INTERNATIONAL JOURNAL OF MERGING TECHNOLOGY AND ADVANCED RESEARCH IN COMPUTING

IJMTARC – VOLUME – VI – ISSUE – 24, OCT-DEC, 2018 ISSN: 2320-1363

DESIGN AND CHARACTERIZATION OF HIGH SPEED MULTIPLIER USING ADDERS

G.LAKSHMI BHARATH[1], N.NAGA MALLIKARJUNA[2]
[1]PG Student, Dept of ECE, SITS, Kadapa, AP, India.
[2]Associate Professor, Dept of ECE, SITS, Kadapa, AP, India.

***Abstract-* Binary adders are known as important elements in the circuit designs. Many fastest adders have been created and developed. Parallel Prefix Adders (PPA) is one among them. We use adders frequently in digital design and VLSI designs, in digital design we use adders such as half adder, full adder. By using both adders we can implement ripple carry adder, using ripple carry adder we can perform addition for any number of bits. It is a serial adder. It has a huge delay problem. With the use of half adder, full adder delay increases. To overcome this Parallel Prefix Adders are preferred. In VLSI implementation parallel prefix adders are known to have the best performance. This paper presents an implementation of various types of carry tree adders (the Kogge- Stone, Sparse Kogge- Stone, ripple Carry adder and carry look ahead adders. And this adders where implemented on Array multipliers to know the performance analysis of the proposed adders. We report on delay, area requirements. These designs of varied on different bit widths and simulated using xilinx14.2 version Spartan 3E FPGA, These carry tree adders support bit width of 256.**

I. INTRODUCTION

Digital computer arithmetic is an aspect of logic design with the objective of developing appropriate algorithms in order to achieve an efficient utilization of the available hardware. The basic operations are addition, subtraction, multiplication and division. In this, we are going to deal with the operation of additions implemented to the operation of multiplication. The hardware implementation of binary addition is a fundamental architectural component in many processors, such as microprocessors, digital signal processors, mobile devices and other hardware applications. In these systems when building arithmetic logic unit (ALU), adders play an important role for performing the basic arithmetic operations, such as addition, subtraction, multiplication, division, etc. Therefore, the hardware implementation of an effective adder is necessary to increase the performance of ALU and, consequently, the processor itself as a whole. Currently, a parallel prefix adder (PPA) is considered effective adder for performing the addition of two multi-bit numbers. Circuit complexity and the speed of PPA are important. Parameters at the stage of efficient hardware implementation and, therefore, in recent years various types of PPA with different characteristics of the parameters have been developed.

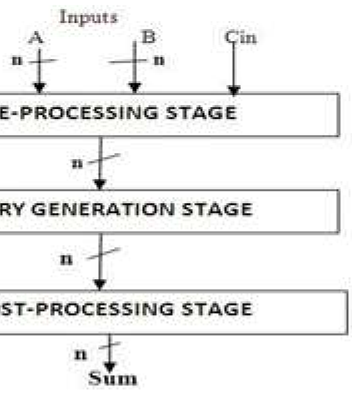

Figure 1. Block diagram of PPA

II. RESEARCH WORK

The parallel prefix adder employs three stages in pre- processing stage the generation of Propagate and Generate signals is carried out. The calculation of Generate (Gi) and Propagate (Pi) are calculated when the inputs A, B are given. As follows

Gi=Ai AND Bi Pi=Ai XOR Bi

Gi indicates whether the Carry is generated from that bit. Pi indicates whether Carry is propagated from that bit. In carry generation stage of PPA, prefix graphs can be used to describe the tree structure. Here the tree structure consists of grey cells, black cells, and buffers. In carry generation stage when two pairs of generate and propagate signals (Gm, Pm), (Gn, Pn) are given as inputs to the carry generation stage. It computes a pair of group generates and group propagate signals (Gm: n, Pm: n) which are calculated as follows

IJMTARC – VOLUME – VI – ISSUE – 24, OCT-DEC, 2018 ISSN: 2320-1363

Gm: n=Gm+ (Pm.Gn), Pm: n=Pm. Pn

The black cell computes both generate and propagate signals as output. It uses two and gates and or gate. The grey cell computes the generate signal only. It uses only and gate, or gate.

In post processing stage simple adder to generate the sum, Sum and carry out are calculated in post processing stage as follows

Si=Pi XOR Ci-1

Cout=Gn-1 XOR (Pn-1 AND Gn-2)

If Cout is not required it can be neglected.

III. EXISTING ADDER DESIGNS

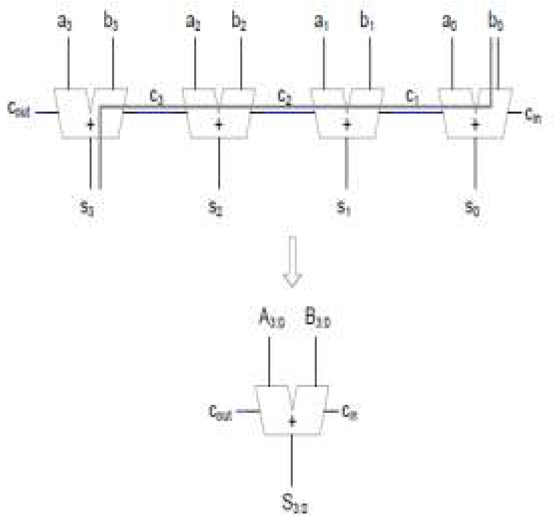

Fig. 2:4-bit Ripple Carry Adder

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 2. Each trapezoidal symbol represents a single-bit full adder. At the top of the figure, the carry is rippled through the adder from c_{in} to c_{out}.

It can be observed in Figure 2 that the critical path, highlighted with a solid line, is from the least significant bit (LSB) of the input (a_0 or b_0) to the most significant bit (MSB) of sum (s_{n-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 delay is calculated as follows.

a_i , $b_i \rightarrow s_i$ = 10/\

a_i , $b_i \rightarrow c_{i+1}$ = 9/\

$c_i \rightarrow s_i$ = 5/\

$c_i \rightarrow c_{i+1}$ = 4/\

The critical path or the worst delay is

t_{RCA} = {9 + (n- 2) x 4 + 5}/\ = {4n + 6}/\

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

A) Carry Look Ahead Adder :

Carry Look Ahead Adder can produce carries faster due to parallel generation of the carry bits by using additional circuitry. This technique uses calculation of carry signals in advance, based on input signals. The result is reduced carry propagation time. For example, ripple adders are slower but use the least energy.

A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower (ripple carry adder), for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits. The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.

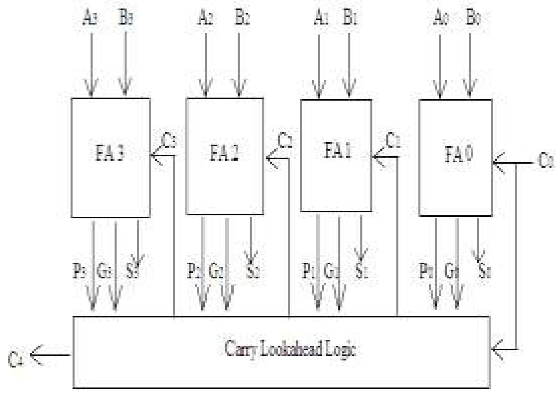

Fig. 3: 4-bit carry look ahead adder

B)4-Bit Full Adder With Look Ahead Carry:

Notice that the final output carry is expressed as a function of the input variables in SOP form, which is a two level AND-OR or equivalent NAND-NAND function. To produce the output carry for any particular stages, it is clear that it requires only that much time required for the signal to pass through two levels only. In effect, we examined the inputs at all the n stages to produce the output carry for the most significant (n-1)th stage. Hence the circuit for carry look ahead carry introduces a delay of two levels. Notice that the full look ahead scheme requires the use of OR gate with (n+1) inputs and AND gates with numbers of

IJMTARC – VOLUME – VI – ISSUE – 24, OCT-DEC, 2018 ISSN: 2320-1363

inputs varying from 1 through n+1. For increasing world lengths, it becomes unwieldy.

Four stages carry look ahead parallel adders are commercially available in integrated chip form represented as a block diagrams. It is possible to have hierarchical levels of look ahead group carry scheme to further reduces the addition time and make it faster. Such scheme involves large number of gates.

Generation of all outputs carrier in the look ahead circuit takes two more levels after the P_i and G_i signals settle into their final values.
Two more levels produce the sums.

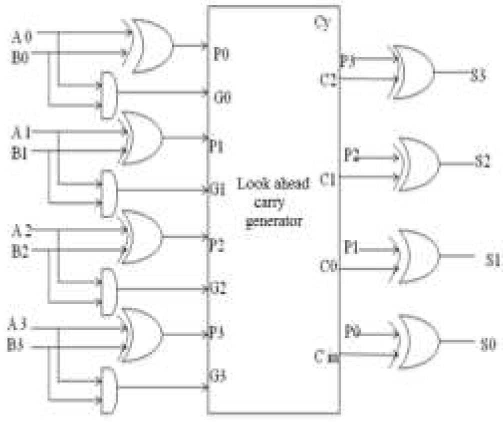

Fig. 4: 4-Bit Carry Look Ahead adder implementation details

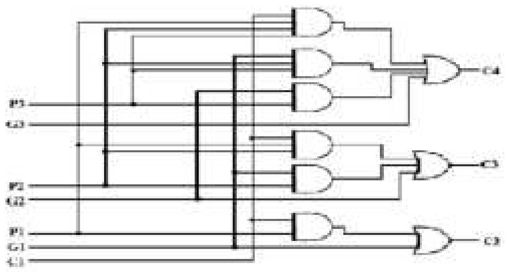

Fig.5: Internal block diagram of carry look ahead generator

III.PROPOSED ADDERS

To resolve the delay of carry look ahead adders, the scheme of multilevel-look ahead adders or parallel-prefix adders can be employed.

The idea is to compute small group of intermediate prefixes and then find large group prefixes, until all the carry bits are computed. 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 pre-computation stage, each bit computes its carry generate/propagate and a temporary sum. 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.

$G_{i:k} = G_{i:j} + P_{i:j} \cdot G_{j-1:k}$

$P_{i:k} = P_{i:j} \cdot P_{j-1:k}$

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.

$s_i = p_i \wedge G_{i:-1}$

$c_{out} = G_i + (P_i \cdot G_{i-1})$

where $G_{i:-1} = c_i$ with the assumption $g_{-1} = c_{in}$. The general diagram of parallel-prefix structures is shown in Figure 6, where an 8-bit case is illustrated.

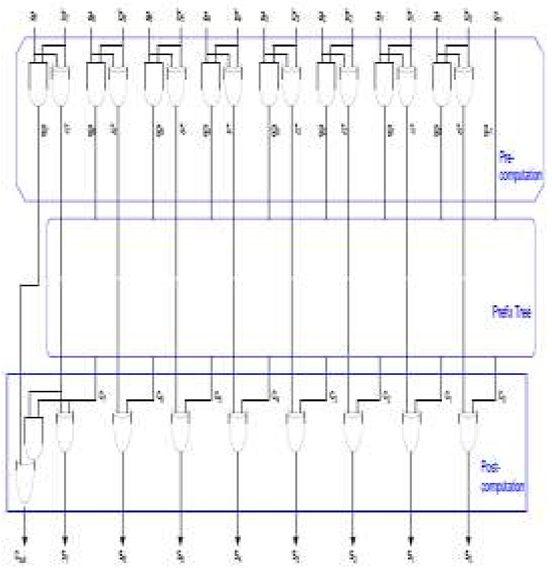

Fig. 6: 8-bit Parallel-Prefix Structure with carry look ahead notation

All parallel-prefix structures can be implemented with the equations above, however, Equation can be interpreted in various ways, which leads to different types of parallel-prefix trees. For example, Kogge stone is known for its sparse topology at the cost of more logic levels.

i)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. 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.

$g_i = a_i \cdot b_i$

$p_i = a_i \wedge b_i$

where $0 < i < n$, $g_{-1} = cin$ and $p_{-1} = 0$.

Sometimes, p_i 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 t_i.

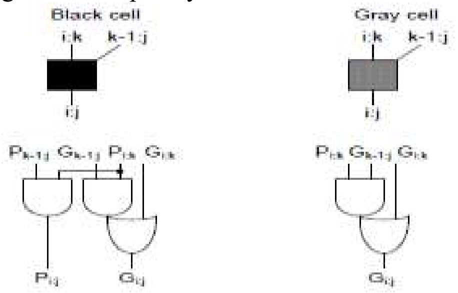

Fig.7: Cell Definitions

In the prefix tree, group generate/propagate signals are computed at each bit.

$G_{i:k} = G_{i:j} + P_{i:j} \cdot G_{j-1:k}$

$P_{i:k} = P_{i:j} \cdot P_{j-1:k}$

In the post-computation, the sum and carry-out are the final output.

$s_i = p_i \cdot G_{i-1:-1}$

$c_{out} = G_{n:-1}$

where "-1" is the position of carry-input. The generate/propagate signals can be grouped in different fashion to get the same correct carries. Based on different ways of grouping the generate/propagate signals, different prefix architectures can be created. Figure7 shows the definitions of cells that are used in prefix structures, including black cell and gray cell. Black/gray cells implement the above two equations, which will be heavily used in the following discussion on prefix trees.

ii)Kogge-Stone Prefix Tree:

Kogge-Stone prefix tree is among the type of prefix trees that use the fewest logic levels. A 8-bit example is shown in Figure 8.

In fact, Kogge-Stone is a member of Knowles prefix tree. 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.

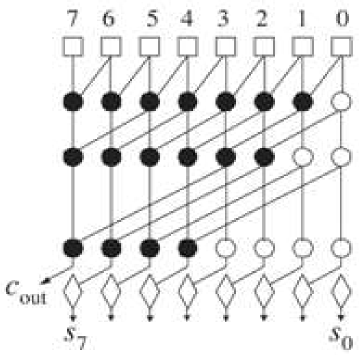

Fig. 8: 8-bit Kogge-Stone Prefix Tree

The key of building a prefix tree is how to implement Equation according to the specific features of that type of prefix tree and apply the rules described in the previous section. Gray cells are inserted similar to black cells except that the gray cells final output carry outs instead of intermediate G/P group.

The reason of starting with Kogge-Stone prefix tree is that it is the easiest to build in terms of using a program concept. The example in Figure 3.3.2.1 is 8-bit (a power of 2) prefix tree.

It is not difficult to extend the structure to any width if the basics are strictly followed.

The number cells for a Kogge-Stone prefix tree can be counted as follows. Each logic level has n-m cells, where $m = 2^{1\ \text{level}\ -\ 1}$. 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 nlog $_2$n subtracting the total number of cells missing at each logic level.

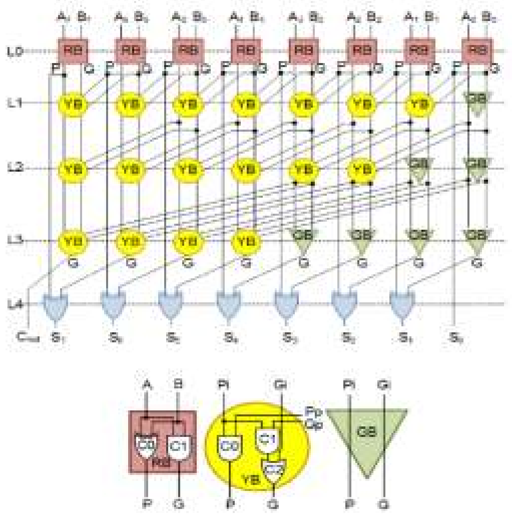

Fig. 9: 8 bit Kogge-Stone adder

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. Here we designate BC as the black cell which generates the ordered pair, the gray cell (GC) generates the left signal only.

The regularity of the Kogge-Stone prefix network has built in redundancy which has implications for fault-tolerant designs. The sparse Kogge-Stone adder, shown in Figure 8, is also studied. This hybrid design completes the summation process with a 4-bit RCA allowing the carry prefix network to be simplified.

iii) SPARSE KOGGE STONE ADDER:

The 8-bit Sparse Kogge Stone Adder is shown in below figure.

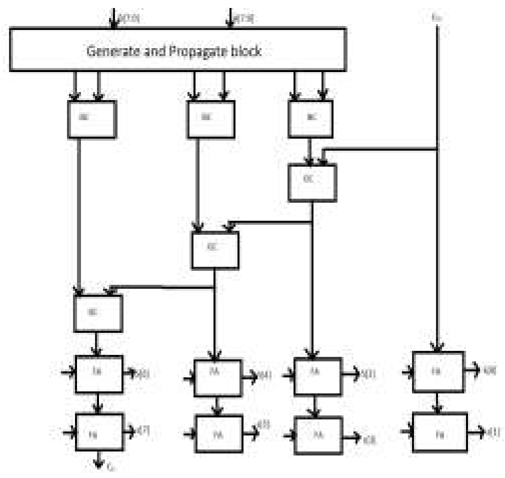

Fig. 10: 8 bit sparse Kogge-Stone adder

IV. IMPLEMENTATION OF PROPOSED ADDERS ON ARRAY MULTIPLIER

Today's digital signal processing applications, multipliers play a major part. The advancement in the technology, many researchers have design different multipliers which offer either high speed, regularity of layout, low power consumption or less area. The combination of above features in one multiplier, result suitable for various high speed and low power applications of VLSI implementation.

The adding and shift procedure is common multiplication method. Mathematical operation which is an abbreviate procedure of adding an integer to itself, a specific number of times is called multiplication. It can be defined as the multiplicand is added to itself a number of times as specified by the multiplier to form a result (product).

Among all arithmetic operations multiplication requires more amount of time and multiplication hardware requires much area. The basic building block in the Digital signal processors is a multiplier unit. The algorithms are performed by Digital signal processors depends on the performance of the multiplier operations. One of the multiplication based operations is Multiply and Accumulate (MAC). These multiplication based operations are used in different applications of Digital Signal Processing such as filtering, convolution, Fast Fourier Transform (FFT). Usually the MAC unit is used in microprocessors arithmetic and logic unit.

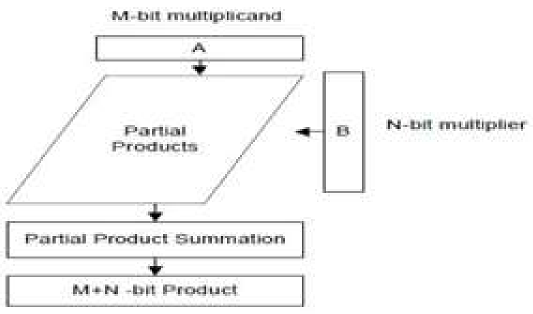

Fig. 11: General Multiplier block

To perform an M-bit by N-bit multiplication shown in the figure 12, the M-bit multiplicand $A = a_{(M-1)}a_{(M-2)....}a_1a_o$ is multiplied by the N-bit multiplier $B = b_{(N-1)}b_{(N-2)....}b_1b_o$ to produce the M+N-bit product $P=P_{(M+N-1)}P_{(M+N-2)}\ldots P_1P_o$. Any multiplier consists of three stages. The first stage is partial products generation stage. In this first stage, the multiplicand and the multiplier are multiplied bit by bit to produce the partial products. Second stage is partial products addition stage. This stage is the most important stage because it is the most complicated stage. This stage determines the speed of the overall multiplier and the third stage is final addition stage. In the last stage, the all row outputs are added using any high speed adder to generate the output result.

Array Multiplier:

Each bit of the multiplicand is multiplied by a bit in the multiplier, generating N partial products. The multiplicand shifted by some amount, or 0 is process to generate all of these partial products. The figure 12 shown illustrated for an M×N multiply operation. The hardware is directly mapped by the figure and this hardware is called the array multiplier.

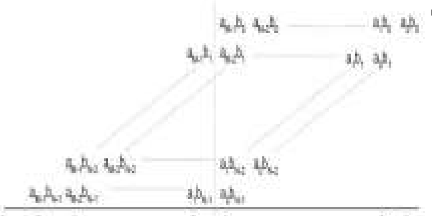

Fig.12: Partial product array for an M × N multiplier

The figure 13 shows a 4x4 unsigned array multiplier. Product of the multiplier and the multiplicand results the partial products.

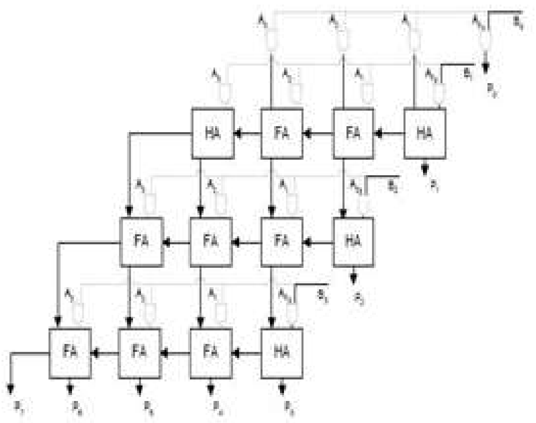

Fig. 13: Block diagram of 4x4 array multiplier

The ripple adders are used to add the partial products which are generated in the multiplication process. Thus, the carry out generated from the least significant bit ripples to the most significant bit of the similar row, and then down the next row of the structure. The partial products are generated by the AND gates and these partial products are added in ripple fashion. Half and Full adders are generally used to add the partial products in each row. A full adder's inputs involve the carrying from the previous full adder in its row and the sum from a full adder in the above row.

V. SIMULATION RESULTS

The simulation results are obtained from XILINX 13.2 simulation software. Fig 14,15 shows the simulation results of black cell and gray cell, which are the basic elements of parallel prefix adders, respectively.

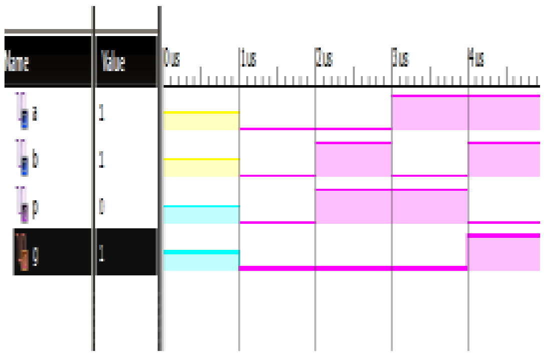

Fig. 14: Simulation results of Black Cell

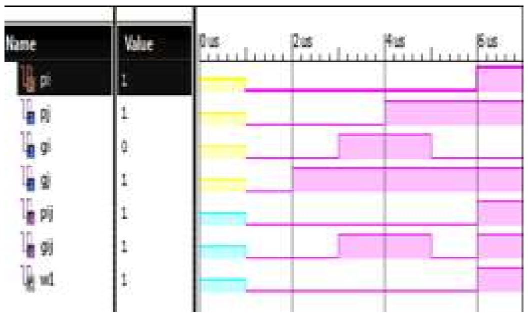

Fig. 15: Simulation results of Gray Cell

Figures 16, 17, 18 and 19 shows the simulation result of 8-bit Ripple carry adder, Carry look ahead adder, Koggestone adder, Sparse koggestone adder respectively.

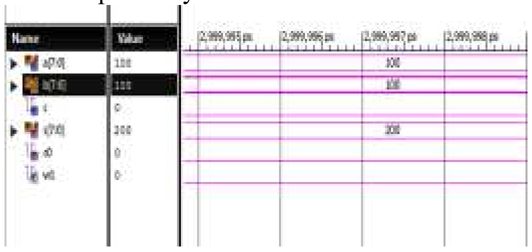

Fig.16: Simulation results of 8-bit RCA

INPUT: A=01100100 (100), B=01100100 (100), C=0
OUTPUT: S=11001000 (200), C_0=0

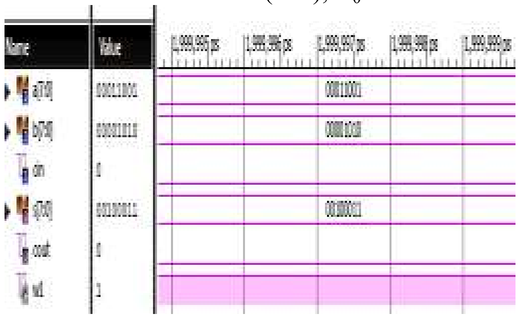

Fig. 17: Simulation results of 8-bit CLA

IJMTARC – VOLUME – VI – ISSUE – 24, OCT-DEC, 2018 ISSN: 2320-1363

INPUT: A=00011001 (25), B=00001010 (10), C_{in}=0
OUTPUT: S=00100011 (35), C_{out}=0

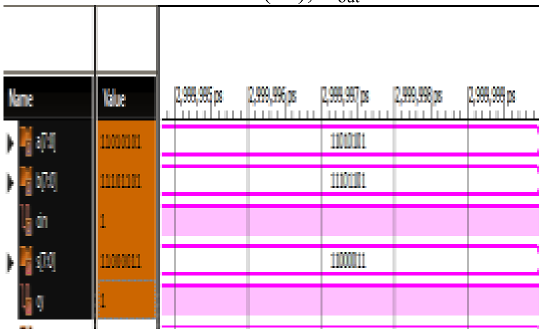

Fig. 18: Simulation results of 8-bit KSA

INPUT: A=11010101 (213), B=11101101 (237),C_{in}=1
OUTPUT: S=11000011 (195), C_y=1

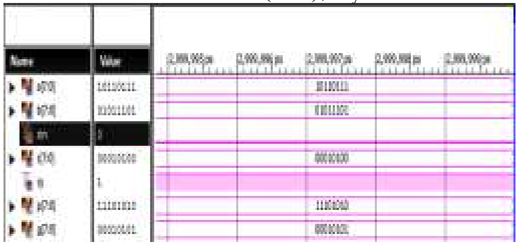

Fig. 19: Simulation results of 8-bit SKSA

INPUT: A=10110111 (183), B=01011101 (93), C_{in}=0
OUTPUT: S=00010100 (20), C_y=1

Figure 20, 21, 22, and 23 shows the simulation results of 8-bit array multipliers using RCA, CLA, KSA and SKSA respectively.

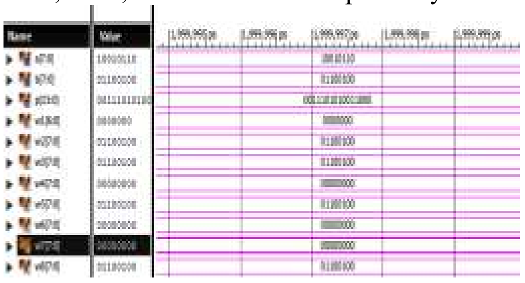

Fig. 20: Simulation result of 8-bit array multiplier using RCA

INPUT: A=10010110 (150), B=01100100 (100)
OUTPUT: P=0011101010011000 (15000)

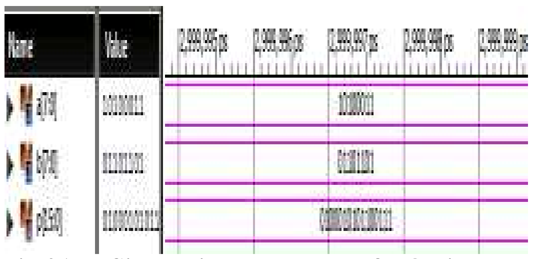

Fig.21: Simulation result of 8-bit array multiplier using CLA

INPUT: A=10100011 (163), B=01101101 (109)
OUTPUT: P=0100010101100111 (17767)

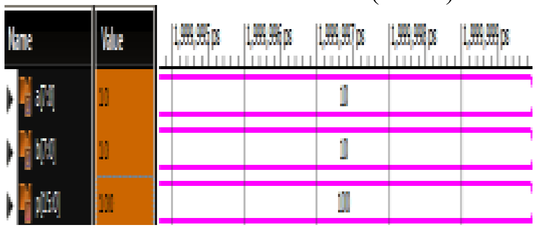

Fig. 22: Simulation results of 8-bit array multiplier using KSA

INPUT: A=00001010 (10), B=00001010 (10)
OUTPUT: P=0000000001100100 (100)

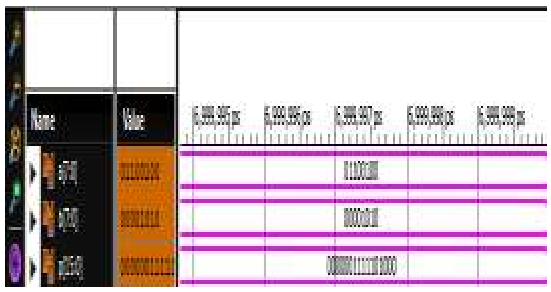

Fig 23: Simulation results of 8-bit array multiplier using SKSA

INPUT: A=011000100 (100), B=00001010 (10)
OUTPUT: P=0000001111101000 (1000)

I.Comparison Tables:

The following table shows comparison of delay in adders and multipliers from synthesis results.

Table 1: Delay values in ns of various adders

No. of bits	No. of IOBs required	Delay (ns)			
		Ripple Carry Adder	Carry Look Ahead Adder	Koggestone Adder	Sparse Koggestone Adder

4	14	8.959	8.920	4.457	8.334
8	26	13.203	13.047	5.800	11.122

Table 2: No. of slice LUTs occupied by various adders

No. of bits	No. of slice LUTs			
	Ripple Carry Adder	Carry Look Ahead Adder	Koggestone Adder	Sparse Koggestone Adder
4	4	9	6	5
8	9	92	25	8

Table 3: Delay values in ns of array multiplier using various adders

No. of bits	No. of IOBs required	Delay (ns)			
		Ripple Carry Adder	Carry Look Ahead Adder	Koggestone Adder	Sparse Koggestone Adder
4	16	17.541	12.929	6.678	13.475
8	32	36.711	28.605	21.579	24.703

Table 4: No. of slice LUTs occupied in array multiplier using various adders

No. of bits	No. of slice LUTs			
	Ripple Carry Adder	Carry Look Ahead Adder	Koggestone Adder	Sparse Koggestone Adder
4	18	8	23	18
8	73	32	196	66

VI. CONCLUSION

In this project, an efficient array multiplier using parallel prefix adders is designed, to improve the performance when compared to conventional array multiplier. From the synthesis results, it is concluded that Kogge Stone Adder is better in terms of speed. And also the Sparse Kogge Stone adder is a compromise between Carry Look Ahead Adder and Kogge Stone Adder in terms of delay and area.

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 Look ahead 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

[11] Krishna Kumari V, Sri Chakrapani Y, and Kamaraju M (2013), "Design and Characterization of Koggestone, Sparse Koggestone, Spanning Tree and Brentkung Adders", International Journal of Scientific & Engineering Research, Vol. 4, No. 10, pp. 1502-1506, ISSN 2229-5518.

[12] Al-Khalili, Dr. A.J. (2006). "Parallel Prefix Adders", Concordia University: Kostas Vitoroulis.