International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

Design and Characterization of Parallel Prefix Adders

Dollu Madhu Mohan[1], S.Mahaboob Basha[2]
[1]P.G. Scholar, [2]Guide, Head of the Department
[1,2] Branch:ECE (VLSI)
[1,2] GEETHANJALI COLLEGE OF ENGG. & TECH.
Email Id: [1]madhumohan329@gmail.com, [2]syedmahaboob45@gmail.com

Abstract

The binary adder is the critical element in most digital circuit designs including digital signal processors (DSP) and microprocessor data path units. As such, extensive research continues to be focused on improving the power delay performance of the adder.

Parallel-prefix adders (also known as carry-tree adders) are known to have the best performance in VLSI designs. This paper investigates three types of carry-tree adders (the Kogge-Stone, sparse Kogge-Stone, and spanning tree adder) and compares them to the simple Ripple Carry Adder (RCA) and Carry Skip Adder (CSA). These designs of varied bit-widths were implemented on a Xilinx Spartan 3E FPGA and delay measurements were made with a high-performance logic analyzer. Due to the presence of a fast carry-chain, the RCA designs exhibit better delay performance up to 128 bits. The carry-tree adders are expected to have a speed advantage over the RCA as bit widths approach 256.

Keywords : - Adders, Delay, Routing, Table lookup, Software, Simulation

INTRODUCTION

Motivation

However, in digital systems, such as a microprocessor, DSP (Digital Signal Processor) or ASIC (Application-Specific Integrated Circuit), binary numbers are more pragmatic for a given computation.

This occurs because binary values are optimally efficient at representing many values. Binary adders are one of the most essential logic elements within a digital system. In addition, binary adders are also helpful in units other than Arithmetic Logic Units (ALU), such as multipliers, dividers and memory addressing. Therefore, binary addition is essential that any improvement in binary addition can result in a performance boost for any computing system and, hence, help improve the performance of the entire system. The major problem for binary addition is the carry chain. As the width of the input operand increases, the length of the carry chain increases. Figure 1.1 demonstrates an example of an 8- bit binary add operation and how the carry chain is affected.

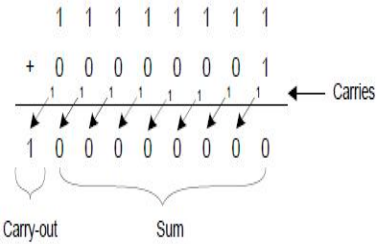

Figure 1.1: Binary Adder

However, because of the structure of the configurable logic and routing resources in FPGAs, parallel-prefix adders will have a different performance than VLSI implementations. In this work , the practical issues involved in designing and implementing tree-based adders on FPGAs are described. Several tree-based adder structures are implemented and characterized on a FPGA and compared

with the Ripple Carry Adder (RCA) and the Carry Skip Adder (CSA). Finally, some conclusions and suggestions for improving FPGA designs to enable better tree-based adder performance are given.

Carry-Propagate Adders

Binary carry-propagate adders have been extensively published, heavily attacking problems related to carry chain problem. Binary adders evolve from linear adders, which have a delay approximately proportional to the width of the adder, e.g. ripple-carry adder (RCA), to logarithmic-delay adder, such as the carry-lookahead adder (CLA). There are some additional performance enhancing schemes, including the carry-increment adder and the Ling adder that can further enhance the carry chain, however, in Very Large Scale Integration (VLSI) digital systems, the most efficient way of offering binary addition involves utilizing parallel-prefix trees, this occurs because they have the regular structures that exhibit logarithmic delay.

This happens within VLSI architectures because a carry-lookahead adder, such as the one implemented in one of Motorola's processors , tends to implement the carry chain in the vertical direction instead of a horizontal one, which has a tendency to increase both wire density and fan-in/out dependence.

BINARY ADDER SCHEMES

Adders are one of the most essential components in digital building blocks, however, the performance of adders become more critical as the technology advances. The problem of addition involves algorithms in Boolean algebra and their respective circuit implementation. Algorithmically, there are linear-delay adders like ripple-carry adders (RCA), which are the most straightforward but slowest. Adders like carry-skip adders (CSKA), carry-select adders (CSEA) and carry-increment adders (CINA) are linear-based adders with optimized carry-chain and improve upon the linear chain within a ripple-carry adder. Carry-lookahead adders (CLA) have logarithmic delay and currently have evolved to parallel-prefix structures. Other schemes, like Ling adders, NAND/NOR adders and carry-save adders can help improve performance as well.

Binary Adder Notations and Operations

As mentioned previously, adders in VLSI digital systems use binary notation. In that case, add is done bit by bit using Boolean equations. Consider a simple binary add with two n-bit inputs A;B and a one-bit carry-in cin along with n-bit output S.

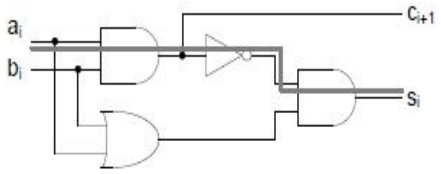

Figure 3.1: 1-bit Half Adder.

$S = A + B + Cin$

where A = an-1, an-2……a0; B = bn-1, bn-2……b0.

The + in the above equation is the regular add operation. However, in the binary world, only Boolean algebra works. For add related operations, AND, OR and Exclusive-OR (XOR) are required. In the following documentation, a dot between two variables (each with single bit), e.g. a _ b denotes 'a AND b'. Similarly, a + b denotes 'a OR b' and a _ b denotes 'a XOR b'.

Considering the situation of adding two bits, the sum s and carry c can be expressed using Boolean operations mentioned above.

$$si = ai \hat{} bi$$

$$ci+1 = ai.bi$$

The Equation of ci+1 can be implemented as shown in Figure 2.1. In the figure, there is a half adder, which takes only 2 input bits. The solid line highlights the critical path, which indicates the longest path from the input to the output.

Equation of ci+1 can be extended to perform full add operation, where there is a carry input.

$si = ai \wedge bi \wedge ci$

$ci+1 = ai \cdot bi + ai \cdot ci + bi \cdot ci$

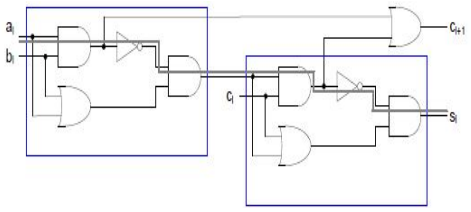

Figure 3.1.2: 1-bit Full Adder.

A full adder can be built based on Equation above. The block diagram of a 1-bit full adder is shown in Figure 2.2. The full adder is composed of 2 half adders and an OR gate for computing carry-out.

Using Boolean algebra, the equivalence can be easily proven.

To help the computation of the carry for each bit, two binary literals are introduced. They are called carry generate and carry propagate, denoted by gi and pi. Another literal called temporary sum ti is employed as well. There is relation between the inputs and these literals.

$$gi = ai \cdot bi$$
$$pi = ai + bi$$
$$ti = ai \wedge bi$$

where i is an integer and 0 _ i < n.

With the help of the literals above, output carry and sum at each bit can be written as

$$ci+1 = gi + pi \cdot ci$$
$$si = ti \wedge ci$$

In some literatures, carry-propagate pi can be replaced with temporary sum ti in order to save the number of logic gates. Here these two terms are separated in order to clarify the concepts. For example, for Ling adders, only pi is used as carry-propagate.

The single bit carry generate/propagate can be extended to group version G and P. The following equations show the inherent relations.

$$Gi:k = Gi:j + Pi:j \cdot Gj-1:k$$
$$Pi:k = Pi:j \cdot Pj-1:k$$

where i : k denotes the group term from i through k. Using group carry generate/propagate, carry can be expressed as expressed in the following equation.

$ci+1 = Gi:j + Pi:j \cdot cj$

Ripple-Carry Adders (RCA)

The simplest way of doing binary addition is to connect the carry-out from the previous bit to the next bit's carry-in. Each bit takes carry-in as one of the inputs and outputs sum and carry-out bit and hence the name ripple-carry adder. This type of adders is built by cascading 1-bit full adders. A 4-bit ripple-carry adder is shown in Figure 3.2. Each trapezoidal symbol represents a single-bit full adder. At the top of the figure, the carry is rippled through the adder from cin to cout.

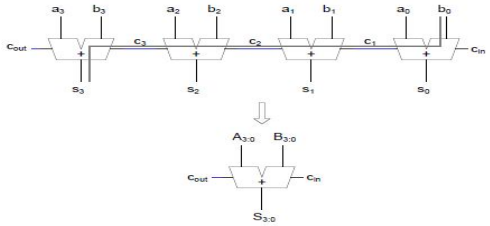

Figure 3.2: Ripple-Carry Adder.

It can be observed in Figure 3.2 that the critical path, highlighted with a solid line, is from the least significant bit (LSB) of the input (a0 or b0) to the most significant bit (MSB) of sum (sn-1). Assuming each simple gate, including AND, OR and XOR gate has a delay of 2/\ and NOT gate has a delay of 1/\. All the gates have an area of 1 unit. Using this analysis and assuming that each add block is built with a 9-gate full adder, the critical path is calculated as follows.

$ai, bi \rightarrow si = 10^{/\backslash}$

$ai, bi \rightarrow ci+1 = 9^{/\backslash}$

$ci \rightarrow si = 5^{/\backslash}$

$ci \rightarrow ci+1 = 4^{/\backslash}$

The critical path, or the worst delay is

$$trca = \{9 + (n-2) \times 4 + 5\}^{/\backslash} = \{f4n + 6\}^{/\backslash}$$

As each bit takes 9 gates, the area is simply 9n for a n-bit RCA.

Carry-Select Adders (CSEA)

Simple adders, like ripple-carry adders, are slow since the carry has to to

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

travel through every full adder block. The method is based on the conditional sum adder and extended to a carry-select adder. With two RCA, each computing the case of the one polarity of the carry-in, the sum can be obtained with a 2x1 multiplexer with the carry-in as the select signal. An example of 16-bit carry-select adder is shown in Figure 3.3.1 . In the figure, the adder is grouped into four 4-bit blocks. The 1-bit multiplexors for sum selection can be implemented as Figure 3.3.2 shows. Assuming the two carry terms are utilized such that the carry input is given as a constant 1 or 0:

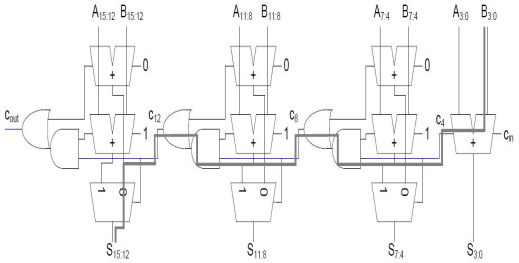

Figure 3.3 : Carry-Select Adder.

In Figure 3.3, each two adjacent 4-bit blocks utilizes a carry relationship

$ci+4 = c0\ i+4 + c1\ i+4 \ .\ ci$

The relationship can be verified with properties of the group carry generate/propagate and c0 i+4 can be written as

$c0\ i+4 = Gi+4:i + Pi+4:i \ .\ 0 \ = Gi+4:i$

Similarly, c1 i+4 can be written as

$c1\ i+4 = Gi+4:i + Pi+4:i \ .1 = Gi+4:i + Pi+4:i$

Then

$c0\ i+4 + c1\ i+4 \ .\ ci = Gi+4:i + (Gi+4:i + Pi+4:i) \ .\ ci$

$= Gi+4:i + Gi+4:i \ .\ ci + Pi+4:i \ .\ ci$

$= Gi+4:i + Pi+4:i \ .\ ci = ci+4$

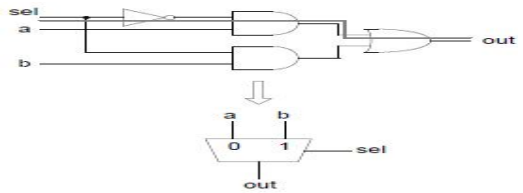

Figure 3.3.1 : 2-1 Multiplexor.

Varying the number of bits in each group can work as well for carry-select adders. temporary sums can be defined as follows.

$s0\ i+1 = ti+1 \ .\ c0\ i$

$s1\ i+1 = ti+1 \ .\ c1\ i$

The final sum is selected by carry-in between the temporary sums already calculated.

$si+1 = cj \ .\ s0\ i+1 + cj \ .\ s1\ i+1$

Assuming the block size is fixed at r-bit, the n-bit adder is composed of k groups of r-bit blocks, i.e. n = r x k. The critical path with the first RCA has a delay of (4r + 5)/\ from the input to the carry-out, and there are k - 2 blocks that follow, each with a delay of 4/\ for carry to go through. The final delay comes from the multiplexor, which has a delay of 5/\, as indicated in Figure 2.5. The total delay for this CSEA is calculated as

$tcsea = 4r + 5 + 4(k - 2) + 5/\backslash$

$= \{4r + 4k + 2\}/\backslash$

The area can be estimated with (2n - r) FAs, (n - r) multiplexors and (k - 1) AND/OR logic. As mentioned above, each FA has an area of 9 and a multiplexor takes 5 units of area. The total area can be estimated

$9(2n - r) + 2(k - 1) + 4(n - r) = 22n - 13r + 2k - 2$

The delay of the critical path in CSEA is reduced at the cost of increased area. For example, in Figure 2.4, k = 4, r = 4 and n = 16. The delay for the CSEA is 34/\ compared to 70/\ for 16-bit RCA. The area for the CSEA is 310 units while the RCA has an area of 144 units. The delay of the CSEA is about the half of the RCA. But the CSEA has an area more than twice that of the RCA.

Carry-Skip Adders (CSKA)

There is an alternative way of reducing the delay in the carry-chain of a RCA by checking if a carry will propagate through to the next block. This is called carry-skip adders.

$ci+1 = Pi:j \ \_ \ Gi:j + Pi:j \ .\ cj$

Figure 3.4 shows an example of 16-bit carry-skip adder.

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

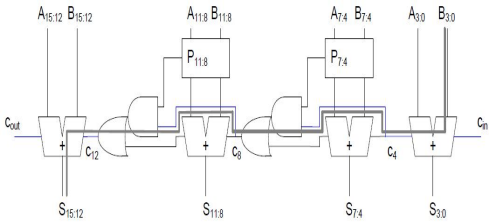

Figure 3.4 : Carry-Skip Adder.

The carry-out of each block is determined by selecting the carry-in and Gi:j using Pi:j. When Pi:j = 1, the carry-in cj is allowed to get through the block immediately. Otherwise, the carry-out is determined by Gi:j. The CSKA has less delay in the carry-chain with only a little additional extra logic. Further improvement can be achieved generally by making the central block sizes larger and the two-end block sizes smaller.

Assuming the n-bit adder is divided evenly to k r-bit blocks, part of the critical path is from the LSB input through the MSB output of the final RCA. The first delay is from the LSB input to carry-out, which is 4r + 5. Then, there are k - 2 skip logic blocks with a delay of 3/\. Each skip logic block includes one 4-input AND gate for getting Pi+3:i and one AND/OR logic. The final RCA has a delay from input to sum at MSB, which is 4r+6. The total delay is calculated as follows.

$tcska = \{4r + 5 + 3(k - 2) + 4r + 6\}/\backslash$
$= \{8r + 3k + 5\}/\backslash$

The CSKA has n-bit FA and k - 2 skip logic blocks. Each skip logic block has an area of 3 units. Therefore, the total area is estimated as 9n + 3(k - 2) = 9n + 3k – 6.

Carry-Look-ahead Adders (CLA)

The carry-chain can also be accelerated with carry generate/propagate logic. Carry-lookahead adders employ the carry generate/propagate in groups to generate carry for the next block. In other words, digital logic is used to calculate all the carries at once. The carry generate/propagate signals gi/pi feed to carry-lookahead generator (CLG) for carry inputs to RFA.

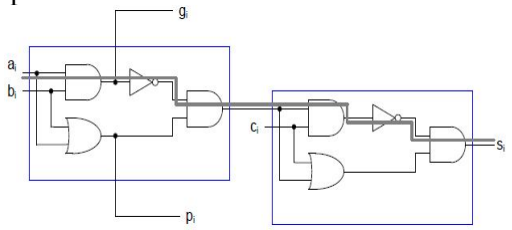

Figure 3.5.1 : Reduced Full Adder.

The theory of the CLA is based on next Equations. Figure 3.5.2 shows an example of 16-bit carry-lookahead adder. In the figure, each block is fixed at 4-bit. BCLG stands for Block Carry Lookahead Carry Generator, which generates generate/propagate signals in group form. For the 4-bit BCLG, the following equations are created.

$$Gi+3:i = gi+3 + pi+3 . gi+2 + pi+3 . pi+2 . gi+1 + pi+3 . pi+2 . pi+1 . gi$$
$$Pi+3:i = pi+3 . pi+2 . pi+1 . pi$$

The group generate takes a delay of 4/\, which is an OR after an AND, therefore, the carry-out can be computed, as follows.

$$ci+3 = Gi+3:i + Pi+3:i . ci$$

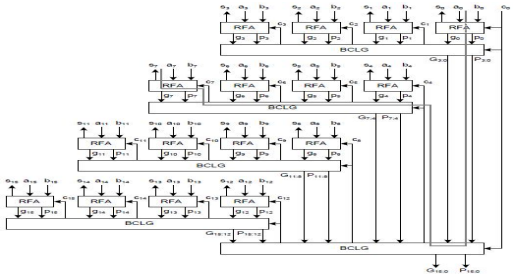

Figure 3.5.2 : Carry-Lookahead Adder.

The carry computation also has a delay of 4/\, which is an OR after an AND. The 4-bit BCLG has an area of 14 units.

The critical path of the 16-bit CLA can be observed from the input operand through 1 RFA, then 3 BCLG and through the final RFA. That is, the critical path shown in Figure 3.5.2 is from a0/b0 to s7. The delay will be the same for a0/b0 to s11 or s15, however, the critical path traverses logarithmically, based on the group size. The delays are listed below.

$$a0 , b0 \rightarrow p0 , g0 = 2/\backslash$$
$$p0 , g0 \rightarrow G3,0 = 4/\backslash$$

G3,0 → c4 = 4/
c4 → c7 = 4/
c7 → s7 = 5/
a0 , b0 → s7 = 19/

The 16-bit CLA is composed of 16 RFAs and 5 BCLGs, which amounts to an area of 16 x 8 + 5 x 14 = 198 units .

Assume the CLA has n-bits, which is divided into k groups of r-bit blocks. It requires dlogrne logic levels. The critical path starts from the input to p0/g0 generation, BLCG logic and the carry-in to sum at MSB. The generation of (p; g) takes a delay of 2/\. The group version of (p; g) generated by the BCLG has a delay of 4/\. From next BCLG, there is a 4/\ delay from the CLG generation and 4/\ from the BCLG generation to the next level, which totals to 8/\. Finally, from ck+r to sk+r, there is a delay of 5/\. Thus, the total delay is calculated as follows.

tcla = {2 + 8(dlogrn - 1) + 4 + 5}/
= {3 + 8dlogrn}/

PARALLEL-PREFIX STRUCTURES

Introduction

To resolve the delay of carry-lookahead adders, the scheme of multilevel-lookahead adders or parallel-prefix adders can be employed. These adders have tree structures within a carry-computing stage similar to the carry propagate adder. However, the other two stages for these adders are called pre-computation and post-computation stages.

In the prefix stage, the group carry generate/propagate signals are computed to form the carry chain and provide the carry-in for the adder below.

Gi:k = Gi:j + Pi:j . Gj-1:k
Pi:k = Pi:j . Pj-1:k

In the post-computation stage, the sum and carry-out are finally produced. The carry-out can be omitted if only a sum needs to be produced.

si = ti ^ Gi:-1
cout = gn-1 + pn-1 _ Gn-2:-1

where Gi:-1 = ci with the assumption g-1 = cin.

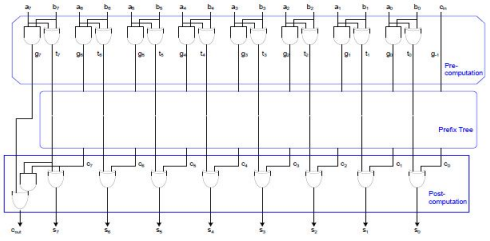

Figure 4.1.1 : 8-bit Parallel-Prefix Structure with carry save notation.

To illustrate a sample prefix structure, an 8-bit Sklansky prefix tree is shown in Figure 4.1.1. Although Sklansky created this prefix structure with relationship to adders, it is typically referred to as a member of the Ladner-Fischer prefix family. More details about prefix structures, including how to build the prefix structures and the performance comparison, will be described the next chapter of this dissertation.

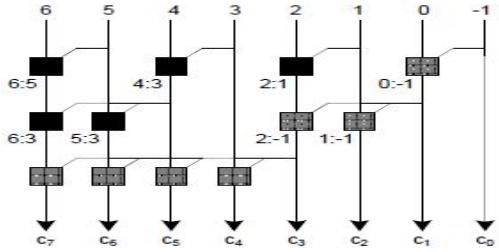

Figure 4.1.2: Sklansky Parallel-Prefix

4.2 Building Prefix Structures

Parallel-prefix structures are found to be common in high performance adders because of the delay is logarithmically proportional to the adder width. Such structures can usually be divided into three stages, pre-computation, prefix tree and post-computation.

An example of an 8-bit parallel-prefix structure is shown in Figure 4.2. In the prefix tree, group generate/propagate are the only signals used. The group generate/propagate equations are based on single bit generate/propagate, which are computed in the pre-computation stage.

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

gi = ai . bi
pi = ai ^ bi
where 0 < I < n. g_{-1} = cin and p_{-1} = 0. Sometimes, pi can be computed with OR logic instead of an XOR gate. The OR logic is mandatory especially when Ling's scheme is applied. Here, the XOR logic is utilized to save a gate for temporary sum ti.
In the prefix tree, group generate/propagate signals are computed at each bit.
Gi:k = Gi:j + Pi:j . Gj-1:k
Pi:k = Pi:j . Pj-1:k

More practically, the above equation can be expressed using a symbol " o " denoted by Brent and Kung . Its function is exactly the same as that of a black cell. That is
(Gi:k; Pi:k) = (Gi:j; Pi:j) o (Gj-1:k; Pj-1:k); or

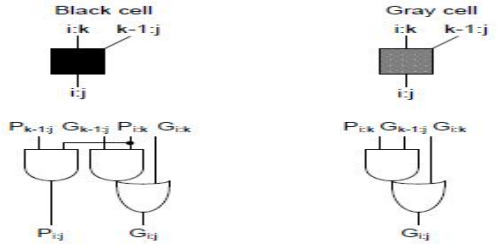

Figure 4.2 : Cell Definitions.

Gi:k = (gi; pi) o (gi-1; pi-1) o …….o (gk; pk)
Pi:k = pi . pi-1 . …… . pk
The "o" operation will help make the rules of building prefix structures. In the post-computation, the sum and carry-out are the final output.
si = pi . Gi-1:-1
cout = Gn:-1
where "-1" is the position of carry-input.

Preparing Prefix Tree

The synthesis rules apply to any type of prefix tree. In this section, the methodology utilized to build fixed prefix structures is discussed.

The l level refers to the logic row where group generate G and propagate P are computed. u=v are the maximum output bit span and input bit span of the logic cells. If the logic level is not the last of the prefix tree, the output of the current logic level will be the input to the next logic level. The maximum bit span sets the limit of the bit span at a certain logic level. The relations between these terms are described by the following equations

$$u = 2^{l\ level,}\ v = 2^{l\ level-1}$$

The value of v is 1/2 of the value of u. To further ease the illustration, the term (Gi:m; Pi:m) is briefed as GPi:m. For example,

GP6:3 = GP6:5 o GP4:3
which is equal to
G6:3 = G6:5 + P6:5 . G4:3
P6:3 = P6:5 . P4:3

For this case, l level = 2; u = 4; v = 2. The inputs are GP6:3 and GP4:3 that have a bit span of 2, as the subscripts of GP indicate. The output is GP6:3, which has a bit span of 4.
Figure 4.3.1 shows an 8-bit example of an empty matrix with only bit lines and dashed boxes filled in. The inputs gi/pi go from the top and the outputs ci are at the bottom. The LSB is labeled as -1 where the carry-input (cin) locates. The objective is to obtain all ci's in the form of Gi-1:-1's,
Where *c0 = G-1:-1; c1 = G0:-1; c2 = G1:-1; …….; cn-1 = Gn-2:-1*

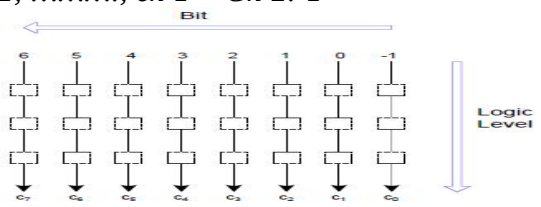

Figure 4.3.1 : 8-bit Empty Prefix Tree.
The way of building a prefix tree can be processed as the arrows indicate (i.e. from LSB to MSB horizontally and then from top logic level down to bottom logic level vertically).

Kogge-Stone Prefix Tree

Kogge-Stone prefix tree is among the type of prefix trees that use the fewest logic levels. A 16-bit example is shown in Figure 4.3.2. In fact, Kogge-Stone is a member of Knowles prefix tree. The 16-bit prefix tree can be viewed as Knowels

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

[1,1,1,1]. The numbers in the brackets represent the maximum branch fan-out at each logic level. The maximum fan-out is 2 in all logic levels for all width Kogge-Stone prefix trees.

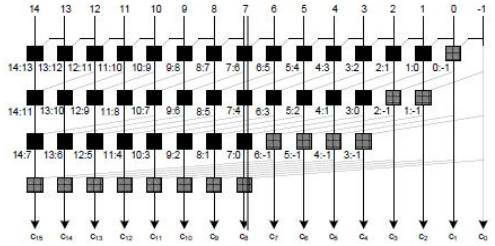

Figure 4.3.2 : 16-bit Kogge-Stone Prefix Tree.

For the Kogge-Stone prefix tree, at the logic level 1, the inputs span is 1 bit (e.g. group (4:3) take the inputs at bit 4 and bit 3). Group (4:3) will be taken as inputs and combined with group (6:5) to generate group (6:3) at logic level 2.

Logic Levels	u	v	Output $(i:i-u+1)$	Input1 $(i:i-v+1)$	Input2 $(i-v:i-u+1)$	Equation Mapping
1	2	1	7 : 6	7 : 7	6 : 6	$GP_{7:6} = GP_7 \circ GP_6$
2	4	2	11 : 8	11 : 10	9 : 8	$GP_{11:8} = GP_{11:10} \circ GP_{9:8}$
3	8	4	14 : 7	14 : 11	10 : 7	$GP_{14:7} = GP_{14:11} \circ GP_{10:7}$
4	16	8	7 : −1	7 : 0	−1 : −1	$GP_{7:-1} = GP_{7:0} \circ GP_{-1}$

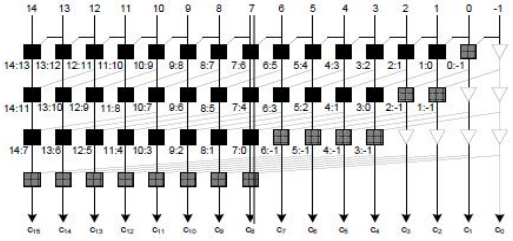

Figure 4.3.3 : 16-bit Kogge-Stone Prefix Tree with Buffers.

The number cells for a Kogge-Stone prefix tree can be counted as follows. Each logic level has n-m cells, where m = $2^{l\,\text{level}-1}$. That is, each logic level is missing m cells. That number is the sum of a geometric series starting from 1 to n/2 which totals to n-1. The total number of cells will be $n\log_2 n$ subtracting the total number of cells missing at each logic level , which winds up with $n\log_2 n - n + 1$. When n = 16, the area is estimated as 49.

Brent-Kung Adder

Brent-Kung adder is a very well-known logarithmic adder architecture that gives an optimal number of stages from input to all outputs but with asymmetric loading on all intermediate stages. The cost and wiring complexity is less in brent kung adders. But the gate level depth of Brent-Kung adders is 0 (log2(n)), so the speed is lower. The block diagram of 4-bit Brent-Kung adder is shown in Fig.

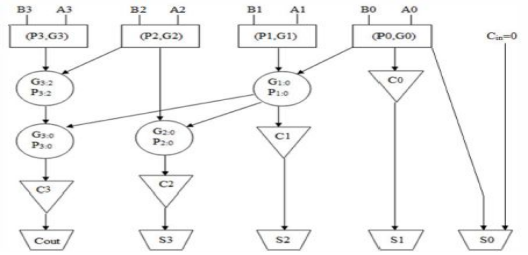

Fig. 4.4 : Block Diagram of 4-Bit Brent Kung Adder

CARRY-TREE ADDER DESIGNS

Kogge-Stone Adder

Parallel-prefix adders, also known as carry-tree adders, pre-compute the propagate and generate signals. These signals are variously combined using the *fundamental carry operator* (fco).

$$(gL, pL) \circ (gR, pR) = (gL + pL \bullet gR, pL \bullet pR)$$

Due to associative property of the *fco*, these operators can be combined in different ways to form various adder structures. For, example the four-bit carry-lookahead generator is given by:

$$c4 = (g4, p4) \circ [\,(g3, p3) \circ [(g2, p2) \circ (g1, p1)]\,]$$

A simple rearrangement of the order of operations allows parallel operation, resulting in a more efficient tree structure for this four bit example:

$$c4 = [(g4, p4) \circ (g3, p3)] \circ [(g2, p2\,) \circ (g1, p1)]$$

It is readily apparent that a key advantage of the tree structured adder is that the critical path due to the carry delay is on the order of *log2N* for an N-bit wide adder. The arrangement of the prefix network gives rise to various families of adders. For this study, the focus is on the Kogge-Stone adder, known for having minimal logic depth and fanout see Figure (5.1). Here we designate BC as the black cell which

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

generates the ordered pair in equation (1); the gray cell (GC) generates the left signal only.

Figure 5.1: 16 bit Kogge-Stone adder

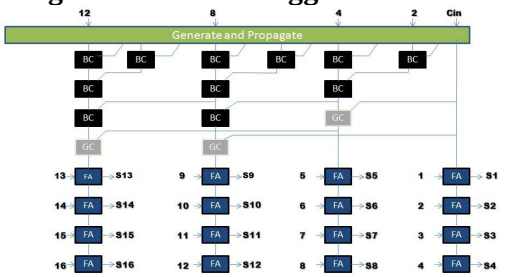

Figure 5.1.2: Sparse 16 bit Kogge-Stone adder

Another carry-tree adder known as the spanning tree carry-lookahead (CLA) adder is also examined [6]. Like the sparse Kogge-Stone adder, this design terminates with a 4- bit RCA. As the FPGA uses a fast carry-chain for the RCA, it is interesting to compare the performance of this adder with the sparse Kogge-Stone and regular Kogge-Stone adders. Also of interest for the spanning-tree CLA is its testability features [7].

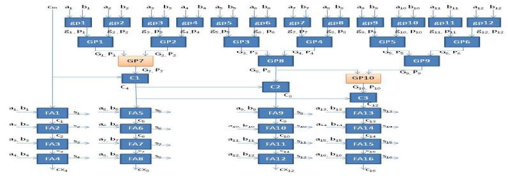

Figure 5.1.3: 16-bit Spanning Tree Carry Lookahead Adder

Modification

Han-Carlson adder constitutes a good trade-off between fanout, number of logic levels and number of black cells. Because of this, Han-Carlson adder can achieve equal speed performance respect to Kogge-Stone adder, at lower power consumption and area .

Therefore it is interesting to implement a speculative Han-Carlson adder. Moved by these reasons, we have generated a Han-Carlson speculative prefix-processing stage by deleting the last rows of the Kogge-Stone part of the adder. As an example, the Fig. 10 shows the Han-Carlson adder in which the two BrentKung rows at the beginning and at the end of the graph are unchanged, while the last Kogge-Stone row is pruned.

As it can be observed in Fig., the length of the propagate chains is only for , while for the propagate chain length is . In general, the computed propagate and generate signals for the speculative Han-Carlson architecture are:

$$\begin{aligned} &(g,p)_{[i:0]} && \text{for}: i \le K \\ &(g,p)_{[i:i-K+1]} && \text{for}: i > K,\ i \text{ odd} \\ &(g,p)_{[i:i-K]} && \text{for}: i > K,\ i \text{ even} \end{aligned} \qquad (13)$$

As it will be apparent in the following, having the propagate lengths equal to for half of the outputs greatly simplifies the error detection.

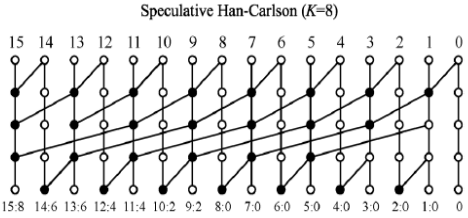

Figure 5.2 : Han-Carlson speculative prefix-processing stage.

XILINX TOOLS

Xilinx ISE Overview

The Integrated Software Environment (ISE) is the Xilinx design software suite that allows you to take your design from design entry through Xilinx device programming. The ISE Project Navigator manages and processes your design through the following steps in the ISE design flow.

Project Navigator Overview

Project Navigator organizes your design files and runs processes to move the design from design entry through implementation

to programming the targeted Xilinx device.

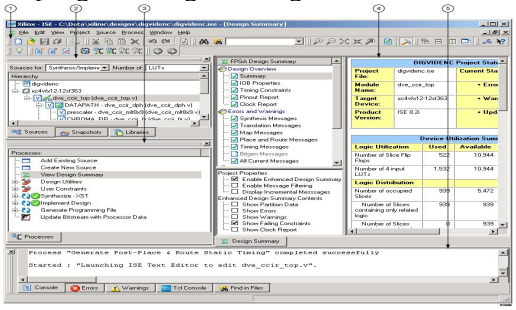

Fig 6.2 Project Navigator window

1. Toolbar
2. Sources window
3. Processes window
4. Workspace
5. Transcript window

The first step in implementing your design for a Xilinx FPGA or CPLD is to assemble the design source files into a project. For information on creating projects and source files, see Creating a Project and Creating a Source File.

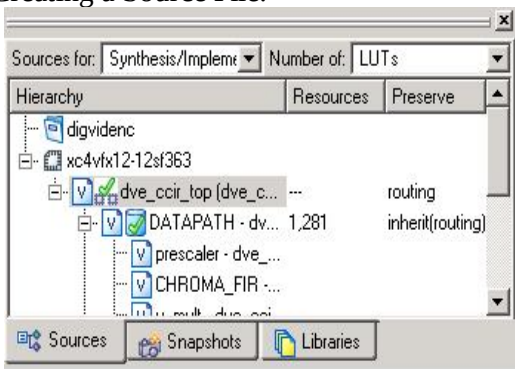

Fig 6.2.1 Design view drop down list

The Design View ("Sources for") drop-down list at the top of the Sources tab allows you to view only those source files associated with the selected Design View (for example, Synthesis/Implementation). The "Number of" drop-down list, Resources column, and Preserve column are available for designs that use Partitions.

You can change the project properties, such as the device family to target, the top-level module type, the synthesis tool, the simulator, and the generated simulation language Depending on the source file and tool you are working with, additional tabs are available in the Sources window:

- Always available: Sources tab, Snapshots tab, Libraries tab
- Constraints Editor: Timing Constraints tab
- Floorplan Editor: Translated Netlist tab, Implemented Objects tab
- IMPACT: Configuration Modes tab
- Schematic Editor: Symbols tab
- RTL and Technology Viewers: Design tab
- Timing Analyzer: Timing tab

The Processes tab in the Processes window allows you to run actions or "processes" on the source file you select in the Sources tab of the Sources window. The Process tab shows the available processes in a hierarchical view. You can collapse and expand the levels by clicking the plus (+) or minus (-) icons. Processes are arranged in the order of a typical design flow: project creation, design entry, constraints management, synthesis, implementation, and programming file creation. Depending on the source file and tool you are working with, additional tabs are available in the Processes window:

- Always available: Processes tab
- Floor plan Editor: Design Objects tab, Implemented - Selection tab
- IMPACT: Configuration Operations tab
- ISE Simulator: Hierarchy Browser tab
- Schematic Editor: Options tab
- Timing Analyzer: Timing Objects tab
The following types of processes are available as you work on your design:
- **Tasks**

  When you run a task process, the ISE software runs in "batch mode," that is, the software processes your source file but does not open any additional software tools in the Workspace.
- **Reports**

  Most tasks include report sub-processes, which generate a summary or

status report, .for example, the Synthesis Report or Map Report. When you run a report process, the report appears in the Workspace.

- **Tools**

  When you run a tools process, the related tool launches in standalone mode or appears in the Workspace where you can view or modify your design source files. . As you work on your design, you may make changes that require some or all of the processes to be rerun. Project Navigator keeps track of the changes you make and shows the status of each process with the following status icons:

- **Running**

  This icon shows that the process is running.

- **Up-to-date**

  This icon shows that the process ran successfully with no errors or warnings and does not need to be rerun. If the icon is next to a report process, the report is up-to-date; however, associated tasks may have warnings or errors. If this occurs, you can read the report to determine the cause of the warnings or errors.

- **Warnings reported**

  This icon shows that the process ran successfully but that warnings were encountered.

- **Errors reported**

  This icon shows that the process ran but encountered an error.

- **Out-of-Date**

  This icon shows that you made design changes, which require that the process be rerun. If this icon is next to a report process, you can rerun the associated task process to create an up-to-date version of the report.

- **No icon**

  If there is no icon, this shows that the process was never run. To run a process, you can do any of the following:

- Double-click the process
- Right-click while positioned over the process, and select Run from the popup menu, as shown in the following fig 6.3

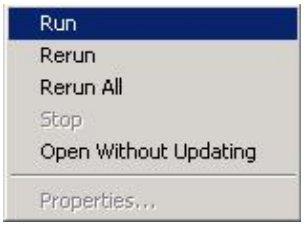

Fig 6.2.2 Running process

Select the process, and then click the Run toolbar button.

- To run the Implement Design process and all preceding processes on the top module for the design, select Process >Implement Top Module, or click the Implement Top Module toolbar button .

When you run a process, Project Navigator automatically processes your design as follows:

- Automatically runs lower-level processes
  When you run a high-level process, Project Navigator runs associated lower-level processes or sub-processes. For example, if you run Implement Design for your FPGA design, all of the following sub-processes run: Translate Map, and Place & Route.
- Automatically runs preceding processes
  When you run a process, Project Navigator runs any preceding processes that are required, thereby "pulling" your design through the design flow. For example, to pull your design through the entire flow, double-click Generate Programming File.

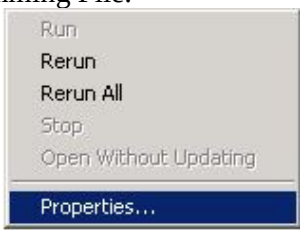

Fig 6.2.3 Selecting properties

When you enable the advanced properties, both standard and advanced properties appear in the Process Properties dialog box.

- Float
- Dock

Depending on the source file and tool you are working with, additional tabs are available in the Transcript window:

- **Always available :** Console tab, Errors tab, Warnings tab, Tcl Shell tab, Find in Files tab
- **ISE Simulator :** Simulation Console tab
- **RTL and Technology Viewers :** View by Name tab, View by Category tab Toolbars provide convenient access to frequently used commands. Click once on a toolbar button to execute a command.

Creating a Project

With your project open in Project Navigator, you can view and run processes on all the files in your design. Project Navigator provides a wizard to help you create a new project, as follows.

1. Select File > New Project.
2. In the New Project Wizard Create New Project page, do the following:

a. In the Project Name field, enter a name for the project. Follow the naming conventions described in File Naming Conventions.

b. In the Project Location field, enter the directory name or browse to the directory.

c. In the Top-Level Source Type drop-down list, select one of the following

HDL

Select this option if your top-level design file is a VHDL, Verilog, or ABEL (for CPLDs) file. An HDL Project can include lower-level modules of different file types, such as other HDL files, schematics, and "black boxes," such as IP cores and EDIF files.

- Schematic

Select this option if your top-level design file is a schematic file. A schematic project can include lower-level modules of different file types, such as HDL files, other schematics, and "black boxes," such as IP cores and EDIF files. Project Navigator automatically converts any schematic files in your design to structural HDL before implementation; therefore, you must specify a synthesis tool when working with schematic projects.

EDIF

Select this option if you converted your design to this file type, for example, using a synthesis tool. Using this file type allows yo to skip the Project Navigator synthesis process and to start with the implementation processes.

- **NGC/NGO**

Select this option if you converted your design to this file type, for example, using a synthesis tool. Using this file type allows you to skip the Project Navigator synthesis process and start with the implementation processes.

3. Click Next.
4. If you are creating an HDL or schematic project, skip to the next step. If you are creating an EDIF or NGC/NGO project, do the following in the Import EDIF/NGC Project page:

a. In the Input Design field, enter the name of the input design file, or browse to the file and select it.

b .Select Copy the input design to the project directory to copy your file to the project directory. If you do not select this option, your file is accessed from the remote location.

c. In the Constraint File field, enter the name of the constraints file, or browse to the file and select it.

d. Select Copy the constraints file to the project directo**ry** to copy your file to the project directory. If you do not select this option, your file is accessed from the remote location.

e. Click Next.

Simulator

Select one of the following simulators and the HDL language for simulation.

- **ISE Simulator (Xilinx.)**

This simulator allows you to run integrated simulation processes as part of your ISE design flow. For more information, see the ISE Simulator Help.

- **ModelSim (Mentor Graphics.)**

You can run integrated simulation processes as part of your ISE design flow using any of the following ModelSim editions: ModelSim Xilinx Edition (MXE), ModelSim MXE Starter, ModelSim PE, or ModelSim SE

- **NC-Sim (Cadence.)**

The NC-Sim simulator is not integrated with ISE and must be run standalone. For more information, see the documentation provided with the simulator.

Preferred Language

The Preferred Language project property controls the default setting for process properties that generate HDL output. If both the Synthesis Tool and Simulator options are set to mixed-language (VHDL/Verilog) tools, you can use the Preferred Language property to select the language in which generated HDL output will be created.

- **Verilog**

Select this option if both Synthesis Tool and Simulation are set to mixed language and you want the default language to be Verilog.

- **VHDL**

Select this option if both Synthesis Tool and Simulation are set to mixed language and you want the default language to be VHDL.

- **N/A**

This option will appear if both Synthesis Tool and Simulation are set to a single language.

Enable Enhanced Design Summary

Select this option to show the number of errors and warnings for each of the Detailed Reports in the Design Summary.

6. If you are creating an EDIF or NGC/NGO project, skip to step 8. If you are creating an HDL or schematic project,
7. Click Next, and optionally, add existing source files to your project in the Add Existing Sources page.
8. Click Next to display the Project Summary page.
9. Click Finish to create the project.

You can perform any of the following:

- Create and add source files to your project.
- Add existing source files to your project.
- Run processes on your source files

Creating a Source File

A source file is any file that contains information about a design. Project Navigator provides a wizard to help you create new source files for your project. Open a project in Project Navigator.

To Create a Source File

1. Select Project > New Source.
2. In the New Source Wizard, select the type of source you want to create.

Different source types are available depending on your project properties (top-level module type, device type, synthesis tool, and language).

3. Enter a name for the new source file in the File Name field. Follow the naming conventions described in File Naming Conventions.
4. In the Location field, enter the directory name or browse to the directory.
5. Select Add to Project to automatically add this source to the project.
6. Click Next.
7. If you are creating a source file that needs to be associated with an existing source file, select the appropriate source file, and click Next. If this does not apply, skip to the next step.

8. In the New Source Wizard - Summary window, verify the information for the new source, and click Finish.

Adding a Source File to a Project:

Project Navigator allows you to add an existing source file to a project. The source file can reside in the project directory or in a remote directory. If you generated your source file using the New Source wizard and selected Add to Project, you do not need to add the source file to your project; it is automatically part of your project.

FPGA Design Flow Overview:

The ISE design flow comprises the following steps: design entry, design synthesis, design implementation, and Xilinx device programming.

This section describes what to do during each step. For additional details on each design step, click a box in the following fig 6.6

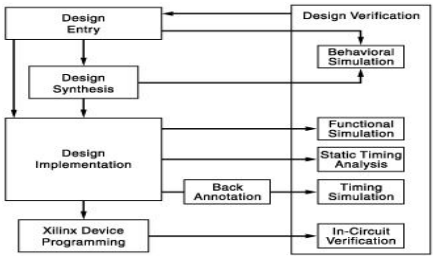

Fig 6.6 FPGA design flow

Create an ISE project as follows:

1. Create a project.
2. Create files and add them to your project, including a user constraints (UCF) file.
3. Add any existing files to your project.
4. Assign constraints such as timing constraints, pin assignments, and area constraints.

Synthesize your design.

Implement your design as follows:

1. Implement your design, which includes the following steps:

- Translate
- Map
- Place and Route

2. Review reports generated by the Implement Design process, such as the Map Report or Place & Route Report, and change any of the following to improve your design:

- Process properties
- Constraints
- Source files

3. Synthesize and implement your design again until design requirements are met. You can verify the timing of your design at different points in the design flow as follows:

- Run static timing analysis at the following points in the design flow:
- After Map
- After Place & Route
- Run timing simulation at the following points in the design flow:
- After Map (for a partial timing analysis of CLB and IOB delays)
- After Place and Route (for full timing analysis of block and net delays)

Program your Xilinx device as follows:

1. Create a programming file (BIT) to program your FPGA.
2. Generate a PROM, ACE, or JTAG file for debugging or to download to your device.
3. Use IMPACT to program the device with a programming cable.

FPGA Basic Flow:

With designs of low to moderate complexity, you can process your design using the ISE™ Basic Flow as follows:

1. Create an ISE project as follows:

a. Create a project.

b. Create files and add them to your project, including a user constraints (UCF) file.

c. Add any existing files to your project.

d. Edit the design files to specify design functionality.

e. Optionally, use the Language Templates to assist in coding of the design.

f. Edit the design test bench or waveform files to drive stimulus for testing the design files. Optionally, do the following:

- Use the Test Bench Waveform Editor to specify stimulus for the design.
- Use the Language Templates to assist in coding of the test bench.

g. Assign constraints such as timing constraints, pin assignments, and area constraints.

2. Run behavioral simulation (also known as RTL simulation).

3. Repeat steps 1 and 2 until desired functionality is achieved.

4. Synthesize your design.

5. Implement your design as follows:

Run timing simulation to verify end functionality and timing of the design.

Program your Xilinx® device as follows:

c. Create a programming file (BIT) to program your FPGA.

d. Generate a PROM, ACE, or JTAG file for debugging or to download to your device.

e. Program the device with a programming cable

FPGA IMPLEMENTATION

Introduction to FPGA

FPGA contains a two-dimensional arrays of logic blocks and interconnections between logic blocks. Both the logic blocks and interconnects are programmable.Now, to get our desired design (CPU), all the sub functions implemented in logic blocks must be connected and this is done by programming the internal structure of an FPGA which is depicted in the following figure 7.1.

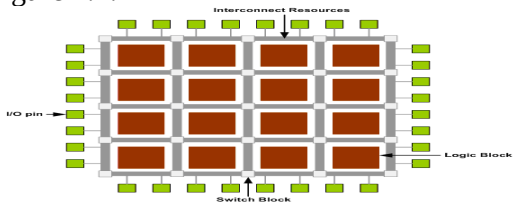

Figure 7.1: FPGA interconnections

FPGAs, alternative to the custom ICs, can be used to implement an entire System On one Chip (SOC). The main advantage of FPGA is ability to reprogram. User can reprogram an FPGA to implement a design and this is done after the FPGA is manufactured.

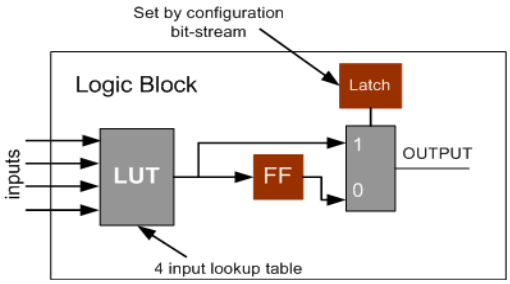

Figure 7.1.2 shows a 4-input LUT based implementation of logic block

LUT based design provides for better logic block utilization. A k-input LUT based logic block can be implemented in number of different ways with tradeoff between performance and logic density.

Interconnects

A wire segment can be described as two end points of an interconnection with no programmable switch between them.

FPGA DESIGN FLOW

In this part of tutorial we are going to have a short intro on FPGA design flow. A simplified version of design flow is given in the flowing diagram.

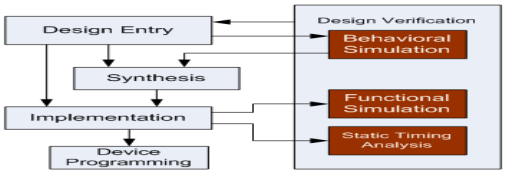

Figure 7.2 FPGA Design Flow

Design Entry

If the designer wants to deal more with Hardware, then Schematic entry is the better choice. When the design is complex or the designer thinks the design in an algorithmic way then HDL is the better choice. Language based entry is faster but lag in performance and density.

Synthesis

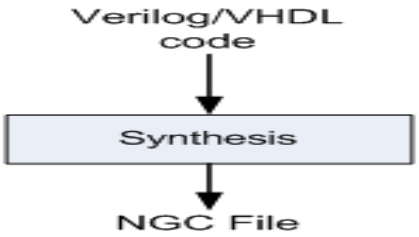

Figure 7.2.2 FPGA Synthesis

The process that translates VHDL/ Verilog code into a device netlist format i.e. a complete circuit with logical elements (gates flip flop, etc…) for the design. The resulting netlist(s) is saved to an NGC (Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

Implementation

This process consists of a sequence of three steps

- Translate
- Map
- Place and Route

Translate:

Process combines all the input netlists and constraints to a logic design file. This information is saved as a NGD (Native Generic Database) file. This can be done using NGD Build program.

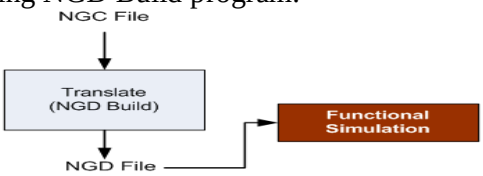

Figure 7.2.3 FPGA Translate

Map:

Process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

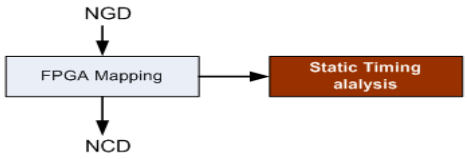

Figure 7.2.3.2 FPGA map

Place and Route:

PAR program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output.

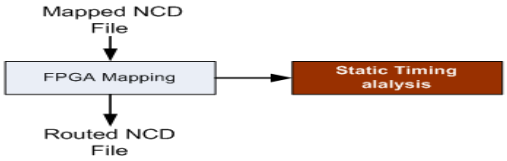

Figure 7.2.3.3 FPGA Place and route

Schematic Diagrams

To investigate the advantages of using our technique in terms of area overhead against "Fully ECC" and against the partially protection, we implemented and synthesized for a Xilinx XC3S500E different versions of a 32-bit, 32-entry, dual read ports, single write port register file. Once the functional verification is done, the RTL model is taken to the synthesis process using the Xilinx ISE tool.

RTL Schematic

The RTL (Register Transfer Logic) can be viewed as black box after synthesize of design is made. It shows the inputs and outputs of the system. By double-clicking on the diagram we can see gates, flip-flops and MUX.

The corresponding schematics of the adders after synthesis is shown below.

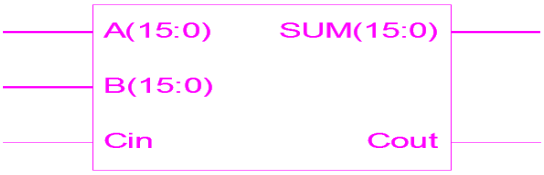

Figure 7.3.1: Top-level of Ripple-Carry Adder

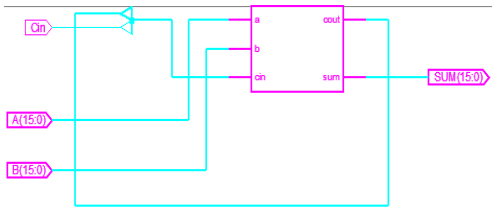

Figure 7.3.2 : Internal block of Ripple-Carry Adder

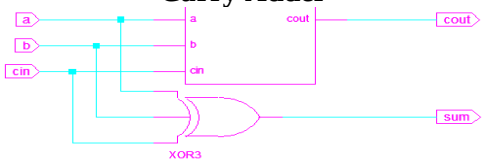

Figure 7.3.3 : Internal block of above figure

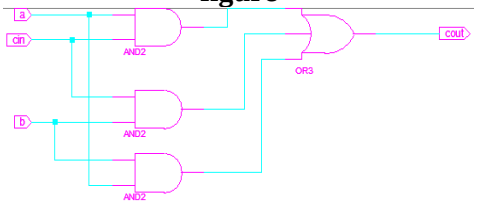

Figure 7.3.4: Internal block of cout

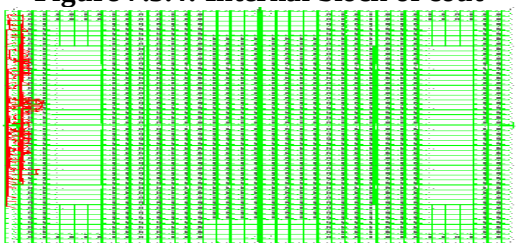

Figure 7.3.5: Area occupied by Ripple-Carry Adder

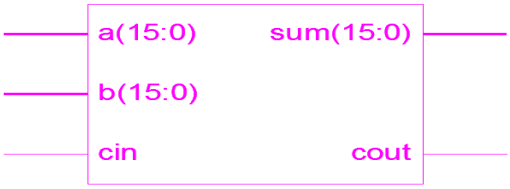

Figure 7.3.6: Top-level of Carry-Select Adder

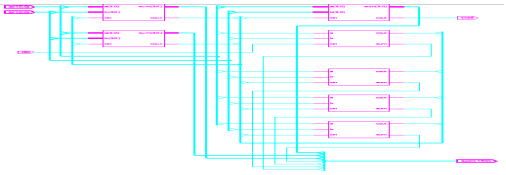

Figure 7.3.7 : Internal block of Carry-Select Adder

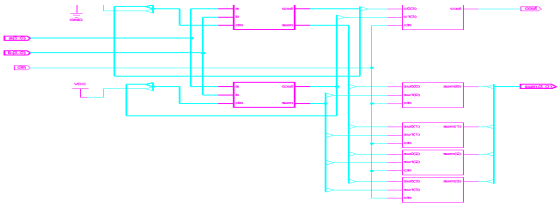

Figure 7.3.8 : Instance of the above block

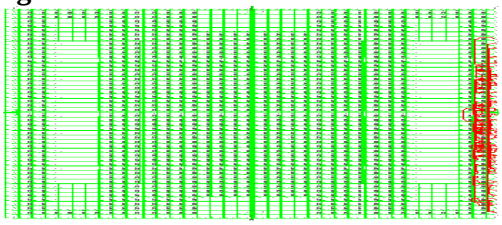

Figure 7.3.9 : Area occupied by Carry-Select Adder

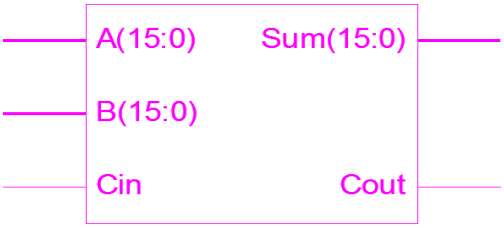

Figure 7.3.10 : Top-level of Carry-Skip Adder

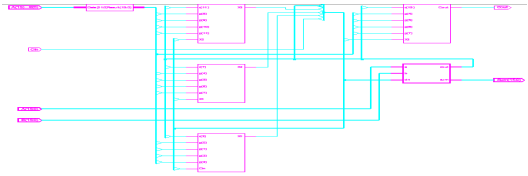

Figure 7.3.11 : internal block of Carry-Skip Adder

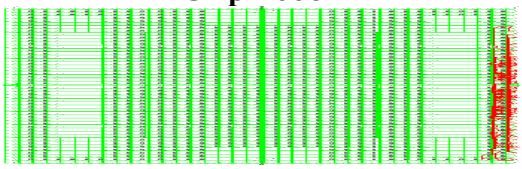

Figure 7.3.12 : Area occupied by 16-bit Carry-Skip Adder

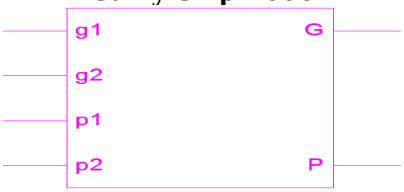

Figure 7.3.13 : Top level of Black Cell

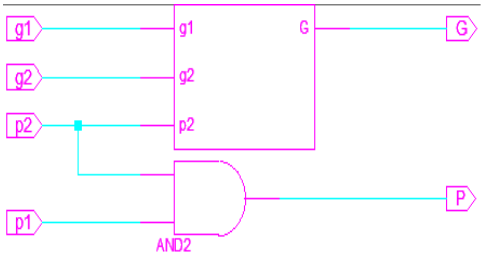

Figure 7.3.14 : Internal block of Black Cell

Figure 7.3.15 : Top level of Gray Cell

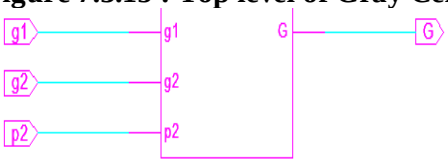

Figure 7.3.16 : Internal block of Gray Cell

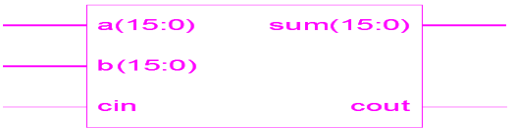

Figure 7.3.16 : Top-level of Kogge-Stone Adder

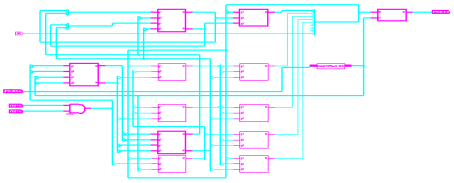

Figure 7.3.17 : Internal block of Kogge-Stone Adder

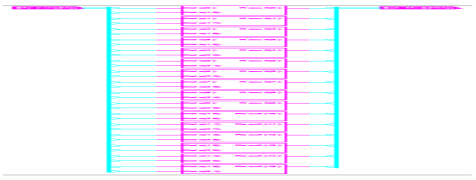

Figure 7.3.18 : Instance of the above block

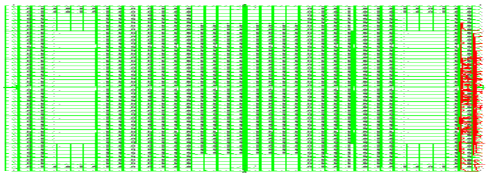

Figure 7..3.19 : Area occupied by 16-bit Kogge-Stone Adder

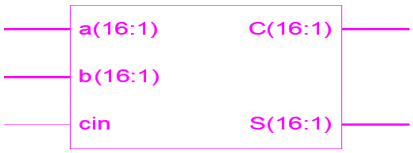

Figure 7.3.20 : Top-level of Sparse Kogge-Stone Adder

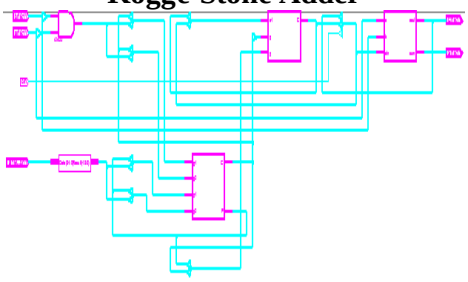

Figure 7.3.21 : Internal block of Sparse Kogge-Stone Adder

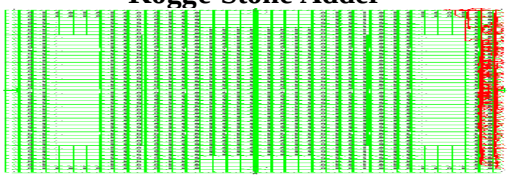

Figure 7.3.22 : Area occupied by 16-bit Sparse Kogge-Stone Adder

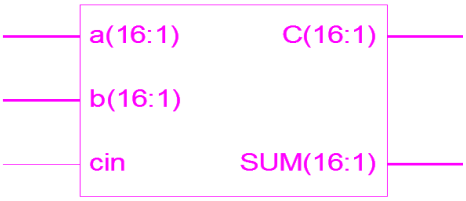

Figure 7.3.23 : Top-level of Spanning Tree Adder

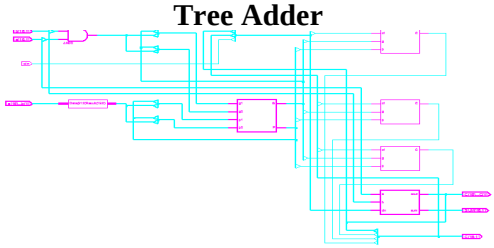

Figure 7.3.24: Internal block of Spanning Tree Adder

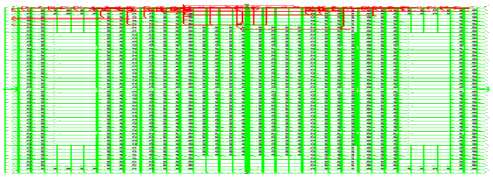

Figure 7.3.25 : Area occupied by 16-bit Spanning Tree Adder

Synthesis Result

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

This device utilization includes the following.

- Logic Utilization
- Logic Distribution
- Total Gate count for the Design

The device utilization summery is shown above in which its gives the details of number of devices used from the available devices and also represented in %. Hence as the result of the synthesis process, the device utilization in the used device and package is shown below.

Table 7-4-1: Synthesis report of Ripple-Carry Adder

par Project Status (08/18/2011 - 12:57:42)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	Ripple_Carry	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	No Warnings
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	32	9,312	1%	
Logic Distribution				
Number of occupied Slices	24	4,656	1%	
Number of Slices containing only related logic	24	24	100%	
Number of Slices containing unrelated logic	0	24	0%	
Total Number of 4 input LUTs	32	9,312	1%	
Number of bonded IOBs	50	232	21%	

Table 7-4-2: Synthesis report of Carry-Select Adder

par Project Status (08/18/2011 - 13:08:00)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	carry_select_adder	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	1 Warning
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	47	9,312	1%	
Logic Distribution				
Number of occupied Slices	26	4,656	1%	
Number of Slices containing only related logic	26	26	100%	
Number of Slices containing unrelated logic	0	26	0%	
Total Number of 4 input LUTs	47	9,312	1%	
Number of bonded IOBs	50	232	21%	

Table 7-4-3: Synthesis report of Carry-Skip Adder

par Project Status (08/18/2011 - 13:14:46)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	carry_skip	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	2 Warnings
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	40	9,312	1%	
Logic Distribution				
Number of occupied Slices	25	4,656	1%	
Number of Slices containing only related logic	25	25	100%	
Number of Slices containing unrelated logic	0	25	0%	
Total Number of 4 input LUTs	40	9,312	1%	
Number of bonded IOBs	50	232	21%	

Table 7-4-4: Synthesis report of Kogge-Stone Adder

par Project Status (08/18/2011 - 12:42:13)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	Kogge_Stone	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	4 Warnings
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	37	9,312	1%	
Logic Distribution				
Number of occupied Slices	24	4,656	1%	
Number of Slices containing only related logic	24	24	100%	
Number of Slices containing unrelated logic	0	24	0%	
Total Number of 4 input LUTs	37	9,312	1%	
Number of bonded IOBs	50	232	21%	

Table 7-4-5: Synthesis report of Sparse Kogge-Stone Adder

par Project Status (08/18/2011 - 12:33:15)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	Sparse_Kogge	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	No Warnings
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	51	9,312	1%	
Logic Distribution				
Number of occupied Slices	30	4,656	1%	
Number of Slices containing only related logic	30	30	100%	
Number of Slices containing unrelated logic	0	30	0%	
Total Number of 4 input LUTs	51	9,312	1%	
Number of bonded IOBs	65	232	28%	

Table 7-4-6: Synthesis report of Spanning Tree Adder

par Project Status (08/18/2011 - 12:45:20)			
Project File:	par.ise	**Current State:**	Placed and Routed
Module Name:	Spanning_Tree	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	3 Warnings
Product Version:	ISE 10.1 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

par Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	32	9,312	1%	
Logic Distribution				
Number of occupied Slices	24	4,656	1%	
Number of Slices containing only related logic	24	24	100%	
Number of Slices containing unrelated logic	0	24	0%	
Total Number of 4 input LUTs	32	9,312	1%	
Number of bonded IOBs	65	232	28%	

SIMULATION RESULTS

8.1 SIMULATION RESULTS

The Simulation Inputs are Taken A has 16 bits taken as 0011001100110011 and B has 16 bits taken as 0011001100110011 and Carry input is taken as 0 then All Adders Simulation Results are Shown in the below figures.

8.1.1 Brent – Kung Adder

Figure 8.1.1 : Brent – Kung Adder

8.1.2 Kogge Stone Adder

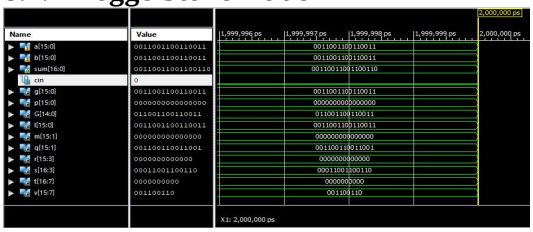

Figure 8.1.2 : Kogge Stone Adder

8.1.3 Ladner – Fischer Adder

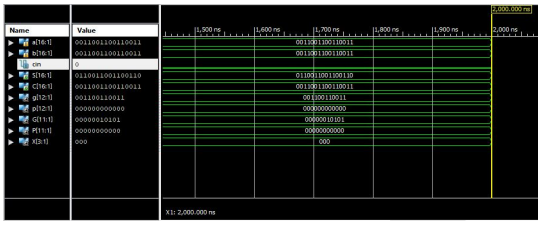

Figure 8.1.3 : Ladner – Fischer Adder

8.1.4 Sklansky Adder

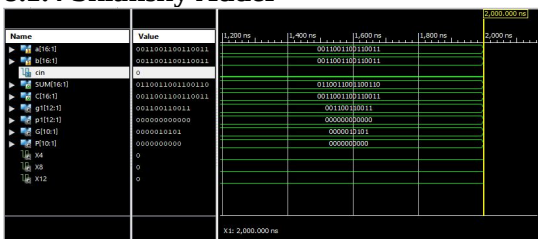

Figure 8.1.4 : Sklansky Adder

CONCLUSION AND FUTURE SCOPE

Both measured and simulation results from this study have shown that parallel-prefix adders are not as effective as the simple ripple-carry adder at low to moderate bit widths. This is not unexpected as the Xilinx FPGA has a fast carry chain which optimizes the performance of the ripple carry adder. However, contrary to other studies, we have indications that the carry-tree adders eventually surpass the performance of the linear adder designs at high bit-widths, expected to be in the 128 to 256 bit range. This is important for large adders used in precision arithmetic and cryptographic applications where the addition of numbers on the order of a thousand bits is not uncommon. Because the adder is often the critical element which determines to a large part the cycle time and power dissipation for many digital signal processing and cryptographical implementations, it would be worthwhile for future FPGA designs to include an optimized carry path to enable tree based adder designs to be optimized for place and routing.

This would improve their performance similar to what is found for the RCA. We plan to explore possible FPGA architectures that could implement a "fast-tree chain" and investigate the possible trade-offs involved. The built-in redundancy of the Kogge-Stone carry-tree structure and its implications for fault tolerance in FPGA designs is being studied.

REFERENCES

[1] N. H. E. Weste and D. Harris, *CMOS VLSI Design*, 4th edition, Pearson–Addison-Wesley, 2011.

[2] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, pp. 260-264, 1982.

[3] D. Harris, "A Taxonomy of Parallel Prefix Networks," in *Proc. 37th Asilomar Conf. Signals Systems and Computers*, pp. 2213–7, 2003.

[4] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. on Computers*, Vol. C-22, No 8, August 1973.

[5] P. Ndai, S. Lu, D. Somesekhar, and K. Roy, "Fine- Grained Redundancy in Adders," *Int. Symp. on Quality Electronic Design*, pp. 317-321, March 2007.

[6] T. Lynch and E. E. Swartzlander, "A Spanning Tree Carry Lookahead Adder," *IEEE Trans. on Computers*, vol. 41, no. 8, pp. 931-939, Aug. 1992.
[7] D. Gizopoulos, M. Psarakis, A. Paschalis, and Y. Zorian, "Easily Testable Cellular Carry Lookahead Adders," *Journal of Electronic Testing: Theory and Applications 19*, 285-298, 2003.
[8] S. Xing and W. W. H. Yu, "FPGA Adders: Performance Evaluation and Optimal Design," *IEEE Design & Test of Computers*, vol. 15, no. 1, pp. 24-29, Jan. 1998.
[9] M. Bečvář and P. Štukjunger, "Fixed-Point Arithmetic in FPGA," *Acta Polytechnica*, vol. 45, no. 2, pp. 67- 72, 2005.
[10] K. Vitoroulis and A. J. Al-Khalili, "Performance of Parallel Prefix Adders Implemented with FPGA technology," *IEEE Northeast Workshop on Circuits and Systems*, pp. 498-501, Aug. 2007. 172