ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

AUTOMATED HDL GENERATION OF TWO'S COMPLEMENT DADDA MULTIPLIER WITH PARALLEL PREFIX ADDERS

Bharat Kumar Potipireddi[1], Dr. Abhijit Asati[2]

PG Student [Microelectronics], Dept. of EEE, BITS-PILANI, Pilani, Rajasthan, India [1]

Assistant Professor, Dept. of EEE, BITS-PILANI, Pilani, Rajasthan, India [2]

ABSTRACT: Dadda multipliers are among the fastest multipliers owing to their logarithmic delay. The partial products of two's complement multiplication are generated by an algorithm described by Baugh-Wooley. The complicated and irregular reduction of partial products by Dadda algorithm and use of Parallel Prefix adders with logarithmic delay in the final stage of addition makes it difficult to write a generic Verilog code for them. To solve this difficulty, we described a C program which automatically generates a Verilog file for a Dadda multiplier with Parallel Prefix adders like Kogge-Stone adder, Brent-Kung adder and Han-Carlson adder of user defined size. We compared their post layout results which include propagation delay, area and power consumption. The Verilog codes have been synthesized using 90 nm technology library. We observed that the multiplier using Kogge-Stone adder in the final stage gives higher speed and lower Power Delay Products when compared to that using Brent-Kung and Han-Carlson adders.

Keywords: Kogge-Stone adder, Dadda multiplier, Brent-Kung adder, Han-Carlson adder

I. INTRODUCTION

High speed multiplication is a fundamental requirement in many high performance digital systems. Parallel multiplication schemes have been developed for this purpose. There are two classes of parallel multipliers, namely array multipliers and tree multipliers. Tree multipliers, also known as column compression multipliers, are known for their higher speeds making them very useful in high speed computations. Their propagation delay is proportional to the logarithm of the operand word length in comparison to array multipliers whose delay is directly proportional to operand word length [1]. Column compression multipliers are faster than array multipliers but have an irregular structure and so their design is difficult. With the improvement in VLSI design techniques and process technology, designs which were previously infeasible or too difficult to be implemented by manual layout can now be implemented through automated synthesis.

Two of the most well-known column compression multipliers have been presented by Wallace [3] and Dadda [4]. Both architectures are similar with the difference occurring in the procedure of reduction of the partial products and the size of the final adder. In Wallace's scheme, the partial products are reduced as soon as possible. On the other hand, Dadda's method does minimum reduction necessary at each level and requires the same number of levels as Wallace multiplier [5]. This paper presents a C program which generates a Verilog file for a Dadda multiplier of specified size. A comparison of post synthesis and post layout results between Dadda multiplier of varying sizes with Parallel Prefix adders in the final stage is also presented. Sections II and III explain Dadda's algorithm and its implementation in C to create an automatic Verilog file generator. Section IV gives the post synthesis and also post layout results for the multiplier of varying sizes.

II. DADDA MULTIPLIER ARCHITECTURE

The Dadda multiplier architecture can be divided into three stages. The first stage involves generation of partial products by two's complement parallel array multiplication algorithm presented by Baugh-Wooley [2]. In their algorithm, signs of all partial product bits are positive. It's different from conventional two's complement multiplication which generates partial product bits with negative and positive signs. The final product obtained after the reduction is also in two's complement form. Fig.1 shows generation of partial products for 4x4 multiplier by Baugh-Wooley method. Fig. 2(a) shows the arrangement of the partial products for an 8x8 multiplier. The dots represent the partial products.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

$$
\begin{array}{cccccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 & \\
 & & & & \mathrm{x} & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & & & \overline{a_3b_0} & a_2b_0 & a_1b_0 & a_0b_0 & pp0 \\
 & & & & \overline{a_3b_1} & a_2b_1 & a_1b_1 & a_0b_0 & & pp1 \\
 & & & \overline{a_3b_2} & a_2b_2 & a_1b_2 & a_0b_0 & & & pp2 \\
 & & a_3b_3 & \overline{a_2b_3} & \overline{a_1b_3} & \overline{a_0b_0} & & & & pp3 \\
 & 1 & & & & 1 & & & & \\
\hline
\end{array}
$$

Fig. 1 Generation of Partial Products of 4x4 Two's Complement multiplier by Baugh-Wooley's algorithm

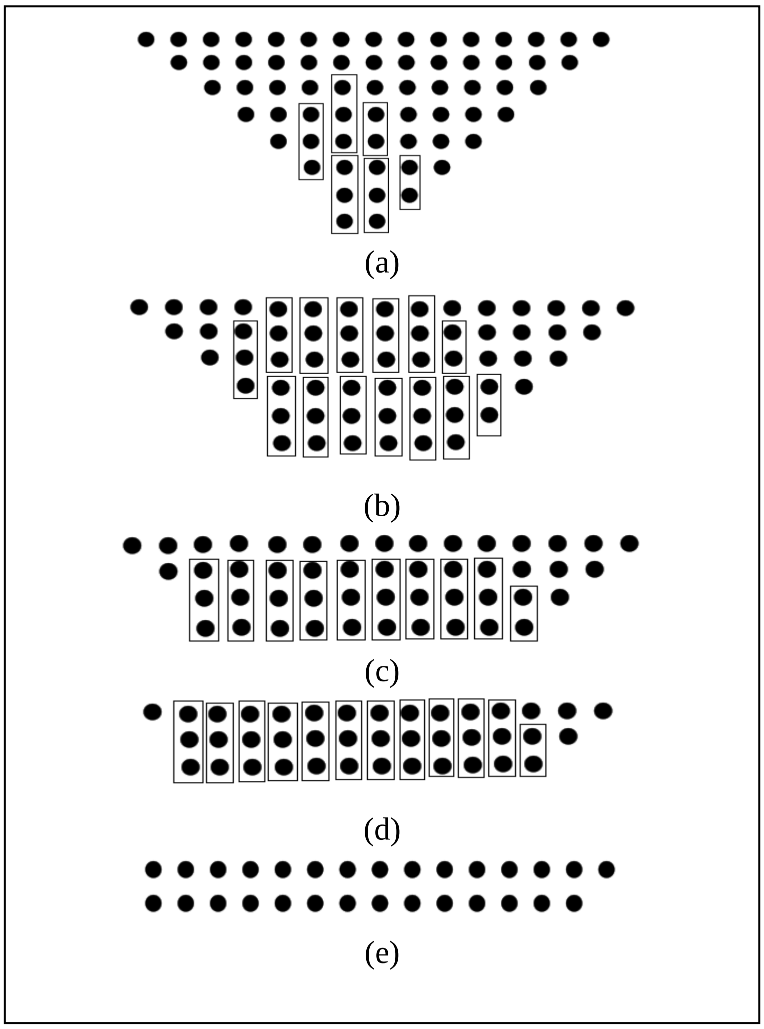

Fig. 2 Column Compression scheme for 8x8 Dadda multiplier

In the second stage, the partial product matrix is reduced to a height of two using the column compression procedure developed by Dadda. The iterative procedure for doing this is as follows:

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

1. Let $h_1 = 2$ and repeat h_{j+1} = floor $(1.5{*}h_j)$ for increasing values of j. Continue this until the largest j is reached, for which there exists at least one column in the present stage of the matrix with more dots than h_j. Using this equation we get h_1=2, h_2=3, h_3=4, h_4=6, h_5=9 and so on. e.g. in the first stage of the 8x8 Dadda multiplication shown in Fig. 2(a), the maximum height of columns is 8 therefore, the value of h_j is 6 means heights of the columns are reduced to a maximum of 6. Similarly in the second stage, shown in Fig. 2(b) the maximum height of column is 6 and value of h_j is 4 heights of the columns are reduced to a maximum of 4.
2. All the columns, with heights greater than h_j, are reduced to a height of h_j using either half adder or full adder. If the column height has to be reduced by one, use a half adder else use a full adder and continue this step till the column height is reduced to h_j.
3. Stop the reduction if the height of the matrix becomes two, after which it can be fed to final adder.

Fig. 2 (b), (c), (d) and (e) show the reduction stages for an 8x8 Dadda multiplier.

Once the height of matrix is reduced to two, a 2N-2 bit adder is used to generate the final product. The paper describes the use of different Parallel Prefix adders for final adder stage which are described later in section III-D.

III. VERILOG CODE GENERATION

The main objective in writing a C program was to output a verilog file for an NxN Dadda multiplier based on the user input N. We implemented the Dadda algorithm and wrote the verilog code to a file using C to achieve this.

A. Initialization of Verilog Modules

Initially, the program prompts the user for the size of the multiplier. After accepting the size, the program creates an empty Verilog file and prints the half adder and full adder modules in it. Next it prints the gate level 'and' primitive for partial product generation. After this the top-level multiplier module is printed using sub-modules generated above.

The top module contains two N-bit 'input' data types, one 2N bit 'output' data type for the multiplier, multiplicand and product bits respectively. The program also sets up N^2 'wire' data type to store the partial products. Once all the above modules and data types are set up in the Verilog file, the program prints the required number of half adder and full adders to reduce column heights as explained earlier in section-II. Column reduction utilizes the dot matrix array in C, which is described below.

B. Dot Matrix Array Creation

For the purpose of implementing the column compression according to Dadda's algorithm, the program creates an equivalent of a dot matrix array internally. The columns of the array are represented by queues (First-In-First-Out data structure). Hence the program creates 2N-1 queues, where each of the queues is implemented as a linked list.

The sizes and pointers to the first and last element of all queues are stored. This allows quicker enqueuing and dequeuing operations. Every element in a queue stores a string of characters, particularly the name of the wires carrying the partial product.

Initially, the queues are loaded with the names of the wires holding the partial products.

C. Array Reduction

Once the array, using the queues, has been created in C, it needs to be reduced according to Dadda's algorithm.

The recursive procedure described in the section-II is implemented in C using a 'while loop'. The size of the queues (height of columns) is reduced by using full adders and half adders. The loop stops executing when the length of all queues (height of all columns) is two or less.

At the start of every iteration, say iteration no. i, the sizes of all the 2N-1 queues are checked and their maximum is stored in L[i]. The value of h_j is also calculated as shown in the 1st part of the iterative procedure defined in section-II. This value of h_j is stored in H[i].

Following the calculation of H[i], the reduction phase is initiated. The sizes of all the queues are checked sequentially starting from Queue 1. If the size of the queue is lesser than or equal to H[i], no changes are made to the queue. If the size of the queue is greater than H[i], the queue needs to be reduced to H[i] using half adders and full adders.

The use of either of these adders depends on the difference etween the length of the queue and H[i]. If the difference is one, half adder is used. The first two elements are dequeued and used as input to the half adder. The sum from the half adder is enqueued to the same queue and the carry out is enqueued to the next queue in sequence. For every half adder used, the size of the queue reduces by one.

If the difference between the length of the queue and H[i] is greater than or equal to two, full adder is used to reduce the queue. When a full adder is used, the first three elements of the queue are dequeued and are supplied as inputs to the full adder. The sum from the full adder is enqueued to the same queue while the carry out is enqueued to the next queue in sequence. For every full adder used, the size of the queue reduces by two. The above procedure is followed recursively till the size of the queue becomes equal to H[i]. After a queue is reduced, the next queue in sequence is taken up for reduction. The carries from previous queues are taken into consideration while checking the length of the queue for reduction. The iterations continue till at most two elements remain in each queue. Once such a state has been reached the 'while' loop exits and the reduction phase is completed.

The array reduction using queues is explained with the help of an example. A 4x4 Dadda multiplication is taken as the example. The state of the queues before the start of the reduction is shown in Fig. 3(a). The reduction of the queues for a 4x4 multiplier is explained below:

1) *Iteration One:* On checking the sizes of all the queues, it is observed that the maximum length (L[1]) for this iteration is 4 and H[1] can be calculated to be 3. Hence all queues with sizes greater than 3 need to be reduced. Sizes of queues 1-3 are lesser than or equal to 3. Since, the length of Queue 4 is greater than the required length by 1, the first two elements of the queue are dequeued. These dequeued elements are summed in a half adder ('ha0') by printing a half adder in the Verilog file. The sum bit of the half adder is assigned to a wire 'ha0s' and the carry out bit of the same half adder is assigned to the wire 'ha0c'. In accordance with the algorithm, 'ha0s' is enqueued to Queue 4 and 'ha0c' is enqueued to Queue 5. Fig. 3(b) shows the elements to be enqueued to Queues 4 and 5. The elements above the arrow in the queue are the ones which are enqueued in this iteration. The addition of one more element to Queue 5 increased the length of the queue to 5. Hence, a full adder ('fa0') is used to reduce its length to 3. The addition of the full adder leads to dequeuing of the first three elements of Queue 5 and enqueuing of the sum of the full adder'fa0s' to Queue 5 and the carry out of the full adder 'fa0c' to Queue 6. This iteration is now complete because the length of all queues is less than or equal to three. The state of the queues at the end of stage one is shown in Fig. 3(c).

2) *Iteration Two*: On checking the sizes of all the queues, it is observed that the maximum length (L[2]) for the second iteration is 3 and H[2] can be calculated to be 2. Hence all queues with sizes greater than 2 need to be reduced. The program starts checking the sizes of the queues sequentially. Queue 3 is the first queue with length greater than 2. Since the length is only exceeded by one, a half adder ('ha1') is used to reduce the length to 2. The first two elements (enclosed by the square) are dequeued as input to half adder and 'ha1s' is enqueued to Queue 3 with 'ha1c' being enqueued to Queue 4. This increases the length of Queue 4 to four which is two more than the allowed size. Hence, a full adder ('fa1') is used to reduce its length to 2. The addition of the full adder leads to dequeuing of the first three elements of Queue 4 and enqueuing of the sum of the full adder'fa1s' to Queue 4 and the carry out of the full adder 'fa1c' to Queue 5. This increases the size of Queue 5 to four. It undergoes the same procedure as Queue 4. Queue 6 also faces the same situation and undergoes the same procedure outlined above. Fig. 3(d) shows the elements dequeued and enqueued to Queues 3, 4, 5 and 6. The final state of the queues at the end of stage two is shown in Fig. 3(e).

D. Final Stage Adder

Once the size of all qeues has been reduced to two or less, the elements in the queues are ready to be summed using an adder. The first elements of all queues form the first input to the adder and the second elements form the second input to the adder. The size of the adder has to be *2N-2* bit for an NxN multiplier. Different Parallel Prefix adders like Kogge-Stone adder, Brent-Kung adder and Han-Carlson adder are used. The implementation of these adders is described below.

1) *Kogge-Stone Adder:* The Kogge-Stone adder generates carry signal in O(log n) time [6]. A radix-2 Kogge-Stone adder of size *2N-2* has been used as the final adder for an NxN multiplier. Fig. 4 shows the example of an 8-bit Kogge-Stone adder with no carry-in. The first bit in the boxes is the propagate bit while the second one is the generate bit. Initially (stage zero) in an N-bit Kogge-Stone adder, the propagate and generate bits are generated according to (1).

$$G_{0,n} = a_n b_n \quad \text{for } 1 \leq n \leq N$$

$$P_{0,n} = a_n + b_n \quad \text{for } 1 \leq n \leq N \qquad (1)$$

where a_n and b_n are the nth bits of the two inputs.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

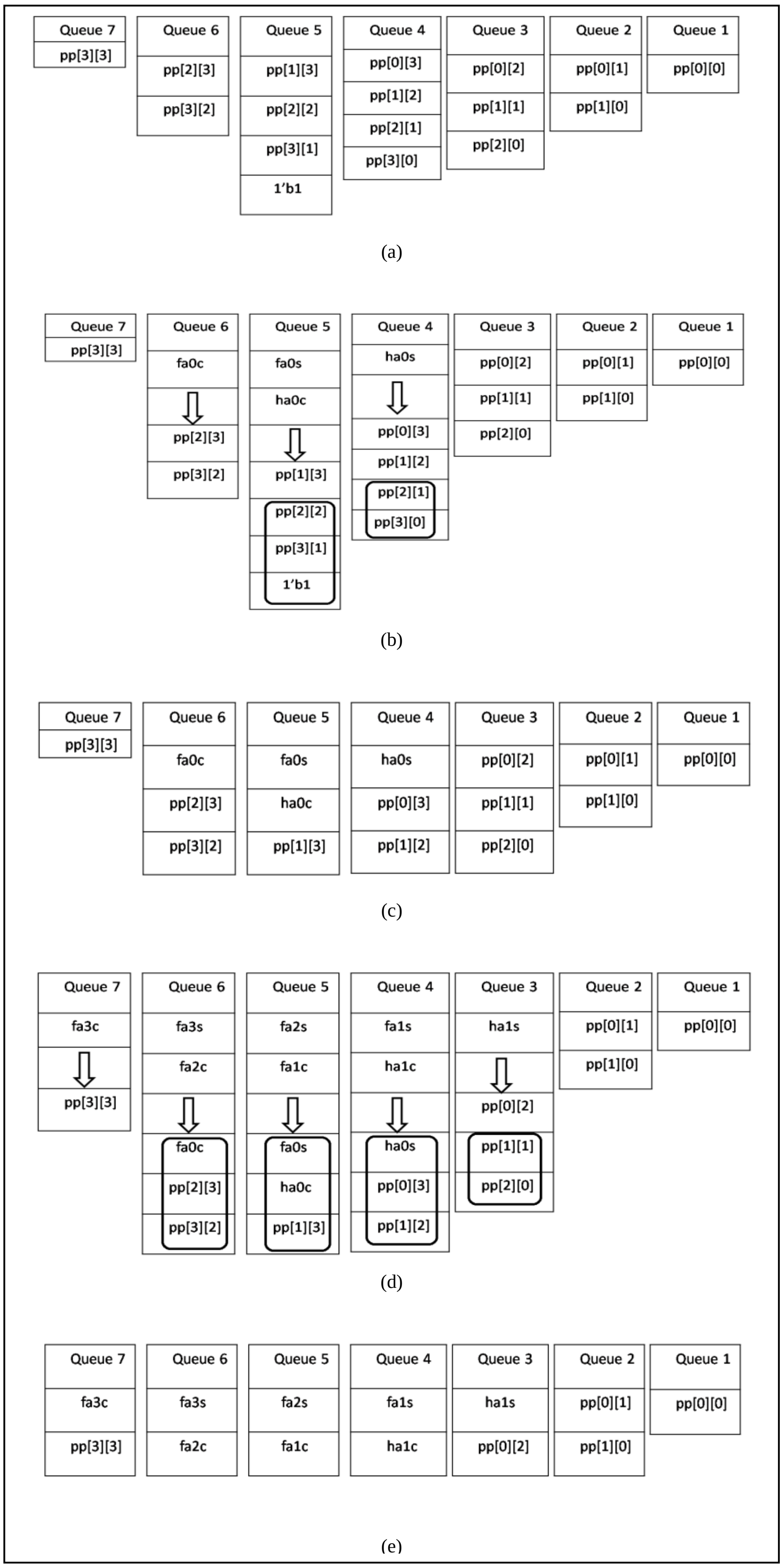

Fig. 3 Reduction of Queues for a 4x4 Dadda multiplier

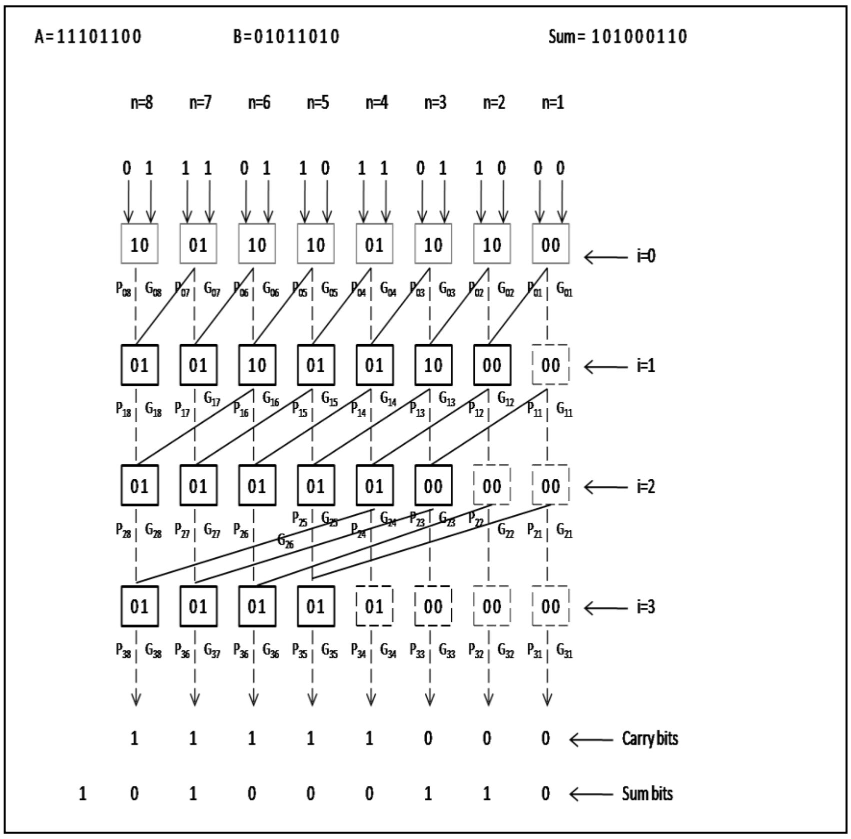

Fig. 4 Example of 8 bit Kogge-Stone adder

In the ith ($i \geq 1$) stage the propagate and generate bits in nth block ($P_{i,n}$ and $G_{i,n}$) are calculated according to (2).

$$G_{i,n} = G_{i-1,n} \qquad \text{for } 1 \leq n \leq 2^{i-1} + 1$$

$$P_{i,n} = P_{i-1,n} \qquad \text{for } 1 \leq n \leq 2^{i-1} + 1$$

$$G_{i,n} = G_{i-1,n} + (P_{i-1,n}\, G_{i-1,m}) \qquad \text{for } 2^{i-1} + 2 \leq n \leq N$$

$$P_{i,n} = P_{i-1,n}\, P_{i-1,m} \qquad \text{for } 2^{i-1} + 2 \leq n \leq N \qquad (2)$$

where m is given by (3)

$$m = n - 2^{i-1} \qquad (3)$$

Finally, the carry and sum bits are calculated according to (4).

$$C_n = G_{f,n}$$

$$S_n = P_{0,n} \oplus C_{n-1} \qquad (4)$$

where S_n and C_n are the nth bits of the sum and carry respectively. $G_{f,n}$ is the generate bit of the nth block in the final stage.

The numbers of stages in the adder depend upon its size. In a Kogge-Stone adder of size N, there are $ceil(log_2 N)+1$ stages. Since we need a $2N$-2 bit adder for the final stage of an NxN multiplier, the number of stages in the adder are $ceil[log_2(2N-2)]+1$. The logarithmic delay of Kogge-Stone adder is crucial in maintaining overall logarithmic delay of the multiplier.

2) *Brent-Kung Adder:* Kogge-Stone Adder has higher performance but it takes larger area to be laid out. Brent-Kung adder can be laid out in lesser area than Kogge-Stone Adder and has lesser wiring congestion, but its time performance is poor[7]. In a Brent-Kung adder of size N, there are $Ceil[2*log_2(N)]$ stages. Initially (stage zero) in an N-bit Brent-Kung adder, the propagate and generate bits are generated according to (5).

$$G_{0,n} = a_n\, b_n \qquad \text{for } 1 \leq n \leq N$$

$$P_{0,n} = a_n + b_n \qquad \text{for } 1 \leq n \leq N \qquad (5)$$

where a_n and b_n are the nth bits of the two inputs

In the ith ($1 \leq i \leq log_2(N)$) stage the propagate and generate bits in nth block ($P_{i,n}$ and $G_{i,n}$) are calculated according to (6).

$$\begin{aligned}
&G_{i,n} = G_{i-1,n} + (P_{i-1,n}\, G_{i-1,m}) && \text{for } n=2^i.k \;\; \forall\, k \in Z^+ \;\&\; n \leq N \\
&P_{i,n} = P_{i-1,n} P_{i-1,m} && \text{for } n=2^i.k \;\; \forall k \in Z^+ \;\&\; n \leq N \\
&G_{i,n} = G_{i-1,n} && \text{for } 1 \leq n \leq N \;\&\; n!=2^i.k \;\; \forall\, k \in Z^+ \\
&P_{i,n} = P_{i-1,n} && \text{for } 1 \leq n \leq N \;\&\; n!=2^i.k \;\; \forall\, k \in Z^+ \qquad (6)
\end{aligned}$$

where m is given by (7)

$$m = n - 2^{i-1} \qquad (7)$$

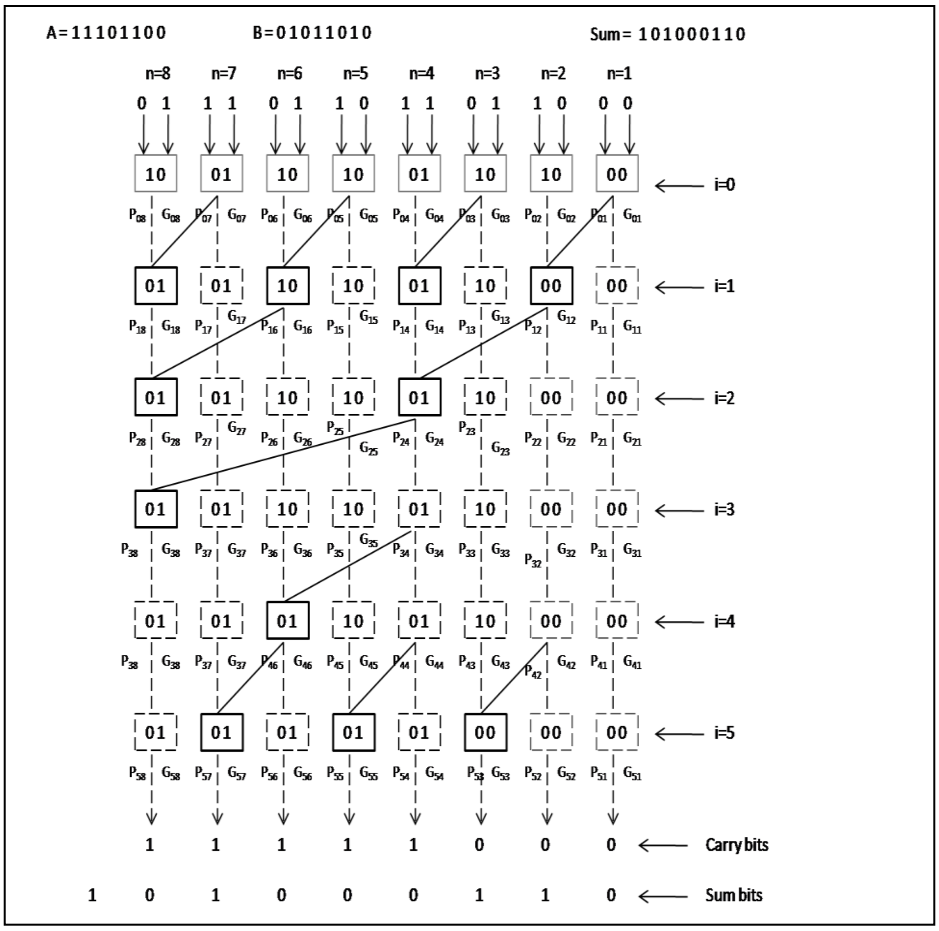

Fig. 5 Example of 8 bit Brent-Kung adder

To calculate generate and propagate bits of ith stage ($log_2(N)+1 \leq i < 2{*}log_2(N)$), we define three variables *count*, *x* and *y*.

count $=k$ where 2^k is the smallest integer $\geq N$ and $x \in Z^+$, $x \geq 2$ and $x=2$ *for* $i= log_2(N)+1$ and is incremented by 1 for every subsequent stage and

$$\begin{aligned}
&z=2^{count} - 2^{count-2} \\
&y= 2^{count} - 2^{count-x} \qquad (8)
\end{aligned}$$

Now in the ith stage the propagate and generate bits in nth block ($P_{i,n}$ and $G_{i,n}$) are calculated according to

$$G_{i,n} = G_{i-1,n} + (P_{i-1,n}\, G_{i-1,m}) \text{ for } n=y\text{-}l \text{ and } N \geq n \geq z/2$$

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

$P_{i,n} = P_{i-1,n} P_{i-1,m}$ for $n=y-l$ and $N \geq n \geq z/2$

$G_{i,n} = G_{i-1,n}$ for $1 \leq n \leq N$ and $n!=y-l$

$P_{i,n} = P_{i-1,n}$ for $1 \leq n \leq N$ and $n!=y-l$ (9)

where *m* and *l* are given by (10) and (11) respectively

$m= n - 2^{count-x}$ (10)

$l= (2^{count} - 2^{count-x+1}).k, \forall k \in Z^{+} U \{0\}$ (11)

Finally, the carry and sum bits are calculated according to (12).

$C_n = G_{f,n}$

$S_n = P_{0,n} \oplus C_{n-1}$ (12)

where S_n and C_n are the nth bits of the sum and carry respectively. $G_{f,n}$ is the generate bit of the nth block in the final stage.

Fig. 5 shows the example of an 8 bit Brent-Kung Adder. Since we need a 2N-2 bit adder for the final stage of an NxN multiplier, the number of stages in the adder are *ceil[2*log₂(2N-2)]*.

3) *Han-Carlson Adder:* Brent-Kung adder reduces area and power but do not produce minimum depth parallel prefix circuits. Their delay time is also high $(2*log_2N - 1)$. Kogge-Stone adder has lesser delay (log_2N) but it has high area and power. By combining B-K & K-S graphs, Han and Carlson obtained a new hybrid prefix graph that achieves intermediate values of area and time[8]. An example of 8 bit Han-Carlson Adder is shown in Fig. 6. The numbers of stages in the adder depend upon its size. In a Han-Carlson adder of size N, there are $Ceil[log_2(N)+2]$ stages.

Initially (stage zero) in an N-bit Han-Carlson adder, the propagate and generate bits are generated according to (13).

$G_{0,n} = a_n b_n$ *for* $1 \leq n \leq N$

$P_{0,n} = a_n + b_n$ *for* $1 \leq n \leq N$ *(13)*

where a_n and b_n are the nth bits of the two inputs

In the ith $(1 \leq i \leq log_2(N))$ stage the propagate and generate bits in nth block ($P_{i,n}$ and $G_{i,n}$) are calculated according to (14)

$G_{i,n} = G_{i-1,n} + (P_{i-1,n} G_{i-1,m})$ *for* $2^{i-1}+2 \leq n \leq N$, *n is even*

$P_{i,n} = P_{i-1,n} P_{i-1,m}$ *for* $2^{i-1}+2 \leq n \leq N$, *n is even*

Let $S=\{n \mid 2^{i-1}+2 \leq n \leq N, n \text{ is even}\}$

$G_{i,n} = G_{i-1,n}$ *for* $1 \leq n \leq N, \forall n \notin S$

$P_{i,n} = P_{i-1,n}$ *for* $1 \leq n \leq N, \forall n \notin S$ (14)

where $m = n - 2^{i-1}$ (15)

In the last stage($i=log_2(N)+1$) the propagate and generate bits in nth block ($P_{i,n}$ and $G_{i,n}$) are calculated according to (16)

$G_{i,n} = G_{i-1,n} + (P_{i-1,n} G_{i-1,m})$ for $3 \leq n \leq N$, *n is odd*

$P_{i,n} = P_{i-1,n} P_{i-1,m}$ for $3 \leq n \leq N$, *n is odd*

Let $R=\{n \mid 3 \leq n \leq N, n \text{ is odd}\}$

$G_{i,n} = G_{i-1,n}$ for $1 \leq n \leq N, n \notin R$

$P_{i,n} = P_{i-1,n}$ for $1 \leq n \leq N, n \notin R$ (16)

Finally, the carry and sum bits are calculated according to (17).

$C_n = G_{f,n}$

$S_n = P_{0,n} \oplus C_{n-1}$ (17)

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

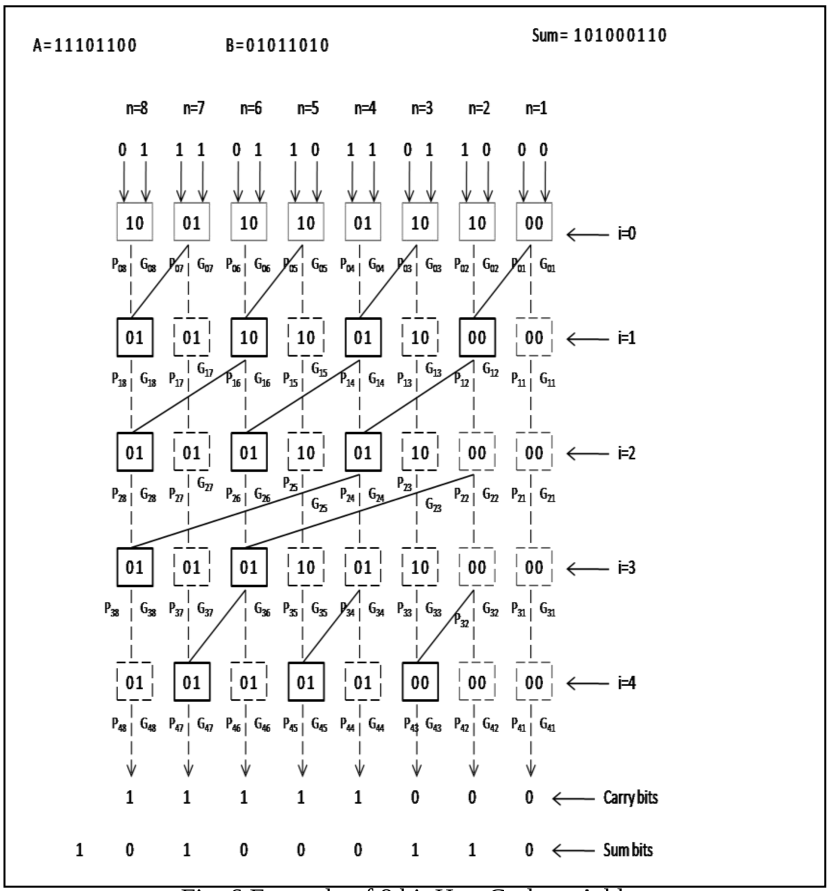

Fig. 6 Example of 8 bit Han-Carlson Adder

where S_n and C_n are the nth bits of the sum and carry respectively. $G_{f,n}$ is the generate bit of the nth block in the final stage.

Since we need a 2N-2 bit adder for the final stage of an NxN multiplier, the number of stages in the adder is given by *ceil[log*$_2$*(2N-2)+2]*.

IV. RESULTS AND DISCUSSION

All the multiplier designs use Verilog as the HDL. The synthesis and post layout results of Dadda multiplier with all the parallel prefix adders discussed in the previous section were compared. This section provides the delay in millisecond, area in square micrometer and power in milliwatt for all the architectures mentioned above. The synthesis was performed in Cadence RTL Compiler using 90 nm UMC technology libraries at typical (tt) conditions. The synthesized netlist was used along with design constraint .sdc file, technology library file and .lef files for generating the layout using SOC Encounter tool. The layouts have been done for Dadda multiplier of different sizes and their post layout results have been tabulated. The post layout results follow the same trend as the post synthesis results.

TABLE I. COMPARISON OF SYNTHESIS AND POST LAYOUT RESULTS FOR DADDA MULTIPLIER WITH KOGGE-STONE ADDER

Size	Synthesis (Pre Layout)		Post Layout	
	Delay (ns)	Area (μm^2)	Delay(ns)	Area(um^2)
16	4.772	9013	4.898	21382
32	6.201	36613	6.466	86858
48	7.202	82801	7.842	196431
64	8.191	146291	8.863	347052
96	10.220	329714	12.633	782193

TABLE II. COMPARISON OF SYNTHESIS AND POST LAYOUT RESULTS FOR DADDA MULTIPLIER WITH BRENT-KUNG ADDER

Size	Synthesis (Pre Layout)		Post Layout	
	Delay (ns)	Area (μm^2)	Delay(ns)	Area(um^2)
16	5.656	8636	5.676	19877
32	9.071	34798	9.612	82552
48	12.149	79141	12.341	187749
64	14.552	141519	14.983	335731
96	18.890	320028	19.507	759214

TABLE III. COMPARISON OF SYNTHESIS AND POST LAYOUT RESULTS FOR DADDA MULTIPLIER WITH HAN-CARLSON ADDER

Size	Synthesis (Pre Layout)		Post Layout	
	Delay (ns)	Area (μm^2)	Delay(ns)	Area(um^2)
16	5.018	8379	5.114	20487
32	6.367	35479	6.685	84168
48	7.388	80622	8.397	132736
64	8.730	14331	10.637	340028
96	10.541	324443	15.959	769686

TABLE IV. COMPARISON OF POST LAYOUT RESULTS FOR DADDA MULTIPLIER WITH PARALLEL PREFIX ADDERS

Size	Kogge-Stone		Brent-Kung		Han-Carlson	
	Power (mW)	PDP	Power (mW)	PDP	Power (mW)	PDP
16	0.401	1.964	0.336	1.907	0.345	1.764
32	1.089	7.041	0.867	8.334	0.967	6.464
48	1.937	15.189	1.263	15.586	1.446	12.142
64	2.955	26.190	2.041	30.580	2.528	26.890
96	5.746	72.589	4.686	91.41	5.064	80.816

The post layout area is the area of core which includes area of standarad cells as well as the area of interconnect wires. Fig.7 shows that the multiplier with Kogge-Stone adder in the final stage is much faster than with all other adders but it's power consumption is the highest of all as shown in Fig. 8. Fig.9 compares the Power Delay Products.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

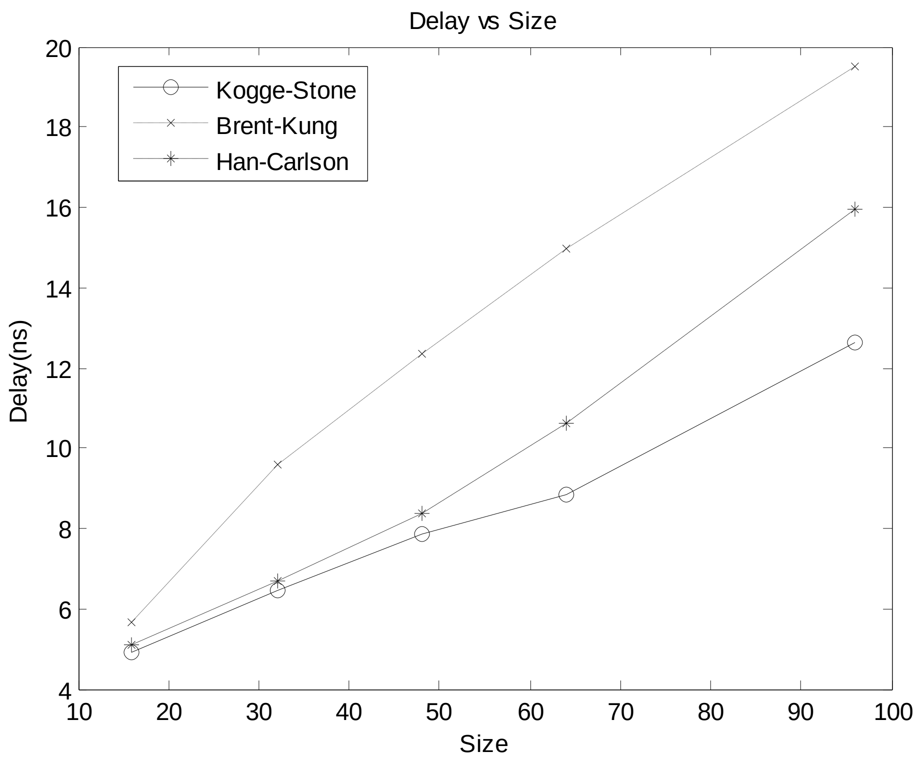

Fig. 7 Post Layout Delay Comparison

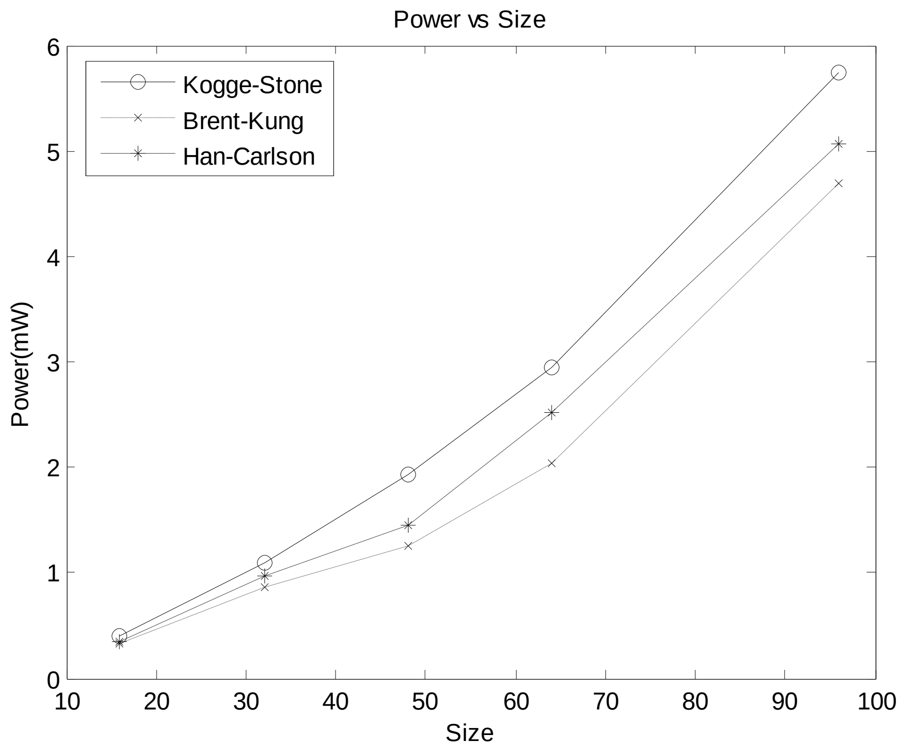

Fig. 8 Post Layout Power Comparison.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 6, June 2013

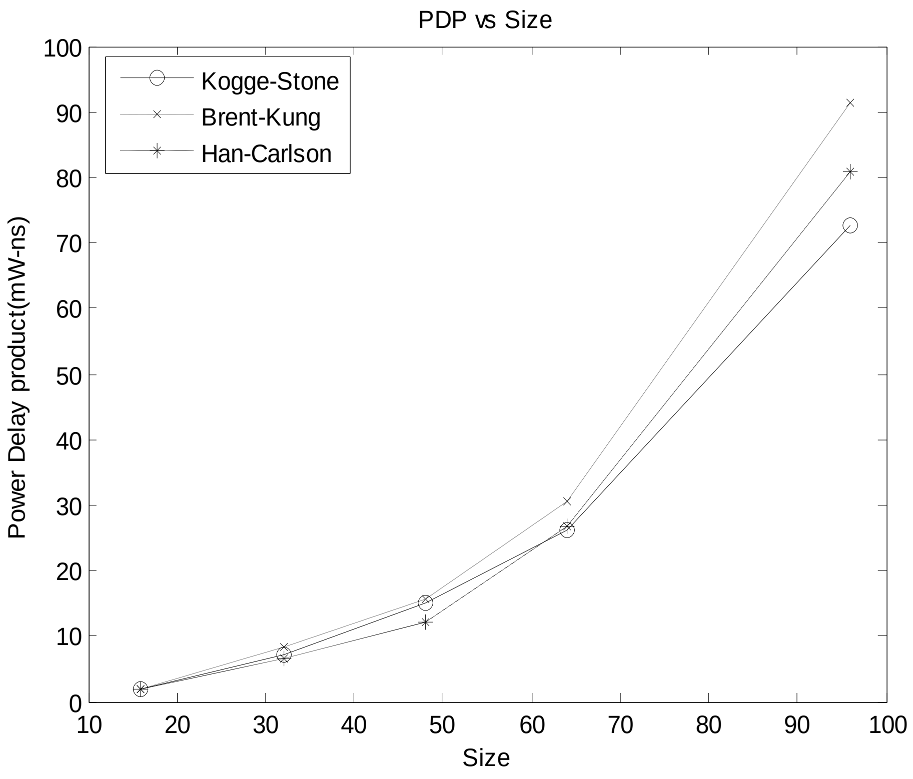

Fig. 9 Power Delay Product comparison

It can be observed that Dadda multiplier with Kogge-Stone and Han-Carlson adders in the final stage has lower Power Delay Products than with Brent-Kung adder. Dadda multiplier with Brent-Kung adder gives lower power and area when compared to that with Kogge-Stone and Han-Carlson adders but owing to the higher delay, their Power Delay Product is also higher.

V. CONCLUSION

This paper explains an easy and efficient method of generating synthesizable Verilog code for Dadda multipliers of user specified size. It also shows that the use of Parallel Prefix adders in the final stage greatly improves the speed of Dadda multiplier and also gives lower Power Delay Products. The logarithmic delay of the Parallel Prefix adders supplements the logarithmic delay of the compression tree to provide an overall logarithmic delay.

REFERENCES

[1] P. R. Cappello and K Steiglitz, “A VLSI layout for a pipe-lined Dadda multiplier,” *ACM Transactions on Computer Systems 1,2(May 1983)* , pp. 157-17
[2] Baugh, Charles R.; Wooley, B.A., "A Two's Complement Parallel Array Multiplication Algorithm," *Computers, IEEE Transactions on* , vol.C-22, no.12, pp.1045,1047, Dec. 1973
[3] Wallace, C. S., "A Suggestion for a Fast Multiplier," *Electronic Computers, IEEE Transactions on* , vol.EC-13, no.1, pp.14,17, Feb. 1964
[4] L. Dadda, “Some schemes for parallel multipliers,” *Alta Frequenza*, vol. 34, pp. 349–356, 1965
[5] Townsend, W. Swartzlander, E. Abraham, J., "A Comparison of Dadda and Wallace Multiplier Delays". SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations XIII.
[6] Kogge, Peter M.; Stone, Harold S., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *Computers, IEEE Transactions on* , vol.C-22, no.8, pp.786,793, Aug. 1973
[7] Brent, Richard P.; Kung, H. T., "A Regular Layout for Parallel Adders," *Computers, IEEE Transactions on* , vol.C-31, no.3, pp.260,264, March 1982
[8] Han, Tackdon; Carlson, D.A., "Fast area-efficient VLSI adders," *Computer Arithmetic (ARITH), 1987 IEEE 8th Symposium on* , vol., no., pp.49,56, 18-21 May 1987