THE REPUBLIC OF TURKEY
BAHÇEŞEHİR UNIVERSITY

EFFICIENT COMBINATIONAL CIRCUITS FOR CONSTANT DIVISION

M.S. Thesis

ANIL BAYRAM

İSTANBUL, 2013

THE REPUBLIC OF TURKEY
BAHÇEŞEHİR UNIVERSITY

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
ELECTRICAL AND ELECTRONICS ENGINEERING

EFFICIENT COMBINATIONAL CIRCUITS FOR CONSTANT DIVISION

M.S. Thesis

ANIL BAYRAM

Supervisor: Assoc. Prof. H. Fatih UĞURDAĞ

İSTANBUL, 2013

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
ELECTRICAL AND ELECTRONICS ENGINEERING

Title of Thesis: Efficient Combinational Circuits For Constant Division
Name/Last Name of the Student: Anıl BAYRAM
Date of Thesis Defense: September 2, 2013

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assoc. Prof. Tunç BOZBURA
Director

This is to certify that we have read this thesis and that we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Science.

Assoc. Prof. Çiğdem Eroğlu ERDEM
Program Coordinator

Examining Committee Members	Signature
Assoc. Prof. H. Fatih UĞURDAĞ:	_____
Assoc. Prof. Çiğdem Eroğlu ERDEM:	_____
Asst. Prof. Yalçın ÇEKİÇ:	_____

To my dearest wife

ACKNOWLEDGEMENTS

It would not have been possible to write this thesis without the support of the kind people around me, to only some of whom it is possible to give particular mention here.

Foremost, I would like to express my sincere gratitude to my supervisor Assoc. Prof. H. Fatih Uğurdağ for the continuous support on my M.S. Study and research, for his motivation, enthusiasm and his immense knowledge, which he was always happy to share.

I also want to thank Assoc. Prof. Sezer Gören Uğurdağ for her help and valuable contributions on the development process of this thesis.

In addition, I would like to thank to Assoc. Prof. Çiğdem Eroğlu Erdem and Asst. Prof. Yalçın Çekiç, my thesis committee members, for taking the time.

I would like to thank my wife Sevgi for her personal support and great patience at all times. She has given me her unequivocal support throughout, as always, for which my mere expression of thanks does not suffice.

I also want to thank my colleagues Fatih Temizkan, Hatice Şahin, Cihan Tunç and Bilgiday Yüce for their support and valuable friendship.

Last but not the least, I would like to thank my parents and my brother for their endless support and patience throughout my life.

Anıl BAYRAM

İstanbul, September 2013

ABSTRACT

EFFICIENT COMBINATIONAL CIRCUITS FOR CONSTANT DIVISION

Bayram, Anıl

Electrical and Electronics Engineering

Thesis Supervisor: Assoc. Prof. H. Fatih Uğurdağ

September 2013, 54 Pages

Division by an integer constant is an operation that occurs so often to justify a customized implementation for it. Many systems that require such divisions mostly demand exact integer quotient and exact integer remainder results instead of a single fractional result. Besides, the constant division operations are usually required to process rapid and repeated calculations in the system, so the performance parameters for constant division gets more importance. This study examines the division circuits for constant integer dividers yielding an integer quotient with an integer remainder (i.e. Euclidean division) and proposes a fast Look-Up Table (LUT) based combinational constant division method with a binary tree like approach. Although general division is a laborious arithmetic operation, constant integer division provides us the luxury of knowing the divisor at compile time. For integer division, the division circuit is uniquely generated for division with a single integer constant. Given the divisor beforehand, LUTs can be used to store the results at compile time. But as the dividend bit-width increases, the LUT sizes increase exponentially, thus resulting an area-inefficient solution. This research introduces a binary tree approach to LUT-based integer division models. The method greatly reduces calculation time, increasing the run-time efficiency. The timing improves over 30% for input bit widths of 24 bits when compared to the latest LUT-based method and gets better as the bit width increases. The study also includes a detailed comparison between LUT based methods and other known integer division methods.

Keywords: Computer Arithmetic, HDL, Logic Synthesis, RTL Generation, Constant Division, LUT

ÖZET

SABİT SAYILARLA BÖLME İÇİN ETKİN BİRLEŞİMSEL DEVRELER

Bayram, Anıl

Elektrik - Elektronik Mühendisliği

Tez Danışmanı: Doç. Dr. H. Fatih Uğurdağ

Eylül, 54 Sayfa

Sabit sayılarla bölme işlemi, özel bir işlem operatörü aranmasını gerektirecek kadar sıklıkla karşılaşılan bir problemdir. Bu işlemi gerektiren sistemlerin çoğu işleyiş açısından kesirli tek bir sonuç yerine ayrı birer bölüm ve kalan içeren sonuçlar gerektirirler. Bununla beraber, sabit sayılarla bölme işlemlerinin sistem içerisinde genellikle hızlı ve tekrarlı bir şekilde çalışıyor olmaları beklenir. Bu çalışma, sabit tam sayılarla bölme ve sonuç olarak yine bir tamsayı bölüm ve tamsayı kalan elde eden sistemleri konu alır ve bunun yanında sayı tablosu tabanlı yeni bir sabit sayılarla bölme metodu sunar. Genel bölme işlemi her ne kadar zahmetli olsa da, sabit sayılarla bölme durumu bize bölen sayısını önceden biliyor olma lüksünü sağlayıp buna bağlı performans iyileştirmelerine olanak tanır. Dijital ortamda sayı tabloları, karmaşık hesaplamalardan kurtulabilmek için derleme zamanında yapılabilecek olan hesaplamaları önceden yaparak hafızada saklayabilmemizi sağlar. Sabit sayılarla bölme durumunda da bölen sayıyı biliyor olduğumuz için sonuçları sayı tablolarında saklama seçeneğimiz vardır. Fakat bütün olası girdiler için bölüm sonucunu sayı tablolarında saklamak devre kaynakları kullanımını olumsuz bir şekilde arttıracak, ve özellikle geniş bit uzunluğunda olan devreler için bu yöntemin uygulanabilirliğini imkansız kılacaktır. Bu çalışmada, geniş bit genişliklerinde de sayı tabloları yönteminin hızından faydalanabilmek için yeni bir yöntem sunuyor ve bu yöntemle en yakın rakip yönteme kıyasla 30% oranında bir hız artışı sağlabildiğini, devre kaynakları kullanımının ise hala makul seviyelerde tutulabildiğini gösteriyoruz. Son olarak değer aralıkları arasında seçim yapabilen, en uygun donanım tanımlama dili kodunu oluşturan bir RTL üreteci sunuyoruz.

Anahtar Kelimeler : Bilgisayar Aritmetiği, Donanım Tanımlama Dilleri, Lojik Sentezi, RTL üreteçleri, Sabit sayılara bölme, Başvuru Çizelgesi

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Area – Timing Product	: ATP
Binary Coded Decimal	: BCD
Digital Signal Processing	: DSP
Design Compiler	: DC
Electronic Design Automation	: EDA
Flip-Flop	: FF
Hardware Description Language	: HDL
Look-Up Table	: LUT
Register Transfer Level	: RTL

1. INTRODUCTION

Division is the most laborious of all basic arithmetic operations. It is encountered in almost every algorithm, therefore achieving optimal division circuits has been subject to numerous studies. On the other hand, constant integer division is a special form of division where the divisor is known at compile time. Constant integer dividers divide their input dividend only by one constant divisor that they are built for.

The division operation is often encountered as a "division by an integer constant" form and such divisions are usually rapid and repeated sequence of operations. Therefore, the timing performance is more important in constant division circuits, justifying the necessity for a special operator for it.

Many types of special purpose systems require rapid and repeated division by a set of known constant divisors. Some are;

i. Base conversions,
ii. Scaling for Normalization,
iii. Exponent alignment in non-binary bases,
iv. Address mappings,
v. Interleaved memory schemes (usually with prime number of modules),
vi. Element locating in multidimensional lists,
vii. Mathematical equations in many algorithms (i.e. algorithms for network traffic processing)

Almost all systems that require division by a constant integer use it for data sorting or mapping, therefore they require the output in the form of an integer quotient with an integer remainder instead of a single fractional result with an accuracy up to a specified digit. Many studies in the literature output fractional results; there are other studies that is focused on correctly rounding the numbers to get the quotient and the remainder, introducing an extra burden to the overall timing performance. The best timing results are achieved in LUT based methods, but as the dividend gets larger, the access time of LUTs increases and their chip area becomes unreasonably large.

1.1 SCOPE

In this thesis we analyze the constant integer division algorithms with exact integer quotient and remainder results. The constant division system that is considered in this study is shown in Figure 1.1.

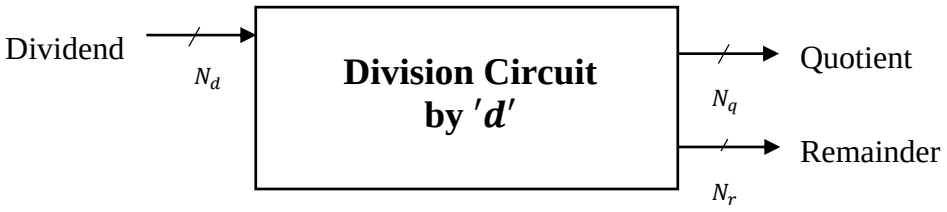

Figure 1.1 The Division System by "d"

A constant division algorithm is introduced which outputs an integer quotient and an integer remainder with an efficient use of Look-Up Tables (LUT) by a binary tree like approach. The proposed method is named as "Binary Tree Based Constant Division" (BTCD).

Some constant division algorithms with fractional results are also considered and mentioned as long as their outputs can be further processed to form an exact quotient and an exact remainder.

The HDL generators for constant division methods and optimization techniques with generators are also included in the scope of this study.

1.2 GOAL OF THE THESIS

The main purpose of this study is to introduce a LUT based constant division method which has better timing performance than its competitors especially in divisions of dividends with large bit widths. The important thing is that while achieving a smaller clock period, we must assure that the occupied chip area remains reasonable.

Along with the algorithm, this study will introduce a different type of categorization for constant integer division methods together with a good comparison in the selected category.

The last aim of this study to set a new method for HDL generators where the HDL generator can be programmed to select an optimal implementation of the method within a given flexibility range.

1.3 OUTLINE OF THE THESIS

Chapter 2 consists of a literature survey. Brief explanations of some selected methods are given. A detailed explanation of our only competitor, the Table Based Constant Division Method (TBCD) is given.

In Chapter 3, our suggested method, BTCD is presented.

Chapter 4 is about how we implemented, verified, and synthesized the circuits. The HDL generators scripts, verification scripts, and our iterative synthesis script will be explained in detail.

Chapter 5 includes the results that we gathered with comparisons of BTCD and TBCD methods in terms of area and timing. In Chapter 5, the experimental results are discussed.

The last chapter concludes the overall process and includes suggestions on how to choose the right constant division method as well as possible future work about the subject.

2. PREVIOUS WORK

Division by a constant integer in hardware has always been considered as an issue and the search for a special operator for it has always been an active research area. This is because the systems that require constant division usually need rapid calculations which usually exceed maximum speeds in case a regular division scheme is used. On the other hand, the designer has the advantage of knowing the divisor at compile time, so all the studies are based on this advantage.

The problem has been examined in many studies so far with good surveys (Srinivasan and Petry, 1994; Doran, 1995; Schwarzbacher, 2000), but comparisons of these studies in means of timing, area, or power consumption have been left unclear because of the classification problem of these methods. Every method processes a constant division, but their output formats and effective bit width ranges are different.

2.1 CLASSIFICATION OF CONSTANT DIVISION ALGORITHMS

A DSP system may require constant integer division with a fractional output, having a pre-specified error tolerance. On the other hand, an address mapping operation may require an address generation solution with rapid constant divisions in which case an exact quotient and an exact remainder will be desired as the output. The desired output format of the constant division system is therefore an important property.

Certain methods can be more efficient in a specific input bit-width range, but they lose their effectiveness outside that range. Likewise, there can be better methods for a specific divisor, or for a property that a group of divisors possess.

The performance of each method may differ for the requirements of the process that it will be used in. These requirements are:

i. The format and the error tolerance of the desired output,
ii. The input bit-width of the system (dividend size),
iii. The divisor itself.

Getting a fractional result with an error margin (previously known and accepted) is an easier job than getting exact quotient and remainder results. (i. e. Euclidean division) There are some algorithms that promise to give exact quotients but not exact remainders. The remainders for such systems can be obtained with reverse multiplication by the quotient and subtraction from the dividend; but it will introduce an extra delay to the circuit. Therefore the format of the output is important and must be carefully analyzed when conducting performance comparisons.

As the input bit-width increases the methods tend to become slower. But some methods are targeted for larger bit-widths and some are targeted for small width operations. The performance comparison of an algorithm must be evaluated within a specific input bit-width range for a healthier judgment.

The pre-determined constant divisor itself also has effect on the performance. Some algorithms are applicable for all integer divisors. But some methods are only for a single divisor or a set of divisors that share a specific property. The simplest example to this would be a divisor that is a power of 2. In such a case we would just shift the dividend to right by the amount of power. Another example is the divisors of the form $2^n \pm 1$, which will be mentioned later.

There are many constant division algorithms based on different methods. All have different performance results but they are all suited for different purposes. Figure 2.1 shows the main bases of methods for constant divisors.

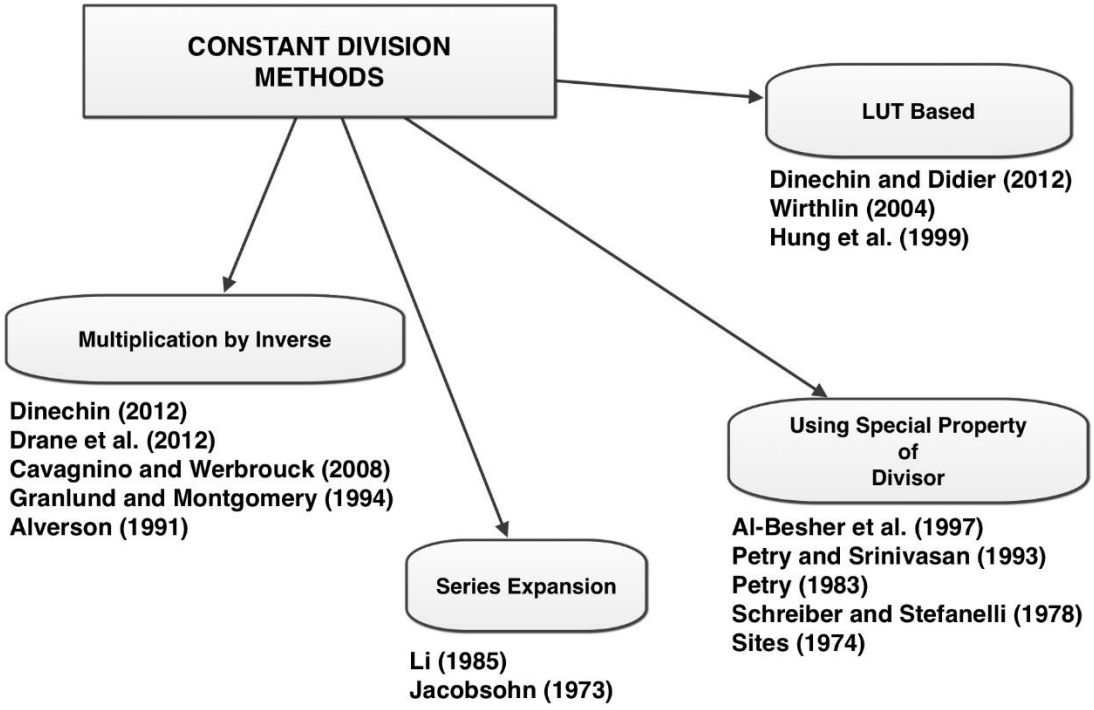

Figure 2.1 Constant Division Methods

Methods that use multiplication by inverse and methods that use series expansion techniques are usually well suited for systems where a fractional result with an error tolerance is acceptable. On the other hand, LUT based systems are good for exact quotient and remainder results with zero error tolerance.

2.2 METHODS

In this section some of the selected works are represented. The works will be briefly summarized and their results will be commented on.

Srinivasan and Petry (1994) presented a survey on constant division algorithms, covering 7 different methods that are shown in Table 2.1. They mostly focused on methods that are applicable to divisors with specific properties, especially the ones in form $2^n \pm 1$ and the divisor 10 which is used for BCD conversion. The main contribution of this work is the suggested comparison technique which is shown in Table 2.1.

Table 2.1 Comparison Table of Srinivasan and Petry
Source: Srinivasan and Petry (1994)

REF.	Divisor Domain	Quotient Determination for nonzero remainders	Remainder Determination	Order of quotient digit generation (high/low-order digits first)
[1]	$d = 10$	Exact divides only	Not computed	Low to high
[2]	$d = 10$	Single subtraction of transformed remainder	Must decode transformed remainder	Low to high
[3]	$2^n \pm 1$	Single subtraction of transformed remainder	Must decode transformed remainder	Low to high
[4]	$2^n \pm 1$	Maximum of d subtractions required	Count of actual subtractions	Low to high
[5]	$2^n - 1$, $d > 0$	No additional steps	R/d computed, multiply by d	High to low
[6]	$d = 10$	No additional steps	R computed directly	High to low
[7]	$2^n \pm 1$	No additional steps	R computed directly	Convergence / High to low

[1] Sites (1974)
[2] Petry (1983)
[3] Johannes et al. (1980)
[4] Artzy et al. (1976)
[5] Jacobsohn (1973)
[6] Schreiber and Stefanelli (1978)
[7] Ci Yun Guei et al. (1985)

The comparison in Table 2.1, compares the methodology of the algorithms. The performance results and comparisons of these algorithms are not considered in the scope of their study.

Another survey study was done by Schwarzbacher (2000) where the constant divider structures with the form $2^n \pm 1$ was examined. However, the methods that are observed in this study are not limited to the methods that are only applicable to divisors in $2^n \pm 1$ form, but LUT based methods are also considered and analyzed. The most important part of this study is that it concludes that the LUT based methods are the fastest among others.

Jacobsohn (1973) approach was based on the multiplication by inverse method. The proposed method requires the inverse of the divisor to be a repeating binary of value less than 1. The results are in the form of a quotient and a remainder, where quotient is an exact integer value while the generation of the remainder is not trivial and has an error tolerance.

Yen and Li (1985) presented an algorithm that turns the technique of approximation via an infinite product into a method of computing the precise value of the quotient without computing the remainder. The remainder than, later be calculated by reverse operation by the quotient. This is of course introducing another delay to the system.

2.2.1 TABLE BASED DIVISION METHOD (TBCD)

Dinechin and Didier (2012) introduced a table based division technique in their paper "Table based division by small integer constants". The algorithm can be operated in different radices and at each iteration a portion of the dividend is calculated with a partial dividend, partial remainder and some digits of the quotient. It outputs an exact integer binary quotient and an exact binary integer remainder, which in our case is a competitor with our BTCD. We will compare our performance with TBCD method, since it is the most recent and fastest LUT based method, aimed to achieve a fast operation.

The method is an adaptation of the regular paper and pencil algorithm in case of small divisors. An example of paper – pencil division is shown in Figure 3.1. The algorithm outputs results q and r_0 of the high radix Euclidean division of the dividend x by the constant d. At each step, the partial dividend y_i, the partial remainder r_i and one radix $2^{\propto}$ digit of the quotient is computed. The alpha constant $\propto$, determines the radix that the operation will be conducted. The steps of the algorithm can be observed in Figure 2.2.

Algorithm 1 LUT-based computation of x/d

procedure CONSTANTDIV(x, d)
  $r_k \leftarrow 0$
  for $i = k - 1$ **down to** 0 **do**
    $y_i \leftarrow x_i + 2^{\alpha} r_{i+1}$ (this $+$ is a concatenation)
    $(q_i, r_i) \leftarrow (\lfloor y_i/d \rfloor,\ y_i \mod d)$ (read from a table)
  end for
  return $q = \sum_{i=0}^{k} q_i . 2^{-\alpha i}$, r_0
end procedure

Figure 2.2 The Steps of TBCD Algorithm
Source: Dinechin and Didier (2012)

The partial results are stored in LUTs to avoid the actual division calculation. The timing complexity grows linearly with the dividend bit-width N. If we increase the radix, we get less calculation steps, therefore we get a faster circuit while the chip area will be badly affected.

3. BINARY TREE BASED CONSTANT DIVISION (BTCD)

3.1 BASIC IDEA AND APPROACH

The motivation behind our work is to achieve a fast constant division method that is especially reasonable for dividends with large bit widths. Our aim is to be able to run simultaneous calculations on different bit portions of the dividend with our “divide and conquer” approach and to take advantage of the LUTs for each calculation so that we can avoid the burden of division. Therefore, we will achieve a fast constant integer division circuit with a reasonable use of the chip resources.

In a constant division system, the only input parameter is the dividend. This limits the number of possible input combinations to the system by only the possible values of the dividend. We can consider this as our advantage. For small dividend bit-widths, we can store all possible results in a single LUT, but as the bit-width of the dividend gets larger, this “brute force” method becomes inefficient since both the occupied chip area and the access time of the LUT increase.

Our approach is to partition the dividend into chunks of bits and treat each chunk as a separate dividend. We create LUTs to store all possible results for each chunk and combine them with a binary tree like structure to have the final result of the division. This way we make the resource consumption reasonable while achieving the best timing for large input bit-widths.

If we process the division from the most significant chunk to the least significant one as we do in regular division techniques, we lose the advantage of using LUTs since each LUT will have to wait for the outcome of its leftmost chunk. To overcome this issue, we have introduced a new parallelized division method where these each chunk is calculated simultaneously. With this new division method, we aimed to reduce timing complexity and have better use of the LUTs.

3.2 A NEW PARALLEL DIVISION METHOD

In some division methods, like the regular paper and pencil division, the operation is carried digit by digit starting from the leftmost one. At each division of the dividend digits, a partial remainder and a partial quotient are gathered. The partial quotients are used to construct the digits of the final quotient, while the partial remainders are inputs for the calculation of the next digits. Figure 3.1 shows a sample process of a regular paper and pencil division in base 10.

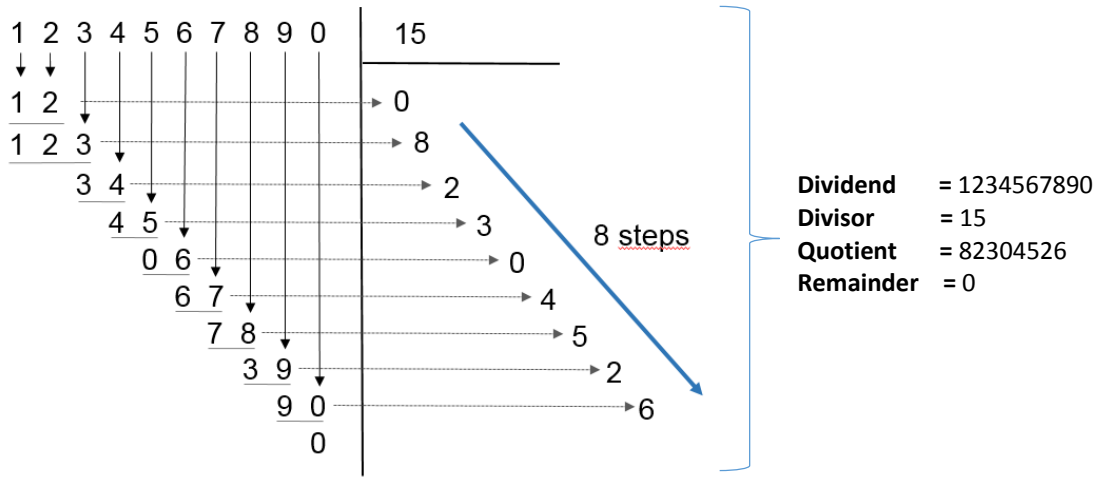

Figure 3.1 Base 10 Regular Division Example

The number of steps to reach the result for this method is 8 which is calculated as:

$$Number\ of\ steps = Number\ of\ dividend\ digits - Number\ of\ divisor\ digits$$

Dinechin and Didier (2012) used LUTs to implement the same paper and pencil division that is shown in Figure 3.1. They pre-calculated each possible division result for each step and stored them inside LUTs in compile time.

This method looks easier from a human point of view since small portion of digits are calculated at each step. But for the sake of ease, we make the less significant bits of the dividend wait until the results from the leftmost bits to arrive. This is a waste of time from hardware point of view.

We propose a new method that uses divide and conquer approach by partitioning the dividend into chunks of digits beforehand. The method simultaneously runs calculations on every bit chunk of the dividend. The quotient and remainder outputs for each chunk are the partial results and processed at the same time. The chunks are then combined 2 by 2 at each step to create the final result. Figure 3.2 shows this method with same variables as in Figure 3.1 for easier comparison.

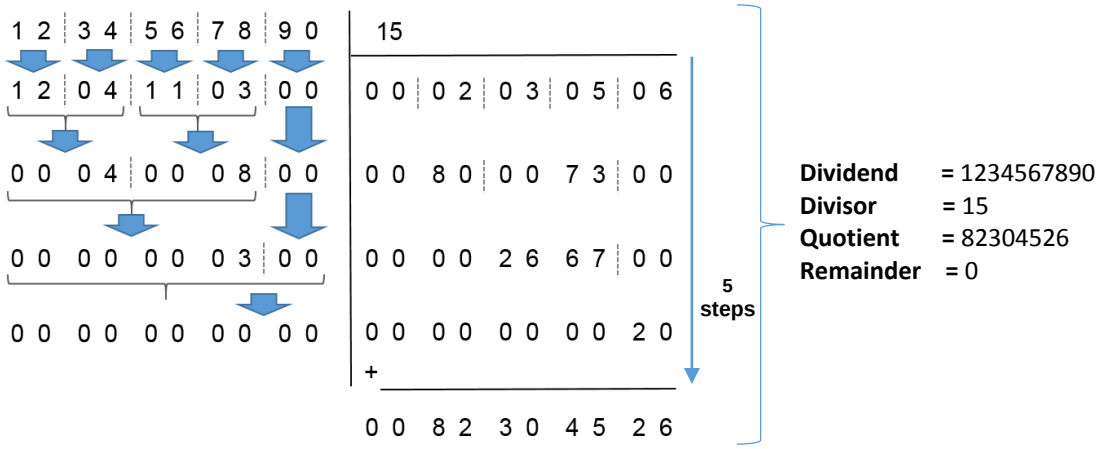

Figure 3.2 Parallelized Implementation of the Regular Division

In Figure 3.2, the dividend is partitioned into 5 chunks, each having 2 digits. Each partition is treated as a separate dividend and the resulting quotients and remainders from their division by 15 is written in the first line. In the following steps the remainders on the left side are grouped 2 by 2 to construct other quotients until a single remainder that is smaller than 15 is reached. For the last step the resulting quotients on the right side are added to get the final quotient.

For human point of view, this method is a complex one. It requires simultaneous calculations of each chunk of bits at each step. But from a computer point of view, we can pre-calculate results for each bit chunk of the dividend and store them inside LUTs. Furthermore, at each combination step, we can fill up transfer LUTs that store the results for possible remainder combinations from the 2 parent LUTs.

This new division method gives us the opportunity of having parallel calculations which reduces the timing complexity and provides a base for using reasonable sized LUTs to avoid the burden of actual division operation.

In the case of parallelized implementation of the regular division, the number of calculation steps are calculated as:

$$Number\ of\ steps = ceil\big(log_2(\#\ of\ chunks)\big) + 2$$

The timing complexities and detailed explanations on the implementation are discussed in Chapter 3.4.

3.3 THE ALGORITHM

In an Euclidean division, the division of the number A, by the number d has the results Q and R, having the mathematical relation shown in Eq. 3.1.

$$\frac{A}{d} = Q + \frac{R}{d} \tag{3.1}$$

In Eq. 3.1, the values A, Q, R, d are nonnegative integers and d is also nonzero. They are all unsigned binary numbers (base 2), since we use unsigned arithmetic in our implementations. In constant division, the divisor d and the bit-width of the dividend, N is known beforehand and they are constants.

Assume A is composed of k chunks of n_l bits where, $k - 1 > l > 0$ and $l \in \mathrm{Z}^+$. Each chunk C_l has an arbitrary portion of the dividend as shown in Figure 3.3. The partition may be done from most significant bit to least significant or vice versa.

$$A = \underbrace{a_{N-1}\, a_{N-2} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{C_{k-1}} \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{} \underbrace{a_{n_1+n_0-1} \ldots\ldots\ldots\ldots}_{C_1} \underbrace{a_{n_0-1} \ldots\ldots\ldots a_1\, a_0}_{C_0}$$

Figure 3.3 Chunks of the dividend

The total bit-width of the input dividend is simply the sum of all chunk widths

$$N = \sum_{l=0}^{k-1} n_l \tag{3.2}$$

where, $N \geq n_l > 0$ and $n_l \in \mathrm{Z}$.

Each chunk of bits C_l, is an n_l bits wide binary. The dividend A can be reconstructed by these bit groups with the following equation:

$$A = C_0 + \sum_{j=1}^{k-1} C_j \, 2^{(\sum_{l=0}^{j-1} n_l)} \tag{3.3}$$

We treat each chunk of bits C_l as a separate dividend. The chunks can be expressed in terms of the divisor d, partial quotient Q_l and a partial remainder R_l as in Eq. 3.4.

$$C_l = Q_l d + R_l \tag{3.4}$$

To reach to the final quotient and remainder, these partial results for each chunk must be combined altogether. The combination of 2 neighboring chunks can be visualized as in Figure 3.4 which shows the group A' and the chunks C_0 and C_1 that generate it

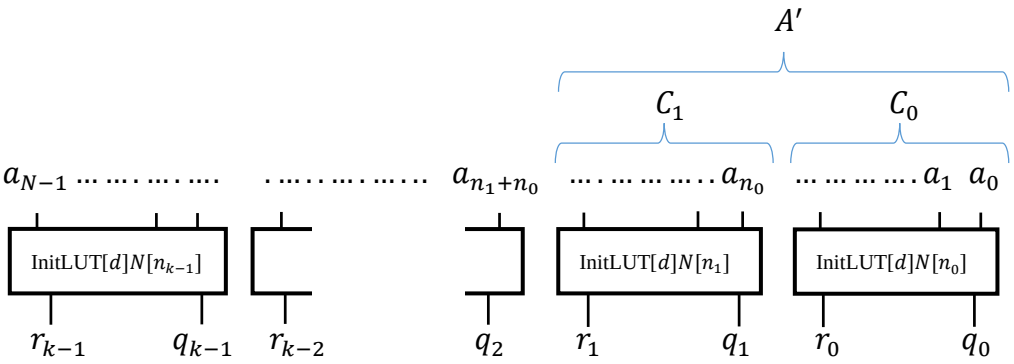

Figure 3.4 Combination of 2 Chunks

Mathematically A' can be expressed as in Eq. 3.5 since the order of the most significant chunk will be higher.

$$A' = C_1 2^{n_0} + C_0 \tag{3.5}$$

Assume that the division of $A'$with the constant divisor d, has the results Q' and R'. By placing Eq. 3.5 in Eq. 3.1, we construct Eq. 3.6;

$$\frac{A'}{d} = Q' + \frac{R'}{d} = (\frac{C_1}{d})2^{n_0} + \frac{C_0}{d} \quad \textbf{(3.6)}$$

Replacing the C_1 and C_0 terms in Eq. 3.6 with their values in Eq. 3.4 we express the equation in terms of the partial results as in Eq. 3.7

$$Q' + \frac{R'}{d} = (\, Q_0 + Q_1 2^{n_0}\,) + (\,\frac{R_1 2^{n_0} + R_0}{d}\,) \quad \textbf{(3.7)}$$

The partial results Q_0, R_0, Q_1 and R_1 are pre-calculated values that are stored in memory blocks called the initial LUTs. The details of the implementation will be given in the next section.

In Eq. 3.7, the term $R_1 2^{n_0} + R_0$ has to be further examined. It is possible that the result of this addition will be a larger number than the divisor d, therefore it is best to represent this term as in Eq. 3.8.

$$\frac{R_1 2^{n_0} + R_0}{d} = \frac{Q_2 d + R'}{d} = Q_2 + \frac{R'}{d} \quad \textbf{(3.8)}$$

To prove the validity of this representation in Eq. 3.8 we can check the summation on the maximum possible values of the remainders. Notice that the maximum value of R_1 is equal to the maximum value of R_0 and equals $d - 1$ since both are remainders after division by d. The maximum value of the term $R_1 2^{n_0} + R_0$ is given as

$$\max(R_1 2^{n_0} + R_0) = R_{1_{max}} 2^{n_0} + R_{0_{max}} \quad \textbf{(3.9)}$$

$$= (2^{n_0} + 1)d - 2$$

where, $n_0 \geq 1$, so for any divisor $d \geq 2$, the maximum value of the expression is greater than the divisor d.

So placing Eq. 3.8 in Eq. 3.7 we have;

$$Q' + \frac{R'}{d} = (Q_0 + Q_1 2^{n_0} + Q_2) + \frac{R'}{d} \quad \textbf{(3.10)}$$

The values Q_2 and R' are pre-calculated and stored in another kind of LUT blocks which we call transfer LUTs. The details of transfer LUTs will be discussed in the next section along with the initial LUTs.

Eq. 3.10 clearly shows that the quotient Q' in above equation can be reached after a shift operation and an addition operation of $Q_0, Q_1,$ and Q_2. Notice that to get the final quotient we only process a single addition operation since $Q_0 + Q_1 2^{n_0}$ is a simple concatenation because their digits do not overlap.

To reach the final results we have to treat the combined chunks A' as the new chunks and do the combination process until we reach a single group containing all N bits of the dividend.

3.4 HARDWARE IMPLEMENTATION

The constant division system should be constructed such that it will take dividend A, from its N_A bit input port, carry the division process by d in its pre-configured combinational hardware and output the results as an $N_{Q_{max}}$ bit quotient Q, and an $N_{R_{max}}$ bit remainder R. In our case we are not interested in any fractional results since our focus is the Euclidean division. The system is outlined in Figure 3.5.

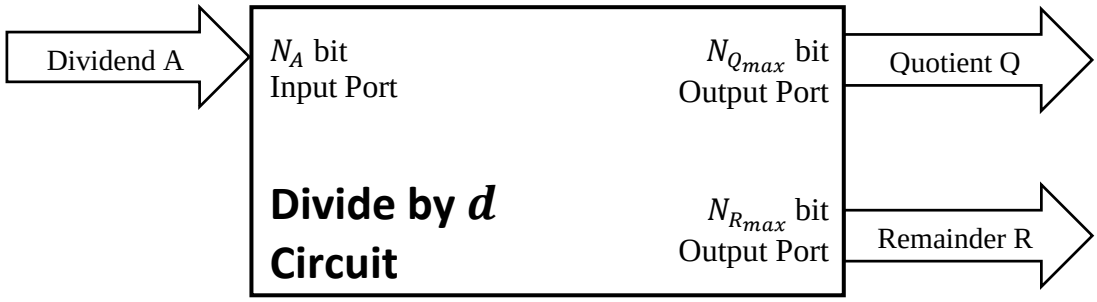

Figure 3.5 Block model of the division by d circuit

The basis of our method is to take advantage of knowing the divisor beforehand by simply pre-calculating all possible input combinations at compile time and store them inside a LUT. That way we can save the time by not actually running calculations for the laborious division operation during run time. Instead we will only experience the access delay of the LUT as the only factor for timing.

The size of this LUT will be calculated as

$$2^{N_A}(N_{Q_{max}} + N_{R_{max}}) \quad \textbf{(3.11)}$$

Where N_A is the input dividend bit length and $N_{Q_{max}}$, $N_{R_{max}}$ are the bit lengths of the maximum possible quotient and remainder respectively. Notice that this is a brute force use of the LUT method and as the dividend bit length gets larger, both the access time of the LUT and the occupied area will dramatically increase and cause an inefficient consumption of the chip resources.

Figure 3.6 shows the exponential increase of the number of stored bits in the LUT as the bit-width gets larger. (for d = 3)

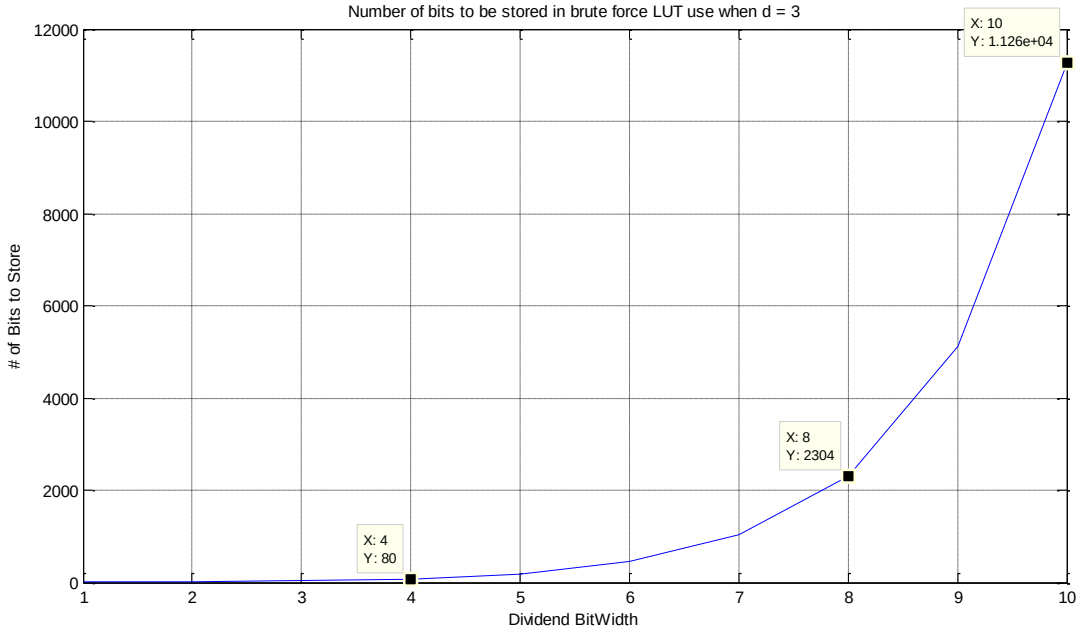

Figure 3.6 Number of Stored Bits for different Bit-widths

Most systems that require a constant division operate on dividends of large bit widths, usually greater than 16 bits. Due to the excessive amount of bits to store for such bit widths, brute force LUT method cannot be used, or simply is not preferred.

Our algorithm suggests that by partitioning the dividend into chunks of bits and storing all possible results of the division for each group, we can reduce the total storage area and with a binary tree like combination method, we can reconstruct the final results Q and R.

3.4.1 PROPERLY PARTITIONING THE DIVIDEND

The first step of the design is to properly partition the dividend A into chunks of bits. The algorithm is valid for any combination of partitioning, but selection of the chunk widths will affect the overall performance with trade-offs between timing and area consumption.

If we select the chunks widths too small, then the resulting binary tree in the combination process will have more calculation steps until it reaches the final results. On the other hand, selecting large chunk widths will result in bigger LUTs for storing possible partial results of each chunk therefore we will have a larger chip area. The number of calculation steps will be reduced but since the LUTs will get bigger, their access time will be worse.

We have selected the chunk sizes to be 4 bits each for general purpose. However, we assign a maximum initial chunk width constraint, n_{max}, which is used when there are some remaining ungrouped bits after the partition. This remaining bit (or bits) is added to the least significant chunk if doing so will decrease the number of calculation steps and if n_{max} will not be exceeded for any chunk. Figure 3.7 shows a case where optimization is held reducing the number of calculation steps for N = 9.

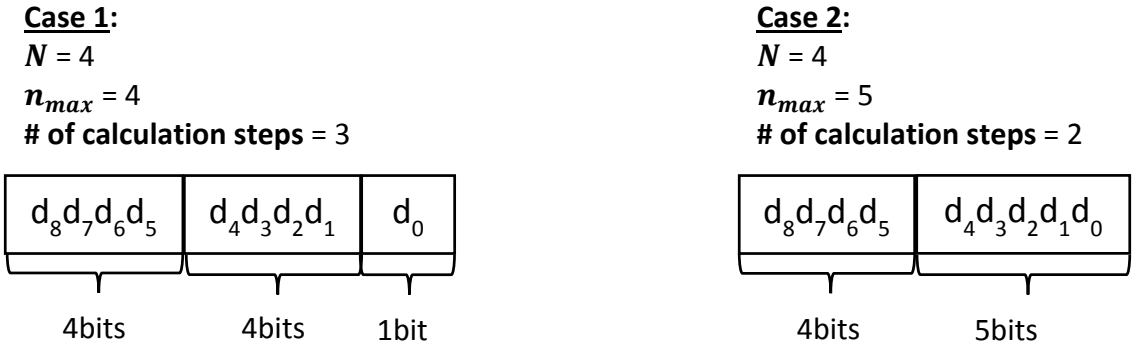

Figure 3.7 Optimization for Dividend Partitioning

3.4.2 INITIAL LUT BLOCKS (iLUT)

We treat each chunk of bits as a separate dividend and construct LUTs for each one. Each LUT stores the pre-calculated results of division by d for the whole range of possible bit combinations of that chunk. We call these the initial LUTs (iLUTs). For each chunk of bits C_l, there exists an iLUT which takes the bits of that group as its input. Figure 3.4 shows the initial LUTs with the corresponding chunks. Figure 3.8 shows the structure of an iLUT.

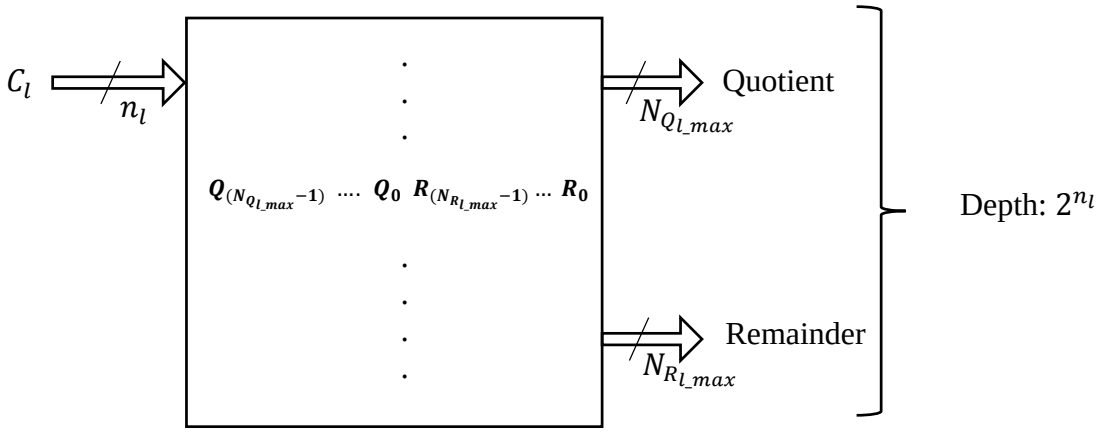

Figure 3.8 Initial LUT Block

The number of bits that are stored in each iLUT is given in Eq. 3.12,

$$size(\ iLUT_l\ ) = 2^{n_l}(\ N_{Q_{l\_max}} + \ N_{R_{l\_max}}) \qquad \textbf{(3.12)}$$

where , $N_{Q_{l\_max}}$ and $N_{R_{l\_max}}$ are the bit lengths of the maximum possible quotient and remainder.

3.4.3 COMBINING THE CHUNKS

The combination process of the partial results are done by combining 2 parent LUTs at a time. The combined block (cBLK) is then treated as the parent LUT for the next combining step. Each cBLK includes 2 parent LUTs, 1 transfer LUT (tLUT), an addition operation and a shift operation. cBLKs are generated at each step until a single cBLK covering the whole input bit-width is reached. The mathematical background for the combination of 2 chunks is explained in Section 3.3.

These combination steps construct a binary tree like structure as shown in figure 3.9.

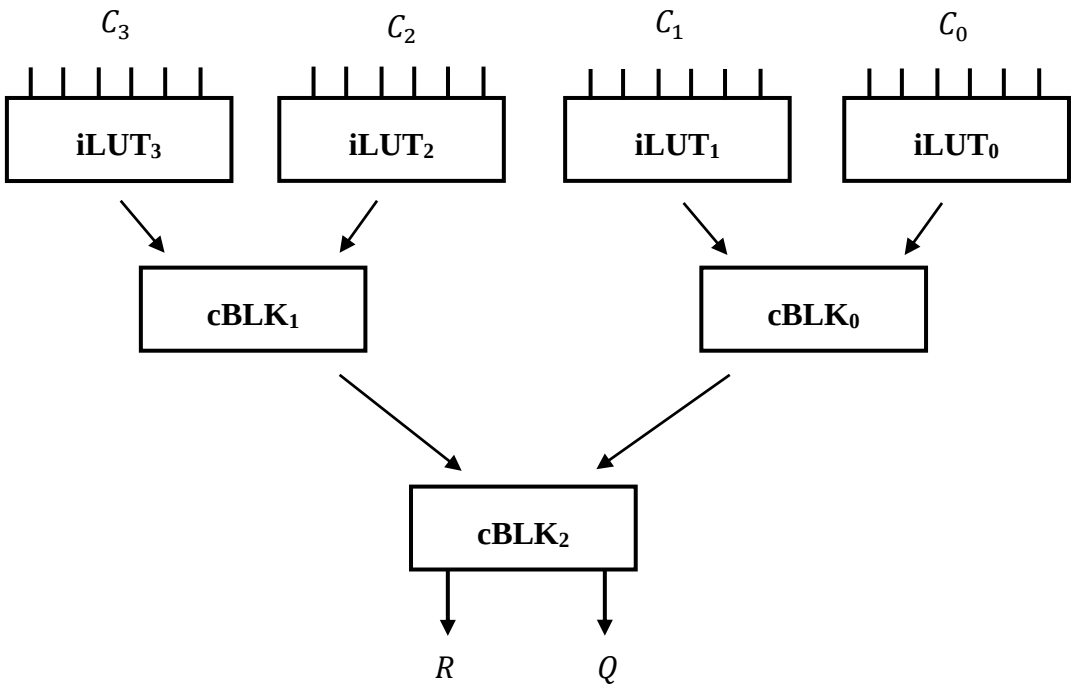

Figure 3.9 The Binary Tree Based Combination

Focusing on each combination step we observe 3 basic blocks, an addition and a shift operation. This can be observed in Figure 3.11 where the overall process of division with 16 bit input is shown.

The 3 basic blocks of the implementation are:

i. Parent LUTs

Parent LUTs store the partial results for each chunk that is being combined. Initial LUTs, that are explained above are the first parent LUTs in the beginning of the binary tree. The combined blocks are then treated as the parent LUTs for the following steps.

ii. Transfer LUTs (tLUT)

When combining the parent LUTs, the remainders from each LUT construct a value $R_1 2^{n_0} + R_0$ which is explained in Eq. 3.8. This value is presented as $Q_2 d + R'$ in which case the calculation of the values Q_2 and R' is also another division operation.

tLUTs store all possible values of Q_2 and R' so that there won't be any time lost for their calculation. The block diagram of a tLUT is shown in Figure 3.10.

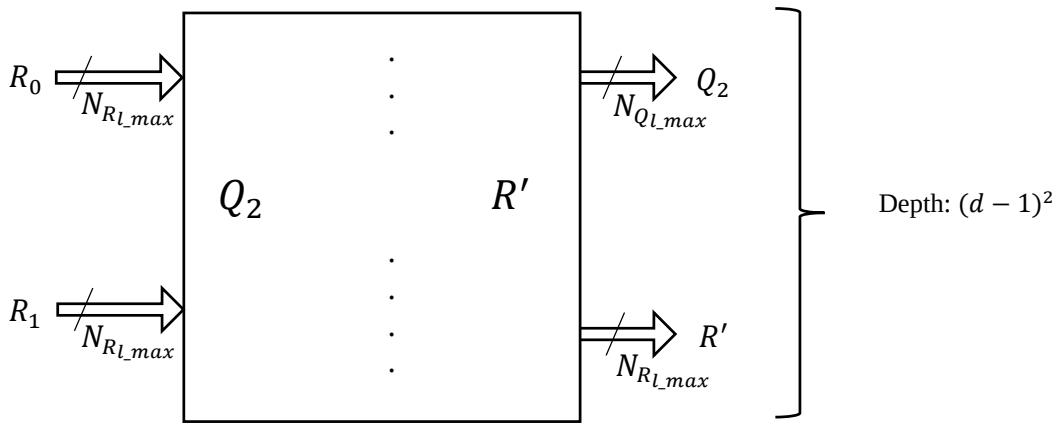

Figure 3.10 The transfer LUT

The size of the transfer block is calculated as in equation 3.13.

$$size(\, tLUT_l \,) = d^2 (\, N_{Q_{2\_max}} + \, N_{d-1}) \qquad \textbf{(3.13)}$$

The important thing here is that we don't have to cover the whole input range since the input is simply the concatenation of the remainders R_0 and R_1. Therefore the depth of the tLUT is d^2.

iii. Combined Blocks (cBLK)

A Combined block is the resulting block after combination of 2 parent LUTs. They include 2 parent LUTs, 1 transfer LUT, an addition and a shift operation inside. Figure 3.11 shows the hardware for division of a 16-bit input, where 3 combined blocks, XN4, XN8, and XN16 are presented. (In this notation, X is the divisor and 4,8 and 16 are dividend bit-widths.)

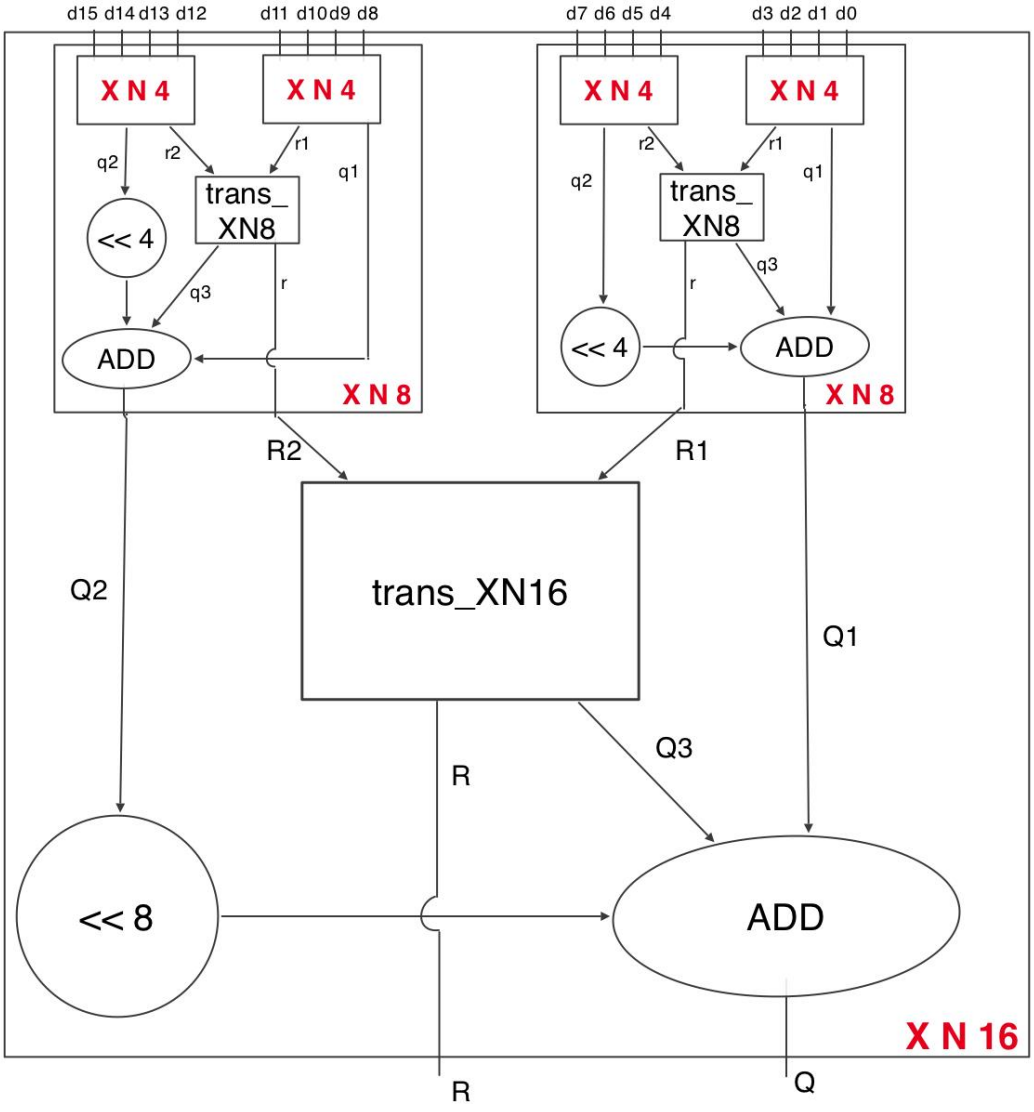

Figure 3.11 The generation of the XN16 cBLK

4. RTL GENERATION AND SYNTHESIS METHODOLOGY

The theoretical description of BTCD method has been explained in the previous section. The next step is to implement the method to validate our claim.

This chapter issues the methodology for implementation, verification, and synthesis of BTCD method as well as the competitor; TBCD method, to prove their applicability and to compare their performance with each other. TBCD method is selected as the closest competitor since it is the most recent study that takes advantage of LUTs. We only compared our results with LUT based works since the timing performance of other methods are proven to be much worse as Schwarzbacher (2000) has also suggested in his survey.

Our design flow starts with proper creation of the RTL design and ends with the extraction of synthesis results. At each step we have implemented detailed mechanisms to catch possible errors and create error logs. Another important thing about the design flow is that it is completely automated with a single script, which processes the overall flow for a range of given divisors and for a range of input bit widths for each divisor. The flow diagram is shown in Figure 4.1.

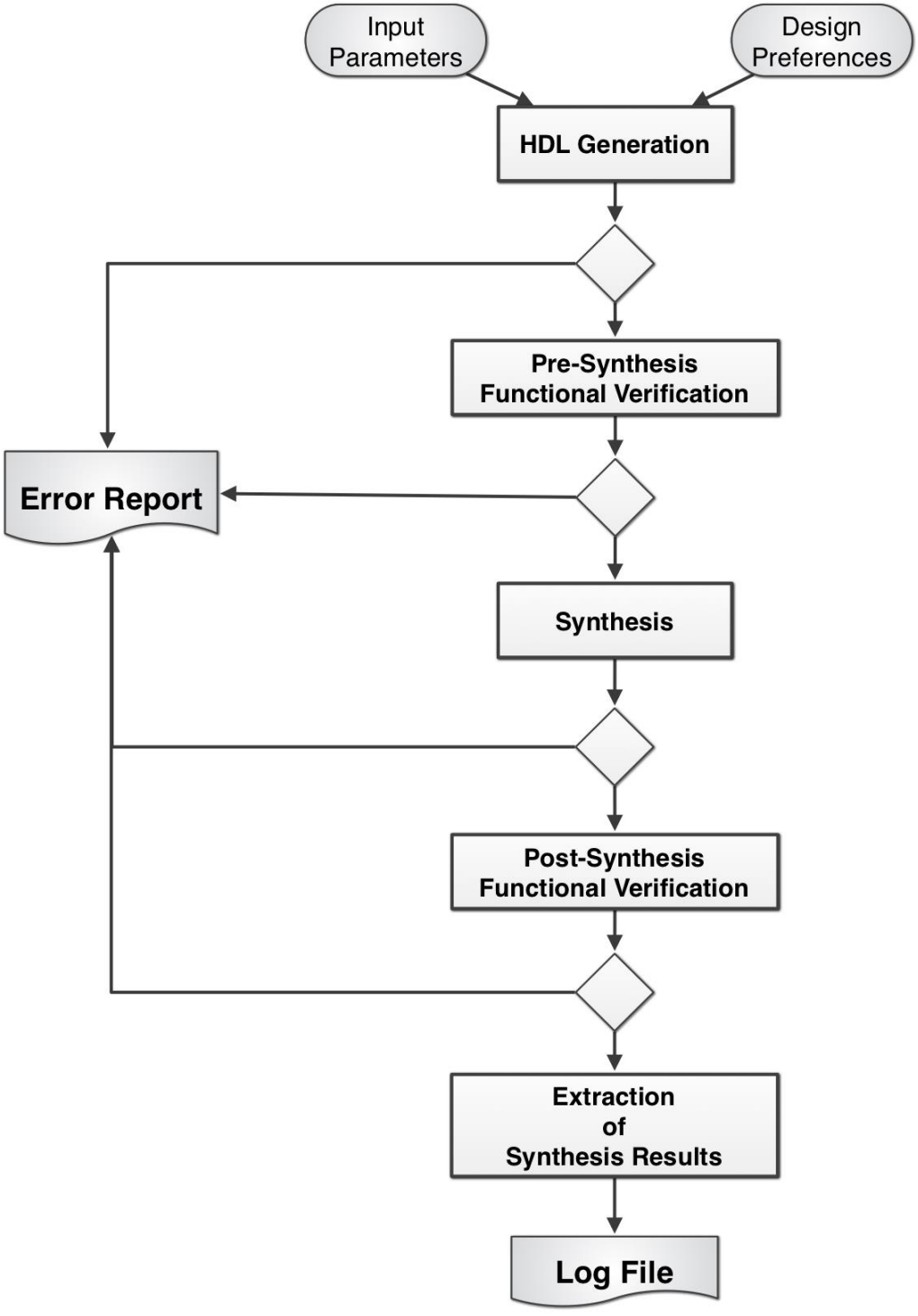

Figure 4.1 Design and Synthesis Flow

4.1 HDL GENERATION

RTL of the designs are described in Verilog HDL. While implementing constant divisor systems as in Figure 3.2, each combination of the 2 parameters, constant divisor d and the input bit width N, the resulting RTL varies. (Different LUTs are constructed, output q and r bit-widths change, etc.) Besides, there are design parameters for each implementation such as the number of bits in partitioned chunks of the dividend, or the alpha constant ($\propto$) in TBCD method. These parameters cause performance tradeoffs in the design, altering the RTL at each variation.

Manually writing a Verilog code for every combination of design specifications would be extremely time consuming. Since the LUT based designs require pre-calculations for filling the contents of the LUTs, getting each result and coding them inside the RTL one by one would be painful. In addition, it would make the design more error prone since there wouldn't be any standardization, so more human errors would be likely to interfere.

Instead of manually coding the design RTL in Verilog, we write HDL generator scripts that generate Verilog sources for given parameters. The HDL generators are written in Perl scripting language.

HDL generators are unique for each method. They are dedicated to create the RTL of the system in Figure 3.2 with the method that it is coded for. The input arguments for the HDL generator scripts are the constant divisor d, the dividend bit width N and the design parameters. The divisor and the input bit-width are necessary for system calls of each HDL generator script. The design parameters, however, are unique for each method and they specify how the method will function. These design parameters are assigned a default value so if they are not specified in the system call, the default values are used.

The generated files are:

Design RTL: Verilog code describing the RTL for the given inputs.
The naming format of the file is as: *[methodName]_[divisor]_N_[dividendBitwidth].v*

Testbench: Verilog code for functional verification. It feeds all possible inputs to the design and compares the results.
The naming format is as: *tb_[methodName]_[divisor]_[dividendBitwidth].v*

Wrapper: Verilog code for synthesis operation. It instantiates the design between flip-flops so that all top module inputs are fed from flip-flops and all top model outputs are sunk by flip-flops.
The naming format is as: *wrapper_[divisor]_[dividendBitwidth].v*

Log File: After each run of the HDL Generator weather the generation is successful of not, a log file is created. This log file explains errors if the generation was not successful. After a successful generation, it includes explanations on the design such as the number of total stored bits in LUTs and selected design preferences in the implementation.
The naming format is as: *[methodName]_[divisor]_[dividendBitwidth].log*

The inputs and the generated files are shown in Figure 4.2.

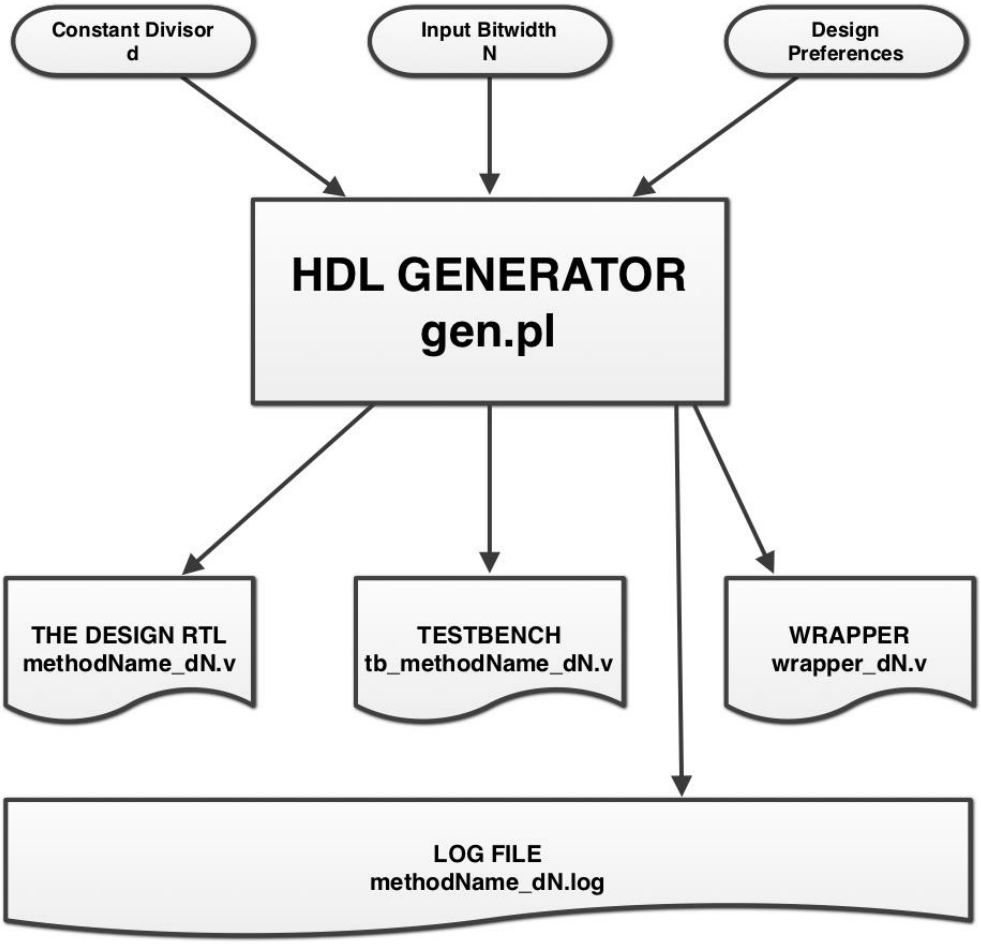

Figure 4.2 The HDL Generator

4.2 PRE-SYNTHESIS FUNCTIONAL VERIFICATION

The generated RTL should be functionally verified before the running the synthesis operation. Usually the HDL generators provide a standardization in the generated RTL and once a design generated with an HDL generator passes the functional verification, all variations generated by the same HDL are supposed to pass. However, this is not the case, since some bugs may not appear for certain set of parameters while they do for others. We functionally verify all our RTL code before synthesis in order to ensure functional correctness.

The testbench file that is generated by HDL generator is used for pre-synthesis verification. The verification script runs the testbench on Icarus Verilog tool. Reference outputs are generated in the testbench to compare results with the design under test (DUT). The test input generator mechanism generates all possible inputs if the range is reasonable. For the DUTs with larger input bit widths, the testbench generates random test inputs to save time.

If the design is verified for all test inputs, the verification script returns ‘1’ to signal that we can proceed to the synthesis operation. In case of an unexpected behavior, the script returns ‘0’ and creates a log file which identifies the details on the error(s).

4.3 SYNTHESIS

The synthesis tool we use is Synopsys DC version 2010.12. Our synthesis script is written in TCL scripting language. Perl scripts are used to invoke Synopsys DC, run the TCL synthesis script and gather the outputs.

In the synthesis process, our aim is to obtain the circuit with the smallest possible clock period. To do this, we supply the synthesis tool a desired target value for the clock period. For the best results, the target value should be selected close to the best value. Unfortunately, we cannot know the optimum clock period beforehand.

To solve this problem we use a binary search based iterative synthesis script. It synthesizes each design 4 times in total in an effort to get the best possible timing result regardless of the area constraint. In the initial synthesis the target clock period is set to 0.1 ns (which corresponds to 10 GHz and is impossible to meet) to see how close the design can reach this value. For the second and third iterations, the largest period that is failed to achieve so far and the achieved clock of the previous iteration is averaged and set as the new desired clock target. The last iteration is to check if any minor adjustments are available to the timing that has met. The last met clock is multiplied by 0.9 and set as the target clock. After all iterations are finished, the best achieved clock period, area and the resulting netlist files are logged. Our experience showed us that if we process more than 4 iterations we cannot improve the timing more, instead

we lose a lot of our computation time since we run the synthesis for many different combinations of the inputs.

The combinational RTL is instantiated inside a wrapper module before the synthesis. This wrapper consists of input and output flip-flops (FFs). The inputs are flopped before being supplied to our combinational logic cloud, and the outputs of our block are also flopped. The start point of a path is clock input of the initial FFs and the end point is the data input of the output FFs.

To obtain timing results, the synthesis tool runs a static timing analysis which checks the timing of all possible paths. When data are launched at the start point, the time elapsed until the end point determines the timing of a path. The path with the largest arrival data travel time is called the critical path and it defines the timing of the whole circuit.

The synthesis script is targeted for obtaining the best possible timing, so the area results are the corresponding areas for the circuits with the best possible timings. Also, since we use the wrapper module only for timing purposes, the synthesis script does not report the area for the FFs of the wrapper. For each iteration, our iterative synthesis script only reports the area of our design RTL in square microns.

4.4 POST-SYNTHESIS VERIFICATION

After the synthesis, we obtain a gate-level netlist that corresponds to the same RTL with our design. The gate-level netlist consists of cells of our standard cell library constructing the RTL that is described in the original design file. Verifying this design is important to check the output of the synthesis tool. The same method described in Section 4.2 is applied replacing the DUT with the gate level netlist.

5. EXPERIMENTAL WORK

In this chapter the results and comparisons will be presented. As our competitor, we implemented Dinechin and Didier's work (TBCD) presented in paper "Table based division by small integer constants" in 2012. The method is described in detail in Chapter 2. We have successfully implemented TBCD method, coded a generator for it and put it through our synthesis design flow.

5.1 EXPERIMENTAL SETUP

Synthesis results are not the same for every test environment. Along with the results, the environment variables that the tests had been conducted with has to me mentioned. These variables are;

i. The synthesis tool and its version,
ii. Synthesis parameters,
iii. Target technology,
iv. The cell library,
v. The wire load model (WLM) that is used.

In our experimental work, we used Synopsys Design Compiler (DC) version 2010.12 with ARM-Artisan TSMC 180 nm worst-case standard-cell library and a WLM for wire load and wire propagation delay considerations.

We used the Verilog HDL for our implementation. The Verilog code is generated by HDL generators written in Perl scripting language. Also the automation process for generation, verification, and synthesis are controlled by Perl scripts. For the functional simulations and other verifications, we used Icarus Verilog.

5.2 RESULTS AND COMPARISON

The comparisons are held with the TBCD method and our BTCD method. The reason that we have chosen the TBCD method as our competitor is its similarities to our work. This method is also LUT based and outputs exact quotient and remainder results. Also the method can be applied to all integer divisors without limitation as ours.

We carefully examined the TBCD method and successfully implemented it. We have written HDL generators that also generates testbenches along with the code. Scripts for generation, verification, and synthesis follow the same methodology with our own work.

The experiments cover the divisor range [3,15]. For each divisor, the dividend bit-widths of range [4, 24] are covered so that we can compare results with small dividend bit-widths to the ones with large input bit-widths. Important point here is that the divisors 2, 4, and 8 are inside our divisor range, but they are excluded because of the obvious reason that they are powers of 2 and binary divisions with such numbers can simply be accomplished by a shift operation to right side

The discussed results in this sections are for the divisors 3, 10, and 15. The results for the whole range can be found in Appendix A.

5.2.1 TIMING RESULTS

The timing comparisons have the key importance since division by a constant is a fundamental problem that demands speed. Our synthesis scripts are targeted to achieve the best possible timing for the two methods without any concern on the area constraints. Therefore, the timing results are the minimum available clock periods in nanoseconds regardless of any area concern.

The timing values are for FF-to-FF values since all inputs are fed from FFs and all outputs are sunk by FFs. The resulting values therefore include clock-to-Q delays and setup times of the FFs with the delay of the critical path in our design.

Timing results for the smallest integer divisor in our range, 3 are presented in Table 5.1. It shows the best achieved timing results for both our BTCD method and TBCD method when divisor is selected as 3. The notation, [divisor]N[dividend_bitwidth], in the leftmost column represents a unique constant division circuit.

Table 5.1 The Best Clock Periods for d = 3

	TIMING RESULTS (ns) for d = 3		
	Our BTCD Method	**TBCD Method**	**Our improvement**
3N4	**1,08**	1,13	4,43%
3N5	**1,36**	1,36	0,37%
3N6	1,86	**1,60**	-16,20%
3N7	2,01	**1,79**	-11,98%
3N8	2,13	**2,02**	-5,45%
3N9	**2,13**	2,18	2,20%
3N10	2,81	**2,44**	-14,81%
3N11	2,84	**2,68**	-5,97%
3N12	2,95	**2,90**	-1,72%
3N13	**2,98**	3,09	3,40%
3N14	**3,00**	3,35	10,45%
3N15	**3,09**	3,52	12,07%
3N16	**3,33**	3,84	13,19%
3N17	**3,33**	3,95	15,70%
3N18	**4,11**	4,29	4,01%
3N19	**4,21**	4,59	8,28%
3N20	**4,19**	4,76	11,97%
3N21	**4,14**	5,02	17,45%
3N22	**4,17**	5,14	18,87%
3N23	**4,20**	5,36	21,60%
3N24	**4,09**	5,76	29,09%

The best timings at each row is marked bold. The improvements column in Table 5.1 shows the percentage of improvement of our algorithm over the TBCD method.

Table 5.1 shows that when the input bit-width is 13 bits or larger, we have better results and as the bit-width gets larger, our timing performance gets much better than our competitor. For a 24-bit input dividend, our method can achieve a clock period of 4,09 ns (which corresponds to 245 MHz clock frequency) while TBCD method remains at 5,76 ns (174 MHz).

To better visualize the variations of clock periods versus the increasing bit-width values, the graph in Figure 5.1 can be observed.

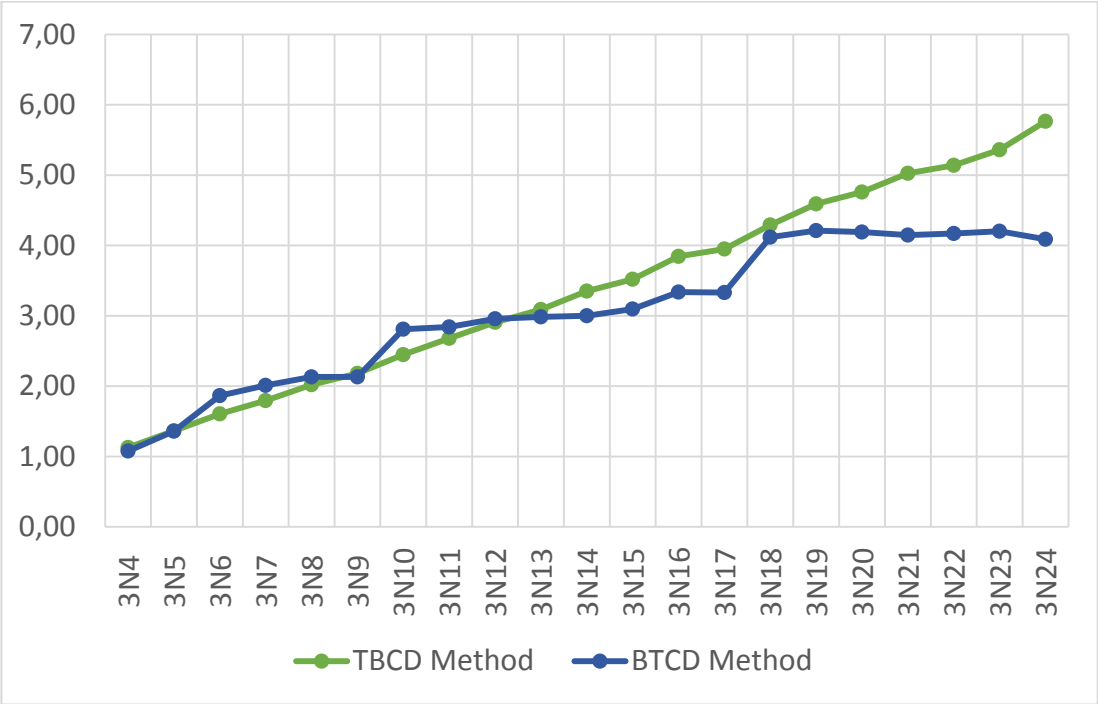

Figure 5.1 Dividend Bit-widths versus Timings (ns) for d=3

The second observations are for d = 10. The timing results for d = 10 is given in Table 5.2. The important point here is that we have better results for bit-widths larger than 9 for this case. The improvement rate for larger bit-widths are also increased.

Table 5.2 The Best Clock Periods for d = 10

	TIMING RESULTS (ns) for d = 10		
	Our BTCD Method	**TBCD Method**	**Our improvement**
10N4	0,74	0,74	0,00%
10N5	1,15	**1,03**	-11,65%
10N6	1,40	**1,38**	-1,22%
10N7	**1,57**	1,67	5,69%
10N8	2,08	**1,99**	-4,92%
10N9	**2,13**	2,23	4,58%
10N10	**2,36**	2,59	9,06%
10N11	**2,57**	2,91	11,51%
10N12	**3,09**	3,16	2,13%
10N13	**3,17**	3,42	7,37%
10N14	**3,24**	3,85	15,71%
10N15	**3,28**	4,08	19,51%
10N16	**3,47**	4,43	21,67%
10N17	**3,49**	4,72	26,12%
10N18	**3,44**	4,98	30,82%
10N19	**3,50**	5,22	33,05%
10N20	**4,35**	5,63	22,77%
10N21	**4,46**	5,91	24,48%
10N22	**4,38**	6,22	29,55%
10N23	**4,43**	6,50	31,75%
10N24	**4,61**	6,76	31,75%

Figure 5.2 shows clearly that for the divisor 10, we have achieved better results for smaller bit-widths than we have achieved when d = 3, and the improvements on large bit-widths are also increased.

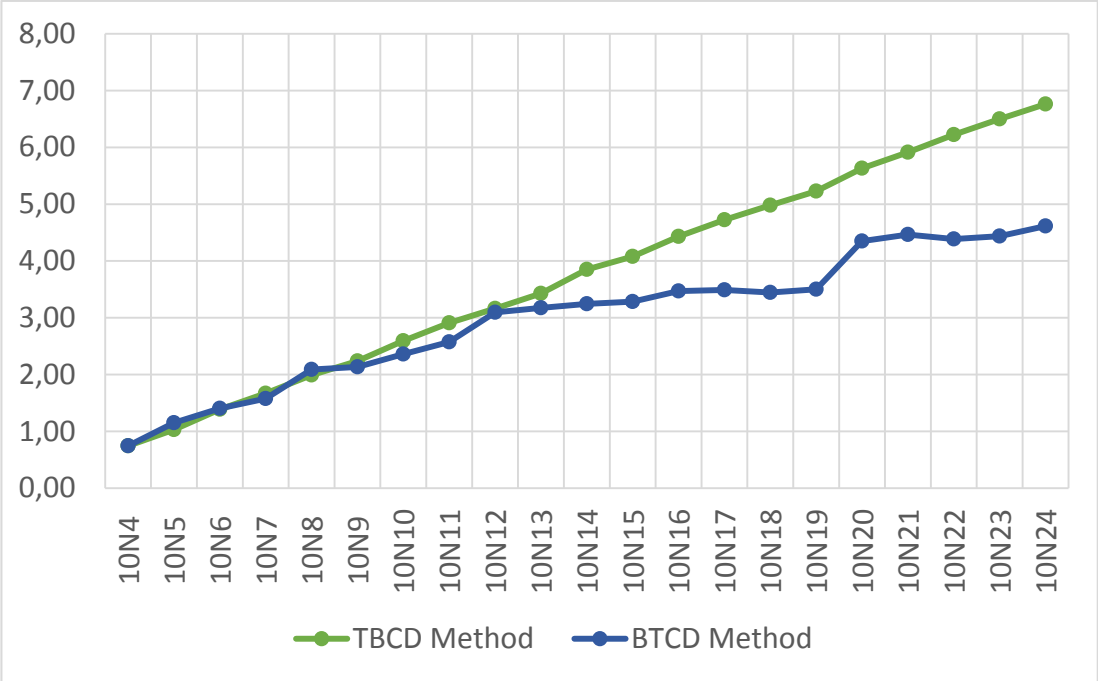

Figure 5.2 Dividend Bit-widths versus Timings (ns) for d = 10

Further observations are done with the divisor being set to 15. Table 5.3 and the corresponding graph in figure 5.3 shows that we have stable better results for even smaller bitwidths and the improvement percentages are slightly improved again.

It is clear that we are getting better timing results for large input bit-widths. This is because by separating the dividend into partitions we process parallel computations on each partition at a time. This is why our timing results are increasing in $log(N)$ rate, while the timing for TBCD method increases linearly with N.

Table 5.3 The Best Clock Periods for d = 15

	TIMING RESULTS (ns) for d = 15		
	Our BTCD Method	**TBCD Method**	**Our improvement**
15N4	0,78	0,78	0,00%
15N5	1,30	**0,99**	-31,19%
15N6	1,52	**1,51**	-0,66%
15N7	**1,72**	1,73	0,58%
15N8	**2,21**	2,25	2,00%
15N9	**2,44**	2,48	1,65%
15N10	**2,68**	2,93	8,53%
15N11	**2,90**	3,17	8,43%
15N12	**3,41**	3,62	5,77%
15N13	**3,57**	3,85	7,24%
15N14	**3,59**	4,33	17,20%
15N15	**3,66**	4,60	20,43%
15N16	**3,76**	5,09	26,08%
15N17	**3,76**	5,30	28,97%
15N18	**3,89**	5,73	31,98%
15N19	**4,05**	5,98	32,20%
15N20	**4,87**	6,53	25,36%
15N21	**4,94**	6,79	27,25%
15N22	**4,96**	7,15	30,60%
15N23	**5,00**	7,41	32,52%
15N24	**5,01**	7,86	36,24%

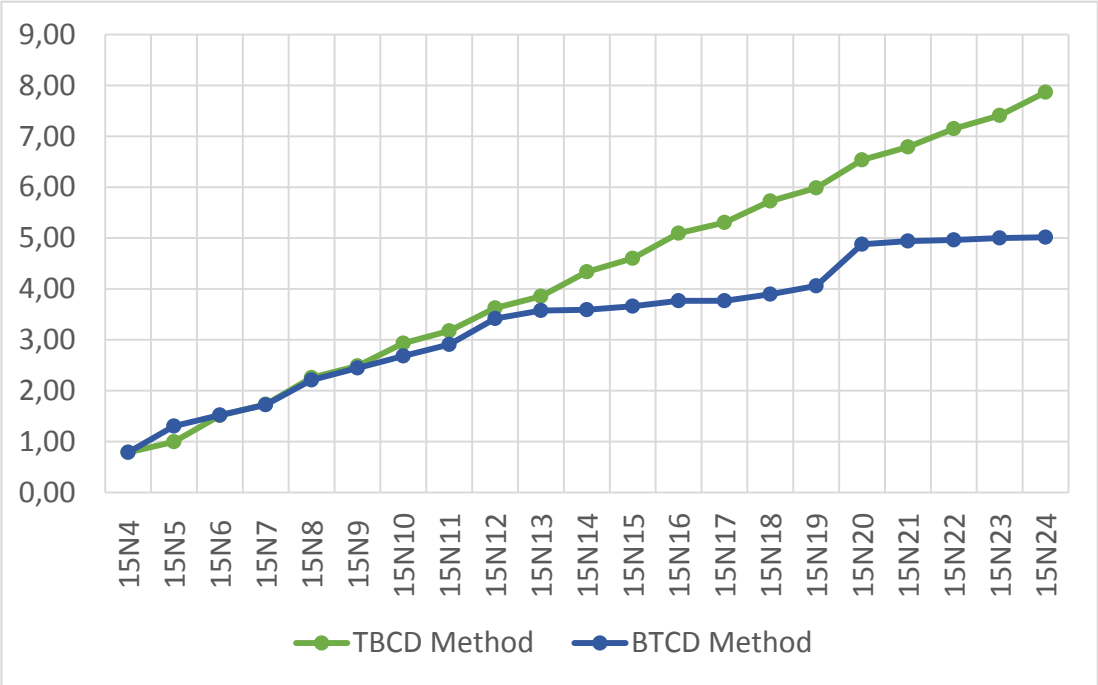

Figure 5.3 Dividend Bit-widths versus Timings (ns) for d=15

5.3.2 AREA RESULTS

As we previously mentioned, all synthesis scripts and methodology are built to achieve the best possible timing results. The area values, on the other hand, are the corresponding areas that the circuits with best clock periods occupy.

When comparing the occupied areas for the two methods, it is not a wise decision to compare the areas of the circuits with best possible timings. This is because synthesis tools sacrifice area to achieve faster clock frequencies, which makes the area and timing costs inversely proportional with themselves. If the circuits with the best timings are re-synthesized with more relaxed timing constraints, their areas will shrink; resulting a different area constraint for a relatively slower circuit.

To be able to compare occupied areas by two different methods, we must also take their best timing values in consideration. To do so, we introduced an area-timing product (ATP) that is simply the product of the best clock period and the corresponding area result for it. We consider the ATP of the two methods when comparing areas.

Furthermore, we normalize the ATP by dividing all the results with the maximum area value of all. That way, we adjust the range of ATP values between 0 and 1. From there we can stretch the range within 0 and any value K, by multiplying the whole set of results by K. In our case, normalized ATP is calculated as in Eq. 5.1

$$\boldsymbol{Normalized\ ATP = \frac{ATP}{\max(ATP_{TBCD}, ATP_{BTCD})}\ x\ 100} \qquad \textbf{(5.1)}$$

The area results when divisor is set to 3, 10, and 15 are shown in Table 5.4, 5.5 and 5.6. The corresponding graphs for these values are in Figure 5.4, 5.5 and 5.6. The results show that our area results are around 2 times the area of TBCD method. This is expected since we are using more resources for the sake of parallel calculations. However, area results are tolerable since there seems to be an exact proportion of 2 between the area constraints.

Table 5.4 Comparison of areas when d = 3

	AREA RESULTS for d = 3			
	Our BTCD Method		TBCD Method	
	AREA	Norm.ATP	AREA	Norm.ATP
3N4	841	0,29	821	0,29
3N5	1247	0,54	1147	0,50
3N6	1866	1,10	1397	0,71
3N7	3366	2,15	1563	0,89
3N8	4094	2,77	2251	1,44
3N9	4380	2,96	2278	1,58
3N10	5288	4,72	2844	2,21
3N11	5767	5,20	3209	2,73
3N12	5767	5,41	3459	3,19
3N13	6972	6,61	3915	3,84
3N14	10544	10,05	4005	4,26
3N15	11748	11,55	4567	5,10
3N16	10052	10,66	4760	5,81
3N17	10674	11,29	4869	6,11
3N18	12563	16,44	5312	7,24
3N19	11838	15,83	5322	7,76
3N20	13022	17,34	6071	9,18
3N21	14829	19,54	6410	10,23
3N22	15381	20,38	6593	10,76
3N23	14589	19,48	7045	11,99
3N24	16808	21,83	7458	13,66

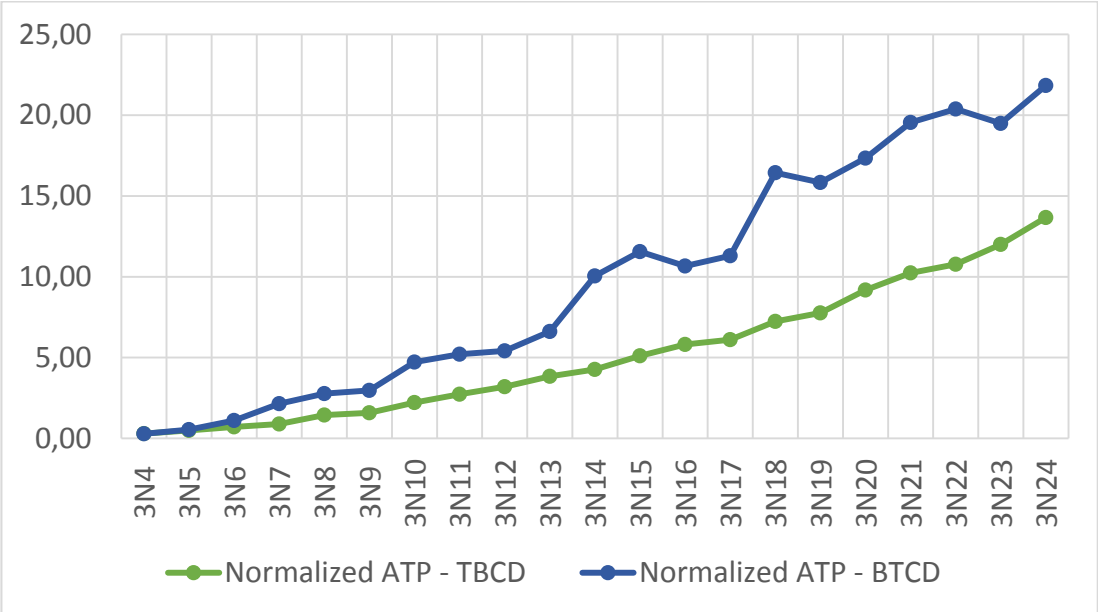

Figure 5.4 Normalized ATP Results for d = 3

Table 5.5 Comparison of areas when d = 10

	AREA RESULTS for d = 10			
	Our BTCD Method		TBCD Method	
	AREA	Norm.ATP	AREA	Norm.ATP
10N4	209	0,05	209	0,05
10N5	731	0,26	745	0,24
10N6	1353	0,60	1307	0,57
10N7	2760	1,38	1663	0,88
10N8	4104	2,72	2807	1,77
10N9	5019	3,40	3336	2,37
10N10	5022	3,76	3555	2,93
10N11	6360	5,20	4177	3,86
10N12	10102	9,93	5385	5,41
10N13	10864	10,96	5854	6,37
10N14	10943	11,28	6646	8,13
10N15	12340	12,87	6912	8,96
10N16	14067	15,51	7720	10,86
10N17	14659	16,25	8359	12,54
10N18	16525	18,09	8635	13,66
10N19	19675	21,88	9719	16,14
10N20	19945	27,56	10298	18,43
10N21	22579	32,03	10830	20,34
10N22	22785	31,74	11792	23,32
10N23	22283	31,41	11765	24,30
10N24	23108	33,88	13002	27,93

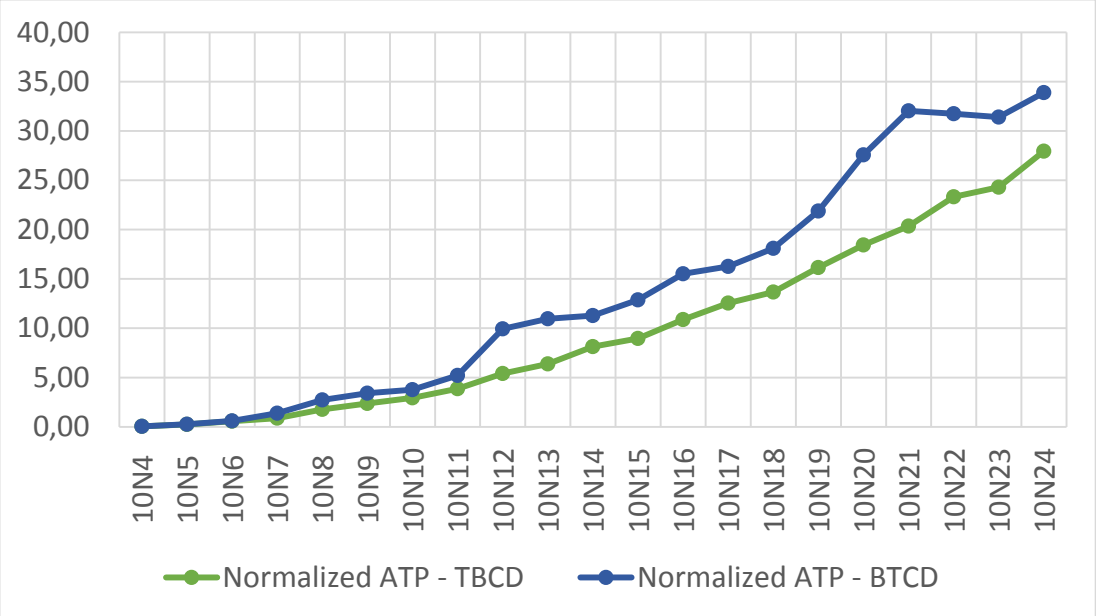

Figure 5.5 Normalized ATP Results for d = 10

Table 5.6 Comparison of areas when d = 15

	AREA RESULTS for d = 15			
	Our BTCD Method		TBCD Method	
	AREA	Norm.ATP	AREA	Norm.ATP
15N4	389	0,10	405	0,10
15N5	1004	0,41	1134	0,35
15N6	2637	1,27	2591	1,24
15N7	4327	2,36	3392	1,86
15N8	9849	6,91	5046	3,61
15N9	12853	9,98	5658	4,46
15N10	14293	12,18	7005	6,52
15N11	16532	15,27	8322	8,39
15N12	21096	22,89	8888	10,23
15N13	21325	24,23	10065	12,33
15N14	23644	26,97	11003	15,16
15N15	23763	27,63	11815	17,27
15N16	32778	39,22	13804	22,34
15N17	36620	43,84	14968	25,23
15N18	39574	48,99	16043	29,20
15N19	44660	57,58	16641	31,65
15N20	45764	70,93	18291	37,98
15N21	46147	72,44	18734	40,42
15N22	47890	75,51	21262	48,30
15N23	48621	77,25	20740	48,83
15N24	53997	86,05	21525	53,79

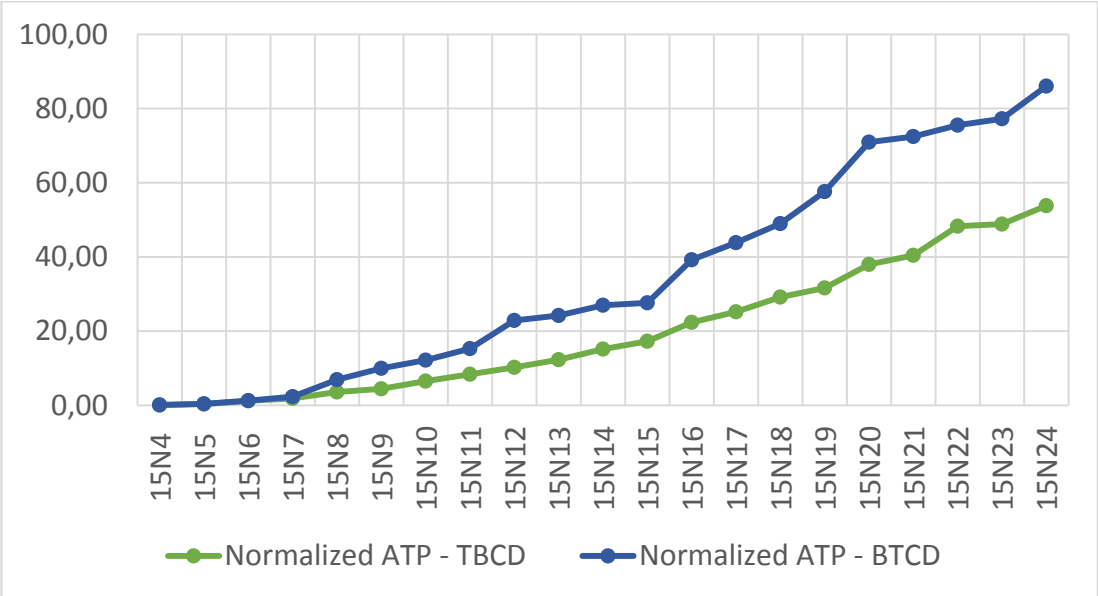

Figure 5.6 Normalized ATP Results for d = 15

6. CONCLUSIONS AND FUTURE WORK

In this study we have dealt with constant divider circuits and presented a new, faster LUT based constant integer division method. The proposed method has a divide and conquer approach with an ability to process parallel computations, which makes it faster with an applicable chip area occupation when compared to competitors.

The systems that require constant divisor operators usually require rapid and repeated division operations where timing is critical. (i. e. address mapping) In such a problem where timing is critical, our method suggests the fastest solution with a timing complexity of $log(N)$ where N is the dividend bit-width. We possess this advantage since we avoid the burden of laborious division calculation by pre-storing the results in LUTs and by partitioning the dividend into chunks of bits to run parallel computations on each partition.

The synthesis results for timing analysis show that we achieve much more efficient timings when compared to TBCD method and the results get better (reaches up to 35% improved clock period for N = 15) as the bit-width N increases.

The area results, have to be compared by considering the timing value of the corresponding area because each area value is a result of a corresponding synthesis operation with a best timing target. Therefore, when the target clock period is relaxed in the synthesis strategy, area result tend to shrink to a better value. To make area results comparable we introduced ATP, which is considered a comparable area result.

The occupied chip resources have to remain in logical borders so that the proposed method can be applicable. Our results show that the area constraints are around 1.1 to 2 times larger than our competitor within the considered range. This is expected and definitely acceptable especially when the timing superiority of our method is considered.

For the future work, our intention is to design a smart HDL generator for our algorithm that will group the input bits in the best possible way with the best possible grouping option at each step. The combination process of the LUTs can be accomplished in

variable amounts of items at each step so this property will be deeply examined and implemented into the HDL generator. Also the LUT sizes can be reduced by observing redundancies at each generation step.

We have conducted a detailed literature survey for this study and implemented some of the proposed methods. As a future work, we will implement an HDL Generator that will choose the most efficient method for the given constraints and implement that. This generator is necessary for the fact that each constant division algorithm can be good for a specific range of divisors or for specific input bit-widths. A generator that chooses between the algorithms can have the ability to always achieve the best available result for given constraints.

REFERENCES

Dinechin, F. and L. Didier., 2012. Table-based division by small integer constants. *Reconfigurable Computing: Architectures, Tools and Applications.* pp. 53-63.

Dinechin, F., 2012. Multiplication by rational constants. *IEEE Transactions on Computers,* vol. C-34, no. 9, pp. 866-869.

Drane, T., Cheung, W. C., & Constantinides, G., 2012. Correctly rounded constant integer division via multiply-add. *Circuits and Systems (ISCAS), 2012 IEEE International Symposium,* pp. 1243-1246.

Moller, N., & Granlund, T., 2011. Improved division by invariant integers. *Computers, IEEE Transactions,* vol.60, no.2, pp. 165-175.

Cavagnino, D., Werbrouck, A. E., 2008. Efficient algorithms for integer division by constants using multiplication. *The Computer Journal,* vol.51, no.4, pp. 470-480.

Muller, J. M., Tisserand, A., De Dinechin, B., & Monat, C. 2005. Division by Constant for the ST100 DSP Microprocessor. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium.* pp. 124-130.

Takagi, N., Kadowaki, S., Takagi, K., 2005. A hardware algorithm for integer division. *Proc. 17th Symp. Computer Artihmetic,* pp. 140-146.

Wirthlin, M. J. 2004. Constant coefficient multiplication using look-up tables. *Journal of VLSI signal processing systems for signal, image and video technology*, vol.36, no.1, pp. 7-15.

Mohan, M., Krishnan, R., Kumar, A., & Balakrishnan, M., 2002. A new divide and conquer method for achieving high speed division in hardware. *Proceedings of the 2002 Asia and South Pacific Design Automation Conference,* pp. 535

Rejeb, B., Henkelmann, H., & Anheier, W., 2001. Integer division in residue number system. *Electronics, Circuits and Systems,. ICECS 2001. The 8th IEEE International Conference,* vol.1, pp. 259-262.

Parhami, B., 2000. Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press.

Schwarzbacher, A., Th., Brutscheck, M., Schwinge, O., Foley, J. B., 2000. Constant Divider Structures of the Form 2n+-1. *Irish Signals and Systems Conference,* Dublin, Ireland. Pp. 368-375.

Hung, P., Fahmy, H., Mencer, O., & Flynn, M. J., 1999. Fast division algorithm with a small lookup table. *Signals, Systems, and Computers,. Conference Record of the Thirty-Third Asilomar Conference,* vol.2, pp. 1465-1468.

Obermann, S. F., & Flynn, M. J., 1997. Division algorithms and implementations. *Computers, IEEE Transactions,* vol.46, no.8, pp. 833-854.

Al-Besher, B, Bouridane, A., Ashur, A. S., 1997. An RNS-based division architecture for constant divisors of the form 2n+1 and 2n-1. *Irish Signals & Systems Conference*

Hitz, M. A., Kaltofen, E., 1995. Integer division in residue number systems. *Computers, IEEE Transactions on,* vol.44, no.8, pp. 983-989.

Doran, R.W., 1995. Special cases of division. *Journal of Universal Computer Science,* vol.1, no.3, pp.67-82.

Granlund, T., & Montgomery, P. L., 1994. Division by invariant integers using multiplication. *ACM SIGPLAN Notices,* vol.29, no.6, pp. 61-72.

Srinivasan, P. and Petry, F., 1994. Constant division algorithms. *IEE Proc. Computers and Digital Techniques.* vol. 141, no. 6. Pp. 334 -340.

Petry, F. E., Srinivasan, P., 1993. Division techniques for integers of the form 2n+1 and 2n-1. *Int. J. Electronics,* vol.74, no.5.

Alverson, R., 1991, June. Integer division using reciprocals. In *Computer Arithmetic, 1991. Proceedings, 10th IEEE Symposium.* pp. 186-190.

Reif, J. H., 1989. Optimal size integer division circuits. *Proceedings of the twenty-first annual ACM symposium on Theory of computing,* pp. 264-273.

Li, S. Y., 1985. Fast constant division routines. *Computers, IEEE Transactions on*, vol.100 no.9, pp. 866-869.

Ci Yun Guei, Yang Xiao Dong, and Wang Bing Shan., 1985. A fast division technique for constant divisors $2^m(2^n \pm 1)$. *Proceedings of the 1st International Conference on Computers and Applications, Beijing, China,* pp. 715-718

Petry, F. E. 1983. Two's complement extension of a parallel binary division by ten. *Electron. Lett.*, vol.19, no.18, pp. 718-720.

Schreiber, F. A., Stefanelli, R. 1978. Two methods for fast integer binary-BCD conversion. *Proceedings of the Fourth IEEE Symposium on Computer Arithmetic, Santa Monica, CA,* pp. 200-207

Artzy, E., Hinds, J. A., and Saal, H. J., 1976. A fast division technique for constant divisors. *Communications of the ACM,* vol.19, pp. 98-101.

Sites, R. L., 1974. Serial binary division by ten. *Computers, IEEE Transactions on*, vol.100, no.12, pp. 1299-1301.

Jacobsohn, D. H., 1973. A combinatoric division algorithm for fixed-integer divisors. *Computers, IEEE Transactions on,* vol.100, no.6, pp. 608-610.

APPENDIX A

Table A.1 Results for d=3

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
3N4	1,078	841,5791	0,288285		1,128	821,6207	0,294503
3N5	1,360	1247,4000	0,539080		1,365	1147,6080	0,497777
3N6	1,865	1866,1104	1,105923		1,605	1397,0880	0,712537
3N7	2,010	3366,3168	2,150106		1,795	1563,4080	0,891755
3N8	2,130	4094,7984	2,771539		2,020	2251,9728	1,445518
3N9	2,132	4380,8688	2,967948		2,180	2278,5840	1,578448
3N10	2,810	5288,9760	4,722661		2,447	2844,0720	2,211934
3N11	2,840	5767,9776	5,205360		2,680	3209,9760	2,733666
3N12	2,956	5767,9776	5,417973		2,906	3459,4560	3,194569
3N13	2,985	6972,1344	6,613311		3,090	3915,1728	3,844309
3N14	3,000	10544,6880	10,052263		3,350	4004,9856	4,263386
3N15	3,095	11748,8448	11,554860		3,520	4567,1472	5,108537
3N16	3,338	10052,3808	10,662625		3,845	4760,0784	5,815932
3N17	3,330	10674,4176	11,295288		3,950	4869,8496	6,112537
3N18	4,118	12563,8128	16,440562		4,290	5312,2608	7,241784
3N19	4,210	11838,6576	15,837748		4,590	5322,2400	7,762757
3N20	4,190	13022,8560	17,339203		4,760	6070,6800	9,182336
3N21	4,148	14829,0912	19,546192		5,025	6409,9728	10,235313
3N22	4,170	15381,2736	20,381551		5,140	6592,9248	10,768373
3N23	4,202	14589,5904	19,483173		5,360	7045,3152	11,999802
3N24	4,088	16808,2992	21,834514		5,765	7457,7888	13,662125

Table A.2 Results for d= 5 and d=6

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
5N4	1,0250	944,6976	0,3077		1,0600	804,9888	0,2711
5N5	1,3650	1453,6368	0,6305		1,3950	1423,6992	0,6311
5N6	1,6660	1849,4784	0,9791		1,6700	1746,3600	0,9267
5N7	2,0400	3752,1792	2,4323		2,0260	2770,8912	1,7839
5N8	2,2750	5036,1696	3,6407		2,2550	3166,7328	2,2692
5N9	2,3140	5601,6576	4,1190		2,6400	4154,6736	3,4854
5N10	2,3450	6197,0832	4,6178		2,9250	4167,9792	3,8740
5N11	3,0250	8202,9024	7,8850		3,1725	5175,8784	5,2179
5N12	3,0475	8661,9456	8,3882		3,4250	5521,8240	6,0097
5N13	3,1350	9686,4768	9,6497		3,7900	6323,4864	7,6156
5N14	3,2050	11602,4833	11,8165		4,0175	7328,0592	9,3552
5N15	3,2950	14559,6528	15,2446		4,4700	7241,5728	10,2861
5N16	3,4800	15188,3425	16,7957		4,6720	7936,7904	11,7830
5N17	3,4425	15617,4481	17,0841		4,9500	8884,8144	13,9753
5N18	3,4600	16365,8881	17,9939		5,2500	9413,7120	15,7047
5N19	4,1400	21016,1953	27,6480		5,6640	10814,1264	19,4636
5N20	4,3140	18358,4017	25,1666		5,9020	10348,4304	19,4081
5N21	4,1950	19555,9057	26,0687		6,2900	11838,6576	23,6626
5N22	4,2600	21960,8929	29,7282		6,4160	12071,5057	24,6113
5N23	4,3050	23145,0913	31,6622		6,7960	12666,9312	27,3548
5N24	4,3100	22852,3681	31,2981		7,0500	12962,9809	29,0404
6N4	0,9060	365,9040	0,1053		0,9000	325,9872	0,0932
6N5	1,1900	578,7936	0,2189		1,1400	881,4960	0,3193
6N6	1,3650	1224,1152	0,5310		1,3400	1087,7328	0,4632
6N7	1,8300	2601,2448	1,5127		1,6140	1596,6720	0,8189
6N8	2,0400	3991,6800	2,5876		1,7850	1935,9648	1,0981
6N9	2,1450	3426,1920	2,3353		2,0520	2431,5984	1,5855
6N10	2,0950	5002,9056	3,3305		2,2040	2508,1056	1,7566
6N11	2,8100	5162,5728	4,6098		2,4300	2757,5856	2,1293
6N12	2,9550	6596,2512	6,1939		2,6900	2890,6416	2,4709
6N13	3,0600	6642,8208	6,4593		2,9080	3209,9760	2,9662
6N14	2,9900	8349,2640	7,9328		3,1550	3422,8656	3,4316
6N15	3,0800	10155,4992	9,9394		3,3480	4024,9440	4,2821
6N16	3,2340	11625,7681	11,9473		3,5650	4593,7584	5,2040
6N17	3,2200	11775,4560	12,0488		3,8300	4740,1200	5,7690
6N18	3,2150	13069,4257	13,3520		4,0100	5145,9408	6,5572
6N19	3,9700	15254,8705	19,2446		4,3020	5229,1008	7,1484
6N20	4,0200	15876,9073	20,2815		4,4520	5265,6912	7,4494
6N21	4,0800	13488,5521	17,4878		4,6620	6034,0896	8,9391
6N22	4,0900	16009,9633	20,8076		4,9450	6030,7632	9,4765
6N23	4,2000	16166,3041	21,5759		5,1780	6067,3536	9,9832
6N24	4,0700	18069,0049	23,3689		5,3650	6925,5648	11,8068

Table A.3 Results for d=7 and d=9

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
7N4	0,9500	685,2384	0,2069		0,9500	748,4400	0,2259
7N5	1,4450	1094,3856	0,5025		1,3500	1683,1584	0,7221
7N6	1,6200	2195,4240	1,1302		1,6950	2128,8960	1,1467
7N7	2,0200	4174,6320	2,6797		2,0050	3499,3728	2,2295
7N8	2,2500	6237,0000	4,4593		2,3300	3748,8528	2,7756
7N9	2,3300	6862,3632	5,0809		2,7250	4913,0928	4,2543
7N10	2,5150	7757,1648	6,1994		2,9580	5242,4064	4,9276
7N11	3,1100	9822,8593	9,7075		3,4000	6283,5696	6,7888
7N12	3,2400	11043,6480	11,3702		3,6080	6775,8768	7,7686
7N13	3,2400	12809,9664	13,1887		3,9800	8276,0832	10,4669
7N14	3,3200	13668,1777	14,4197		4,2525	8492,2992	11,4757
7N15	3,3720	15983,3521	17,1263		4,6800	9280,6560	13,8017
7N16	3,4900	18488,1313	20,5035		4,9400	9902,6928	15,5449
7N17	3,5575	17510,1697	19,7945		5,3550	10764,2304	18,3169
7N18	3,5700	19945,0945	22,6263		5,6700	10973,7936	19,7719
7N19	4,3800	21165,8833	29,4591		6,0000	12763,3969	24,3347
7N20	4,4800	22769,2081	32,4142		6,2825	13282,3153	26,5164
7N21	4,4040	23607,4609	33,0374		6,7250	13871,0881	29,6423
7N22	4,4260	23361,3073	32,8562		6,9200	14433,2496	31,7380
7N23	4,4660	26777,5201	38,0013		7,3400	16272,7489	37,9547
7N24	4,4700	27935,1073	39,6795		7,5700	16036,5745	38,5759
9N4	0,8580	415,8000	0,1134		0,8400	455,7168	0,1216
9N5	1,3620	1077,7536	0,4665		1,1450	1510,1856	0,5495
9N6	1,6350	2182,1184	1,1337		1,6600	2232,0144	1,1774
9N7	1,7460	3346,3584	1,8566		1,8740	3632,4288	2,1631
9N8	2,1640	6779,2032	4,6617		2,3580	3802,0752	2,8489
9N9	2,3640	8276,0832	6,2170		2,5500	5678,1648	4,6011
9N10	2,5300	9417,0384	7,5708		3,0800	6210,3888	6,0782
9N11	2,7050	9952,5888	8,5548		3,3100	7477,7472	7,8652
9N12	3,2100	13538,4480	13,8097		3,7500	8196,2496	9,7669
9N13	3,4220	14556,3264	15,8285		3,9700	9939,2832	12,5388
9N14	3,4500	14290,2145	15,6663		4,4300	9456,9552	13,3126
9N15	3,4900	18165,4705	20,1456		4,7050	11053,6273	16,5262
9N16	3,6560	22217,0257	25,8108		5,0700	12394,1664	19,9680
9N17	3,6550	23401,2241	27,1791		5,3825	14463,1872	24,7376
9N18	3,7140	25659,8497	30,2834		5,9000	14040,7345	26,3239
9N19	3,7550	27978,3505	33,3842		6,0640	15251,5440	29,3888
9N20	4,5820	30423,2545	44,2965		6,5280	15324,7248	31,7894
9N21	4,6475	29125,9585	43,0139		6,8340	17629,9201	38,2855
9N22	4,7140	31404,5426	47,0426		7,1700	18581,2704	42,3354
9N23	4,7700	31271,4865	47,3997		7,6100	19076,9041	46,1319
9N24	4,8080	33506,8274	51,1925		7,8680	19935,1153	49,8416

Table A.4 Results for d=10 and d=11

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
10N4	0,7450	209,5632	0,0496		0,7450	209,5632	0,0496
10N5	1,1500	731,8080	0,2674		1,0300	745,1136	0,2439
10N6	1,4050	1353,8448	0,6044		1,3880	1307,2752	0,5766
10N7	1,5750	2760,9120	1,3818		1,6700	1663,2000	0,8826
10N8	2,0880	4104,7776	2,7235		1,9900	2807,4816	1,7753
10N9	2,1350	5019,5376	3,4054		2,2375	3336,3792	2,3722
10N10	2,3600	5022,8640	3,7668		2,5950	3555,9216	2,9322
10N11	2,5750	6360,0768	5,2041		2,9100	4177,9584	3,8634
10N12	3,0950	10102,2768	9,9355		3,1625	5385,4416	5,4120
10N13	3,1750	10864,0224	10,9608		3,4275	5854,4640	6,3764
10N14	3,2450	10943,8560	11,2848		3,8500	6646,1472	8,1309
10N15	3,2840	12340,9440	12,8784		4,0800	6912,2592	8,9617
10N16	3,4700	14067,3456	15,5114		4,4300	7720,5744	10,8683
10N17	3,4900	14659,4448	16,2574		4,7240	8359,2432	12,5483
10N18	3,4450	16525,5553	18,0906		4,9800	8635,3344	13,6652
10N19	3,5000	19675,6561	21,8830		5,2280	9719,7408	16,1473
10N20	4,3500	19945,0945	27,5698		5,6325	10298,5344	18,4326
10N21	4,4650	22579,6033	32,0366		5,9125	10830,7584	20,3488
10N22	4,3850	22785,8401	31,7500		6,2240	11792,0880	23,3222
10N23	4,4360	22283,5537	31,4112		6,5000	11765,4769	24,3014
10N24	4,6150	23108,5009	33,8885		6,7620	13002,8977	27,9399
11N4	0,8580	372,5568	0,1016		0,8660	289,3968	0,0796
11N5	1,4000	1350,5184	0,6008		1,1400	1570,0608	0,5688
11N6	1,6025	3199,9968	1,6295		1,6060	2870,6832	1,4650
11N7	1,9200	5708,1024	3,4826		1,8460	4434,0912	2,6010
11N8	2,2560	8192,9232	5,8734		2,3450	5275,6704	3,9312
11N9	2,5080	11715,5808	9,3368		2,5600	6433,2576	5,2333
11N10	2,7000	13212,4608	11,3359		3,0900	7361,3232	7,2281
11N11	3,0080	15527,6353	14,8420		3,3000	8189,5968	8,5879
11N12	3,4450	18105,5953	19,8203		3,8160	9446,9760	11,4554
11N13	3,4540	18115,5745	19,8831		4,0300	11419,5312	14,6239
11N14	3,5700	21265,6753	24,1244		4,5600	12177,9504	17,6461
11N15	3,7180	24392,4913	28,8187		4,7625	13292,2944	20,1161
11N16	3,8200	29066,0834	35,2825		5,2600	14419,9440	24,1023
11N17	3,8600	32282,7121	39,5974		5,5200	16518,9025	28,9754
11N18	3,9480	34464,8306	43,2377		6,0700	16967,9665	32,7286
11N19	4,1900	38030,7314	50,6358		6,2900	18188,7553	36,3549
11N20	4,8300	41247,3602	63,3071		6,7300	18837,4033	40,2851
11N21	4,8520	40056,5090	61,7594		6,9940	19708,9201	43,8023
11N22	5,0300	39018,6722	62,3662		7,4800	21847,7953	51,9300
11N23	5,0640	42844,0322	68,9434		7,7540	22582,9297	55,6436
11N24	5,2140	46024,0706	76,2544		8,2350	23451,1201	61,3672

Table A.5 Results for d=12 and d=13

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
12N4	0,6200	76,5072	0,0151		0,6200	76,5072	0,0151
12N5	0,9540	302,7024	0,0918		0,9200	335,9664	0,0982
12N6	1,0800	751,7664	0,2580		1,1100	801,6624	0,2828
12N7	1,3050	1493,5536	0,6194		1,3250	1150,9344	0,4846
12N8	1,8550	2528,0640	1,4902		1,5580	1450,3104	0,7180
12N9	1,9050	3346,3584	2,0257		1,7250	1696,4640	0,9299
12N10	1,9400	3678,9984	2,2680		1,9750	2105,6112	1,3215
12N11	2,1475	4324,3200	2,9509		2,1140	2820,7872	1,8949
12N12	2,7225	6250,3056	5,4073		2,4000	2834,0928	2,1614
12N13	2,8500	7371,3024	6,6757		2,5775	3193,3440	2,6155
12N14	3,0980	6732,6336	6,6279		2,8650	3256,5456	2,9648
12N15	3,0875	8429,0976	8,2698		3,0340	3170,0592	3,0563
12N16	3,1100	9826,1856	9,7108		3,3000	3742,2000	3,9242
12N17	3,1200	10245,3120	10,1575		3,4100	3698,9568	4,0081
12N18	3,1100	11472,7536	11,3380		3,6720	5189,1840	6,0550
12N19	3,1450	11369,6353	11,3626		3,8600	4520,5776	5,5449
12N20	4,0940	14103,9360	18,3484		4,1350	5475,2544	7,1943
12N21	4,0950	14406,6385	18,7467		4,2820	5272,3440	7,1740
12N22	4,1600	13382,1073	17,6900		4,4800	6493,1328	9,2436
12N23	4,1300	14639,4865	19,2125		4,7650	5980,8672	9,0560
12N24	4,4400	14223,6865	20,0680		4,9700	6217,0416	9,8186
13N4	0,8250	419,1264	0,1099		0,8150	412,4736	0,1068
13N5	1,3640	1417,0464	0,6142		1,1200	1417,0464	0,5043
13N6	1,6580	2867,3568	1,5107		1,6325	3356,3376	1,7411
13N7	1,8450	6872,3424	4,0291		1,9140	4716,8352	2,8688
13N8	2,3900	10444,8960	7,9325		2,4500	6336,7920	4,9334
13N9	2,6900	12813,2929	10,9527		2,7600	6812,4672	5,9748
13N10	2,9340	16272,7489	15,1715		3,3275	9590,0112	10,1402
13N11	3,1200	19359,6481	19,1938		3,5750	9822,8592	11,1589
13N12	3,6650	23091,8689	26,8932		4,1150	11379,6144	14,8801
13N13	3,7400	25167,5425	29,9103		4,3700	12829,9249	17,8162
13N14	3,7425	26198,7265	31,1567		4,9000	14310,1728	22,2818
13N15	3,8720	30047,3713	36,9701		5,1000	16143,0192	26,1616
13N16	4,0100	35725,5361	45,5231		5,7375	17716,4064	32,3003
13N17	4,0300	37278,9649	47,7395		5,9900	18747,5905	35,6846
13N18	4,1700	41523,4514	55,0223		6,5575	21185,8417	44,1462
13N19	4,4250	44730,1010	62,8959		6,8000	22147,1713	47,8560
13N20	5,1300	47477,7074	77,3955		7,3580	23540,9329	55,0418
13N21	5,1700	52573,7522	86,3711		7,6200	25300,5985	61,2625
13N22	5,1540	53318,8659	87,3241		8,1600	25649,8705	66,5096
13N23	5,3000	52803,2738	88,9294		8,4160	26710,9921	71,4339
13N24	5,3200	59153,3715	100,0000		9,0250	28354,2337	81,3156

Table A.6 Results for d=14 and d=15

	Our BTCD Method				TBCD Method		
	Best Timing (ns)	Area	Normalized ATP		Best Timing (ns)	Area	Normalized ATP
14N4	0,7300	96,4656	0,0224		0,7300	96,4656	0,0224
14N5	1,1350	585,4464	0,2112		0,9540	718,5024	0,2178
14N6	1,3540	1639,9152	0,7056		1,3660	1596,6720	0,6931
14N7	1,5300	3306,4416	1,6075		1,6000	2162,1600	1,0993
14N8	2,1080	6173,7984	4,1355		2,0350	3199,9968	2,0693
14N9	2,2600	7644,0672	5,4896		2,2600	3918,4992	2,8141
14N10	2,4880	8771,7168	6,9350		2,7600	4816,6272	4,2244
14N11	2,6200	10085,6449	8,3968		2,9360	4943,0304	4,6117
14N12	3,3000	15224,9328	15,9653		3,3900	5954,2560	6,4141
14N13	3,4350	14759,2369	16,1101		3,5950	6346,7712	7,2504
14N14	3,4875	17822,8513	19,7515		4,0850	7447,8096	9,6678
14N15	3,5075	19609,1281	21,8557		4,3300	7690,6368	10,5818
14N16	3,6740	22938,8545	26,7806		4,6950	8748,4320	13,0519
14N17	3,6300	24941,3473	28,7697		4,9550	9367,1424	14,7489
14N18	3,6400	26521,3873	30,6765		5,3950	10039,0752	17,2105
14N19	3,7275	28670,2417	33,9592		5,5800	10893,9600	19,3165
14N20	4,6325	33556,7234	49,3974		6,0550	11858,6160	22,8169
14N21	4,7100	34837,3873	52,1405		6,3200	12001,6512	24,1028
14N22	4,7980	33796,2242	51,5273		6,7200	13002,8976	27,7663
14N23	4,7900	35366,2849	53,8312		6,9550	13967,5537	30,8693
14N24	4,8400	38090,6066	58,5831		7,4050	14486,4720	34,0876
15N4	0,7875	389,1888	0,0974		0,7875	405,8208	0,1016
15N5	1,3040	1004,5728	0,4163		0,9940	1134,3024	0,3583
15N6	1,5200	2637,8352	1,2741		1,5100	2591,2656	1,2434
15N7	1,7200	4327,6464	2,3653		1,7300	3392,9280	1,8652
15N8	2,2100	9849,4704	6,9169		2,2550	5046,1488	3,6159
15N9	2,4450	12853,2097	9,9862		2,4860	5658,2064	4,4698
15N10	2,6825	14293,5408	12,1840		2,9325	7005,3984	6,5280
15N11	2,9075	16532,2081	15,2742		3,1750	8322,6528	8,3968
15N12	3,4160	21096,0289	22,8996		3,6250	8888,1408	10,2383
15N13	3,5760	21325,5505	24,2330		3,8550	10065,6864	12,3304
15N14	3,5900	23644,0513	26,9727		4,3360	11003,7312	15,1614
15N15	3,6600	23763,8017	27,6380		4,6000	11815,3728	17,2709
15N16	3,7660	32778,3458	39,2262		5,0950	13804,5600	22,3499
15N17	3,7680	36620,3377	43,8472		5,3050	14968,8001	25,2337
15N18	3,8960	39574,1809	48,9936		5,7280	16043,2273	29,2014
15N19	4,0580	44660,2466	57,5893		5,9850	16641,9792	31,6503
15N20	4,8780	45764,6114	70,9382		6,5350	18291,8736	37,9850
15N21	4,9400	46147,1474	72,4404		6,7900	18734,2849	40,4218
15N22	4,9620	47890,1810	75,5113		7,1500	21262,3489	48,3088
15N23	5,0000	48621,9889	77,2523		7,4100	20740,1040	48,8358
15N24	5,0150	53997,4514	86,0504		7,8650	21525,1345	53,7964

CURRICULUM VITAE

FULL NAME: Anıl Bayram

ADDRESS: Şemsettin Günaltay Cd. No: 4/15 Bostancı Kadıköy / İSTANBUL

BIRTH PLACE / YEAR: İstanbul / 1985

LANGUAGE: Turkish (Native), English

HIGH SCHOOL: Vefa Anatolian High School, 2003

BS: Electrical & Electronics Engineering, Bilkent University, Ankara, 2008

MS: Electrical & Electronics Engineering, Bahcesehir University, Istanbul, 2013

NAME OF INSTITUTE: Natural and Applied Sciences

NAME OF PROGRAM: Electrical and Electronics Engineering

WORK EXPERIENCE: 11.2011 – Ongoing

ASIC / FPGA Design Engineer

CMOSVision Ltd, Kocaeli