Optimisation of Mutually Exclusive Arithmetic Sum-Of-Products

T. Drane* and G. Constantinides†
*Imagination Technologies Ltd
Home Park Estate, Kings Langley
Hertfordshire, WD4 8LZ
†Department of Electrical and Electronic Engineering
Imperial College London
Exhibition Road, London, SW7 2BT

***Abstract*—Arithmetic blocks consume a major portion of chip area, delay and power. The arithmetic sum-of-product (SOP) is a widely used block. We introduce a novel binary integer linear program (BLP) based algorithm for optimising a general class of mutually exclusive SOPs. Benchmarks drawn from existing literature, standard APIs and constructed for demonstration purposes, exhibit speed improvements of up to 16% and area reduction of up to 57% in a 65nm TSMC process.**

I. Introduction

Datapath continues to consume relatively large amounts of silicon area, delay and power within Digital Signal Processing and Graphics applications. Some of the most prolific of operations can be expressed as fixed-point sum-of-products. These include adders, subtractors, multipliers, squarers, multiply-accumulators, chained additions, decrementors, incrementors, *etc*. Standards, such as graphics APIs, require integer operations of the form $\pm a \pm b$, $\pm ab$ and $\pm ab \pm c$, constructing an arithmetic logic unit to perform all of these operations would require implementing mutually exclusive SOPs. While industry standard tools perform well on each of these operations in isolation, this paper demonstrates that substantial area and delay savings can be made when these mutually exclusive operations are combined.

II. Background

SOPs can be efficiently implemented as their partial products can all be summed in parallel, as was noted in the integer part of a floating point multiply accumulator [1]. The final carry propagate adder of an SOP can be optimized to adapt to the delay profile of the array reduction, [2]. In [3], inverted partial product arrays were shown to improve quality of results. Designs implementing operations of the form $\sum k_i x_i y_i$ where k_i are constants and x_i and y_i are input operands have been considered in [4]. Here multiplication by a constant is performed by using the canonical signed digit recoding and $x_i \times y_i$ is computed in redundant carry-save form. There is a wealth of design options for SOP or POS (product-of-sum) expressions by manipulating the Booth encoded multipliers in a variety of styles [5].

978-3-9810801-7-9/DATE11/©2011 EDAA

Despite the existence of efficient implementations of SOP and POS expressions, most datapath synthesis cannot exploit these highly optimal blocks due to non SOP expressions found within the datapath. Muxing and shifting found within SOP expressions prevent full and efficient merging. In [6], data flow graphs have been locally manipulated to increase the proportion of the datapath which can be expressed as a single SOP, hence reducing delay and area. For example one of the transformations includes $(a + b + c) << d = (a << d) + (b << d) + (c << d)$, hence shifters can be moved through summations; a fact exploited fully in [7].

In terms of considering mutually exclusive SOP expressions, an example can be found in [6]: $sel?a + b : c = (sel?a : c) + (sel?b : 0)$. However such optimizations were restricted to localized regions. A fuller consideration of merging mutually exclusive operations can be found in [8]. In this instance the SOP is split into partial generation, array reduction and final carry propagate adder with muxing on inputs to each of these units.

The contribution of this paper is to algorithmically take a set of mutually exclusive SOPs and derive an equivalent single SOP with minimal control via the construction of a binary linear program (BLP). The algorithm also utilizes a novel approach to producing negative terms within an SOP. Mutually exclusive SOPs occurs naturally when multiplying sign magnitude numbers and so arise within floating point modules, *e.g.* multiply accumulate, dp2, dp3, *etc*; see [9] for an example. This contribution is orthogonal to the majority of previous research into SOP implementation and moreover can be viewed as a RTL to RTL transformation which would fit into a high level synthesis flow.

The novelty in this paper is as follows:

1 algorithmic merging of mutually exclusive SOPs into one SOP,
2 usage of a Linear Program in RTL to RTL polynomial transformations,
3 novel handling of negative terms within an SOP

III. Motivation

In order to motivate the algorithm we consider a simple mutually exclusive set of SOPs, see the pseudo code in Fig. 1. Our aim is to consider reformulating these equations in such a way that the design's hardware resource utilisation is as close to that of $ab + c$ has possible. That is, we wish to minimise the cost of any muxing and negation that is introduced. To this end we seek to reformulate the equations such that only one SOP is required. The first step in doing this is to reorder the multiplications and additions in such a way to minimize the amount of muxing between operands once merged. In this case we produce Fig. 2 by writing each SOP in the form $AB + C$ and choosing the order of A and B such that the second term in the multiplication is always b. In Fig. 3 we have merged the SOPs together by noting that if we use the standard two's complement identity $-x = \overline{x} + 1$ then $(-1)^{neg}a = a \oplus neg + neg$ where $t = a \oplus neg$ is a signed number 1 bit larger than a such that if a is n bits in length then $t[i] = a[i] \quad XOR \quad neg$ for $n > i \geq 0$ and $t[n] = neg$. Note also that s is assumed to be two bits in length and we use $s[1]$ and $s[0]$ as notation for the most and least significant bits of s respectively. We had to optionally negate the product in Fig. 3 but the product is the most delay and area expensive part of the SOP. To minimise this cost we wish to minimise the logic that provides inputs to it; hence it would be advantageous to rewrite the SOPs such that the product is always positive. To do this we will need to use another negation identity; consider replacing x with $x-1$ in the formula $-x = \overline{x}+1$, simplifying we get $-x = \overline{x-1}$. We can exploit this freedom to rewrite each SOP such that the product is always positive, see Fig. 4; inserting these back into the formula for y_1 results in Fig. 5. In this case the merging results in Fig. 6. Fig. 3 has more and larger addends than Fig. 6, moreover Fig. 6 adds little in the way of extra hardware over $ab + c$; hence this is a desirable form to aim for when implementing mutually exclusive SOPs. This reformulation is an RTL transformation, [8] cannot be used in this way as it requires access to the carry-save result from a multiplier.

IV. Problem Statement

Following the example set by the motivating sample set of SOPs we seek now to formalize this process. For ease of exposition the following simplifying assumptions will be made:

A) each SOP has the same number of terms and each term is the product of two variables,
B) operands are unsigned, non constant and of identical word length,
C) operands within each of the mutually exclusive SOPs are all distinct.

In light of these restrictions we may write our intended function as having a select signal sel which muxes between n SOPs, each with m products and where the operands are drawn from an alphabet of k elements, $\{x_1, x_2, ..., x_k\}$:

$$f_i = \sum_{r=0}^{m-1} (-1)^{s_{i,r}} \alpha_{i,r} \beta_{i,r} \quad s_{i,r} = \{0, 1\} \quad i = 0...n-1$$

$$\begin{aligned} y = &(sel == 0)? \quad f_0 : \\ &(sel == 1)? \quad f_1 : \\ &\dots \\ &(sel == n-2)? \quad f_{n-2} : f_{n-1} \end{aligned}$$

$$\begin{aligned} y_1 = &(s == 0)? \quad ab + c : \\ &(s == 1)? \quad bc - a : \\ &(s == 2)? \quad c - ab : \\ &\qquad\quad - a - bc \end{aligned}$$

Fig. 1. Sample mutually exclusive SOPs.

$$\begin{aligned} y_1 = &(s == 0)? \quad ab + c : \\ &(s == 1)? \quad cb - a : \\ &(s == 2)? - ab + c : \\ &\qquad\quad - cb - a \end{aligned}$$

Fig. 2. Reordered sample SOPs.

$$\begin{aligned} A &= s[0]?c : a \\ C &= s[0]?a : c \\ y_1 &= (-1)^{s[1]}Ab + (-1)^{s[0]}C \\ &= A(b \oplus s[1]) + As[1] + (C \oplus s[0]) + s[0] \end{aligned}$$

Fig. 3. Merged SOPs: First Attempt.

$$\begin{aligned} &ab + c \\ &cb - a = cb + \overline{a} + 1 \\ &-ab + c = -(ab + \overline{c} + 1) = \overline{ab + \overline{c}} \\ &-cb - a = -(cb + a) = \overline{cb + a - 1} \end{aligned}$$

Fig. 4. SOP Rewriting.

$$\begin{aligned} y_1 = &(s == 0)? \quad ab + c : \\ &(s == 1)? \quad cb + \overline{a} + 1 : \\ &(s == 2)? \quad \overline{ab + \overline{c}} : \overline{cb + a - 1} \end{aligned}$$

Fig. 5. Premerged SOPs: Second Attempt.

$$\begin{aligned} A &= s[0]?c : a \\ C &= (s[0]?a : c) \oplus (s[1] \oplus s[0]) \\ y_1 &= (Ab + C + s[1] \oplus s[0] - s[1]) \oplus s[1] \end{aligned}$$

Fig. 6. Optimized sample SOPs.

Where $\alpha_{i,r}$ and $\beta_{i,r}$ are drawn from the k possible inputs. For ease of notation we will use the following nomenclature: $y = \text{mux}(\{f_0, f_1, ..., f_{n-1}\}, sel)$. Based upon the structure of Fig. 6, we seek automatic transformations of y into the general form:

$$\begin{aligned}
\alpha_r &= \text{mux}((x_1, x_2, ..., x_k), g_{1,r}) \quad r \in \{0, 1, .., m-1\} \\
\beta_r &= \text{mux}((x_1, x_2, ..., x_k), g_{2,r}) \quad r \in \{0, 1, .., m-1\} \\
y &= (-1)^{g_4} \sum_{r=0}^{m-1} (-1)^{g_{3,r}} \alpha_r \beta_r \\
&= \left(\left(\sum_{r=0}^{m-1} \beta_r (\alpha_r \oplus g_{3,r}) + \beta_r g_{3,r} \right) - g_4 \right) \oplus g_4 \qquad (1)
\end{aligned}$$

Where $g_{1,r}$, $g_{2,r}$, $g_{3,r}$ and g_4 are all functions of sel. Here α_r and β_r are the result of muxing between the k operands and producing the terms that will produce the rth produce in the merged SOP. If we assume $g_4 = 0$ then we have a functional form that would produce an architecture of the form found in Fig. 3. However by using the function g_4 we can optionally negate whole SOPs and thus create architectures like those found in Fig. 6. In particular we seek transformations such that the functions $g_{1,r}$ and $g_{2,r}$ are 'minimal', a notion that will be clearly defined in Part I of the algorithm. There is clearly freedom in choosing $g_{3,r}$ and g_4, in the motivating SOPs this freedom was chosen to keep the largest product positive. Given that, for ease of exposition, it has been assumed that all inputs are of equal word length and so there is no largest product, it will be assumed that we make the first product positive. This issue will be addressed in Part IV where the assumptions stated above will be relaxed.

V. Algorithm Description

Following the structure of the optimization of the motivating SOPs our algorithm seeks to formalize the process by which Fig. 1 is transformed into Fig. 2 (Part I) and then perform the subsequent transformations (Part II).

A. Part I

The first step is to minimize the operand muxing; to facilitate this process consider the n by $2m$ matrix:

$$\mathbf{\Gamma} = \begin{pmatrix} \alpha_{0,0} & \beta_{0,0} & \cdots & \alpha_{0,m-1} & \beta_{0,m-1} \\ \alpha_{1,0} & \beta_{1,0} & \cdots & \alpha_{1,m-1} & \beta_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \alpha_{n-1,0} & \beta_{n-1,0} & \cdots & \alpha_{n-1,m-1} & \beta_{n-1,m-1} \end{pmatrix}$$

In order the use the motivating SOPs in Fig. 1, we introduce a new variable whose value is always 1. In this case $n = 4$, $m = 2$, $k = 4$ and our alphabet is the set $\{a, b, c, 1\}$, hence Γ can be constructed:

$$\begin{aligned}
y_1 = (s == 0)? \quad & a \times b + c \times 1 : \\
(s == 1)? \quad & b \times c - a \times 1 : \\
(s == 2)? \quad & c \times 1 - a \times b : \\
& -a \times 1 - b \times c
\end{aligned} \qquad \mathbf{\Gamma} = \begin{pmatrix} a & b & c & 1 \\ b & c & a & 1 \\ c & 1 & a & b \\ a & 1 & b & c \end{pmatrix}$$

Minimizing the operand muxing can then be viewed as permuting the elements in each row (while maintaining the products) in such a way as to minimize the number of distinct elements in each column. Our approach is to use binary variables and Binary Linear Programs, this is appropriate given that the BLP runtimes are acceptable for the benchmarks we consider and, we believe, the size of problems occurring in industry. To this end we decompose $\mathbf{\Gamma}$ into k binary matrices X_r corresponding to the alphabet present, such that:

$$\mathbf{\Gamma} = \sum_{r=1}^{k} x_r X_r$$

So in the case of the motivating SOPs, this decomposition becomes:

$$\begin{aligned}
\mathbf{\Gamma} = & a \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} + b \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \\
& + c \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + 1 \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad (2)
\end{aligned}$$

Given this decomposition we can view transformations of the SOPs as simply manipulating the binary matrices X_r. For X_r to still represent the original SOP, certain conditions must be satisfied. To simplify the following matrix equations we introduce some useful notation. Let $\mathbf{1}_{n_1,n_2}$ represent an n_1 by n_2 matrix entirely consisting of ones and $\mathbf{e}_i$ represent the ith standard basis vector, *i.e.* a vector with one in position i and zero elsewhere. Now our first condition is that X_r must still represent n SOPs of m products, hence:

$$\sum_{r=1}^{k} X_r = \mathbf{1}_{n,2m} \qquad (3)$$

Note that this constitutes $2mn$ constraints. Secondly, consider the product $\alpha_{i,r}\beta_{i,r}$ and assume $\alpha_{i,r} = x_p$ and $\beta_{i,r} = x_q$ we are free to position this product in $2m$ ways, *i.e.* x_p can be located in one of $2m$ places within the ith SOP, the x_q location is then fixed. This corresponds to the following restriction:

$$(X_p)_{i,j} = (X_q)_{i,j+(-1)^j} \quad \forall i, r, j \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \qquad (4)$$

Note that this constitutes a total of $2nm^2$ constraints and only holds provided that each operand is distinct within each SOP. However these constraints only check that if x_p appears in the SOP then it will be multiplied by x_q, it does not ensure that the product $x_p x_q$ exists within the SOP. Given that $\alpha_{i,r} = x_p$ and $\beta_{i,r} = x_q$, it is sufficient to check that there is a 1 within the ith row of matrix X_p:

$$\mathbf{e}_i^T X_p \mathbf{1}_{2m,1} = 1 \quad \forall i, r \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \qquad (5)$$

Note that there are nm such constraints. We are now free to choose X_r as long as the $mn(3+2m)$ constraints presented in (3), (4) and (5) hold, as the result can still be interpreted as a

valid transformation of the original set of SOPs. Referring again to the motivating SOP, we performed the following transformation between Fig. 1 and Fig. 2:

$$\Gamma = \begin{pmatrix} a & b & c & 1 \\ b & c & a & 1 \\ c & 1 & a & b \\ a & 1 & b & c \end{pmatrix} \Rightarrow \Gamma' = \begin{pmatrix} a & b & c & 1 \\ c & b & a & 1 \\ a & b & c & 1 \\ c & b & a & 1 \end{pmatrix} \quad (6)$$

We can quantify the reduction in muxing of the final SOP by this transformation. Pre-transform, operand a was involved in muxing in 2 locations, b in 4, c in 4 and '1' in 2. Post-transform a in 2, b in 1, c in 2 and '1' in 1. For a measure of the muxing cost we can sum the total number of times an operand is required within the muxing, in this case we have reduced this *total* from 12 to 6. In general the muxing *cost* can be captured by:

$$\sum_{r=1}^{k} \sum_{j=0}^{2m-1} \bigvee_{i=0}^{n-1} (X_r)_{i,j} \quad (7)$$

This is the sum of the result of 'OR'ing each column of the k matrices X_r. We can now state the optimization that will minimize the amount of final SOP muxing:

$$\begin{aligned} \min \quad & \sum_{r=1}^{k} \sum_{j=0}^{2m-1} \bigvee_{i=0}^{n-1} (X'_r)_{i,j} \\ s.t. & \sum_{r=1}^{k} X'_r = \mathbf{1}_{n,2m} \\ & (X'_p)_{i,j} = (X'_q)_{i,j+(-1)^j} \quad \forall i,r,j \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \\ & \mathbf{e}_i^T X'_p \mathbf{1}_{2m,1} = 1 \quad \forall i,r \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \end{aligned}$$

By introducing $2mk$ variables encapsulated by k vectors v_r of length $2m$, we can transform this optimization into the BLP found in Fig. 7. This program has $2m(n+1)k$ binary variables and $mn(3+2m)+2mk$ constraints. The resultant optimized

$$\begin{aligned} \min \quad & \sum_{r=1}^{k} \sum_{j=0}^{2m-1} (v_r)_j \\ s.t. & \, (v_r)_j \geq (X'_r)_{i,j} \quad \forall r,i \\ & \sum_{r=1}^{k} X'_r = \mathbf{1}_{n,2m} \\ & (X'_p)_{i,j} = (X'_q)_{i,j+(-1)^j} \quad \forall i,r,j \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \\ & \mathbf{e}_i^T X'_p \mathbf{1}_{2m,1} = 1 \quad \forall i,r \quad \alpha_{i,r}\beta_{i,r} = x_p x_q \\ & (v_r)_j, (X'_r)_{i,j} \in \{0,1\} \end{aligned}$$

Fig. 7. BLP to minimize final SOP muxing.

matrices X'_r will then be used to construct the transformed Γ:

$$\mathbf{\Gamma}' = \sum_{r=1}^{k} x_r X'_r \quad (8)$$

B. Part II

Part I of the algorithm was not concerned with any of the signs $s_{i,r}$. We need to perform the bookkeeping of updating the signs given the transformation in Part I and proceed with the Fig. 2 to Fig. 5 transformation. To do so we construct the n by m matrix of signs:

$$\mathbf{S} = \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix}$$

We now need to extract the necessary information from X_r and X'_r in order to produce the transformed sign matrix S'. For practical purposes S' can be trivially created by inspecting the non zero terms in the corresponding rows of X_r and X'_r. However, for completeness, we present the matrix formulation which states how to construct S' a row at a time. Equation 9 contains the formula for calculating the ith row of S' *i.e.* $\mathbf{e}_i^T S'$:

$$\begin{aligned} P_r = X_r \left(I_m \otimes \mathbf{1}_{2,1}\right) \quad P'_r = X'_r \left(I_m \otimes \mathbf{1}_{2,1}\right) \\ \mathbf{e}_i^T S' \stackrel{def}{=} \mathbf{e}_i^T S \left(\bigcup_{r=1}^{k} \left(\left(\mathbf{e}_i^T P_r\right)^T \otimes \left(\mathbf{e}_i^T P'_r\right) \right) \right) \end{aligned} \quad (9)$$

Where $\otimes$ is the Kronecker product of matrices. P_r and P'_r are n by m matrices that contain where the products within the SOPs have been moved. The expression which is a union over matrices is a permutation matrix so (9) states that the ith row of S' is a permutation of the ith row of S. Having constructed S' we have now reached the point of algorithmically constructing the transformation resulting in Fig. 2. In this case the transformation to S' is:

$$S = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \Rightarrow S' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \quad (10)$$

In line with the comment in the problem statement we now look at the sign of the first product and use the identity $-x = \overline{x-1}$ to guarantee that this will only ever be positive. We split up the matrix S' into two matrices: an n by 1 vector GS of the global signs which is simply the first column of S' (these are the signs of the product $r = 0$ in (1)) and an n by m matrix LS of the local signs (signs of the products taking into account the global signs). Note that the first column of LS will be, by construction, zero.

$$\begin{aligned} GS &= S' e_1 \\ (LS)_{i,j} &= (S')_{i,j} \oplus (S')_{i,0} \end{aligned} \quad (11)$$

So for the example of the motivating SOPs:

$$S' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \Rightarrow GS = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} LS = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad (12)$$

C. Part III

The steps of the algorithm so far are:

1 Construct the binary matrices X_r of the original problem statement.
2 Solve the BLP found in Fig. 7 producing the optimized X'_r.
3 Construct the sign matrix S and then using X_r and X'_r compute S' according to (9).
4 Construct GS and LS from (11).
5 Construct Γ' according to (8).

We are now in a position to use these results in conjunction with the problem statement in (1) to state the result of the algorithm:

$$\alpha_r = \text{mux}(\Gamma' \mathbf{e}_{2r}, sel) \quad r \in \{0, 1, .., m-1\}$$
$$\beta_r = \text{mux}(\Gamma' \mathbf{e}_{2r+1}, sel) \quad r \in \{0, 1, .., m-1\}$$
$$y = \left(\left(\sum_{r=0}^{m-1} \beta_r(\alpha_r \oplus \text{mux}(LS\mathbf{e}_r, sel)) + \beta_r \text{mux}(LS\mathbf{e}_r, sel) \right) - \text{mux}(GS, sel) \right) \oplus \text{mux}(GS, sel) \quad (13)$$

Recall that $\alpha_r \oplus \text{mux}(LS\mathbf{e}_r, sel)$ results in a signed number, 1 bit larger than the size of α_r. Applying this to the motivating SOPs, combining (2), (6), (10) and (12):

$$\alpha_1 = \text{mux}((a, c, a, c), sel) \quad \alpha_2 = \text{mux}((c, a, c, a), sel)$$
$$\beta_1 = \text{mux}((b, b, b, b), sel) \quad \beta_2 = \text{mux}((1, 1, 1, 1), sel)$$
$$y = \Big(\beta_1(\alpha_1 \oplus \text{mux}((0, 0, 0, 0), sel)) + \beta_1 \text{mux}((0, 0, 0, 0), sel) + \beta_2(\alpha_2 \oplus \text{mux}((0, 1, 1, 0), sel)) + \beta_2 \text{mux}((0, 1, 1, 0), sel) - \text{mux}((0, 0, 1, 1), sel)\Big) \oplus \text{mux}((0, 0, 1, 1), sel)$$

We rely on standard synthesis tools to reduce this to:

$$\alpha_1 = sel[0]?c : a \quad \alpha_2 = sel[0]?a : c$$
$$y = \Big(\alpha_1 b + (\alpha_2 \oplus (sel[1] \oplus sel[0])) + (sel[1] \oplus sel[0]) - sel[1]\Big) \oplus sel[1]$$

which is identical in structure to Fig. 6.

D. Part IV

In order to facilitate the benchmarks used in the next section we relax two of the assumptions made during the algorithm.

1) SOPs with differing numbers of terms: Introduce two new symbols into the alphabet for every missing product: a zero symbol '0' and a don't care 'X' and use $0 \times X$ for the missing product. Given that 'X's do not contribute to muxing cost they do not need an associated variable vector v_r during the BLP.

2) SOPs with non distinct operands: In this case introduce new symbols into the alphabet for each duplicated version of the operand, but use the same additional variable v_r for all additional symbols; as the muxing cost is the same for all of the 'new' symbols.

VI. Benchmark SOPs

The set of benchmarks used are drawn from existing literature, a standard API and examples created to demonstrate the benefits of the algorithm. We use the motivating SOP y_1 in Fig. 1 as well as those found in Fig. 8. We use y_2 to demonstrate the value of the identity $-x = \overline{x-1}$, y_3 and y_4 come from [8], y_5 implements three integer operations from a standard graphics API, y_6 and y_7 purely exercise the BLP. Inputs are assumed to be unsigned 16 bit operands. The BLP was performed by the optimization software CPLEX 9.0.0 [10] and synthesis was performed by Synopsys' Design Compiler 2009.06-SP5 in ultra mode using the TSMC 65nm library Tcbn65lpwc. Design Compiler uses either a carry-save synthesis model or a delay-optimized flexible Booth Wallace architecture, the reduction and model selection is constraint and timing driven. Note that design y_6 required the largest solution of a BLP with 1152 binary variables and 1344 constraints, a design which is of an unusually large size but for which the BLP completes in 2 seconds. Table I contains area comparison figures for seven benchmark SOPs, for each pair of original and optimized design we have used the same loose timing constraint. We took a closer look at design y_1, we requested the synthesis tool to synthesize the design to achieve different delays; by applying Boolean optimization techniques and utilizing different standard cells, Design Compiler seeks the design with smallest area that meets the required delay. Thus we can see the full delay and area trade off of the motivating SOPs y_1 in Fig. 9. When both designs target minimal delay the optimized design is 16% faster. Fig. 9 also demonstrates that the optimized y_1 is strictly better throughout the range of delays presented and shows that the optimized design is not simply a different point on the area/delay curve of the original design. On average a 53% area improvement is achieved. The synthesis results of y_6 and y_7 demonstrate that the algorithm derives its real benefit from SOPs with differing numbers of terms as well as dealing with negative terms effectively.

TABLE I
Benchmark SOPs Synthesis Results.

SOP	Delay (ns)	Original RTL Area (μm^2)	Proposed RTL Area (μm^2)	% Improvement	Runtime BLP (s)
1	2.5	7834	3359	57	0.046
2	2.0	7500	3779	50	0.028
3	2.5	6520	3463	47	0.032
4	2.5	4319	2917	32	0.035
5	2.5	7343	3735	49	0.058
6	3.0	11471	11449	0	2.000
7	3.0	2947	2641	10	0.037

VII. Improvements and Future Work

Three assumptions remain from those made in the problem statement. Incorporating signed inputs means modifying the products in the final SOP to perform signed multiplication and performing sign extensions appropriately. SOPs with non identical word lengths would mean that the single SOP formed in the optimized design will contain non identical word lengths

$$
\begin{aligned}
y_2 &= (s == 0)? \quad ab : -ab \\
y_3 &= (s == 0)? \quad ab : \\
&\quad (s == 1)? \quad cd + e : \\
&\quad (s == 2)? \quad f + g : h - k \\
y_4 &= (s == 0)? \quad ab : \\
&\quad (s == 1)? \quad ab + e : \\
&\quad (s == 2)? \quad a + b : a - b \\
y_5 &= (s == 0)? \quad ab : \\
&\quad (s == 1)? \quad -ab : \\
&\quad (s == 2)? \quad c : \\
&\quad (s == 3)? \quad -c : \\
&\quad (s == 4)? \quad ab + c : \\
&\quad (s == 5)? \quad ab - c : \\
&\quad (s == 6)? \quad -ab + c : -ab - c \\
y_6 &= (s == 0)? \quad ab + cd + ef + gh : \\
&\quad (s == 1)? \quad bc + de + fg + ha : \\
&\quad (s == 2)? \quad cd + ef + gh + ab : \\
&\quad (s == 3)? \quad de + fg + ha + bc : \\
&\quad (s == 4)? \quad ef + gh + ab + cd : \\
&\quad (s == 5)? \quad fg + ha + bc + de : \\
&\quad (s == 6)? \quad gh + ab + cd + ef : \\
&\qquad\qquad\quad ha + bc + de + fg \\
y_7 &= (s == 0)? \quad ab + c : b + d
\end{aligned}
$$

Fig. 8. Benchmark SOPs.

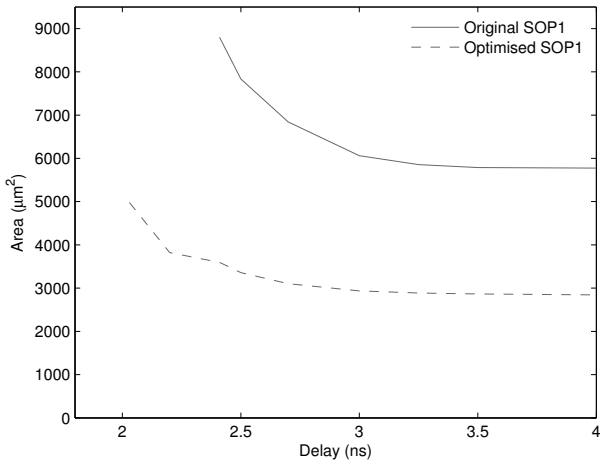

Fig. 9. Benchmark y_1 Delay Area Curves.

which in turn implies that some re-orderings of the individual SOPs are invalid as they will not 'fit' into the final SOP; this translates into forcing certain entries in X_r to zero.

The benchmarks considered did not suffer from BLP runtime issues, however the size of the alphabet may result in a BLP being incapable of solving the optimization problem within an acceptable time frame, in which case other optimization techniques would have to be explored. However, within the authors' experience, designs of the size that test the limits of the approach are beyond those found within industry. Fully incorporating constants into the algorithm is part of future work.

VIII. Conclusion

We have presented a new algorithm for resource sharing mutually exclusive sum-of-products that can deliver substantial delay and area benefits. Hence the approach is applicable to arithmetic computations in timing critical and non timing critical datapath domains. The algorithm presented can be extended to any general sum-of-product expressions and runs in acceptable time for the industrial size benchmarks. It is thus believed that such an approach would be of significant benefit to high level datapath synthesis.

Acknowledgment

The authors would like to acknowledge Imagination Technologies Ltd for supporting this research and the DATE 2011 reviewers for their feedback.

References

[1] K. P. Acken, M. J. Irwin, R. M. Owens, and A. K. Garga, "Architectural optimizations for a floating point multiply-accumulate unit in a graphics pipeline," in *ASAP: International Conference on Application Specific Systems, Architectures and Processors*, August 1996, pp. 65–71.

[2] S. Das and S. P. Khatri, "A timing-driven synthesis approach of a fast four-stage hybrid adder in sum-of-products," in *MWSCAS: 51st Symposium on Circuits and Systems*, August 2008, pp. 507–510.

[3] ——, "An inversion-based synthesis approach for area and power efficient arithmetic sum-of-products," in *VLSID: 21st International Conference on VLSI Design*, January 2008, pp. 653–659.

[4] D. Kumar and B. Erickson, "Asop: Arithmetic sum-of-products generator," in *ICCD: IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1994, pp. 522–526.

[5] R. Zimmerman and D. Q. Tran, "Optimized synthesis of sum-of-products," in *37th Asilomar Conference on Signals, Systems and Computers*, vol. 1, November 2003, pp. 867–872.

[6] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," in *ICCAD: IEEE/ACM International Conference on Computer Aided Design*, November 2004, pp. 791–798.

[7] S. Das and S. P. Khatri, "A merged synthesis technique for fast arithmetic blocks involving sum-of-products and shifters," in *VLSID: 21st International Conference on VLSI Design*, January 2008, pp. 572–579.

[8] ——, "Resource sharing among mutually exclusive sum-of-product blocks for area reduction," *TODAES: ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 51–57, July 2008.

[9] S. Jain *et al.*, "A 90mw/gflop 3.4ghz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm," in *VLSID: 23rd International Conference on VLSI Design*, January 2010, pp. 252–257.

[10] *ILOG Cplex 9.0 User's Manual*, ILOG, October 2003.