A Timing-Driven Synthesis Approach of a Fast Four-Stage Hybrid Adder in Sum-of-Products

Sabyasachi Das
University of Colorado
Boulder, CO, USA
Email: sabyasac@colorado.edu

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

Abstract— **In state-of-the-art integrated circuits, the arithmetic Sum-of-Products (SOP) is an important and computationally intensive unit, which tend to be in the timing-critical path of the design. Several arithmetic blocks like multipliers, multiply-accumulators (MAC), squarers etc. are special cases of the generalized Sum-of-Product block. The final carry propagate adder inside a Sum-of-Product block consumes about 30%-40% of the total delay of the SOP block and hence plays an important role in determining the performance of the overall design. In this paper, we present a novel approach to develop a fast implementation for the final adder block in a Sum-of-Product module. In our approach, we design a hybrid adder, which consists of four different sub-adders. The width of each of the sub-adders are computed based on the arrival times of the input signals to the hybrid adder. We have tested our approach using a variety of SOP blocks implemented under varying timing constraints and technology libraries. Experimental results demonstrate that our proposed solution is 14.31% faster than the corresponding block generated by a commercially available best-in-class datapath synthesis tool.**

I. Introduction

As VLSI circuits continue to migrate toward ultra deep sub micron feature sizes, the complexity of these designs continues to increase. Sum of Product (SOP) block [1] is one of the most commonly encountered timing-critical arithmetic blocks in communication, multimedia and graphic applications. Each SOP block contains a final carry propagate adder (a two-operand adder), which contributes about 30%-40% to the total delay of the SOP block [10]. Hence, it is critically important to develop more delay-efficient implementations of the carry propagate adder, which would reduce the delay of these SOP blocks and thereby improve the performance of the chip.

Several techniques have been proposed to design fast adder circuits. Carry lookahead adders based on the parallel prefix computation methods are relatively faster. The Kogge-Stone (KS) approach [2] produces a delay-efficient parallel prefix adder. The Brent-Kung (BK) approach [3] produces an area-efficient parallel prefix adder. A zero-deficiency prefix adder with minimal depth was introduced in [4]. In [5], the authors present new algorithms to construct a class of depth-size optimal parallel prefix circuits. In [6], one-logic level is saved leading to faster performance. In [7], analysis was performed between different prefix adders. An algorithmic approach to generate an irregular parallel prefix adder is discussed in [8]. Domino logic is used to generate an efficient parallel prefix architecture in [9]. All these adders are designed for stand-alone operation and are not very efficient in an SOP block.

In this paper, we propose a new timing-driven approach to synthesize a fast adder which is present in a Sum-of-Product block. This adder is hybrid and consists of four sub-adders having different architectures. We exploit available knowledge about the arrival times of the input signals of the adder, and compute the size of each of the four sub-adders accordingly, so that the overall delay of the hybrid adder is minimized.

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

II. Preliminaries

In this section, we explain the concepts of an arithmetic *Sum-of-Product*. A generalized SOP block is expressed as:

$$z = \sum_{i=1}^{n}(a_i * b_i) + \sum_{j=1}^{m} c_j,$$

where a_i, b_i, c_j are arithmetic operands.

In this SOP block, there are n product terms ($a_i * b_i$) and m sum terms (c_j). A Sum-of-Product block can be used to implement the addition of an arbitrary number of (including zero) product terms and sum terms. An SOP block can be used to implement multiplier, multiply-accumulator (MAC), squarer, comparator, tree-of-adders or combinations thereof.

Once the SOP block is synthesized, the resulting netlist consists of the following three parts: i) Partial Product Generator, ii) Partial Product Reduction Tree and iii) Carry Propagate Adder. Since carry propagation is an expensive operation, this 2-input adder contributes a significant amount of delay (30%-40%) to the total delay of the SOP block. Hence a technique to reduce the delay of this addition sub-block becomes quite critical to improve the performance of the SOP module.

III. Our Approach

To facilitate the explanation of our approach, we use the example of the final carry propagate hybrid adder in a multiplier. Note that our approach is applicable to the hybrid adders of any Sum-of-Product block. Throughout the rest of this paper, we assume that the partial product reduction tree produces two n-bit wide output vectors x and y, which become the inputs to the final carry propagate hybrid adder (CPA).

Due to the inherent tree-like structure of the column-compression operation, the partial product reduction tree produces a skewed timing profile at the outputs of the reduction tree [10]. This results in a non-uniform input arrival time profile for the final carry propagate adder. Because of this skewed pattern of arrival-times to the carry propagate adder, several traditional stand-alone timing-driven addition schemes do not work well in the context of Sum-of-Products.

Our proposed synthesized hybrid adder module consists of four sub-adder blocks. These are (from LSB to MSB):

- SubAdder$_1$: A slow ripple adder for w_1 bits near the LSB.
- SubAdder$_2$: A fast *Kogge-Stone* adder for next w_2 bits.
- SubAdder$_3$: A carry-select adder for next set of w_3 bits.
- SubAdder$_4$: Another carry-select adder for the remaining w_4 bits near the MSB. In our implementation for SubAdder$_3$ and SubAdder$_4$, we use 2 copies of the *Brent-Kung* architecture (one with $carry_{in}$=$1'b0$ and the other one with $carry_{in}$=$1'b1$).

In the following sub-sections, we discuss the techniques to generate each of the sub-adders in detail.

A. Determination of the width of the SubAdder$_1$

In our proposed hybrid adder, the sub-adder for the least significant w_1 bits is slow ripple architecture because these bits arrive relatively early. The timing-skew pattern of the inputs to the hybrid adder shows that the arrival times of the bits near the LSB are less than those of the bits in the middle [10]. We exploit this skew and ensure that the ripple adder sub-block produces the output carry to the faster Kogge-Stone adder sub-block (SubAdder$_2$) *before* the middle bits arrive.

To decide the bit-width of the ripple carry adder, we use the following approach. We analyze the technology library cells and identify a Full-Adder cell having the least amount of delay from the input to the carry$_{out}$ pin. This Full-Adder cell and its associated input to carry$_{out}$ delay (denoted as δ) is used throughout the algorithm. We start the timing analysis from the LSB (0^{th} bit) and test if the Full-Adder cell would produce the carry$_{out}$ (which becomes carry$_{in}$ to the 1^{st} bit) before the latest arriving signals among x_1 and y_1 arrive. If the result of the test is true, then we include the 0^{th} bit in the ripple carry adder. This test is carried out for the 0^{th}, 1^{st}, 2^{nd}, 3^{rd} bits and so on. In addition, if for some bit (let's say, the i^{th} bit), the timing analysis shows that the carry$_{out}$ (or the carry$_{in}$ to the $(i+1)^{th}$ bit) is generated *after* the latest arriving signals of bit $(i+1)$ are ready, then it is still possible that we need to include the i^{th} bit in the ripple carry adder. This is because the increase in arrival time for the x_i and y_i signals does not have a constant slope as i increases towards the middle bits [10]. To reduce risk of accepting a locally optimal solution, we perform *hill climbing*. Instead of quitting immediately, we analyze upto 2 additional bits (i.e, a total of 3 bits i, i+1 and i+2). If in *all* these 3 bits, the analysis confirms that the carry$_{out}$ is generated later than the latest arriving input signal of the next bit, then our algorithm quits and returns the value i as the width of the ripple carry adder. On the other hand, if during the 3 bits of hill climbing, the analysis in any bit j indicates that the carry$_{out}$ is generated *earlier* than the latest arriving input signal in the next bit, then we include bit j in the ripple carry adder block. In such a situation, we reset our hill climbing mode and switch back to the original mode and continue our algorithm from the next most significant bit. Our experiments with different designs and timing profiles have proved the usefulness of the hill-climbing phase.

B. Determination of the width of the SubAdder$_2$

After determining that the ripple adder (near the LSB) should be w_1 bit wide, the next set of w_2 bits are added by a fast Kogge-Stone (KS) sub-adder block. This KS architecture is the fastest possible implementation of the parallel-prefix computation when only two-input blocks are allowed.

This scheme of using the fast SubAdder$_2$ and the subsequent carry-select adders (SubAdder$_3$ and SubAdder$_4$) is very useful in reducing the overall delay of the adder. However, if the widths of the SubAdder$_3$ and SubAdder$_4$ are small, then it is better to use a single (n-w_1) bit adder for SubAdder$_2$. Hence if (n-w_1) is less than 8 bits, we use a (n-w_1) bits wide SubAdder$_2$ implemented using Kogge-Stone architecture.

On the other hand, when (n-w_1) is greater than 8, then we want to use w_2-bit wide Kogge-Stone adder for SubAdder$_2$ (in addition to the carry-select adders in subsequent addition subblocks). To maximize the timing improvement from the SubAdder$_2$, we keep the width of this adder as a power of 2. This is because the Kogge-Stone architecture suggests that the number of levels of the prefix computation cells will be identical for any adder of width between (2^k+1) and 2^{k+1}.

We compute w_2 based on the following equation:

- $w_2 = n\text{-}w_1$ for $(n\text{-}w_1)\leq 8$ (1)
- $w_2 = 2^p$ where p = $\lfloor log_2(n-w_1)\rfloor$ for $(n\text{-}w_1)>8$ (2)

By using the Kogge-Stone adder architecture we ensure that most of the carry outputs (for critical bits) of the parallel-prefix computation graph go through exactly the same number of prefix computation cells, which results in a reduced output skew. Since the *middle bits* of the combined hybrid adder have the largest arrival time, these signals play the most important role in determining the delay of the overall hybrid adder module. Our usage of the Kogge-Stone adder for SubAdder$_2$ reduces the overall delay of the hybrid adder block.

C. Determination of the widths of SubAdder$_3$ and SubAdder$_4$

After determining that the ripple carry adder (near the LSB) is w_1 bits wide and the Kogge-Stone adder is w_2 bits wide, the remaining (n-w_1-w_2) bits need to be split between the remaining two Brent-Kung (BK) carry-select adders (SubAdder$_3$ and SubAdder$_4$) near the most significant bit (MSB).

We need to determine the arrival-time of the carry$_{in}$ signal of SubAdder$_3$. Using the input arrival-times and the fastest carry operator delay, we estimate the following numbers:

1) The time when sum output will be ready at the most critical sum pin of SubAdder$_2$. We refer to this as T2$_s$.
2) The time when the $carry_{out}$ of SubAdder$_2$ will be ready to be fed to SubAdder$_3$. We refer to this as T2$_c$.

In our algorithm to identify the widths of SubAdder$_3$ and SubAdder$_4$, we *analyze* several configurations of these two adder blocks and then select the best combination of widths, such that the most critical sum outputs of SubAdder$_3$ and SubAdder$_4$ become available at reasonably close time. This time should be close to the time T2$_s$ when the most critical sum output of the SubAdder$_2$ module becomes available.

Let us assume that a configuration q indicates that bit-width of SubAdder$_3$ is q bits and the bit-width of the SubAdder$_4$ is (n-w_1-w_2-q) bits. We start our algorithm with the configuration c_1, where c_1 = $\lceil(n$-w_1-$w_2)/2\rceil$. In other words, configuration c_1 means the width of the SubAdder$_3$ is $\lceil(n$-w_1-$w_2)/2\rceil$ bits and the width of the SubAdder$_4$ is $\lfloor(n$-w_1-$w_2)/2\rfloor$ bits.

Now, we have to *estimate* the delay of this configuration. Performing a timing-driven analysis based on the arrival times of all the input signals and the delay of the fastest carry operator, we can estimate the following timing numbers:

1) The time when sum output will be ready at the most critical sum pin of SubAdder$_3$. We refer to this as T3$_s$.
2) The time when $carry_{out}$ of SubAdder$_3$ block will be ready to be fed to SubAdder$_4$. We refer to this as T3$_c$.
3) The time when sum output will be ready at the most critical sum pin of SubAdder$_4$. We refer to this as T4$_s$.
4) The time when the $carry_{out}$ of SubAdder$_4$ block will be ready. We refer to this as T4$_c$. This $carry_{out}$ bit is important, because this is the $(n+1)^{th}$ sum output bit of the overall hybrid adder.

The time when the sum output of the most critical sum pin of combined adder will be ready is: T =Max (T2$_s$, T3$_s$, T4$_s$, T4$_c$). The value T represents the time when the most critical output signal of the SOP block will be ready if the bit-width of SubAdder$_3$ is $\lceil(n$-w_1-$w_2)/2\rceil$ bits and the bit-width of SubAdder$_4$ is $\lfloor(n$-w_1-$w_2)/2\rfloor$.

Now, we perform a similar analysis for several other configurations (bit-widths). Out of all the analyzed configurations, we select the configuration which has the lowest value of T. If the width of the SubAdder$_4$ in the configuration $c_1$is less than 4 bits, then we conclude that further exploration should be performed in the direction of reduced width of SubAdder$_3$. On the other hand, if the width of the SubAdder$_4$ is greater than 4 bits, then we choose the two configurations (c_1+1) and (c_1-1). After computing the T values for both of these configurations, if T(c_1+1) $>$ T(c_1-1), then further exploration is performed only in the direction of reduced bit-width of SubAdder$_3$. Otherwise, further exploration is performed only in the direction of increased bit-width of the SubAdder$_3$ block. If further exploration is performed in the direction of reduced bit-width of SubAdder$_3$, then the width of SubAdder$_3$ ranges from 0 bits to c_1 bits. On the other hand, if further exploration is performed in the direction of increased width of the SubAdder$_3$, then the width of SubAdder$_3$ ranges from c_1 bits to (n-w_1) bits. Once the direction of further exploration (reduced or increased bit-width for SubAdder$_3$) is decided, we perform a bit-width exploration in a fashion which is similar to a binary search algorithm. In the rest of the section, we explain the situation where further exploration happens in the direction of increased bit-width of the SubAdder$_3$ block. The reverse situation is analogous, hence it is not separately explained.

Now, to perform further exploration of other configurations (assuming that further exploration is performed in the direction of increased bit-width of the SubAdder$_3$), we select the mid-point between c_1 and (n-w_1-w_2) as the next bit-width of the SubAdder$_3$. Let us call that the configuration c_2. After analyzing the delay through that configuration, if we find out that $T(c_2) < T(c_1)$, then

- We mark c_2 as our best configuration so far.
- For the exploration of next configuration, we choose the two configurations (c_2+1) and (c_2-1) and continue the process explained earlier.

On the other hand, if we find that $T(c_2) > T(c_1)$, then

- Since c_1 is our best configuration so far, we discard c_2.
- For the exploration of the next configuration, we select the mid-point between c_2 and c_1.
- We do not need to explore any configuration of the SubAdder$_3$ having $w_2 \geq c_2$ bits.

We repeat this process until the algorithm converges to one configuration. In this way, we determine the bit-widths of SubAdder$_3$ and SubAdder$_4$ of our adder block.

Name of the SOP Block	Widths of the Input Signals of SOP Block	Name of the Hybrid Adder	Widths of Two Inputs of Adder
Mult-1	41 , 36	Adder-75	75
Mult-2	21 , 16	Adder-35	35
Mult-3	27 , 27	Adder-52	52
Mac-1	32, 32, 32	Adder-63	63
Mac-2	22, 29, 32	Adder-50	50
Mac-3	16, 16, 16	Adder-31	31
Sop-1	24, 25, 13, 18	Adder-48	48
Sop-2	36, 31, 42, 27	Adder-68	68
Sop-3	32, 26, 16, 16	Adder-57	57
Sqr-1	25	Adder-47	47
Sqr-2	32	Adder-61	61
Sqr-3	46	Adder-89	89

TABLE I

CHARACTERISTICS OF DIFFERENT ADDER BLOCKS (INSIDE SOP)

IV. EXPERIMENTAL RESULTS

To collect different data-points regarding the quality of results of our proposed four-stage hybrid final adder in a Sum-of-Product block, we used the following variations:

- Multiple hybrid adders (in different SOP blocks) with different expressions and bit-widths (as listed in Table I).
- A commercial technology library (L$_1$) for a $0.13\mu m$ technology and another one (L$_2$) for a $0.09\mu m$ technology.
- Following two types of input arrival time constraints.
  - Type-A constraint: different input bits arrive at different times. We believe that this category represents the actual timing situations in most of the SOP blocks in real-life designs. If we denote $Arr(a_i)$ as the arrival time of the i^{th} bit of n-bit wide input signal a and if k is a constant and δ is the delay of the fastest 2-input AND-gate in the technology library, the following are some examples of the Type-A timing constraints. Similar expressions for arrival times would apply to all the bits of all other input signals of the SOP block.

			Topology (Bit-Width) of four sub-adders in the Hybrid Adder				Worst-case Delay (ps)		
Design Name	Technology Library	Timing Constraint	SubAdder$_4$ (Brent-Kung)	SubAdder$_3$ (Brent-Kung)	SubAdder$_2$ (Kogge-Stone)	SubAdder$_1$ (Ripple)	Commercial Tool	Our Approach	% Improved
Adder-75	L$_1$	Type-A1	15 bits	10 bits	32 bits	18 bits	1073	898	16.29%
Adder-35	L$_1$	Type-A1	9 bits	6 bits	16 bits	4 bits	891	772	13.37%
Adder-52	L$_1$	Type-A1	8 bits	5 bits	32 bits	7 bits	957	836	12.64%
Adder-63	L$_1$	Type-A1	11 bits	7 bits	32 bits	13 bits	994	837	15.73%
Adder-50	L$_1$	Type-A1	7 bits	5 bits	16 bits	3 bits	938	807	13.96%
Adder-31	L$_1$	Type-A1	8 bits	4 bits	32 bits	6 bits	745	664	10.85%
Adder-48	L$_1$	Type-A1	7 bits	4 bits	32 bits	5 bits	925	813	12.18%
Adder-68	L$_1$	Type-A1	16 bits	12 bits	32 bits	8 bits	1026	862	15.98%
Adder-57	L$_1$	Type-A1	10 bits	6 bits	32 bits	9 bits	981	841	14.26%
Adder-47	L$_1$	Type-A1	8 bits	5 bits	16 bits	4 bits	906	798	11.97%
Adder-61	L$_1$	Type-A1	11 bits	7 bits	32 bits	11 bits	1014	861	15.08%
Adder-89	L$_1$	Type-A1	8 bits	8 bits	64 bits	9 bits	1237	1019	17.62%
Adder-75	L$_2$	Type-B	18 bits	13 bits	32 bits	12 bits	681	573	15.83%
Adder-35	L$_2$	Type-B	10 bits	3 bits	16 bits	6 bits	564	487	13.61%
Adder-52	L$_2$	Type-B	9 bits	6 bits	32 bits	3 bits	609	519	14.78%
Adder-63	L$_2$	Type-B	17 bits	13 bits	32 bits	11 bits	626	530	15.35%
Adder-50	L$_2$	Type-B	7 bits	4 bits	16 bits	4 bits	609	522	14.26%
Adder-31	L$_2$	Type-B	9 bits	4 bits	32 bits	5 bits	483	432	10.41%
Adder-48	L$_2$	Type-B	7 bits	3 bits	32 bits	6 bits	612	528	13.76%
Adder-68	L$_2$	Type-B	19 bits	8 bits	32 bits	9 bits	686	576	16.05%
Adder-57	L$_2$	Type-B	13 bits	7 bits	32 bits	5 bits	651	556	14.64%
Adder-47	L$_2$	Type-B	7 bits	5 bits	16 bits	5 bits	540	463	14.26%
Adder-61	L$_2$	Type-B	12 bits	8 bits	32 bits	9 bits	593	507	14.53%
Adder-89	L$_2$	Type-B	13 bits	7 bits	64 bits	5 bits	807	671	16.79%
Average									14.31%

TABLE II

TOPOLOGY AND DELAY COMPARISON OF ADDER BLOCKS (INSIDE SOP) GENERATED BY OUR 4-STAGE APPROACH

1) $Arr(a_i) = i * k * \delta; \qquad 0 \leq i < n$
2) $Arr(a_i) = i^2 k\delta; \qquad 0 \leq i < n$
3) $Arr(a_i) = 0; \qquad 0 \leq i < \lceil n/2 \rceil$
   $Arr(a_i) = k\delta; \qquad \lceil n/2 \rceil \leq i < n$
4) $Arr(a_i) = 0; \qquad 0 \leq i < \lceil n/4 \rceil$
   $Arr(a_i) = k\delta; \qquad \lceil n/4 \rceil \leq i < \lceil n/2 \rceil$
   $Arr(a_i) = 2k\delta; \qquad \lceil n/2 \rceil \leq i < \lceil 3n/4 \rceil$
   $Arr(a_i) = 3k\delta; \qquad \lceil 3n/4 \rceil \leq i < n$

– Type-B constraint: all bits of all the input signals arrive at the same time. This category represents the timing situations if the SOP block is placed immediately after a register-bank, or if the primary inputs of the design are fed to the SOP block. This Type-B constraint can be represented as:
  $Arr(a_i) = ... = Arr(d_i) = k; \quad 0 \leq i < n$

We compared our approach against a commercially available best-in-class datapath synthesis tool. The synthesis tool generates arithmetic-optimized architectures for the arithmetic blocks and then it performs technology-independent optimizations, constant propagation, technology mapping, timing-driven optimization, area-driven optimization, incremental optimization etc. Due to the licensing agreements, we are unable to mention the name of the commercial tool we used.

In the Table-II, we report 24 sets of data-points about the worst-case delay results obtained for the hybrid adder blocks (present in Sum-of-Product) generated by the commercial synthesis tool and from our approach. We also report the bit-widths of the four sub-adders in the hybrid adder. Average of all 24 data-points shows that our approach results in about 14.31% faster implementation of the adder in the SOP block, with 6.62% area penalty. Since modern designs have very strict timing goals, most designers would be willing to accept delay improvement at the expense of area.

Since the SOP is a highly compute intensive operation and the hybrid adder architecture plays a significant role in determining the overall performance of the design, many real-life designs can significantly benefit from our algorithm.

V. CONCLUSION

In this paper, we have presented a new approach to implement a faster hybrid adder in a sum-of-products (SOP) block, which would be very useful when the critical path of the design goes through the SOP block. This technique synthesizes the hybrid adder by dividing it into four different sub-adders, and implementing these sub-adders by using different architectures. The experimental results indicate that our implementation of the hybrid adder is significantly faster (on an average, 14.31%) than the hybrid adder generated by a commercially available best-in-class 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*, pp. 974–984, 1998.

[2] P.M. Kogge, H.S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," in *IEEE Transactions on Computers*, C-22(8):783-91, 1973.

[3] R.P. Brent, H.T. Kung, "A regular layout for parallel adders," in *IEEE Transactions on Computers*, C-31(3):260-64, 1982.

[4] C. K. Cheng, H. Zhu, R. Graham. "On the construction of zero-deficiency parallel prefix circuits with minimum depth," in *ACM TODAES*, 11(2):387-409, 2006.

[5] Y. C. Lin, C. Y. Su. "Faster optimal parallel prefix circuits: new algorithmic construction," in *Journal of Parallel Computing*, 65(12):1585-1595, 2005.

[6] G. Dimitrakopoulos, D. Nikolos. "High-speed parallel-prefix vlsi ling adders," in *IEEE Transactions on Computers*, 54(2):225–231, 2005.

[7] E. Oruklu, V. Dave, J. Saniie. "Performance evaluation of flagged prefix adders for constant addition," in *Electro/information Technology Conf*, 415-420, 2006.

[8] H. Zhu J. Liu, S. Zhou and C. K. Cheng. "An algorithmic approach for generic parallel adders," in *IEEE/ACM ICCAD*, 734, 2003.

[9] S. Olariu R. Lin, K. Nakano and A. Y. Zomaya. "An efficient parallel prefix sums architecture with domino logic," in *IEEE Transactions on Parallel and Distributed Systems*, 14(9):922-931, 2003.

[10] P.F.Stelling, V.G.Oklobdzija, "Design strategies for optimal hybrid final adders in a parallel multiplier," in *Journal of VLSI Signal Processing*, 14(3):321-31, 1996.

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