An Inversion-Based Synthesis Approach for Area and Power efficient Arithmetic Sum-of-Products

Sabyasachi Das
Synplicity Inc
Sunnyvale, CA, USA
Email: sabya@synplicity.com

Sunil P. Khatri
Texas A&M University
College Station, TX, USA
Email: sunilkhatri@tamu.edu

***Abstract*— In state-of-the-art Digital Signal Processing (DSP) and Graphics applications, the arithmetic Sum-of-Product (SOP) is an important and computationally intensive operation, consuming a significant amount of area, delay and power. This paper presents a new algorithmic approach to synthesize a non-timing critical SOP block in an area-efficient and power-efficient way, which can be very useful to reduce the size and power consumption of the non timing-critical portion in the design. We have divided the problem of generating the SOP into three parts: inversion-based creation of the BitClusters (sets of individual partial-product bits, which belong to the i^{th} bitslice), propagation-based reduction of the BitClusters and selective-inversion based computation of the final sum result. Techniques used in these three steps help to reduce the implementation area and power consumption for the SOP block. Our experimental data shows that the SOP block generated by our approach is significantly smaller (8.59% on average) and marginally faster (0.42% on average) than the SOP block generated by a commercially available best-in-class datapath synthesis tool. In addition, our proposed SOP netlist consumes significantly less dynamic power (7.92% on average) and leakage power (5.65% on average) than the netlist generated by the synthesis tool. These improvements were verified on placed-and-routed designs as well.**

I. Introduction

As we migrate toward ultra deep sub-micron feature sizes, designs are becoming increasingly complex, with very aggressive optimization goals. In all circuits, some portions of the design are timing-critical and other portions are not timing-critical. It is very important to use area-efficient and power-efficient architectures in the non timing-critical portion of the chip. This would reduce the overall size and power consumption of the design, with secondary improvement in circuit delay as well. In addition, it would also provide more options to the placement and routing of the timing-critical portions of the circuit, potentially leading to improved performance.

Sum-of-Product (SOP) blocks have been extensively used in DSP and Graphics algorithms. Some of the specific applications of SOP are Multiply-Accumulation (MAC), vector quantization, computation of the euclidean distance between two points, adaptive filtering, pattern recognition, image compression, decoding, etc. Hence, an area-efficient and power-efficient SOP architecture is becoming increasingly important.

There have been several techniques proposed, which can be used to improve the area of a Sum-of-Product block. In [1], [2], [3], [4], [5]; the authors have presented different ways to use carrysave arithmetic on multiple arithmetic blocks

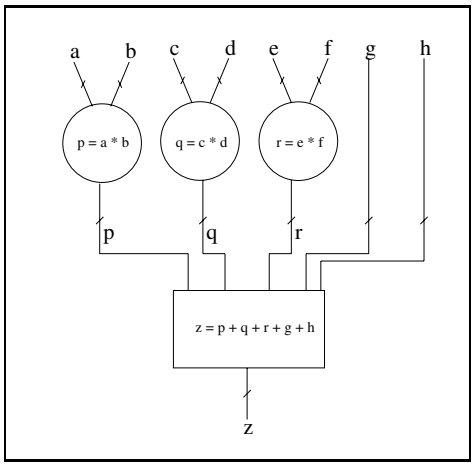

Fig. 1. Block Diagram of an 8-input Sum-of-Product (SOP) block

to design large SOP blocks. These techniques emphasizes the usefulness of SOP blocks over a collection of cascaded arithmetic blocks performing unit operations (like additions, subtractions, multiplications etc). There are several papers focusing on the generation of multiplication and addition units, which can be applied to the design of the SOP blocks also. In [6], a modified Wallace tree construction is discussed, to save most of the wasted area in the multiplier layout. In [7], a dependence graph and modified Booth algorithm is used to design a merged multiply-accumulate (MAC) hardware unit. The authors in [8] describe a split array multiplier organized in a left-to-right leapfrog structure, leading to less power consumption. In [9], [10], [11], different techniques have been proposed to reduce the partial products in multiplication and SOP units. Competitive analysis between different reduction approaches are presented in [12] and [13]. Among the adder architectures, Ripple carry adder is the smallest and it is widely used in non timing-critical path [14]. A mix of the above-mentioned architectures can be used to generate an SOP block.

In this paper, we propose a new scheme to synthesize SOP blocks in an area and power efficient manner. In our approach, we define the notion of a BitCluster for every bit in the SOP block. To generate the BitClusters for all bits, we use an inversion-based scheme. After the BitClusters are created, we perform a tree-reduction operation to reduce the BitClusters

1063-9667/08 $25.00 © 2008 IEEE
DOI 10.1109/VLSI.2008.18

to two addends. In the third step, we add the two addends by using a selective-inversion based adder to produce the output.

We have organized the rest of the paper as follows: In Section II, we present some background information. We discuss our proposed approach in Section III. The experimental setup is explained in Section IV. Section V presents the experimental results. Conclusions are drawn in Section VI.

II. Preliminaries

In this section, we briefly explain the concept of a generalized Sum-of-Product (SOP) block. The block diagram of an 8-input Sum-of-Product block is shown in the Figure 1. In this block, there are eight inputs (a, b, c, d, e, f, g and h), which produce the output z. In this SOP block, there are three product terms or multiplicative terms ($a*b$, $c*d$ and $e*f$) and two input sum terms or additive terms (g and h). A generalized Sum-of-Product block can be used to implement the addition of an arbitrary number of (including zero) product terms and sum terms. As a consequence, an SOP block is quite general.

Since a multiplier has only one product term ($a*b$) and no sum term, it can be considered as a special case of the Sum-of-Product block. On the other hand, a 2-input adder has only one sum term ($a+b$) and no product term, it can also be considered as a special case of the Sum-of-Product block. In addition to the multiplier and adder blocks, a generalized Sum-of-Product block can be used to implement the multiply-accumulator (MAC), subtractor, squarer, comparator, shared multiplier-adder, tree-of-adders or combinations thereof.

III. Our Approach

Figure 2 describes our overall flow, which consists of three steps. In the following sub-sections, we discuss each step in detail.

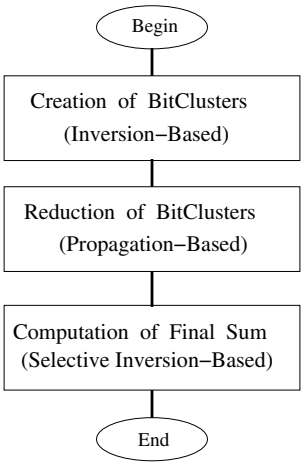

Fig. 2. Our Flow to Synthesize an Area and Power Efficient SOP Block

A. Creation of BitClusters

We define the BitCluster for the i^{th} bit as the set of individual partial-product bits, which belong to the i^{th} bitslice. To explain the creation of BitClusters, let us consider the following Sum-of-Product (SOP) block: $Z = (a*b) + (c*d)$ where a, b are 4-bit wide and c, d are 2 bit-wide each. If we denote signal a by (a_3, a_2, a_1, a_0); signal b by (b_3, b_2, b_1, b_0); signal c by (c_1, c_0) and signal d by (d_1, d_0) then, the BitClusters are:

- $BitCluster_0$ = $\{a_0 \wedge b_0,\ c_0 \wedge d_0\}$
- $BitCluster_1$ = $\{a_1 \wedge b_0,\ a_0 \wedge b_1,\ c_1 \wedge d_0,\ c_0 \wedge d_1\}$
- $BitCluster_2$ = $\{a_2 \wedge b_0,\ a_1 \wedge b_1,\ a_0 \wedge b_2,\ c_1 \wedge d_1\}$
- $BitCluster_3$ = $\{a_3 \wedge b_0,\ a_2 \wedge b_1,\ a_1 \wedge b_2,\ a_0 \wedge b_3\}$
- $BitCluster_4$ = $\{a_3 \wedge b_1,\ a_2 \wedge b_2,\ a_1 \wedge b_3\}$
- $BitCluster_5$ = $\{a_3 \wedge b_2,\ a_2 \wedge b_3\}$
- $BitCluster_6$ = $\{a_3 \wedge b_3\}$

For any given (m-bit$*n$-bit) + (p-bit$*q$-bit) Sum-of-Product, we can compute all the BitClusters by performing 2-input NAND operations between the appropriate bits of the multiplicand and multiplier in each product expression. In such an approach, we need $(m*n+p*q)$ number of 2-input NAND gates to generate max(m+n-1, p+q-1) BitClusters. In practice, due to the use of NAND gates, all the elements in the BitClusters contain a logical inversion. In CMOS technology, inverting functions (like NAND, NOR etc.) are typically more efficient in terms of area, delay and power than non-inverting functions (like AND, OR etc). We have found that all of the commercially available 0.13μ and 0.09μ technology libraries (that we have explored) have smaller and faster 2-input NAND gates than 2-input AND gates. Throughout the rest of this section, we denote the total number of BitClusters as N. In Algorithm 1, we present the way to create the BitClusters.

B. Reduction of BitClusters

After generating the BitClusters, most of the BitClusters contain more than two elements. For an SOP which implements the expression $a*b+c*d+e*f+g+h$ (and a, b, c, d, e, f, g and h all have the same bit-width); all the N BitClusters have more than two elements. In this step, we reduce each BitCluster to a maximum of two elements.

For the reduction of partial products, two techniques proposed in the context of multipliers are *Wallace Tree* [9] and *Dadda Tree* [10] reduction schemes. In these approaches, an n-input *Wallace Tree* or *Dadda Tree* reduces its k-bit inputs to two (k+$\log_2 n$-1)-bit outputs. In the *Wallace Tree*, the number of operands are reduced at the earliest opportunity. On the other hand, in the *Dadda Tree*, the number of operands are reduced in a more area-efficient way without any significant impact in the delay of the reduction tree. The authors of [12], [13] present a comparative study between different reduction techniques.

All these techniques use different types of *counters*. A *(p:q) counter* is defined as a functional block, which adds its p single-bit inputs and produces q single-bit outputs; where p and q satisfy the following equation:

$$q = \lfloor log_2 p + 1 \rfloor$$

In our approach for area and power efficient SOP block, we use the Dadda-tree reduction scheme with a modified (3:2) and (2:2) counters. In the *Reduction of BitClusters* phase, each BitCluster would possibly need multiple modified (3:2)

Algorithm 1 :Creation of all the BitClusters

N = Total number of BitClusters in SOP

// Loop-1: Initialize All BitClusters to NULL
for i = 0 to $(N-1)$ **do**
  $BitCluster_i = \{NULL\}$
end for

// Loop-2: Compute All BitClusters
// (for each multiplicative-term or product-term)
for (Each Multiplicative term in the SOP expression) **do**
  // (assume that the expression is $m1 * m2$.)
  $w1$ = Width($m1$)
  $w2$ = Width($m2$)
  for k = 0 to $(w1-1)$ **do**
    for l = 0 to $(w2-1)$ **do**
      $BitCluster_{(k+l)} = \{BitCluster_{k+l} \vee \overline{(m1_k \wedge m2_l)}\}$
    end for
  end for
end for

// Loop-3: Update All BitClusters
// (for each additive-term or sum-term)
for (Each Additive term in the SOP expression) **do**
  // (assume that the additive expression is $m3$.)
  $w3$ = Width($m3$)
  for k = 0 to $(w3-1)$ **do**
    $BitCluster_{(k)}$ = $\{BitCluster_k \vee \overline{m3_k}\}$
  end for
end for

return all (N) BitClusters

and (2:2) counters. After generating the BitClusters, all the elements in the BitClusters contain a logical inversion (due to the presence of the NAND gate in the BitCluster Creation phase). To ensure that the outputs at the end of the reduction of BitClusters also contain the logical inversion, we modify the functionality of the (3:2) and (2:2) counters. A modified (3:2) counter accepts 3 inputs signals (x_i, y_i and $carry_i$) belonging to i^{th} column (BitCluster) in the partial-products and produces 1 output signal (sum_i) for the i^{th} column (BitCluster) and 1 more output signal ($carry_{i+1}$) for the $(i+1)^{th}$ BitCluster. The functionality of the sum_i and $carry_{i+1}$ of our modified (3:2) counter is written as:

- $sum_i = \overline{x_i \odot y_i \odot carry_i}$
  where $\odot$ represents a 2-input XNOR gate
- $carry_{i+1} = (x_i \vee y_i) \wedge (y_i \vee carry_i) \wedge (carry_i \vee x_i)$

Similarly, a modified (2:2) counter accepts 2 inputs signals (x_i and y_i) belonging to i^{th} column (BitCluster) in the partial-products and produces 1 output signal (sum_i) for the i^{th} column (BitCluster) and 1 more output signal ($carry_{i+1}$) for the $(i+1)^{th}$ BitCluster. The functionality of the sum_i and $carry_{i+1}$ of our modified (2:2) counter is written as:

- $sum_i = x_i \odot y_i$
- $carry_{i+1} = (x_i \vee y_i)$

In each BitCluster of the reduction-tree, we are able to use instantiations of the same counter structure. The sum_i output of each counter in $BitCluster_i$ gets fed to either another counter in the same $BitCluster_i$ or to the final adder stage (described in the next section of this paper). The $carry_{i+1}$ output of each counter in $BitCluster_i$ gets fed to either another counter in $BitCluster_{i+1}$, or to the final adder stage. At the output of the final level in each BitCluster, the inverted result is produced, and would get fed to the final adder stage. In this way, the reduction-tree structure $propagates$ the inversion property to the final stage of the SOP block.

C. Computation of Final Sum

Since the Sum-of-Product circuit needs to present the final result in the single binary vector format, all the BitClusters have to be added (with the inversion property taken care of) by a final carry propagation adder circuit. After performing the reduction of the BitClusters, each BitCluster consists of ≤ 2 elements. Hence, we can view the N vertical BitClusters as two horizontal vectors or operands, each having N elements. Therefore, the problem of final addition of BitClusters gets converted to the problem of a specialized 2-operand addition. In this addition, the inputs are two inverted vectors of width N bits and the output is one non inverted vector of width $(N+1)$ bits. In this section, we refer to these two operands as

- vector x $(x_{N-1},\ x_{N-2},\ ...,\ x_1,\ x_0)$ and
- vector y $(y_{N-1},\ y_{N-2},\ ...,\ y_1,\ y_0)$

To have an area and power efficient SOP implementation in the non-timing-critical path of a design, we definitely need to use a low-area architecture for the final carry propagate addition. In datapath designs, the ripple-carry architecture is very useful in non-timing-critical portions of the design (if it can satisfy the timing requirement of the off-critical path). Our final adder is a modified version of the ripple-carry addition scheme. In our final adder, every bit (i) has a modified-full-adder cell; which takes 3 inputs (x_i, y_i and $carry_i$) and produces two outputs (sum_i and $carry_{i+1}$). The sum_i output from every modified-full-adder cell would have the correct polarity (non-inverted) and is the final result for the i^{th} bit position. On the other hand, $carry_{i+1}$ remains in the $inverted$ state. The Boolean expressions for the functionality of the modified-full-adder cell is the following:

- $sum_i = \overline{x_i \oplus y_i \oplus carry_i}$
- $carry_{i+1} = \overline{(x_i \wedge y_i)} \wedge \overline{(y_i \wedge carry_i)} \wedge \overline{(carry_i \wedge x_i)}$

Based on the De Morgan's law, we note that the above-mentioned equation for $carry_{i+1}$ is identical to the $carry_{i+1}$ output of a traditional (3:2) counter.

If a particular bit has less than three elements to be fed to the modified-full-adder cell, then the remaining input pins of the modified-full-adder cell need to be tied to the global logic-1 signal. This is due to the fact that all the inputs of the

modified-full-adder are in inverted state. This situation could happen frequently in the least significant bit of the SOP.

The algorithm for the Computation of the Final Sum is presented in Algorithm 2.

Algorithm 2 :Computation of Final Sum

for i = 0 to $(N-1)$ **do**
  // Handle all the non-existent elements
  if x_i does not exist **then**
    $x_i = 1'b1$
  end if
  if y_i does not exist **then**
    $y_i = 1'b1$
  end if
  if $carry_i$ does not exist **then**
    $carry_i = 1'b1$
  end if

  // Perform the addition in the i^{th} bit
  Instantiate modified-full-adder cell with three inputs x_i, y_i, $carry_i$ and two outputs sum_i and $carry_{i+1}$
end for
sum_N = $carry_N$

return the $(N+1)$-bit wide sum vector

IV. Experimental Setup

We implemented our proposed approach in the C++ programming language. The experiments were performed with different Sum-of-Product RTL designs written in the Verilog hardware description language.

To collect different data-points regarding the quality of results for the Sum-of-Product blocks in the non timing-critical portion of the design, we used the following variations:

- Multiple types of Sum-of-Product designs of different expressions and input bit-widths:
  In the Table I, we report different configurations of the designs that are used in our experiments. Following is a brief description of the different SOP blocks presented in the Table I.
  - Two multiplier blocks ($Z = (a * b)$) having different bit-widths. We refer to these as Mult-1 and Mult-2.
  - Two Multiply-Accumulate blocks ($Z = (a * b) + c$). We refer to these designs as Mac-1 and Mac-2.
  - Two general SOP blocks. The block Sop-1 represents the functionality of $Z = (a*b)+(c*d)$ and the block Sop-2 represents the functionality of $Z = (a * b) + (c * d) + e$.
  - Two Squarer blocks ($Z = (a * a)$). We refer to these blocks as Sqr-1 and Sqr-2.
- The different technologies and libraries, we used are:
  - Two libraries (L_1 and L_2) for 0.13μ technology.
  - Two libraries (L_3 and L_4) for 0.09μ technology.

Name of the Sum-of-Product (SOP) Block	Widths of the Input Signals of the SOP Block	Width of the Output Signal of the SOP
Mult-1	16 , 16	32
Mult-2	24 , 31	55
Mac-1	32, 32, 32	64
Mac-2	28, 24, 32	52
Sop-1	34, 35, 23, 28	69
Sop-2	16, 23, 21, 17, 31	39
Sqr-1	25	50
Sqr-2	18	36

TABLE I
Characteristics of Different Sum-of-Product Blocks

- Different input arrival time constraints:
  To facilitate the explanation, let us assume that the expression of the SOP is $Z = (a * b) + (c * d)$ and each of the four input signals is n-bit wide. We have used the following types of input arrival time constraints:
  - All input bits of all the signals arrive at the same time. We refer to this constraint as Type-A. If we denote $Arr(a_i)$ as the arrival time of the bit a_i and if k is a constant number, then this Type-A constraint can be represented as:
    $Arr(a_i) = k; \quad 0 \le i < n$
    $Arr(b_i) = k; \quad 0 \le i < n$
    $Arr(c_i) = k; \quad 0 \le i < n$
    $Arr(d_i) = k; \quad 0 \le i < n$
    This category represents the actual timing situations if the SOP block is placed immediately after a register-bank or the primary inputs of the design are fed to the SOP block.
  - Different input bits arrive at different times. We refer to this category of timing constraints as Type-B. We believe that this category represents the actual timing situations in most of the Sum-of-Product blocks in real-life designs. Assuming that k is a constant number and δ is the delay of the fastest 2-input AND-gate in the given technology library, the following are some specific examples of the Type-B timing constraints. Here we have explained the arrival times for signal a_i. Similar expressions for arrival times applies to all the bits of signals b, c and d as well.
    1) $Arr(a_i) = i * k * \delta; \quad 0 \le i < n$
    2) $Arr(a_i) = i^2 k\delta; \quad 0 \le i < n$
    3) $Arr(a_i) = 0; \quad 0 \le i < \lceil n/2 \rceil$
       $Arr(a_i) = k\delta; \quad \lceil n/2 \rceil \le i < n$
    4) $Arr(a_i) = 0; \quad 0 \le i < \lceil n/4 \rceil$
       $Arr(a_i) = k\delta; \quad \lceil n/4 \rceil \le i < \lceil n/2 \rceil$
       $Arr(a_i) = 2k\delta; \quad \lceil n/2 \rceil \le i < \lceil 3n/4 \rceil$
       $Arr(a_i) = 3k\delta; \quad \lceil 3n/4 \rceil \le i < n$
    5) $Arr(a_i) = 0; \quad 0 \le i < \lceil n/4 \rceil$
       $Arr(a_i) = ik\delta; \quad \lceil n/4 \rceil \le i < \lceil n/2 \rceil$
       $Arr(a_i) = 2ik\delta; \quad \lceil n/2 \rceil \le i < \lceil 3n/4 \rceil$
       $Arr(a_i) = 3ik\delta; \quad \lceil 3n/4 \rceil \le i < n$

			Area (μ^2)			Worst-case Delay (ps)		
Design Name	Technology Library	Timing Constraint	Commercial Tool	Our Approach	(%) Improvement	Commercial Tool	Our Approach	(%) Improvement
Mult-1	L_1	Type-A	2697	2472	8.34%	2148	2137	0.51%
Mult-2	L_1	Type-A	4619	4268	7.59%	2671	2646	0.94%
Mac-1	L_1	Type-A	5281	4983	5.63%	2836	2860	-0.84%
Mac-2	L_1	Type-A	4306	3959	8.07%	2792	2761	1.11%
Sop-1	L_1	Type-A	10743	9892	7.92%	3018	2982	1.19%
Sop-2	L_1	Type-A	6937	6343	8.56%	2675	2651	0.89%
Sqr-1	L_1	Type-A	2368	2151	9.16%	1864	1849	0.81%
Sqr-2	L_1	Type-A	1873	1710	8.65%	1753	1738	0.86%
Mult-1	L_3	Type-A	2953	2683	9.14%	1582	1573	0.69%
Mult-2	L_3	Type-A	5816	5372	7.62%	1739	1758	-1.09%
Mac-1	L_3	Type-A	6372	5826	8.57%	1951	1934	0.87%
Mac-2	L_3	Type-A	4901	4421	9.81%	1814	1829	-0.83%
Sop-1	L_3	Type-A	11694	10493	10.26%	2139	2126	0.61%
Sop-2	L_3	Type-A	8165	7401	9.36%	1863	1882	-1.02%
Sqr-1	L_3	Type-A	2687	2419	9.94%	1481	1453	1.89%
Sqr-2	L_3	Type-A	2342	2092	10.68%	1107	1084	2.07%
Mult-1	L_1	Type-B1	2103	1959	6.82%	2762	2735	0.97%
Mult-2	L_1	Type-B1	4592	4248	7.49%	2918	2874	1.51%
Mac-1	L_1	Type-B1	5286	4915	7.03%	3859	3841	0.46%
Mac-2	L_1	Type-B1	3741	3421	8.56%	3724	3759	-0.94%
Sop-1	L_1	Type-B1	9837	9036	8.14%	4206	4238	-0.76%
Sop-2	L_1	Type-B1	6372	5897	7.45%	3562	3540	0.61%
Sqr-1	L_1	Type-B1	1983	1812	8.62%	2844	2829	0.53%
Sqr-2	L_1	Type-B1	1562	1448	7.28%	2181	2164	0.78%
Mult-1	L_3	Type-B1	2972	2664	10.41%	1702	1737	-2.06%
Mult-2	L_3	Type-B1	4619	4213	8.79%	1867	1851	0.86%
Mac-1	L_3	Type-B1	5384	4937	8.30%	2914	2896	0.69%
Mac-2	L_3	Type-B1	4063	3652	10.12%	2876	2829	1.63%
Sop-1	L_3	Type-B1	8745	7946	9.14%	3285	3258	0.82%
Sop-2	L_3	Type-B1	5691	5209	8.47%	2973	2961	0.40%
Sqr-1	L_3	Type-B1	2284	2048	10.28%	2049	2084	-1.71%
Sqr-2	L_3	Type-B1	1956	1781	8.92%	1631	1612	1.16%
Average					8.59%			0.42%

TABLE II

AREA AND DELAY COMPARISON OF SUM-OF-PRODUCT BLOCKS GENERATED BY A COMMERCIAL SYNTHESIS TOOL AND BY OUR APPROACH

V. EXPERIMENTAL RESULTS

We compared the netlist produced by our approach against the output netlist of a commercially available best-in-class datapath synthesis tool. The synthesis tool generates arithmetic-optimized architectures for all the arithmetic blocks (like sum-of-products) and then it performs general-purpose operations like technology-independent optimizations, constant propagation, redundancy removal, technology mapping, timing-driven optimization, area-driven optimization, low-power optimization etc. While running the synthesis tool, we turned on all the above-mentioned optimizations. In the Table II and Table III, we report 32 sets of data-points (worst-case delay, total area, dynamic and leakage power consumption) involving SOPs having different widths and expressions, timing constraints and technology libraries. If we compute the average of all the 32 data-points in the Table II and compare our results with the results produced by the implementation of the commercial datapath synthesis tool, we see a 8.59% area savings in the SOP block (column 6 of Table II) with a marginal 0.42% speed improvement (column 9 of Table II). Similarly, the average dynamic and leakage power consumption of our SOP block is significantly less (7.92% for dynamic power and 5.65% for leakage power) than that of the SOP produced by the synthesis tool (columns 6 and 9 in the Table III).

We observe that in 8 cases, the delay of our SOP is slightly worse than the baseline. As expected, in each of these cases, the area and the power of our SOP is much better than the baseline. Since our approach is designed to be used in the area-critical portions of the design, savings in area and power are considered to be the primary goal and the blocks do not go though rigorous timing optimization phase. As a result, a marginal degradation in timing is not considered significant. Similarly, the slight improvement in timing in all the other 24 cases are also considered insignificant.

To keep the sizes of the Table II and Table III relatively brief, we did not report the results for all possible combinations of designs, timing constraints and technology libraries. Note that the results in each of the combinations, which are not reported here also supported our conclusion that, the proposed approach produces area and power efficient SOP blocks.

To verify the correlation of post-synthesis experimental data with the post place-and-route data, we performed placement and routing on Mult-1 and Mac-1. For these two testcases, the average post-routing total area of the SOP block generated by our proposed approach is 0.89 (normalized to the total area of the SOP generated by the commercial synthesis tool). Similarly, the post-routing total power consumption of the SOP block generated by our technique is 0.91 (normalized to the total power of the SOP generated by the synthesis tool). In addition, the post-routing worst delay of the SOP generated by the synthesis tool and our techniques are comparable. The individual results for the Mult-1 and Mac-1 designs correlate with the post-synthesis numbers reported in the Table II and Table III. These post-routing data confirm our conclusion about the area and power efficiency of our approach.

With this observation, we conclude that our area and power improvement is consistent across multiple types of SOPs,

			Dynamic Power (μW)			Leakage Power (μW)		
Design Name	Technology Library	Timing Constraint	Commercial Tool	Our Approach	(%) Improvement	Commercial Tool	Our Approach	(%) Improvement
Mult-1	L_1	Type-A	42	38	9.52%	9	9	0.00%
Mult-2	L_1	Type-A	76	71	6.57%	18	17	5.55%
Mac-1	L_1	Type-A	83	80	3.61%	19	17	10.53%
Mac-2	L_1	Type-A	68	64	5.88%	15	15	0.00%
Sop-1	L_1	Type-A	164	157	4.27%	34	32	5.89%
Sop-2	L_1	Type-A	107	99	7.48%	26	23	11.54%
Sqr-1	L_1	Type-A	30	30	0.00%	6	6	0.00%
Sqr-2	L_1	Type-A	29	27	6.90%	6	5	16.66%
Mult-1	L_3	Type-A	69	61	11.60%	21	20	4.76%
Mult-2	L_3	Type-A	114	103	9.65%	34	31	8.82%
Mac-1	L_3	Type-A	136	122	10.29%	40	37	7.50%
Mac-2	L_3	Type-A	97	94	3.08%	29	28	3.44%
Sop-1	L_3	Type-A	213	197	7.51%	63	63	0.00%
Sop-2	L_3	Type-A	165	146	11.52%	52	49	5.77%
Sqr-1	L_3	Type-A	51	45	11.76%	16	15	6.25%
Sqr-2	L_3	Type-A	42	39	7.14%	11	11	0.00%
Mult-1	L_1	Type-B1	37	34	8.11%	8	7	12.50%
Mult-2	L_1	Type-B1	68	63	7.35%	14	13	7.14%
Mac-1	L_1	Type-B1	71	66	7.04%	15	15	0.00%
Mac-2	L_1	Type-B1	67	59	11.94%	15	13	13.33%
Sop-1	L_1	Type-B1	146	138	5.48%	29	28	3.45%
Sop-2	L_1	Type-B1	95	90	5.26%	18	16	11.11%
Sqr-1	L_1	Type-B1	26	25	3.84%	5	5	0.00%
Sqr-2	L_1	Type-B1	21	19	9.52%	3	3	0.00%
Mult-1	L_3	Type-B1	62	55	11.29%	19	17	10.52%
Mult-2	L_3	Type-B1	99	91	8.08%	28	25	10.71%
Mac-1	L_3	Type-B1	138	121	12.32%	41	39	4.87%
Mac-2	L_3	Type-B1	94	85	9.57%	27	26	3.70%
Sop-1	L_3	Type-B1	205	187	8.78%	63	60	4.76%
Sop-2	L_3	Type-B1	147	136	7.53%	46	45	2.17%
Sqr-1	L_3	Type-B1	48	43	10.42%	11	11	0.00%
Sqr-2	L_3	Type-B1	39	35	10.25%	10	9	10.00%
Average					7.92%			5.65%

TABLE III

POWER COMPARISON OF SUM-OF-PRODUCT BLOCKS GENERATED BY A COMMERCIAL SYNTHESIS TOOL AND BY OUR APPROACH

timing constraints and technology libraries. This underscores the strength of our approach. Since the SOP is a very area and power intensive block, we believe that the non-timing critical portions of many real-life datapath designs can significantly benefit from our approach.

VI. CONCLUSION

In this paper, we have presented a new approach for implementing an area and power efficient sum-of-product (SOP) block, which would be very useful in the non timing-critical portion of the design. Our inversion and propagation based approach works seamlessly with different types of SOP blocks, and across different technology libraries (0.13μ, 0.09μ). Our experimental data shows that the SOP block generated by our approach is significantly smaller (8.59% on average) and marginally faster (0.42% on average) than the Sum-of-Product block generated by a commercially available best-in-class datapath synthesis tool. In addition, our proposed Sum-of-Product netlist consumes significantly less dynamic power (7.92% on average) and leakage power (5.65% on average) than the netlist generated by the datapath synthesis tool.

REFERENCES

[1] T. Kim, W. Jao, S. Jjiang. "Circuit optimization using carry-save-adder cells," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-17*, pp. 974–984, 1998.

[2] A. Mathur, S. Saluja, "Improved merging of datapath operators using information content and required precision analysis," in *Proceedings of IEEE* 38^{th} *Conference on Design Automation*, pp. 462–467, 2001.

[3] A. K. Verma, P. Ienne. "Improved Use of the Carry-Save Representation for the Synthesis of Complex Arithmetic Circuits," in *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pp.791-798, 2004.

[4] A. Fayed, W. Elgharbawy, M. Bayoumi. "A data merging technique for high-speed low-power multiply accumulate units," in *Proceedings of IEEE Internation Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 145–148, 2004.

[5] P. F. Stelling, V. G. Oklobdzija, "Implementing Multiply-Accumulate Operation in Multiplication Time," in *Proceedings of* 13^{th} *IEEE Symposium on Computer Arithmetic* pp. 99, 1997.

[6] N. Itoh, Y. Tsukamoto, T. Shibagaki, K. Nii, H. Takata, H. Makino, "A 32/spl times/24-bit multiplier-accumulator with advanced rectangular styled Wallace-tree structure," in *IEEE International Symposium on Circuits and Systems*, pp. 73-76 vol. 1, 2005.

[7] F. Elguibaly, "A fast parallel multiplier-accumulator using the modified Booth Algorithm," in *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol: 47(9), pp. 902–908, 2000.

[8] Z. Huang, M. D. Ercegovac, "High-performance low-power left-to-right array multiplier design," in *IEEE Transactions on Computers*, vol: 54, issue: 3, pp. 272-283, 2005

[9] C. S. Wallace, "A suggestion for a fast multiplier," in *IEEE Transactions on Electronic Computers*, EC-13(2):14-17, 1964.

[10] L. Dadda, "Some schemes for parallel multipliers," in *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[11] V. G. Oklobdzija, D. Villeger, "Improving multiplier design by using improved column compressiontree and optimized final adder in CMOS technology," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, issue. 2, pp. 292-301, 1995.

[12] T. Whitney, S. Earl, A. Jacob, "A comparison of Dadda and Wallace multiplier delays," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII. Edited by Luk, Franklin T. Proceedings of the SPIE*, vol. 5205, pp. 552-560, 2003.

[13] K. C. Bickerstaff, E. E. Swartzlander, M. J. Schulte, "Analysis of column compression multipliers," in *Proceedings of* 15^{th} *IEEE Symposium on Computer Arithmetic*, pp. 33–39, 2001.

[14] M. D. Ercegovac, T. Lang. "Digital Arithmetic," Morgan Kaufmann Publishers, San Francisco, 2004.