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% 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.

II. PRELIMINARIES

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

\[ z = \sum_{i=1}^{n} (a_i \cdot b_i) + \sum_{j=1}^{m} c_j, \]

where \( a_i, b_i, c_j \) are arithmetic operands.

In this SOP block, there are \( n \) product terms \( (a_i \cdot b_i) \) 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).

\[ 978-1-4244-2167-1/08/$25.00 ©2008 IEEE \]

Authorized licensed use limited to: Texas A M University. Downloaded on May 20, 2009 at 03:47 from IEEE Xplore. Restrictions apply.
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):

- SubAdder₁: A slow ripple adder for \( w₁ \) bits near the LSB.
- SubAdder₂: A fast Kogge-Stone adder for next \( w₂ \) bits.
- SubAdder₃: A carry-select adder for next set of \( w₃ \) bits.
- SubAdder₄: Another carry-select adder for the remaining \( w₄ \) bits near the MSB. In our implementation for SubAdder₃ and SubAdder₄, we use 2 copies of the Brent-Kung architecture (one with \( carry_{y_{in}}=1'b0 \) and the other one with \( carry_{y_{in}}=1'b1 \)).

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 SubAdder₁

In our proposed hybrid adder, the sub-adder for the least significant \( w₁ \) 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 (SubAdder₂) 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 \( \delta \)) is used throughout the algorithm. We start the timing analysis from the LSB (0ᵗʰ bit) and test if the Full-Adder cell would produce the carry-out (which becomes carryₓₐₐ to the 1ˢᵗ bit) before the latest arriving signals among \( x₁ \) and \( y₁ \) arrive. If the result of the test is true, then we include the 0ᵗʰ bit in the ripple carry adder. This test is carried out for the 0ᵗʰ, 1ˢᵗ, 2ⁿᵈ, 3ʳᵈ bits and so on. In addition, if for some bit (let’s say, the \( iᵗʰ \) bit), the timing analysis shows that the carry-out (or the carryᵢₐₐ 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 \( iᵗʰ \) bit in the ripple carry adder. This is because the increase in arrival time for the \( xᵢ \) and \( yᵢ \) 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 up to 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 SubAdder₂

After determining that the ripple adder (near the LSB) should be \( w₁ \) bit wide, the next set of \( w₂ \) 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 SubAdder₂ and the subsequent carry-select adders (SubAdder₃ and SubAdder₄) is very useful in reducing the overall delay of the adder. However, if the widths of the SubAdder₃ and SubAdder₄ are small, then it is better to use a single \( (n-w₁) \) bit adder for SubAdder₂. Hence if \( (n-w₁) \) is less than 8 bits, we use a \( (n-w₁) \) bits wide SubAdder₂ implemented using Kogge-Stone architecture.

On the other hand, when \( (n-w₁) \) is greater than 8, then we want to use \( w₂ \)-bit wide Kogge-Stone adder for SubAdder₂ (in addition to the carry-select adders in subsequent addition subblocks). To maximize the timing improvement from the SubAdder₂, 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 \( (2ᵏ+1) \) and \( 2ᵏ+1 \).

We compute \( w₂ \) based on the following equation:

\[ w₂ = n-w₁ \quad \text{for } (n-w₁) \leq 8 \quad (1) \]
\[ w₂ = 2^p \quad \text{where } p = \lfloor \log₂(n-w₁) \rfloor \quad \text{for } (n-w₁) > 8 \quad (2) \]

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 SubAdder₂ reduces the overall delay of the hybrid adder block.

C. Determination of the widths of SubAdder₃ and SubAdder₄

After determining that the ripple carry adder (near the LSB) is \( w₁ \) bits wide and the Kogge-Stone adder is \( w₂ \) bits wide, the remaining \( (n-w₁-w₂) \) bits need to be split between the remaining two Brent-Kung (BK) carry-select adders (SubAdder₃ and SubAdder₄) near the most significant bit (MSB).

We need to determine the arrival-time of the \( carry_{in} \) signal of SubAdder₃. 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 SubAdder₂. We refer to this as \( T_{2s} \).
2) The time when the \( carry_{out} \) of SubAdder₂ will be ready to be fed to SubAdder₃. We refer to this as \( T_{2c} \).
In our algorithm to identify the widths of SubAdder3 and SubAdder4, we analyze several configurations of these two adder blocks and then select the best combination of these widths, such that the most critical sum outputs of SubAdder3 and SubAdder4 become available at reasonably close time. This time should be close to the time T2s when the most critical sum output of the SubAdder2 module becomes available.

Let us assume that a configuration q indicates that bit-width of SubAdder3 is q bits and the bit-width of the SubAdder4 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 SubAdder3 is \(\lceil (n-w_1-w_2)/2 \rceil\) bits and the width of the SubAdder4 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 SubAdder3. We refer to this as \(T_3\).
2. The time when \(carry_{out}\) of SubAdder block will be ready to be fed to SubAdder4. We refer to this as \(T_3\).
3. The time when sum output will be ready at the most critical sum pin of SubAdder4. We refer to this as \(T_4\).
4. The time when the \(carry_{out}\) of SubAdder4 block will be ready. We refer to this as \(T_4\). 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: \(T = \text{Max} (T_2, T_3, T_4, T_s)\). The value \(T\) represents the time when the most critical output signal of the SOP block will be ready if the bit-width of SubAdder3 is \(\lceil (n-w_1-w_2)/2 \rceil\) bits and the width of SubAdder4 is \(\lfloor (n-w_1-w_2)/2 \rfloor\) bits.

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 SubAdder4 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 SubAdder3. On the other hand, if the width of the SubAdder4 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 SubAdder3. Otherwise, further exploration is performed only in the direction of increased bit-width of the SubAdder3 block. If further exploration is performed in the direction of reduced bit-width of SubAdder3, then the width of SubAdder3 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 SubAdder3, then the width of SubAdder3 ranges from \(c_1\) bits to \((n-w_1)\) bits. Once the direction of further exploration (reduced or increased bit-width for SubAdder3) 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 SubAdder3 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 SubAdder3), we select the mid-point between \(c_1\) and \((n-w_1-w_2)\) as the next bit-width of the SubAdder3. 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 SubAdder3 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 SubAdder3 and SubAdder4 of our adder block.

<table>
<thead>
<tr>
<th>Name of the SOP Block</th>
<th>Name of the Hybrid Adder</th>
<th>Widths of Two Inputs of Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-1</td>
<td>Adder-25</td>
<td>35</td>
</tr>
<tr>
<td>Multi-2</td>
<td>Adder-30</td>
<td>35</td>
</tr>
<tr>
<td>Multi-3</td>
<td>Adder-35</td>
<td>35</td>
</tr>
<tr>
<td>Mac-1</td>
<td>Adder-40</td>
<td>50</td>
</tr>
<tr>
<td>Mac-2</td>
<td>Adder-45</td>
<td>50</td>
</tr>
<tr>
<td>Mac-3</td>
<td>Adder-50</td>
<td>50</td>
</tr>
<tr>
<td>Sop-1</td>
<td>Adder-55</td>
<td>50</td>
</tr>
<tr>
<td>Sop-2</td>
<td>Adder-65</td>
<td>60</td>
</tr>
<tr>
<td>Sop-3</td>
<td>Adder-70</td>
<td>60</td>
</tr>
<tr>
<td>Sqr-1</td>
<td>Adder-75</td>
<td>60</td>
</tr>
<tr>
<td>Sqr-2</td>
<td>Adder-80</td>
<td>60</td>
</tr>
<tr>
<td>Sqr-3</td>
<td>Adder-85</td>
<td>60</td>
</tr>
</tbody>
</table>

**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 (L1) for a 0.13\(\mu m\) technology and another one (L2) 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 \(\delta\) 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.

509
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


