

Optimized Synthesis of Sum-of-Products

Reto Zimmermann and David Q. Tran
DesignWare, Solutions Group, Synopsys, Inc.
2025 NW Cornelius Pass Rd., Hillsboro, OR 97124

Abstract **– In our latest approach to datapath synthesis from RTL, datapaths are extracted into largest possible sum-of-product (SOP) blocks, thus making extensive use of carry-save intermediate results and reducing the number of expensive carry-propagations to a minimum. The sum-of-product blocks are then implemented by constraint- and technology-driven generation of partial products, carry-save adder tree and carry-propagate adder. A smart generation feature selects the best among alternative implementation variants. Special datapath library cells are used where available and beneficial. All these measures translate into better performing circuits for simple and complex datapaths in cell-based design.**

I. Introduction

While current datapath synthesis tools for cell-based design (i.e., automated netlist synthesis for standard-cell or gate-array technologies) deliver satisfying performance for simple arithmetic operations – like adders and multipliers – there is still potential for improvement in the synthesis of more complex datapaths. In order to achieve that goal, two levels have to be addressed. First, more sophisticated partitioning of complex datapaths and more arithmetic optimizations have to be applied. Second, datapath generators have to become more advanced and flexible in order to efficiently implement the more complex datapath partitionings in a context-driven way. Context-driven means that on one hand the timing-context – i.e., input arrival and output required times – of a datapath block as part of a bigger design is taken into account, and that on the other hand the characteristics of the technology library – i.e., the performance of individual library cells and the availability of special cells – is considered. This paper describes our latest approach on tackling the second level, the optimized synthesis of complex datapaths, in particular of sum-of-products.

Section II introduces and justifies the approach of addressing datapath synthesis at the sum-of-products level. Section III describes the possibilities and limitations for the implementation of product-of-sums. Sections IV-VI describe in detail the algorithms and techniques that are used for context-driven synthesis of the sum-of-product building blocks: partial-product generation, carry-save addition, and carry-propagate addition. Quantitative measures for the observed performance gains are given based on experiments with a large number of modern cell libraries for standard-cell (down to 0.13µm feature sizes) and gate-array technologies. Section VII finally gives a quick overview of the smart generation feature that is used to explore different implementation alternatives and to choose the best one.

II. Sum-of-Product Synthesis

The elementary and most important arithmetic operations in datapaths are multioperand addition and multiplication, while other operations like two-operand addition, subtraction, increment, comparison or squaring are just special cases thereof [1]-[3]. Hardware implementations of multiplication – the most complex of above operations – include three parts: *partial-product generation (PPG)*, *carry-save addition (CSA)*, and *carry-propagate addition (CPA)*. All other operations can be implemented by the same, possibly simplified, hardware. While the output of the carry-propagate adder (also referred to as *final adder* in multipliers) is in irredundant *binary representation*, the outputs of the partial-product generation and carry-save adder can be regarded as intermediate results in redundant *partial-product* (= arbitrary number of bits per bit position) or *carry-save representation* (= two bits per bit position).

It is a common technique to speed up datapaths by keeping intermediate results in a redundant representation and by performing a time-consuming carry-propagation only at the end and where necessary. This is possible because the addition operation is associative and can accept operands in redundant representation, which allows to implement a sum of multiple products and addends – or a *sum-of-products (SOP)* – by one big carry-save adder followed by one single final carry-propagate adder. Therefore, largest possible sum-of-products – and simple arithmetic operations are just special cases thereof – are extracted from RTL and implemented as one single datapath block containing multiple parallel partial-product generators, one carry-save adder and one carry-propagate adder. The same can be done for magnitude and equality comparison of sum-of-products with the only difference that the final adder provides the comparison flags instead of the sum.

For datapaths that allow sharing of resources and common subexpressions, hardware can be shared among multiple SOPs through flexible datapath partitioning (i.e., the appropriate choice of representations for shared results and the use and arrangement of CSA and CPA blocks) in order to trade hardware sharing versus duplication, or circuit area versus speed. As an example, consider the following datapath:

$$X = A \times B + C,$$
$$Y = A \times B + D.$$

Hardware for the multiplication $A \times B$ can be shared in different ways, resulting in the datapath partitionings depicted in Fig. 1. It can be easily seen that the different partitionings result in different circuit performance: (a) implements the slowest datapath with two carry-propagations in series and low area requirements, (c) and (d) implement the fastest datapaths with big area requirements due to the duplicated carry-save adder, while (b) represents a good trade-off between area and speed.

III. Product-of-Sum Synthesis

As opposed to addition, multiplication has more restrictions on the representation of its input operands. While it is theoretically possible to allow redundant representations with arbitrary

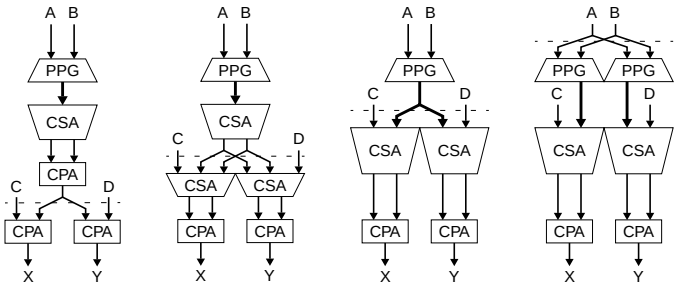

Fig. 1. Datapath partitioning with multiplier hardware being shared up to the (a) binary, (b) carry-save, (c) partial-product, and (d) input operand level.

numbers of bits per bit position on operands, such multipliers become slow and prohibitively large because of the large number of partial-product bits that have to be generated and summed up. However, there exist techniques that result in performance benefits for multiplication with one operand in carry-save and one operand in binary representation [4], [5] (see *carry-save multipliers* in Section IV). This allows for some limited implementation of *product-of-sums (POS)* – e.g., an addition or multiplication followed by a multiplication or, more general, several SOPs in series – within one datapath block with one final carry-propagate adder but no intermediate carry-propagations, thus resulting in shorter delays.

IV. Partial-Product Generation

In the first step of sum-of-product synthesis, all partial-products are generated in parallel. While the partial-product generation for addends is trivial (i.e., simply add the addend to the set of partial-products), this section elaborates on alternatives of partial-product generation for multiplication.

Note that since all partial-product bits are generated independently in parallel and in constant time, there are no optimizations necessary or possible for nonuniform input arrival times. Therefore, the partial-product generators do not work in a constraint-driven way (i.e., input arrival and output required times do not influence the circuit structure).

A. Constant Multiplication

In constant multiplication, a common technique to reduce the number of partial-products – and thus to reduce circuit area and delay for adding them up – is to reduce the number of nonzero digits in the constant operand representation by using a redundant *signed-digit (SD)* representation that is based on the digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ represents the value -1 and results in a negative partial-product. The widely used *canonic signed-digit (CSD)* representation is a minimal SD representation (i.e., minimal number of nonzero digits) with no consecutive nonzero digits [2]. It is obtained by applying the following replacement pattern repeatedly from right to left to the binary representation of the constant operand:

$$01\{1\}1 \rightarrow 10\{0\}\bar{1}.$$

Because negative digits $\bar{1}$ require the multiplicand to be complemented (i.e., invert all bits and add a '1'), some hardware can be saved by minimizing the number of negative digits as well. This is achieved in the *modified canonic signed-digit (MCSD)* representation by applying the replacement pattern above only if the number of nonzero digits is actually reduced (note that this is not the case in $011 \rightarrow 10\bar{1}$). The following replacement pattern is used instead:

$$011\{1\}\{01\}1 \rightarrow 100\{0\}\{\bar{1}0\}\bar{1}.$$

The benefits of a CSD over a binary representation heavily depend on the value of the constant and can range from 0% to –75% delay reduction and 0% to –90% area reduction for a constant multiplier. Area reduction of MCSD compared to CSD is only small on average but can go up to –20% for certain constants, while delays do not change significantly.

B. Binary Non-Booth Multiplication

For unsigned multiplication of two binary operands (no Booth recoding), the partial-products are generated in a straightforward way using AND gates. For 2's complement multiplication, the *modified Baugh-Wooley* scheme [2] is used, which reduces the extra correction bits to constants in noncritical columns in a very effective way.

C. Carry-Save Non-Booth Multiplication

For multiplication of one carry-save with one binary operand (no Booth recoding), the carry-save operand (*AC*, *AS*) is first converted to *delayed-carry* representation (*C*, *S*) by feeding it to a row of half-adders:

$$(c_{i+1}, s_i) = ac_i + as_i.$$

Due to the property of the delayed-carry representation that not both carry and sum outputs from a half-adder can be 1 at the same time (i.e., $c_{i+1}s_i = 0$), the two partial-product bits $c_{i+1}b_{k-1}$ and s_ib_k cannot be 1 at the same time either and therefore can be added together by a simple OR gate and thus be reduced to a single partial-product bit:

$$pp_{i,k} = c_{i+1}b_{k-1} + s_ib_k.$$

The number of partial-product bits for a $m \times n$ carry-save multiply is thereby reduced from $2mn$ to $(m+1)n$ at the expense of more complex generation logic. An equivalent scheme was presented in [5] together with a scheme for both operands in carry-save representation, which however seems not to translate into any significant benefits.

Compared to the alternative implementation of a carry-propagate adder (to reduce the carry-save operand to binary) followed by a binary multiplier, the described carry-save multiplier trades the slow but rather small carry-propagate adder (with width-dependent delay and area-per-bit) for a faster but larger partial-product generation (with constant delay and area-per-bit), resulting in an overall delay improvement (bigger with increasing width) but a considerable area penalty (smaller with increasing width). For the simple product-of-sum $(A+B)\times C$ and operand widths between 8 and 64 bits, a delay reduction of about –5% to –15% at area increases of +40% down to +20% are observed.

D. Binary Booth Multiplication

Booth recoding is widely used to reduce the number of partial products in multipliers [1]. The benefit is mainly an area reduction in multipliers with medium to large operand widths (8 or 16 bits and higher) due to the massively smaller adder tree, while delays remain roughly in the same range. Different

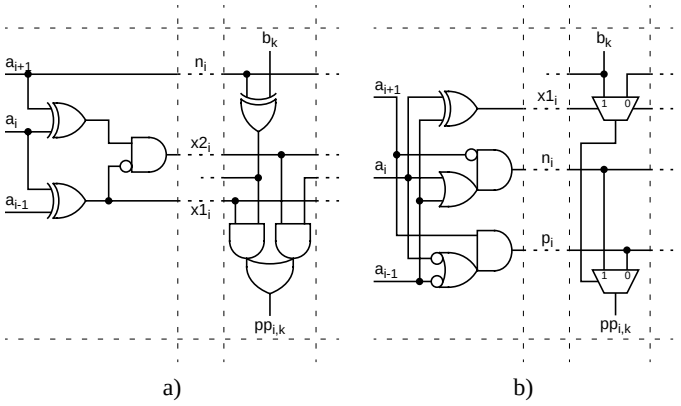

Fig. 2. Binary Booth recoder (left) and selector (right) for (a) XOR-based and (b) mux-based implementation.

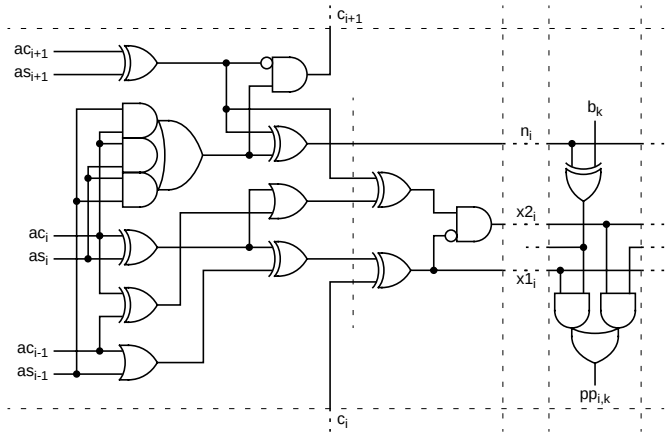

Fig. 3. Carry-save Booth recoder for XOR-based implementation.

recodings exist resulting in different gate-level implementations and performance. In this work, two alternatives are implemented: the XOR-based implementation (Fig. 2a) gives lowest area and delay numbers in most technologies due to the small selector size and the well-balanced signal paths, while the mux-based implementation (Fig. 2b) can give best results in some multiplexer-based technologies (e.g. gate-arrays).

Compared to nonrecoded multipliers, a Booth-recoded multiplier saves between −7% and −30% area for operand widths of 8 through 64 bits while delays remain in the range of ±3%.

E. Carry-Save Booth Multiplication

Booth recoding can be extended for a redundant multiplier operand [4]. While [4] starts with a signed-bit Booth recoder and adds a special preprocessing step in order to deal with a carry-save operand, the redundant Booth recoder in this work has been optimized for the carry-save representation and reduces the critical path by one XOR gate. As depicted in Fig. 3, the right part of the carry-save Booth recoder as well as the selector are equivalent to the recoder and selector of the XOR-based binary Booth implementation (Fig. 2a). The left part of the recoder passes a carry to the next higher recoder slice.

Compared to the alternative implementation of a carry-propagate adder followed by a binary Booth multiplier, the described carry-save Booth multiplier trades the slow carry-propagate adder (with width-dependent delay and area-per-bit) for a faster, more complex Booth recoding logic (with constant delay and area-per-bit), resulting in an overall delay improvement (bigger with increasing width) but a potential area penalty (smaller with increasing width). For the simple product-of-sum $(A+B)\times C$ and operand widths between 8 and 64 bits, a delay reduction of up to −16% at area increases of +40% down to +5% are observed.

As in binary multiplication, Booth recoding proves beneficial over nonrecoding in carry-save multiplication as well, reducing circuit area for medium to large operand widths by up to −30% at comparable delays.

V. Carry-Save Addition

Carry-save addition reduces all partial-products down to a carry-save number by summing them up in an adder tree [3]. The adder tree is constructed column-based by compressing all bits in one column (= bit position) down to two bits at a time in a constraint- and technology-driven way that minimizes delay:

1) Column clean-up: Columns are cleaned up by performing some optimizations and simplifications, e.g., replacing duplicate bits in one column by a single bit in the next higher column ($a + a = 2a$) and by replacing complementing bits by a constant 1 ($a + \overline{a} = 1$).

2) Column compression: All columns are compressed from the LSB (least-significant bit) column to the MSB (most-significant bit) column. Intermediate carries are forwarded to the next higher column.

A column is compressed by arranging compressor cells – half-adders (2:2 compressor), full-adders (3:2 compressor), and 4:2 compressors, if available – in a tree structure. A delay-optimized tree structure, which takes into account input arrival times (constraint-driven) and individual cell pin-to-pin delays (technology-driven), is constructed by arranging all bits of a column (inputs as well as intermediate sum and carry bits) in a set of bits sorted by their arrival time, from which bits are consumed by compressor inputs and to which new bits are added by compressor outputs, as previously described in [6]:

1) Add all input bits of a column to its set of bits.

2) Sort the set of bits according to their arrival times.

3) Instantiate a compressor cell and connect the earliest bit from the set to the slowest cell input pin. For the remaining input pins, connect the latest bits from the set that add no or only minimal extra delay to the cell outputs. Remove the connected bits from the set. Calculate the arrival times of the bits connected to the cell's output pins and add the sum bit to the set of the current column and the carry bit(s) to the set of the next higher column.

4) Repeat steps 2) and 3) until there are only two bits left in the set of the current column. These two bits form the carry-save output of the carry-save adder for this column.

In addition, the type of compressor cell to be used is determined by the following rules:

1) If the number of bits in the set is odd and ≥3, instantiate one single 2:2 compressor (half-adder). This reduces the number of bits in the set to an even number (multiple of 2).

2) If the number of bits in the set is a multiple of 4 and ≥4, instantiate one single 3:2 compressor (full-adder). This reduces the number of bits in the set to a multiple of 4 plus 2.

3) If 4:2 compressors are available/desired, instantiate 4:2 compressors until only 2 bits are left in the set. Each 4:2 compressor reduces the number of bits in the current column by 4 (i.e., 5 inputs to 1 output).

4) If no 4:2 compressors are available/desired, instantiate 3:2 compressors (full-adder) until only 2 bits are left in the set. Each 3:2 compressor reduces the number of bits in the current column by 2.

As elaborated in [6], the resulting reduction tree has minimal delay and a minimal number of compressor cells (i.e., minimal area). Since the minimal number of compressor cells is only determined by the number of bits in a column [3], no area can be saved by other, possibly slower tree arrangements. For that reason always the fastest reduction tree is constructed and therefore output required times are not considered.

The usage of 4:2 compressor cells does not inherently result in faster or smaller adder trees. The reason is that the number of XOR levels, which dominates the overall delay, in a delay-optimized full-adder tree can be as low as in a tree of 4:2 compressors [6]. 4:2 compressors can give somewhat better performance if implemented efficiently. However, experimental results have not shown any significant benefits in cell-based design. The decision whether to use 4:2 compressor cells is made as part of the smart generation feature (see Section VII).

Compared to a Dadda adder tree [2], which simply minimizes the number of full-adders on the critical path without taking actual pin-to-pin delays in to account, adder trees generated by the described timing-driven algorithm reduce overall delay in a multiplier by –3% to –8% for operand widths between 8 and 64 bits.

VI. Carry-Propagate Addition

Carry-propagate addition finally converts the redundant carry-save output from the carry-save adder into irredundant binary representation by performing a carry-propagation [1]. A variety of different schemes exist to speed up carry-propagation that trade off area versus speed. The relevant adder architectures and their characteristics are summarized in Table I. Two principles have to be distinguished here: the *prefix structure* employed to propagate carries from lower to upper bits and the *sum bit generation* that determines how the sum bits are calculated from the carries.

A. Prefix Structure

As widely known, carry-propagation in binary addition is a *prefix problem* [8], which can be calculated using *prefix structures*. Besides the straightforward *serial-prefix* structure (implemented by the *ripple-carry* adder) many different *parallel-prefix* structures exist, which speed up carry-propagation at the cost of increased area requirements [7]. They basically differ in terms of depth (= circuit speed), size (= circuit area) and maximum fanout, which can be *bounded* (constant) or *unbounded* (dependent on the operand width) and influences circuit speed and area in a more subtle way. The internal signals of a prefix implementation can be coded in different ways, resulting in different possible logic implementations. Most common are the use of *generate/propagate* signal pairs computed by AND-OR gates and *carry-in-0/carry-in-1* signal pairs computed by multiplexers [7]. Table I shows that all adder architectures except the carry-skip adder use some kind of serial- or parallel-prefix structure for propagating the carries.

TABLE I
Adder Architecture Characteristics

Architecture	Area	Speed	Prefix based	Sum bit generation	Maximum Fanout	Included in work
Ripple-carry	lowest	lowest	yes	–	bounded	yes
Carry-skip	low	low	no	–	bounded	no
Carry-select	medium	medium	yes	select	unbounded	yes
Carry-increment	medium	medium	yes	lookahead	unbounded	yes
Brent-Kung PP [a]	medium	medium	yes	lookahead	bounded	yes [b]
Sklansky PP [a]	high	highest	yes	lookahead	unbounded	yes [b]
Kogge-Stone PP [a]	highest	highest	yes	lookahead	bounded	yes
Conditional-sum	highest	highest	yes	select	unbounded	yes

[a] Parallel-prefix adder
[b] Indirectly through appropriate timing constraints

B. Sum bit generation

There are two different schemes to calculate the sum bits from the carries:

1) Carry-lookahead scheme: The prefix structure is used to calculate (look-ahead) the final carry for each bit position, which then computes the sum bit through an additional XOR gate. All parallel-prefix adders from Table I and the carry-increment adder use this scheme.

2) Carry-select scheme: For each bit position the two possible sum bits for a carry-in of 0 and 1 are calculated in advance and then selected through a series of multiplexers controlled by all levels of carries from the prefix structure. Compared to the carry-lookahead scheme, this scheme usually results in bigger circuit size (multiple levels of multiplexers versus one level of XOR gates) but potentially shorter delay (no final XOR gate necessary). The carry-select and conditional-sum adders from Table I use this scheme.

C. Architecture Performance Comparison

The relative performance of all these adder architectures varies greatly among different technology libraries, so that only qualitative characteristics regarding area and speed are summarized in Table I instead of quantitative comparison results. In addition, the following observations can be made:

- The *ripple-carry* adder implemented using full-adder cells is always the smallest and slowest adder.
- The *carry-skip* adder [1] massively speeds up the ripple-carry adder at a very moderate area penalty but is still slower than any other architecture. However, due to its false paths it cannot readily be used in synthesis-based design.
- The *carry-select* [1] adder is very area efficient for medium speeds if special carry-select adder cells are available in the library. Its prefix structure has the special property of allowing maximally 2 prefix nodes per bit position[7].
- The *carry-increment* adder [7], [9] is an optimization of the carry-select adder that uses the carry-lookahead scheme instead of the carry-select scheme for the same prefix structure. It has the same delay but a 30% smaller gate count.

• The *Brent-Kung parallel-prefix* adder [10] gives a good trade-off between area and speed, lying in the range of –15% to –30% area reduction at +15% to +30% delay increase as compared to the faster Sklansky parallel-prefix adder.

• The *Sklansky parallel-prefix* adder (uses the prefix structure first proposed by Sklansky for conditional-sum adders [11]) has a prefix structure of minimal depth and therefore is among the fastest adder architectures. Its unbounded-fanout property helps reduce circuit area (fewer prefix nodes) but adds some extra delay for driving the high-fanout nodes.

• The *Kogge-Stone parallel-prefix* adder [12] also has a minimal depth prefix structure. Its bounded-fanout property eliminates the need for driving high-fanout nodes, making it the fastest adder in most technologies, but comes at the cost of much bigger area (more prefix nodes) and more wiring. Compared to the Sklansky prefix adder, it shows an area increase between +23% (8 bit) and +75% (128 bit) at a fairly constant delay reduction of around –4% (all widths).

• The *conditional-sum* [11] adder has one less logic level on the critical path but higher fanouts and more cells compared to parallel-prefix adders with the same prefix structure, and its implementation bases mainly on multiplexers instead of AND-OR gates. This results in generally bigger area and longer delays for most cell libraries, but can also result in up to –20% delay reduction for multiplexer-based gate-array technologies as compared to the Kogge-Stone parallel-prefix adder. The traditional conditional-sum adder [11] uses the Sklansky prefix structure, but other prefix structures can be employed equally.

It is important to note that the high-fanout nodes in unbounded-fanout architectures do not deteriorate circuit speed as much as is often reported, if handled properly. With appropriate fanout-decoupling along the critical paths (i.e., shielding the high fanout from a critical node by an additional buffer/inverter), only one single high-fanout node is left on each signal path, which can be buffered accordingly. Furthermore, unbounded-fanout architectures have many signal paths that are noncritical, which allows to size down many noncritical gates and thus decrease loads and delays on critical paths. On the other hand, most signal paths in bounded-fanout architectures are critical and need bigger sized gates. The resulting larger input loads together with the tendentially longer wires increase average node capacitances and add some extra delay. This is why bounded-fanout architectures (e.g., Kogge-Stone) show only slightly shorter delays but excessively larger area as compared to unbounded-fanout architectures (e.g., Sklansky).

D. Parallel-Prefix Adder Synthesis

From the previous paragraphs it becomes obvious that all relevant adder architectures for the whole range of area-delay trade-offs are based on a variety of prefix structures. But instead of generating and choosing among different static prefix structures, an algorithm has been implemented that generates flexible prefix graphs[1] that are optimized for a given context [7], [13]. The algorithm repeatedly applies depth- and size-decreasing prefix transforms [14] (Fig. 4) in order to minimize overall prefix graph size for a given maximum depth:

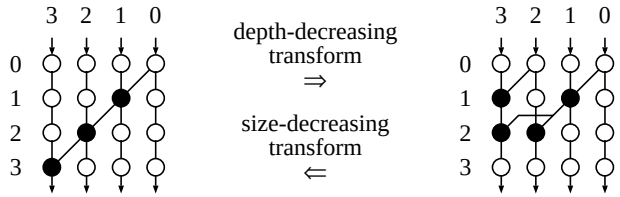

Fig. 4. Depth-decreasing and size-decreasing prefix transforms.

1) Prefix graph compression (depth minimization): Depth-decreasing transforms are applied in right-to-left bottom-up graph traversal order.

2) Depth-controlled prefix graph expansion (size minimization): Size-decreasing transforms are applied in left-to-right top-down graph traversal order, if maximum depth constraint is not violated. An additional shift-down step is introduced to decouple high-fanout nodes from the critical path.

With this prefix graph optimization algorithm, prefix adders that are area-optimized under given timing constraints can be synthesized as follows:

1) Translate timing constraints into prefix graph constraints.

2) Generate a serial-prefix graph.

3) Perform prefix graph compression.

4) Perform depth-controlled prefix graph expansion.

5) Map the prefix graph to prefix adder logic, using either the carry-lookahead or the carry-select scheme.

The algorithm can process input arrival times and output required times at the bit level and therefore can optimize adders for arbitrary nonuniform signal arrival and required profiles. In particular, it can generate

• ripple-carry adders under very loose timing constraints,

• Brent-Kung parallel-prefix adders under medium timing constraints,

• Sklansky parallel-prefix adders under tight timing constraints (see Fig. 5 with fanout-decoupling),

• carry-select and carry-increment adders by limiting the number of prefix nodes per column to 2,

• mixed ripple-carry/parallel-prefix adders under loose constraints by generating mixed serial/parallel-prefix structures (Fig. 7, full-adder cells can be used in serial-prefix part), and

• optimized multiplier final adders that take into account the typical signal arrival profiles of adder tree outputs in multipliers (Fig. 6-8), similar but more flexible than the adders in [6].

Because the prefix optimization algorithm does not take fanouts into account, no bounded-fanout parallel-prefix structures (i.e., Kogge-Stone) can be generated at this time. For these prefix structures, a simple static algorithm is used.

The presented constraint- and technology-driven adder synthesis generates area-optimized adders for arbitrary timing constraints and provides a flexible one-fits-all adder architecture. Compared to static adder architectures, it results in better circuit performance for nonuniform and relaxed timing contexts and helps to reduce synthesis runtime by eliminating the need for generating and evaluating different adder architectures in order to find the best one. Delay of multipliers can be reduced by up to –4% using such an optimized final adder, and

[1] Prefix structures can be visualized and manipulated using prefix graphs, which use an array arrangement of black nodes (prefix operation, prefix logic) and white nodes (no operation/feed-through, buffers/inverters) where columns denote bits and rows denote prefix levels.

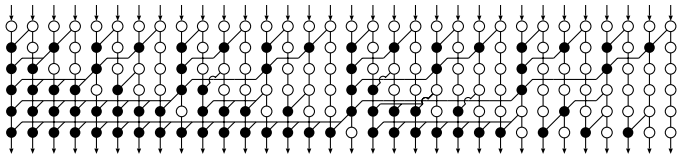

Fig. 5. Parallel-prefix structure for 32-bit adder optimized for delay (Sklansky) with fanout-decoupling on critical paths.

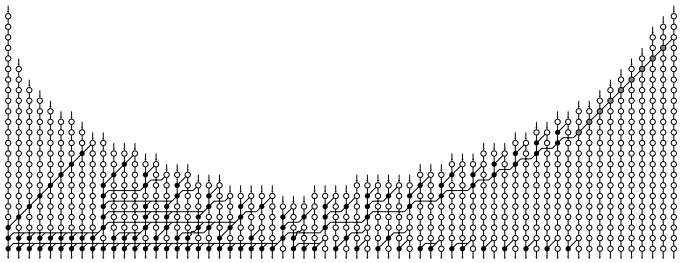

Fig. 6. Parallel-prefix structure for final adder of 32-bit multiplier optimized for tight timing constraints. Full-adder cells can be used for grey nodes

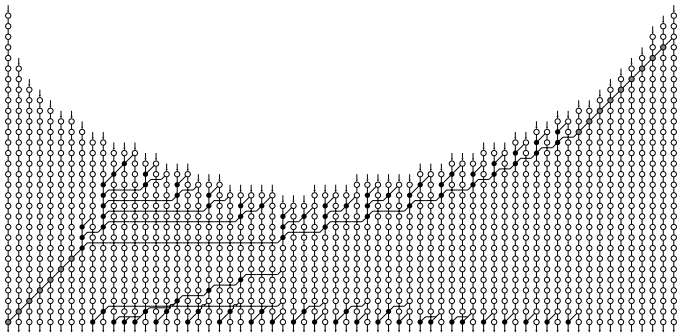

Fig. 7. Parallel-prefix structure for final adder of 32-bit multiplier optimized for relaxed timing constraints. Full-adder cells can be used for grey nodes.

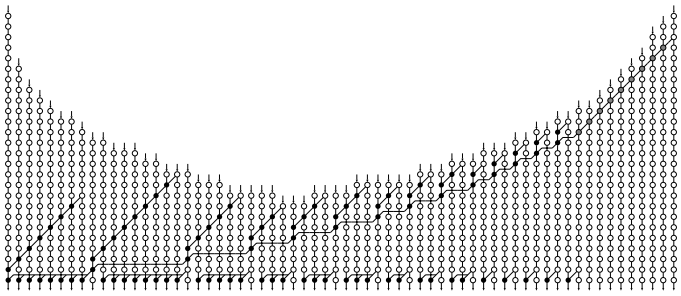

Fig. 8. Parallel-prefix structure for final adder of 32-bit multiplier optimized for carry-select/carry-increment architecture. Full-adder cells (grey nodes) and carry-select adder cells (black nodes) can be used.

much higher delay reductions can be obtained in complex datapaths that involve late arriving signals (e.g., from truncation).

VII. Smart Generation

Despite the flexibility of the presented datapath synthesis algorithms, there still exist several implementation alternatives that need to be evaluate and selected. These include

- the usage of special library cells (such as 4:2 compressor, carry-select adder, and Booth encoder cells),
- bounded or unbounded fanout in parallel-prefix adders,
- carry-lookahead or carry-select scheme in adders, and
- Booth recoding or no recoding in multipliers.

A smart generation feature has been implemented that evaluates and selects these alternatives ad hoc during the synthesis process. This also extends to some degree into higher levels where different datapath partitionings are evaluated and the best one is chosen for implementation.

VIII. Conclusions

Algorithms for the constraint- and technology-driven synthesis of sum-of-products and product-of-sums as well as the techniques that are employed to improve circuit area and speed for cell-based design have been described. While the potential for performance gains in simple arithmetic operations, like adders and multipliers, is moderate, circuit area and delay of complex datapaths – which, e.g., include common subexpressions, product-of-sums, truncations, and comparisons – can be significantly reduced through optimized partitioning and implementation. The universality of the presented datapath generators allows for more elaborate datapath partitioning and arithmetic optimizations, and their flexibility and ability to account for arbitrary delay profiles and library characteristics enables the efficient implementation of the resulting datapaths. The real performance gains heavily depend on the actual datapath, but in many cases they are well beyond the gains reported for individual operations throughout this paper. Furthermore, smart generation allows in a runtime-efficient way to explore various implementation alternatives at different levels and to select the optimal ones based on the current design context.

References

[1] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

[2] B. Parhami, *Computer Arithmetic: Algorithms and Hardware*, Oxford University Press, 2000.

[3] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann Publishers, 2004.

[4] C. N. Lyu and D. W. Matula, "Redundant Binary Booth Recoding," *Proc. 12th Symp. Computer Arithmetic*, July 1995, pp. 50–57.

[5] Y. Dumonteix and H. Mehrez, "A Family of Redundant Multipliers Dedicated to Fast Computation for Signal Processing," *Proc. IEEE Int. Symp. Circuits and Systems*, May 2000, pp. 325–328.

[6] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers*, vol. 45, no. 3, pp. 294–305, March 1996.

[7] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, Ph.D. thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.

[8] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[9] A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders," *IEEE Trans. Computers*, vol. 42, no. 10, pp. 1162–1170, Oct 1993.

[10] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Computers*, vol. 31, no. 3, pp. 260–264, Mar 1982.

[11] J. Sklansky, "Conditional Sum Addition Logic," IRE Trans. Electronic Computing, vol. EC-9, no. 6, pp. 226-231, Jun 1960.

[12] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Computers*, vol. 22, no. 8, pp. 786–793, Aug 1973.

[13] R. Zimmermann, "Non-Heuristic Optimization and Synthesis of Parallel-Prefix Adders," Proc. Int. Workshop on Logic and Architecture Synthesis, Dec 1996, pp. 123-132.

[14] J. P. Fishburn, "A Depth-Decreasing Heuristic for Combinational Logic; or How to Convert a Ripple-Carry Adder into a Carry-Lookahead Adder or Anything In-Between," Proc. 27th Design Automation Conference, 1990, pp. 361-364.