputs if the input transitions before and after hold time. Hold time is calculated between two different sequential cells *in the same clock edge*. Thus, for this analysis, the shortest delay path is used.

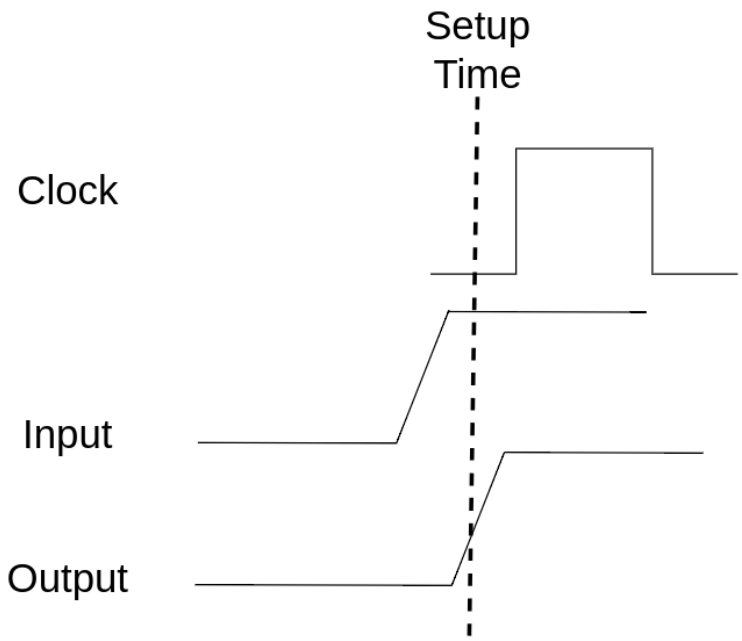

Figure 2.4: Signal transition before setup time

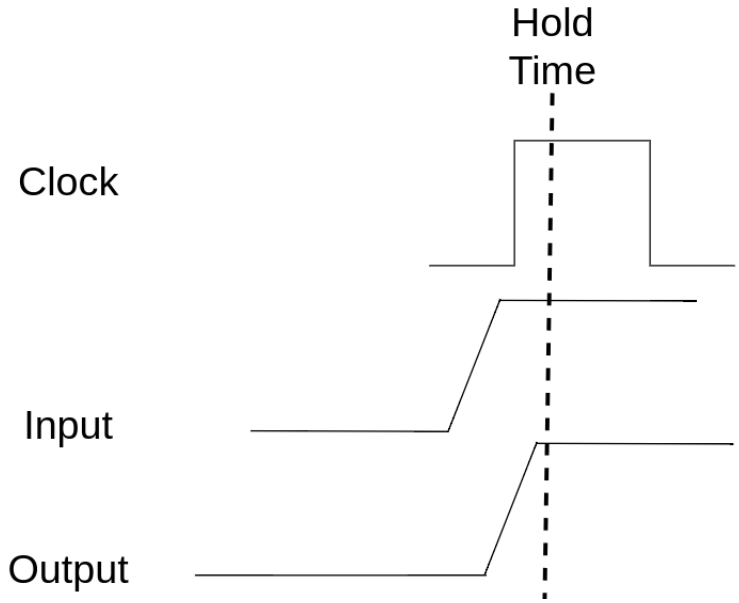

Figure 2.6: Signal transitions before hold time

Based on these two constraints, it can be easily concluded that there is a *latest* time a signal is allowed to transition *before* clock arrival time and an *earliest* time a signal can transition *after* clock arrival time without compromising circuit's functionality. These two times are referred to as **setup/hold required arrival times (RATs)**. Combined with the actual time a signal arrives on the flip-flop (AAT), setup and hold *slack* can be defined as follows:

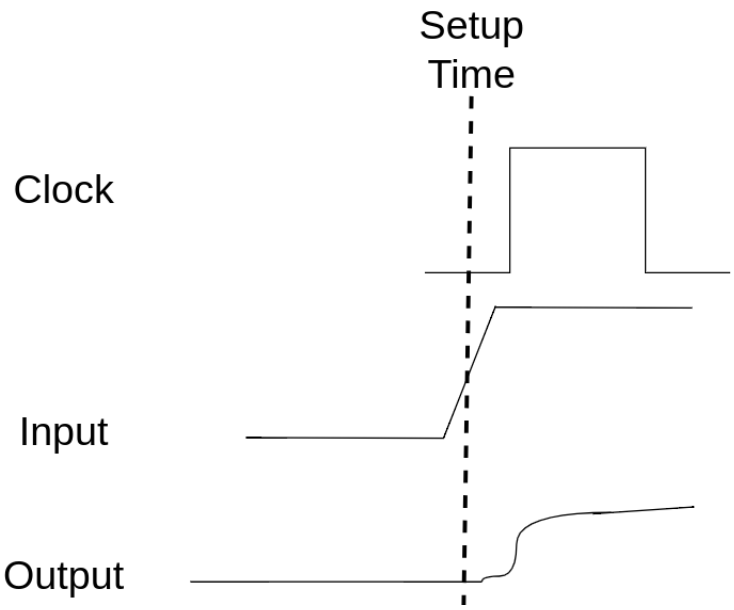

Figure 2.5: Signal transition after setup time

$$Setup\ slack = Setup\ RAT - AAT =>$$

$$Setup\ slack = Clock\ arrival\ time - setup\ time - AAT =>$$

$$Setup\ slack = Clock\ first\ edge + 1T + clock\ network\ delay - setup\ time - AAT$$

$$Hold\ slack = AAT - Hold\ RAT =>$$

$$Hold\ slack = AAT - (clock\ arrival\ time + hold\ time) =>$$

$$Hold\ slack = AAT - (Clock\ first\ edge + 1T + clock\ network\ delay + hold\ time)$$

Slack is essentially the timing window a signal can transition into without causing any metastability issues. Negative slack means there is no time for the signal to change state and is considered a timing violation.

2.4.4 Timing models

As mentioned in subsection 2.4.2, gate delay depends on two factors: a) input pin transition time and b) output pin capacitive load. This can be represented by a two-dimensional table, with one axis being input pin transition and the other being output pin capacitance. This

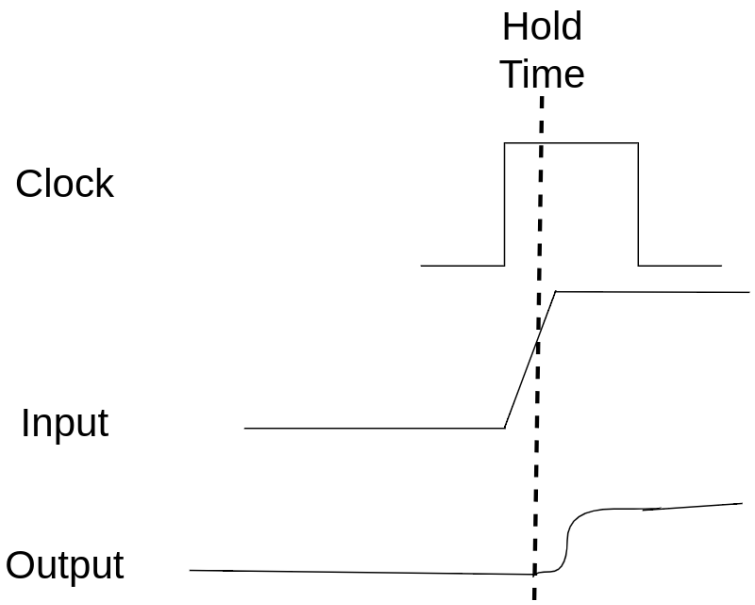

Figure 2.7: Signal transitions after hold time

model is known as *Non-Linear Delay Model (NLDM)*. Figure 2.8 shows an example of this model, where $index\_1$ is the input pin transition time, $index\_2$ is the output pin capacitance and $values$ is the calculated delay values for each input slew – output capacitance pair. These values have been calculated through a method called *characterization*. In case a specific gate has values on one or both of the factors that are not specified in the library but are within the boundaries indices 1 and 2 specify, the final delay is computed through a 2D linear interpolation. NLDM models have been widely used so far, however, they may produce inaccurate results since a) their accuracy depends on the table's size and measurement quality and b) a circuit gate may have values that are *outside* of the table's boundaries in which case *extrapolation* must be performed. Extrapolation, however, is prone to producing inaccurate results, since there is no reliable method to calculate values outside of the given array's boundaries.

```
related_pin        : "A1";
timing_sense       : positive_unate;

cell_fall(Timing_7_7) {
        index_1 ("0.00231025,0.0112628,0.0426883,0.102700,0.196195,0.327379,0.500000");
        index_2 ("0.365616,1.893040,3.786090,7.572170,15.144300,30.288700,60.577400");
        values ("0.0765980,0.0862464,0.0950826,0.109025,0.131636,0.170516,0.242432", \
                "0.0810751,0.0907333,0.0995695,0.113522,0.136131,0.175012,0.246932", \
                "0.0989261,0.108529,0.117371,0.131331,0.153954,0.192846,0.264768", \
                "0.135565,0.145269,0.154090,0.168072,0.190773,0.229716,0.301656", \
                "0.181379,0.192982,0.203116,0.218219,0.242152,0.281772,0.353798", \
                "0.229951,0.243802,0.255686,0.272965,0.299023,0.340559,0.413847", \
                "0.281255,0.297392,0.311290,0.331178,0.359969,0.403899,0.478602");
}
cell_rise(Timing_7_7) {
        index_1 ("0.00231025,0.0112628,0.0426883,0.102700,0.196195,0.327379,0.500000");
        index_2 ("0.365616,1.893040,3.786090,7.572170,15.144300,30.288700,60.577400");
        values ("0.0630239,0.0787480,0.0961212,0.129093,0.193632,0.321810,0.577399", \
                "0.0674459,0.0831711,0.100537,0.133506,0.198055,0.326241,0.581824", \
                "0.0840800,0.0996979,0.116960,0.149818,0.214346,0.342599,0.598289", \
                "0.111098,0.127055,0.144257,0.176866,0.241183,0.369419,0.625170", \
                "0.137221,0.154429,0.171997,0.204249,0.268543,0.396458,0.652147", \
                "0.158896,0.178441,0.197069,0.229757,0.293570,0.421467,0.676922", \
                "0.174683,0.196972,0.217621,0.251499,0.315047,0.442340,0.697757");
}
```

Figure 2.8: Two NLDM tables, for rise and fall delay with respect to A1 input pin

For newer technologies, where phenomena like coupling, electromigration etc have a bigger impact on circuit performance, other delay models such as the *Composite Current Source Model (CCSM)* and *Effictive Current Source Model (ECSM)* [12] are used, as they can hold more fine-grained information and represent electrical phenomena such as noise, crosstalk etc more accurately.

2.4.5 Timing arcs

Timing arcs describe the timing relations between a cell's input and output pin. If an arc between input A and output Z of a cell exists, there can be a timing path from A to Z. Arcs are also characterized by their *timing sense*, otherwise known as *unateness*. This describes the behaviour the output pin has with relation to the input. An arc is **positive unate** when the output follows the same transition as the input (input rise ==> output rise, input fall ==> output fall). An arc is **negative unate** when the output follows the opposite transition of the input (input rise ==> output fall, input fall ==> output rise). Finally, an arc is **non-unate** or **binate** when the output's transition depends on multiple pin, as it is in the case of a XOR gate. Figure 2.9 shows the timing arcs of a buffer cell.

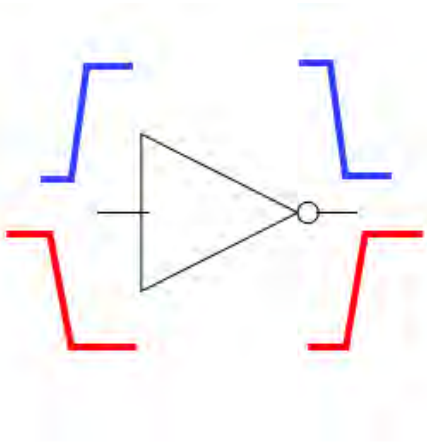

Figure 2.9: Inverter timing arcs

2.5 STA in Detail

The STA algorithm performs the following steps to determine the worst slack on the circuit:

1. Clock network annotation
2. Delay propagation/Calculation
3. Slack computation
4. RAT propagation/Calculation

2.5.1 Clock network annotation

In this step, STA annotates clock network's delay, to calculate the latency between a clock source and a clock sink, i.e a sequential cell's clock input pin; The user has defined that one or more of the design's inputs will act as the design's clock(s). Starting from these inputs, delay is computed and propagated throughout the clock network until a clock leaf (sequential cell clock inputs) is reached. To propagate delay from one cell to the next, topological sort or Breadth-First-Search [13] are used. These graph traversal algorithms ensure that a cell will be visited only if all of its predecessors have been visited. This step can be summarized in algorithms 2.1, 2.3, 2.2:

Listing 2.1: AnnotateCellInputDelay

```
Description: Calculate delay on input pin based on predecessor's delay
 Inputs: Cell input pin (inPin)
 Outputs: Rise and fall input pin delay (riseInPinDelay, fallInPinDelay)

 riseInPinDelay = 0
 fallInPinDelay = 0
 predPin = get_predecessor(inPin)
 riseInPinDelay = delay_at(predPin, rise) + wire_delay(predPin, inPin, rise)
 fallInPinDelay = delay_at(predPin, fall) + wire_delay(predPin, inPin, fall)

 return riseInPinDelay, fallInPinDelay
```

Listing 2.2: ClockNetworkDelayPropagation

```
Description: Traverse the clock network using a levelized graph traversal and compute
    delay for each clock network gate
Inputs: Clock input pins (CIs)
Outputs:

pinQueue = []
append CIs in pinQueue

while pinQueue is not empty do
   currentPin = pinSQueue.dequeue()

   foreach inPin in currentPin successors do
     riseDelay[inPin], fallDelay[inPin] = AnnotateCellInputDelay(inPin)
      if inPin is clock tree leaf then
         continue
      end if

      foreach outPin in inPin successors do
         predecessorNum[outPin]++
         if predecessorNum[outPin] == in_degree(outPin) then
            riseDelay[outPin], fallDelay[outPin] = AnnotateCellOutputDelay(outPin)
            pinQueue.append(outPin)
      end for
   end for
end while
```

Listing 2.3: AnnotateCellOutputDelay

```
Description: Calculate output pin delay based on gate's input pins delay
 Inputs: Cell output pin (outPin)
 Outputs: Rise and fall output pin delay (riseOutPinDelay, fallOutPinDelay)
 riseOutPinDelay, fallOutPinDelay = -inf

 foreach predPin in get_predecessors(outPin) do
     if cell_is_positive_unate() then
         predRiseDelay = delay_at(predPin, rise)
         currentDelay = predRiseDelay + arc_delay(predPin, outPin, rise)
         if currentDelay > riseOutPinDelay then
             riseOutPinDelay = currentDelay
         end if

         predFallDelay = delay_at(predPin, fall)
         currentDelay = predFallDelay + arc_delay(predPin, outPin, fall)
         if currentDelay > fallOutPinDelay then
             fallOutPinDelay = currentDelay
         end if
     else if cell_is_negative_unate() then
         predRiseDelay = delay_at(predPin, rise)
         currentDelay = predRiseDelay + arc_delay(predPin, outPin, fall)
         if currentDelay > riseOutPinDelay then
             riseOutPinDelay = currentDelay
         end if

         predFallDelay = delay_at(predPin, fall)
         currentDelay = predFallDelay + arc_delay(predPin, outPin, rise)
         if currentDelay > fallOutPinDelay then
             fallOutPinDelay = currentDelay
         end if
     else // non-unate cell //
         predRiseDelay = delay_at(predPin, rise)
         predFallDelay = delay_at(predPin, fall)
         predDelay = max(predRiseDelay, predFallDelay)

         currentDelay = predDelay + arc_delay(predPin, outPin, rise)
         if currentDelay > riseOutPinDelay then
             riseOutPinDelay = currentDelay
         end if

         currentDelay = predDelay + arc_delay(predPin, outPin, fall)
         if currentDelay > fallOutPinDelay then
             fallOutPinDelay = currentDelay
         end if
     end if
 end for
 return riseOutDelay, fallOutDelay
```

2.5.2 Delay Propagation/Computation

The second STA step calculates and propagates the delay in the rest of the graph, from circuit input ports and sequential cell outputs (startpoints) to circuit outputs and sequential cell inputs (endpoints). The algorithms are the same as the ones used for the clock network annotation. The reason these steps are separate is that the respective graph traversals stop at different pins and because clock network changes more rarely; thus, re-calculating its delay can be avoided in the majority of STA's runs.

The user has specified some delay values at circuit's inputs and outputs with respect to one of the specified clocks. These delay values are then propagated using Topological Sort or Breadth-First Search. Each time a cell's output pin is reached, the slowest (for setup analysis) or fastest (for hold analysis) arc is selected and the final delay value is computed. This delay is then propgated to the next level of input pins, where the wire delay is also accumulated. Delay propagation can be summed up in the following algorithm:

Listing 2.4: CircuitDelayPropagation

```
1  Description: Traverse the graph using levelized graph traversal and compute delay for
       all circuit gates apart from clock network
2  Inputs: Circuit input pins (CircuitInputs)
3  Outputs:
4
5  pinQueue = []
6  append CircuitInputs in pinQueue
7
8  while pinQueue is not empty do
9     currentPin = pinQueue.dequeue()
10
11    foreach inPin in currentPin successors do
12      riseDelay[inPin], fallDelay[inPin] = AnnotateCellInputDelay(inPin)
13       if inPin is FF data pin or Circuit Output then
14          continue
15       end if
16       foreach outPin in inPin successors do
17          predecessorNum[outPin]++
18          if predecessorNum[outPin] == in_degree(outPin) then
19             riseDelay[outPin], fallDelay[outPin] = AnnotateCellOutputDelay(outPin)
20             pinQueue.append(outPin)
21       end for
22    end for
23 end while
```

2.5.3 Slack Computation

Once delay has been propagated to the entirety of the graph, slack computation takes place; As mentioned in subsection 2.4.3, slack is essentially the timing margin a signal is allowed to transition within. Negative slack means there is no such margin, thus timing is violated at the pin the method is examined. The pin with the lowest slack is the critical pin and the path leading up to it is also known as the *critical path*, since it is the one with the highest impact on the circuit's timing.

Using the formula for slack computation mentioned in subsection 2.4.3, the following algorithm can be used to determine worst slack and critical endpoint:

Listing 2.5: computeSlack

```
1  Description: Compute slack for all graph endpoints
2  Input: List of endpoint pins (endpoints[])
3  Output: Worst negative slack and critical endpoint (WNS, CE)
4
5  WNS = inf
6  CE = None
7
8  foreach endpoint in endpoints do
9     endpointClk = get_related_clock(endpoint)
10    CNDelay = get_clock_network_delay(enpointClk, endpoint)
11
12    currentRAT = first_active_edge(endpointClk) + get_clock_period(endpointClk) + CNDelay
          - get_setup_time(endpoint)
13    currentAAT = delay_at(endpoint)
14
15    currentSlack = currentRAT - currentAAT
16    if WNS > currentSlack then
17       WNS = currentSlack
18       CE = endpoint
19    end if
20 end for
21
22 return WNS, CE
```

2.5.4 RAT calculation and propagation

In the final STA step, RAT and slack must be calculated for every pin of the graph. RAT propagation is completely symmetrical to delay propagation; A levelized graph traversal (Topological Sort or BFS) takes place starting from graph endpoints (circuit output or sequential cell inputs) all the way back to the startpoints (circuit inputs or sequential cell outputs). At each pin, setup/hold RAT and setup/hold slack are computed, to ensure no timing violations occur in the rest of the design. Figure 2.10 shows the direction of the RAT traversal, with the green dots representing the endpoints and the red dots representing the startpoints.

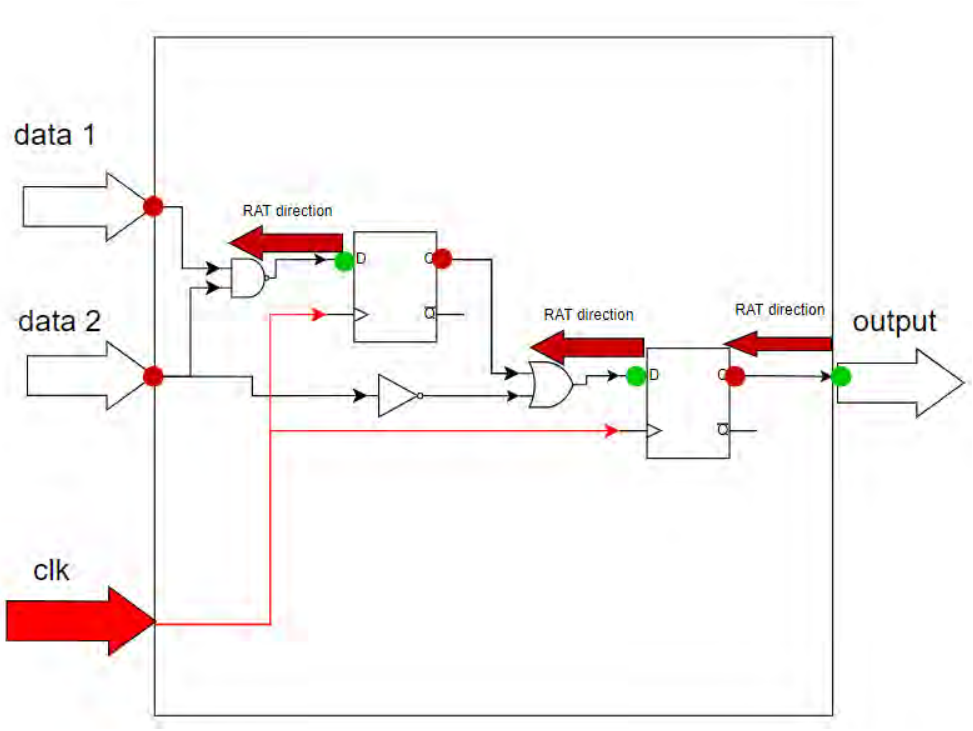

Figure 2.10: RAT propagation, from endpoints to startpoints

This step can be summarized in algorithms 2.6, 2.7 and 2.8

Listing 2.6: calculateRATatInputs

```
1  Description: Update the RAT at an input pin based on the lowest RAT of its successors
2  Input: Input pin (inPin)
3  Output: RAT and slack for input pin (inRAT, inSlack)
4
5  pinSuccessors = get_pin_successors(inPin) // cell's output pins
6   inRAT, inSlack = inf
7
8   foreach successor in pinSuccessors do
9       foreach arc in get_arcs(inPin, successor) do
10          currentDelay = get_arc_delay(arc) // return delay depending on unateness
11
12          currentRAT = RAT_at(successor) - currentDelay
13          if inRAT > currentRAT then
14              inRAT = currentRAT
15          end if
16
17          currentSlack = currentRAT - delay_at(inPin)
18          if inSlack > currentSlack then
19              inSlack = currentSlack
20          end if
21      end for
22   end for
23   return inRAT, inSlack
```

Listing 2.7: calculateRATatOutputs

```
1  Description: Update the Required Arrival Time at an output pin based on the lowest RAT
       of its successors
2  Input: Output pin (outPin)
3  Output: RAT and slack for output pin (outRAT, outSlack)
4
5  pinSuccessors = get_pin_successors(outPin) // receiver cells in the net
6  outRAT, outSlack = inf
7
8  foreach successor in pinSuccessors do
9     connectionDelay = wire_delay(outPin, successor)
10
11    currentRAT = RAT_at(successor) - connectionDelay
12    if outRAT > currentRAT then
13       outRAT = currentRAT
14    end if
15
16    currentSlack = currentRAT - delay_at(outPin)
17    if outSlack > currentSlack then
18       outSlack = currentSlack
19    end if
20 end for
21
22 return outRAT, outSlack
```

Listing 2.8: propagateRAT

```
Description: Update RAT on the graph, starting from endpoints and moving backwards
    towards the startpoints using levelized traversal
Input: Graph endpoints (GEs)
Outputs:

initialise visitedSuccessors for all pins to 0
pinQueue = []
append GEs to pinQueue

while pinQueue is not empty do
   currentPin = pinQueue.dequeue()

   driverPin = get_predecessor(currentPin)
   visitedSuccessors[driverPin]++

   // all driverPin successors have been visited
   if visitedSuccessors[driverPin] == out_degree(driverPin) then
      RATValues[driverPin], slackValues[driverPin] = calculateRATatOutputs(driverPin)

      if driverPin is circuit input or sequential cell output then
         break
      end if

      foreach inPin in get_predecessors(driverPin) do
         visitedSuccessors[inPin]++
         if visitedSuccessors[inPin] == out_degree(inPin) then
            RATValues[inPin], slackValues[inPin] = calculateRATatInputs(inPin)
            pinQueue.append(inPin)
         end if
      end for
   end if
end for
```

Chapter 3

Clock Domain Crossings

A *clock domain* refers to a subgroup of gates that are constrained by one specific clock input. A circuit consisting of more than clock domains may have certain nodes where paths starting from different clocks converge to. Such nodes are known as Clock Domain Crossing (CDC) points.

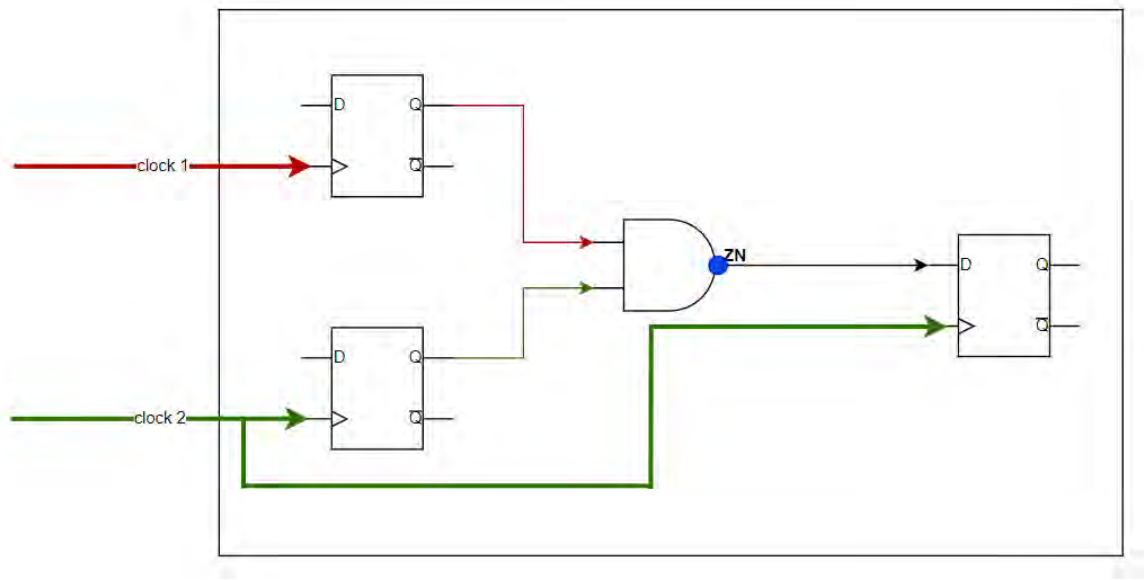

Figure 3.1: Clock Domain Crossing example. Blue dot represents the node where the paths from the two clocks converge

3.1 Importance of Clock Domain Crossings

As described in [14], CDCs are able to cause a number of issues in the design, such as *metastability, data loss* and *data incoherence*.

3.1.1 Metastability

Metastabilty was briefly mentioned in section 2.4.3; it is the state when a signal is unable to fully transition to a new state, either retaining its old value or reaching a portion of the new one. This can be caused from a number of reasons such as setup/hold violations, where data reaching a sequential element transition very close to the respective clock edge and particle strikes that excite a circuit path when no signal goes through it [15]. Metastability can cause serious issues to the circuit's functionality, as the signal can theoretically remain in this unstable state forever [16].

3.1.2 Data Loss

Data loss is another important issue caused by clock domain crossings. When data launching from one clock domain are captured from a *slower* clock domain, it is possible that data launched at $t = t_n$ are lost because they are overwritten by the data launched at $t = t_n + T1$. In the example shown in figure 3.2, the data from the fast clock (green) are lost once since the slow clock (red) will capture the ones sent in the next period.

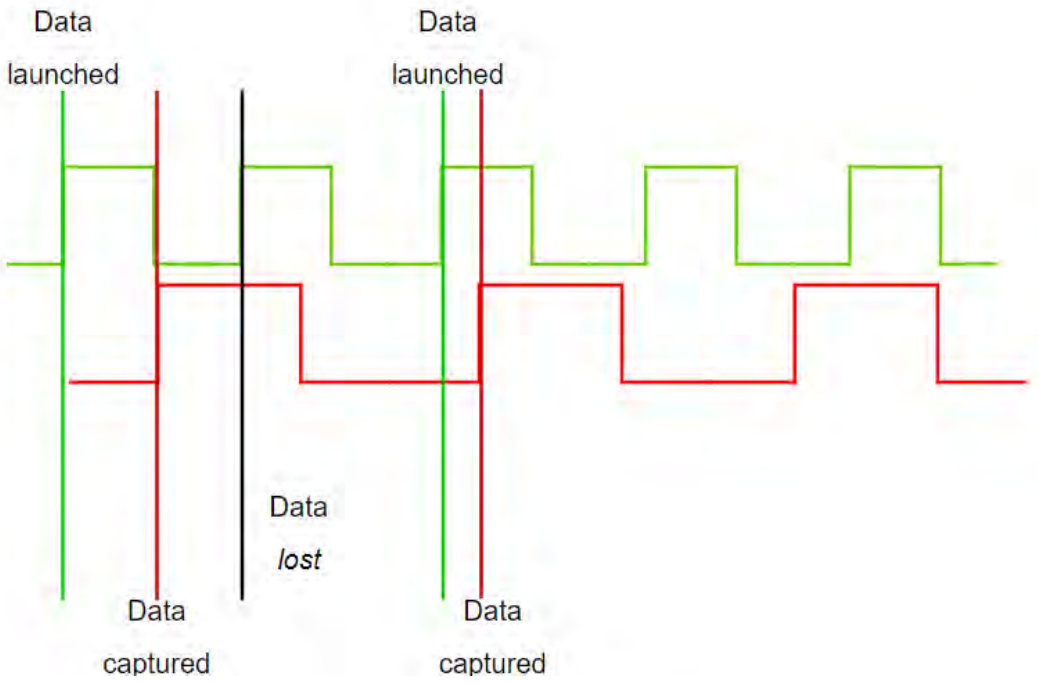

Figure 3.2: Data can be lost when launched from a fast clock to a slower one

3.1.3 Solutions to the Clock Domain Crossings issue

The majority of the academic research, such as [17, 14, 18], focuses on the structural/-functional analysis of CDCs and provides methods to detect and resolve them in the Register Transfer Level (RTL), before moving on to circuit's physical design. The work conducted on [19] proposes a method to independently detect and resolve CDCs during physical design which can also be integrated with other timing tools, even the industrial ones. However, resolving CDCs can be an expensive process, as it includes implementing and adding extra synchronization circuitry which can have a significant impact on the total area and power consumption. As a result, at times, it might be more preferable to work around CDCs, ensuring that they will not compromise circuit's timing. Thus, timing algorithms must be able to properly detect and compute the worst slack on circuits with CDCs in order to provide an accurate representation of the circuit's timing state.

In order to enable CDC support in conventional STA, the following steps have to be taken:

1. Determine clock firing times
2. Reconfigure delay propagation
3. Reconfigure slack computation

4. Reconfigure RAT propagation

3.2 Clock firing times

As it will be shown further on, clock firing times have a significant impact on timing analysis. The first step to enable CDC support in STA is to derive the closest launch and capture edges between all clock pairs. There are four possible clock combinations between two clocks that depend on their period and their first active edge:

1. Same period, launch clock fires before capture clock (figure 3.3)
2. Same period, launch clock fires after capture clock (figure 3.4)
3. Different period, $T_{launch} < T_{capture}$ (figure 3.5)
4. Different period, $T_{launch} > T_{capture}$ (figure 3.6)

3.2.1 Synchronous Clock Domain Crossings

Cases 1 and 2 are fairly easy to handle, as the difference between the two clocks is constant and equal to their phase shift. Such cases are also refered to as *Synchronous Clock Domain Crossings*. The launch clock firing time (LCFT) can be the clock's first active edge in both cases. For case 1, capture clock firing time (CCFT) will equal to $LCFT + phase\_shift$ whereas for case 2, CCFT will equal to $LCFT + T_{LCFT} - phase\_shift$.

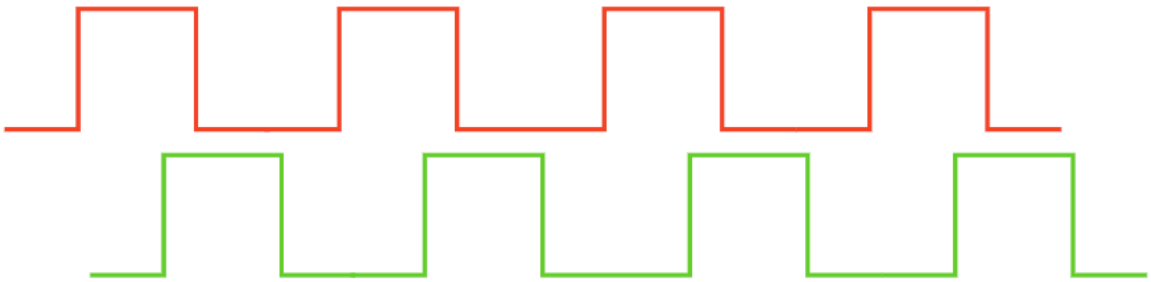

Figure 3.3: Same period, launch clock fires before capture clock

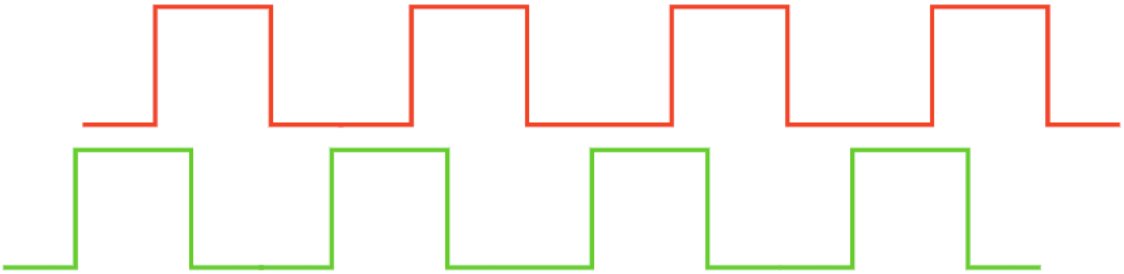

Figure 3.4: Same period, launch clock fires after capture clock

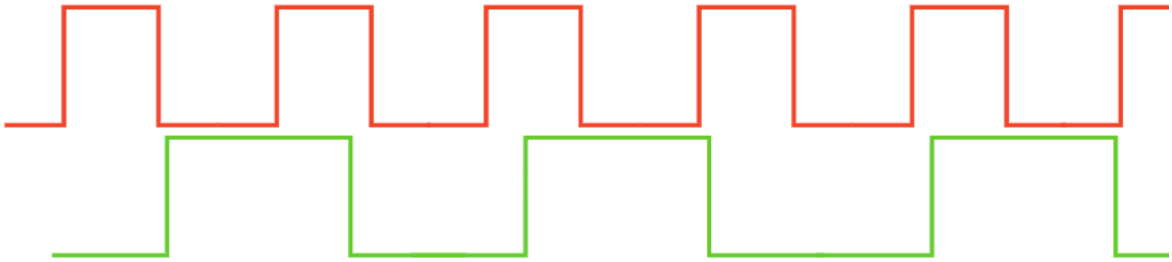

Figure 3.5: Different period, $T_{launch} < T_{capture}$

3.2.2 Asynchronous Clock Domains

Asynchronous Clock Domains are clock domains that have no relation between their periods. Such cases cannot be solved by simply finding the phase shift between the two waveforms, as the distance between the edges varies. The following is an example of such unrelated clocks:

Clk 1: Period = 2, Waveform = 0,1

Clk 2: Period = 4.5, Waveform = 2, 3

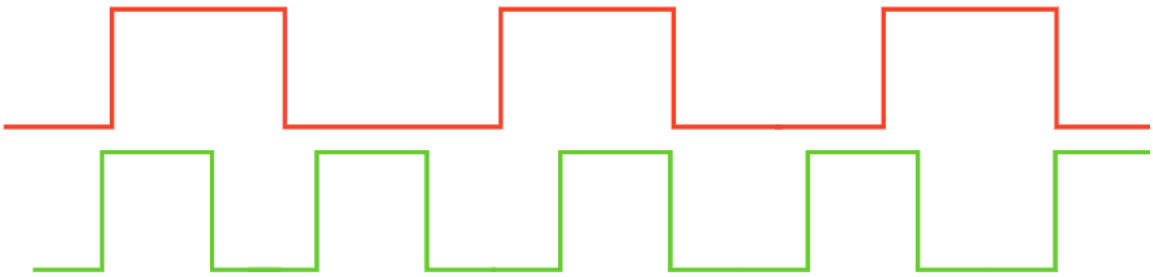

Figure 3.6: Different period, $T_{launch} > T_{capture}$

By unfolding these two waveforms, a pattern emerges:

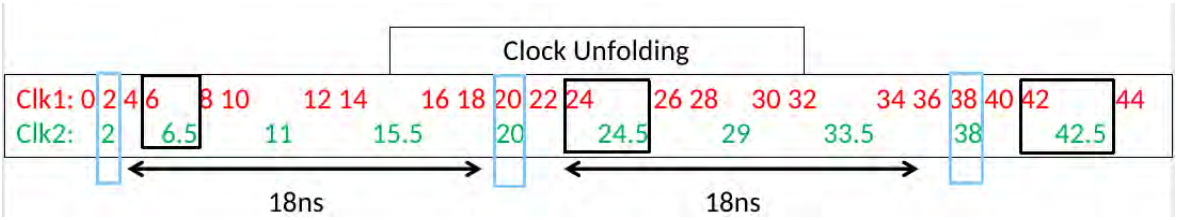

Figure 3.7: Clock waveform unfolding

First and foremost, it is obvious that, despite the clocks being completely unrelated, there are times when the two waveforms overlap. Second, the distance between the overlaps is the same and equal to the periods' Least Common Multiplier (LCM). Third, the smallest distance between the two clocks is the same and can be found between two overlapping points. Finally, the overlapping times are always equal to LCM + an offset. This offset is the first time the two waveforms align.

Consequently, as described also in [20], in order to find the two closest edges between the clocks, the waveforms must be unfolded within a certain timing window. This step can be summarized by algorithm 3.1:

Listing 3.1: Compute Unfolding Timing Window Bounds (CUTWB)

```
1  Description: Compute two times the two waveforms overlap to set the unfolding timing
       window
2  Inputs: p1, e1, p2, e2 >= 0
3  Outputs: lowerBound, upperBound
4     if p1 > p2 then
5        largeT = p1
6        largeActiveEdge = e1
7        smallT = p2
8        smallActiveEdge = e2
9     else
10       largeT = p2
11       largeActiveEdge = e2
12       smallT = p1
13       smallActiveEdge = e1
14    endif
15
16    periodLCM = LCM(p1, p2)
17    lowerBound = periodLCM
18    currLE = largeActiveEdge
19    while currLE < periodLCM do
20       currSE = currLE - (currLE % smallT) + smallActiveEdge
21       if currLE == currSE then
22          lowerBound = currLE
23          break
24       endif
25
26       currentLE += largeT
27    endwhile
28
29    upperBound = lowerBound + periodLCM
30    return lowerBound, upperBound
```

Having acquired the two bounds, the two closest edges can be found using algorithm 3.2:

Listing 3.2: Compute Waveforms' Closest Edges

```
Description: Aquire the unfolding timing window and compute the closest clock waveform
    edges between two clocks
Inputs: LCP, LAE, CCP, CAE >= 0
Outputs: launchClockEdge, captureClockEdge
   if LCP == CCP then
      phase_shift = abs(LAE - CAE)
      launchClockEdge = LAE
      if LAE < CAE then
         captureClockEdge = CAE
      else
         captureClockEdge = LAE + LCP - phase_shift
      endif
      return launchClockEdge, captureClockEdge
   endif

   lowBound, highBound = CUTWB (LCP, LAE, CCP, CAE)

   if CCP < LCP then
      if lowBound == CAE then
         captureClockEdge = CAE
         launchClockEdge = CAE - LCP
         return launchClockEdge, captureClockEdge
      endif

      capEdge = lowBound
      while capEdge < highBound do
         edgeDist = capEdge % (LCP + LAE)
         launchClockEdge = capEdge - edgeDist
         capEdge += CCP
         keep edges with minimum edgeDist
      endwhile
   else
      if lowBound == LAE then
         launchClockEdge = LAE
         captureClockEdge = LAE + CCP
         return launchClockEdge, captureClockEdge
      endif

      launchEdge = lowBound
      while launchEdge < highBound do
         edgeDist = launchEdge % (CCP + CAE)
         captureClockEdge = launchEdge + edgeDist
         launchEdge += LCP
         keep edges with minimum edgeDist
      endwhile
   endif
```

3.3 Delay Propagation and Calculation

As discussed in chapter 2, single-clock STA traverses the circuit graph in from startpoints to endpoints. Delay at input gate pins is computed as

$$delay_{in} = delay_{pred} + interconnect\_delay_{pred->in}$$

whereas delay at output pins is computed as follows:

$$gate\_arc\_delay = LUT\_delay(Tr_{in}, C_{out})$$

$$delay_{out} = max(delay_{ins} + gate\_arc\_delay)$$

When CDCs are introduced into the design, however, simply selecting the maximum delay to propagate at output pins will lead to inaccurate results. Following, is an example of a simple CDC circuit: Using the methods discussed in section 3.2, when having this clock

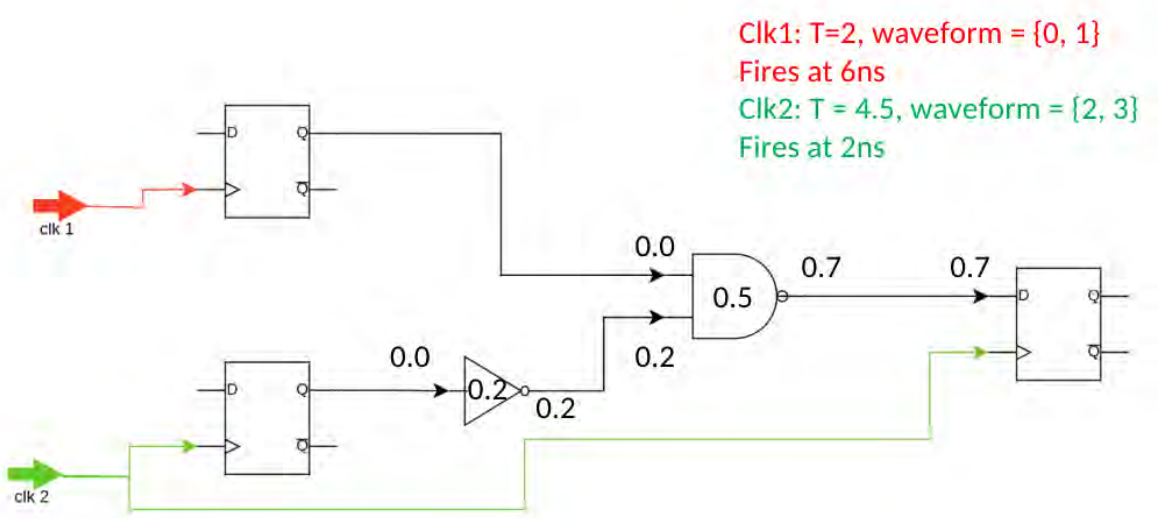

Figure 3.8: CDC corner case; Numbers in black represent single-clock STA's delay annotation

configuration launch clock firing times can be calculated as 6ns for launch FF at clk1 and 2ns for launch FF at clk2. If FF at clk1 fires at 6ns, the corresponding signal will reach the capturing FF at 6.5ns (by adding the NAND gate's delay to it). Launch FF at clk2 would send a signal which would be captured by the rightmost FF at 2.7ns (by adding inverter's and NAND gate's delays to it).

As a result, propagating only one delay value when dealing with CDCs can filter out values that may prove useful when computing slack at the FF's data pin. To prevent this, a vector of N

delay values is stored at each pin and propagated forwards, with N=the number of launching clocks that reach gate. At CDC points, where multiple clock domains converge, the maximum delay values *per individual clock domain* are stored. Figure 3.9 shows the vector of delays

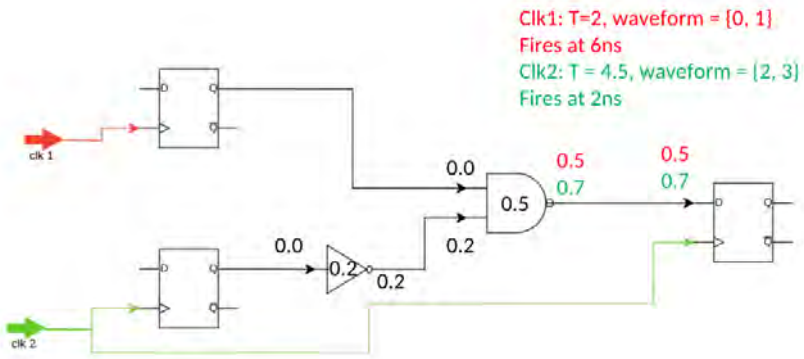

Figure 3.9: Delay vector propagation, per individual clock domain

that is propagated after the CDC point at NAND gate's output pin.

As far as delay calculation is concerned, the same method as in single-clock STA is used, with the exception that it takes place for every clock domain of every pin's predecessor. The algorithms are modified as follows:

Listing 3.3: CDC DelayCalcInputs

```
1   Inputs: inputPin
2   Outputs: delayV
3
4   delayV = []
5   predPin = predecessor(inputPin)
6   foreach clkDomain in clockDomains(predPin) do
7      domainDelay = delay(clkDomain)
8      delayV[clkDomain] = domainDelay + wireDelay(predPin, inputPin)
9   end for
10
11  return delayV
```

Listing 3.4: CDC DelayCalcOutputs

```
1  Inputs: outPin
2  Outputs: delayV
3
4  delayV = []
5  foreach predPin in predecessors(outPins) do
6     foreach clkDomain in clkDomains(predPin) do
7        domainDelay = delay(clkDomain)
8        foreach arcDelay in pinArcs(outPin, predPin) do
9           totalDelay = domainDelay + arcDelay
10       end for
11
12       if clkDomain not in clkDomains(outPin) or totalDelay > delayV[clkDomain] then
13          delayV[clkDomain] = totalDelay
14       end if
15    end for
16 end for
17
18 return delayV
```

3.4 Slack Computation

The slack computation formula shown in chapter 2 assumes that launch and capture clock firing times are the same and therefore, they are eliminated. A more detailed slack formula can be the following:

$$RAT = t_{CCF} + delay_{capture\_clock\_network} - FF\_setup\_time$$

$$AT = t_{LCF} + delaylaunch\_clock\_network + CLD(launchclockdomain)$$

$$Slack = RAT - AT$$

where t_{CCF} is the time of the capture clock firing, t_{LCF} is the time of the launch clock firing and CLD is the Combinational Logic Delay for the launch clock domain. Slack computation algorithm is modified as follows:

Listing 3.5: CDC Slack Computation

```
1   Inputs: FF data pin (D)
2   Outputs: FF data pin slack
3
4   pinSlack = inf
5   capClk = clock(D)
6   foreach launchClk in launchClks(D) do
7      launchEdge, capEdge = closestEdges(launchClkPeriod, launchActiveEdge, capClkPeriod,
           capActiveEdge)
8      launchDomainDelay = launchEdge + combDelay(launchClk, D)
9      D_RAT = capEdge + clkNetworkDelay(capClk) - setup(FF)
10
11     currSlack = D_RAT - launchDomainDelay
12
13     if pinSlack > currSlack then
14        pinSlack = currSlack
15     end if
16  end for
17
18  return pinSlack
```

Applying algorithm 3.5 to the previous example, the results are the following:

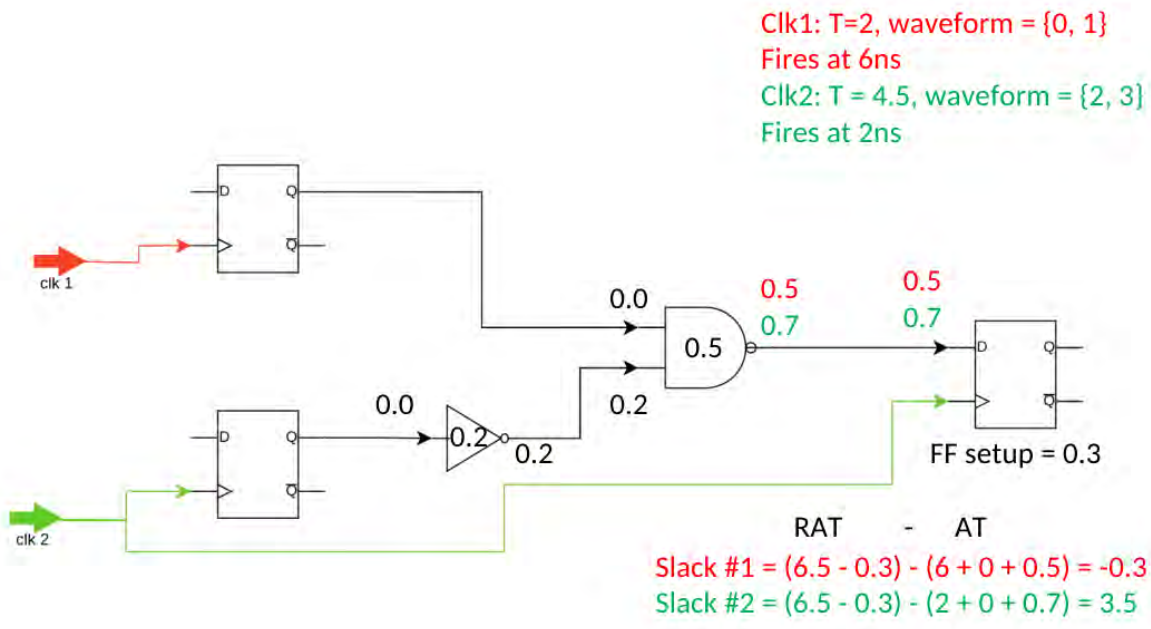

Figure 3.10: Slack with CDCs

Not only is the path with the *smaller* combinational delay the critical one, it actually *violates* setup slack, something that would not be seen by using single-clock STA's formulae for delay propagation/calculation and slack computation.

3.5 RAT Propagation

At the final step of CDC STA, RAT is propagated from the graph endpoints to its start-points and slack is computed for every pin. This step is completely symmetrical to delay propagation and calculation mentioned in section 3.3:

- Caption clocks are propagated backwards, similar to the launch clocks that are propagated during delay propagation

- At each traversed point, the RAT which results in the worst slack for said point is stored and propagated backwards to the rest of traversal's points.

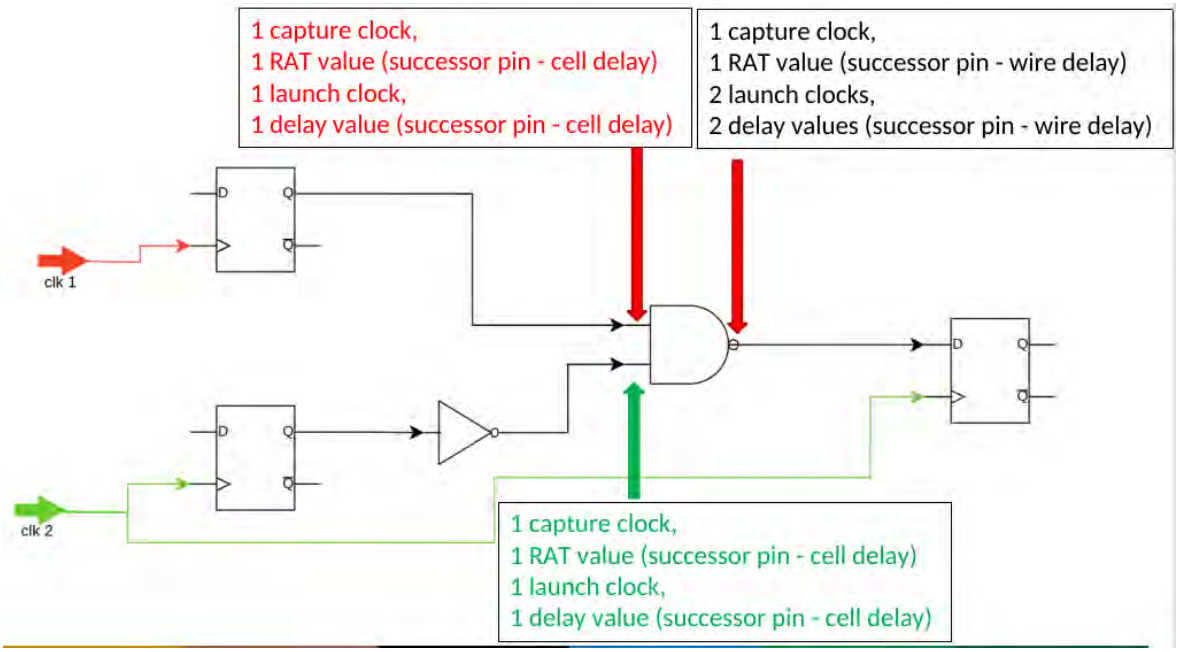

Figure 3.11: CDC RAT propagation, 1 capture clock

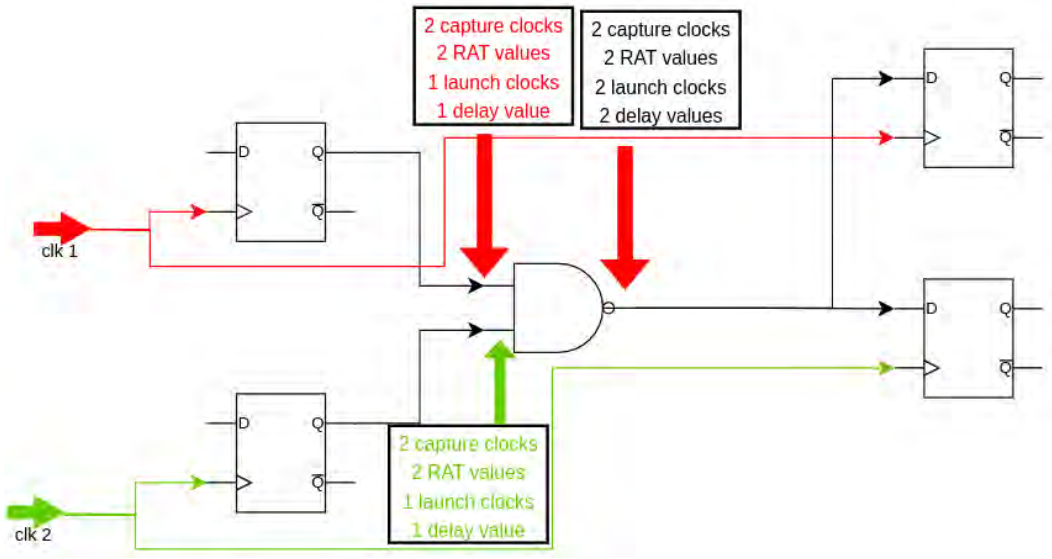

Figure 3.12: CDC RAT propagation, 2 capture clocks

This step can be summarized in the following algorithms:

Listing 3.6: CDC_RAT_AnnotateInputs

```
1  Inputs: Input pin (inPin), input pin delays per launch clock (delayV)
2  Outputs: Input pin RAT, slack (inRAT, inSlack)
3
4  succPin = get_successor(inPin)
5  inSlack = inf
6  inRAT = inf
7  foreach launchClock in launchClocks[inPin] do
8     foreach capClock in capClocks[succPin] do
9        if capClock not in capClocks[inPin] then
10          append capClock to capClocks[inPin]
11       end if
12       launchEdge, capEdge = closestEdges(launchClkPeriod, launchActiveEdge, capClkPeriod
             , capActiveEdge)
13       launchDomainDelay = launchEdge + delayV[launchClock]
14       foreach arcDelay in arcDelays[inPin, succPin] do
15          currRAT = capEdge - arcDelay
16
17          currSlack = currRAT - launchDomainDelay
18
19          if inSlack > currSlack then
20             inSlack = currSlack; inRAT = currRAT
21          end if
22       end for
23    end for
24 end for
25
26 return inRAT, inSlack
```

Listing 3.7: CDC_RAT_AnnotateOutputs

```
1  Inputs: Output pin (outPin), output pin delays per launch clock (delayV)
2  Outputs: Output pin RAT, slack (outRAT, outSlack)
3
4  succPin = get_successor(outPin)
5  outSlack = inf
6  outRAT = inf
7
8  foreach launchClock in launchClocks[outPin] do
9     foreach succPin in get_successors(outPin) do
10       foreach capClock in captureClocks[succPin] do
11          if capClock not in captureClocks[outPin] then
12             append capClock in captureClocks[outPin]
13          end if
14
15          launchEdge, capEdge = closestEdges(launchClkPeriod, launchActiveEdge,
                capClkPeriod, capActiveEdge)
16          launchDomainDelay = launchEdge + delayV[launchClock]
17          currRAT = capEdge - wire_delay(outPin, succPin)
18          currSlack = pinRAT - launchDomainDelay
19
20          if outSlack > currSlack then
21             outSlack = currSlack; outRAT = currRAT
22          end if
23       end for
24    end for
25 end for
26
27 return outRAT, outSlack
```

Chapter 4

Experiments

4.1 Experimental Flow

The goal of the following experiments is to showcase the accuracy of the CDC timing analysis methodology when compared to the industrial standards. Figure 4.1 shows the steps taken to achieve this. Experiments inputs are:

- A design in the form of a Verilog synthesized netlist
- The technology's timing library
- Design constraints defining clock periods, design input and output delays etc

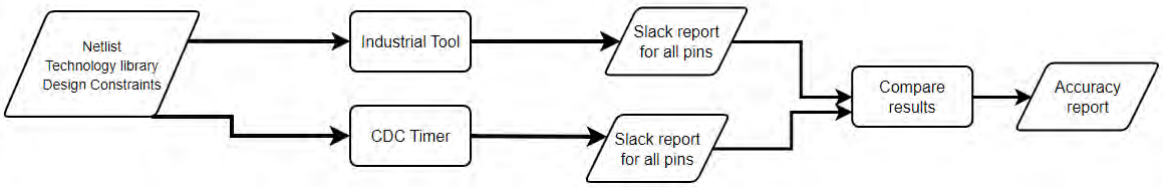

Figure 4.1: CDC Experiment Flow

These inputs were then given to the industrial tool and to CDC timer. Each tool produced a report with the worst computed slack for all the pins of the design. The two slack values were then compared for each pin of the design with a 0.01% error threshold.

4.2 Experiment Input Description

Experiment inputs were selected to cover all possible cases the CDC timer might encounter. The designs include:

1. One CDC point of two clocks with one capturing flip-flop (fig. 4.2)
2. One CDC point of two clocks, with a buffer chain on one of the clock domain paths, to enforce it as the slowest path. CDC point leads to one capturing flip-flop (fig. 4.3)
3. One CDC point of two clocks, with two capturing flip-flops (fig. 4.4)
4. One CDC point of two clocks, with a buffer chain on one of the clock domain paths, to enforce it as the slowest path. CDC point leads to two capturing flip-flops (fig. 4.5)
5. Two CDC points of 2-3 clocks, with two capturing flops (fig. 4.6)
6. Two CDC points of 2-3 clocks, with two capturing flops and buffers randomly inserted to the clock network (fig. 4.7)

Design constraints include:

1. A synchronous CDC (2-3 clocks where their periods are multiples),
2. An asynchronous CDC (2-3 clocks where their periods are unrelated)
3. Single clock, to showcase the difference between signle-clock STA and CDC STA

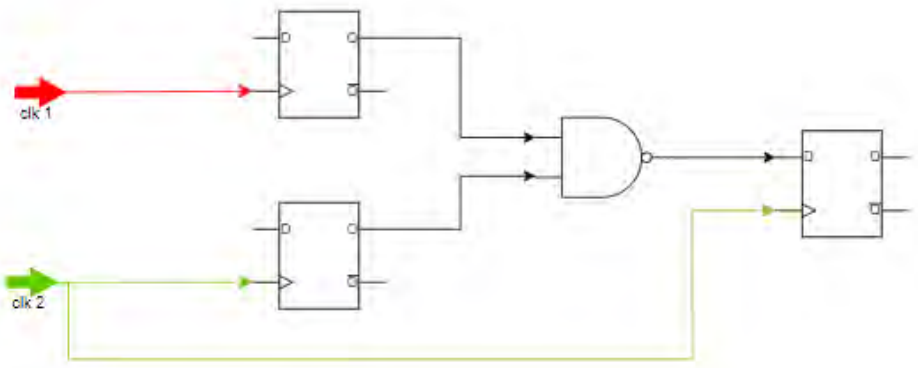

Figure 4.2: CDC Case 1

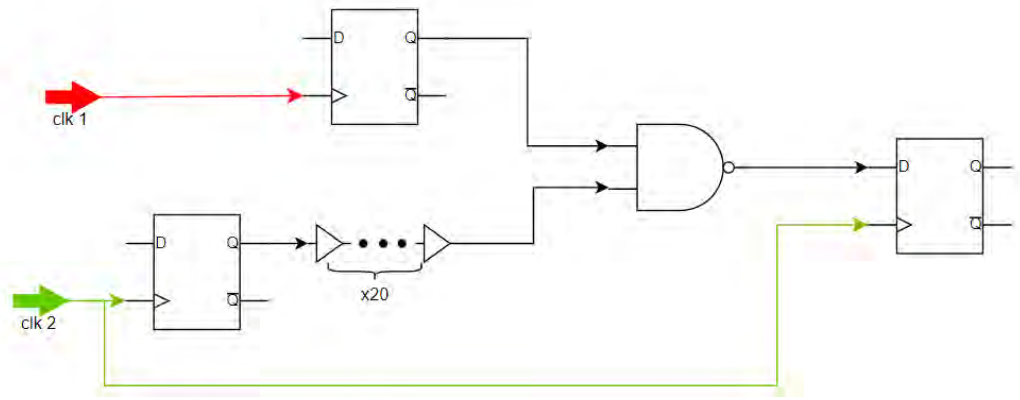

Figure 4.3: CDC Case 2

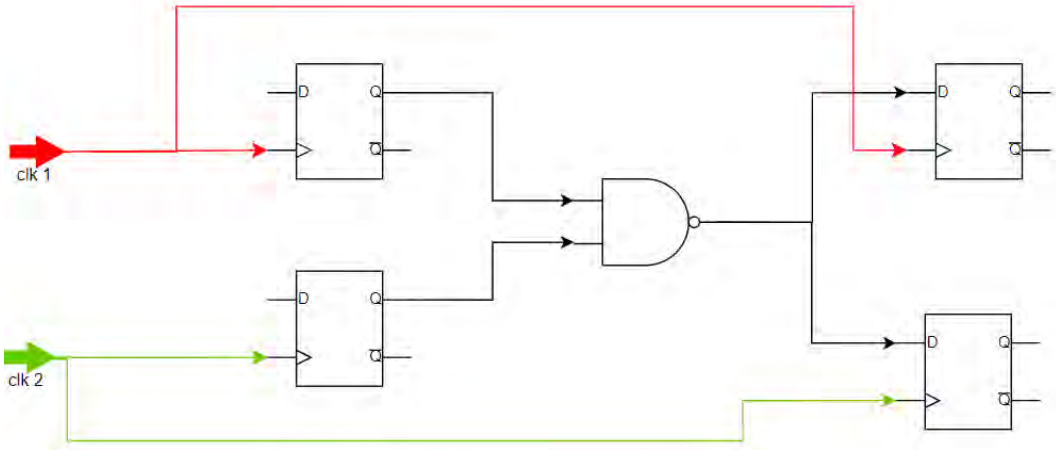

Figure 4.4: CDC Case 3

Listing 4.1: Single-Clock STA constraints

```
1 create_clock clk1 -period 2 -waveform {0 1}
2 create_clock clk2 -period 2 -waveform {0 1}
3 create_clock clk3 -period 2 -waveform {0 1}
4
5 set_input_delay 0 -clock clk1 [all_inputs]
6 set_output_delay 0.5 -clock clk1 [all_outputs]
```

Listing 4.2: Synchronous CDC constraints

```
1 create_clock clk1 -name clk1 -period 2 -waveform {0 1}
2 create_clock clk2 -name clk2 -period 4 -waveform {0 2}
3 create_clock clk3 -name clk3 -period 8 -waveform {0 4}
4
5 set_input_delay 0 -clock clk1 [all_inputs]
6 set_output_delay 0.5 -clock clk1 [all_outputs]
```

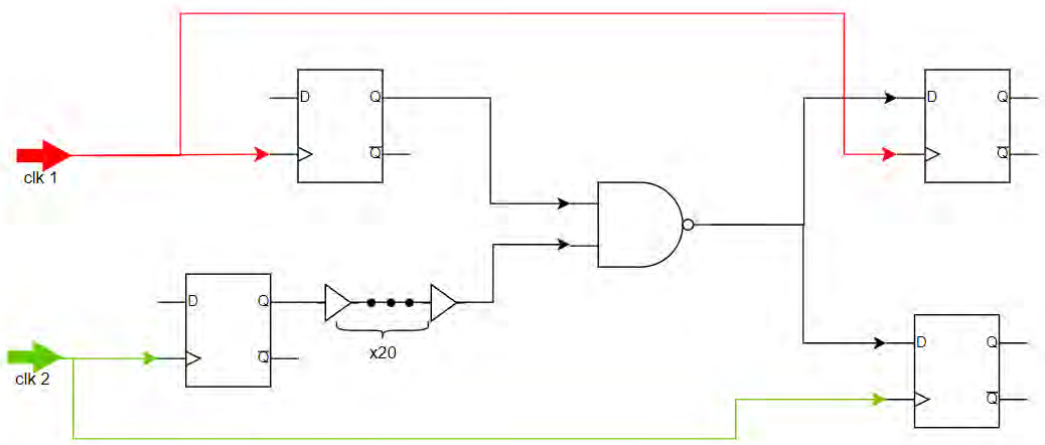

Figure 4.5: CDC Case 4

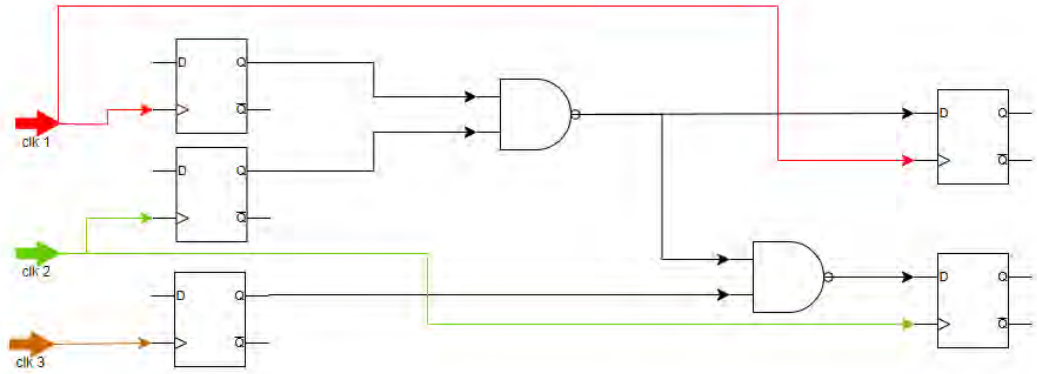

Figure 4.6: CDC Case 5

Listing 4.3: Asynchronous CDC constraints

```
1 create_clock clk1 -period 2 -waveform {0 1}
2 create_clock clk2 -period 4.5 -waveform {0 2.25}
3 create_clock clk3 -period 5 -waveform {0 2.5}
4
5 set_input_delay 0 -clock clk1 [all_inputs]
6 set_output_delay 0.5 -clock clk1 [all_outputs]
```

4.3 CDC STA vs Single-Clock STA

The difference between CDC vs Single-Clock STA can be clearly seen in cases 4.3 and 4.5. When applying the constraints shown in 4.1 to case 4.3, the critical path passes through the buffer chain (figure 4.8), as expected. However, the critical path is different when applying

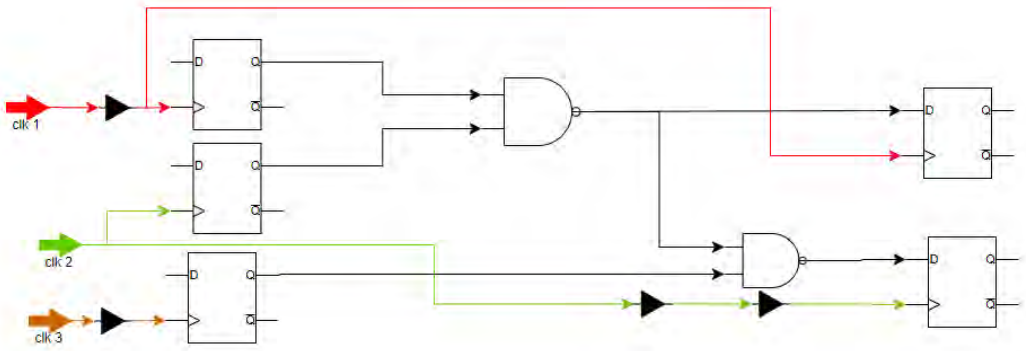

Figure 4.7: CDC Case 6

the constraints from 4.3, without modifying the circuit at all. (figure 4.9).

4.4 Experimental Results

Table 4.1 shows the results of the CDC timer when compared to the industrial tool. Each cell shows the total error percentage in slack values for each design, with all constraint combinations. The difference in slack values appeared after the 6-7th decimal digit, meaning there is a *1e-7ns* deviation in the two tools, which can even be attributed to rounding errors.

	Single-Clock	Synchronous CDC	Asynchronous CDC
Design 1	0%	0%	0%
Design 2	0%	0%	0%
Design 3	0%	0%	0%
Design 4	0%	0%	0%
Design 5	0%	0%	0%
Design 6	0%	0%	0%

Table 4.1: Experimental results

```
---------------------------------------
Dumping Longest Path to Gatepin cdc2_single_clock/capture/D
1: clk1 (Delay = 0.0000000 r) (r: 0.0000000, f: 0.0000000) (Load: 0.011400) (Wireload: 0.000000)
2: launch2/CP (dfp3d1) (Delay = 0.0000000 r) (r: 0.0000000, f: 0.0000000) (Load: 0.000000) (Wireload: 0.000000)
3: launch2/Q (dfp3d1) (Delay = 0.2814659 r) (r: 0.1021520, f: 0.0862444) (Load: 0.003800) (Wireload: 0.000000)
4: buf1/I (ni01d4) (Delay = 0.2814659 r) (r: 0.1021520, f: 0.0862444) (Load: 0.000000) (Wireload: 0.000000)
5: buf1/Z (ni01d4) (Delay = 0.4586436 r) (r: 0.0317983, f: 0.0250015) (Load: 0.003800) (Wireload: 0.000000)
6: buf2/I (ni01d4) (Delay = 0.4586436 r) (r: 0.0317983, f: 0.0250015) (Load: 0.000000) (Wireload: 0.000000)
7: buf2/Z (ni01d4) (Delay = 0.6026890 r) (r: 0.0307448, f: 0.0237219) (Load: 0.003800) (Wireload: 0.000000)
8: buf3/I (ni01d4) (Delay = 0.6026890 r) (r: 0.0307448, f: 0.0237219) (Load: 0.000000) (Wireload: 0.000000)
9: buf3/Z (ni01d4) (Delay = 0.7461984 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
10: buf4/I (ni01d4) (Delay = 0.7461984 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
11: buf4/Z (ni01d4) (Delay = 0.8897055 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
12: buf5/I (ni01d4) (Delay = 0.8897055 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
13: buf5/Z (ni01d4) (Delay = 1.0332126 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
14: buf6/I (ni01d4) (Delay = 1.0332126 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
15: buf6/Z (ni01d4) (Delay = 1.1767197 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
16: buf7/I (ni01d4) (Delay = 1.1767197 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
17: buf7/Z (ni01d4) (Delay = 1.3202268 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
18: buf8/I (ni01d4) (Delay = 1.3202268 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
19: buf8/Z (ni01d4) (Delay = 1.4637339 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
20: buf9/I (ni01d4) (Delay = 1.4637339 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
21: buf9/Z (ni01d4) (Delay = 1.6072411 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
22: buf10/I (ni01d4) (Delay = 1.6072411 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
23: buf10/Z (ni01d4) (Delay = 1.7507482 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
24: buf11/I (ni01d4) (Delay = 1.7507482 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
25: buf11/Z (ni01d4) (Delay = 1.8942553 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
26: buf12/I (ni01d4) (Delay = 1.8942553 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
27: buf12/Z (ni01d4) (Delay = 2.0377624 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
28: buf13/I (ni01d4) (Delay = 2.0377624 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
29: buf13/Z (ni01d4) (Delay = 2.1812695 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
30: buf14/I (ni01d4) (Delay = 2.1812695 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
31: buf14/Z (ni01d4) (Delay = 2.3247766 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
32: buf15/I (ni01d4) (Delay = 2.3247766 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
33: buf15/Z (ni01d4) (Delay = 2.4682837 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
34: buf16/I (ni01d4) (Delay = 2.4682837 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
35: buf16/Z (ni01d4) (Delay = 2.6117908 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
36: buf17/I (ni01d4) (Delay = 2.6117908 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
37: buf17/Z (ni01d4) (Delay = 2.7552979 r) (r: 0.0307402, f: 0.0237193) (Load: 0.003800) (Wireload: 0.000000)
38: buf18/I (ni01d4) (Delay = 2.7552979 r) (r: 0.0307402, f: 0.0237193) (Load: 0.000000) (Wireload: 0.000000)
39: buf18/Z (ni01d4) (Delay = 2.8936717 r) (r: 0.0276402, f: 0.0215438) (Load: 0.002300) (Wireload: 0.000000)
40: nand/A2 (nd21d1) (Delay = 2.8936717 r) (r: 0.0276402, f: 0.0215438) (Load: 0.000000) (Wireload: 0.000000)
41: nand/ZN (nd21d1) (Delay = 2.9774820 f) (r: 0.0779000, f: 0.0549053) (Load: 0.002300) (Wireload: 0.000000)
42: capture/D (dfp3d1) (Delay = 2.9774820 f) (r: 0.0779000, f: 0.0549053) (Load: 0.000000) (Wireload: 0.000000)
---------------------------------------
Data required arrival time: 1.5998758
Data arrival time: 2.9774820
Slack: (VIOLATED)        -1.377606
```

Figure 4.8: Case 2 critical path with single-clock STA constraints applied

```
---------------------------------------
Dumping Longest Path to Gatepin cdc2/capture/D
1: clk1 (Delay = 6.0000000 r) (r: 0.0000000, f: 0.0000000) (Load: 0.003800) (Wireload: 0.000000)
2: launch1/CP (dfp3d1) (Delay = 6.0000000 r) (r: 0.0000000, f: 0.0000000) (Load: 0.000000) (Wireload: 0.000000)
3: launch1/Q (dfp3d1) (Delay = 6.3988102 f) (r: 0.0844842, f: 0.0710000) (Load: 0.001700) (Wireload: 0.000000)
4: nand/A1 (nd21d1) (Delay = 6.3988102 f) (r: 0.0844842, f: 0.0710000) (Load: 0.000000) (Wireload: 0.000000)
5: nand/ZN (nd21d1) (Delay = 6.5220090 r) (r: 0.0779000, f: 0.0549053) (Load: 0.002300) (Wireload: 0.000000)
6: capture/D (dfp3d1) (Delay = 6.5220090 r) (r: 0.0779000, f: 0.0549053) (Load: 0.000000) (Wireload: 0.000000)
---------------------------------------
Data required arrival time: 6.1269476
Data arrival time: 6.5220090
Slack: (VIOLATED)        -0.395061
```

Figure 4.9: Case 2 critical path with async CDC STA constraints applied

Chapter 5

Conclusion

This thesis presented a Static Timing Analysis algorithm, which is used to determine whether a circuit satisfy all timing constraints under a user specified environment. Although STA is typically the fastest methodology for detecting timing violations, it is also the most pessimistic, as it does not take into account the circuit's functionality. Furthermore, the thesis presented Clock Domain Crossings, a case where more than one clock have been specified and paths constrained by different clocks converge to a single point. An algorithm for detecting the closest firing edges between two clocks has been proposed and the original STA algorithm was expanded to support CDCs. Experimental analysis covering all possible scenarios of CDCs showed 0% error compared to the industry standards.

Further development would be needed in order to improve the methodology's QoR in terms of runtime and memory consumption. This, among others, could include better data caching, a pre-processing step to identify the regions where many clocks converge for better memory management and more. Future work could include an integration of the current methodology with a Clock Tree Synthesis algorithm for more accurate clock network delays, providing feedback to an optimization loop to change clock network delay in such a way that CDC points do not violate timing and reporting CDC points in such a way that they are resolved in the RTL level.

Bibliography

[1] Stavros Simoglou, Christos Sotiriou, Dimitris Valiantzas, and Nikolaos Sketopoulos. Sta for mixed cyclic, acyclic circuits. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 392–397, 2020.

[2] O. Neiroukh, S.A. Edwards, and X. Song. An efficient algorithm for the analysis of cyclic circuits. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 6 pp.–, 2006.

[3] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H Madden. Performance optimization of vlsi interconnect layout. *Integration*, 21(1-2):1–94, 1996.

[4] Shyh-Chyi Wong, Gwo-Yann Lee, and Dye-Jyun Ma. Modeling of interconnect capacitance, delay, and crosstalk in vlsi. *IEEE Transactions on Semiconductor Manufacturing*, 13(1):108–111, 2000.

[5] John F Croix and DF Wong. Blade and razor: cell and interconnect delay analysis using current-based models. In *Proceedings of the 40th annual Design Automation Conference*, pages 386–389, 2003.

[6] Arif Ishaq Abou-Seido, Brian Nowak, and Chris Chu. Fitted elmore delay: a simple and accurate interconnect delay model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):691–696, 2004.

[7] A.B. Kahng and S. Muddu. An analytical delay model for rlc interconnects. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1507–1514, 1997.

[8] Olivier Coudert. An efficient algorithm to verify generalized false paths. In *Proceedings of the 47th Design Automation Conference*, DAC '10, page 188–193, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Jing-Jia Nian, Shih-Heng Tsai, and Chung-Yang Huang. A unified multi-corner multi-mode static timing analysis engine. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 669–674, 2010.

[10] Gor Petrosyan, Sargis Abovyan, and Tigran Harutyunyan. Modeling on-chip variations in digital circuits using statistical timing analysis. In *2010 East-West Design Test Symposium (EWDTS)*, pages 37–39, 2010.

[11] A. Krstic, Yi-Min Jiang, and Kwang-Ting Cheng. Pattern generation for delay testing and dynamic timing analysis considering power-supply noise effects. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):416–425, 2001.

[12] Igor Keller, King Ho Tam, and Vinod Kariat. Challenges in gate level modeling for delay and si at 65nm and below. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, page 468–473, New York, NY, USA, 2008. Association for Computing Machinery.

[13] C.J. Alpert, Dinesh Mehta, and Sachin Sapatnekar. Handbook of algorithms for physical design automation. 11 2008.

[14] Saurabh Verma and Ashima S Dabare. Understanding clock domain crossing issues. *EE Times*, 2007.

[15] David Rennie, David Li, Manoj Sachdev, Bharat L. Bhuva, Srikanth Jagannathan, ShiJie Wen, and Richard Wong. Performance, metastability, and soft-error robustness trade-offs for flip-flops in 40 nm cmos. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(8):1626–1634, 2012.

[16] T.C. Tang. Experimental studies of metastability behaviors of sub-micron cmos asic flip flops. In *[1991] Proceedings Fourth Annual IEEE International ASIC Conference and Exhibit*, pages P7–4/1, 1991.

[17] Gupta Vrinda, Kasim Mohammad, and Jebin Mohandas. Towards improving clock domain crossing verification for socs. *Journal of electrical and electronics engineering*, 14(2):19–24, 2021.

[18] Ang Boon Chong. Clock domain crossing verification challenges. In *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, pages 383–387, 2021.

[19] Shubhyant Chaturvedi. Static analysis of asynchronous clock domain crossings. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1122–1125. IEEE, 2012.

[20] J. Bhasker and R. Chadha. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer US, 2009.