ANALOG CLOCK TREE SYNTHESIS

A Thesis

by

Gökhan Güner

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Electrical and Electronics Engineering

Özyeğin University
January 2014

ANALOG CLOCK TREE SYNTHESIS

Approved by:

Assistant Professor H. Fatih Uğurdağ,
Advisor
Department of Electrical and Electronics
Engineering
Özyeğin University

Professor Günhan Dündar
Department of Electrical and Electronics
Engineering
Boğaziçi University

Assistant Professor Ahmed Akgiray
Department of Electrical and Electronics
Engineering
Özyeğin University

Date Approved: 16 January 2014

To my family

ABSTRACT

Design of clock distribution circuits for Sampled Data Analog Circuits (SDACs), is a manual process that takes serious work hours and is susceptible to errors that cause silicon respins. Providing an automatic or even a semi-automatic solution to this problem will benefit the industry greatly. The equivalent problem in the digital domain, named clock tree synthesis, is fully automated, and there are commercial software that handle it. This encouraged us to work on an automated flow for the analog problem. The analog version of the problem is similar to the digital version but there are key differences. While the goal in the digital problem is to distribute a source clock to thousands of end points with zero skew, the analog problem aims to distribute a source clock to a few hundred points with deliberate skew between some end points. In the analog problem, sometimes generating divided versions of the source clock and constraining their skew with respect to the source clock may also be part of the problem. Our approach not only speeds up the design of clock circuits for SDACs but also reduces the chances of a respin. As an added benefit, it speeds up the design of analog circuit as the designer does not need to spend time to make sure the clock routes are symmetric inside the analog design. Our proposed flow has four phases, namely, requirements analysis, target determination, design & synthesis, and verification. The first phase, requirements analysis, starts by interviewing the designer, continues with extraction of some physical parameters from the analog design, and results with a list of clock phases and timing constraints between them. The second phase of target determination has several graph-oriented tools. In this phase, we solve a specialized longest path problem efficiently to come up with a schedule of clock edges as a result that satisfies the constraints discovered in the first phase. In the third phase, we break up the clock circuit synthesis problem into two levels, namely, intrinsic and extrinsic clock trees, and drive a commercial clock tree synthesis software in an automated fashion with targets produced in the previous phase. The last phase is verification, in which we check to see if we satisfied the timing constraints we put together in the first phase. In this phase, we also do SPICE simulations and check if the circuit as a whole has acceptable figures of merit such as effective number of bits (ENOB). The conclusion is that our flow saves considerable design time and makes it less error-prone. The ENOBs obtained after our flow, when the flow is applied to a particular test design (a 10-bit 0.18 micron 2-step differential input 60 MSps Flash ADC), show that with this flow we are able to achieve ENOBs that are quite close to the best possible ENOBs under the given timing constraints. Last not but least, we have to mention that three phases of our flow (except the third one, design & synthesis) can be used with a manual clock tree design approach to make it more systematic, hence faster and less error-prone (i.e., the semi-automatic flow).

ÖZETÇE

Anahtarlamalı analog devrelerin (AAD) saat ağı tasarımı ciddi zaman gerektiren ve yongaların tekrar elden geçirilmesine sebep olabilecek hatalara açık, elle yapılan bir işlemdir. Bu probleme otomatik ya da yarı-otomatik bir çözüm getirmenin sektöre büyük katkısı olacaktır. Problemin dijital alandaki karşılığı olan saat ağı sentezinin otomasyonu mümkün olup, halihazırda bu işi yapan ticari yazılımlar bulunmaktadır. Bu durum, bizi problemin analog alandaki karşılığının otomasyonu üzerinde çalışmaya teşvik etti. Problemin analog hali dijital karşılığıyla benzer olmakla beraber bazı kilit noktalarda farklılıklar bulunmaktadır. Dijital problemdeki amaç kaynak saat sinyalini sıfır kaykı ile binlerce noktaya iletmek iken, analog devrelerde hedef kaynak saat sinyalini yüzlerce noktaya bazı noktalar arasında bir miktar kaykı olacak şekilde iletmektir. Ayrıca analog tasarımda kaynak saat sinyalinin bölünmüş versiyonlarını üretip, bu üretilmiş sinyallerin kaykılarını kaynak saat sinyaline bağlı olarak sınırlandırmak da problemin bir parçası olabilir. Yaklaşımımız AAD saat ağı devre tasarımını hızlandırmakla kalmayıp, tasarımın tekrarına sebep olacak hataların oluşumunu da azaltmaktadır. Yaklaşımımızın bir faydası da devredeki saat yollarının simetrik ayarlanması için vakit harcamaya gerek kalmamasından dolayı analog devrenin tasarımının süresinin de kısalmasıdır. Önerdiğimiz akış, gereksinim analizi, hedef belirleme, dizayn & sentez ve doğrulama adında dört aşamadan oluşmaktadır. İlk aşama olan gereksinim analizi, analog tasarımcı ile mülakat yaparak başlayıp, analog tasarımdan bazı fiziksel parametrelerin elde edilmesi ile devam edip, saat fazları ve bu fazlar arasındaki ilişkileri içeren bir listenin hazırlanmasıyla bitmektedir. Hedef belirleme isimli ikinci aşamada graf bazlı programlar kullanılmaktadır. Burada, özelleşmiş bir ”en uzun yol” problemini birinci fazda elde edilen kısıtları sağlayan bir saat çizelgesi çıkarmak amacıyla etkin bir biçimde çözmekteyiz. Üçüncü aşamada saat devre sentezi problemini iç ve dış saat ağacı adında iki seviyeye ayırıp, bir önceki aşamada üretilen hedefleri otomatik olarak kullandığımız ticari saat ağı sentez yazılımına iletmekteyiz. Son aşama olan doğrulama aşamasnda, ilk aşamada elde ettiğimiz zamanlama kısıtlarını sağlayıp sağlamadığımızı test etmekteyiz. Burada ayrıca devrenin etkin bit sayısı (EBS) gibi başarım ölçütlerini sağlayıp sağlamadığını kontrol etmek amacıyla SPICE simulasyonları da yapmaktayız. Vardığımız sonuç, önerdiğimiz akışın tasarım süresini önemli ölçüde azaltıp, hata ihtimalini oldukça azalttığı yönündedir. Akış sonucunda elde edilen EBS değerleri, akışı test tasarımımıza (10-bit 0.18 mikron 2-basamaklı farksal girişli 60 MSps Flash ADC) uyguladığımızda elimizdeki zaman kısıtları altında elde edebileceğimiz en olası EBS değerlerine oldukça yakın olduğumuzu göstermektedir. Son olarak, önerdiğimiz akışın üç fazını (üçüncü faz olan dizayn & sentez hariç) elle yapılan saat ağı tasarımında kullandığımız takdirde, elle yapılan tasarımı da daha sistematik, hızlı ve hataya toleranslı hale getirebildiğimizi (yarı-otomatik akış) belirtmeliyiz.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Assistant Prof. H. Fatih Uğurdağ for his continuous support throughout my M.S. studies.

Besides my advisor, I would like to thank the committee members Prof. Günhan Dündar and Assistant Prof. Ahmed Akgiray for sparing their valuable time.

I am also grateful to Dr. İskender Agi, who came up with the original problem at Intersil in 2006. The project is based on Dr. Agi's and Assistant Prof. H. Fatih Uğurdağ's collaboration during H. Fatih Uğurdağ's appointment at Intersil in Summer 2006 as a visiting professor.

I would also like to thank TÜBİTAK for both approving the project (under TÜBİTAK-1001 EEEAG-110E061), and for partially supporting me during my M.S. studies.

My sincere thanks go to "ACTreS" (Analog Clock Tree Synthesis) project team. This thesis wouldn't be possible without the efforts of professors Assistant Prof. H. Fatih Uğurdağ, Prof. Günhan Dündar, Assistant Prof. İ. Faik Başkaya and students Bilgiday Yüce, Vahap Barış Esen, Seyrani Korkmaz, Fatih Temizkan, Cihan Tunç.

I would also like thank my fellow labmates at Özyeğin University: Cihan Tunç, Fatih Temizkan, Abdullah Yıldız, Bilgiday Yüce, Furkan Şahin, Okan Palaz, Mert Kaya and Waqas Hussain for making my graduate life easier.

Last but not the least, I would like to thank my family for their support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Sampled Data Analog Circuits (SDACs) are used widely in many applications. Since they do not include resistors, as it can be seen from an example shown in Figure 1, their die areas are reasonable, their transfer functions are not too sensitive to process variations, and most importantly they can be implemented in CMOS. SDACs are used in designs such as capacitor filters, instrumentation amplifiers, voltage-to-frequency converters, balanced modulators, peak detectors, oscillators, data converters (ADCs and DACs) and Programmable Capacitor Arrays. A table that shows SDACs' region of operation can be seen in Table 1.

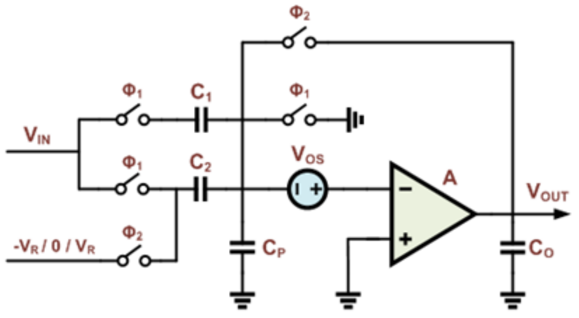


Figure 1: An SDAC example

Table 1: Classification of circuits by their voltage and temporal behaviors

		Time	
		Continuous	*Discrete*
Voltage	*Continuous*	Analog Circuits	SDACs
	Discrete	Asynchronous Digital Circuits	Synchronous Digital Circuits

Today's chips (analog or digital) are extremely complex and can not be designed without automation software. When design tasks are automated:

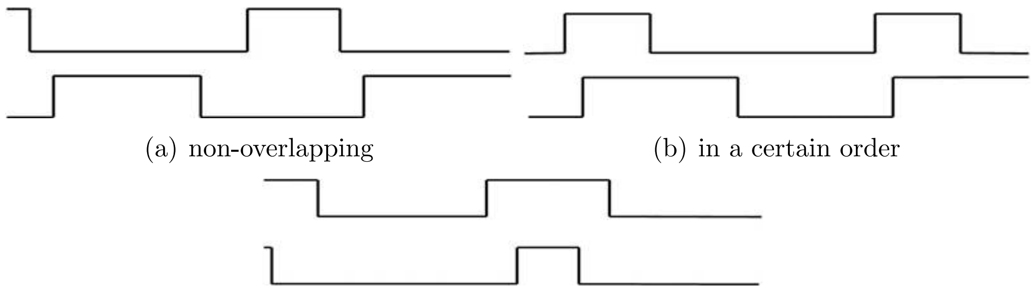
- development time is reduced,
- different architectures may be explored,
- errors are minimized,
- first manufactured chip is more likely to succeed.

SDACs require clock trees to operate, and designing SDAC clock trees manually takes too much time and forces the designers to use practices that are not exactly area friendly. Its design to product effort and methodology is not straightforward and may include several iterations. It is also more error prone, therefore it is more likely to waste much time and more resources this way.

Since SDACs require clock signals to operate, they behave somewhat like digital circuits. The sampling concept present in these circuits bear significant resemblance to their digital counterparts, like latches and flip-flops.

Transistors present in SDACs are controlled by different clock signals and these clock signals' timings are important to schedule the analog blocks accordingly. For instance, it may be required that a clock signal A should rise 200ps later than clock signal B's rising edge, or clock signal C should not be overlapping with clock signal A. There are 3 possible requirements in a clock tree that are mentioned below.

- non-overlapping, requires that some clock signals are not high at the same time. An example is in Figure 2(a).
- specific order, requires that some clock signals should come after/before other clock signals, as seen in Figure 2(b)
- enclosing, requires that a clock's high time envelopes another clock's high time, like in Figure 2(c).



(a) non-overlapping (b) in a certain order

(c) enclosing

Figure 2: Types of Clock Requirements for Clock Trees

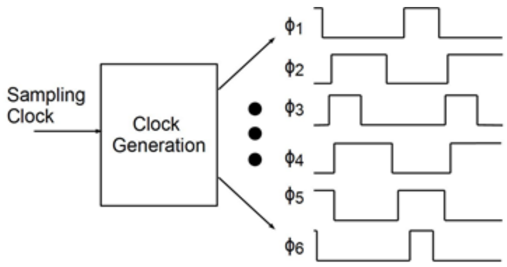
These different clock phases required in SDACs are actually derivatives of a main clock. The manual approach to SDAC clock tree design revolves around symmetrical design practices. This way, the designer can arrange timings by adjusting connection lengths.

Clock distribution network can be imagined as a two-level design. The first level is called "clock generation". Here, clock phases are generated from one main clock signal. The second level is the clock distribution part. At this level, phases generated at clock generation level are distributed to the receiving pins. Figures 3(a) and 3(b) visualize these levels.

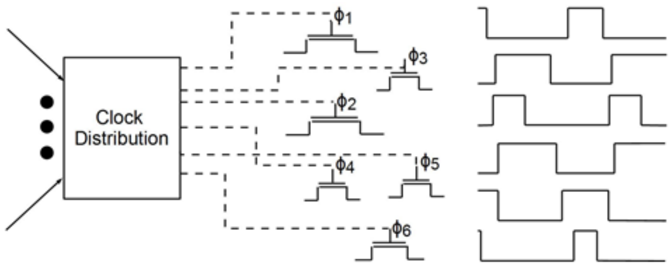
Looking at the "clock distribution network problem" above, it is evident that the main objectives are:

1. producing several phases from a single clock signal, and
2. maintaining the relationship between these phases while fanning each phase out to dozens of switches by keeping the skew under control.

Distribution of the main clock and/or its periodic multiples to a number of flip-flops in digital design is a similar situation. In case of using techniques such as "time stealing" and "good clock skew", a clock tree with each clock having different period, delay and duty cycle is necessary. In digital design, this problem is solved automatically by clock tree synthesis tools that are present in digital design software. Knowing



(a) Clock Tree Generation



(b) Clock Tree Distribution

Figure 3: Clock Tree Generation and Distribution

that digital design handles this problem automatically, while a similar problem in SDACs is handled manually is the idea behind this thesis.

In order to automate the process, providing a methodology is essential. In fact, formal approach provided by the methodology is a significant asset even without automation. The methodology proposed in this thesis consists of three main objectives that are:

1. providing a formal approach to SDAC clock tree design and developing analytical methods to calculate the target timings for the clock tree,
2. automation of the clock tree target timing calculation process,
3. interpreting the target timings calculated above to CTS (Clock Tree Synthesis) software, and running the CTS software from a script for full automation.

In order to test our methodology, our collaborators in Boğaziçi University tried to build a 10-bit, 0.18 micron, 180MHz flash ADC with 2-phase differential input [1]. However, our methodology was successful in building a 60MHz flash ADC with the remaining specifications. Our design can actually work at higher clock frequencies, but the effective number of bits (ENOB) goes down with the increased frequency. Thus, testing our methodology with 60MHz ADC seems to give the best trade-off according to the resolution vs speed relation.

The automation scripts take the clock requirements in a constraint list, and then checks whether the requirements are attainable or not. If the requirements seem satisfiable, the scripts use this constraint list to form a target list. Since this target list is the best that can be achieved theoretically, it is useful even if the clock tree is designed manually. So, automating the clock tree generation process shortens design time and minimizes errors for even manual approach. In addition to providing the target clocks to achieve, the methodology can also identify unattainable clock frequencies and clock constraints. In short, design automation for SDAC clock trees can be beneficial for even manual design.

Using the generation and automation scripts, a design flow that automatically generates required clock signals is established. Running the flow with the test design, a clock tree can be synthesized automatically. Proof of the methodology is done as follows.

1. Target clock values are applied to the test circuit schematic, and the effective number of bits (ENOB) are obtained.

2. Using time analysis of the layout synthesized by the Cadence Encounter tool, a clock list is obtained. Using the clock list on the test circuit schematic, another ENOB value is obtained.

3. First ENOB value is compared with the second. The more the second value is closer to the first value, the more successful the methodology is.

The ENOB values obtained by the flow above can be found at Chapter 6, and the results show that the methodology can be considered successful.

Figure 4 shows the critical stages of our flow that constitute ACTreS. **Requirements Analysis** is the initial stage that creates the base files such as the constraint file (which holds the relation between each clock phase) that will be processed throughout automation. **Target Determination** stage generates target clock phases that fit the constraints. Target clock phases calculated in **Target Determination** are fed to the digital CTS tool and the clock tree circuit that satisfies the targets is synthesized at **Design and Synthesis** stage. In **Verification**, the resulting clock phase timings are confirmed with the constraints in the constraint file. If timing values pass the confirmation, SPICE files that represent the clock tree are generated. Each stage will be thoroughly explained in their respective chapters.

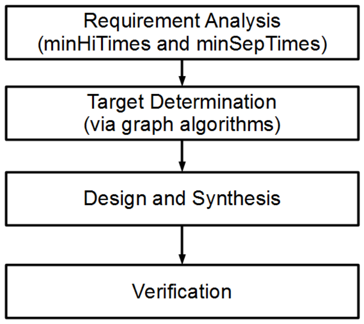


Figure 4: Stages of ACTreS

There are many publications on clock tree synthesis automation for digital chips in the literature as observed from the survey in [2]. However, there is no work directly dealing with general SDAC clock tree synthesis. There are works such as [3] and [4] that attempt to automate the complete circuit (even layout) design process for a particular subclass of SDACs. However, they do not include much detail in terms of CTS and how it can be generalized. There are also works that focus on the generation of two non-overlapping clocks ([5], [6]). Two non-overlapping clocks is enough for some SDACs, but for many SDACs we need more phases such as multi-rate SDACs [7], some SDACs for power electronics applications [8], and SDACs with interleaved operation [9], which is a subclass that our test circuit also falls in.

Even though there are also numerous publications and mostly patents on non-overlapping clock generation for specific designs, there is no publication for generalized SDAC clock tree generation. The reasons could be that,

1. Researchers that are proficient in analog design generally work on specification of a common design. These specific design problems are seriously hard and overall design automation is generally impossible.

2. Even it is time consuming and error prone, it is still possible to do SDAC clock tree generation manually. Analog designer workforce in the industry may be sufficient so that the task is not a priority.

3. Researches that have the required expertise on clock tree automation for SDACs are most probably working on digital clock tree automation. Since these people have limited knowledge on analog design, they may not be aware of such automation problem for SDACs.

Creating a software from scratch for a serious design project like our ADC design requires considerable amount of workforce and resources. On the other hand, the automation practices present in commercial software used in the industry (in our case, CTS tool in Cadence Encounter) are sophisticated and up-to-date. So, using a digital CTS tool for SDAC clock tree synthesis was a viable option.

The normal digital CTS paradigm is about distributing a clock to many end points with zero skew if possible. That is useful in our problem in distributing the generated clock phases but does not help generate the various clock phases an SDAC needs. Thanks to the idea of clock-skew scheduling [10], digital CTS tools have the capability of generating deliberate timing offsets between end points and hence can be used to generate different phases of a main clock. Note that a preliminary and condensed version of our work was published in [11]. At the time we were trying to clock our test circuit at 180 MHz. However, we had some analog design challenges (not in the CTS part) that we could not overcome. To successfully complete the analog design of our test circuit, we had to lower the clock frequency to 60 MHz.

Further information on our design flow can be found in the following chapters.In Chapter II (Requirements Analysis), we cover the base file generation (like the constraint file) for automation. In Chapter III (Target Determination), we generate target clock phases that fit the constraints. Chapter IV (Design and Synthesis) guides us through the clock tree synthesis step. In Chapter V (Verification), we explain confirmation process for the synthesized clock phase timings.

Figure 4 shows the critical stages of our flow that constitute ACTreS. **Requirements Analysis** is the initial stage that creates the base files such as the constraint file (which holds the relation between each clock phase) that will be processed throughout automation. **Target Determination** stage generates target clock phases that fit the constraints. Target clock phases calculated in **Target Determination** are fed to the digital CTS tool and the clock tree circuit that satisfies the targets is synthesized at **Design and Synthesis** stage. In **Verification**, the resulting clock phase timings are confirmed with the constraints in the constraint file. If timing values pass the confirmation, SPICE files that represent the clock tree are generated. Each stage will be thoroughly explained in their respective chapters.

CHAPTER II

REQUIREMENTS ANALYSIS

CTS software and the design scripts require certain files to solve the clock tree problem. Clock requirements and other important parameters such as pin capacitance, delay and location values of the SDAC should be provided in a predefined structure. Moreover, for maximum performance, each library cell used in clock tree generation should also be analyzed. In this chapter, the preliminary work handled before running the generation scripts is instructed. Flowchart that shows the steps that make up Requirements Analysis can be seen in Figure 5.

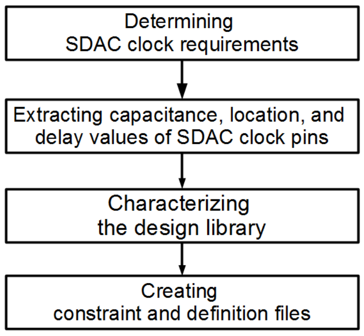

Determining
SDAC clock requirements
Extracting capacitance, location, and
delay values of SDAC clock pins
Characterizing
the design library
Creating
constraint and definition files

Figure 5: Requirements Analysis flowchart

2.1 Determining the Clock Requirements for Sampled Data Analog Circuits

In order to design the SDAC clock tree, the clock requirements should be determined first by observing the SDAC circuit. For this purpose, the general architecture of our test design (ADC) is provided in Figure 6. Each switch symbol provided in the figure represents an array of switches instead of one.

Figure 7 shows the timing diagram derived from the test circuit. We can also

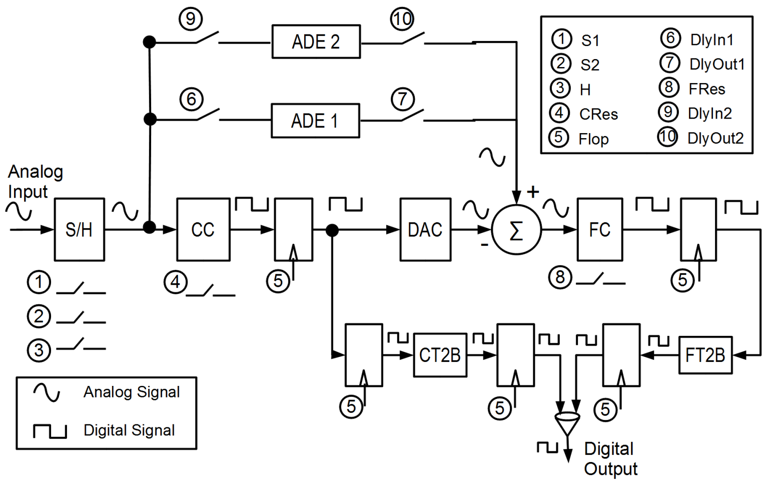


Figure 6: Test ADC Design Architecture

observe the work done at each time slot, and the pipelining of the process if we look at time steps 5-6. Our ADC operation consists of 10 sub-processes that are: ***Smp, H, CRes, CC, DlyIn1/DlyIn2, DlyOut1/DlyOut2, FRes, FC, CT2B, FT2B***.

Taking the diagram on Figure 7 into account, we can observe that the throughput of our circuit is 2 time steps, while its latency is 6 time steps. We must keep in mind that each time step seen in Figure 7 equals half of the main clock period.

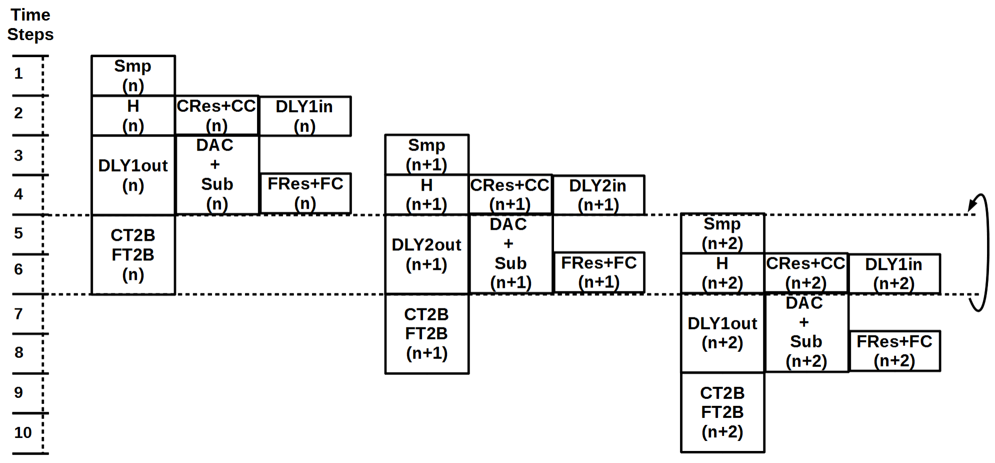


Figure 7: Timing Diagram of ADC Design

The test ADC's operations are explained briefly below (following Figure 7 step by step):

- Smp: Here we do sampling. Sampling is controlled by switches S1 and S2 on Figure 6. Our S1 clock phase should cover the S2 clock phase. They have the same high time and period values which are $T_{clk}/2$ and T_{clk}

- H: H is the abbreviation for hold. When H switch on Figure 6 is closed, the sampled input signal during Smp operation is sent to the circuit for processing.

- CC + CRes: CC is the Coarse Comparison operation. By coarse comparison, we try to determine the upper part of our ADC's output (most significant bits). CRes (Coarse Reset) resets the coarse comparator output to zero. When CRes switch on Figure 6 is open CC does comparison, otherwise it is zero.

- DlyIn: is a sub-operation of Analog Delay Element (ADE) operation, which is a 3 step long operation. It is one step long.

- DlyOut: is the second sub-operation of ADE. It takes two steps.

- DAC + Sub: Digitized output of CC is converted to an analog signal again by the DAC block. The analog output from DAC block is subtracted from ADE blocks by Sub operation. These two blocks do not require a clock phase.

- FC + FRes: FC is the Fine Comparison block. Fine comparison tries to determine the lower part of our ADC's output (least significant bits). FRes (Fine Reset) resets the fine comparator output to zero. When FRes switch on Figure 6 is open FC does comparison, otherwise it is zero.

- CT2B + FT2B: 31-bit thermometer codes produced by CC and FC blocks are converted to 5-bit binary codes by CT2B and FT2B blocks respectively. As result, we finally obtain a 10-bit digital ADC output.

Looking at the time diagram on Figure 7, we are able to determine the ideal clock waveforms on Figure 8.

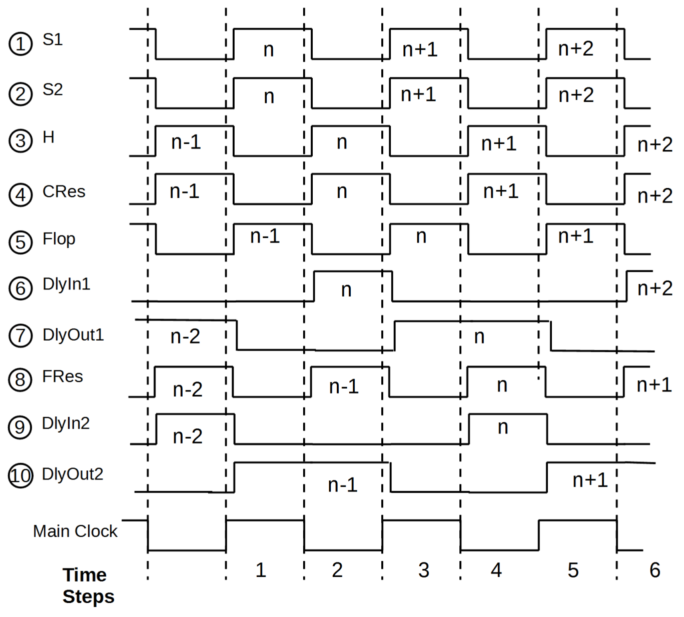


Figure 8: Ideal clock phases

Looking at both Figure 7 and Figure 8, clock requirements below are determined:

1. S2 w/in S1 (1r $\rightarrow$ 2r and 2f $\rightarrow$ 1f)

2. S1 and H NO (1f $\rightarrow$ 3r and 3f $\rightarrow$ 1r)

3. 'negedge CRes' before 'posedge Flop' (4f $\rightarrow$ 5r)

4. DlyIn w/in H (3r $\rightarrow$ 6r (9r) and 6f (9f) $\rightarrow$ 3f)

5. 'Flop posedge' before 'H negedge' (5r $\rightarrow$ 3f)

6. 'DlyOut posedge' before 'DlyIn negedge' (7r (10r) $\rightarrow$ 6f (9f))

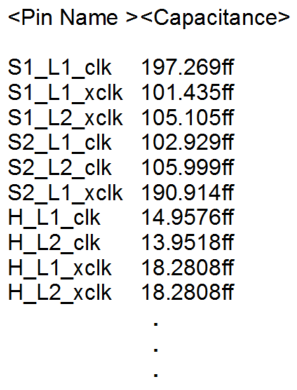
7. 'Flop posedge' before 'DlyOut posedge' (5r $\rightarrow$ 7r (10r))

8. 'CRes posedge' before 'H negedge' (4r $\rightarrow$ 3f)

Here ’NO’ represents Non-overlapping, ’w/in’ is within, posedge and negedge correspond to positive and negative edges of a signal, ’a → b’ means ’from a to b’ and ’f’ and ’r’ are for falling and rising edges respectively.

2.2 *Extraction of Capacitance, Location and Delay Values of the SDAC Clock Pins*

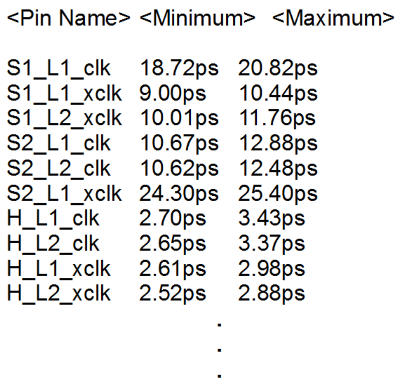
After determining the clock requirements and synthesizing the SDAC (without clock tree), capacitance (pincaps.txt), location (pinlocs.txt) and delay (pinrcs.txt) values should be extracted from the synthesized analog circuit. The extraction can be done by the analog design software that was used in synthesis, and the resulting files are copied into the ’pin_locs’ folder that can be seen on Figure 48(c). ACTreS then refers to these files whenever they are required. An example of these files’ contents can be observed in Figure 9.



```
<Pin Name ><Capacitance>

S1_L1_clk     197.269ff
S1_L1_xclk    101.435ff
S1_L2_xclk    105.105ff
S2_L1_clk     102.929ff
S2_L2_clk     105.999ff
S2_L1_xclk    190.914ff
H_L1_clk      14.9576ff
H_L2_clk      13.9518ff
H_L1_xclk     18.2808ff
H_L2_xclk     18.2808ff
                  .
                  .
                  .
```

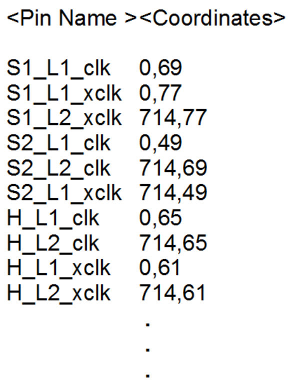
(a) pin_caps.txt



```
<Pin Name> <Minimum>  <Maximum>

S1_L1_clk     18.72ps   20.82ps
S1_L1_xclk    9.00ps    10.44ps
S1_L2_xclk    10.01ps   11.76ps
S2_L1_clk     10.67ps   12.88ps
S2_L2_clk     10.62ps   12.48ps
S2_L1_xclk    24.30ps   25.40ps
H_L1_clk      2.70ps    3.43ps
H_L2_clk      2.65ps    3.37ps
H_L1_xclk     2.61ps    2.98ps
H_L2_xclk     2.52ps    2.88ps
                      .
                      .
                      .
```

(b) pin_rcs.txt



```
<Pin Name ><Coordinates>

S1_L1_clk     0,69
S1_L1_xclk    0,77
S1_L2_xclk    714,77
S2_L1_clk     0,49
S2_L2_clk     714,69
S2_L1_xclk    714,49
H_L1_clk      0,65
H_L2_clk      714,65
H_L1_xclk     0,61
H_L2_xclk     714,61
                 .
                 .
                 .
```

(c) pin_locs.txt

Figure 9: Extracted file contents from SDAC analog synthesis

2.3 *Design Library Characterization*

It is sometimes required to know how fast the design fares at fast PVT corner in respect to slow and normal PVT corners. To obtain a rough estimate, each library cell was tested on a binary tree. Resulting delays can be seen in Table 2.

Table 2: Delay values for the CTS design library cells

$Cell$ $Name$	$FastPVT$ (ns)	$NormalPVT$ (ns)	$SlowPVT$ (ns)	$Normal/Fast$ $(c_{fast-normal})$	$Slow/Fast$ $(c_{fast-slow})$
BUF1CK	0.10	0.15	0.25	1.50	2.50
BUF2CK	0.08	0.13	0.20	1.63	2.50
BUF3CK	0.09	0.12	0.20	1.33	2.22
BUF4CK	0.08	0.12	0.19	1.50	2.38
BUF6CK	0.07	0.11	0.17	1.57	2.43
BUF8CK	0.08	0.10	0.17	1.25	2.13
INV1CK	0.11	0.14	0.23	1.27	2.09
INV2CK	0.08	0.11	0.16	1.38	2.00
INV3CK	0.07	0.10	0.15	1.43	2.14
INV4CK	0.07	0.09	0.14	1.29	2.00
INV6CK	0.06	0.08	0.14	1.33	2.33
INV8CK	0.06	0.08	0.13	1.33	2.17

By looking at the worst case values on Table 2, fast-normal and fast-slow coefficients are determined as 1.6 and 2.5 respectively.

2.4 Creating Constraint and Definition Files

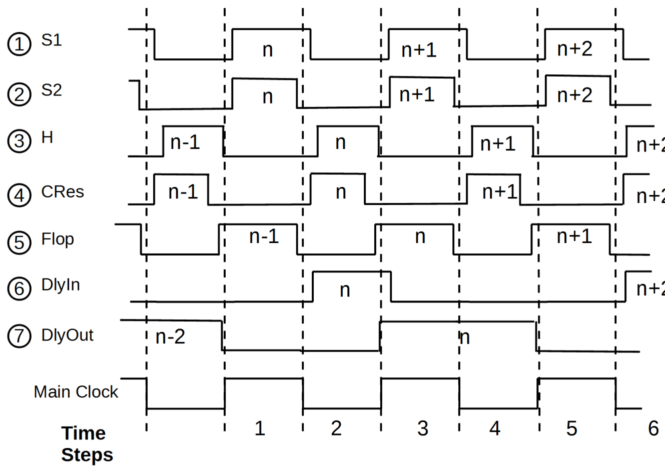
After determining the clock requirements, minimum timing for each requirement should be determined and the constraint file (constraints.txt) should be created. Here, the ideal clock phases determined for the ADC design depicted on Figure 8 are converted to an operable format.

As first step, it is important to determine the identical clock phases in order to create a simpler constraint file. This situation can be observed in our ADC circuit. CRes and FRes switch arrays have the exact same requirements. DlyIn1 and DlyIn2 clock phases have the same situation. Moreover, regular output of DlyOut1 (_clk) has the same requirements with the inverted output of DlyOut2 (_xclk). Taking these identical phases into account, we are going to use CRes (CRes and FRes), DlyIn (DlyIn1 and DlyIn2) and DlyOut (DlyOut1 and DlyOut2) for the phases mentioned above.

The second step is to identify the critical clocks and their timings. By critical clocks, we refer to the clocks that affect the overall performance more than the other clocks. We need to determine the minimum high time (minHiTime) for these critical clock phases. To determine, we rely on the mathematical error modelling of the analog circuit and the analog designer's experience. S2, DlyIn and DlyOut are the critical clock phases for our ADC design. After adding minHiTime values of the critical clock phases to the requirement list, we obtain the constraint file in Figure 10(a) and the realistic clock phases given in Figure 10(b) for T$_{clk}$ = 16.6ns.

```
1r  2r      0
2f  1f    150
1f  3r    150
3f  1r    150
4f  5r    900
3r  6r      0
6f  3f    150
5r  3f    150
7r  6f   1000
5r  7r      0
4r  3r    300
3r  4f  -3300
6r  6f  -1000
2r  2f  -5800
7r  7f  -4600
```

(a) constraint file 'constraints.txt'



(b) realistic clock phases

Figure 10: Constraint file and the clock phase waveform the constraint file is obtained from

As observed from Figure 10(a), the constraint file has the following format:

$<1^{st}$ clock number$><1^{st}$ clock edge$>$ $<2^{nd}$ clock number$><2^{nd}$ clock edge$>$ $<$t$_2$-t$_1$$>$

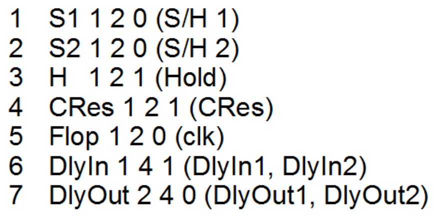
Here t$_2$-t$_1$ denotes the timing distance between two edges. The constraint file contains constraints such as minSepTime, minHiTime and minDurTime. With these constraints, we are able to determine a safe operation zone in n-dimensional (n =

number of clock phases) space for our SDAC.

Definition file is a simple input file that holds the following information for each clock phase:

<phase number> <phase name> <high time> <period> <xclk> <comment>

A definition file example can be seen on Figure 11.



```
1   S1 1 2 0 (S/H 1)
2   S2 1 2 0 (S/H 2)
3   H   1 2 1 (Hold)
4   CRes 1 2 1 (CRes)
5   Flop 1 2 0 (clk)
6   DlyIn 1 4 1 (DlyIn1, DlyIn2)
7   DlyOut 2 4 0 (DlyOut1, DlyOut2)
```

Figure 11: Definition file 'defFile.txt'

High time and period values in the definition file are in $T_{clk}/2$. The 'xclk' column in the format determines whether CTi will produce the regular clock (0) or the inverted clock (1).

CHAPTER III

TARGET DETERMINATION

The constraint file created in the previous chapter defines a safe zone from where our clock phases can be selected. However, feeding these constraints directly to the CTS tool may result in two potential problems given below.

- Some of the synthesized clock phases may be beyond bounds of the safe zone,
- Performance of the overall circuit may be suboptimal.

The chapter will focus on achieving and verifying suitable targets that will be fed to the CTS tool. Target generation methodology and scripts will be thoroughly explained in the process. Flowchart seen in Figure 12 shows the steps that make up the **Target Determination** phase of ACTreS.

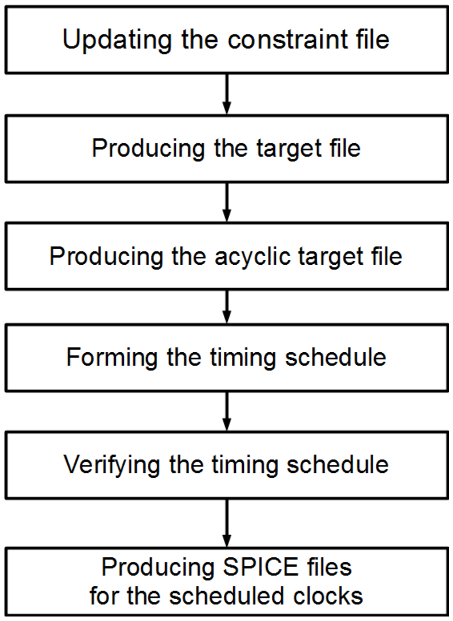

Updating the constraint file
Producing the target file
Producing the acyclic target file
Forming the timing schedule
Verifying the timing schedule
Producing SPICE files for the scheduled clocks

Figure 12: Target Determination flowchart

3.1 Updating the Constraint File and Observing its Graph

The analog circuit should satisfy the constraints for all PVT (Process, Voltage, Temperature) corners. Different constraint types in the constraint file achieve their lowest values at different PVT corners. For example, minSepTime constraints are the smallest at fast PVT corners. If a minHiTime or minDurTime constraint is smaller than the main clock's period, its value is smallest at fast PVT corner, else its value is smallest at slow PVT corner.

If we want better performance, we should update our constraint list according to the PVT corners. For this reason, PVT corner to aim should be determined for synthesis. It is clear that the design will not work properly if some minSepTime constraints can't be achieved. In the event of not satisfying some minHiTime constraints, the design will work with lower performance. Based on these information, it is evident that satisfying the minSepTime constraints is more critical. Therefore we need to aim for fast PVT corners.

To update the constraint file, speed ratio between fast and slow corners should be determined first. If we refer to Section 2.3, this ratio can be determined as $c_{(fast-slow)} = 2.5$. The constraint file can be updated by dividing negative minHiTime and minDurTime values to $c_{(fast-slow)}$. The difference between the original constraint file and the updated constraint file (constraints_fc.txt) can be observed in Figure 13.

Design automation starts with this task. The process of obtaining updated constraint file from the original one is handled by a Perl script called 'script1.pl'. Further information about the script can be found in Table 3.

The constraint files observed in Figure 13 can also be identified as directed graphs. This enables the use of directed graph algorithms on the constraint files. Directed graphs for the constraint files at Figure 13 can be seen in Figure 14. Each clock edge in the constraint file is implemented as a node, and each constraint in the files is an edge on the graph.

```
1r  2r      0
2f  1f    150
1f  3r    150
3f  1r    150
4f  5r    900
3r  6r      0
6f  3f    150
5r  3f    150
7r  6f   1000
5r  7r      0
4r  3r    300
3r  4f  -3300
6r  6f  -1000
2r  2f  -5800
7r  7f  -4600
```

(a) Original constraint file 'constraints.txt'

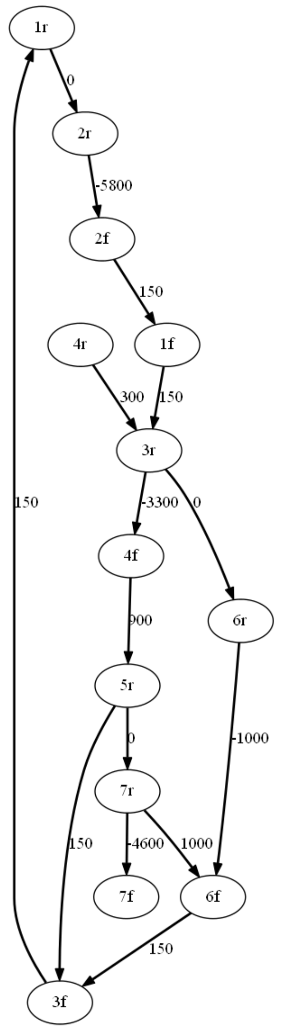
```
1r  2r      0
2f  1f    150
1f  3r    150
3f  1r    150
4f  5r    900
3r  6r      0
6f  3f    150
5r  3f    150
7r  6f   1000
5r  7r      0
4r  3r    300
3r  4f  -1320
6r  6f   -400
2r  2f  -2320
7r  7f  -1840
```

(b) Updated constraint file 'constraints_fc.txt'

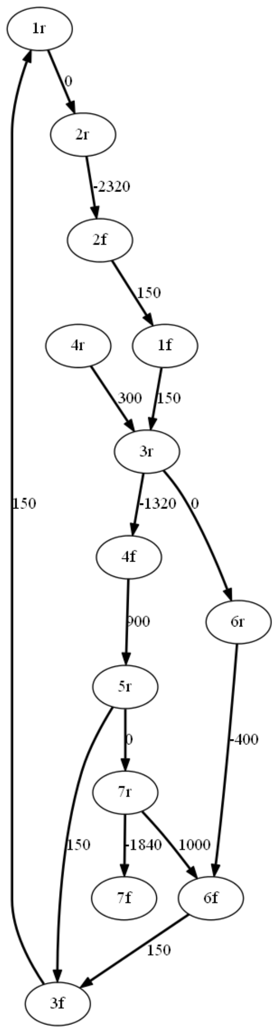
Figure 13: Updating the constraint file for fast corner

Table 3: 'script1.pl' summary

Function	Updating the constraint file for fast PVT corner.
Input	Original constraint file 'constraints.txt'
Output	Updated constraint file 'constraints_fc.txt'
Usage	*perl script1.pl*



(a) Graph for the original constraint file 'constraints.txt'



(b) Graph for the updated constraint file 'constraints_fc.txt'

Figure 14: Directed graphs depicting constraint files

The graphs provided in Figure 14 were printed by an open source software called 'GraphViz'. For easily handling debug issues, we automated this process by using a script named 'consToGraph.pl'. More information about the script and the graph generation process can be found in Table 4 and in Figure 15 respectively.

Table 4: 'consToGraph.pl' summary

Function	Conversion of the constraint file to a '.dot' file in Graphviz format
Input	Any file in constraint file format
Output	GraphViz Input '.dot' file
Usage	Generation of graphViz input by using *perl consToGraph.pl constraints_fc.txt constraints_fc.dot* Printing the graph file by using the *dot -Tpng constraints_fc.dot -o constraints_fc.png* commands.

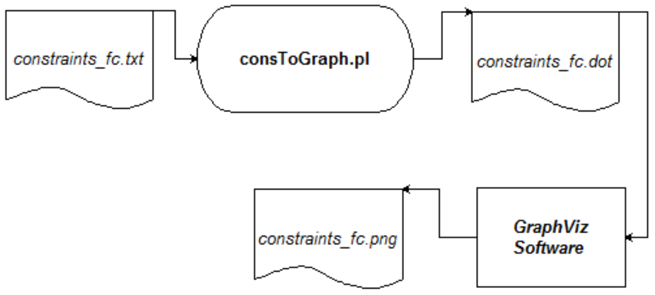


Figure 15: Obtaining directed graphs from the constraint file

3.2 Producing the Target File

The updated constraint file defines a safe zone for each clock phase. Providing these constraints directly to the CTS design tool may result in two potential problems given below.

- some of the clock phases may be out of bounds,
- performance of the overall circuit may suffer from the boundary clock phases.

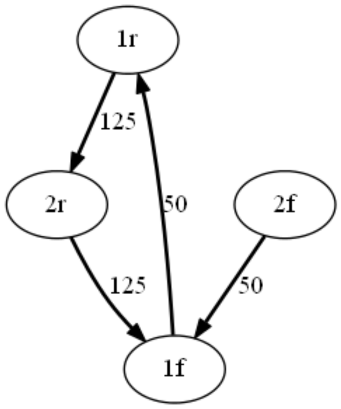
In order to overcome these potential problems, extra reaction time should be added to each constraint. The new constraints are stored in a target file named 'targets_fc.txt'.

Two different algorithms were implemented for target file generation. The first algorithm relies only on centering the safe zone, while the second algorithm tries maximizing minHiTime constraints for better performance.

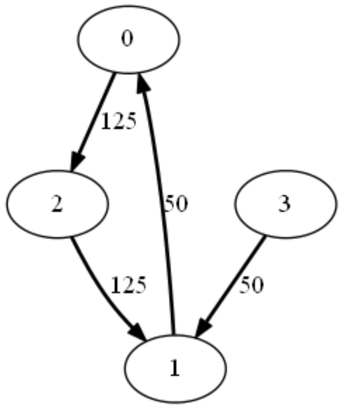
The Perl script 'target_noloop.pl', whose details can be seen in Table 5 calls the Python script 'cycles.py' to determine the loops in the updated constraint file. The script 'cycles.py' is an outside source that was obtained from a blog site created by a researcher named Josch Schauer [12]. Shauer states that the script was created by using Robert Tarjan's paper 'Enumeration of the Elementary Circuits of a Directed Graph' [13]. Author of the paper claims that redundant operations were eliminated by using 'backtracking with lookahead' method. The algorithm was updated more than once in time to overcome its limitations. However, since these limitations don't affect our work, the script based on the original algorithm was used.

'cycles.py' takes the number of nodes and the list of edges as input. Since the script is an outside source, required input data recovered from the updated constraint file should be interpreted in 'cycles.py' format. This requires representing each edge as a different natural number, and more so the numbers should increase arbitrarily. After obtaining the loops, the number values should be converted back to the original updated constraint file format. Both conversion operations are carried by the 'target_noloop.pl' script, however these operations are not shown in Figure 22 for simplicity. An example to conversion operation can be observed on Figure 16. The 'cycles.py' command for this example should be *"python cycles.py 4 0,2 2,1 3,1 1,0"*, and the output of such command would be the line *"0 2 1"*.

After running the script 'cycles.py' using the data from the updated constraint file for the test ADC design, the following loops marked with red edges shown in Figure



(a) Graph format for the constraint file



(b) Graph format for the script 'cycles.py'

Figure 16: Graph conversion for 'cycles.py'

17 are obtained.

After determining the loops present in the constraint file, target file is generated. Target file is obtained by adding extra reaction time (total slack) to the constraints, which results in increased performance. Two different algorithms are used to determine the slack. The script 'target_noloop.pl' chooses one of the algorithms by checking the variable '$slackMode'. Details of the algoritms are can be observed with an example according to the graph in Figure 17 below.

1. The length of each loop in the updated constraint file is calculated. The length of a loop is the sum of the edge weights on the loop. Length of each loop seen in Figure 17 is given below.

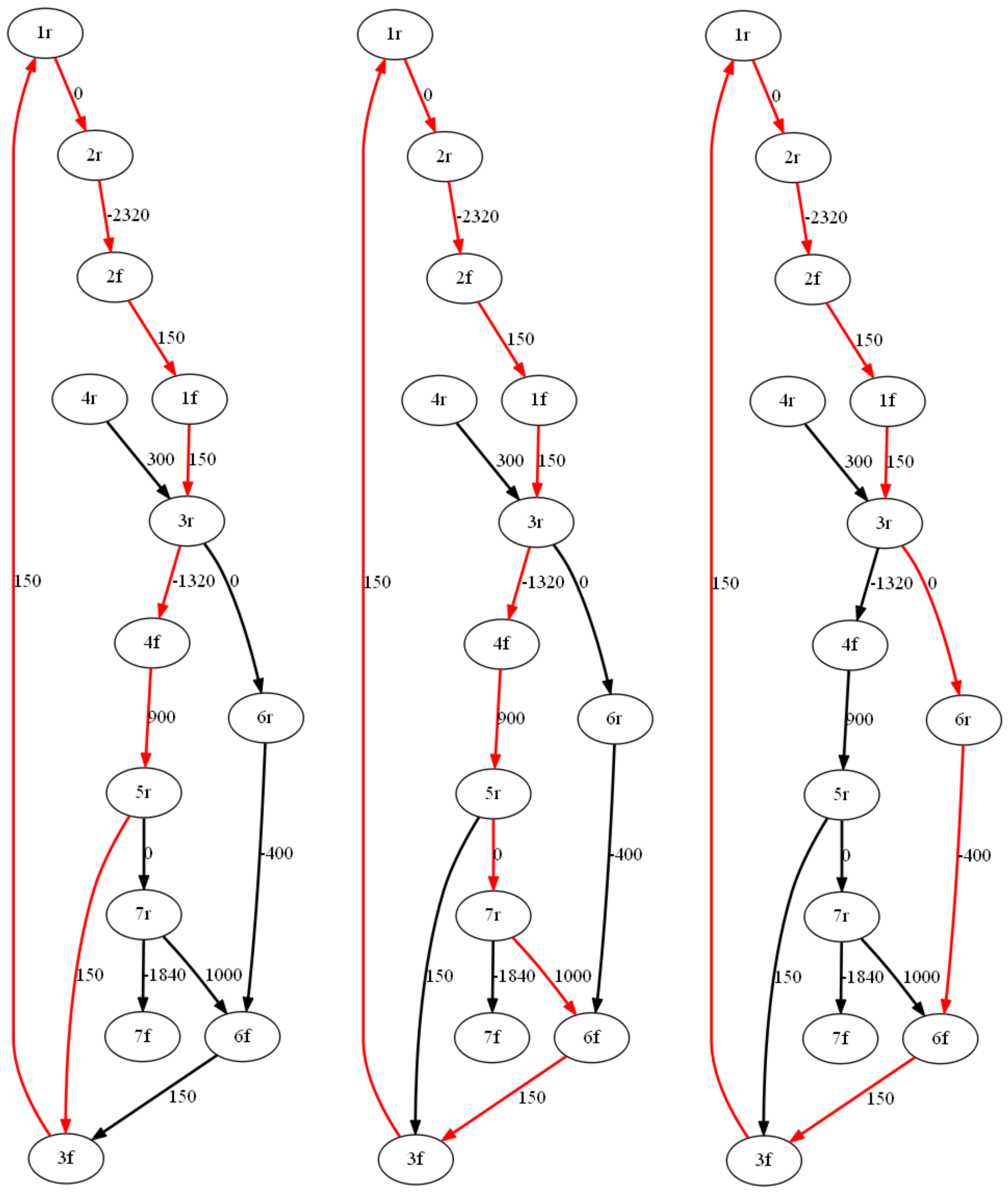
   loop length$_1$ = -2140

   loop length$_2$ = -1140

   loop length$_3$ = -2120

2. Slack for each loop is calculated. It is the average extra reaction time given for each edge in the loop calculated by the formula below.

   slack[j] = loop length[j] / # of vertices in the loop



(a) loop 1 (b) loop 2 (c) loop 3

Figure 17: Loops present in the updated constraint file

Calculated slack values for the loops shown in Figure 17 are given below.

slack_1 = -267.5

slack_2 = -114.0

slack_3 = -265.0

The two algorithms differ at this point. The first algorithm that sets the operating point at the center of the safe zone (active when $slackMode = 1) takes the maximum of these slack values (-114 for this example) and adds that value to each constraint in the updated constraint file. The resulting values are stored in the target file ('targets_fc.txt').

The second algorithm which keeps the minHiTime constraints as high as possible while trying to center the safe zone moves on to the next step.

3. The aim of this step is the calculation of the slack values for each minSepTime constraint. For each minSepTime constraint, the maximum of the constraint's eligible slack values is chosen. Eligible slack values are the slack values of the loops the constraint is in and the default slack value the user determines. Looking at the example in Figure 17, it can be observed that the edge '1r 2r 0' is a minSepTime edge that is inside all three loops. The slack value for this edge is calculated as below (default slack value is determined as -100 for this example).

   max(slack_1, slack_2, slack_3, -100)

   = max(-267.5, -114, -265, -100)

   = -100

4. Each slack value calculated above is subtracted from the total slacks it's part of to determine the remaining total slacks. The remaining total slacks for each loop in Figure 17 are calculated below.

$$\text{remaining total slack}_1 = -2140 - 7 \times (-100) = -1440$$

$$\text{remaining total slack}_2 = -1140 - 9 \times (-100) = -240$$

$$\text{remaining total slack}_3 = -2120 - 6 \times (-100) = -1520$$

5. At this step, slack values for each minHiTime constraint in a loop are calculated. Each remaining total slack value calculated at the previous step is divided to the number of minHiTime constraints present in that loop. Each quotient is a slack candidate for the minHiTime constraints present in the loop quotient belongs to. Then for each minHiTime constraint, the maximum of the candidates is chosen as slack. Slack for the Figure 17's minHiTime constraint '2r 2f -2320' which is present in all three loops is calculated as below.

   $$\text{average of remaining total slack}_1 = \text{remaining total slack}_1/1 = \text{-1440}$$

   $$\text{average of remaining total slack}_2 = \text{remaining total slack}_2/1 = \text{-240}$$

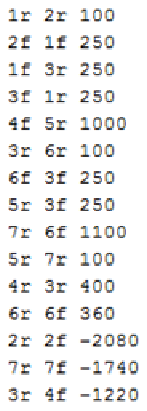
   $$\text{average of remaining total slack}_3 = \text{remaining total slack}_3/2 = \text{-760}$$

   Taking the values above into account, the slack for minHiTime constraint '2r 2f -2320' is calculated as:

   $$\max(\text{-1440, -240, -760}) = \text{-240}$$

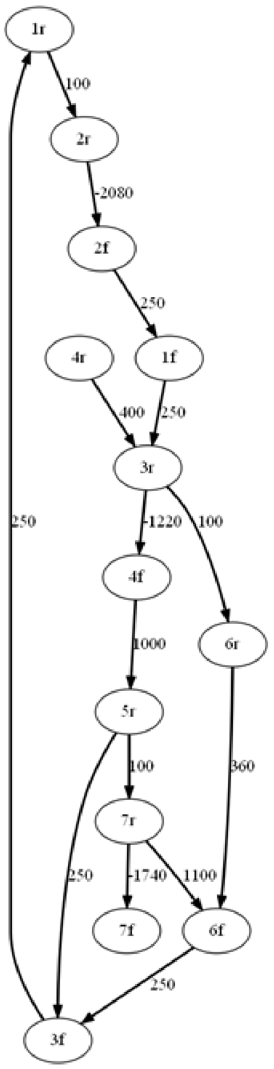
6. The steps above determine slack values for each loop constraint. Slack value for all constraints that don't belong to any loop is calculated as the maximum of the slacks of each loop and the standard slack. After appointing every slack value, target file is generated by adding the slacks to the updated constraint file. Resulting target file and its graph for our example can be seen at Figure 18.

Pseudocode describing target file generation steps given above can be seen in Figure 19.



```
1r 2r 100
2f 1f 250
1f 3r 250
3f 1r 250
4f 5r 1000
3r 6r 100
6f 3f 250
5r 3f 250
7r 6f 1100
5r 7r 100
4r 3r 400
6r 6f 360
2r 2f -2080
7r 7f -1740
3r 4f -1220
```

(a) Target file 'targets_fc.txt'



(b) Graph representing the target file

Figure 18: Target file and its graph representation

```
cSlack = -100

foreach cycle {
    foreach edge in a cycle
        totalSlack[cycle] += edgeDistance;
    avgSlack[cycle] = totalSlack[cycle]/#Edge[cycle];
}

if (slackMode == 0) {
    foreach cycle{
        foreach min-sep edge {
            if (slack[edge] = "")
                slack[edge] = cSlack;
            else if (slack[edge] < avgSlack[cycle])
                slack[edge] = avgSlack[cycle];
        }
    }

    foreach min-sep edge {
        foreach cycle
            if (min-sep edge is at cycle)
                usedSlack[cycle] += min-sep edge;
    }

    foreach cycle {
        totalSlack[cycle] -= usedSlack[cycle];
        avgSlack[cycle] = totalSlack[cycle]/#min-highEdge[cycle];
    }

    foreach cycle {
        foreach min-high edge {
            if(slack[edge] == "")
                slack[edge] = avgSlack[cycle];
            else if(slack[edge] < avgSlack[cycle])
                slack[edge] = avgSlack[cycle];
        }
    }

    foreach edge {
        if (slack[edge] == "")
            slack[edge] = cSlack;
    }
} else {
    foreach cycle {
        if (maxSlack == "")
            maxSlack = cSlack;
        else if (maxSlack < avgSlack[cycle])
            maxSlack = avgSlack[cycle];
    }

    foreach edge {
        slack[edge] = maxSlack;
    }
}
```

Figure 19: Pseudocode describing generation of the target constraints

3.3 Producing the Acyclic Target File

Target file generated above needs to be loopless for the scheduling phase. The loop clearing phase is handled by the script 'acycl.pl' and the resulting output is stored in the file 'noloop_fc.txt'. The loop clearing algorithm is based on 'Deep First Search (DFS)' method, whose steps will be given below.

The process starts with equating 'visited' and 'onPath' variables of each node to zero. The variable 'visited' determines whether the node is visited or not, whereas (onPath == 1) represents the edges that do not cause loops. There are also two pointers named 'parent' and 'child'. Parent is the node that current node comes from, while child is the node current node goes to. For an edge '1r 2f 300', '1r' is the parent of '2f', whereas '2f' is a child of '1r'. The algorithm goes through the steps given below for each node.

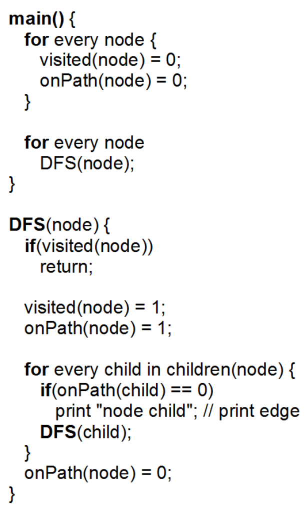
1. For the current node (parent) the algorithm inspects whether it was processed or not by checking the variable 'visited(parent)'. The algorithm returns to Step 1 with the next node if 'visited(parent)' equals 1, otherwise it continues on Step 2.

2. At this step variables 'visited(parent)' and 'onPath(parent)' are equated to 1.

3. Each 'children' of the parent is identified, and for each child variable 'onPath(child)' is checked. If 'onPath(child)' equals 0, parent and its child are both printed out. Otherwise they are not printed out, so that the loop is severed.

4. Here, a selected child node becomes the parent node and the algorithm returns back to Step 1 recursively. If no child node is left, then by assigning variable 'onPath(parent)' to 0, the algorithm climbs up the hierarchy and continues the process with the remaining child nodes.

Pseudocode for the script 'acyclic.pl' is given in Figure 20. Loopless target file 'noloop_fc.txt' generated by the algorithm above and its graph can be observed on Figure 21.



```
main() {
  for every node {
    visited(node) = 0;
    onPath(node) = 0;
  }

  for every node
    DFS(node);
}

DFS(node) {
  if(visited(node))
    return;

  visited(node) = 1;
  onPath(node) = 1;

  for every child in children(node) {
    if(onPath(child) == 0)
      print "node child"; // print edge
    DFS(child);
  }
  onPath(node) = 0;
}
```

Figure 20: Pseudocode for the script 'acycl.pl'

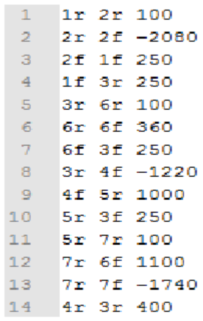
To automate target file and acyclic target file generation, a Perl script named 'target_noloop.pl' is used. The script whose flow is given in Figure 22 calls a Python script named 'cycles.py' and another Perl script named 'acyclic.pl' during operation. Details of these scripts can be observed at Tables 5, 6 and 7.

Table 5: 'target_noloop.pl' summary

Function	Generating loopless target file from the updated constraint file
Input	Updated constraint file 'constraints_fc.txt' Definition file 'defFile.txt'
Output	Loopless target file 'noloop_fc.txt'
Usage	*perl target_noloop.pl*

3.4 Forming the Timing Schedule

Up until this point, delay values were defined as a constraint. A constraint such as '1r 2f 300' can be translated as '1r < 2f -300'. However, for synthesis each clock edge



```
1r 2r 100
2r 2f -2080
2f 1f 250
1f 3r 250
3r 6r 100
6r 6f 360
6f 3f 250
3r 4f -1220
4f 5r 1000
5r 3f 250
5r 7r 100
7r 6f 1100
7r 7f -1740
4r 3r 400
```

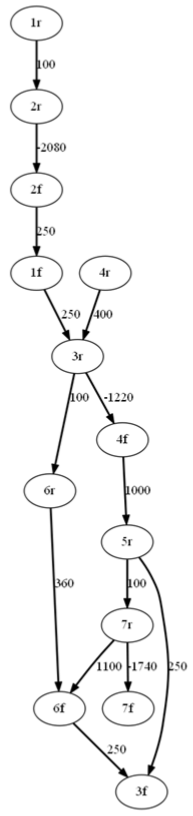


Figure 21: Loopless target file 'noloop_fc.txt' and its graph

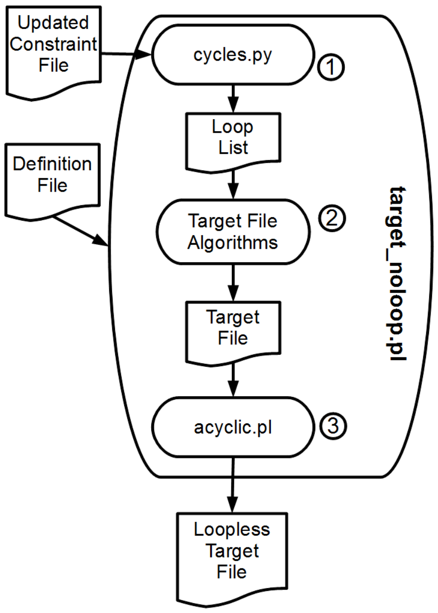


Figure 22: Workflow for the script 'target_noloop.pl'

Table 6: 'cycles.py' summary

Function	Identifying and storing the loops present in the constraint file
Input	Number of vertices and the list of edges present in the updated constraint file
Output	Loop list file 'pyFile'
Usage	*python cycles.py <number of corners> <vertex list(v1, v2 v3,v7 ...)>*

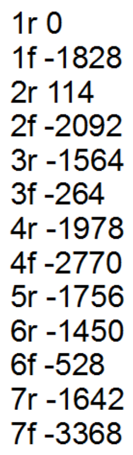
Table 7: 'acycl.pl' summary

Function	Clearing the target file off loops
Input	Target file 'targets_fc.txt'
Output	Loopless target file 'noloop_fc.txt'
Usage	*perl acycl.pl*

should target an exact time value rather than a constraint such as '1r 1200' that is '1r = 1200' when translated. Here, the edge '1r' aims an exact delay of 1200 ps. According to the constraint '1r 2f 300', 2f will aim for a delay greater than 1500 ps (1200ps + 300 ps) which can subsequently change depending on the other constraints.

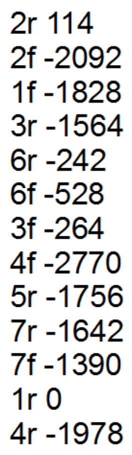
Delay values for both edges of each clock phase is calculated here by using the loopless target file 'noloop_fc.txt' generated above. Scheduled clock edges are stored in the file 'schedule_fc.txt'. While generating the clock tree, two different algorithms named 'sch_hfu' and 'sch_gg' were used. 'sch_gg' was used in development phase, therefore it doesn't yield optimal solutions. The second algorithm, 'sch_hfu' was proposed later during development. It gives the optimum results, moreover timing wise it is more efficient. Selection between two algorithms is done by checking the '$schFlag' variable. The resulting timing schedules for both algorithms are given in Figure 23. The values observed below are the target delay timings for each edge in CTi synthesis.

'sch_hfu' is a graph algorithm that uses Depth-First Search (DFS)method for scheduling. The purpose of scheduling is to determine each node's position depending



```
1r 0
1f -1828
2r 114
2f -2092
3r -1564
3f -264
4r -1978
4f -2770
5r -1756
6r -1450
6f -528
7r -1642
7f -3368
```

(a) Schedule file generated using 'sch_hfu'



```
2r 114
2f -2092
1f -1828
3r -1564
6r -242
6f -528
3f -264
4f -2770
5r -1756
7r -1642
7f -1390
1r 0
4r -1978
```

(b) Schedule file generated using 'sch_gg'

Figure 23: Generated schedule files

on the loopless target file. For positioning, a reference node should be chosen and its value (delay) should be assigned as zero. To determine the values of other nodes, the following two steps should be repeated till every node value on the graph has been determined.

1. Starting the search from the identified nodes and on every direction, each childless node (root node) is determined using DFS. This step is called 'Discovery' or 'D'.

2. From each discovered node above, an assigned node should be searched using DFS. After reaching an assigned node, search should turn back to the source and assign values to each node on the way. This step is called 'Scheduling' or 'S'.

The example given below will step by step explain the algorithm briefly described above. In the example, red circles marked by an 'S' are the scheduled nodes, while the blue ones marked by a 'D' are the discovered nodes. A discovered node is a node that can be scheduled with the aid of the other scheduled nodes. Each node given in the example represents a clock edge, and each arrow between nodes represents a constraint defined in the loopless target file. Starting graph of the example can be observed on Figure 24.

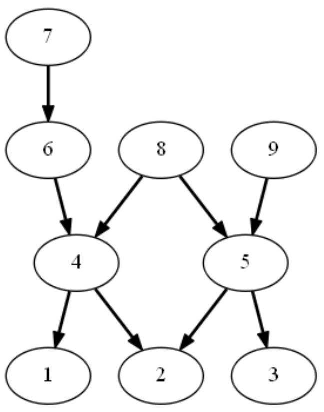


Figure 24: Initial unscheduled graph

1. Node 4 is chosen as the reference point and identified as scheduled as seen in Figure 25. The scheduled node list is updated as {4}.

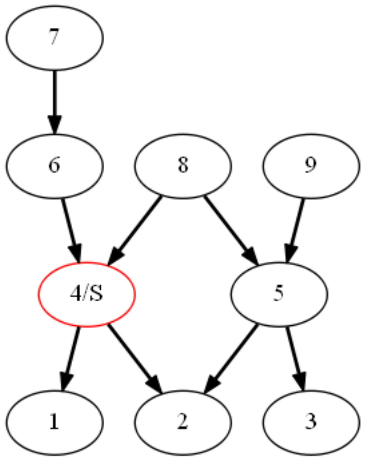


Figure 25: scheduled node list: {4}

2. Starting from the only scheduled node 4, root nodes on both directions are identified as discovered.Going along the arrows, nodes 1 and 2 are discovered. Proceeding in the opposite direction, nodes 7 and 8 are identified as discovered.

Scheduled node list stays the same, discovered node list is updated as {1, 2, 7, 8}. The graph will now look like in Figure 26.

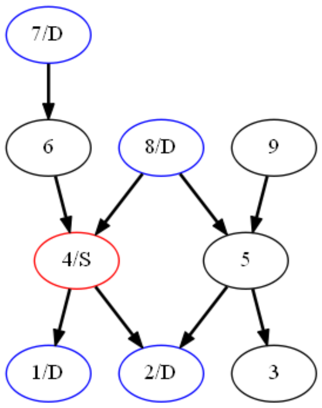


Figure 26: scheduled node list: {4}
discovered node list: {1, 2, 7, 8}

3. Starting from the discovered node list, node 1 is selected for scheduling. Searching a scheduled node gives result immediately as node 1 is directly connected to the initially scheduled node 4. So node 1 is scheduled according to the relation between node 1 and node 4 as observed from Figure 27. Node discovery from the newly scheduled node 1 yields no results. Scheduled node list is updated as {4, 1}, discovered node list is updated as {~~1~~, 2, 7, 8}.

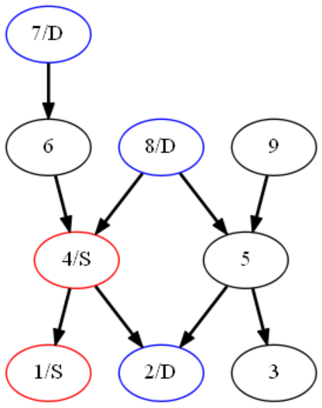


Figure 27: scheduled node list: {4, 1}
discovered node list: {2, 7, 8}

4. Going through the next node in the discovered node list, node 2 is selected for scheduling. Searching a scheduled node only gives node 4 like the previous step same as before. Using the scheduled node 2, node 9 is discovered. Scheduled node list is updated as {4, 1, 2}, discovered node list is updated as {~~1~~, ~~2~~, 7, 8, 9}. The resulting graph can be seen in Figure 28.

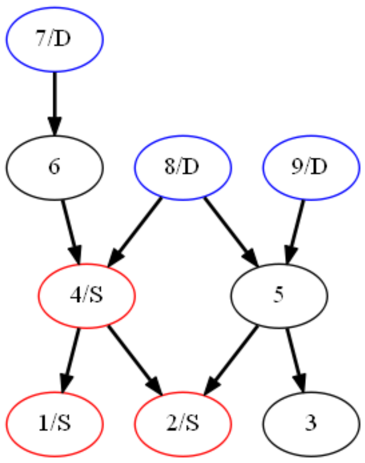
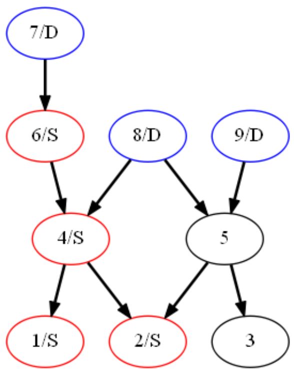


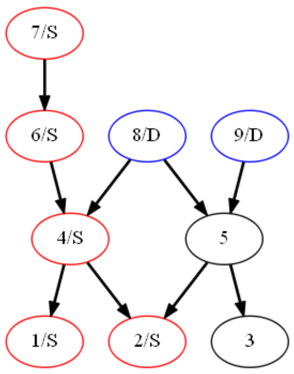
Figure 28: scheduled node list: {4, 1, 2}
discovered node list: {7, 8, 9}

5. Moving on the discovered node list, node 7 is chosen for scheduling. Search finds node 4 as a scheduled node via node 6. After finding the scheduled node, scheduling starts in the opposite direction. Using node 4, node 6 is scheduled as seen in Figure 29(a). After that, node 7 is scheduled according to the value of node 6 and the constraint between node 6 and 7. Discovery search from both nodes 6 and 7 yields no results. After this step, scheduled node list is updated as {4, 1, 2, 6, 7}, discovered node list is updated as {~~1~~, ~~2~~, ~~7~~, 8, 9}. The resulting graph can be observed in Figure 29(b).

6. Next node in the discovery list is node 8. To schedule node 8, the scheduled nodes 2 and 4 are found. Since node 8 should satisfy both branches, both



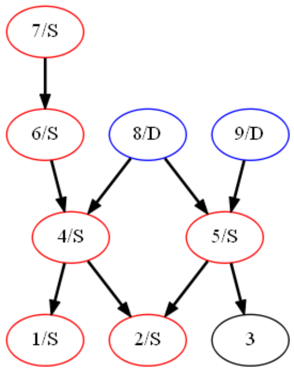
(a) Scheduling node 6.



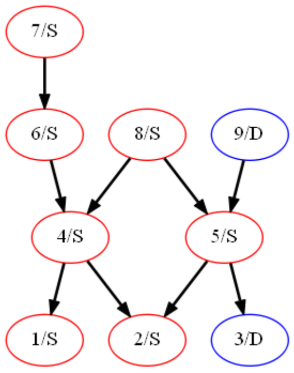
(b) Scheduling node 7.

Figure 29: scheduled node list: {4, 1, 2, 6, 7}
discovered node list: {8, 9}

branches are explored. Returning back to node 8 from node 2, node 5 is scheduled as seen in Figure 30(a). A candidate value for node 8 is found by returning further back to node 8 from node 5. Another candidate value is found returning back to node 8 from node 4. The value that satisfies both relations is chosen as schedule value for node 8. Using the newly scheduled node 8, node 3 is discovered. Scheduled node list is updated as {4, 1, 2, 6, 7, 5, 8}, discovered node list is updated as {~~1~~, ~~2~~, ~~7~~, ~~8~~, 9, 3}. The resulting graph can be observed in Figure 30(b).



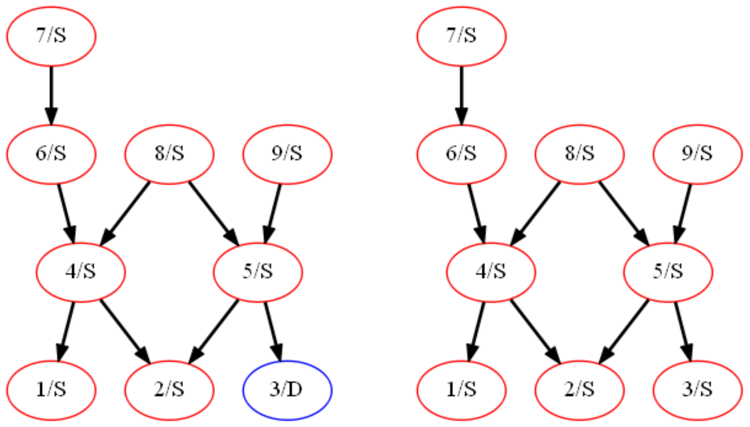
(a) Scheduling node 5.



(b) Scheduling node 8.

Figure 30: scheduled node list: {4, 1, 2, 6, 7, 5, 8}
discovered node list: {9, 3}

7. Moving onto the remaining nodes in the node list nodes 3 and 9 are scheduled according to the value of node 5. After scheduling these last two nodes, no unscheduled node will be left. Finally, all nodes are scheduled as observed in Figure 40(c).



(a) Scheduling node 9. (b) Scheduling node 3.

Figure 31: Scheduling the final nodes

The algorithm explained above is implemented with a C program called 'schedule', whose details can be observed from Table 8. Pseudocode for the algorithm is given in Figure 32

Table 8: 'schedule.pl' summary

Function	Creating a schedule file that contains timings for every clock phase using the loopless target file.
Input	Loopless target file (noloop_fc.txt)
Output	Schedule file (schedule_fc.txt)
Usage	*'./schedule <# of nodes> <# of edges> noloop_fc.txt'*

3.5 *Verifying the Timing Schedule*

The timing schedule (schedule_fc.txt) created in the previous section is confirmed after checking the values in the constraint file (constraints.txt). The confirmation is

```
main() {
 visitedFlag = 0;
 for every node {
  visited(node) = 0;
  time(node) = UNKNOWN;
 }

 rootList = ((node0, +1), (node0, -1));
 time(node0) = 0;

 while(rootList != EMPTY) {
  visitedFlag++;
  (node, dir) = removeOneFromList(rootList);
  DFS(node, dir);
 }
}

DFS(node, dir) {
 if(visited(node) == visitedFlag)
  return;
 if((time(node) != UNKNOWN) && (node != node0) )
  return;

 visited(node) = visitedFlag;
 children = children(node, dir);
 if(children == EMPTY)
  add node to rootList with -dir;

 first = 1;
 for every child in children {
  DFS(child, dir);

  if(time(child) == UNKNOWN)
   next;

  if(first) {
   time(node) = time(child)-dir*edge(node,child);
   first = 0;
  } else
   tmp = time(child)-dir*edge(node,child);
   if(dir*tmp < dir*time(node))
    time(node) = tmp;
  }
}
```

Figure 32: Pseudocode for the scheduling algorithm

done with a Perl script named 'schedule_chk.pl', whose details can be found below in Table 9.

Table 9: 'schedulechk.pl' summary

Function	Verification of the schedule file
Input	Schedule file (schedule_fc.txt) Constraint file (constraints.txt)
Output	The script outputs true or false for each constraint on the console
Usage	*perl scheduleChk.pl*

The script uses the scheduled values for each clock edge on the constraint file to do the confirmation. Suppose there is a constraint '1r 2r 300' and the scheduled values for these two clock edges are '1r 500' and '2r 100' in the schedule file. Script does the check '(1r-2r) <? 300', which gives '(500-100) $\not<$ 300' and prints 'FALSE' on the console.

3.6 Producing SPICE Files for the Scheduled Clocks

The schedule file contains target values for each clock phase that will be given to the CTS tool. Before going through synthesis, it is beneficial to check these target values on the analog circuit. SPICE files for fast(fc), slow(sc) and typical(tc) PVT corners should be generated for simulation. Schedule file generation for each corner is done with the script 'fast2corners.pl', and the SPICE file generation is handled with the script 'sch2spice.pl'. 'fast2corners.pl' whose details can be seen on Table 10, scales the schedule file for typical and slow corners using the coefficients obtained during characterization. 'sch2spice.pl' whose details are given in Table 11, generates the SPICE code from the schedule files.

Table 10: 'fast2corners.pl' summary

Function	Scaling the schedule file for slow and normal PVT corners
Input	Schedule file (schedule_fc.txt)
Output	Schedule file for slow PVT corner (schedule_sc.txt) Schedule file for normal PVT corner (schedule_tc.txt)
Usage	*perl fast2corners.pl*

Table 11: 'sch2spice.pl' summary

Function	Generating the SPICE code from schedule files
Input	Schedule files (schedule_fc.txt, schedule_tc.txt, schedule_sc.txt) Definition file (defFile.txt)
Output	SPICE files (targets_fc.txt, targets_tc.txt, targets_sc.txt)
Usage	*perl sch2spice.pl schedule_fc.txt defFile.txt targets_fc.cir*

CHAPTER IV

DESIGN AND SYNTHESIS

Target clock delays were calculated and stored in the file 'schedule.txt' in the previous chapter. The next step is to synthesize a circuit that generates and distributes clock signals according to this file. It was repeatedly stated in the previous chapters that clock tree generation and distribution are seperate tasks and they are handled by subcircuits CTi and CTe respectively. CTi and CTe generation will be explained step by step throughout the chapter, following through Figure 33.

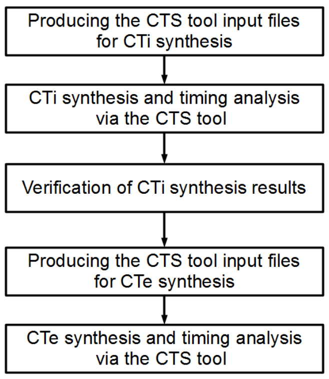

Producing the CTS tool input files for CTi synthesis
CTi synthesis and timing analysis via the CTS tool
Verification of CTi synthesis results
Producing the CTS tool input files for CTe synthesis
CTe synthesis and timing analysis via the CTS tool

Figure 33: Design and Synthesis flowchart

4.1 Proposed Clock Tree Architecture

As discussed in Chapter 1 before, clock tree should generate the required clock phases like in Figure 3(a) and distribute these phases without further shift among phases like in Figure 3(b). Since it will be easier to handle these two tasks separately, the architecture is divided into two parts. Clock tree generation part is named as Clock Tree intrinsic (CTi), whereas distribution of the generated clocks is called Clock Tree extrinsic (CTe). CTi takes the main clock signal and produces required clock phases, then CTe distributes the generated phases without changing the relation between them. CTe even provides the reverse of a generated clock phase if required.

In Figure 34, a diagram that depicts the roles of CTi and CTe can be observed. The triangles in the figure represent clock trees while the small circles represent inverters. Clock trees present in the figure consist of buffer and inverter logic gates existing in our design library. Also in the same figure it can be observed that the clock tree of CTi is not horizontally aligned like the ones in CTe. This is because in CTi, the aim is to generate clocks with their individual expected delays, so every clock is delayed according to their timings. The main concern in CTe is to protect the relation that was achieved in CTi. In order to do that, all clocks generated in CTi should be sent to their respective analog pins with the same delay, resulting in horizontally aligned triangles.

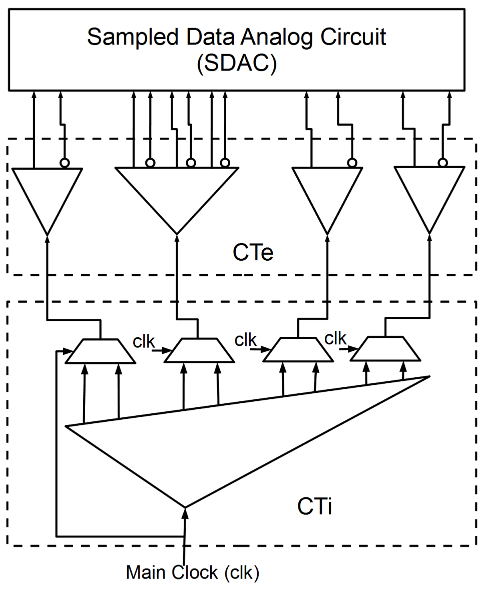


Figure 34: CTi and CTe forming the clock tree for the SDAC circuit

Detailed info on CTi and CTe architectures will be given on subsections 4.1.1 and 4.1.2 respectively.

4.1.1 Clock Tree intrinsic (CTi)

Since CTi handles the timing relations between clock signals, it is the more crucial part of our two part circuit. It is the relation between the clock signals that enables the SDAC to perform as intended.

CTi circuit requires a 2x1 multiplexer and numerous buffers to generate each clock phase. Clock phases are generated by feeding two delayed phases of the main clock to a multiplexer. For example, to generate a clock signal with period T_{clk} and duty cycle ($T_{clk}/2 + \Delta t_A - \Delta t_B$), two phases of the main clock (clk_A and clk_B) with delays Δt_A and Δt_B should be generated initially. Inputting these two phases (clk_A and clk_B) to a multiplexer, a clock signal with the above mentioned period and duty cycle values can be generated. Using the main clock signal as select input, the multiplexer will output clk_A when the main clock is 0, and clk_B otherwise. The circuit depicting this example can be seen in Figure 35.

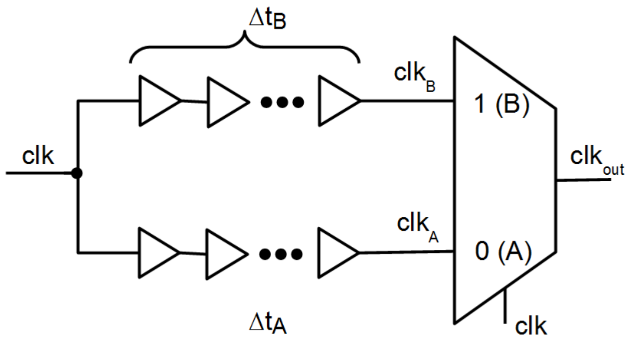


Figure 35: Clock derivation from the main clock using a multiplexer

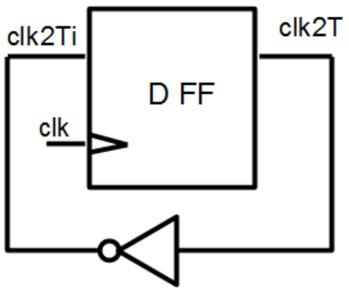
Each generated clock drives a switch in SDACs. These switches actually consist of an NMOS and/or a PMOS transistor. That's why each generated clock has a negative pair. These positive and negative clocks will be named with "_clk" and "_xclk" suffixes respectively.

The reason we generate a regular or an inverted clock can be explained on Figure

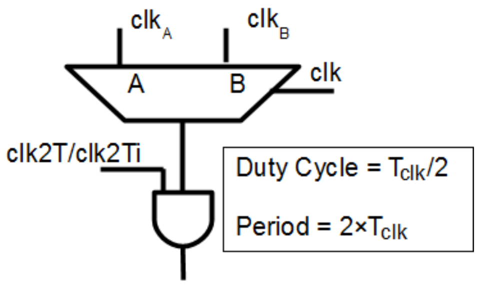
35 as follows: rising time of $\mathrm{clk_A}$ and falling time of $\mathrm{clk_A}$ that drive the multiplexer are different edges, and $\mathrm{clk_B}$ signal has the same issues. Moreover, rising time of $\mathrm{clk_A}$ and falling time of $\mathrm{clk_B}$ have the same edge reference problem. Reverse case is also valid. Considering these, we should declare all relations according to either rising or falling edge of the main clock. For example, if we were to declare first clock phase's falling edge to be earlier than the second phase's rising edge (1f $\rightarrow$ 2r) and if we generated regular clock for both of them, we would have to relate 1f with falling edge of the main clock, whereas 2r with rising edge of the main clock. To avoid confusion, we can generate the complement of first phase and regular of the second. In our project, we generated regular clocks (_clk) if the generated clock is high with the main clock, and complementary clocks (_xclk) if the generated clock is high when the main clock is low.

Another important issue that should be addressed here is the exceptional case of generating $2\mathrm{T}_{clk}$ period clock phases. Generally SDACs require phases with $\mathrm{T}_{clk}/2$ duty cycle and T_{clk} period, however in some circumstances phases with $2\mathrm{T}_{clk}$ period may also be required. Even duty cycles might differ for some phases. In our test design we used $2\mathrm{T}_{clk}$ period clocks with duty cycles T_{clk} and $\mathrm{T}_{clk}/2$. In order to generate phases with these periods and duty cycles, the circuit should generate new main clocks as shown in Figure 36.

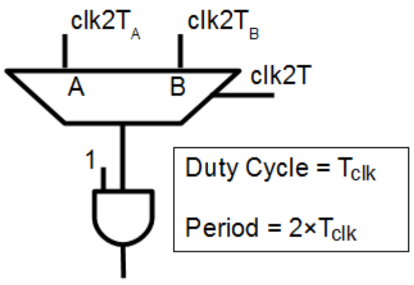
To be able to generate clocks with $2\mathrm{T}_{clk}$ period, we should first generate a reference clock with $2\mathrm{T}_{clk}$ period. We are using a one-bit counter shown in Figure 36(a) to generate the reference clock. Generating a clock with $2\mathrm{T}_{clk}$ period and $\mathrm{T}_{clk}/2$ duty cycle requires driving the multiplexer's data and select inputs with the main T_{clk} period $\mathrm{T}_{clk}/2$ duty cycle clock, and connecting the output and $2\mathrm{T}_{clk}$ reference clock to an AND gate like shown on Figure 36(b). A clock with $2\mathrm{T}_{clk}$ period and T_{clk} duty cycle can be generated by driving a multiplexer's inputs and select with a $2\mathrm{T}_{clk}$ period reference clock, and ANDing the output with logic HIGH (Figure 36(c). Clock



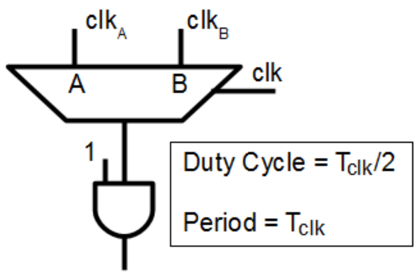
(a) Main clock with 2T$_{clk}$ period



(b) Clock with T$_{clk}$/2 duty cycle and 2T$_{clk}$ period



(c) Clock with T$_{clk}$ duty cycle and 2T$_{clk}$ period



(d) Clock with T$_{clk}$/2 duty cycle and T$_{clk}$ period

Figure 36: Modified CTi circuit

with T_{clk} period and $T_{clk}/2$ duty cycle can be generated in similar way, except we feed the multiplexer inputs and select with the main T_{clk} period clock like on Figure 36(d). The AND gates in Figures 36(c) and 36(d) may seem redundant but they are actually required to apply the AND gate delay to every clock phase.

To sum all up, the rising and falling edge of each clock phase are generated separately, and each phase is generated by giving required delays via buffers. After generating the edge signals, each phase is generated by selecting the required clock edge via a multiplexer as shown in Figure 35.

4.1.2 Clock Tree extrinsic (CTe)

The role of CTe is to deliver the clock phases generated by CTi to the analog circuit without changing the relation between phases. Since SDAC is treated as a 'hard macro', it can't be interfered with, such as inserting a metal line within SDAC boundaries. But since the location of each connection on SDAC boundaries is known, clock signals can be delivered just outside the SDAC boundaries, ensuring metal on both sides connect to each other.

Each SDAC clock pin has a capacitance value of its own. Moreover, delay between each clock pin and the switch associated with the pin is different. Thus since clock trees inside CTe rectify these effects, they act like an impedance transformer. Figure 37 illustrates CTe behavior.

In CTe we generate a clock tree for each CTi output, and every clock tree distributes its clock and the clock's complement to the switches on analog macro as seen in Figure 34. Each clock tree in CTe is very much like the ones in synchronous digital systems, that is each clock tree tries to distribute the CTi generated clock signal to the related switches with the smallest skew possible. One other goal of CTe is to equalize the insertion delay (from source to destination) of each clock signal like in Figure 37. In this example the software tries to assert "$ID_1 = ID_2 = ... = ID_n$" so

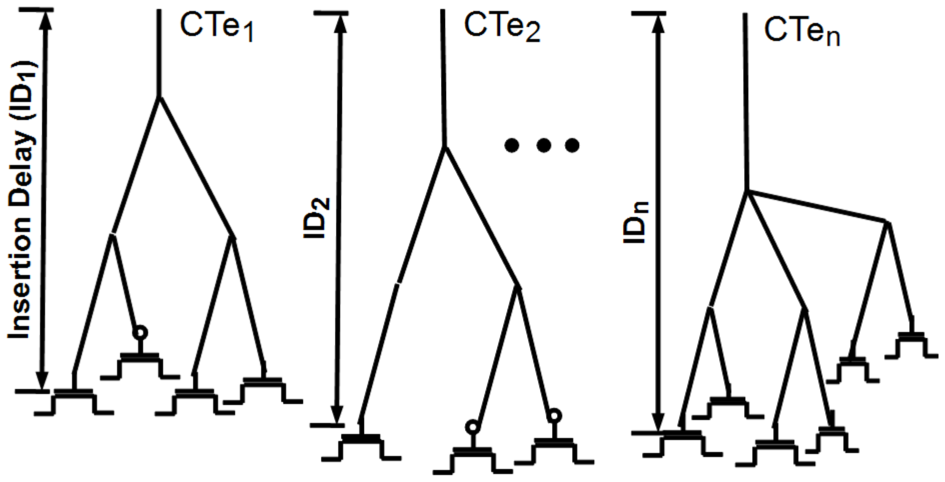


Figure 37: CTe behavior

that the clock relations achieved in CTi are preserved.

4.2 Producing the CTS Tool Input Files for CTi Synthesis and Analysis

The schedule file generated in the previous sections contains the target delays for both edges of each clock phase. However, the schedule file needs to be interpreted to CTS software for synthesis and analysis. The files that are generated for the CTS software and the scripts that generate the files are given below.

- *cti.cmd:* The file should be created in Tcl scripting language. It contains the necessary instructions that are given to the Cadence Encounter tool for clock synthesis. It also determines the information that will be saved after synthesis. The Perl script 'createCTiCmd.pl' (Table 12) generates this file automatically.

- *cti.conf:* This file contains the commands that set the Verilog files, library files and LEF files that will be used by Encounter. It is generated automatically by the script 'createCTiConf.pl' (Table 13).

Table 12: 'createCTiCmd.pl' summary

Function	Generating 'cti.cmd' Tcl script file
Input	Definition file (defFile.txt)
Output	'cti.cmd' Tcl script file
Usage	*perl createCTiCmd.pl defFile.txt*

Table 13: 'createCTiConf.pl' summary

Function	Generating 'cti.conf' command file
Input	–
Output	'cti.conf' file
Usage	*perl createCTiConf.pl*

- *cti.v:* 'cti.v' is a Verilog file that describes the logical structure of the CTi design. It is generated automatically by the script 'createCTiVerilog.pl' (Table 14).

Table 14: 'createCTiVerilog.pl' summary

Function	Generating 'cti.v' Verilog file
Input	Definition file (defFile.txt)
Output	'cti.v' file
Usage	*perl createCTiVerilog.pl defFile.txt*

- *umc18.v:* This file contains Verilog codes for the circuits in the standard design library. Analog designer should provide this file in the folder CTi synthesis takes place in.

- *umc.lib:* Contains characterization information (such as timing and power consumption) for the standard design library. For clock tree synthesis, lib file for the fast PVT corner will be used. Analog designer should provide this file in the folder CTi synthesis takes place in.

- *umc.lef:* Contains node geometries for the nodes present in the standard design library. Analog designer should provide this file in the folder CTi synthesis takes place in.

- *cti.ctstch:* 'clock tree specification' file contains the necessary definitions and constraints required for clock tree synthesis. Cadence Encounter synthesizes the clock tree according to the contents of this file. It is generated by the script 'ctstchgen2.pl' (Table 15).

Table 15: 'ctstchGen2.pl' summary

Function	Generating 'cti.ctstch' file
Input	Definition file (defFile.txt) Schedule file (schedule_fc.txt)
Output	'cti.ctstch' file
Usage	*perl ctstchGen2.pl defFile.txt schedule_fc.txt*

'Clock tree specification file' is the most important file above, because it interprets the schedule file to the CTS tool. The details of this interpretation is given below.

A CTS tool aims to deliver the main signal (root node) to every recipient node at the same time. Each node in the clock tree has different capacitance and internal delay values. Suppose that there is a root node called n0, and there are 3 recipient nodes n1, n2 and n3 with internal delays 200ps, 300ps and 400ps respectively. The CTS tool will try to satisfy the following equation.

$\Delta t_1 + d_1 = \Delta t_2 + d_2 = \Delta t_3 + d_3$

Using the equation above, the latencies CTS tool will synthesize between u_0-u_1 (Δt_1), u_0-u_2 (Δt_2) and u_0-u_3 (Δt_3) will be determined as below.

$\Delta t_1 = \max(d_1, d_2, d_3) - d_1 = 200,$

$\Delta t_2 = \max(d_1, d_2, d_3) - d_2 = 100,$

$\Delta t_3 = \max(d_1, d_2, d_3) - d_3 = 0.$

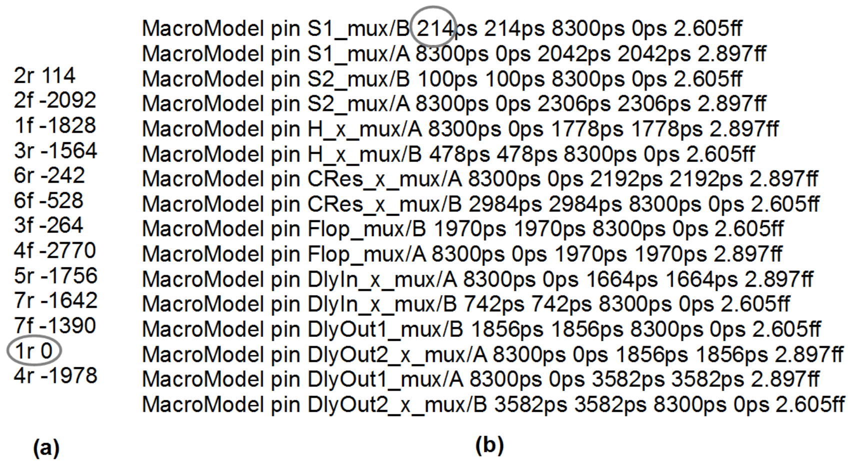
The CTS tool tries to deliver each signal at the same time by supplying each signal with different delay lengths as observed above. For CTi synthesis, this feature is used along with multiplexing. The schedule file contains the Δt_i values like above. However, CTS tool only accepts the d_i values as inputs. So, the equations above should be modified in the format below.

$d_i = \max(\Delta t_0, \Delta t_1, ..., \Delta t_n) + \varepsilon - \Delta t_i$

The values in the schedule file are converted using the formula above in the 'clock tree specification' file. The tolerance delay (ε) is provided to ensure that every delay is bigger than zero, because CTe insertion delays are bigger than zero. Figure 38 shows the example schedule file and its corresponding clock tree specification file. Using the formula above for the schedule file line '1r 0' (assuming ε = 100ps),

$d_{1r} = 114 + 100 - 0 = 214,$

which is enclosed in Figure 38 is obtained.



```
2r 114
2f -2092
1f -1828
3r -1564
6r -242
6f -528
3f -264
4f -2770
5r -1756
7r -1642
7f -1390
1r 0
4r -1978
```

(a)

```
MacroModel pin S1_mux/B 214ps 214ps 8300ps 0ps 2.605ff
MacroModel pin S1_mux/A 8300ps 0ps 2042ps 2042ps 2.897ff
MacroModel pin S2_mux/B 100ps 100ps 8300ps 0ps 2.605ff
MacroModel pin S2_mux/A 8300ps 0ps 2306ps 2306ps 2.897ff
MacroModel pin H_x_mux/A 8300ps 0ps 1778ps 1778ps 2.897ff
MacroModel pin H_x_mux/B 478ps 478ps 8300ps 0ps 2.605ff
MacroModel pin CRes_x_mux/A 8300ps 0ps 2192ps 2192ps 2.897ff
MacroModel pin CRes_x_mux/B 2984ps 2984ps 8300ps 0ps 2.605ff
MacroModel pin Flop_mux/B 1970ps 1970ps 8300ps 0ps 2.605ff
MacroModel pin Flop_mux/A 8300ps 0ps 1970ps 1970ps 2.897ff
MacroModel pin DlyIn_x_mux/A 8300ps 0ps 1664ps 1664ps 2.897ff
MacroModel pin DlyIn_x_mux/B 742ps 742ps 8300ps 0ps 2.605ff
MacroModel pin DlyOut1_mux/B 1856ps 1856ps 8300ps 0ps 2.605ff
MacroModel pin DlyOut2_x_mux/A 8300ps 0ps 1856ps 1856ps 2.897ff
MacroModel pin DlyOut1_mux/A 8300ps 0ps 3582ps 3582ps 2.897ff
MacroModel pin DlyOut2_x_mux/B 3582ps 3582ps 8300ps 0ps 2.605ff
```

(b)

Figure 38: Schedule file and the corresponding clock tree specification file

In addition to the synthesis related files explained above, a command file that analyses and records the synthesis results should also be generated. This is automatically handled by a Perl script called 'createCTiCommands' whose details can be found on Table 16.

Table 16: 'createCTiCommands.pl' summary

Function	Generating the commands that handle CTi analysis
Input	Definition file (defFile.txt)
Output	'cti_report_timing_commands' command file
Usage	*perl createCTiCommands.pl defFile.txt*

4.3 CTi Synthesis and Timing Analysis via the CTS Tool

At this step CTi synthesis and analysis is done with Cadence Encounter software. The cti generation top script (topScript_cti.pl) should initialize the startup for Cadence Encounter in order to fully automate CTi synthesis and analysis. Although the software also has visual interface, console mode was used for automation. The software is run with the command *system("/ECE/Cadence/IC/EDI_9.12/bin/encounter -nowin -init cti_top.tcl > encounterout.txt")* in the top script. The arguments are explained below.

- *-nowin:* This argument determines that the program will be run on the console.
- *-init cti_top.tcl:* This argument will run the program with script 'cti_top.tcl'. The script uses 2 command files that are:
  - *cti.cmd* contains the necessary instructions that are given to the Cadence Encounter tool for clock synthesis. It also determines the information that will be saved after synthesis.
  - *cti_report_timing_commands* contains the 'report timing' commands for analysis. It also saves timing results for all clock phases to each PVT edge file (cti_results_fc.txt, cti_results_tc.txt, cti_results_sc.txt).

- > *encounterout.txt:* This argument prints the Encounter outputs into file 'encounterout.txt' instead of displaying them on console.

Layout generated from CTi synthesis can be observed in Figure 39.

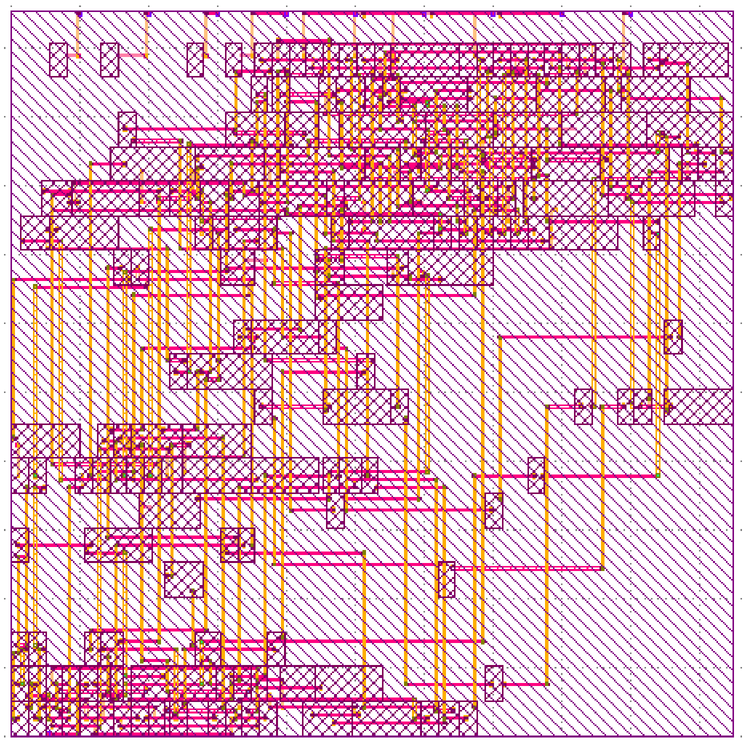


Figure 39: Layout of CTi circuit

4.4 Verification of CTi Synthesis Results

The verification is handled by the scripts 'b1.pl' and 'b2.pl'. The script 'b1.pl' skims through CTi synthesis files and grabs rising delay, rising slew rate, falling delay and falling slew rate for each clock phase and stores these values in the file 'b1Out.txt' with the following format.

<Clock Edge> <rising delay> <rising skew> <falling delay> <falling skew>

The script 'b2.pl' uses the values in the file 'b1Out.txt' on the constraint file, and writes the results for each PVT edge in the file 'b2Out.txt'. Details of both scripts are given in Tables 17 and 18.

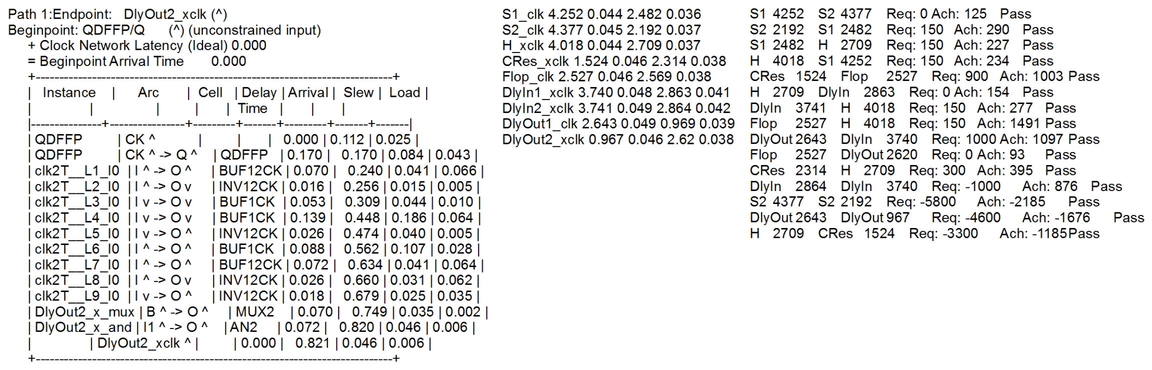
Table 17: 'b1.pl' summary

Function	Obtaining rising and falling delay and slew rate values for each clock edge from CTi analysis results
Input	Encounter result files (cti_results_fc.txt, cti_results_sc.txt, cti_result_tc.txt)
Output	PVT analysis result files (b1Out_fc.txt, b1Out_sc.txt, b1Out_tc.txt)
Usage	*perl b1.pl*

Table 18: 'b2.pl' summary

Function	Verification of the synthesis results using the constraint file
Input	PVT analysis result files (b1Out_fc.txt, b1Out_sc.txt, b1Out_tc.txt)
Output	Files (b2Out_fc.txt, b2Out_sc.txt, b2Out_tc.txt) Each line in these files contains 'Pass' or 'Fail' determining whether the constraints are met or not
Usage	*perl b2.pl*

With the verification of CTi synthesis results complete, CTi design phase is finished and CTe synthesis can start. A verification output example can be seen in Figure 40.



```
Path 1:Endpoint:   DlyOut2_xclk (^)
Beginpoint: QDFFP/Q      (^) (unconstrained input)
    + Clock Network Latency (Ideal) 0.000
    = Beginpoint Arrival Time       0.000
    +---------------------------------------------------------------------+
    |  Instance   |     Arc      | Cell   | Delay | Arrival |  Slew |  Load |
    |             |              |        |       |  Time   |       |       |
    |-------------+--------------+--------+-------+---------+-------+-------|
    | QDFFP        | CK ^          |        |       |  0.000 | 0.112 | 0.025 |
    | QDFFP        | CK ^ -> Q ^   | QDFFP   | 0.170 |   0.170 | 0.084 | 0.043 |
    | clk2T__L1_I0  | I ^ -> O ^    | BUF12CK | 0.070 |   0.240 | 0.041 | 0.066 |
    | clk2T__L2_I0  | I ^ -> O v    | INV12CK | 0.016 |   0.256 | 0.015 | 0.005 |
    | clk2T__L3_I0  | I v -> O v    | BUF1CK  | 0.053 |   0.309 | 0.044 | 0.010 |
    | clk2T__L4_I0  | I v -> O v    | BUF1CK  | 0.139 |   0.448 | 0.186 | 0.064 |
    | clk2T__L5_I0  | I v -> O ^    | INV12CK | 0.026 |   0.474 | 0.040 | 0.005 |
    | clk2T__L6_I0  | I ^ -> O ^    | BUF1CK  | 0.088 |   0.562 | 0.107 | 0.028 |
    | clk2T__L7_I0  | I ^ -> O ^    | BUF12CK | 0.072 |   0.634 | 0.041 | 0.064 |
    | clk2T__L8_I0  | I ^ -> O v    | INV12CK | 0.026 |   0.660 | 0.031 | 0.062 |
    | clk2T__L9_I0  | I v -> O ^    | INV12CK | 0.018 |   0.679 | 0.025 | 0.035 |
    | DlyOut2_x_mux | B ^ -> O ^     | MUX2    | 0.070 |   0.749 | 0.035 | 0.002 |
    | DlyOut2_x_and | I1 ^ -> O ^    | AN2     | 0.072 |   0.820 | 0.046 | 0.006 |
    |             | DlyOut2_xclk ^ |        | 0.000 |   0.821 | 0.046 | 0.006 |
    +---------------------------------------------------------------------+
```

(a) 'b1.pl' output.

```
S1_clk 4.252 0.044 2.482 0.036
S2_clk 4.377 0.045 2.192 0.037
H_xclk 4.018 0.044 2.709 0.037
CRes_xclk 1.524 0.046 2.314 0.038
Flop_clk 2.527 0.046 2.569 0.038
DlyIn1_xclk 3.740 0.048 2.863 0.041
DlyIn2_xclk 3.741 0.049 2.864 0.042
DlyOut1_clk 2.643 0.049 0.969 0.039
DlyOut2_xclk 0.967 0.046 2.62 0.038
```

(b) 'b2.pl' output.

```
S1  4252   S2  4377   Req: 0 Ach: 125   Pass
S2  2192   S1  2482   Req: 150   Ach: 290   Pass
S1  2482   H   2709   Req: 150   Ach: 227   Pass
H   4018   S1  4252   Req: 150   Ach: 234   Pass
CRes   1524   Flop   2527   Req: 900   Ach: 1003 Pass
H   2709   DlyIn   2863   Req: 0 Ach: 154   Pass
DlyIn   3741   H   4018   Req: 150   Ach: 277   Pass
Flop   2527   H   4018   Req: 150   Ach: 1491 Pass
DlyOut 2643   DlyIn   3740   Req: 1000 Ach: 1097 Pass
Flop   2527   DlyOut 2620   Req: 0 Ach: 93    Pass
CRes   2314   H   2709   Req: 300   Ach: 395   Pass
DlyIn   2864   DlyIn   3740   Req: -1000    Ach: 876   Pass
S2  4377   S2  2192   Req: -5800    Ach: -2185    Pass
DlyOut 2643   DlyOut 967    Req: -4600    Ach: -1676    Pass
H   2709   CRes   1524   Req: -3300    Ach: -1185Pass
```

(c) 'b3.pl' output.

Figure 40: CTi verification file examples

4.5 Producing the CTS Tool Input Files for CTe Synthesis

There are numerous files required for CTe synthesis like its CTi counterpart. Most of the files are generated automatically by our scripts. However, some of the analog information should be obtained from the analog designer. Luckily, most analog design software can generate these files automatically as well. The required files and the scripts that generate them (if any) can be found below.

- *pinlocs.txt:* contains location information of the clock pins on the analog circuit. Analog designer can provide them using the analog design software.
- *pincaps.txt:* contains capacitance values for each clock pin on the analog circuit. Analog designer can obtain these values using the analog design software.
- *pinrcs.txt:* contains parasitic delays for each clock pin on the analog circuit. Analog designer can provide this file using the analog design software.
- *cti.lef:* contains the pin geometries and locations for each CTi clock pin. It is generated by the script 'ctiLefGen.pl' (Table 19).

Table 19: 'ctiLefGen.pl' summary

Function	Generating 'cti.lef'
Input	'cti.cmd'
Output	'cti.lef'
Usage	*perl ctiLefGen.pl*

- *adc.lef:* This file contains layout information of the analog circuit's pins such as layer, location and geometry. It is generated automatically by the script 'adcLefGen.pl' (Table 20).
- *adc.lib:* Contains the capacitance values of the analog circuit's clock pins. It is generated by the script 'adcLibGen.pl'.

Table 20: 'adcLefGen.pl' summary

Function	Generating 'adc.lef'
Input	File containing analog circuit's clock pin locations (pinlocs.txt)
Output	'adc.lef'
Usage	*perl adcLefGen.pl*

Table 21: 'adcLibGen.pl' summary

Function	Generating 'adc.lib'
Input	File containing analog circuit's clock pin capacitance values (pinlocs.txt)
Output	'adc.lib'
Usage	*perl adcLibGen.pl*

- *cte.cmd:* is a command script written in Tcl language format. The script inputs the files that will be used and the required commands to Cadence Encounter software for CTe synthesis and analysis. It is generated automatically by the script 'createCTeCmd.pl'.

Table 22: 'createCTeCmd.pl' summary

Function	Generating 'cte.cmd'
Input	Files 'cti.lef' and 'adc.lef'
Output	'cte.cmd'
Usage	*perl createCTeCmd.pl cti.lef adc.lef*

- *cte.conf:* The file introduces the CTe design and the standard design library to the Encounter session. It contains the names of Verilog files (.v), standard library characterization files (.lib) and the LEF files (.lef) that will be used for synthesis. It is generated automatically by the script 'createCTeConf.pl'.

- *cte.v:* contains the Verilog file that generates the structural frame of the CTe design. It is created automatically by the script 'createCTeVerilog.pl'.

Table 23: ’createCTeConf.pl’ summary

Function	Generating ’cte.conf’
Input	-
Output	’cte.conf’
Usage	*perl createCTeConf.pl*

Table 24: ’createCTeVerilog.pl’ summary

Function	Generating ’cte.v’
Input	Definition file ’defFile.txt’
	File that keeps the SDAC clock pin capacitance values ’pincaps.txt’
Output	’cte.v’
Usage	*perl createCTeVerilog.pl defFile.txt pincaps.txt*

- *cte.ctstch:* The clock tree specification file ’cte.ctstch’ contains the required definitions and constraints necessary for CTe synthesis. It assists Cadence Encounter during synthesis. Details of the script that generates the file can be found in Table 25.

Table 25: ’ctstchGen.pl’ summary

Function	Generating ’cte.ctstch’
Input	’cti.cmd’
	File that keeps SDAC pin capacitance values ’pincaps.txt’
	File that contains SDAC pin parasitic delays ’pinrcs.txt’
Output	’cte.ctstch’
Usage	*perl ctstchGen.pl*

- *cte_report_timing_commands:* contains the commands that are necessary for CTe result analysis. It is generated by the script ’createCTeCommands.pl’ automatically.

Table 26: 'createCTeCommands.pl' summary

Function	Generating commands necessary for CTe analysis
Input	Definition file 'defFile.txt' File that keeps SDAC pin capacitance values 'pincaps.txt'
Output	Command file 'cte_report_timing_commands'
Usage	*perl createCTeCommands.pl defFile.txt pincaps.txt*

4.6 CTe Synthesis and Timing Analysis via the CTS Tool

Cadence Encounter runs the file 'cte.cmd' for CTe synthesis, and 'cte_top.tcl' for CTe analysis and closing Encounter. The head script 'topScript_cte.pl' calls Encounter with the command

system("/ECE/Cadence/IC/EDI_9.12/bin/encounter -nowin -init cte_top.tcl > encounterout.txt");

Details of the command was given in section 4.3. During CTe synthesis CTi results are also taken into account. CTi clock pin locations and CTi clock rise and fall skew values are all supplied to Encounter for analysis. This way CTi circuit can be emulated during CTe synthesis. Resulting layout for the CTi + CTe circuit can be observed in Figure 41.

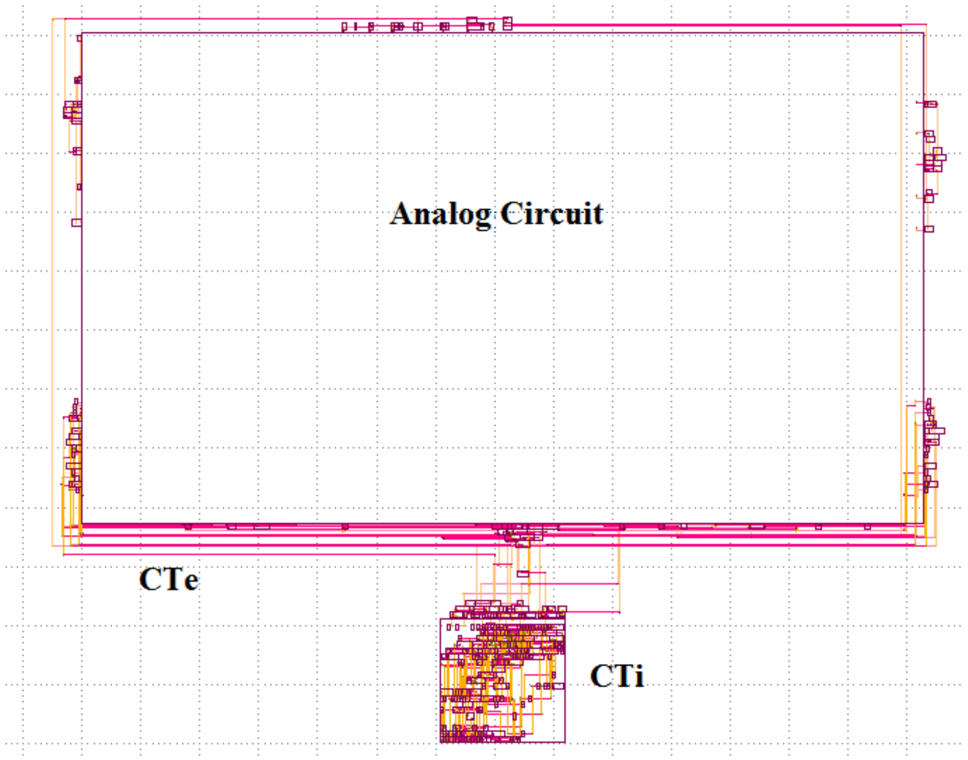
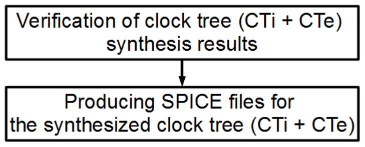


Figure 41: CTi + CTe layout

CHAPTER V

VERIFICATION

Up to this point CTi and CTe synthesis have been done, completing clock tree generation. However synthesized clock tree should be verified by checking whether the design is compatible with the constraints or not before merging it with the SDAC. Verification consists of two steps as shown in Figure 42.



Verification of clock tree (CTi + CTe) synthesis results
Producing SPICE files for the synthesized clock tree (CTi + CTe)

Figure 42: Verification steps

5.1 *Verification of Clock Tree (CTi + CTe) Synthesis Results*

As explained in Chapter 1 before, each clock phase generated in CTi may feed a group of switches rather than one. So, each of these clock phases should be delivered to several switches at once. This means, for CTe synthesis a CTi signal might be represented with several subsignals. For a CTi signal S1, there may be variants such as S1_L1 and S1_L2 in CTe. The example in Figure 43 shows such case.

Looking at the example in Figure 43, $S1_{target}$ is the aimed clock generated at CTi. The clock signals S1_L1_clk, S1_L1_xclk and S1_L2_xclk are the clock variants synthesized at CTe. The signal $S1_{synthesis}$ is the worst case scenario combination of these variants. The combined signal has two rising edges (r_{max}, r_{min}) and two falling edges (f_{max}, f_{min}). Worse edge is selected depending on the constraint. As an example, for the constraint '1f 3r 150' one should choose the maximum value for '1f' and the

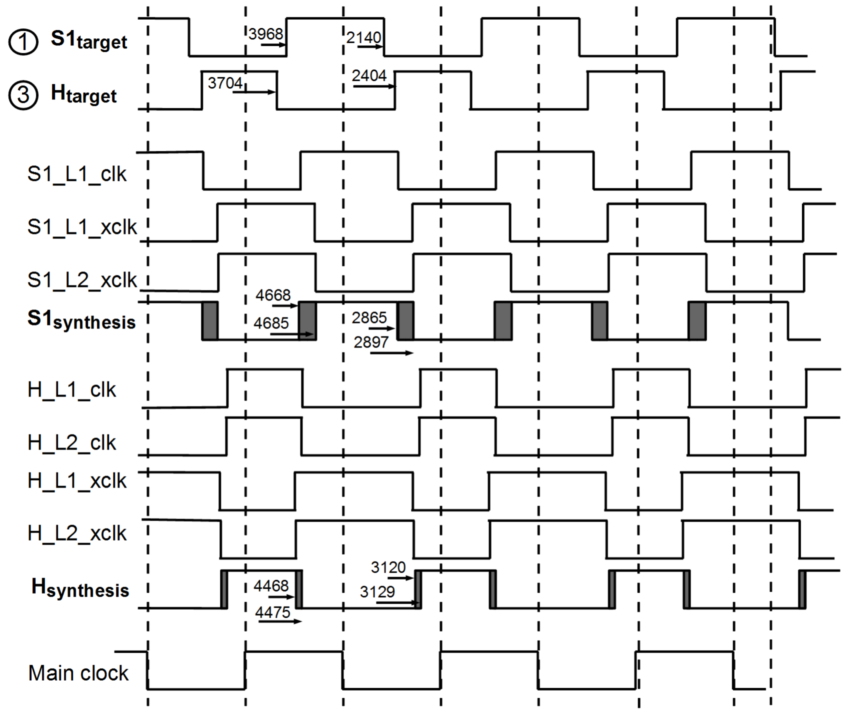


Figure 43: A waveform depicting target clocks, generated clock variants, and the combinations of each clock variant

minimum value for '3r'. Then the constraint will be verified with the values '1f$_{max}$ 3r$_{min}$ 150' and entering the values from Figure 43, the result will be '3120-2897=223' which satisfies the constraint. Since the constraint was checked with the worst case combination, it is evident that each variant will pass the verification.

Verification is done in three steps given below.

1. The resulting delay values from both CTi and CTe synthesis are added up and rising time, rising skew, falling time and falling skew values for each clock pin is obtained.

2. Minimum and maximum edge values for each clock edge is calculated.

3. Verification of the constraint file with the values obtained at step 2.

The scripts that handle these steps and their details can be found below.

- *cte1.pl:* This script calculates the rising time, rising skew, falling time and falling skew values for each analog clock pin for each PVT edge.

Table 27: 'cte1.pl' summary

Function	Calculating rise time, fall time, rise edge and fall edge values for each clock pin using the resuls from CTi and CTe synthesis
Input	File containing CTi synthesis results 'cti_results.txt' File containing CTe synthesis results 'cte_results.txt'
Output	'cte1Out.txt'
Usage	*perl cte1.pl cti_results.txt cte_results.txt cte1Out.txt*

- *cte2.pl:* determines the minimum and maximum delays for each clock edge (r$_{max}$, r$_{min}$, f$_{max}$, f$_{min}$). The script is run for each PVT edge.

- *cte3.pl:* verifies the constraint file with the worst possible scenario values obtained from 'cte2Out.txt'.

Table 28: 'cte2.pl' summary

Function	Calculating maximum and minimum delays for each clock edge (r_{max}, r_{min}, f_{max}, f_{min})
Input	cte1Out.txt Definition file 'defFile.txt'
Output	'cte2Out.txt'
Usage	*perl cte2.pl cte1Out.txt defFile.txt cte2Out.txt*

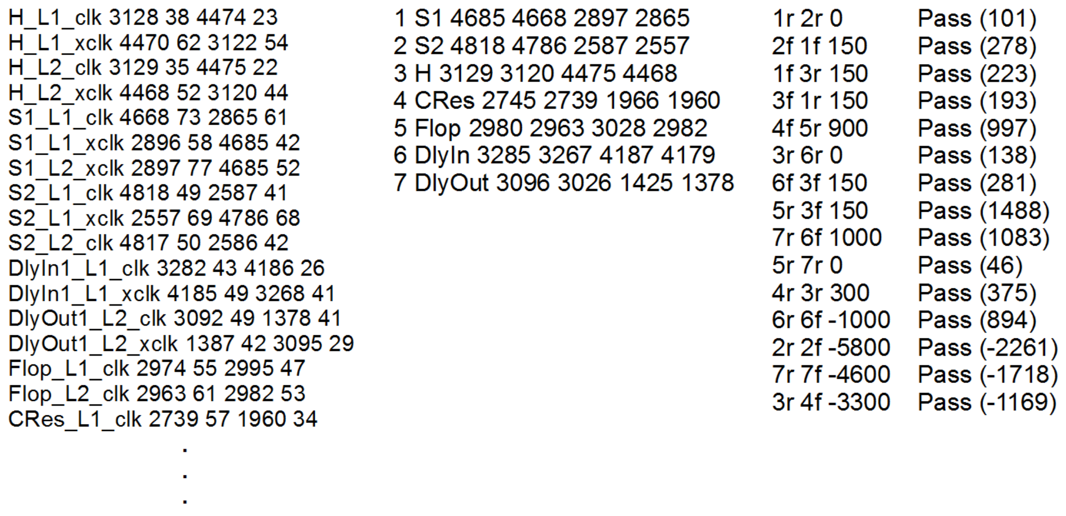
Table 29: 'cte3.pl' summary

Function	Verification of the constraint file using synthesis results
Input	cte2Out.txt Constraint file 'constraints.txt'
Output	'cte3Out.txt'
Usage	*perl cte3.pl cte2Out.txt constraints.txt cte3Out.txt*

If each constraint pass the verification phase, the clock tree is generated successfully and it can be merged with the SDAC for analog verification. The resulting outputs from the verification scripts can be seen in Figure 44.

5.2 Producing SPICE Files for the Synthesized Clock Tree

In this step, SPICE files for the clock tree are generated for the three PVT corners, enabling analog simulation of the merged (analog circuit and clock tree) circuit. The files are generated automatically by a Perl script. The analog simulation results can be observed in Chapter 6.



```
H_L1_clk 3128 38 4474 23
H_L1_xclk 4470 62 3122 54
H_L2_clk 3129 35 4475 22
H_L2_xclk 4468 52 3120 44
S1_L1_clk 4668 73 2865 61
S1_L1_xclk 2896 58 4685 42
S1_L2_xclk 2897 77 4685 52
S2_L1_clk 4818 49 2587 41
S2_L1_xclk 2557 69 4786 68
S2_L2_clk 4817 50 2586 42
DlyIn1_L1_clk 3282 43 4186 26
DlyIn1_L1_xclk 4185 49 3268 41
DlyOut1_L2_clk 3092 49 1378 41
DlyOut1_L2_xclk 1387 42 3095 29
Flop_L1_clk 2974 55 2995 47
Flop_L2_clk 2963 61 2982 53
CRes_L1_clk 2739 57 1960 34
                  .
                  .
                  .
```

(a) 'cte1.pl' output.

```
1 S1 4685 4668 2897 2865
2 S2 4818 4786 2587 2557
3 H 3129 3120 4475 4468
4 CRes 2745 2739 1966 1960
5 Flop 2980 2963 3028 2982
6 DlyIn 3285 3267 4187 4179
7 DlyOut 3096 3026 1425 1378
```

(b) 'cte2.pl' output.

```
1r 2r 0        Pass (101)
2f 1f 150      Pass (278)
1f 3r 150      Pass (223)
3f 1r 150      Pass (193)
4f 5r 900      Pass (997)
3r 6r 0        Pass (138)
6f 3f 150      Pass (281)
5r 3f 150      Pass (1488)
7r 6f 1000     Pass (1083)
5r 7r 0        Pass (46)
4r 3r 300      Pass (375)
6r 6f -1000    Pass (894)
2r 2f -5800    Pass (-2261)
7r 7f -4600    Pass (-1718)
3r 4f -3300    Pass (-1169)
```

(c) 'cte3.pl' output.

Figure 44: Outputs of the scripts cte1.pl, cte2.pl and cte3.pl respectively

CHAPTER VI

RESULTS

As explained in the previous chapter, design of the SDAC is handled in two parts. Analog circuit is designed without taking clock tree into consideration, and the clock tree is generated via ACTreS software. Since both circuits can be implemented in CMOS, these two circuits are merged into the actual SDAC circuit afterwards.

ACTreS obtains clock definitions and constraints along with clock pin locations, capacitance values and RC delays from the analog designer. Layout of the synthesized clock tree and resulting SPICE files are generated and provided to the analog designer in return. After merging the clock tree with the SDAC, the designer can test the circuit using an analog simulation software. The results obtained from the simulation are then analyzed on MATLAB and the effective number of bits (ENOB) can be determined. Figure 45 shows the SDAC clock tree testing environment used for verification.

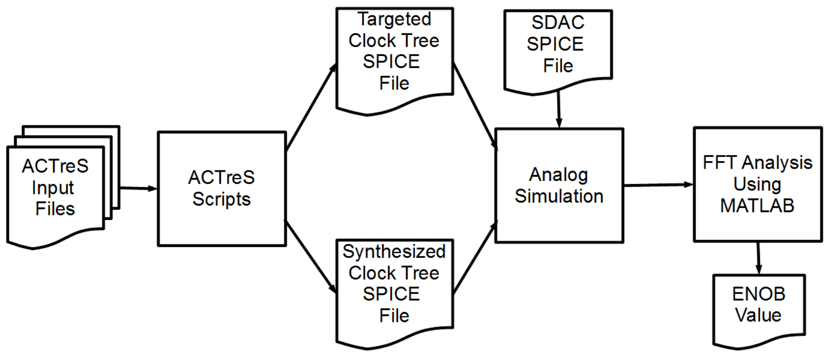

ACTreS Input Files
ACTreS Scripts
Targeted Clock Tree SPICE File
Synthesized Clock Tree SPICE File
SDAC SPICE File
Analog Simulation
FFT Analysis Using MATLAB
ENOB Value

Figure 45: Testing environment for SDAC clock tree

Using the test environment above, one can determine ENOB (Effective Number of Bits), SNR (Signal to Noise Ratio), SFRD (Spurious-Free Dynamic Range), INL (Integral Non-Linearity) and DNL (Differential Non-Linearity) values as seen in Table 30. The results in the table were obtained for:

- both slack algorithms (the one that tries to center each constraint slack(mean) and the one that tries to center each constraint with highest possible HiTime slacks (HiTime))
- targeted clock tree and synthesized clock tree (targeted/synthesized)
- each PVT corner (fast/normal/slow)

Table 30: ACTreS Verification Results

Target File Algorithm	**Targeted/ Synthesized**	**PVT Corner**	**ENOB**	**SNR (dB)**	**SFDR (dB)**	**Energy/bit (pJ/Cs)**	**INL/DNL (LSB)**
Mean	Targeted	Slow	7.58	47.39	56.23	6.5	
Mean	Synthesized	Slow	6.78	42.57	55.30	11.45	
Mean	Targeted	Normal	7.83	48.89	59.56	5.4	
Mean	Synthesized	Normal	7.66	47.87	58.41	6.18	
Mean	Targeted	Fast	6.61	41.55	55.28	12.79	
Mean	Synthesized	Fast	6.77	42.52	55.26	11.45	
HiTime	Targeted	Slow	7.63	47.69	57.35	6.31	
HiTime	Synthesized	Slow	7.00	43.90	56.14	9.76	
HiTime	Targeted	Normal	7.78	48.60	58.84	5.6	0.3/0.4
HiTime	Synthesized	Normal	7.63	47.69	58.60	6.31	0.3/0.5
HiTime	Targeted	Fast	6.61	41.55	55.64	12.79	
HiTime	Synthesized	Fast	6.75	42.27	55.16	11.77	

SNR is the ratio between the base signal and its noise components. The values at **SNR** column are calculated by MATLAB using Fast Fourier Transform (FFT). The following measures were taken to ensure reliability.

- the results were taken for 1024 data samples,

- sampling frequency was chosen so that $\frac{inputfrequency}{samplingfrequency} \notin \mathbb{Z}$ to prevent sample repetition.
- settling time of the circuit was taken into account, data received during settling time was not calculated.

Considering the results in Table 30, SNR is around 10%, which is consistent with the 10% SNR goal of the project.

ENOB values were obtained directly from SNR values using the formula below.

'ENOB = (SNR - 1.76)/6.02'

Looking at the ENOB column in Table 30, it can be determined that for a certain method and PVT corner, ENOB values differ around 4% between targeted and synthesized clock trees. These values prove performance of the design is affected minimally from using a digital CTS tool, it greatly depends on the generated clock list. Looking at the ENOB values for synthesized clock tree, it can be observed that the values change between 6.75 and 7.63. The results seem to be close to the targeted ENOB value 10, considering the theoretical ENOB values should differ from the theoretical values and the maximum available ENOB value is not certain. It was stated earlier that the analog designer needs a target clock tree and the target clock tree which can be found in 'schedule.txt' can be used by the analog designer.So if the analog designer uses the target clocks generated by ACTreS, designing the clock tree manually or by using Cadence Encounter CTS tool will differ around 4% according to Table 30.

SFDR can be determined as the ratio between the base signal and the greatest harmonic noise component. It is somewhat similar to SNR, however it only takes the greatest noise component into account contrary to SNR. SFDR values for both targeted and synthesized clock trees are almost the same (with 2% deviation at most) according to Table 30.

Energy per Conversion Step is a widely used criterion for ADC circuit comparison. It is the amount of energy spent for a complete conversion. The formula for this criterion is given below.

$E/Cs = Power/(2^{ENOB} xSamplingFrequency)$

As it can be observed from the formula, Energy per Conversion step requires ENOB, sampling frequency, and Power. ENOB values were calculated using MATLAB, as instructed above. Sampling frequency is determined by the tester. Power value was obtained from Mentor Graphics software.

INL and DNL are values that give information about the static performance of an ADC circuit. INL (Integral Non-Linearity) is the distance (in LSB) between the ideal curve and transfer characteristics of the ADC circuit. DNL (Differential Non-Linearity) is the analog distance (in LSB) between two consecutive digital codes.

There are numerous INL/DNL calculation methods in literature, however most of these methods take up much time and they specifically uncover the mismatch problems. Taking these into account, we discarded detailed INL/DNL simulations and obtained INL/DNL values for our HiTime algorithm on typical PVT corner with 0.1 LSB sensitivity, presuming the other 5 comparison would yield similar results.

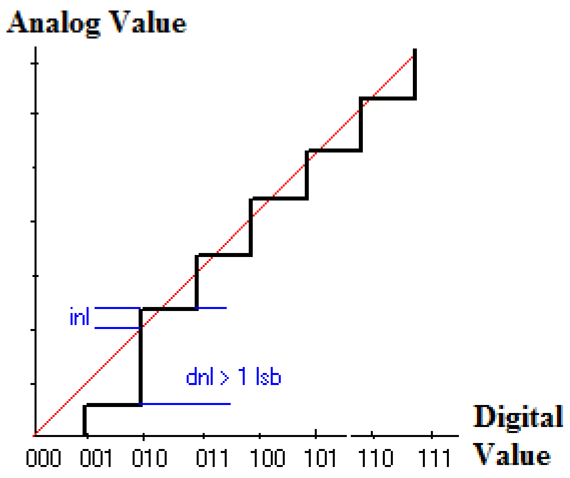


Figure 46: Ramp signal input and INL/DNL values

We used ramp signal input method for calculation whose sensitivity rises with the sampling rate. Approximately 100000 samples should be taken for each clock set for a sensitivity value of 0.01 LSB. We have taken 10240 samples which resulted in 0.1 LSB sensitivity. As it can be seen from Table 30, difference between targeted and synthesized values are not more than 0.1 LSB. An example for ramp signal input method and the corresponding INL and DNL values can be seen in Figure 46.

CHAPTER VII

CONCLUSION

Inspired by the automation of digital clock trees, we have devised a methodology that can automatically generate SDAC (Sampled Data Analog Circuit) clock trees. Our work enables faster and less error-prone SDAC design. To achieve our goals, we proposed a systematic approach to the problem, which is also beneficial for traditional manual SDAC clock tree design. Moreover, our systematic approach also shortens analog design phase, since symmetrical design practices used in SDAC clock tree design are less crucial.

For testing purposes, our collaborators at Boğaziçi University designed an ADC (Analog-Digital Converter) [1]. Simulating this ADC along with the generated clock tree in SPICE, we obtained ENOB (Effective Number Of Bits) values that are sufficiently close to our ideal expectations.

ACTreS (Analog Clock Tree Synthesis) works automatically after obtaining the circuit requirements from the analog designer. We encountered 3 core graph problems during our research that are,

- finding loops in a graph,
- getting rid of the loops encountered above,
- generating a tight schedule from a constraints graph (a longest path problem).

We used an existing solution for the first problem and created a fast and simple algorithm for the second problem. However, regarding the third problem, we think that we were able to find a solution that will benefit the literature.

Although we achieved our initial goals, ACTreS can be further optimized in the future. During our discussions we found out that our target file generation algorithm could be further optimized for better performance.

Our ADC design showed us that we can automate the process successfully, working on other possible design problems and further collaboration with the industry may help us in greater optimization and possible bug-fixing.

APPENDIX

AUTOMATION SETUP

ACTreS is a set of scripts and files that follow a strict hierarchy, so it might be beneficial to give a brief software overview. Various programming and scripting languages such as Perl, C++, Python and Tcl are used throughout the project.

ACTReS automation software is formed by clusters of scripts. It consists of the CTi synthesis cluster (namely topScript_cti.pl) and CTe synthesis cluster (topScript_cte.pl). These two scripts are at the top of the hierarchy, each followed by several smaller scripts. Figure 47 shows the script hierarchy where CTi and CTe script hierarchies can be observed respectively at Figures 47(a) and 47(b).

```
| topScript_cti.pl
| | script1.pl
| | consToGraph.pl
| | target_noloop.pl
| | | cycles.py
| | | acycl.pl
| | sch.pl
| | schedule
| | scheduleChk.pl
| | sch2spice.pl
| | fast2corners.pl
| | ctstchGen2.pl
| | createCTiCmd.pl
| | createCTiCommands.pl
| | createCTiConf.pl
| | createCTiVerilog.pl
| | cti_top.tcl
| | b1.pl
| | b2.pl
```

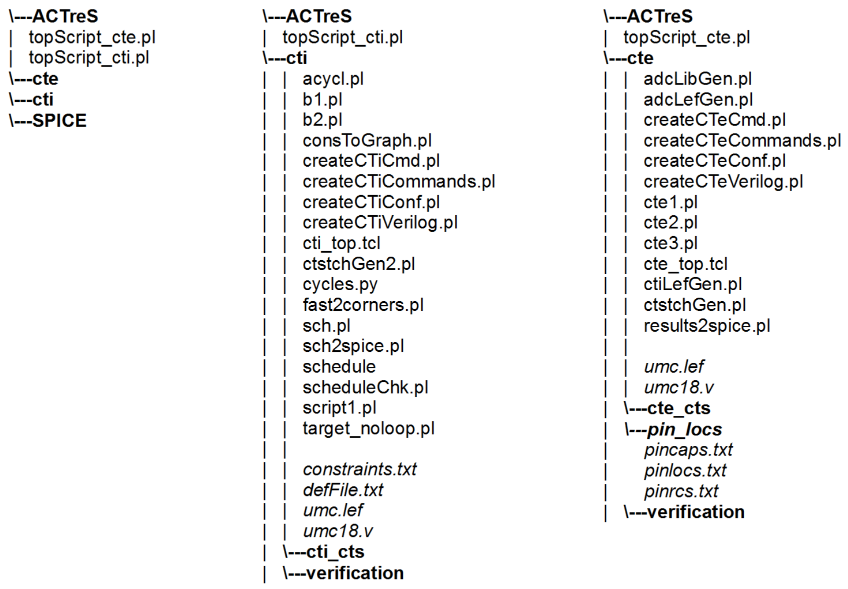
(a) CTi script hierarchy

```
| topScript_cte.pl
| | ctiLefGen.pl
| | adcLefGen.pl
| | adcLibGen.pl
| | createCTeCmd.pl
| | createCTeCommands.pl
| | createCTeConf.pl
| | createCTeVerilog.pl
| | ctstchGen.pl
| | cte_top.tcl
| | cte1.pl
| | cte2.pl
| | cte3.pl
| | results2spice.pl
```

(b) CTe script hierarchy

Figure 47: ACTreS script hierarchy

Since automation scripts exchange data in form of files, a certain folder hierarchy is required. Figure 48 shows the folder system in use. In this figure, bold names denote folders, italic names are used to identify the files that are obtained externally, and the non-bold, non-italic names represent the script files. There is a top folder called ACTReS, and under it we have 3 other folders as shown in Figure 48(a). The CTi folder architecture can be observed in Figure 48(b), and CTe folder architecture can be seen in Figure 48(c).



```
\---ACTreS
|  topScript_cte.pl
|  topScript_cti.pl
\---cte
\---cti
\---SPICE
```

(a) top folder

```
\---ACTreS
|  topScript_cti.pl
\---cti
|  |  acycl.pl
|  |  b1.pl
|  |  b2.pl
|  |  consToGraph.pl
|  |  createCTiCmd.pl
|  |  createCTiCommands.pl
|  |  createCTiConf.pl
|  |  createCTiVerilog.pl
|  |  cti_top.tcl
|  |  ctstchGen2.pl
|  |  cycles.py
|  |  fast2corners.pl
|  |  sch.pl
|  |  sch2spice.pl
|  |  schedule
|  |  scheduleChk.pl
|  |  script1.pl
|  |  target_noloop.pl
|  |
|  |  constraints.txt
|  |  defFile.txt
|  |  umc.lef
|  |  umc18.v
|  \---cti_cts
|  \---verification
```

(b) CTi folder hierarchy

```
\---ACTreS
|  topScript_cte.pl
\---cte
|  |  adcLibGen.pl
|  |  adcLefGen.pl
|  |  createCTeCmd.pl
|  |  createCTeCommands.pl
|  |  createCTeConf.pl
|  |  createCTeVerilog.pl
|  |  cte1.pl
|  |  cte2.pl
|  |  cte3.pl
|  |  cte_top.tcl
|  |  ctiLefGen.pl
|  |  ctstchGen.pl
|  |  results2spice.pl
|  |
|  |  umc.lef
|  |  umc18.v
|  \---cte_cts
|  \---pin_locs
|     pincaps.txt
|     pinlocs.txt
|     pinrcs.txt
|  \---verification
```

(c) CTe folder hierarchy

Figure 48: ACTreS folder hierarchy

There are some folders and files that are important to the user, so we should explain those briefly. The SPICE folder holds the circuit model of the generated clock tree, which can be used for testing and manufacturing purposes. Files 'constraints.txt' and 'defFile.txt' should be created by collaborating with the analog designer, according to the timing specifications of the SDAC. Folder 'pin_locs' and its contents should be provided by the analog designer according to the SDAC analog design. File 'pinlocs.txt' contains the physical coordinates of each clock pin, 'pincaps.txt' holds the capacitance values of said pins, and finally 'pinrcs.txt' has the parasitic delay values for each clock pin. Since all these values can be extracted using analog design tools (assuming we have finished the SDAC design with or without taking clock tree phase into account), they can be easily acquired. It can also be seen in Figure 48 that both CTi and CTe folders have 'verification' folders. If synthesis is not successful, the user can look at these folders to find out the problem.

Bibliography

[1] V. B. Esen., "10-Bit 60 MS/s Two Step Flash ADC Design," M.S. thesis, Dept. Electrical & Electronics Eng., Bogazici Univ., Istanbul, Turkey, 2013.

[2] M. R. Guthaus, G. Wilke, and R. Reis. (2013, Mar.). Revisiting Automated Physical Synthesis of High-Performance Clock Networks. ACM Trans. Design Automation of Electronic Systems (TODAES). 18(2), article no. 31.

[3] J. B. Hughes, K. W. Moulding, J. Richardson, J. Bennet, W. Redman-White, M. Bracey, and R. S. Soin. (1996, Jul.). Automated Design of Switched-Current Filters. IEEE Journal of Solid-State Circuits. 31(7), pp. 898907.

[4] P. Sniatata and R. Rudnicki, "Automated Design and Layout Generation for Switched Current Circuits," in Proc. ISCAS, Kos, Greece, 2006, pp. 637640.

[5] M. Karlsson, M. Vesterbacka, and W. Kulesza, "A Non-overlapping Two-phase Clock Generator with Adjustable Duty Cycle," in Proc. GigaHertz Symp., Linkping, Sweden, 2003, poster no. 008:028.

[6] S. Stronski, "Load-adapted Clock Generator in CMOS Circuits," U.S. Patent 4 761 568, Aug. 2, 1988.

[7] U. Seng-Pan, R. P. Martins, J. E. da Franca, Design of Very High-Frequency Multirate Switched-Capacitor Circuits. Springer, 2006.

[8] D. Ma and F. Luo. (2008, Jun.). Robust Multiple-Phase Switched-Capacitor DCDC Power Converter with Digital Interleaving Regulation Scheme. IEEE Trans. VLSI Systems, 16(6), pp. 611619.

[9] G. Manganaro, "An Improved Phase Clock Generator for Interleaved and Double-Sampled Switched-Capacitor Circuits," in Proc. ICECS, Malta, 2001, pp. 15531556 vol. 3.

[10] J. L. Neves and E. G. Friedman. (1996, Jun.). Design Methodology for Synthesizing Clock Distribution Networks Exploiting Nonzero Localized Clock Skew. IEEE Trans. VLSI Systems, 4(2), pp. 286291.

[11] B. Yuce, S. Korkmaz, V. B. Esen, F. Temizkan, C. Tunc, G. Guner, I. F. Baskaya, I. Agi, G. Dundar, and H. F. Ugurdag, "Synthesis of Clock Trees for Sampled-Data Analog IC Blocks," in Proc. EWDTS, Kharkiv, Ukraine, 2012, pp. 46-49. (Note: Paper appeared on IEEExplore as part of Proc. EWDTS, Rostov-on-Don, Russia, 2013.)

[12] J. Schauer. (Accessed: 2013, March). Finding all the elementary circuits of a directed graph. URL: https://github.com/josch/cycles_tarjan

[13] R. E. Tarjan. (1972). Enumeration of the Elementary Circuits of a Directed Graph. SIAM Journal Computing, 2(3), pp. 211216.

VITA

Name Surname: Gökhan Güner

Address: Mustafapaşa M. Devrim C. 705/1 S. No: 44 Gebze 41400 Kocaeli-Turkey

Birthplace / Year: İstanbul / 1986

Languages: Turkish (native) - English

Education:

- **High School:** İzmir Fen Lisesi - 2004
- **BS:** Yeditepe University - 2010
- **MS:** Özyeğin University - 2014
- **Name of Program:** M.Sc. in Electrical and Electronics Eng.

Publications:

- B. Yuce, S. Korkmaz, V. B. Esen, F. Temizkan, C. Tunc, **G. Guner,** I. F. Baskaya, I. Agi, G. Dundar, H. F. Ugurdag, Synthesis of clock trees for sampled-data analog IC blocks, *East-West Design & Test Symposium,* Kharkov, 2012.
- F. Temizkan, H. Sahin, **G. Guner,** H. F. Ugurdag, S. Goren, Look-up table based polynomial approximation for exponential functions on ASICs, *Computer Science Student Workshop,* Istanbul, 2012.