

Copyright

by

Aditi Ranganath

2017

**The Report Committee for Aditi Ranganath  
Certifies that this is the approved version of the following report:**

**A Comparative Study of Novel Variable Set Multiplier Scheme and  
other Column Compression Multipliers**

**APPROVED BY  
SUPERVISING COMMITTEE:**

**Supervisor:**

---

EARL E. SWARTZLANDER, Jr.

---

LIZY KURIAN JOHN

**A Comparative Study of Novel Variable Set Multiplier Scheme and  
other Column Compression Multipliers**

**by**

**Aditi Ranganath, B.E.**

**Report**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**May 2017**

## **Abstract**

# **A Comparative Study of Novel Variable Set Multiplier Scheme and other Column Compression Multipliers**

Aditi Ranganath, M.S.E.

The University of Texas at Austin, 2017

Supervisor: Earl E. Swartzlander, Jr.

The extensive use of multipliers in high-performance processors has been driving the demand for low-power, high-speed multipliers. This report presents a novel fast multiplier scheme that performs reduction based on non-uniform grouping in the partial product matrix. In every stage, the partial products are grouped into sets of variable heights before reduction. The algorithm is presented and compared with Wallace, Dadda, and Reduced Area Multipliers in terms of gate complexity, delay, interconnects and power utilization for 4-bit, 7-bit, 8-bit and 12-bit versions. The proposed design is observed to have a gate count of 3% lower than Wallace multipliers, comparable to Dadda in most of the cases but slightly higher than Reduced Area multipliers. To fulfill the study, a 12-bit version of each multiplier was implemented in structural Verilog with basic gates and synthesized using Xilinx Vivado. The net cell usage statistics and the power utilization values were extracted and compared. It was observed that the proposed design has better power efficiency than the other multipliers for the examined cases.

## Table of Contents

List of Tables .....	vi
List of Figures .....	vii
<b>1. INTRODUCTION .....</b>	<b>1</b>
<b>2. PARALLEL MULTIPLIER REDUCTION SCHEMES .....</b>	<b>3</b>
2.1 ARRAY MULTIPLIERS .....	3
2.2 COLUMN COMPRESSION MULTIPLIERS .....	4
2.2.1 WALLACE MULTIPLIER .....	6
2.2.2 DADDA MULTIPLIER .....	8
2.2.3 REDUCED AREA MULTIPLIER .....	9
2.2.4 VARIABLE SET MULTIPLIER .....	11
<b>3. COLUMN COMPRESSION MULTIPLIER COMPARISONS .....</b>	<b>15</b>
3.1 GATE COMPLEXITY .....	16
3.2 GATE DELAY .....	18
3.3 POST IMPLEMENTATION CELL USAGE.....	19
3.4 POWER UTILIZATION .....	20
3.5 ROUTING STATUS: INTERCONNECTS .....	22
3.6 SCHEMATIC AND POST IMPLEMENTATION LAYOUTS.....	22
3.6.1 WALLACE MULTIPLIER .....	22
3.6.2 DADDA MULTIPLIER .....	24
3.6.3 REDUCED AREA MULTIPLIER .....	24
3.6.4 VARIABLE SET MULTIPLIER .....	26
<b>4. CONCLUSION .....</b>	<b>28</b>
Appendix .....	29
References .....	39

## **List of Tables**

Table 1: Component Complexity in parallel multipliers.....	16
Table 2: Total gate count using Carry Lookahead Adders. ....	17
Table 3: Total gate count using Ripple Carry Adders. ....	18
Table 4: Summary of Cell Usage Report. ....	20
Table 5: Power Utilization Report. ....	21

## List of Figures

Figure 1: 4x4 Array Multiplier. ....	4
Figure 2: Column Compression Scheme [9]. ....	5
Figure 3: 12x12 Wallace Multiplier [9]. ....	7
Figure 4: 12x12 Dadda Multiplier [9]. ....	9
Figure 5: 12x12 Reduced Area Multiplier [9]. ....	11
Figure 6: 12x12 Variable Set Multiplier. ....	13
Figure 7: (A) Total number of (3,2) counters used in each multiplier block; (B) Total number of (2,2) counters used in each multiplier block; (C) Width of the Carry Propagate Adder to be used in the final stage in each multiplier. ....	17
Figure 8: Total Gate Complexity with (A) Carry Lookahead Adder; (B) Ripple Carry Adder ....	18
Figure 9: Abstract Gate Delays using RCA as CPA. ....	19
Figure 10: Cell Usage Report from Vivado. ....	20
Figure 11: On-Chip Power Distribution Report. ....	21
Figure 12: Routing Complexity. ....	22
Figure 13: Schematic of 12x12 Wallace Multiplier. ....	23
Figure 14: Layout of 12x12 Wallace Multiplier. ....	23
Figure 15: Schematic of 12x12 Dadda Multiplier. ....	24
Figure 16: Layout of 12x12 Dadda Multiplier. ....	25
Figure 17: Schematic of 12x12 Reduced Area Multiplier. ....	25
Figure 18: Layout of 12x12 Reduced Area Multiplier. ....	26

Figure 19: Schematic of 12x12 Variable Set Multiplier.....	27
Figure 20: Layout of 12x12 Variable Set Multiplier. ....	27

# 1. INTRODUCTION

Microprocessors are the modern building blocks of the information world. In the past, their performance has grown drastically due to the advances in scaling technology and microarchitectural developments that exploited Moore's law. The future of these processors are heavily dependent on their energy utilization levels and computation speed thereby creating a demand for high-performance, low-speed processors. An important requirement for building such processors is the use of high speed multiplier blocks. The exhaustive usage of these blocks drives its design towards low-latency and high-performance speeds. In addition to this, advances in technology has propelled the design targets of these blocks to consider the regularity of layout in order to optimize it for area.

Over the last few years, there has been a paradigm shift in multiplier block design from using slower, add-and-shift multiplier algorithms to using faster, parallel multiplier schemes. The two well-known classes of parallel multipliers are the array multipliers [1] and the column compression multipliers [2]. Array multipliers are better suited for designs that need low-power and smaller area. These multipliers can be easily pipelined and are now tending towards power optimal designs. On the other hand, the column compression multipliers are more applicable for high-speed designs. They are proven to be much faster than the array multipliers as their total delay measurements vary logarithmically with the operand sizes unlike array multipliers that scale linearly. Majority of the work on column compression multipliers are centered around improving their speed due to which other factors such as area optimization, interconnect complexity and power efficiency merit further attention. A few of the well-known column

compression based fast multiplier schemes are the Wallace [3], Dadda [4] and Reduced Area [5] multipliers. All these multiplier schemes consist of three general steps:

- Step 1: Compute the partial products in parallel
- Step 2: Reduce height of the partial product matrix to two rows using parallel (3,2) and (2,2) counters
- Step 3: Use carry propagate adder to add the two rows and obtain the sum.

Other implementations of fast multipliers include the one by Wesley Chu et al [6] where carry lookahead adders (CLA) were used in place of individual full adders. Although using CLAs reduces the number of stages and hence the computation delay, an increase in complexity of the multiplier block was observed.

This report presents a novel fast multiplier scheme that performs reduction based on non-uniform grouping in the partial product matrix. In every stage, the partial products are grouped into sets whose heights are varied based on the overall stage height prior reduction. The algorithm is also compared with the existing Wallace, Dadda, and Reduced Area multiplier schemes. The proposed design is observed to have a gate count of 3% lower than Wallace multipliers, comparable to Dadda in most of the cases but slightly higher than Reduced Area multipliers in complexity.

Chapter 2 presents various Parallel Multiplier Schemes and examples for the same. Further, in Chapter 3, all the simulation and synthesis results are discussed and compared across the variants of column compression multipliers. Chapter 4 summarizes and concludes the report. It is followed by an Appendix with source code for the proposed Variable Set Multiplier.

## 2. PARALLEL MULTIPLIER REDUCTION SCHEMES

In the early days, multiplier block logic was based off parallel adders and registers that performed a series of shifts and additions. The multiplier design improvised over time with advances in technology and algorithms. From parallel adders to Booth multipliers [7], the design has now evolved into parallel multiplier schemes. Based on the multiplication scheme used, the parallel multipliers are broadly classified into two classes – the array multipliers and the column compression multipliers. In this section we briefly discuss about each of these classes and the corresponding multiplication schemes.

### 2.1 ARRAY MULTIPLIERS

The first class of parallel multiplier is known as iterative array multipliers, or just, array multipliers [8]. It uses a regular array structure comprised of a rectangular array of identical combinatorial cells to generate the partial products and their sum. This makes the design of array multiplier easy to layout and also contributes to its throughput making it a very popular multiplier scheme. *Figure 1* shows a 4x4 array multiplier where the partial products of the operands form a matrix layout. Each partial product is called a summand and can be easily generated in parallel using a series of AND gates. The input columns with only two summands uses a half adder while the rest use the full adders. Although the regularity in its structure eases the implementation of its layout, the speed of the array multipliers is not competitive. The delay of these multipliers is generally proportional to the word length of the input operands.

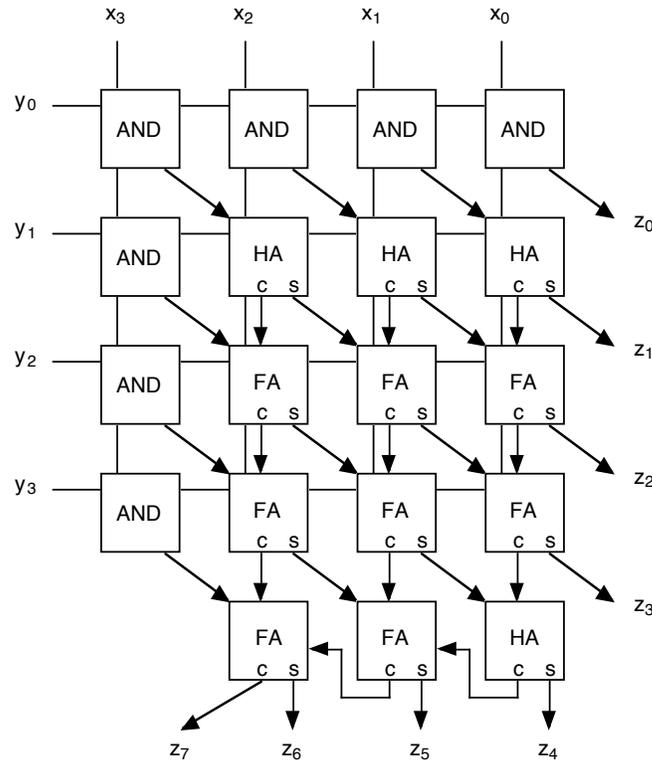


Figure 1: 4x4 Array Multiplier.

## 2.2 COLUMN COMPRESSION MULTIPLIERS

The idea of column compression evolved from using adders to add two numbers together. Wallace [3] first introduced a scheme that uses addition of three numbers to produce two results having different weights, called sum and carry. He called these (3,2) counters as “pseudo-adders” [9]. When a (3,2) counter is used, three input bits are processed from a column and two output bits are produced, a sum bit that remains in the same column and a carry bit that is entered in the next significant column. Likewise, a (2,2) counter accepts two input bits from a column and produces a sum bit in the same column and a carry bit in the next column. He then proposed a scheme to reduce the number of partial products generated, which was later refined by Dadda [4]. Similarly,

improvements on strategic use of the (3,2) and (2,2) counters resulted in the development of the Reduced Area multiplier scheme [5]. In each of these cases, the reduction of partial products is performed to accelerate the addition of the summands. The parallel application of these counters to compress the column height of the partial products makes the multiplication process faster than the array multipliers. With column compression algorithms, the total delay is proportional to the logarithm of the operand word length as against array multipliers where delay grows linearly with word length. Once the partial product matrix is reduced to a height of two, a carry propagate adder is used to compute the product by adding the final summands. Hence the name column compression algorithms. *Figure 2* shows the generic algorithm for all column compression schemes.

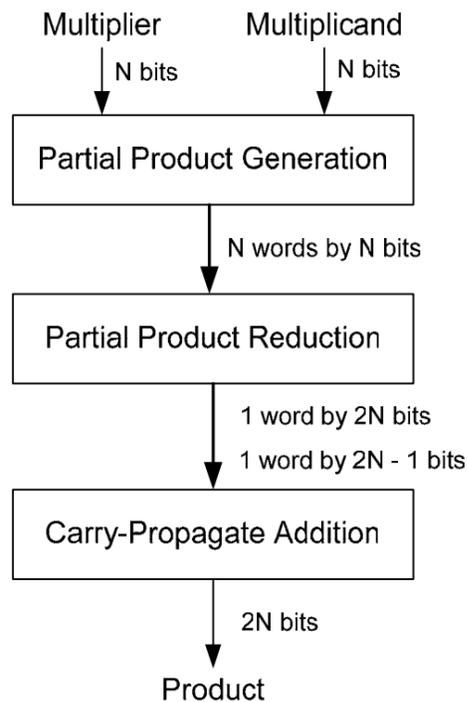


Figure 2: Column Compression Scheme [9].

In this section we discuss 4 column compression schemes:

- Wallace multiplier scheme
- Dadda multiplier scheme
- Reduced Area multiplier scheme
- Variable Set multiplier scheme – novel scheme proposed in this report

Dot diagrams are used to illustrate examples of each of these multiplier schemes in which each partial product is represented by a dot. The output of each (3,2) counter is represented as a plain diagonal line connecting its sum and carry dots. The output of a (2,2) counter is represented by a crossed diagonal line connecting its sum and carry bits.

### **2.2.1 WALLACE MULTIPLIER**

This method, proposed by Wallace in 1964, is a three step process used to multiply two operands as described below:

- Step 1: The partial product bit matrix formation: each bit of the first operand is AND-ed with every bit in the second operand
- Step 2: Reducing the bit product matrix to a two row matrix. This is performed using carry save adders, which is popularly known as the Wallace Tree.
- Step 3: Using a Fast Carry Propagate Adder (CPA) to sum the two rows in order to obtain the product.

In each stage of reduction, there is a preliminary grouping of the partial product rows into sets of three. The (3,2) and (2,2) counters are then used simultaneously within each three row set to reduce it to a two row set. This process repeats until the final column height in the partial product matrix is reduced to two. It can be observed that the Wallace tree multiplier uses a lot of hardware (counters) in the early few stages to reduce

the partial product matrix as quickly as possible. Overall, the time required to compute the product varies logarithmically with the size of the operands.

The dot diagram for a 12x12 Wallace multiplier is shown in *Figure 3*. The Stage-1 of the reduction has four sets of three rows each due to the preliminary grouping of partial products. In case the number of rows is not divisible by three, the additional rows are carried over to the next stage without any reduction. On repeating this operation in the subsequent stages, the partial product is reduced to a height of two rows in Stage 5. In this stage, 18-bit carry propagate adder is used to compute the final product by adding the two partial product rows. All in all, a standard 12x12 Wallace multiplier requires 102 (3,2) counters, 34 (2,2) counters, and an 18 bit carry-propagate adder.

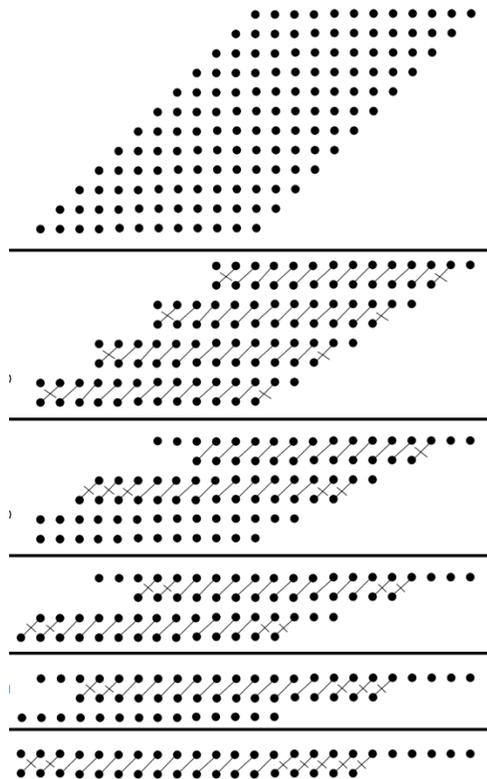


Figure 3: 12x12 Wallace Multiplier [9].

### 2.2.2 DADDA MULTIPLIER

At an abstract level, the Dadda multiplier also has similar steps as that of Wallace:

- Step 1: Partial product formation
- Step 2: Use counters to reduce the height of the partial product matrix to two rows
- Step 3: Add the last two rows with a carry propagation adder to obtain the product.

However, Dadda conservatively chooses the usage of (3,2) and (2,2) counters depending on the reduction stage and height of partial product matrix. Dadda proposed a stage height requirement in the reduction process according to which each stage height should be no greater than the integer portion of 1.5 times the next stage height. The 1.5 times is due to the usage of (3,2) counters, which inputs 3 partial product bits and reduces it to 2 bits giving a 1.5:1 ratio. With this rule, the stage heights in Dadda reduction are: 2, 3, 4, 6, 9, 13, 19, and so on. The delay for partial product reduction is a logarithmic function of the input operand size. Further, the CPA also has a logarithmic delay with respect to the word size.

*Figure 4* illustrates a 12x12 Dadda multiplier. It can be observed that Dadda multiplier performs fewer reductions, thereby uses fewer adders in comparison to Wallace multipliers. Thus, Dadda multipliers have a less expensive reduction phase. However, they require a slightly bigger adder in the CPA stage. Also, since Dadda multipliers have less regular structure than Wallace multipliers, it is more challenging to layout Dadda multipliers in design. With this scheme, a standard 12x12 Dadda multiplier requires 99 (3,2) counters, 11 (2,2) counters and a 22-bit carry-propagate adder.

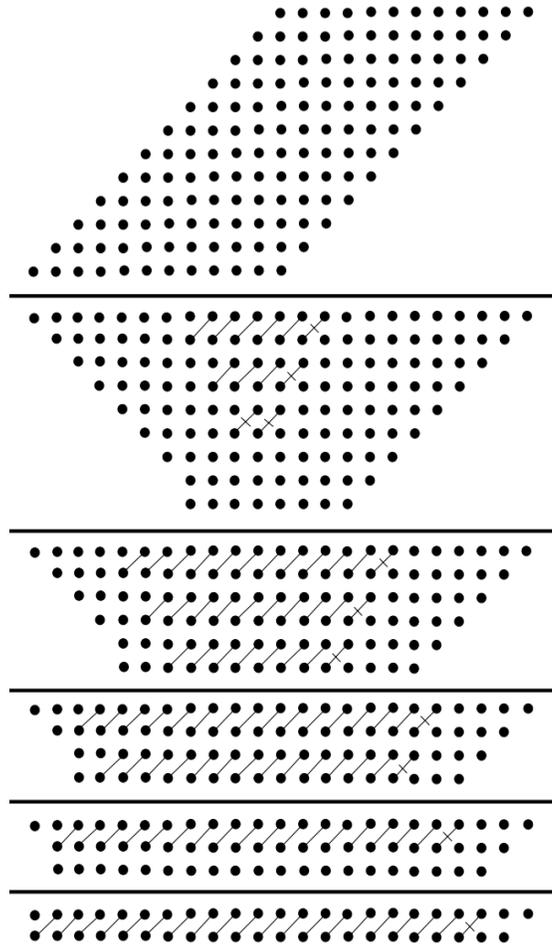


Figure 4: 12x12 Dadda Multiplier [9].

### 2.2.3 REDUCED AREA MULTIPLIER

The Reduced Area multiplier is a combination of some of the refined methodologies used in Wallace and Dadda multipliers. Specifically, Reduced Area multipliers use a large number of (3,2) counters or full adders as early as possible to reduce the partial product matrix quickly. This reduces the number of bits being propagated between stages during the reduction phase. This scheme also uses (2,2)

counters carefully with an intention to reduce the word size of the carry propagate adder. Overall, these methodologies reduce the number of interconnects and the complexity of the circuit. Thus, Reduced Area multipliers, as the name suggests, tend to require lower area as compared to Wallace and Dadda multipliers.

In each stage, a large number of (3,2) counters are used to reduce the number of partial products entering the next stage. Using half adders in the right most column with two bits reduces the width of the carry propagate adder by an amount equal to the number of stages in the multiplier algorithm. It can also be observed that the (2,2) counters are used only in the following scenarios:

- To reduce the right most column containing exactly two bits in order to reduce the word size of the CPA
- To reduce the number of bits in a column to the height of the matrix specified by Dadda's height series.

The dot diagram of a 12x12 Reduced Area multiplier is shown in *Figure 5*. The example shown uses 5 stages to compute the product of two 12bit inputs with 104 (3,2) counters and 11 (2,2) counters. The width of the final CPA adder is 17bits.

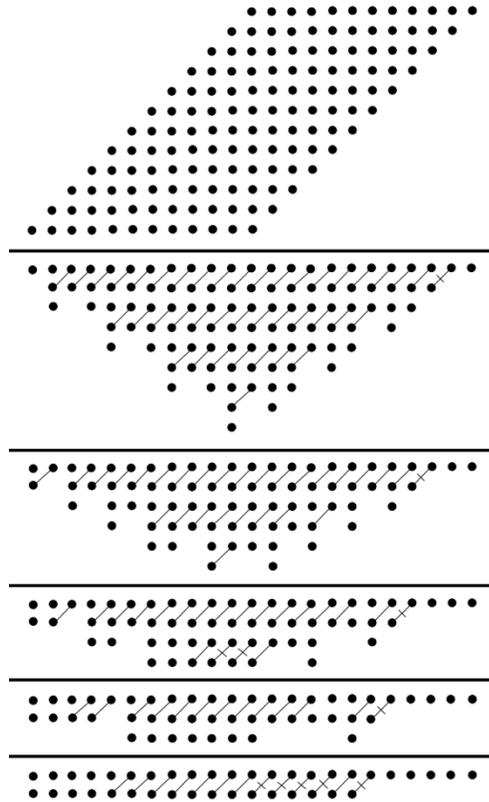


Figure 5: 12x12 Reduced Area Multiplier [9].

#### 2.2.4 VARIABLE SET MULTIPLIER

Variable set multiplier is also a type of column compression parallel multiplier with a novel reduction scheme that performs grouping of the partial products in variable heights depending on the stage and height of the partial product matrix. The proposed design aims at lowering power utilization and bringing more regularity into the partial product reduction stages, which can make it easier to draw the layout of the multiplier. Similar to the other column compression schemes, this algorithm also uses three basic steps in computing the product:

- Step 1: Partial product matrix generation
- Step 2: Partial product reduction to a matrix height of two rows
- Step 3: Using a carry propagate adder to add the final two rows and compute the product.

The dot diagram of the proposed Variable Set multiplier scheme is as shown in *Figure 6*. The algorithm used to reduce the partial product is as explained below:

- Group the rows of the PP matrix as follows:
  - Into sets of 4 rows if the number of rows is a multiple of four
  - Into sets of 3 rows if the number of rows is not a multiple of four
- In each set, use (3,2) counters or full adders to reduce three bits of each column to two bits thus reducing the height of the sets from 4 to 3 in case of four row sets or from 3 to 2 in case of three row sets.
- The (2,2) counters or half adders are used only in the following scenarios:
  - To reduce the right most column containing exactly two bits to minimize the CPA width
  - If reduction of bits in a column can result in reduction in height for a particular stage

If neither of the above benefits are obtained, the (2,2) counters are avoided and the partial product bits are passed unmodified to the next stage.

- The pattern for set height reduction is 4 -> 3 -> 2 throughout the design. When multiple 2 row sets are obtained in each stage, regroup the sets accordingly as described in the first step and repeat the process till the matrix is reduced to two rows.
- Use a suitable carry-propagate adder to compute product by adding the final two rows of the matrix.

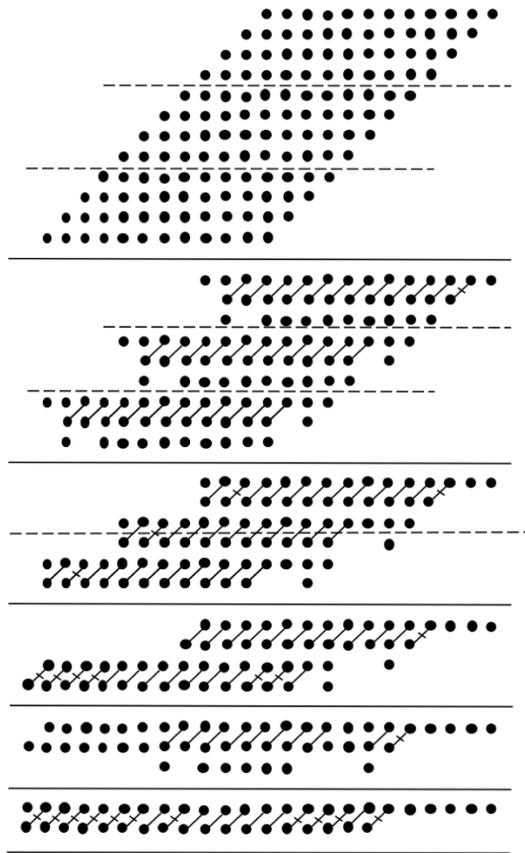


Figure 6: 12x12 Variable Set Multiplier.

To better understand the reduction scheme, an example walk-through is presented below. In case of the illustrated 12x12 variable set multiplier, the partial product matrix is reduced as follows:

- Stage 1: Since the overall matrix height is a multiple of 4, the partial products are grouped into sets of four rows each. Each of these 4 rows are reduced to 3 row sets using full adders. One half adder is used only to reduce the rightmost column containing 2 bits.

- Stage 2: The 3-row sets are further reduced to 2-row sets using full adders. Half adders are not only used to reduce the right most bits but also to reduce the column bits such that the set height can be reduced to 2. The three 2-row sets (6 rows -> not a multiple of 4 but a multiple of 3) are then regrouped into sets of 3-rows.
- Stage 3: The re-grouped 3-row sets are reduced to sets of 2-rows. The two 2-row sets are now regrouped into 4 row sets as the number of rows is a multiple of four.
- Stage 4: The regrouped 4-row set is reduced to a 3-row set using appropriate number of full adders and a half adder
- Stage 5: The final stage in which the 3-row set is reduced to 2-row set, which is then fed into an 18-bit carry propagate adder.

### 3. COLUMN COMPRESSION MULTIPLIER COMPARISONS

In this section we discuss and compare the proposed Variable Set multiplier with Dadda, Wallace and Reduced Area multipliers in terms of gate complexity, delay, interconnects and power utilization for 4bit, 7bit, 8bit and 12bit versions. Further, a 12bit version of each multiplier was implemented in structural Verilog with basic gates, simulated on Mentor QuestaSim and synthesized using Xilinx Vivado. The net cell usage statistics and the power utilization values were extracted and compared. All the simulation and implementation results are computed by using Ripple Carry Adder for the Carry Propagate Adder stage. Some of the important formulae used for computations are listed below:

#### A. GATE DELAYS

- CLA Gate Delay =  $2 + 4 \lceil \log_r n \rceil$  (assuming  $r = 4$  for all computations)
- RCA Gate Delay for sum =  $2n + 4$
- Full Adder Delay = 6
- Half Adder Delay = 3

#### B. GATE COMPLEXITIES

- CLA Gate complexity =  $12\frac{2}{3}n - 2\frac{2}{3}$
- RCA Gate complexity =  $9n$
- Full Adder Gate complexity = 9
- Half Adder Gate complexity = 4

### 3.1 GATE COMPLEXITY

*Table 1* illustrates the number of (3,2) counters or FA, (2,2) counters or HA and width of CPA adder used by each of the multiplier schemes for 4bit, 7bit, 8bit and 12bit versions. The 7bit version has been discussed to demonstrate the performance of the proposed design in case of non-conventional input word size. *Figure 7* graphically represents each of these component complexities.

Table 1: Component Complexity in parallel multipliers.

WORD SIZE	Wallace			Dadda			Reduced Area			Variable Set		
	# FA	# HA	CPA width	# FA	# HA	CPA width	# FA	# HA	CPA width	# FA	# HA	CPA width
<b>4x4</b>	5	3	4	3	3	6	5	3	4	5	3	4
<b>7x7</b>	27	12	9	24	6	12	28	6	8	27	10	9
<b>8x8</b>	38	15	11	35	7	14	39	7	10	38	11	11
<b>12x12</b>	102	34	18	99	11	22	104	11	17	98	24	18

Further, to analyze the exact gate count per multiplier block, each of the above listed components are broken down into number of corresponding basic gates. *Table 2* illustrates the total gate count per multiplier block obtained using Carry Lookahead adder in the final CPA stage while *Table 3* illustrates the total gate count obtained using Ripple Carry Adders as the CPA, which is later used for all simulations. *Figure 8* shows the total gate complexity for each multiplier block using CLA and RCA respectively.

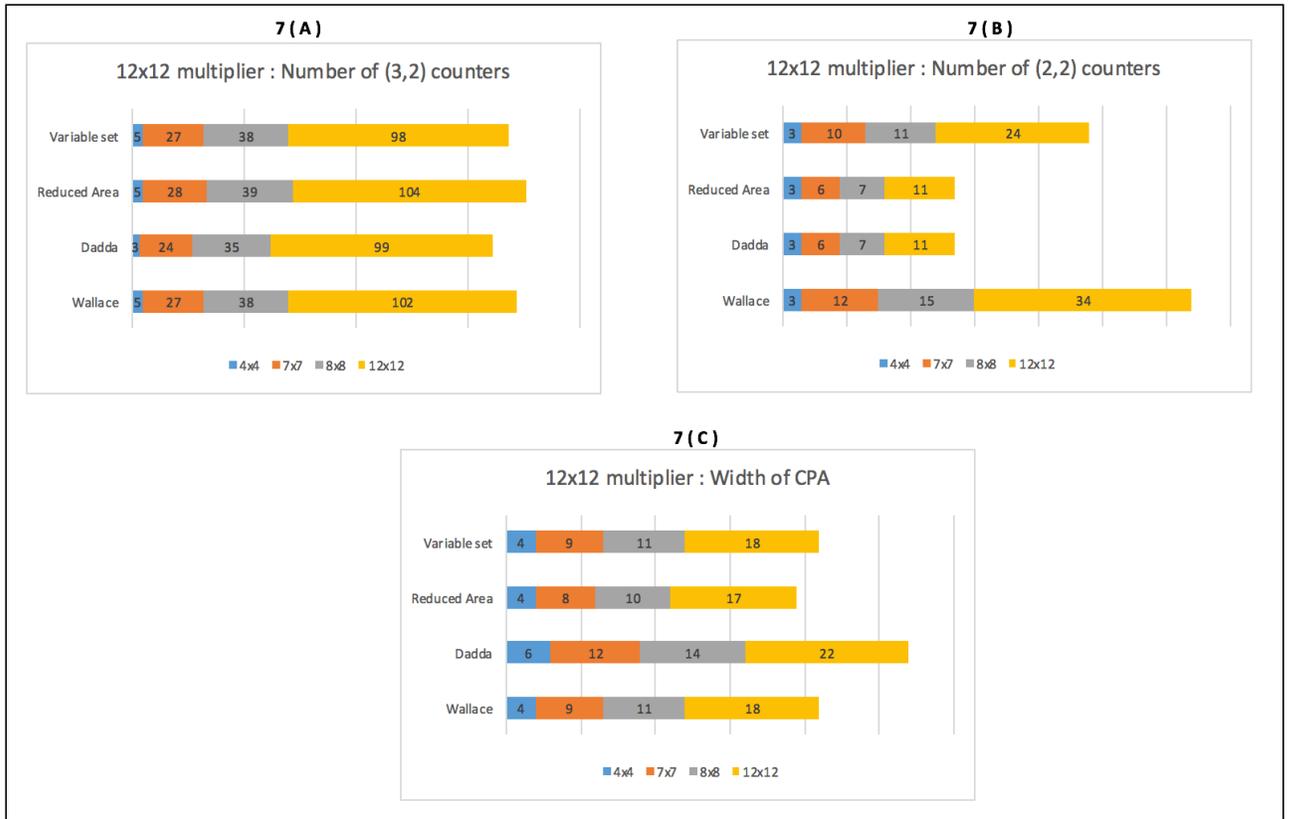


Figure 7: (A) Total number of (3,2) counters used in each multiplier block; (B) Total number of (2,2) counters used in each multiplier block; (C) Width of the Carry Propagate Adder to be used in the final stage in each multiplier.

Table 2: Total gate count using Carry Lookahead Adders.

WORD SIZE	Wallace	Dadda	Reduced Area	Variable Set
4x4	105	113	105	105
7x7	402	389	375	394
8x8	539	518	503	523
12x12	1280	1211	1193	1204

Table 3: Total gate count using Ripple Carry Adders.

WORD SIZE	Wallace	Dadda	Reduced Area	Variable Set
4x4	93	93	93	93
7x7	372	348	348	364
8x8	501	469	469	485
12x12	1216	1133	1133	1140

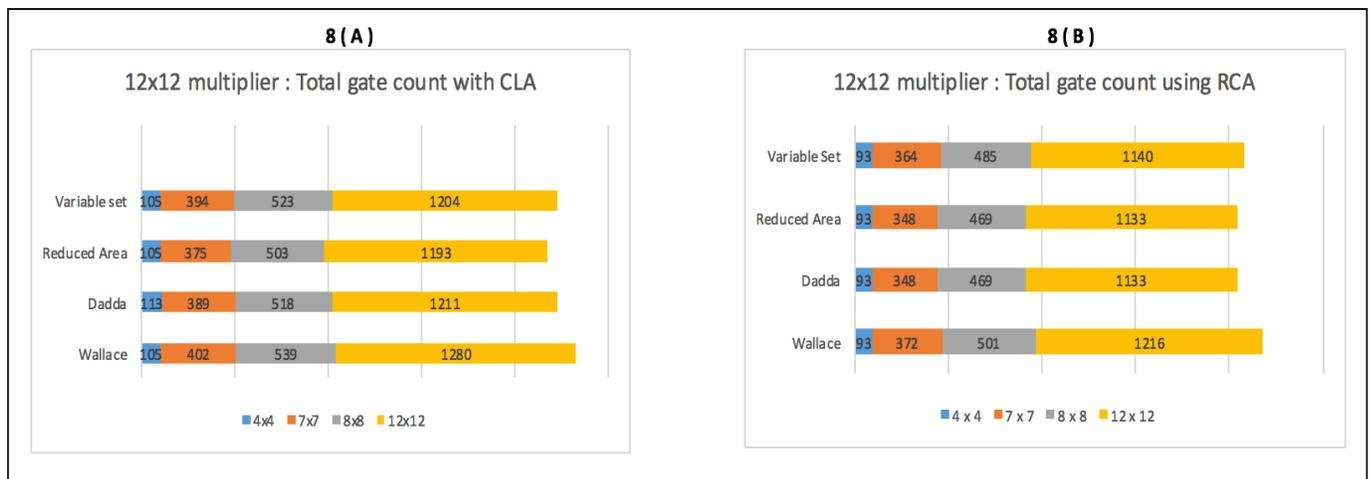


Figure 8: Total Gate Complexity with (A) Carry Lookahead Adder; (B) Ripple Carry Adder

### 3.2 GATE DELAY

The gate delay for any parallel multiplier is largely dependent on the number of stages involved in the reduction process and the delay due to the carry propagate adder. Since all multipliers have the same number of reduction stages and since Wallace and Variable Set multipliers use lower width CPA, they are expected to be faster than Dadda multipliers. However, more detailed analysis as shown in [10] proves otherwise. In this

section we only analyze the coarse gate delay values observed using Ripple Carry Adder to get a sense of the effect of CPA width on Gate delays. *Figure 9* shows a graphical representation of Gate delays for various multiplier blocks.

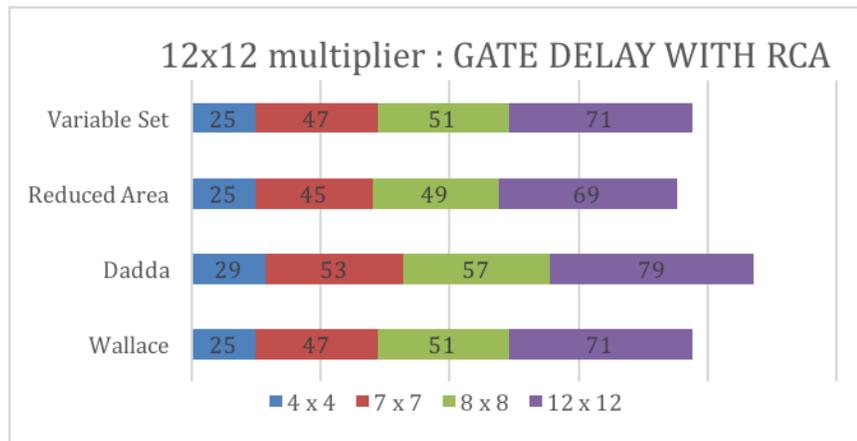


Figure 9: Abstract Gate Delays using RCA as CPA

### 3.3 POST IMPLEMENTATION CELL USAGE

The 12x12 version of all the parallel multipliers were simulated in structural Verilog and synthesized on Xilinx Vivado for Artix-7 FPGA architecture. *Table 4* summarizes the Cell Usage report for each multiplier block and *Figure 10* displays the breakdown of the number of lookup tables used (LUTs) during implementation of the multiplier blocks. It can be observed that the Reduced Area Multiplier has the lowest cell utilization values due to optimal area and gate counts.

Table 4: Summary of Cell Usage Report.

CELL USAGE REPORT	Wallace	Dadda	Reduced Area	Variable Set
LUT Flip Flop pairs	190	184	177	187
SLICEL	41	40	36	42
SLICEM	13	13	13	14

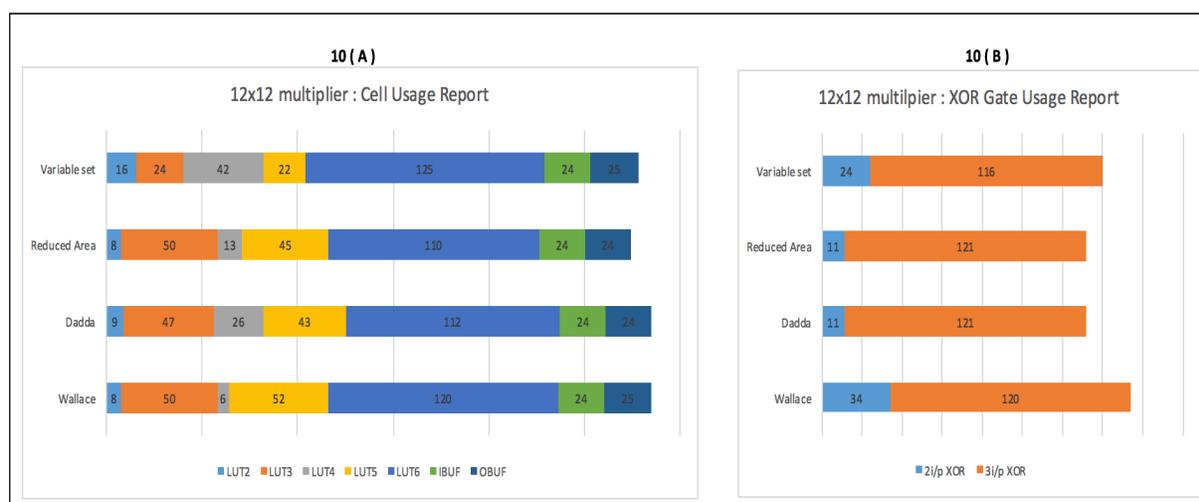


Figure 10: Cell Usage Report from Vivado

### 3.4 POWER UTILIZATION

Low-power utilization is an essential design feature that is required in semiconductor industry. In order to truly compare the performance of all the column compression multipliers, it is required to observe their power utilization values. *Table 5* summarizes the on-chip and dynamic power utilization values in watts(W). *Figure 11* represents the distribution of the dynamic power between the logic blocks for each multiplier. It is observed that for a 12x12 multiplier block, the Variable Set multiplier is more power efficient than its counterparts.

Table 5: Power Utilization Report.

POWER (Temp = 125 degree C)	Wallace	Dadda	Reduced Area	Variable Set
On – Chip power (W)	23.699	24.012	24.082	23.399
Dynamic Power (W)	23.214	23.528	23.597	22.915

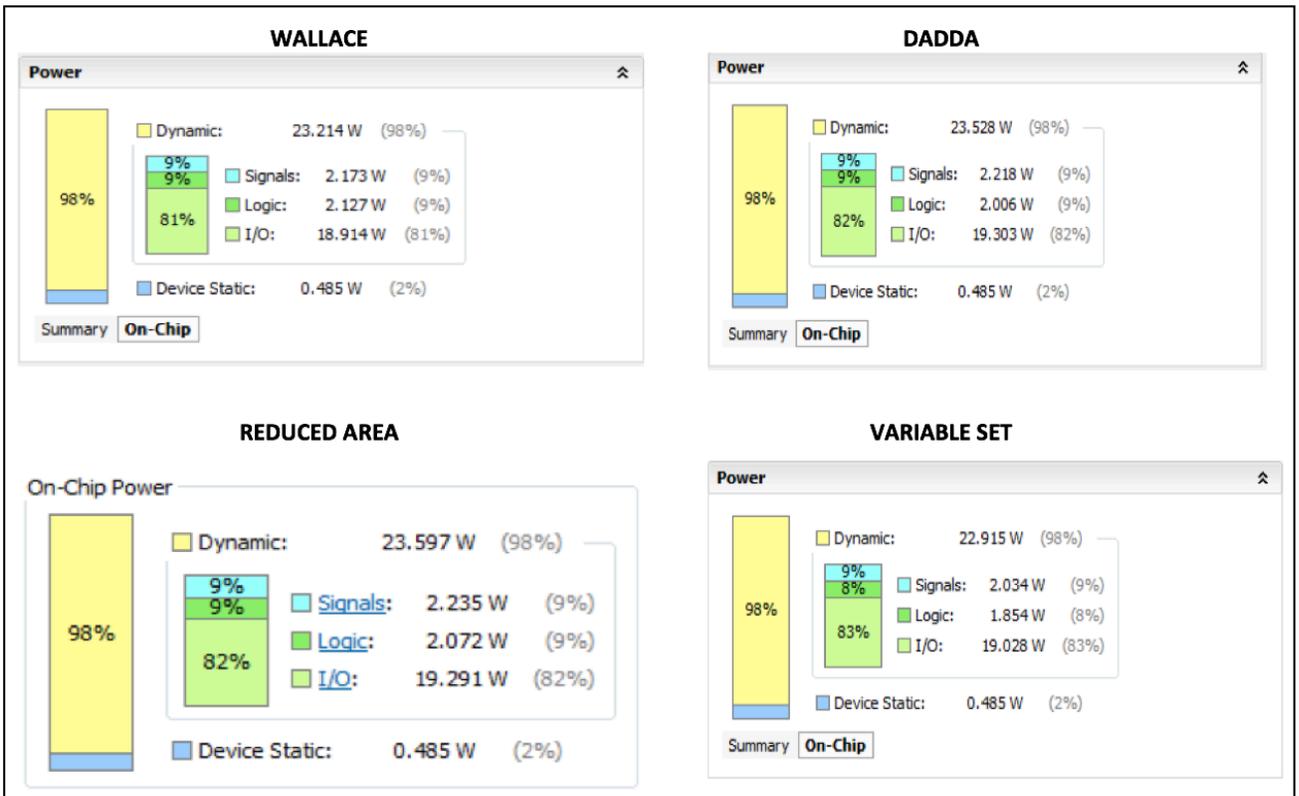


Figure 11: On-Chip Power Distribution Report.

### 3.5 ROUTING STATUS: INTERCONNECTS

*Figure 12* shows the number of nets used as interconnects to implement the design. As discussed previously, Reduced Area Multipliers have lower interconnect complexity due to the aggressive reduction of many partial product bits in the early stages of multiplication algorithm. The routing complexity observed for Variable Set multiplier is observed to be better than that of Wallace and Dadda multipliers.

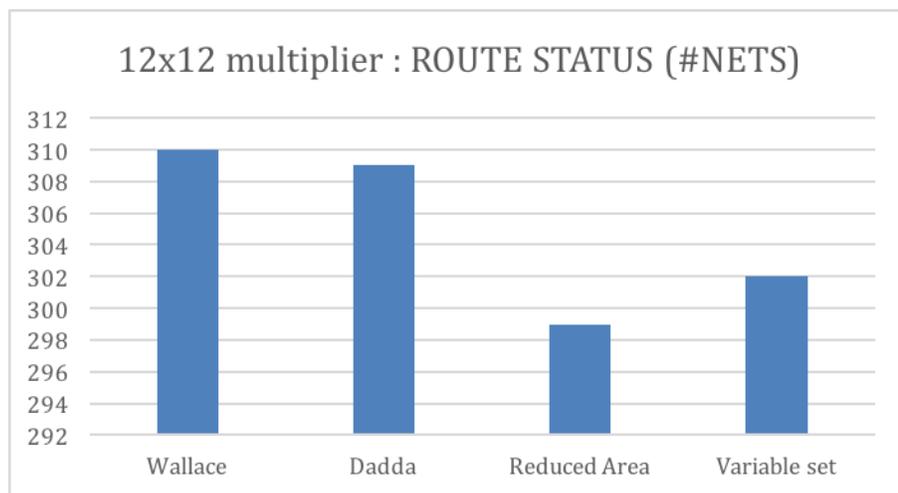


Figure 12: Routing Complexity.

### 3.6 SCHEMATIC AND POST IMPLEMENTATION LAYOUTS

This section illustrates the schematic and layouts obtained during the synthesis and implementation of the 12bit multipliers.

#### 3.6.1 WALLACE MULTIPLIER

*Figure 13* illustrates the schematic of a 12bit Wallace Multiplier and *Figure 14* depicts the corresponding device layout obtained after implementation of the design.

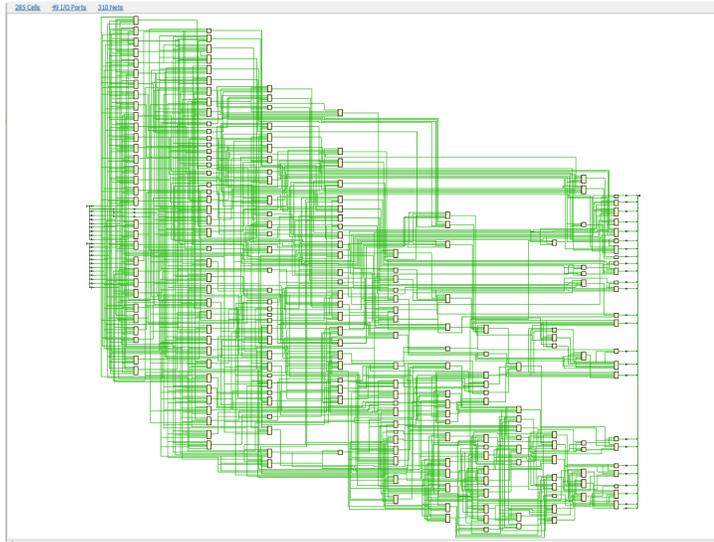


Figure 13: Schematic of 12x12 Wallace Multiplier.

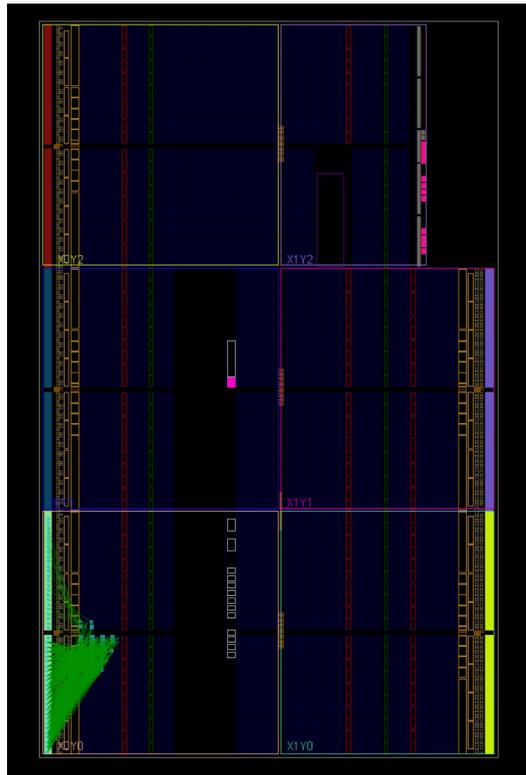


Figure 14: Layout of 12x12 Wallace Multiplier.

### 3.6.2 DADDA MULTIPLIER

*Figure 15* illustrates the schematic of a 12bit Dadda Multiplier and *Figure 16* depicts the corresponding device layout obtained after implementation of the design.

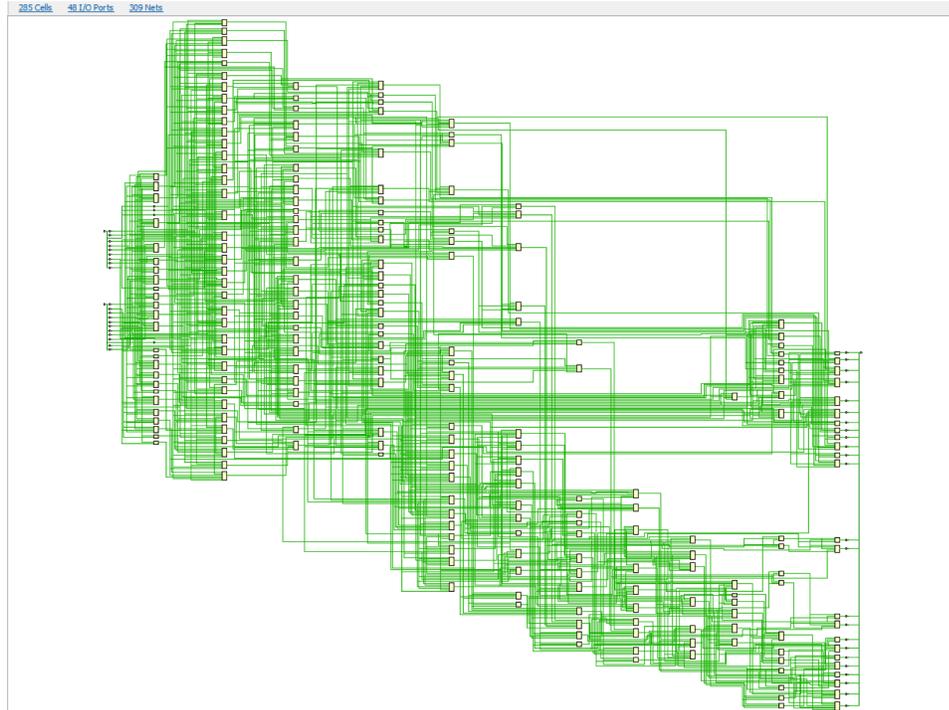


Figure 15: Schematic of 12x12 Dadda Multiplier.

### 3.6.3 REDUCED AREA MULTIPLIER

*Figure 17* illustrates the schematic of a 12bit Reduced Area Multiplier and *Figure 18* depicts the corresponding device layout obtained after implementation of the design.



Figure 16: Layout of 12x12 Dadda Multiplier.

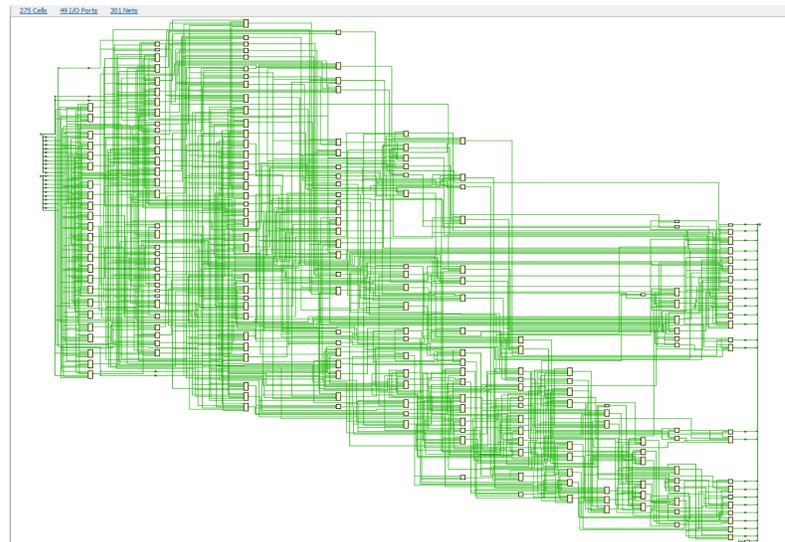


Figure 17: Schematic of 12x12 Reduced Area Multiplier.

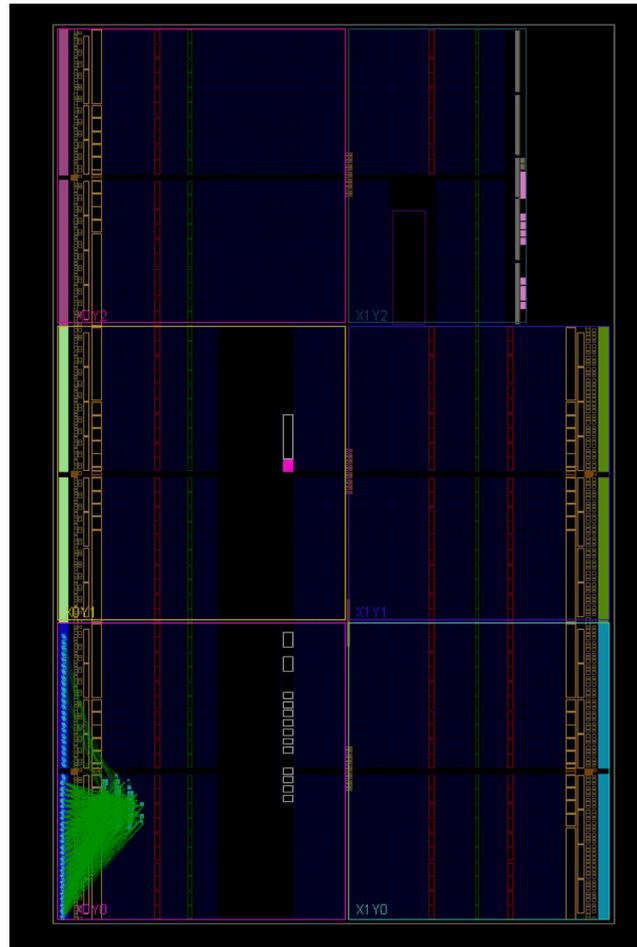


Figure 18: Layout of 12x12 Reduced Area Multiplier.

### 3.6.4 VARIABLE SET MULTIPLIER

*Figure 19* illustrates the schematic of a 12bit Variable Set Multiplier and *Figure 20* depicts the corresponding device layout obtained after implementation of the design.

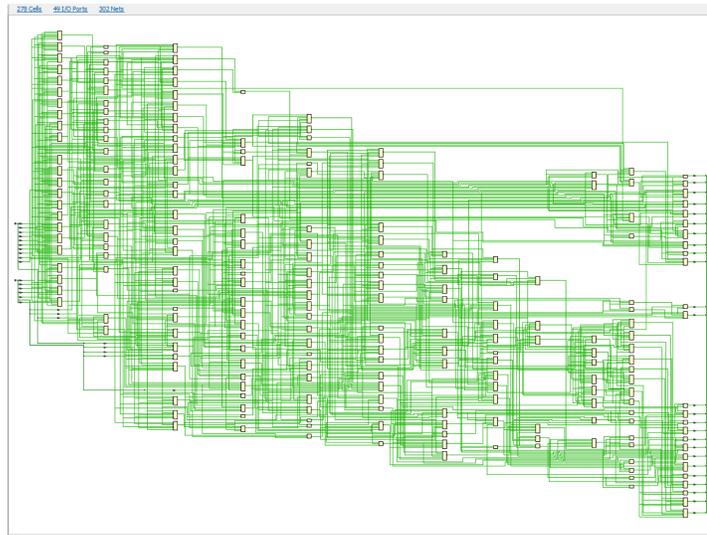


Figure 19: Schematic of 12x12 Variable Set Multiplier.

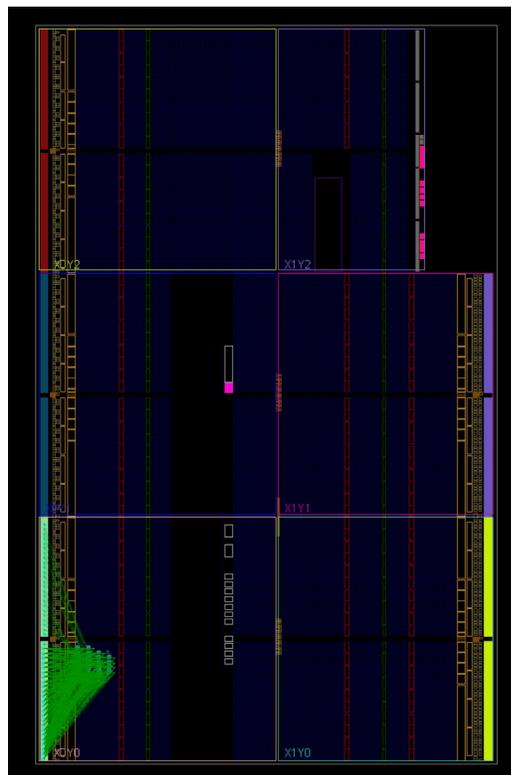


Figure 20: Layout of 12x12 Variable Set Multiplier.

## 4. CONCLUSION

A novel fast multiplier scheme called Variable Set multiplier has been proposed and described in this report. The Variable Set multiplier performs reduction based on non-uniform grouping of rows in the sets across various stages of the reduction process. In every stage, a number of partial products rows, which varies depending on the stage height of the reduction scheme, are grouped into sets. The design methodologies, complexity, delay and power consumption of four column compression fast multipliers namely, Wallace, Dadda, Reduced Area and Variable Set multipliers have been compared and presented. It is observed that, though Wallace multipliers have fairly regular structure making it easier to draw its layout, it has very high gate complexity and thus high area and power utilization. On the other hand, Dadda multipliers are more area efficient than Wallace but harder to layout and consume more power. In case of Reduced Area multipliers, it is observed that the gate complexity and hence the area is the lowest in majority of the cases and it also has low interconnect overhead making it very attractive for area constraint applications. In case of Variable Set multipliers, its complexity is found to be around 3% lower than that of Wallace, comparable or slightly higher than Dadda and Reduced Area multipliers. However, in case of on-chip and dynamic power utilization, the Variable Set multiplier was found to be more efficient than its counterparts. Hence the proposed design can be further explored for its usage in low-power applications.

## Appendix

The code used to simulate and synthesize the Variable Set multiplier is presented in this section.

```
module partial_product(
  output [0:11] pp0, pp1, pp2, pp3,
  output [0:11] pp4, pp5, pp6, pp7,
  output [0:11] pp8, pp9, pp10, pp11,
  input [11:0] op1, op2
);

  and p00 (pp0[0], op1[0], op2[0]);
  and p01 (pp0[1], op1[0], op2[1]);
  and p02 (pp0[2], op1[0], op2[2]);
  and p03 (pp0[3], op1[0], op2[3]);
  and p04 (pp0[4], op1[0], op2[4]);
  and p05 (pp0[5], op1[0], op2[5]);
  and p06 (pp0[6], op1[0], op2[6]);
  and p07 (pp0[7], op1[0], op2[7]);
  and p08 (pp0[8], op1[0], op2[8]);
  and p09 (pp0[9], op1[0], op2[9]);
  and p010 (pp0[10], op1[0], op2[10]);
  and p011 (pp0[11], op1[0], op2[11]);

  and p10 (pp1[0], op1[1], op2[0]);
  and p11 (pp1[1], op1[1], op2[1]);
  and p12 (pp1[2], op1[1], op2[2]);
  and p13 (pp1[3], op1[1], op2[3]);
  and p14 (pp1[4], op1[1], op2[4]);
  and p15 (pp1[5], op1[1], op2[5]);
  and p16 (pp1[6], op1[1], op2[6]);
  and p17 (pp1[7], op1[1], op2[7]);
  and p18 (pp1[8], op1[1], op2[8]);
  and p19 (pp1[9], op1[1], op2[9]);
  and p0110 (pp1[10], op1[1], op2[10]);
  and p0111 (pp1[11], op1[1], op2[11]);

  and p20 (pp2[0], op1[2], op2[0]);
  and p21 (pp2[1], op1[2], op2[1]);
  and p22 (pp2[2], op1[2], op2[2]);
  and p23 (pp2[3], op1[2], op2[3]);
  and p24 (pp2[4], op1[2], op2[4]);
  and p25 (pp2[5], op1[2], op2[5]);
  and p26 (pp2[6], op1[2], op2[6]);
  and p27 (pp2[7], op1[2], op2[7]);
  and p28 (pp2[8], op1[2], op2[8]);
  and p29 (pp2[9], op1[2], op2[9]);
```

```

and p210 (pp2[10], op1[2], op2[10]);
and p211 (pp2[11], op1[2], op2[11]);

and p30 (pp3[0], op1[3], op2[0]);
and p31 (pp3[1], op1[3], op2[1]);
and p32 (pp3[2], op1[3], op2[2]);
and p33 (pp3[3], op1[3], op2[3]);
and p34 (pp3[4], op1[3], op2[4]);
and p35 (pp3[5], op1[3], op2[5]);
and p36 (pp3[6], op1[3], op2[6]);
and p37 (pp3[7], op1[3], op2[7]);
and p38 (pp3[8], op1[3], op2[8]);
and p39 (pp3[9], op1[3], op2[9]);
and p310 (pp3[10], op1[3], op2[10]);
and p311 (pp3[11], op1[3], op2[11]);

and p40 (pp4[0], op1[4], op2[0]);
and p41 (pp4[1], op1[4], op2[1]);
and p42 (pp4[2], op1[4], op2[2]);
and p43 (pp4[3], op1[4], op2[3]);
and p44 (pp4[4], op1[4], op2[4]);
and p45 (pp4[5], op1[4], op2[5]);
and p46 (pp4[6], op1[4], op2[6]);
and p47 (pp4[7], op1[4], op2[7]);
and p48 (pp4[8], op1[4], op2[8]);
and p49 (pp4[9], op1[4], op2[9]);
and p410 (pp4[10], op1[4], op2[10]);
and p411 (pp4[11], op1[4], op2[11]);

and p50 (pp5[0], op1[5], op2[0]);
and p51 (pp5[1], op1[5], op2[1]);
and p52 (pp5[2], op1[5], op2[2]);
and p53 (pp5[3], op1[5], op2[3]);
and p54 (pp5[4], op1[5], op2[4]);
and p55 (pp5[5], op1[5], op2[5]);
and p56 (pp5[6], op1[5], op2[6]);
and p57 (pp5[7], op1[5], op2[7]);
and p58 (pp5[8], op1[5], op2[8]);
and p59 (pp5[9], op1[5], op2[9]);
and p510 (pp5[10], op1[5], op2[10]);
and p511 (pp5[11], op1[5], op2[11]);

and p60 (pp6[0], op1[6], op2[0]);
and p61 (pp6[1], op1[6], op2[1]);
and p62 (pp6[2], op1[6], op2[2]);
and p63 (pp6[3], op1[6], op2[3]);
and p64 (pp6[4], op1[6], op2[4]);
and p65 (pp6[5], op1[6], op2[5]);
and p66 (pp6[6], op1[6], op2[6]);
and p67 (pp6[7], op1[6], op2[7]);
and p68 (pp6[8], op1[6], op2[8]);

```

```

and p69 (pp6[9], op1[6], op2[9]);
and p610 (pp6[10], op1[6], op2[10]);
and p611 (pp6[11], op1[6], op2[11]);

and p70 (pp7[0], op1[7], op2[0]);
and p71 (pp7[1], op1[7], op2[1]);
and p72 (pp7[2], op1[7], op2[2]);
and p73 (pp7[3], op1[7], op2[3]);
and p74 (pp7[4], op1[7], op2[4]);
and p75 (pp7[5], op1[7], op2[5]);
and p76 (pp7[6], op1[7], op2[6]);
and p77 (pp7[7], op1[7], op2[7]);
and p78 (pp7[8], op1[7], op2[8]);
and p79 (pp7[9], op1[7], op2[9]);
and p710 (pp7[10], op1[7], op2[10]);
and p711 (pp7[11], op1[7], op2[11]);

and p80 (pp8[0], op1[8], op2[0]);
and p81 (pp8[1], op1[8], op2[1]);
and p82 (pp8[2], op1[8], op2[2]);
and p83 (pp8[3], op1[8], op2[3]);
and p84 (pp8[4], op1[8], op2[4]);
and p85 (pp8[5], op1[8], op2[5]);
and p86 (pp8[6], op1[8], op2[6]);
and p87 (pp8[7], op1[8], op2[7]);
and p88 (pp8[8], op1[8], op2[8]);
and p89 (pp8[9], op1[8], op2[9]);
and p810 (pp8[10], op1[8], op2[10]);
and p811 (pp8[11], op1[8], op2[11]);

and p90 (pp9[0], op1[9], op2[0]);
and p91 (pp9[1], op1[9], op2[1]);
and p92 (pp9[2], op1[9], op2[2]);
and p93 (pp9[3], op1[9], op2[3]);
and p94 (pp9[4], op1[9], op2[4]);
and p95 (pp9[5], op1[9], op2[5]);
and p96 (pp9[6], op1[9], op2[6]);
and p97 (pp9[7], op1[9], op2[7]);
and p98 (pp9[8], op1[9], op2[8]);
and p99 (pp9[9], op1[9], op2[9]);
and p910 (pp9[10], op1[9], op2[10]);
and p911 (pp9[11], op1[9], op2[11]);

and p100 (pp10[0], op1[10], op2[0]);
and p101 (pp10[1], op1[10], op2[1]);
and p102 (pp10[2], op1[10], op2[2]);
and p103 (pp10[3], op1[10], op2[3]);
and p104 (pp10[4], op1[10], op2[4]);
and p105 (pp10[5], op1[10], op2[5]);
and p106 (pp10[6], op1[10], op2[6]);
and p107 (pp10[7], op1[10], op2[7]);

```

```

    and p108 (pp10[8], op1[10], op2[8]);
    and p109 (pp10[9], op1[10], op2[9]);
    and p1010 (pp10[10], op1[10], op2[10]);
    and p1011 (pp10[11], op1[10], op2[11]);

    and p110 (pp11[0], op1[11], op2[0]);
    and p111 (pp11[1], op1[11], op2[1]);
    and p112 (pp11[2], op1[11], op2[2]);
    and p113 (pp11[3], op1[11], op2[3]);
    and p114 (pp11[4], op1[11], op2[4]);
    and p115 (pp11[5], op1[11], op2[5]);
    and p116 (pp11[6], op1[11], op2[6]);
    and p117 (pp11[7], op1[11], op2[7]);
    and p118 (pp11[8], op1[11], op2[8]);
    and p119 (pp11[9], op1[11], op2[9]);
    and p1110 (pp11[10], op1[11], op2[10]);
    and p1111 (pp11[11], op1[11], op2[11]);

endmodule

module variable_set12 (
    output[24:0] product,
    input [11:0] op1,
    input [11:0] op2
);

    //intermediate stage signals
    wire p0_0, p0_1, p0_2, p0_3, p0_4, p0_5, p0_6, p0_7, p0_8, p0_9,
    p0_10, p0_11;
    wire p1_1, p1_2, p1_3, p1_4, p1_5, p1_6, p1_7, p1_8, p1_9, p1_10,
    p1_11, p1_12;
    wire p2_2, p2_3, p2_4, p2_5, p2_6, p2_7, p2_8, p2_9, p2_10,
    p2_11, p2_12, p2_13;
    wire p3_3, p3_4, p3_5, p3_6, p3_7, p3_8, p3_9, p3_10, p3_11,
    p3_12, p3_13, p3_14;
    wire p4_4, p4_5, p4_6, p4_7, p4_8, p4_9, p4_10, p4_11, p4_12,
    p4_13, p4_14, p4_15;
    wire p5_5, p5_6, p5_7, p5_8, p5_9, p5_10, p5_11, p5_12, p5_13,
    p5_14, p5_15, p5_16;
    wire p6_6, p6_7, p6_8, p6_9, p6_10, p6_11, p6_12, p6_13, p6_14,
    p6_15, p6_16, p6_17;
    wire p7_7, p7_8, p7_9, p7_10, p7_11, p7_12, p7_13, p7_14, p7_15,
    p7_16, p7_17, p7_18;
    wire p8_8, p8_9, p8_10, p8_11, p8_12, p8_13, p8_14, p8_15, p8_16,
    p8_17, p8_18, p8_19;
    wire p9_9, p9_10, p9_11, p9_12, p9_13, p9_14, p9_15, p9_16,
    p9_17, p9_18, p9_19, p9_20;
    wire p10_10, p10_11, p10_12, p10_13, p10_14, p10_15, p10_16,
    p10_17, p10_18, p10_19, p10_20, p10_21;
    wire p11_11, p11_12, p11_13, p11_14, p11_15, p11_16, p11_17,
    p11_18, p11_19, p11_20, p11_21, p11_22;

```

```

    wire s1p0_1, s1p0_2, s1p0_3, s1p0_4, s1p0_5, s1p0_6, s1p0_7,
    s1p0_8, s1p0_9, s1p0_10, s1p0_11, s1p0_12;
    wire s1p1_2, s1p1_3, s1p1_4, s1p1_5, s1p1_6, s1p1_7, s1p1_8,
    s1p1_9, s1p1_10, s1p1_11, s1p1_12, s1p1_13;
    wire s1p3_6, s1p3_7, s1p3_8, s1p3_9, s1p3_10, s1p3_11, s1p3_12,
    s1p3_13, s1p3_14, s1p3_15, s1p3_16;
    wire s1p4_7, s1p4_8, s1p4_9, s1p4_10, s1p4_11, s1p4_12, s1p4_13,
    s1p4_14, s1p4_15, s1p4_16, s1p4_17;
    wire s1p6_10, s1p6_11, s1p6_12, s1p6_13, s1p6_14, s1p6_15,
    s1p6_16, s1p6_17, s1p6_18, s1p6_19, s1p6_20;
    wire s1p7_11, s1p7_12, s1p7_13, s1p7_14, s1p7_15, s1p7_16,
    s1p7_17, s1p7_18, s1p7_19, s1p7_20, s1p7_21;

    wire s2p0_2, s2p0_3, s2p0_4, s2p0_5, s2p0_6, s2p0_7, s2p0_8,
    s2p0_9, s2p0_10, s2p0_11, s2p0_12, s2p0_13;
    wire s2p1_3, s2p1_4, s2p1_5, s2p1_6, s2p1_7, s2p1_8, s2p1_9,
    s2p1_10, s2p1_11, s2p1_12, s2p1_13, s2p1_14;
    wire s2p2_7, s2p2_8, s2p2_9, s2p2_10, s2p2_11, s2p2_12, s2p2_13,
    s2p2_14, s2p2_15, s2p2_16, s2p2_17;
    wire s2p3_8, s2p3_9, s2p3_10, s2p3_11, s2p3_12, s2p3_13, s2p3_14,
    s2p3_15, s2p3_16, s2p3_17, s2p3_18;
    wire s2p4_11, s2p4_12, s2p4_13, s2p4_14, s2p4_15, s2p4_16,
    s2p4_17, s2p4_18, s2p4_19, s2p4_20, s2p4_21;
    wire s2p5_12, s2p5_13, s2p5_14, s2p5_15, s2p5_16, s2p5_17,
    s2p5_18, s2p5_19, s2p5_20, s2p5_21, s2p5_22;

    wire s3p0_3, s3p0_4, s3p0_5, s3p0_6, s3p0_7, s3p0_8, s3p0_9,
    s3p0_10, s3p0_11, s3p0_12, s3p0_13, s3p0_14;
    wire s3p1_4, s3p1_5, s3p1_6, s3p1_7, s3p1_8, s3p1_9, s3p1_10,
    s3p1_11, s3p1_12, s3p1_13, s3p1_14, s3p1_15;
    wire s3p2_9, s3p2_10, s3p2_11, s3p2_12, s3p2_13, s3p2_14,
    s3p2_15, s3p2_16, s3p2_17, s3p2_18, s3p2_19, s3p2_20, s3p2_21,
    s3p2_22 ;
    wire s3p3_10, s3p3_11, s3p3_12, s3p3_13, s3p3_14, s3p3_15,
    s3p3_16, s3p3_17, s3p3_18, s3p3_19, s3p3_20, s3p3_21, s3p3_22,
    s3p3_23;

    wire s5p0_5, s5p0_6, s5p0_7, s5p0_8, s5p0_9, s5p0_10, s5p0_11,
    s5p0_12, s5p0_13, s5p0_14, s5p0_15, s5p0_16, s5p0_17, s5p0_18,
    s5p0_19, s5p0_20, s5p0_21, s5p0_22;
    wire s5p1_6, s5p1_7, s5p1_8, s5p1_9, s5p1_10, s5p1_11, s5p1_12,
    s5p1_13, s5p1_14, s5p1_15, s5p1_16, s5p1_17, s5p1_18, s5p1_19,
    s5p1_20, s5p1_21, s5p1_22, s5p1_23;

    //partial product formation
    partial_product getPP ( {p0_0, p0_1, p0_2, p0_3, p0_4, p0_5,
    p0_6, p0_7, p0_8, p0_9, p0_10, p0_11},
    {p1_1, p1_2, p1_3, p1_4, p1_5, p1_6,
    p1_7, p1_8, p1_9, p1_10, p1_11, p1_12},
    {p2_2, p2_3, p2_4, p2_5, p2_6, p2_7,

```

```

p2_8, p2_9, p2_10, p2_11, p2_12, p2_13},
      {p3_3, p3_4, p3_5, p3_6, p3_7, p3_8,
p3_9, p3_10, p3_11, p3_12, p3_13, p3_14},
      {p4_4, p4_5, p4_6, p4_7, p4_8, p4_9,
p4_10, p4_11, p4_12, p4_13, p4_14, p4_15},
      {p5_5, p5_6, p5_7, p5_8, p5_9, p5_10,
p5_11, p5_12, p5_13, p5_14, p5_15, p5_16},
      {p6_6, p6_7, p6_8, p6_9, p6_10, p6_11,
p6_12, p6_13, p6_14, p6_15, p6_16, p6_17},
      {p7_7, p7_8, p7_9, p7_10, p7_11, p7_12,
p7_13, p7_14, p7_15, p7_16, p7_17, p7_18},
      {p8_8, p8_9, p8_10, p8_11, p8_12, p8_13,
p8_14, p8_15, p8_16, p8_17, p8_18, p8_19},
      {p9_9, p9_10, p9_11, p9_12, p9_13, p9_14,
p9_15, p9_16, p9_17, p9_18, p9_19, p9_20},
      {p10_10, p10_11, p10_12, p10_13, p10_14,
p10_15, p10_16, p10_17, p10_18, p10_19, p10_20, p10_21},
      {p11_11, p11_12, p11_13, p11_14, p11_15,
p11_16, p11_17, p11_18, p11_19, p11_20, p11_21, p11_22},
      op1, op2
    );

```

```

//stage-1 reduction

```

```

half_adder ha1 (s1p1_2, s1p0_1, p0_1, p1_1);
full_adder fa1 (s1p1_3, s1p0_2, p0_2, p1_2, p2_2);
full_adder fa2 (s1p1_4, s1p0_3, p0_3, p1_3, p2_3);
full_adder fa3 (s1p1_5, s1p0_4, p0_4, p1_4, p2_4);
full_adder fa4 (s1p1_6, s1p0_5, p0_5, p1_5, p2_5);
full_adder fa5 (s1p1_7, s1p0_6, p0_6, p1_6, p2_6);
full_adder fa6 (s1p1_8, s1p0_7, p0_7, p1_7, p2_7);
full_adder fa7 (s1p1_9, s1p0_8, p0_8, p1_8, p2_8);
full_adder fa8 (s1p1_10, s1p0_9, p0_9, p1_9, p2_9);
full_adder fa9 (s1p1_11, s1p0_10, p0_10, p1_10, p2_10);
full_adder fa10 (s1p1_12, s1p0_11, p0_11, p1_11, p2_11);
full_adder fa11 (s1p1_13, s1p0_12, p1_12, p2_12, p3_12);
full_adder fa12 (s1p4_7, s1p3_6, p4_6, p5_6, p6_6);
full_adder fa13 (s1p4_8, s1p3_7, p4_7, p5_7, p6_7);
full_adder fa14 (s1p4_9, s1p3_8, p4_8, p5_8, p6_8);
full_adder fa15 (s1p4_10, s1p3_9, p4_9, p5_9, p6_9);
full_adder fa16 (s1p4_11, s1p3_10, p4_10, p5_10, p6_10);
full_adder fa17 (s1p4_12, s1p3_11, p4_11, p5_11, p6_11);
full_adder fa18 (s1p4_13, s1p3_12, p4_12, p5_12, p6_12);
full_adder fa19 (s1p4_14, s1p3_13, p4_13, p5_13, p6_13);
full_adder fa20 (s1p4_15, s1p3_14, p4_14, p5_14, p6_14);
full_adder fa21 (s1p4_16, s1p3_15, p4_15, p5_15, p6_15);
full_adder fa22 (s1p4_17, s1p3_16, p5_16, p6_16, p7_16);
full_adder fa23 (s1p7_11, s1p6_10, p8_10, p9_10, p10_10);
full_adder fa24 (s1p7_12, s1p6_11, p8_11, p9_11, p10_11);
full_adder fa25 (s1p7_13, s1p6_12, p8_12, p9_12, p10_12);
full_adder fa26 (s1p7_14, s1p6_13, p8_13, p9_13, p10_13);
full_adder fa27 (s1p7_15, s1p6_14, p8_14, p9_14, p10_14);
full_adder fa28 (s1p7_16, s1p6_15, p8_15, p9_15, p10_15);

```

```

full_adder fa29 (s1p7_17, s1p6_16, p8_16, p9_16, p10_16);
full_adder fa30 (s1p7_18, s1p6_17, p8_17, p9_17, p10_17);
full_adder fa31 (s1p7_19, s1p6_18, p8_18, p9_18, p10_18);
full_adder fa32 (s1p7_20, s1p6_19, p8_19, p9_19, p10_19);
full_adder fa33 (s1p7_21, s1p6_20, p9_20, p10_20, p11_20);

//stage-2 reduction
half_adder ha2 (s2p1_3, s2p0_2, s1p0_2, s1p1_2);
half_adder ha3 (s2p1_13, s2p0_12, s1p0_12, s1p1_12);
half_adder ha4 (s2p3_17, s2p2_16, s1p3_16, s1p4_16);
half_adder ha5 (s2p5_21, s2p4_20, s1p6_20, s1p7_20);
full_adder fa34 (s2p1_4, s2p0_3, s1p0_3, s1p1_3, p3_3);
full_adder fa35 (s2p1_5, s2p0_4, s1p0_4, s1p1_4, p3_4);
full_adder fa36 (s2p1_6, s2p0_5, s1p0_5, s1p1_5, p3_5);
full_adder fa37 (s2p1_7, s2p0_6, s1p0_6, s1p1_6, p3_6);
full_adder fa38 (s2p1_8, s2p0_7, s1p0_7, s1p1_7, p3_7);
full_adder fa39 (s2p1_9, s2p0_8, s1p0_8, s1p1_8, p3_8);
full_adder fa40 (s2p1_10, s2p0_9, s1p0_9, s1p1_9, p3_9);
full_adder fa41 (s2p1_11, s2p0_10, s1p0_10, s1p1_10, p3_10);
full_adder fa42 (s2p1_12, s2p0_11, s1p0_11, s1p1_11, p3_11);
full_adder fa43 (s2p1_14, s2p0_13, p2_13, s1p1_13, p3_13);
full_adder fa44 (s2p3_8, s2p2_7, s1p3_7, s1p4_7, p7_7);
full_adder fa45 (s2p3_9, s2p2_8, s1p3_8, s1p4_8, p7_8);
full_adder fa46 (s2p3_10, s2p2_9, s1p3_9, s1p4_9, p7_9);
full_adder fa47 (s2p3_11, s2p2_10, s1p3_10, s1p4_10, p7_10);
full_adder fa48 (s2p3_12, s2p2_11, s1p3_11, s1p4_11, p7_11);
full_adder fa49 (s2p3_13, s2p2_12, s1p3_12, s1p4_12, p7_12);
full_adder fa50 (s2p3_14, s2p2_13, s1p3_13, s1p4_13, p7_13);
full_adder fa51 (s2p3_15, s2p2_14, s1p3_14, s1p4_14, p7_14);
full_adder fa52 (s2p3_16, s2p2_15, s1p3_15, s1p4_15, p7_15);
full_adder fa53 (s2p3_18, s2p2_17, p6_17, s1p4_17, p7_17);
full_adder fa54 (s2p5_12, s2p4_11, s1p6_11, s1p7_11, p11_11);
full_adder fa55 (s2p5_13, s2p4_12, s1p6_12, s1p7_12, p11_12);
full_adder fa56 (s2p5_14, s2p4_13, s1p6_13, s1p7_13, p11_13);
full_adder fa57 (s2p5_15, s2p4_14, s1p6_14, s1p7_14, p11_14);
full_adder fa58 (s2p5_16, s2p4_15, s1p6_15, s1p7_15, p11_15);
full_adder fa59 (s2p5_17, s2p4_16, s1p6_16, s1p7_16, p11_16);
full_adder fa60 (s2p5_18, s2p4_17, s1p6_17, s1p7_17, p11_17);
full_adder fa61 (s2p5_19, s2p4_18, s1p6_18, s1p7_18, p11_18);
full_adder fa62 (s2p5_20, s2p4_19, s1p6_19, s1p7_19, p11_19);
full_adder fa63 (s2p5_22, s2p4_21, p10_21, s1p7_21, p11_21);

//stage-3 reduction
half_adder ha6 (s3p1_4, s3p0_3, s2p0_3, s2p1_3);
half_adder ha7 (s3p3_11, s3p2_10, s2p3_10, s1p6_10);
half_adder ha8 (s3p3_12, s3p2_11, s2p3_11, s2p4_11);
half_adder ha9 (s3p3_20, s3p2_19, s2p4_19, s2p5_19);
half_adder ha10 (s3p3_21, s3p2_20, s2p4_20, s2p5_20);
half_adder ha11 (s3p3_22, s3p2_21, s2p4_21, s2p5_21);
half_adder ha12 (s3p3_23, s3p2_22, p11_22, s2p5_22);
full_adder fa64 (s3p1_5, s3p0_4, s2p0_4, s2p1_4, p4_4);

```

```

full_adder fa65 (s3p1_6, s3p0_5, s2p0_5, s2p1_5, p4_5);
full_adder fa66 (s3p1_7, s3p0_6, s2p0_6, s2p1_6, s1p3_6);
full_adder fa67 (s3p1_8, s3p0_7, s2p0_7, s2p1_7, s2p2_7);
full_adder fa68 (s3p1_9, s3p0_8, s2p0_8, s2p1_8, s2p2_8);
full_adder fa69 (s3p1_10, s3p0_9, s2p0_9, s2p1_9, s2p2_9);
full_adder fa70 (s3p1_11, s3p0_10, s2p0_10, s2p1_10, s2p2_10);
full_adder fa71 (s3p1_12, s3p0_11, s2p0_11, s2p1_11, s2p2_11);
full_adder fa72 (s3p1_13, s3p0_12, s2p0_12, s2p1_12, s2p2_12);
full_adder fa73 (s3p1_14, s3p0_13, s2p0_13, s2p1_13, s2p2_13);
full_adder fa74 (s3p1_15, s3p0_14, p3_14, s2p1_14, s2p2_14);
full_adder fa75 (s3p3_10, s3p2_9, s2p3_9, p8_9, p9_9);
full_adder fa76 (s3p3_13, s3p2_12, s2p3_12, s2p4_12, s2p5_12);
full_adder fa77 (s3p3_14, s3p2_13, s2p3_13, s2p4_13, s2p5_13);
full_adder fa78 (s3p3_15, s3p2_14, s2p3_14, s2p4_14, s2p5_14);
full_adder fa79 (s3p3_16, s3p2_15, s2p3_15, s2p4_15, s2p5_15);
full_adder fa80 (s3p3_17, s3p2_16, s2p3_16, s2p4_16, s2p5_16);
full_adder fa81 (s3p3_18, s3p2_17, s2p3_17, s2p4_17, s2p5_17);
full_adder fa99 (s3p3_19, s3p2_18, s2p3_18, s2p4_18, s2p5_18);

```

```
//stage-4 reduction
```

```

half_adder ha13 (s4p1_5, s4p0_4, s3p0_4, s3p1_4);
full_adder fa82 (s4p1_6, s4p0_5, s3p0_5, s3p1_5, p5_5);
full_adder fa83 (s4p1_9, s4p0_8, s3p0_8, s3p1_8, s2p3_8);
full_adder fa84 (s4p1_9, s4p0_8, s3p0_8, s3p1_8, s2p3_8);
full_adder fa85 (s4p1_10, s4p0_9, s3p0_9, s3p1_9, s2p3_9);
full_adder fa86 (s4p1_11, s4p0_10, s3p0_10, s3p1_10, s2p2_10);
full_adder fa87 (s4p1_12, s4p0_11, s3p0_11, s3p1_11, s2p2_11);
full_adder fa88 (s4p1_13, s4p0_12, s3p0_12, s3p1_12, s2p2_12);
full_adder fa89 (s4p1_14, s4p0_13, s3p0_13, s3p1_13, s2p2_13);
full_adder fa90 (s4p1_15, s4p0_14, s3p0_14, s3p1_14, s2p2_14);
full_adder fa91 (s4p1_16, s4p0_15, s3p1_15, s2p2_15, s2p3_15);

```

```
//stage-5 reduction
```

```

half_adder ha14 (s5p1_6, s5p0_5, s4p0_5, s4p1_5);
half_adder ha15 (s5p1_8, s5p0_7, s3p0_7, s3p1_7);
half_adder ha16 (s5p1_9, s5p0_8, s4p0_8, p8_8);
half_adder ha17 (s5p1_10, s5p0_9, s4p0_9, s4p1_9);
half_adder ha18 (s5p1_16, s5p0_15, s4p0_15, s4p1_15);
half_adder ha19 (s5p1_18, s5p0_17, s3p2_17, s3p3_17);
half_adder ha20 (s5p1_19, s5p0_18, s3p2_18, s3p3_18);
half_adder ha21 (s5p1_20, s5p0_19, s3p2_19, s3p3_19);
half_adder ha22 (s5p1_21, s5p0_20, s3p2_20, s3p3_20);
half_adder ha23 (s5p1_22, s5p0_21, s3p2_21, s3p3_21);
half_adder ha24 (s5p1_23, s5p0_22, s3p2_22, s3p3_22);
full_adder fa92 (s5p1_7, s5p0_6, s3p0_6, s4p1_6, s3p1_6);
full_adder fa93 (s5p1_11, s5p0_10, s4p0_10, s4p1_10, s3p3_10);
full_adder fa94 (s5p1_12, s5p0_11, s4p0_11, s4p1_11, s3p3_11);
full_adder fa95 (s5p1_13, s5p0_12, s4p0_12, s4p1_12, s3p3_12);
full_adder fa96 (s5p1_14, s5p0_13, s4p0_13, s4p1_13, s3p3_13);
full_adder fa97 (s5p1_15, s5p0_14, s4p0_14, s4p1_14, s3p3_14);

```

```

full_adder fa98 (s5p1_17, s5p0_16, s3p2_16, s4p1_16, s3p3_16);

//stage-6 CLA
ripple_carry18 cpa ( product[24], product[23:6],
                    {s5p0_6, s5p0_7, s5p0_8, s5p0_9, s5p0_10,
s5p0_11, s5p0_12, s5p0_13, s5p0_14, s5p0_15, s5p0_16, s5p0_17,
s5p0_18, s5p0_19, s5p0_20, s5p0_21, s5p0_22, s3p3_23},
                    {s5p1_6, s5p1_7, s5p1_8, s5p1_9, s5p1_10,
s5p1_11, s5p1_12, s5p1_13, s5p1_14, s5p1_15, s5p1_16, s5p1_17,
s5p1_18, s5p1_19, s5p1_20, s5p1_21, s5p1_22, s5p1_23},
                    1'b0);

    assign product[5] = s5p0_5;
    assign product[4] = s4p0_4;
    assign product[3] = s3p0_3;
    assign product[2] = s2p0_2;
    assign product[1] = s1p0_1;
    assign product[0] = p0_0;

endmodule

module ripple_carry18 (
    output cout,
    output [17:0] sum,
    input [0:17] a,
    input [0:17] b,
    input cin
);

    wire [16:0] w;

    full_adder fa1 ( w[0], sum[0], a[0], b[0], cin);
    full_adder fa2 ( w[1], sum[1], a[1], b[1], w[0]);
    full_adder fa3 ( w[2], sum[2], a[2], b[2], w[1]);
    full_adder fa4 ( w[3], sum[3], a[3], b[3], w[2]);
    full_adder fa5 ( w[4], sum[4], a[4], b[4], w[3]);
    full_adder fa6 ( w[5], sum[5], a[5], b[5], w[4]);
    full_adder fa7 ( w[6], sum[6], a[6], b[6], w[5]);
    full_adder fa8 ( w[7], sum[7], a[7], b[7], w[6]);
    full_adder fa9 ( w[8], sum[8], a[8], b[8], w[7]);
    full_adder fa10 ( w[9], sum[9], a[9], b[9], w[8]);
    full_adder fa11 ( w[10], sum[10], a[10], b[10], w[9]);
    full_adder fa12 ( w[11], sum[11], a[11], b[11], w[10]);
    full_adder fa13 ( w[12], sum[12], a[12], b[12], w[11]);
    full_adder fa14 ( w[13], sum[13], a[13], b[13], w[12]);
    full_adder fa15 ( w[14], sum[14], a[14], b[14], w[13]);
    full_adder fa16 ( w[15], sum[15], a[15], b[15], w[14]);
    full_adder fa17 ( w[16], sum[16], a[16], b[16], w[15]);
    full_adder fa18 ( cout, sum[17], a[17], b[17], w[16]);

```

```

endmodule

module half_adder (
    output cout,
    output sum,
    input a,
    input b
);

    and g1 ( cout, a, b );
    xor g2 ( sum, a, b );

endmodule

module full_adder (
    output cout,
    output sum,
    input a,
    input b,
    input cin
);

    // intermediate signals
    wire w1, w2, w3;

    // carry out
    and g1( w1, a, b );
    and g2( w2, b, cin );
    and g3( w3, a, cin );
    or  g4( cout, w1, w2, w3 );

    // sum
    xor g5 ( sum, a, b, cin );

endmodule

```

## References

- [1] Charles R. Baugh and Bruce A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045-1047, 1973.
- [2] K. C. Bickerstaff, Earl E. Swartzlander, and Michael J. Schulte, "Analysis of column compression multipliers," *Proceedings 15th IEEE Symposium on Computer Arithmetic*, IEEE, 2001, pp. 33-39.
- [3] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14-17, 1964.
- [4] Luigi Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, August 1965.
- [5] K. C. Bickerstaff, Michael Schulte, and E. E. Swartzlander, "Reduced area multipliers," *Proceedings International Conference on Application-Specific Array Processors*, 1993, pp. 478-489, 1993.
- [6] Wesley Chu, Ali I. Unwala, Pohan Wu, and Earl E. Swartzlander, Jr., "Implementation of a High Speed Multiplier Using Carry Lookahead Adders," *Forty-Seventh Asilomar Conference on Signals, Systems and Computers*, pp. 400-404, 2013.
- [7] O. L. MacSorley, "High-Speed Arithmetic in Binary Computers," *Proceedings of the IRE*, vol. 49, pp. 67-91, 1961.
- [8] J. Rabaey, (2003, September 12). Multiplier Design. Retrieved from <http://www.cse.psu.edu/~kxc104/class/cmpen411/16s/lec/C411L20Multiplier.pdf>
- [9] K. Bickerstaff, *Optimization of column compression multipliers*, Dissertation, The University of Texas at Austin, 2007.