T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ

A New Carry Save Tree Algorithm

Master's Thesis

OKAN KESKİN

İstanbul, 2010

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Electrical and Electronics Engineering

A New Carry Save Tree Algorithm

Master's Thesis

Okan Keskin

Supervisor: Asst. Prof. H. Fatih UĞURDAĞ

İstanbul, 2010

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Electrical and Electronics Engineering

Title of the Master's Thesis: A New Carry Save Tree Algorithm
Name/Last Name of the Student: Okan Keskin
Date of Thesis Defense: September 13, 2010

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

Signature

Asst. Prof F. 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.

Examining Committee Members

Asst. Prof. H. Fatih UĞURDAĞ (Supervisor):

Asst. Prof. Levent EREN:

Asst. Prof. Yalçın ÇEKİÇ:

Acknowledgments

I would like to start by thanking my supervisor Asst. Prof. H. Fatih UĞURDAĞ, who has been of all a great help and source of motivation.

I would like to thank to Asst. Prof. Levent EREN and Asst. Prof. Yalçın ÇEKİÇ, my thesis committee members, for taking the time.

I would also like to thank all the people that worked on the problem addressed by this thesis beside myself and my thesis advisor. Some of them worked on this problem even earlier than I did. I want to share their names in the chronological order (from past to more recent) they started their work: Soner Dedeoğlu, Gürbey Fıçı, Cihan Tunç, Fatih Temizkan. At this point I would especially thank Cihan Tunç for providing significant help at the benchmarking stage of this thesis work.

In addition I would like to thank TÜBİTAK for supporting me with BİDEB scholarship program in 2008 and 2009.

Finally, I would like to thank my family for their unending support and encouragement.

Okan Keskin
İstanbul, September 2010

ABSTRACT

A NEW CARRY SAVE TREE ALGORITHM

Keskin, Okan

Electrical and Electronics Engineering
Thesis Advisor: Asst. Prof. H. Fatih Uğurdağ

Date (September, 2010), 50 Pages

Carry Save Adder (CSA) trees are special logic circuit structures for summation. They are also mainly used for multiplication which is a special form of summation. They target to compute the sum of three or more *n*-bit binary numbers and produce two numbers instead of one as a result. Addition of these two numbers by a final adder gives the actual result. To do so it uses logic blocks named counters like half adder and full adder. This process is also named as column compression. We will use Carry Save Tree (CST) name instead of Carry Save Adder (CSA) tree. In this thesis a novel CST algorithm is introduced which we named as Plowing based Carry Save Tree (PCST) generation. Plowing word is used to define a special technique we use as part of PCST that reduces the bitwidth of the final adder. Though the algorithm is defined by the use of full adders and half adders, it can be defined with different counter structure with more inputs, because it is actually a topology not a single design definition. For the synthesis and functional testing purposes a HDL code generator is written in Perl. This generator is capable of applying different CST algorithms with minor changes. Wallace tree, Dadda tree and PCST are already defined in this generator. Most vital point about this generator is its time routed CST generation capability. What we mean by time routing is the matching process of bits according to their delays with the half adder and full adder inputs considering that there isn't a fixed delay for each input to output path. This provides an optimization for the critical path of the CST circuit. PCST yields circuits with better than even commercial software synthesis tools (e.g., Synopsys DC/DesignWare).We implemented Wallace tree, Dadda tree, and PCST for saturated unsigned summation and multiplication.

Time routed PCST, Wallace tree, and Dadda Tree generator outputs gave better timing results than ordinary ones up to 88.68 percent of the cases. PCST gave better timing results of 9.43 percent for saturated unsigned multiplication cases and 53.85 percent for saturated unsigned summation cases.

Keywords: CSA tree, Column Compression, Wallace tree, Dadda tree.

ÖZET

YENİ BİR CARRY SAVE TREE ALGORİTMASI

Keskin, Okan

Elektrik-Elektronik Mühendisliği
Tez Danışmanı: Yrd. Doç. Dr. H. Fatih Uğurdağ

Tarih (Eylül, 2010), 50 Sayfa

Elde taşıyıcılı toplayıcı ağaç yapıları, Carry Save Adder (CSA) tree, toplama için tasarlanmış özel mantık devre yapılarıdır. İkilik tabanda üç veya daha fazla n-bit sayının toplamını tek bir sonuç yerine iki sayıdan oluşan bir ara sonuç olarak hesaplamayı amaçlarlar. Bu iki sayının son bir toplayıcıyla toplanması istenen sonuca ulaşılmasını sağlar. Bu işlemi gerçekleştirebilmek için yarım toplayıcı ve tam toplayıcı gibi sayıcı devreler kullanılmaktadır. Bu işleme aynı zamanda sütun sıkıştırma da denmektedir. Biz CSA ismi yerine elde taşıyıcılı ağaç yapıları, Carry Save Tree (CST) ismini kullanacağız.Bu tez çalışmasında sürme tabanlı CST (PCST) oluşturma adlı özgün bir CST algoritması sunulmuştur. PCST nin bir parçası olarak kullandığımız ve son toplayıcıda bit genişliğini azaltan özel tekniğimize sürme (Plowing) adını verdik. Bu tez çalışması içerisinde PCST, yarım ve tam toplayıcılarla tanımlanmış olsada esnek yapısından ötürü diğer çok girişli sayıcı devrelerlede kullanılabilir. Sentez ve test amacıyla kullanılmak üzere Perl dilinde donanım tanımlama dili (HDL) kodu üreten bir script hazırlanmıştır. Bu script farklı CST algoritmalarını ufak değişikliklerle çalıştırabilmektedir. Wallace ağaç yapısı ve Dadda Ağaç yapısını hali hazırda desteklemektedir. Bu scriptin en önemli özelliği zamana bağlı olarak CST algoritmalarını gerçeklemesidir. Zamana bağlıdan kastedilen bitlerin gecikmelerine bağlı olarak yarım toplayıcı ve tam toplayıcı devrelerin girişleriyle eşleştirilmesidir bu yapılırken tam toplayıcı ve yarım toplayıcı devrelerinin girişten çıkışa olan gecikmelerinin birbirlerine eşit olmadığı göz önünde bulundurulmuştur. Bu yöntem CST devresinin toplam gecikmesinin optimizasyonunu sağlar. PCST endüstriyel sentez yazılımlarından (örnek, Synopsys DC/DesignWare) daha iyi sonuçlar üretebilen devreler oluşturabilmektedir. Tezde Wallace ağaç yapısı, Dadda ağaç yapısı ve PCST kullanarak belli bir değerde doyuma ulaşan işaretsiz toplama ve çarpma işlemleri gerçeklenmiştir. Yapılan sentezlerin yüzde 88.68 inde zamana bağlı üretilen PCST, Wallace ağaç yapısı ve Dadda ağaç yapısı daha iyi sonuçlar üretmiştir.

PCST algoritması belli bir değerde doyuma uylaşan işaretsiz çarpma işlemleri için yapılan sentezlerin yüzde 9.43'ünde, belli bir değerde doyuma ulaşan işaretsiz toplama işlemleri için yapılan sentezlerin yüzde 53.85'inde diğerlerine göre daha iyi sonuçlar üretmiştir.

Anahtar Kelimeler: CSA ağaç yapıları, Satır Sıkıştırma, Wallace ağaç yapısı, Dadda ağaç yapısı

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET vi
LIST OF TABLES x
LIST OF FIGURES xi
LIST OF ABBREVIATIONS xiii

1. INTRODUCTION 1
    1.1 SCOPE AND CONTRIBUTIONS OF THE THESIS 1
    1.2 WALLACE'S IDEA 2
    1.3 HISTORY OF THIS RESEARCH WORK 3
    1.4 OUTLINE OF THESIS REPORT 3
2. PREVIOUS WORK 5
    2.1 CARRY SAVE TREE 5
    2.2 SATURATED SUMMATION 8
3. PLOWING BASED CARRY SAVE TREE (PCST) GENERATION 9
    3.1 NOTATION 9
    3.2 OVERVIEW OF PCST 11
    3.3 GUIDANCE 14
    3.4 PLOWING 14
4. SATURATED UNSIGNED SUMMATION AND MULTIPLICATION 16
    4.1 CONVENTIONAL SATURATION APPROACH 16
    4.2 OUR IMPLEMENTATION 18
5. HDL GENERATOR 20
    5.1 GENERATOR ALGORITHM 20
        5.1.1 Parsing 21
        5.1.2 Saturation 22
        5.1.3 Column Compression 24
        5.1.4 Time Routing 25
        5.1.5 File Generation 28
    5.2 GENERATOR IMPLEMENTATION 32

6. BENCHMARK RESULTS 36
6.1 EXPERIMENTAL SETUP 36
6.2 CELL COUNTS AND FINAL ADDER WIDTH COMPARISON 37
6.3 SATURATED UNSIGNED MULTIPLICATION 37
6.4 SATURATED UNSIGNED ADDITION 41
7. CONCLUSION AND FUTURE WORK 43
REFERENCES 46
APPENDICES 48
CURRICULUM VITAE 49

LIST OF TABLES

Table 6-1: Cell Count and Final Adder Width for Multiplication..38

Table 6-2: Timing Results for Saturated Unsigned Multiplication 40

Table 6-3: Timing Results for Saturated Unsigned Multiplication Continued 41

Table 6-4: Timing Results for Saturated Unsigned Summation 42

Table 6-5: Timing Results for Saturated Unsigned Summation Continued 43

LIST OF FIGURES

Figure 1-1: Different Summation Strategies 2
Figure 3-1: Dot Notation 10
Figure 3-2: Our Notation 10
Figure 3-3: Digital Multiplication Steps 11
Figure 3-4: Full Adder Schematic and Truth Table 12
Figure 3-5: Half Adder Schematic and Truth Table 12
Figure 3-6: First Level Full Adder Placement 13
Figure 3-7: 8 bit by 8 bit Unsigned Multiplication with PCST 15
Figure 4-1: Multiplication of A and B 17
Figure 4-2: Unsigned Multiplication using PCST with Implemented OverflowDetection Logic 19
Figure 5-1: Configurable OR Tree 23
Figure 5-2: Default OR Tree 24
Figure 5-3: Half Adder Delay Paths 26
Figure 5-4: Full Adder Delay Paths 26
Figure 5-5: Timing.log File Example 28
Figure 5-6: Block Diagram of the HDL Generator 31

LIST OF ABBREVIATIONS

Column Compression:	CC
Carry Look-Ahead Adder:	CLA
Carry Save Adder:	CSA
Carry Save Tree:	CST
Electronic Design Automation:	EDA
Full Adder:	FA
Half Adder:	HA
Hardware Description Language:	HDL
Plowing Based Carry Save Tree:	PCST
Register Transfer Level:	RTL
Design Compiler:	DC

1. INTRODUCTION

In this chapter, the scope of thesis is given and the contributions are listed. The history of this thesis and beyond all, the idea that inspired the algorithm of this thesis is told.

1.1 SCOPE AND CONTRIBUTIONS OF THE THESIS

Our thesis introduces a novel Carry Save Adder (CSA) tree algorithm additional to already existing Wallace tree and Dadda tree algorithms. We named this algorithm as PCST (Plowing based Carry Save Tree generation). Due to the wide scope of CSA tree algorithms application areas and possible improvement trials in this we had to narrow the scope. As a result we mainly concentrated over saturated unsigned summation and multiplication using PCST. While doing so we compared cell usage and final adder width with Wallace and Dadda. We applied saturation technique which is first introduced by (Schulte, Balzola, Akkas, Brocato, 2000). But this work was mainly concentrated over multiplication operation and only compared against conventional methods. Additionally to that we tested our algorithm against Wallace tree, Dadda tree versions and the Synopsys DC's DesignWare both for unsigned summation and multiplication.

For functional test and synthesis purposes a fully functional Verilog HDL generator is written which is capable of generating our algorithms RTL, its wrapper, test bench and functional equivalent with high level coding. This generator can be used for other CST algorithms by a small manipulation of the code without necessity of rewriting.

Another important thing about the generator is the capability of time based routing which is a matching process of bits and HA and FA inputs according to the delay of bits and critical path of HA and FA. By this critical path of CST algorithm is optimized.

1.2 WALLACE'S IDEA

Since addition operation of three or more numbers is a complicated and widely used computer arithmetic concept. (Wallace, 1964) brought a new point of view into the already existing picture. Instead of going for the result directly he proposed a method of pre-summation which is composed of several (3:2) reduction stages. In other words every stage of Wallace tree reduces the numbers that are being summed by one over tree percent roughly. This pre-summation gives two numbers instead of one that will produce the result when they are added with a final adder.

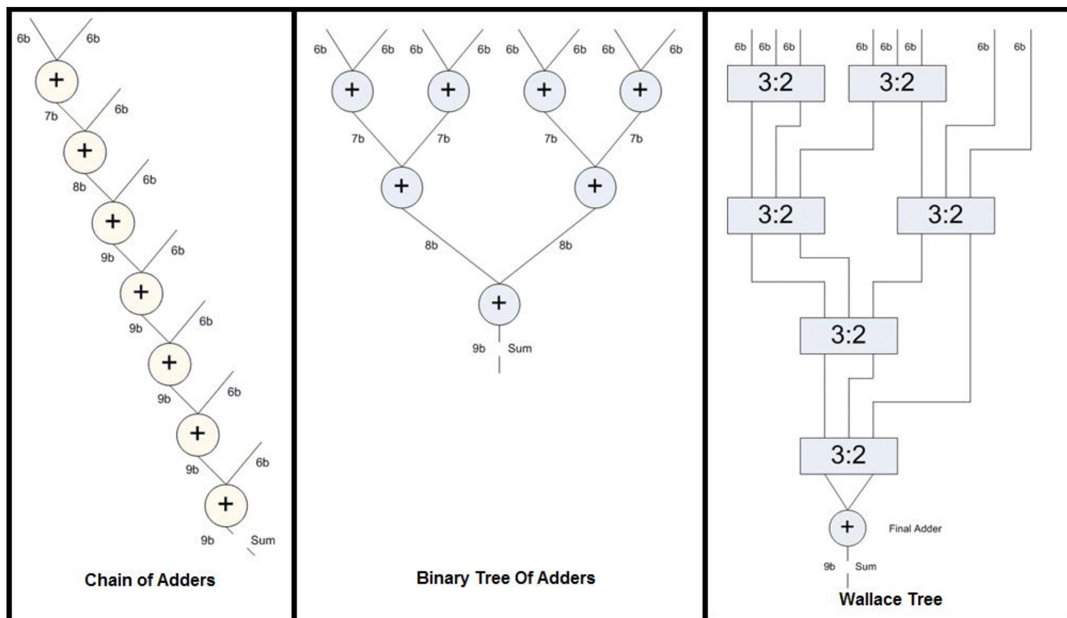


Figure 1-1: Different Summation Strategies

Figure 1-1 shows different summation strategies including the Wallace tree for eight six bit numbers.

A multiplication operation can be thought to understand the benefit of the Wallace tree which has only $O(\log_2 n)$ reduction layers, and each layer has $O(1)$ propagation delay. Producing the partial products is $O(1)$ and the final addition is $O(\log_2 n)$, the multiplication is only $O(\log_2 n)$, not much slower than addition (however, much more expensive in the gate count). But on the other hand naively adding partial products with regular adders would require $O(\log_2 n)^2$ time.

1.3 HISTORY OF THIS RESEARCH WORK

In Bahçeşehir University digital design and related to that computer arithmetic concepts started to gain more importance for the last six years. One of the core topics carry save adder trees which are a widely used method in summation, multiplication and other possible combinations of these arithmetic operations due to this usage area attracted our interest more. For the purpose of providing better understanding of carry save tree algorithms using a new and simpler notation instead of dot notation seen feasible. While working over the new notation a possibility of improving already existing Wallace tree and Dadda tree methods is noticed by Asst. Prof. H. Fatih Uğurdağ. Since that day several pupils of Asst. Prof. H. Fatih Uğurdağ worked over it under his guidance but concluding it to that point is achieved with this master's thesis.

1.4 OUTLINE OF THESIS REPORT

To be able to keep track of this master's thesis report a simple outline is given in this chapter which explains each chapter's content with a few words.

In the second chapter previous work done relevant to our thesis scope is introduced. Also possible improvements suggested for Wallace and Dadda that can also be implemented for PCST are introduced.

In the third chapter PCST algorithm is described by three sub algorithms that construct it and the notation capable of showing levels used cells and their positions are described.

In the fourth chapter conventional approach and the approach we used for the implementation of saturation arithmetic is explained.

In the fifth chapter Verilog HDL generator written for test, synthesis and benchmarking purposes is explained both from algorithmic and implementation perspective.

In the sixth chapter synthesis results of our method and the rival methods are presented for benchmarking of area and timing.

In the seventh chapter due to the results strong and weak sides of our algorithm are commented and a conclusion statement is given. Also possible future improvement and trial ideas are given with reasoning.

2. PREVIOUS WORK

In literature main subject of our interest is defined as Carry Save Adder tree algorithm. Carry Save Addition is used to compute the sum of three or more *n*-bit binary numbers and produces two numbers instead of one as a result. Addition of these two numbers gives the actual result. If the addends are thought as a bit matrix what it does is actually to compress this matrix to a height of two and due to that reason it can be also found in literature as Column Compression tree. We will refer CSA tree as Carry Save Tree (CST).

2.1 CARRY SAVE ADDER TREE

As a result of our literature survey we found two main algorithms for carry save adder tree which are Wallace tree and Dadda tree and different improvements over these algorithms over the years. These improvements mainly concentrate over different counter structure usage, CSA tree delay restructuring for optimum delay and usage of pipelining for these algorithms.

Wallace (1964) defined column compression architecture for fast multiplication operation. Multiplication starts with the partial product generation in parallel using AND gate array. Than partial products are reduced to two numbers by applying (3,2) and (2,2) counters and this reduction algorithm is known as Wallace tree. Finally the two numbers are added using a fast carry propagate adder. Column compression multipliers are faster than array multipliers because their total delays are proportional to the logarithm of the operand word length while the other one grows linearly with operand size.

Dadda (1965) refined Wallace's method by defining a unique counter placement strategy for reduction stages. This strategy is known as Dadda tree. Additionally Dadda brought the bit level approach to the problem against Wallace's word-level approach. Dadda's technique minimizes the use of (3,2) and (2,2) counters but the resulting fast carry propagate adder is larger. He also considers the use of higher order counters like (7,3) and (15,4) in his paper.

Wang, Jullien, Miller (1995) presented a new design technique for column-compression (CC) multipliers. This design technique brings considerable flexibility for implementation of the CC multiplier, including the allocation of adders and choosing the length of the final fast adder.

In Itoh, Naemura, Makino, Nakase, Yoshihara, Horiba (2001), they present an efficient layout method for a high- speed multiplier. In order to do so, a rectangular Wallace-tree construction method is proposed which will reduce the area used. This method divides the partial products into two groups and adds them up in the opposite direction.

Paterson, Pippenger, Zwick (1992) defined a general theory to obtain the shallowest depth for a carry save network (i.e. CST). For multiplication, multiple addition and multiple carry save addition upper bounds and a restricted form of lower bound obtained.

Townsend, Swartzlander, Abraham (2003) performed a gate level comparison of Wallace and Dadda multiplier areas and timing. A delay methodology is presented to route bits to full adder and half adder inputs considering their critical path difference. As a result of this delay methodology, it is shown that against the common belief Dadda multiplier results has better timing than Wallace multiplier.

Dadda (1976) analyzed the feasibility of parallel counters for a large number of inputs. He did not offer an exact solution for the optimization problem but due to the advancing technology, considered the use of counters with large number of inputs.

He also gave design strategies for some of them. His work is useful in reducing the design complexity.

Mehta, Parmar, Swartzlander, (1991) designed and discussed a novel scheme for parallel multiplication using (7,3) counter circuits. As a result of this study it is shown that parallel multipliers implemented using (7,3) counters have better performance than parallel multipliers designed by using (7,3) compressors.

Oklobdzija and Villeger (1993) discussed ways of compressing the bits of the multiplier tree. Different counters of (3,2), (4,2), and (9,2) are implemented, and their delay profile are compared. As a result, usage of (3,2) counter is warranted.

Oklobdzija and Villeger (1995) extended the content of their paper in 1993. They applied different counters to flatten the delay structure of column compression tree output and reduce the longest delay path.

Usage of counters with large number of inputs has been a subject since Dadda's papers Dadda (1965), Dadda (1976) and after that there has been several research related to optimizing multiplication by usage of different counters. This research topic is also directly related with the technological advancement because as long as timing and area properties of these counters improve their feasibility for CSA tree algorithms needs to be researched.

Zimmermann (2009) summarized the circuit architectures and techniques used in a commercial synthesis tool to optimize cell-based datapath netlists for timing, area and power. Several ideas for carry-save addition and Wallace Tree implementation are also given.

Kim, Jao, Jiang (1998) defined a relationship between the properties of arithmetic computations and used CSA's to derive better results than manual implementations. They introduced two important concepts, operation duplication and operation split.

These are the main driving techniques of their algorithm which are used for an extensive utilization of CSA's.

Capello and Steiglitz (1983) specified a design first for a pipelined parallel counter, and then for a complete multiplier. They analyzed the complexity of the resulting design using a VLSI model of computation, showing that it is optimal with respect to both its period and latency.

Breveglieri, Dadda, Piuri (1995) presented a study over the introduction of pipelining in parallel VLSI multipliers that use column compression (CC) design techniques. They afforded to introduce pipelining for several column compression (CC) design techniques and compare them in terms of required number of components and operation frequency

Pipelining is used to increases the throughput of the system when processing a stream of data. In digital design it is an important concept for operations that require high speed. As the main area of CSA tree algorithms pipelining a multiplication or summation operation is critical.

2.2 SATURATED SUMMATION

Schulte, Balzola, Akkas, Brocato (2000) presented efficient methods for performing unsigned or signed (i.e. two's complement) integer multiplication with overflow detection or saturation. These methods require significantly less area and delay than conventional methods for integer multiplication with overflow detection or saturation.

Gok, Schulte, Balzola (2001) presented a method for integer multiplication with overflow detection for unsigned and two's complement numbers. Another method for combining unsigned and two's complement integer multiplication with overflow detection is also presented. Both of methods result in more efficient circuits than previous ones both in number of gates and delay.

3. PLOWING BASED CARRY SAVE TREE (PCST) GENERATION

In this chapter the algorithms and notation that construct the Plowing based Carry Save Tree generation (PCST) are explained. First the notation we found and used during this research is explained. This notation will be used in several chapters and it's the output format of Verilog HDL generator. Second the source of the idea behind PCST and its structure is explained. Third the calculation of guidance value that affects the flow of PCST at each level is explained. As last the plowing algorithm that targets to minimize the size of the final adder is explained.

3.1 NOTATION

Most commonly used notation for CSA tree algorithms is the dot notation developed by Dadda and in this thesis this dot notation is also used. Dadda's notation is a graphic based notation as shown in Figure 3-1. Here dot notation is used to show a multiplication operation of two eight bit unsigned numbers by Dadda's method. Each dot represents a bit, two dots joined by a diagonal line means that these two dots are the outputs of a (3,2) counter and similarly two dots joined by a crossed diagonal line means that these two dots are the outputs of a (2,2) counter.

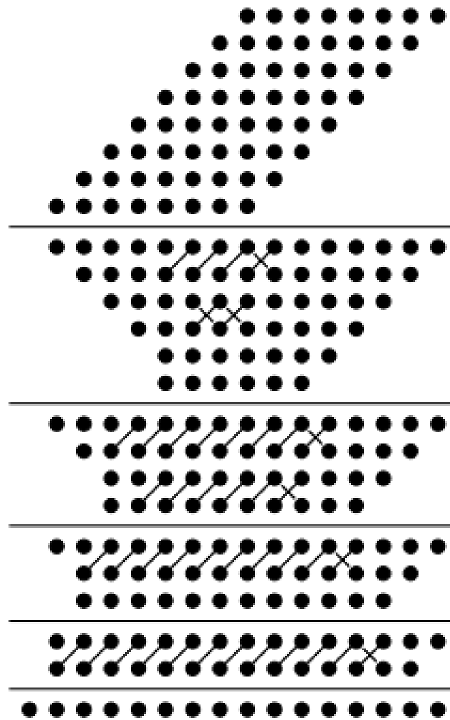


Figure 3-1: Dot Notation

Instead of dot notation our notation is a text based notation that is capable of showing bits, half adders and full adders used at each level like dot notation and additionally it shows the guidance value for each level. The term “guidance” is explained briefly in Chapter 3.3. The same multiplication operation given at Figure 3-1 is shown at our notation in Figure 3-2.

```
0  1  2  3  4  5  6  7  8  7  6  5  4  3  2  1  b
0  0  0  0  0  0  1  2H 2H 1H 0  0  0  0  0  0  c
0  1  2  3  4  6  6  6  6  6  6  5  4  3  2  1  b 6
0  0  0  0  1  2  2  2  2  2  2H 1H 0  0  0  0  c
0  1  2  4  4  4  4  4  4  4  4  4  4  3  2  1  b 4
0  0  0  1  1  1  1  1  1  1  1  1  1H 0  0  0  c
0  1  3  3  3  3  3  3  3  3  3  3  3  3  2  1  b 3
0  0  1  1  1  1  1  1  1  1  1  1  1  1H 0  0  c
0  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1  b 2
```

Figure 3-2: Our Notation

The lines that end with b character represent the number of bits at each column of that level.

The lines that end with c character shows the cells (FA's and HA's) that uses the bits shown above as inputs. For the lines that show the cells if there is just a number like x this means x full adders are used at that column if the number is used with a H character next to it like xH this means x-1 Full Adders and a Half Adder is used for that column. The guidance values for a level are given next to the line that shows the bits of that level.

3.2 OVERVIEW OF PCST

The carry save adder tree algorithms mainly targets to compress an addition process to two binary numbers that will be added by a final adder in the most optimized way from both area and timing perspective. Due to that reason they are commonly used at addition and at the second step of multiplication operations. A multiplication operation can be defined in three steps which are forming a partial product matrix, reducing this partial product matrix to a height of two and as last adding up these two rows by the help of a final adder as shown in Figure 3-3.

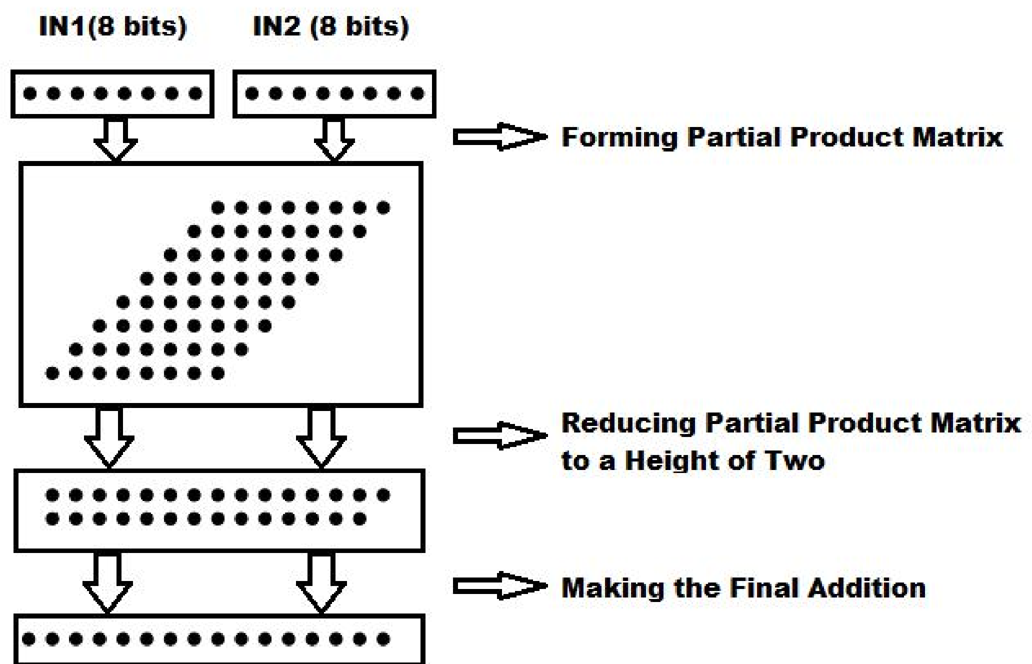
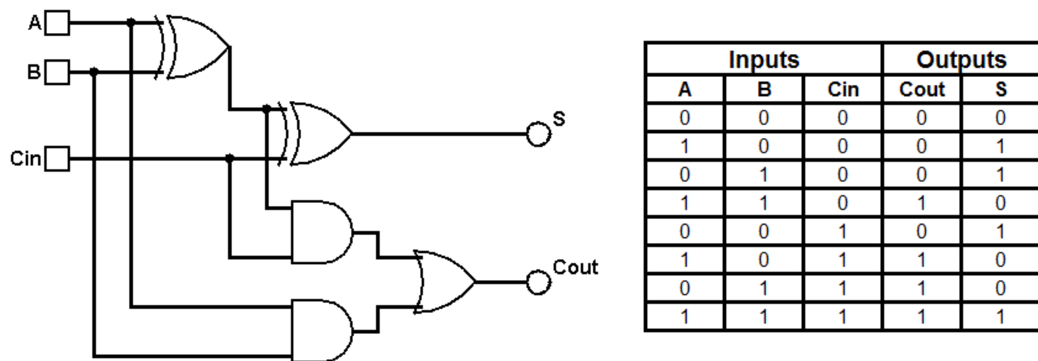


Figure 3-3: Digital Multiplication Steps

For the reduction operation there are several counters that are mainly used. These are (3,2) counters and (2,2) counters. Although other sizes of counters are possible in this thesis scope full adder as a (3,2) counter and half adder as a (2,2) counter are used. The truth table and schematic of full adder is given in Figure 3-4. The truth table and schematic of half adder is given in Figure 3-5.



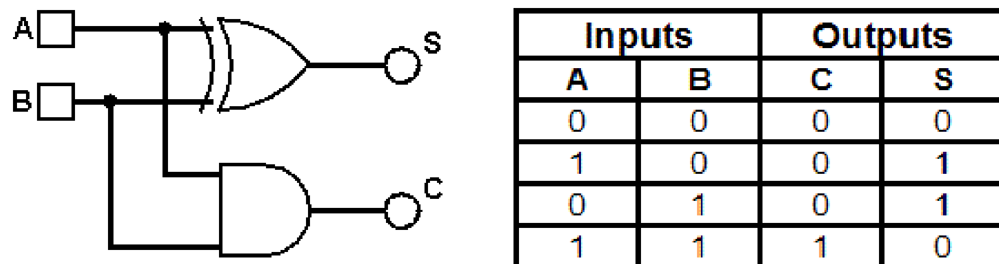
Inputs			Outputs	
A	B	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

Figure 3-4: Full Adder Schematic and Truth Table

Following Boolean functions that a full adder circuit performs can be obtained from the truth table given at Figure 3-4.

$$S = A \oplus B \oplus C_{in} \quad \textbf{(3.1)}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B)) \quad \textbf{(3.2)}$$



Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

Figure 3-5: Half Adder Schematic and Truth Table

PCST can be defined in three steps which are deciding usage of full adders, checking guidance and using half adders if necessary and plowing decision. Second and third steps will be explained parallel to the explanation process of “guidance” and “plowing” terms in related chapters. These three steps are applied at each level until the height of bit matrix is reduced to two.

PCST groups each three bits at every column and inputs them to a full adder. The total number of full adders used by PCST can be calculated by using the Formula 3-3.

$$T = \sum_{i=0}^{N} \sum_{j=0}^{M_i} \left\lfloor \frac{X_{ij}}{3} \right\rfloor \quad \textbf{(3.3)}$$

T = Total number of full adders

N = Number of levels

$\mathbf{M_i}$ = i’th levels bit width

$\mathbf{X_{ij}}$ = Number of bits at j’th column of i’th level

Figure 3-6 shows this operation for the first level of two eight bit unsigned numbers multiplication operation.

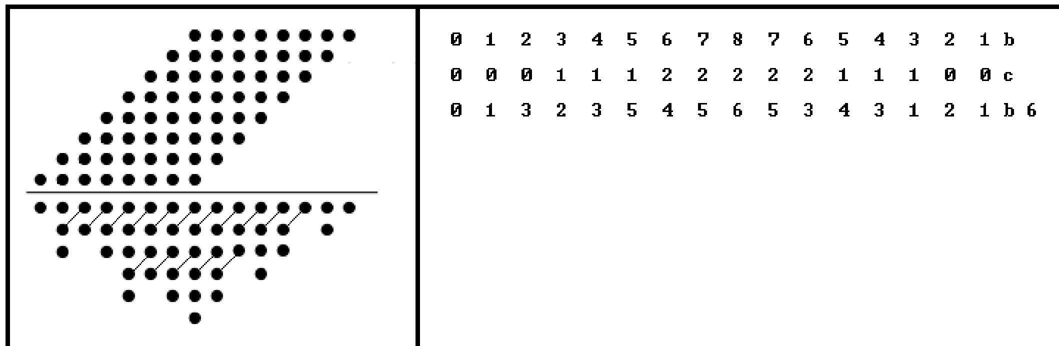


Figure 3-6: First Level Full Adder Placement

3.3 GUIDANCE

After placement of the cells a certain value for the number of bits at each column of the bit matrix shouldn't be exceeded for the next level.
This value is called the guidance value because it also determines the flow of PCST and optimum number of levels required. This value can be calculated as given at Formula 3-4 for each level.

$$G_{i+1} = H_i - \left\lfloor \frac{H_i}{3} \right\rfloor \tag{3.4}$$

G_i = Guidance for the i'th level
H_i = Height of the i'th level

At each level after placement of full adders height of the bit matrix needs to be checked and if it exceeds the guidance value a half adder is added to each necessary column. The half adders added as a result of the guidance check are taken into green circles in Figure 3-8.

3.4 PLOWING

As explained in Chapter 3.2 both at multiplication and addition after the use of a CSA tree algorithm a final adder is needed to get the final result. At this point the size and the critical path of this final adder are determined by the width of the CSA tree's output. To minimize the size of the final adder plowing aims carrying two bits on the right of the bit matrix to left using half adders at each level. Number of used half adders, full adders and the final adder width are given for Wallace tree, Dadda tree and PCST for several unsigned multiplication operations in Table 6-1.

For deciding these half adders at each level starting from zero index bit numbers at each column are checked if the number is equal to two a half adder is placed to that column this check is kept for that level until the bit number exceeds two. The half adders added as a result of the guidance check are taken into red circles in Figure 3-7.

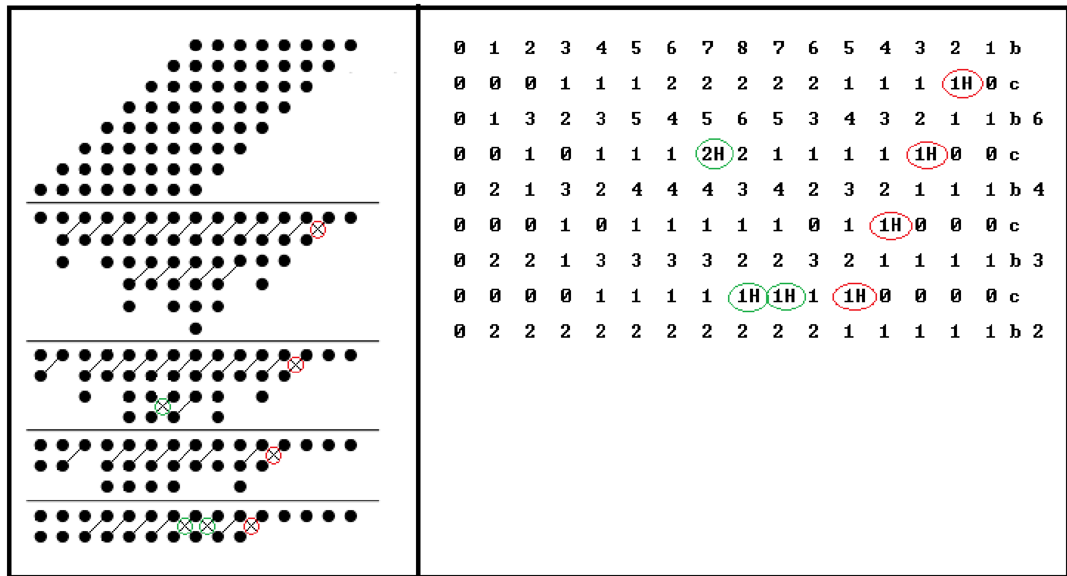


Figure 3-7: 8 bit by 8 bit Unsigned Multiplication with PCST

4. SATURATED UNSIGNED SUMMATION AND MULTIPLICATION

Saturation arithmetic is used to limit the result of arithmetic operations like saturated unsigned summation and multiplication to a fixed range between a minimum and maximum value. If the result exceeds the maximum it is set to the maximum, while if it is below the minimum it is set to the minimum. The name comes from how the value becomes "saturated" once it reaches the extreme values; further additions to a maximum or subtractions from a minimum will not change the result.

In this chapter conventional method for saturation and the method we applied is explained. The saturation operation we used saturates to a maximum value of $2^n - 1$ if the result exceeds the desired bit length n. More detailed information can be obtained from (Gok, 2000) and (Schulte, Balzola, Akkas, Brocato, 2000)

4.1 CONVENTIONAL SATURATION APPROACH

In Figure 4-1 a multiplication operation of two n bit unsigned numbers is shown which produces a 2n bit product. Operands and the result are shown below.

$$\begin{aligned} A &= a_{n-1}\, a_{n-2}\, a_{n-3} \cdots\ a_1\, a_0 \quad (Multiplicand) \\ B &= b_{n-1}\, b_{n-2}\, b_{n-3} \cdots\ b_1\, b_0 \quad (Multiplier) \\ P &= p_{2n-1}\, p_{2n-2}\, p_{2n-2} \cdots\ p_1\, p_0 \quad (Product) \end{aligned} \qquad \textbf{(4.1)}$$

Overflow occurs when the actual product needs more than n bits while the result is desired to be n bits. In other words overflow occurs if the product is greater than or equal to2^n.

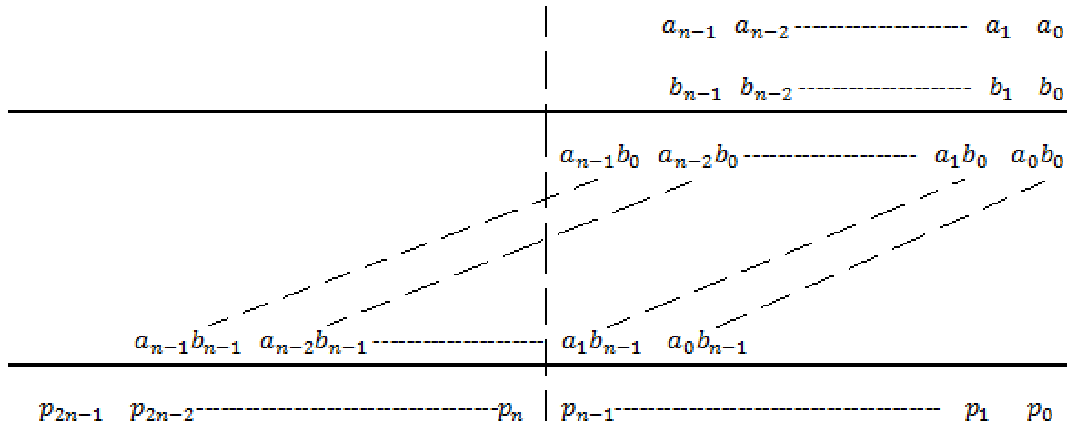


Figure 4-1: Multiplication of A and B

Conventional methods detect the overflow after the 2n bit result is produced. Overflow detection is done by ORing together the most significant n bits $p_{2n-1} \dots p_n$ as shown in Equation 4.2. The O value is equal to one if the overflow occurred.

$$O = p_{2n-1} + p_{2n-2} + p_{2n-2} + \dots + p_n \; (Overflow\ Bit) \qquad \textbf{(4.2)}$$

After the calculation of overflow bit if the saturation is required saturated value can be computed by ORing n least significant bits of the product individually with overflow bit as shown in Equation 4.3. Bits in parenthesis like (p_x) shows the saturated bits.

$$(p_x) = p_x + O \quad 0 \le x \le n-1 \qquad \textbf{(4.3)}$$

This operation sets the product to $2^n - 1$ if the overflow occurred.

This approach is applicable to addition operation too. But calculating the final result of the addition or multiplication operation leads to unnecessary area and delay.

4.2 OUR IMPLEMENTATION

The method we used is first introduced by (Schulte, Balzola, Akkas, Brocato, 2000). Method calculates the overflow bit parallel to the CSA tree levels. To do so at each level of CSA tree bits that belong to a higher column index than the desired saturation index are ORed to reduce the bit count systematically. Figure 4-2 shows an eight bit by eight bit unsigned multiplication operation with PCST where the symbol "□" output of a two input OR gate. Desired maximum saturation output is eight bits in other words result will be equal to $2^8 - 1$ if the overflow occurs.

The OR gates used for the overflow detection can have different input numbers but the critical point here is to keep the delay of the OR gates less than or equal to the delay of the partial product reduction stages. Doing so causes the worst case delay for overflow detection to be equal to the delay of partial product generation , plus partial product reduction, plus CLA (Carry Look-ahead Adder) delay plus one OR gate delay to include the final carry out.

For the given example in Figure 4-2 after the overflow bit detection instead of using a multiplexer overflow bit can be ORed with the n least significant bits of the product to get the result.

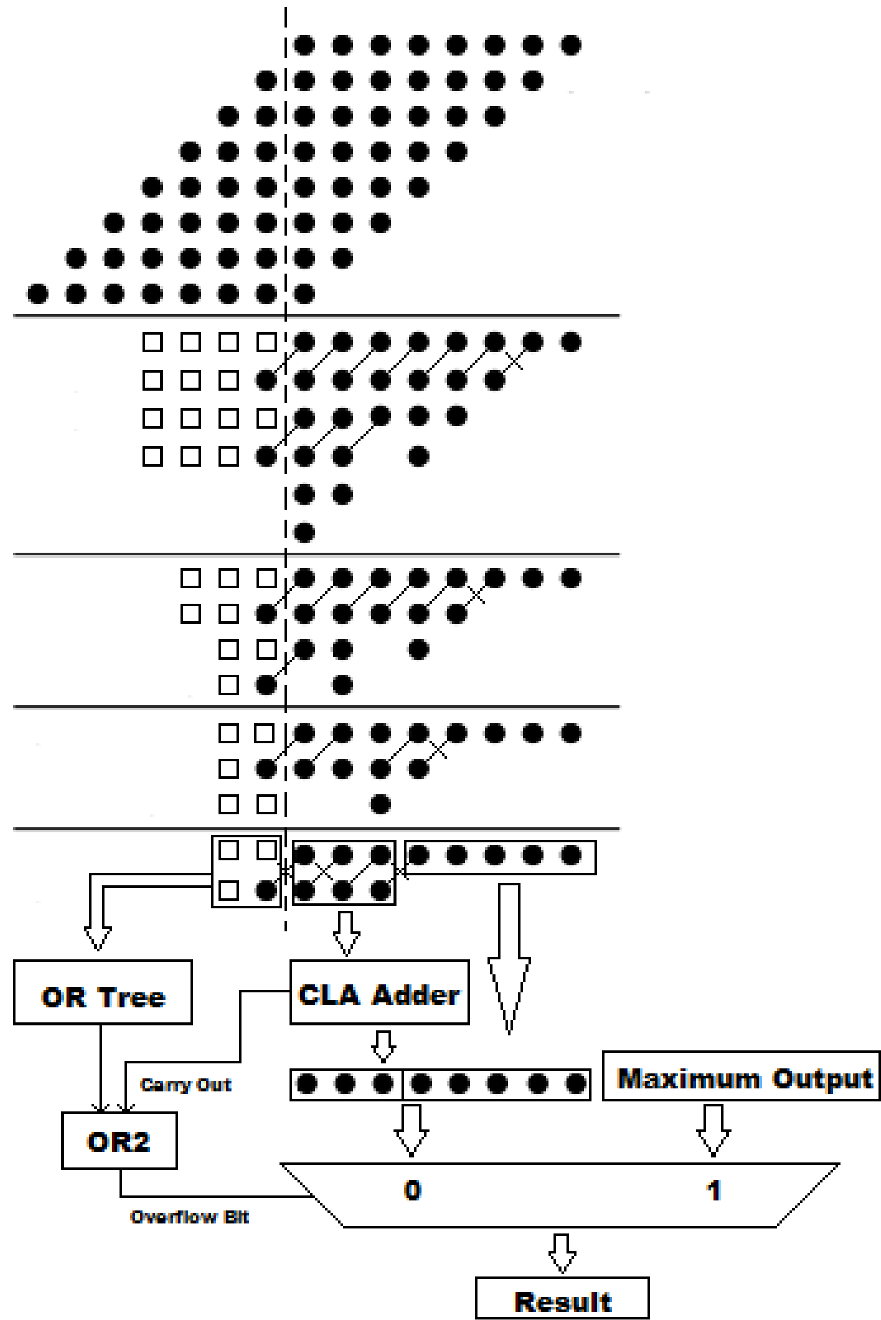


Figure 4-2: Unsigned Multiplication using PCST with Implemented Overflow Detection Logic

5. HDL GENERATOR

In this chapter algorithm and implementation of the console based script which is designed for the purpose of generating the Verilog HDL files for synthesis and test is explained. This script is purely written in Perl. It is capable of generating Verilog HDL files for multiplication and addition saturation using PCST but using VHDL instead of Verilog is just a matter of syntax. It can easily be changed to VHDL. There are several supported input types and switches for this script that enable different routing options for PCST and OR tree structures for saturation. All the generated Verilog files are created according to Verilog-95 syntax. Block diagram of the generator is shown in Figure 5-6.

5.1 GENERATOR ALGORITHM

Algorithm of the script can be divided into five sub blocks. First block is the parsing block which takes the arguments and separates them to set the inputs and switches. Second one is the saturation block which arranges the OR tree structure according to the related switch. Third block is the column compression block which applies the PCST algorithm. Fourth block is the time routing block that decides the input connections of half adders and full adders according to bit and cell delays. Fifth and the last block is the file generation block which generates necessary Verilog HDL files and some log files.

5.1.1 Parsing

Script gets two mandatory inputs and two optional inputs with switches. Mandatory inputs needed to be given in order first the input type and second the width of saturation output. Then two possible switches can be used in any order as the below format.

>Mult.pl <input_type> <saturation_output_width> <switch_1> < switch_2>

The three possible input types:

- MxNb (Adds up M, N bit unsigned numbers)
- MbxNb (Multiplies an M bit unsigned number with a N bit unsigned number)
- “x1 x2 x3 x4 …” (Space separated numbers between commas. To show the first level bits of an unsigned summation operation)

The two possible switches:

- -r K (r switch means the user wants to use random routing for PCST with randomization seed K if not used time routing algorithm will be used)
- -t L (t switch means the user wants to use OR trees with L bit inputs at max if not used default OR tree structure is used)

Parsing algorithm is designed for understanding the input type, saturation output width, switches if any used and to store and make the sanity check of the values entered. The algorithm first decides the input type using regular expression checks. Then according to the input type it calculates the maximum output width of the multiplication or addition operation. At the same step it also forms the bit matrix of the first level that will be processed by the saturation and the PCST algorithms. The saturation output width is stored and checked if it is smaller than the maximum output width otherwise code terminates with an error message. After this if any switch is used entered values are stored after a sanity check and related flags are set. If anything is wrong with the switch format code terminates with an error message.

5.1.2 Saturation

Saturation algorithm works in a loop with the PCST algorithm until the height of the bit matrix which is formed at parsing step is reduced to two. The algorithm needs to keep the bit names that will enter the OR tree structure in an array for the verilog HDL generation.

These bits can come from the first level can be the outputs of the OR tree structure of the previous level or the carry out bits of the full adders or half adders of the previous level. For that reason algorithm first checks the level if it is the first level it stores the names of the bits that have a higher column index than the saturation output width otherwise it stores the carry out bits that come from the previous levels full adders and half adders . Then the bit numbers of the columns of the bit matrix that have a higher index than the saturation output width are set to zero because these bits will not be processed by the PCST algorithm. The height of the bit matrix is checked if it is higher than two then the switch flag is checked. If a value is set by the user the algorithm calculates how many trees needed to be formed for that level. Then starts to form the OR trees by grouping bits by two using the assign statement of Verilog. If a single bit remains it directly passes to next level without any operation. To do this the array that keeps the bit names is used just like a FIFO. Used bits are popped from the array and newly formed bits are pushed to the array and this brings a flexible coding if someone wants to try a new OR tree structure. The OR tree formed for the first level of a eight bit by eight bit unsigned multiplications first level is shown in Figure 5-1. At this example saturation output width is set to ten and maximum or tree width is set to eight.

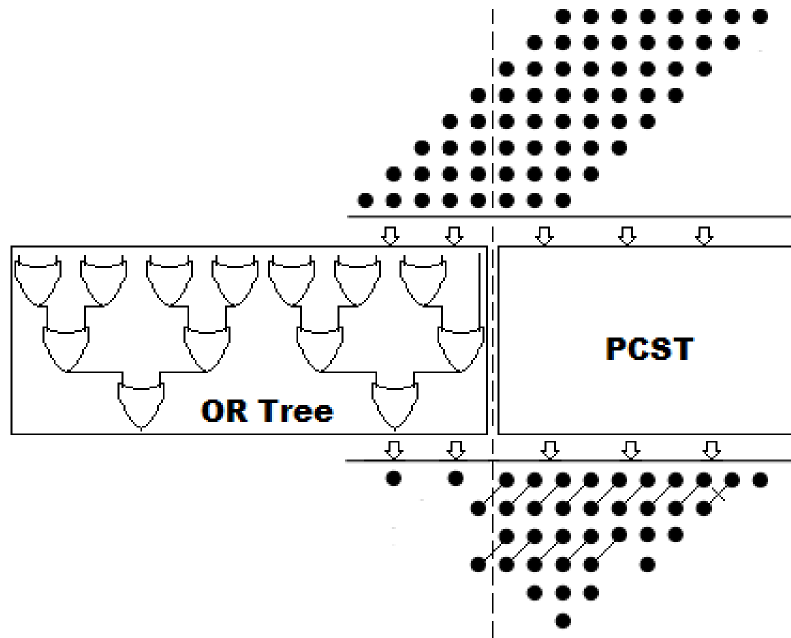


Figure 5-1: Configurable OR Tree

The main purpose of this maximum or tree width switch is to arrange the OR tree depth to set the OR tree delay. By this optimum OR tree structure can be formed for different synthesis libraries to keep OR tree delay less than the PCST delay of the same level while maximizing the bits that enter the OR tree. If the user has not used the -t switch to set a value for maximum or tree width algorithm places a default OR tree structure. This structure formed by OR4's and OR2's of the used library. Every four bits of the level are fed as input to OR4's and the remaining bits are fed to an OR2 if possible. If a single bit remains it directly passes to next level without any operation. The default OR tree formed for the first level of an eight bit by eight bit unsigned multiplications first level is shown in Figure 5-2.

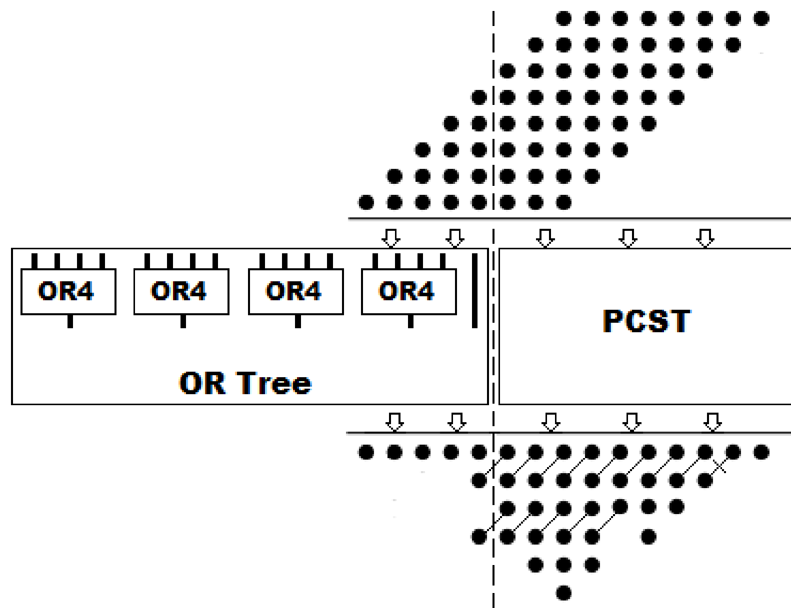


Figure 5-2: Default OR Tree

If the height of the bit matrix is equal to two algorithm creates a final or tree which will work parallel to the final adder. At this step even if the –t switch is used algorithm does not take into account the maximum or tree width and inputs all bits to a OR tree that groups bits by two using assign statement of Verilog and outputs a single bit.

Otherwise it uses OR4's, OR3's and OR2's to create this single bit. The reason for OR3 use at this step is to prevent an extra level creation at this step.

Name of the single bit generated parallel to the final adder is kept to be ored with the carry out of the final adder to decide if the result will be saturated or not.After this step code exits from the saturation and PCST algorithm loop and enters to routing block.

5.1.3 Column Compression

PCST algorithm first determines the guidance value for the next level by using the current levels bit matrix height using the Formula 3-4. To do so it uses return values of a function which takes the bit matrix array and returns the max bit value (height of bit

matrix) and the index of this value. To decide the half adders positions that will be used for plowing a loop starts to check the bit values starting from the zero indexes up to the index of the max bit value if it sees a value bigger than two gives up the search for that level otherwise for each value equal to two it keeps a flag for that column index. At this step the bit matrix is not modified. As the next step algorithm checks the number of bits at each column and determines the number of full adders that will be used for each column. The number of full adders used at each column is stored at a one dimensional array and the bit matrix is modified according to the result of these full adders. Then the height of the bit matrix is checked and if it exceeds the guidance value a half adder is added to that column and the bit matrix is modified. To finalize the PCST the pre calculated half adders for plowing are used to modify the bit matrix and recorded to the array that keeps the full adders and half adders.

5.1.4 Time Routing

Regarding full adders and half adders as black boxes to which all inputs should be supplied simultaneously and which then returns all outputs after a fixed delay is a naive approach. In truth the full adders and half adders do not produce their outputs simultaneously, nor do they require the inputs to be fed at the same time.

This creates an opportunity to optimize total delay of a carry save adder tree. For this reason we brought a theoretical approach to minimize PCST's total delay and applied this approach at our script as the default algorithm to generate Verilog HDL. The optional algorithm makes this routing randomly. A full adder has six different delays which are composed of three inputs to sum delays and three inputs to carry out delays as shown in Figure 5-3.

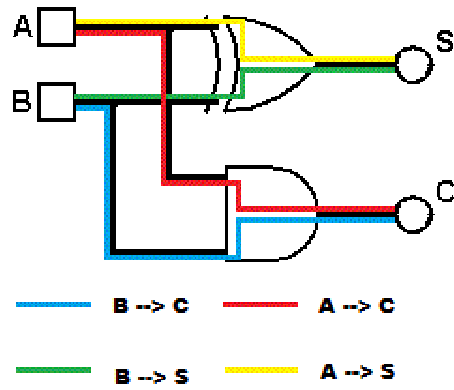


Figure 5-3: Half Adder Delay Paths

Similarly a half adder has four different delays which are composed of two inputs to sum delays and two inputs to carry out delays as shown in Figure 5-4. Main aim of our approach is to match a full adder or a half adder's input that has the highest delay with the bit which has the lowest delay and repeat this until all inputs are matched according to their delay in a descending order. For that reason at first step delays belong to the same input are compared and the higher delay is taken into account for half adder and full adder. This operation reduces ten possible delays to five and then these delays are sorted from maximum to minimum and stored in a delay array.

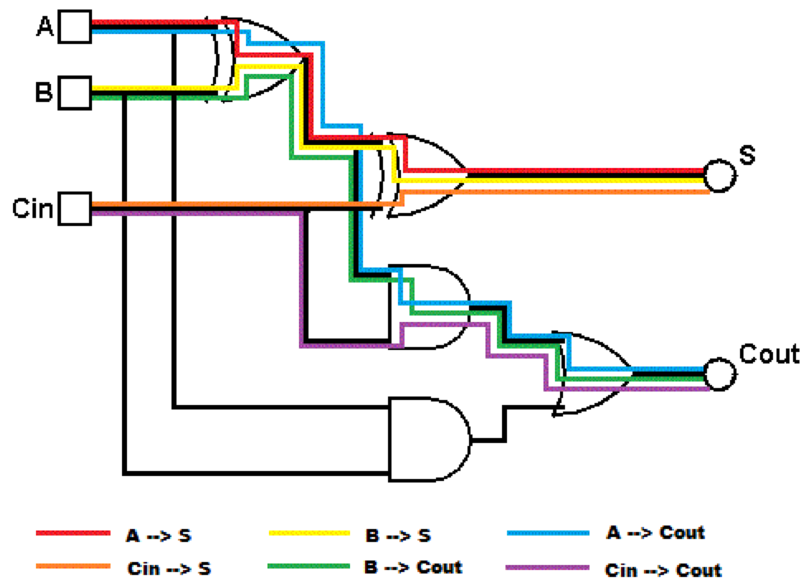


Figure 5-4: Full Adder Delay Paths

These delays changes according to the architecture and silicon technology so the user needs to enter the values for the synthesis library used to get meaningful results.

As the next step a loop starts that decides the routing for each level of PCST. In this loop at firstly the –r switch is checked if the user has entered a seed value for randomization. If so each column of the bit matrix for that level is shuffled randomly. This is the step that creates the random routing. Otherwise the bits at each column are sorted according to their delay value in an ascending order for the current level.

Secondly for each column each path delay starting from the highest stored in the delay array is compared with the cell array elements. If the delay belongs to the same cell the bit name is matched with the cell input of the corresponding delay and the bit is marked as a used bit. This used bit information is used at third step to create next levels bit array. At that point the delay of the carry out bit and the sum bit of the cell needs to be determined. For that reason used bit's delay is summed up with the path delay and if it is an input to sum delay it is compared to other input to sum delays plus the matched bit delay the highest one is stored as the delay of the sum bit of that cell. Same comparison is made for determining the delay of the carry out bit of that cell.
When this matching operation is finished what we have at hand is the cells of the current level with the input bits and the maximum delay values for the carry out and sum bits.

As the third and the last step of the rooting loop, bits of the next level needs to be constructed with the name and delay information. The bits that construct the next level can be sum bits of the cells at the same column, carry out bits of the cells of the previous column and the unused bits of the same column of the current level for that reason at first the column index is checked. If it is the first column bits can only be sum bits and unused bits. To decide sum bits cell array is checked and for each cell a sum bit is created with the pre calculated delay value. And the unused bits are copied for the next level calculations. If it is not the first level same operation for creating sum bits is done and additionally cell arrays previous column index is checked to create carry out bits with the pre calculated delay.

After the rooting loop completed all the necessary information is obtained to move file generation block.

5.1.5 File Generation

File generation block generates several files for synthesis, simulation and informative purposes. Generated files and their content can vary according to the input of the script. All possible generated files will be described in this chapter according to their generation order.

Timing.log: After the completion of routing algorithm script first generates this file. It contains the delay information of each bit at each level of PCST. Format of the file is shown in Figure 5-5. Generation of this file is not related to any input type or switch.

```
0.625  0.625  0.625  0.625  0.625  0.625  0.625  0.625
0.625  0.625  0.625  0.625  0.625  0.625  0.625
0.625  0.625  0.625  0.625  0.625  0.625
0.625  0.625  0.625  0.625  0.625
0.625  0.625  0.625  0.625
0.625  0.625  0.625
0.625  0.625
0.625

0.625  0.625  1.69   0.625  0.625  1.25   1.635  0.625
0.625  1.69   2.535  0.625  1.69   2.535
1.69   1.69   2.535  1.69   2.535
1.69   2.535         2.535
2.535  2.535
2.535


2.755  2.535  2.755  2.535  3.16  3.545  1.635  0.625
3.6    2.535  4.445  2.535  3.6
3.6    3.6           3.6
       3.6



4.665  3.6    2.755  4.225  4.61  3.545  1.635  0.625
5.51   4.445  4.445  4.445
              4.665



6.52  5.455  6.355  5.455  4.61  3.545  1.635  0.625
5.07  5.73   5.07
```

Figure 5-5: Timing.log File Example

FullAdder.v: Contains hardware description of a full adder written in Verilog. Used for simulation and synthesis purposes and can be replaced with design ware equivalent at synthesis. Generation of this file is not related to any input type or switch.

HalfAdder.v: Contains hardware description of a half adder written in Verilog. Used for simulation and synthesis purposes and can be replaced with design ware equivalent at synthesis. Generation of this file is not related to any input type or switch.

OR4.v: Contains the hardware description of an OR gate with four inputs written in Verilog. Used for simulation and synthesis purposes and can be replaced with design ware equivalent at synthesis. Only generated if the –t switch is not used.

OR3.v: Contains the hardware description of an OR gate with three inputs written in Verilog. Used for simulation and synthesis purposes and can be replaced with design ware equivalent at synthesis. Only generated if the –t switch is not used.

OR2.v: Contains the hardware description of an OR gate with two inputs written in Verilog. Used for simulation and synthesis purposes and can be replaced with design ware equivalent at synthesis. Only generated if the –t switch is not used.

Wrapper.v: Contains the wrapper that instantiates the top module of the design to flop its inputs and outputs for clock definition at synthesis. Always generated and its content changes according to the input type and saturation output width.

Add.v: Contains the hardware description of the final adder that takes PCST blocks outputs as input written in Verilog. For this addition operation a Design Ware Library IP can be used by the infer method of Synopsys DC. Our script infers a Brent-Kung adder from the Design Ware Library. This file is always generated and its content does not change by the input type or the switches.

Saturate.v: Contains hardware description of the same functionality with a high level coding style in Verilog. Its content changes according to the input type and saturation output width. This file is always generated.

Mult.v: Contains the top module which instantiates OR gates, half adders, full adders and the final adder. This file is always generated according to the input type, saturation output width and the used switches.

TestBench.v: Contains the test bench written in Verilog that verifies the correctness of the design. Generation of this file is not related to any input type or switch.

PCST.log: Shows the PCST levels with our notation which is defined in Chapter 3-1. Format of the file is shown in Figure 3-2. Generation of this file is not related to any input type or switch.

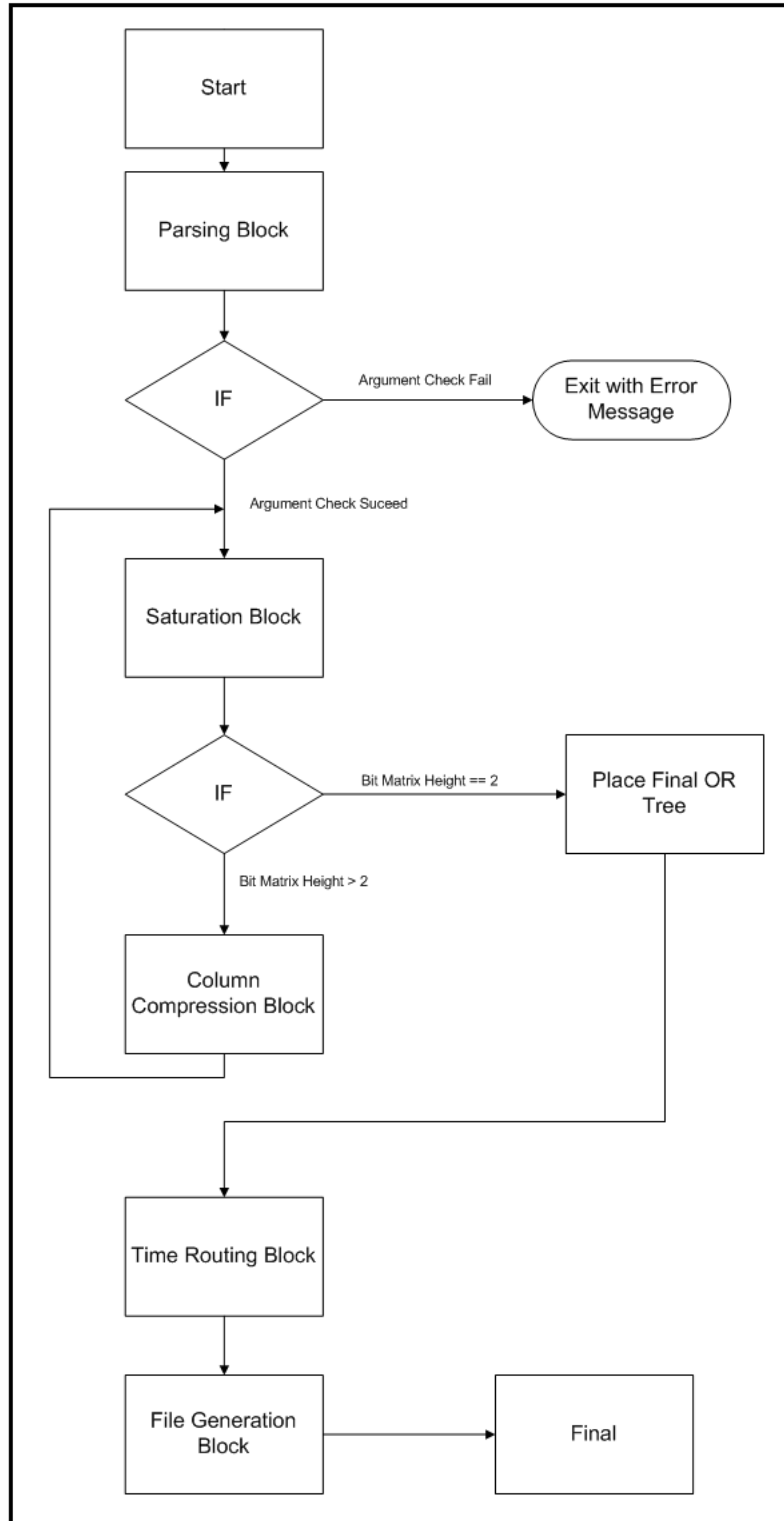


Figure 5-6: Block Diagram of the HDL Generator

5.2 GENERATOR IMPLEMENTATION

In the data structures given in this chapter, bit notations and array indexing are given with purposes for better understanding of the script. Additionally to that error messages that the script can print are explained with their reasons. Code starts with the definition of structures for keeping several related data as a pack necessary for sorting and other operations.

Each bit related to the PCST is kept with a structure named Bit which has a name element, delay element and an element named in as shown below.

```
struct Bit =>
{
        name => '$',# Example “Bit_0_2_1”
         delay => '$',# Example “1.625”
         in => '$', # can be used or un used bit (used: 1, un used: 0)
};
```

Name element keeps the name of the bit which is used at Verilog generation. Bit names have a notation as “Bit_1_2_3”. The first number after the “_” character is the level number that the bit belongs, second number is the row position of the bit and the third number is the column position of the bit. But there is an exception to this notation. If a bit is not fed as an input to a cell it directly passes to the next level without a name change. This is to avoid unnecessary assignments.

Delay element is to keep the delay values of the bits for the sorting made at the routing block of the algorithm.

In element is set if the bit is fed to a cell. By checking this element unused cells are directly passed to the next level.

Information related to the delay paths of full adder and half adder cells are kept with a structure named Delay which has a celltype element, outtype element, bitno element and delay element as shown below.

```
struct Delay =>
{
        celltype => '$',#can be halfadder or full adder (FA:1, HA:0)
        outtype => '$',#can be sum delay or cout delay (Sum:1, Cout:0)
        bitno => '$',#bit number(in0:0, in1:1, in2:2)
        delay => '$',#delay amount
};
```

Celltype element is to keep the information if the delay belongs to a full adder or a half adder. It is used at the routing block of the algorithm to match the delay with the correct cell type.

Outtype element is to keep the information if the delay is input to sum delay or an input to carry out delay.

Bitno element is to keep the information of which input does the delay belong to. This information is also used at routing block of the algorithm to match bit with the right input of the cell.

Delay element keeps the delay value and it is first used to reduce ten possible delays to five as explained in Chapter 5.1.4. Then it is used at the routing part of the algorithm to calculate the output bit delays of the cells.

Information related to full adder is kept with a structure named FA which has an in element, sum element, cout element, maxsumdelay element, maxcoutdelay element and a celltype element as shown below.

```
struct FA =>
{
	in => '@',#Example (Bit_0_1_5, Bit_0_2_5, Bit_0_3_5)
	sum => '$',
	cout => '$',
	maxsumdelay => '$',
	maxcoutdelay => '$',
	celltype => '$',#Cell type 1
};
```

In element is an array with size of tree which keeps the names of the bits matched with the inputs. These bit names are decided at the routing block of the algorithm and used at the file generation block to create full adder instances.

Sum element is to keep the name of the sum bit generated by the full adder. It is decided at the routing block of the algorithm and used at the file generation block to create full adder instances.

Cout element is to keep the name of the carry out bit generated by the full adder. It is decided at the routing block of the algorithm and used at the file generation block to create full adder instances.

Maxsumdelay element is to keep the critical path delay of the sum output. This value is calculated at the routing block of the algorithm and transferred to the delay element of the created bit structure for the sum.

Maxcoutdelay element is to keep the critical path delay of the carry out output. This value is calculated at the routing block of the algorithm and transferred to the delay element of the created bit structure for the carry out.

Celltype element is to separate this structure from half adder.

Information related to half adder is kept with a structure named HA which has an in element, sum element, cout element, maxsumdelay element, maxcoutdelay element and a celltype element as shown below. It is elements and their purposes are same with the full adder.

```
struct HA =>
{
        in => '@',#2 elements in0,in1
        sum => '$',
        cout => '$',
        maxsumdelay => '$',
        maxcoutdelay => '$',
        celltype => '$',#Cell type 0
};
```

There are three main arrays that keeps delay, bit and cell information. The arrays shown below are used to keep path delays of full adder and half adder.

```
@FASumDelay = (1.910, 1.910, 0.710); # (in0 to sum, in1 to sum, in2 to sum)
@FACoutDelay = (1.065, 1.065, 1.065); # (in0 to carry out, in1 to carry out, in2 to
carry out)
@HASumDelay = (1.01, 1.01); # (in0 to sum, in1 to sum)
@HACoutDelay = (0.625, 0.625); # (in0 to carry out, in1 to carry out)
```

These values are taken from the LSI 10K library at hand and needs to be updated according to the synthesis library used. According to this array the delay array which is mentioned at the routing block algorithm is formed.

Bit array shown below keeps the bit structures related to the PCST part of the algorithm.

```
@bitarray [<level>] [<row>] [<column>];
```

First index is used for the level information of the bit, second index is used for the row information of the bit and the third index is used for the column information of the bit.
There is also a cell array which has the same indexing structure as bit array as shown below.

@cellarray [<level>] [<row>] [<column>];

The possible error messages that the generator can print are given below.

Enter a valid operand size!!! : When MxNb or MbxNb input format is used if M or N is less than one code exits with this error message.

Saturation index is too big!!! : If the entered saturation output width is bigger or equal to the possible multiplication or addition output length code exits with this error message.

Enter a valid or tree length!!! : When –t switch is used the entered value cannot be less than two if so code exits with this error message.

Enter a valid switch!!! : If the user entered an undefined switch code exits with this error message.

Wrong format!!! : If there is any other violation related to the input format other than the ones listed above code exits with this error message.

There are no bits to put in OR tree, possible cause saturation index is too big!!! : Even if the saturation output width is less than or equal to the multiplication or addition output length there is still a possibility of no bits entering the saturation logic and if this happens code exits with this error message.

6. BENCHMARK RESULTS

In this chapter, the experimental setup and experimental results for PCST, Dadda Tree and Wallace Tree are explained. There are two types of results presented for these algorithms in this chapter. First one is the cell count and final adder width relationship information gathered from generators and the second one is the timing and area information gathered from an EDA tool.

6.1 EXPERIMENTAL SETUP

For determining the number of full adders and half adders used and the final adder bit width Verilog HDL generation scripts are used.

To synthesize the saturated multiplication and saturated addition operations Synopsys DC version 2000.05-1 is used. The synthesis library used is the LSI 10K library at hand. Delay values used for time routing algorithm are obtained from this library by taking the average of intrinsic rise and intrinsic fall times of the cells. For half adder HA1 cell and for full adder FA1 cell from the synthesis library are used to calculate path delays.

For multiplication operations initial delay of the bits that enter the CST algorithms are taken as the delay of AN2 cell from the library which corresponds to an AND gate with two inputs. For addition operations initial delay of the bits that enter the CST algorithms are taken as zero.

6.2 CELL COUNTS AND FINAL ADDER WIDTH COMPARISON

Full adder and half adder counts and the final adder width comparisons for three methods including PCST are given in Table 6-1. According to the results PCST achieves the narrowest final adder while it uses fewer cells than the Wallace algorithm. On the other hand Dadda uses the least amount of cells but has the worst final adder width which will result in a bigger CLA adder and sometimes a slower one.

Multiplier	# Full Adder	# Half Adder	Final Adder Width
8 by 8 Dadda	35	7	14
8 by 8 Wallace	38	15	11
8 by 8 PCST	39	7	10
16 by 16 Dadda	195	15	30
16 by 16 Wallace	200	54	25
16 by 16 PCST	201	15	24
32 by 32 Dadda	899	31	62
32 by 32 Wallace	906	164	55
32 by 32 PCST	907	31	54
64 by 64 Dadda	3843	63	126
64 by 64 Wallace	3850	459	117
64 by 64 PCST	3853	63	116

Table 6-1: Cell Count and Final Adder Width for Multiplication

6.3 SATURATED UNSIGNED MULTIPLICATION

Timing results of unsigned saturated multiplication operations are given in Table 6-2 and Table 6-3. In these tables first column gives the bit width of multiplier and the multiplicand. Second column gives the bit width of the saturation. Third column gives the information related to the used algorithm which achieved the best timing for the given case. Fourth column gives the best timing. Fifth column gives the difference of best and second timing proportional to the second timing as a percentage.

Sixth column gives the difference of best and worst timing proportional to the worst timing as a percentage. The last column gives the difference of best and the high level RTL design timing proportional to the high level RTL design timing as a percentage.

Meanings of the indexes given at the third column are listed below:

1. PCST with time routing
2. PCST with random routing
3. Dadda with time routing
4. Wallace with time routing
5. Dadda with random routing
6. Wallace with random routing

According to the results PCST gives the best timing for the 9.43 percent of the cases. Additionally to that time routing which is an important contribution of this thesis gives better result than random routing for 88.68 percent of the cases.

The main reason for time routing not to give better timing results in all cases is the complex model of exact timing which depends to several concepts like wire delay, wire load, fan-in, and fan-out. What we used for our model is only the inner delay of the cells given in the synthesis library which won't be enough to calculate exact timing.

Input Type	Saturation Output Width	Method	Timing (ns)	Difference with Second (%)	Difference with Worst (%)	Difference with Synopsys (%)
8bx8b	8b	2	14.33	0.14	0.35	19.99
8bx8b	9b	2	14.35	0.07	0.35	19.97
8bx8b	10b	2	14.38	0	0.14	18.71
8bx8b	11b	5	14.39	0.07	3.68	19.25
8bx8b	12b	1	14.69	0.41	1.8	16.49
8bx8b	13b	5	14.46	2.3	9.17	18.49
8bx8b	14b	5	15.12	1.05	3.2	14.86
16bx16b	16b	3	16.54	5.05	13.4	31.26
16bx16b	17b	3	17.67	1.45	11.16	25.79
16bx16b	18b	3	17.83	4.35	8.52	23.8
16bx16b	19b	3	18.25	3.95	6.84	24.24
16bx16b	20b	3	18.26	5.68	9.38	24.23
16bx16b	21b	3	18.43	5.44	8.99	22.3
16bx16b	22b	3	19.19	1.64	4.15	19.17
16bx16b	23b	3	19.15	2.69	5.85	18.51
16bx16b	24b	3	19.16	3.77	7.35	18.92
16bx16b	25b	3	19.35	3.25	8.94	16.23
16bx16b	26b	3	19.97	2.16	4.4	12.72
16bx16b	27b	3	20.2	0.98	3.86	15.9
16bx16b	28b	3	20.02	3.05	6.1	12.54
16bx16b	29b	3	20.33	2.63	4.37	12.48
16bx16b	30b	3	20.32	2.4	5	9.53
32bx32b	32b	5	23	0.61	6.28	25.35
32bx32b	33b	3	23.56	1.09	4.27	22.98
32bx32b	34b	3	23.82	1.16	6.22	22.76
32bx32b	35b	3	24.18	0.08	5.14	20.9
32bx32b	36b	3	24.33	0.08	4.4	20.05

Table 6-2: Timing Results for Saturated Unsigned Multiplication

Input Type	Saturation Output Width	Method	Timing (ns)	Difference with Second (%)	Difference with Worst (%)	Difference with Synopsys (%)
32bx32b	37b	3	23.84	1.49	3.95	22.42
32bx32b	38b	3	24.45	0.2	5.93	18.99
32bx32b	39b	4	24.41	0.57	4.91	18.96
32bx32b	40b	1	24.12	0.94	4.7	19.97
32bx32b	41b	3	24.54	2.73	5.43	18.69
32bx32b	42b	3	24.48	2.51	6.74	18.1
32bx32b	43b	3	24.7	2.64	6.93	17.03
32bx32b	44b	3	24.58	2.58	8.01	18.74
32bx32b	45b	3	24.75	1.75	6.53	16.78
32bx32b	46b	3	24.88	1.39	5.9	16.03
32bx32b	47b	3	24.08	4.71	10.05	20.32
32bx32b	48b	3	24.08	4.82	9.91	19.6
32bx32b	49b	3	24.11	4.67	11.23	19.79
32bx32b	50b	3	24.58	4.8	9.3	16.82
32bx32b	51b	3	24.42	5.64	8.64	17.39
32bx32b	52b	3	24.78	2.02	8.56	16.45
32bx32b	53b	3	24.81	2.48	7.6	15.98
32bx32b	54b	3	24.74	1.71	6.47	14.48
32bx32b	55b	3	24.94	1.5	5.28	13.31
32bx32b	56b	3	25	2.19	5.98	13.85
32bx32b	57b	3	25.1	1.45	4.67	11.62
32bx32b	58b	3	25.13	2.07	5.03	12.8
32bx32b	59b	3	25.14	1.02	4.59	12.01
32bx32b	60b	3	24.93	1.7	5.92	11.41
32bx32b	61b	3	25.18	1.06	5.62	12.2
32bx32b	62b	3	25.05	1.73	6.74	9.76

Table 6-3: Timing Results for Saturated Unsigned Multiplication Continued

6.4 SATURATED UNSIGNED SUMMATION

Timing results of saturated unsigned summation operations are given in Table 6-4 and Table 6-5. Each column has the same properties defined in Chapter 6.3.

According to the results PCST gives the best timing for the 53.85 percent of the cases. Additionally to that time routing which is an important contribution of this thesis gives better result than random routing for 82.05 percent of the cases.

Input Type	Saturation Output Width	Method	Timing (ns)	Difference with Second (%)	Difference with Worst (%)	Difference with Synopsys (%)
3x8b	8	1	11.95	0.00	5.91	16.90
4x8b	8	2	14.16	0.00	1.05	1.67
5x8b	8	1	13.93	0.00	2.99	3.67
5x8b	9	3	14.02	1.41	2.57	2.64
6x8b	8	2	14.34	0.00	0.28	0.42
6x8b	9	3	14.18	0.56	1.39	1.94
7x8b	8	3	14.38	0.07	0.14	0.21
7x8b	9	3	14.32	0.21	0.49	2.19
3x16b	16	1	14.15	0.00	1.19	1.74
4x16b	16	2	14.08	0.00	2.15	2.22
5x16b	16	2	14.37	0.00	0.14	15.96
5x16b	17	1	14.31	0.00	0.56	15.67
6x16b	16	2	14.39	0.00	0.07	16.00
6x16b	17	2	14.39	0.00	0.07	16.96
7x16b	16	1	14.4	0.00	3.55	18.74
7x16b	17	1	14.39	0.00	5.20	17.20
8x16b	16	5	14.82	1.59	2.44	28.44
8x16b	17	1	14.99	0.00	4.03	27.44
9x16b	16	3	14.81	2.89	3.27	45.87

Table 6-4: Timing Results for Saturated Unsigned Summation

Input Type	Saturation Output Width	Method	Timing (ns)	Difference with Second (%)	Difference with Worst (%)	Difference with Synopsys (%)
9x16b	17	1	15.02	0.00	5.83	44.12
9x16b	18	1	15.26	0.00	5.51	41.33
10x16b	16	3	15.67	4.51	6.78	34.05
10x16b	17	3	16.08	0.74	7.05	33.53
10x16b	18	4	16.18	1.04	6.31	30.97
11x16b	16	3	16.44	1.02	4.31	40.09
11x16b	17	1	16.74	0.95	5.32	37.47
11x16b	18	1	16.73	1.30	5.91	37.67
12x16b	16	3	16.87	0.59	2.65	29.33
12x16b	17	1	17.01	0.64	4.17	29.77
12x16b	18	4	17.08	0.23	5.11	29.54
13x16b	16	3	17.08	0.58	2.29	36.72
13x16b	17	1	17.27	0.29	4.11	35.32
13x16b	18	1	17.24	0.29	4.33	35.24
14x16b	16	3	17.67	1.40	5.10	35.13
14x16b	17	1	17.95	0.00	6.12	34.22
14x16b	18	4	17.96	1.54	6.21	32.00
15x16b	16	3	17.69	1.67	5.95	42.06
15x16b	17	4	18.02	0.55	6.83	39.31
15x16b	18	4	17.99	0.99	7.03	42.69

Table 6-5: Timing Results for Saturated Unsigned Summation Continued

7. CONCLUSION AND FUTURE WORK

In this thesis a novel carry save tree algorithm is introduced. For this algorithm a fully functional Verilog HDL generator is developed that applies PCST and saturation logic for summation and multiplication of unsigned numbers. This generator can be classified as a Circuit Generation Framework which can easily utilize other CST algorithms. Additionally to that this generator is capable of applying a time based routing for any CST algorithm.

PCST algorithm is applied with (3,2) and (2,2) counters only. The main reason for that is the non complex structure of these counters which makes them a good choice for timing. But with the technological advancement different counters with higher input are becoming available and their timing and delay properties are improving. Due to that reason applying CST algorithms with different counters have been a research topic and our algorithm can also be applied with different counters.

One of the main usage areas of our algorithm is multiplication. Digital multiplication is considered in three steps partial product generation, reducing partial products matrix to a height of two (column compression) and final addition. The second step is where PCST fits but by considering the other two parts it can be turned into a complete multiplication solution. At this point time routing strategy needs to be extended for the other two steps.

Another important part is the time routing process. This part mainly depends on the delay values of the synthesis library used but critical path calculation of digital design is a more complicated process that takes into account wire delay, wire load, fan-out etc.

But these kinds of delay calculations depends to the EDA tool used for synthesis for that reason making an exact delay calculation is not possible. At this point if our algorithm is plugged in a synthesis tool it is clear that it will produce better timing results.

Usage of custom generators for combining different algorithmic operations can give better results than synthesis tools that use IP blocks like Synopsys DC. Kim, Jao, Jiang (1998) in their paper establishes relationship between the properties of arithmetic computations and several optimizing transformations using CSA's for better results. Because correlating logic in consecutive IP blocks can be optimized. In this thesis usage of saturation logic and PCST is a good example to that.

REFERENCES

Brent, R.P. and Kung, H.T., 1982. A Regular Layout for Parallel Adders. *IEEE Trans. On Computers*, C-31, pp.260-264

Breveglieri, L., Dadda, L., & Piuri, V., 1995. Column Compression Pipelined Multipliers. *Proceedings 1995 International Conference on Application Specific Array Processors*, pp. 93-103, 1995.

Capello, P.R. and Steiglitz, K., 1983. A VLSI Layout for a Pipelined Dadda Multiplier. *ACM Transactions on Computer Systems*, vol. 1, pp. 157-174, 1983

Dadda, L., 1965. Some Schemes for ParallelMultipliers, *Alta Frequenza*, vol. 34, pp. 349-356.

Dadda, L., 1976. On Parallel Digital Multipliers. *Alta Frequenza*, vol. 45, pp. 574-580,

Gok, M., 2000. Integer Multiplication with Overflow Detection or Saturation. Master's thesis, Lehigh University, 19 Memorial Dr. West, Bethlehem, PA, 18015, 2000.

Gok, M., Schulte, M.J. & Balzola, P.I., 2001. Efficient Integer Multiplication Overflow Detection Circuits. *Signals, Systems and Computers, 2001.* Conference Record of the Thirty-Fifth Asilomar Conference, vol. 2, pp.1661-1665

Itoh, N., Naemura, Y., Makino, H., Nakase, Y., Yoshihara, T. & Horiba, Y., 2001. A 600-MHz 54 x 54-bit Multiplier with Rectangular-Styled Wallace Tree. *JSSC*, vol.36, no. 2, February 2001.

Kim, T., Jao, W. & Jiang, T., 1998. Circuit Optimization Using Carry–Save–Adder Cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 974-984, October 1998

Mehta, M., Parmar, V. & Swartzlander, Jr. E.E., 1991. High-Speed Multiplier Design Using Multi-Input Counter and Compressor Circuits. *Proceedings of the 10th International Symposium on Computer Arithmetic*, pp. 43-50, 1991.

Oklobdzija, V.J. and Villeger, D., 1993. Multiplier Design Utilizing Improved Column Compression Tree and Optimized Final Adder in CMOS Technology. *Proc. 10th Anniv. 1993 Int. Symp. VLSI Tech*, May 1993.

Oklobdzija, V.J. and Villeger, D., 1995. Improving Multiplier Design by Using Improved Column Compression Tree and Optimized Final Adder in CMOS Technology. *IEEE Trans. VLSI*, vol. 3, no. 2, pp. 292-301, June 1995.

Paterson, M.S., Pippenger, N. & Zwick U., 1992. Optimal Carry Save Networks. *Proceedings of the London Mathematical Society symposium on Boolean function complexity*, London, United Kingdom

Schulte, M.J., Balzola, P.I., Akkas, A. & Brocato, R.W., 2000. Integer Multiplication with Overflow Detection or Saturation. *IEEE Transactions on Computers*, vol. 49, no. 7, pp.681-691.

Townsend, W., Swartzlander, E. & Abraham , J., 2003. A Comparison of Dadda and Wallace Multiplier Delays. *SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*.

Wallace, C.S. , 1964. Suggestion for a Fast Multiplier, IEEE Trans. *Electronic Computers*, vol. 13, pp. 14-17,

Wang, Z., Jullien, G.A. & Miller W.C., 1995. A New Design Technique for Column Compression Multipliers. *IEEE Transactions on Computers*, vol. 44, pp. 962-970, 1995.

Zimmermann, R., 2009. Datapath Synthesis for Standard-Cell Design. *19th IEEE Symposium on Computer Arithmetic*, Portland, Oregon, USA, June 8-10, 2009.

APPENDICES

CURRICULUM VITAE

Okan Keskin

Tel: 0 (537) 303 38 55
Adress: Aladoğan Sokak, No 10, Daire 6, Ortaköy/İSTANBUL

Birth Date: 23.09.1983
Birth Place: İstanbul
Nationality: T.C.
Marital Status: Single

Educational :

MS, September 2010 (expected), EEE, Bahçeşehir University, İstanbul
Full support by both Bahçeşehir University (TA) & TÜBİTAK • GPA: 3.54
Thesis: Carry Save Tree Generation

BS, June 2007, EEE, Bahçeşehir University, İstanbul
Full ÖSYM scholarship for all 4 years • GPA: 3.70 • Ranked 1st in EEE among 23 students • 19 credits of coursework in Computer Science

Military Service: No.

Foreign Languages:

English
(Advanced)
Spanish
(Low Level)

Job Experience :

Digital Design and Verification Engineer, ST Ericsson Istanbul Design Center, November 2009 – Ongoing

Teaching Assistant (TA), Bahçeşehir University EEE Dept., İstanbul, November 2007 – November 2009
So far assisted Digital IC Design Lab (head asst.), Digital System Design Lab (head asst.), Digital Design Lab (head asst.), Microprocessors Lab, and Electronics Lab. TAship duties include teaching lab lectures, problem sessions, office hours, maintaining/modifying lab instruments, designing conducting experiments and more.

Undergrad Lab Assistant, Bahçeşehir University, İstanbul, February 2005 – June 2006
Helped graduate assistants help students in C/C++ programming labs and electronics labs.

Relevant Knowledge:

FPGA/ASIC design knowledge with Verilog, VHDL, Xilinx ISE, Synopsys DC, Modelsim, Simvision • Verification knowledge with OVM and SystemVerilog • C/C++, Perl, Tcl, Matlab • Linux • PIC firmware experience

Hobbies: Cinema, Music, Travelling.