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 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].

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 \((-1)^{\neg a}a = a \oplus \neg a\) where \(t = a \oplus \neg a\) is a signed number 1 bit larger than \(a\) such that if \(a\) is \(n\) bits in length then \(t[i] = a[i] \text{ XOR } \neg a\) for \(n > i \geq 0\) and \(t[n] = \neg a\). 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 \(-x = x - 1\). 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 formalise 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 mixes between \(n\) SOPs, each with \(m\) products and where the operands are drawn from an alphabet of \(k\) elements, \(\{x_1, x_2, ..., x_k\}\):

\[
y_1 = (s == 0)? \, ab + c : \\
= (s == 1)? \, bc - a : \\
= (s == 2)? \, c - ab : \neg a - bc
\]

Fig. 1. Sample mutually exclusive SOPs.

\[
y_1 = (s == 0)? \, ab + c : \\
= (s == 1)? \, cb - a : \\
= (s == 2)? \, -ab + c : \neg cb - a
\]

Fig. 2. Reordered sample SOPs.

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

Fig. 3. Merged SOPs: First Attempt.

\[
ab + c  \\
= cb - a = cb + \overline{c} + 1  \\
-ab + c = -(ab + \overline{c} + 1) = ab + \overline{c}  \\
- cb - a = -(cb + a) = cb + a - 1
\]

Fig. 4. SOP Rewriting.

\[
y_1 = (s == 0)? \, ab + c : \\
= (s == 1)? \, cb + \overline{c} + 1 : \\
= (s == 2)? \, \overline{ab} + \overline{b} + \overline{c} : \overline{cb} + a - 1
\]

Fig. 5. Premerged SOPs: Second Attempt.

\[
A = s[0]?c : a  \\
C = (s[0]?a : c) \oplus (s[1] \oplus s[0])  \\
y_1 = (Ab + C + s[1] \oplus s[0] - s[1]) \oplus s[1]
\]

Fig. 6. Optimized sample SOPs.
For ease of notation we will use the following nomenclature: Where
\( \alpha \) and \( \beta \) are the result of muxing between the \( k \) operands and producing the terms that will produce the stated above will be relaxed.

This issue will be addressed in Part IV where the assumptions it will be assumed that we make the first product positive. the functions \( g \) found in Fig. 6. In particular we seek transformations such that subsequent transformations (Part II).

The first step is to minimize the operand muxing; to minimize 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 \( \Gamma \) into \( k \) binary matrices \( X_r \) corresponding to the alphabet present, such that:

\[
\Gamma = \sum_{r=1}^{k} x_r X_r
\]

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

\[
\Gamma = a \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix} + b \begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} + c \begin{pmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

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 \( 1_{n_1,n_2} \) represent an \( n_1 \) by \( n_2 \) matrix entirely consisting of ones and \( e_i \) represent the \( i \)th 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:

\[
\sum_{r=1}^{k} X_r = 1_{n,2m}
\]

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 \( r \)th SOP, the \( x_q \) location is then fixed. This corresponds to the following restriction:

\[
(X_p)_{i,j} = (X_q)_{i,j+(-1)i} \quad \forall i,r,j \quad \alpha_{i,r} \beta_{i,r} = x_p x_q
\]

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 \( i \)th row of matrix \( X_p \):

\[
e_i^T X_p 1_{2m,1} = 1 \quad \forall i,r \quad \alpha_{i,r} \beta_{i,r} = x_p x_q
\]

Note that there are \( mn \) 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:

$$\mathbf{X} = \begin{pmatrix} a & b & c & 1 \\ b & c & a & 1 \\ c & a & b & 1 \\ a & 1 & b & c \end{pmatrix} \Rightarrow \mathbf{\Gamma}' = \begin{pmatrix} a & b & c & 1 \\ c & b & a & 1 \\ a & b & c & 1 \\ c & b & a & 1 \end{pmatrix}$$

(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:

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

(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:

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

s.t. $\sum_{r=1}^{k} X_r' = 1_{n,2m}$

$(X_r^t)_{i,j} = (X_r^t)_{i,j+(-1)}$ \quad $\forall i, r, j, \quad \alpha_i \beta_{i,r} = x_p x_q$

$e_r^T X_p 1_{2m,1} = 1$ \quad $\forall i, r, \quad \alpha_i \beta_{i,r} = x_p x_q$

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 matrices $X_r'$ will then be used to construct the transformed $\mathbf{\Gamma}'$:

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

(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 $n \times m$ matrix of signs:

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

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 $i$th row of $S'$ i.e. $e_i^T S'$:

$$P_r = X_r (I_m \otimes 1_{2,1}) \quad P_r' = X_r' (I_m \otimes 1_{2,1})$$

$$e_i^T S' = e_i^T S \bigcup_{r=1}^{k} (e_i^T P_r) \otimes (e_i^T P_r')$$

(9)

Where $\otimes$ is the Kronecker product of matrices. $P_r$ and $P_r'$ are $n \times m$ 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 $i$th row of $S'$ is a permutation of the $i$th 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:

$$S = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \Rightarrow S' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

(10)

In line with the comment in the problem statement we now look at the sign of the first product and use the identity $-x = x - 1$ to guarantee that this will only ever be positive. We split up the matrix $S'$ into two matrices: an $n \times 1$ 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 $n \times m$ 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.

$$GS = S'e_1$$

$$\quad (LS)_{i,j} = (S')_{i,j} \oplus (S')_{i,0}$$

(11)

So for the example of the motivating SOPs:

$$S' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \Rightarrow GS = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$LS = \begin{pmatrix} 0 & 0 \end{pmatrix}$$

(12)
Delay 3779 57 0 2641 50
R TL Area 3.0 47 2.5
(0.046 2917 0.037 3.0 7500 3735 2947 2.0 %
vement 6520 (49 4319 Impro-
32 2.000 7834 Runtime 0.058 3463 2.5 2.5 2.5
additional symbols; as the muxing cost is the same for all of
the operand, but use the same additional variable
new symbols into the alphabet for each duplicated version of
cost they do not need an associated variable
missing product. Given that ‘X’s do not contribute to muxing
zero symbol ‘0’ and a don’t care ‘X’ and use
new symbols into the alphabet for every missing product: a

D. Part IV

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 \(\Gamma’\) 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:

\[
\alpha_r = \text{mux}(\Gamma’ \epsilon_{2r}, \text{sel}) \quad r \in \{0, 1, \ldots, m - 1\}
\]

\[
\beta_r = \text{mux}(\Gamma’ \epsilon_{2r+1}, \text{sel}) \quad r \in \{0, 1, \ldots, m - 1\}
\]

\[
y = \left( \sum_{r=0}^{m-1} \beta_r (\alpha_r \oplus \text{mux}(LS e_r, \text{sel})) + \beta_r \text{mux}(LS e_r, \text{sel}) \right)
    - \text{mux}(GS, \text{sel}) \oplus \text{mux}(GS, \text{sel})
\]

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

\[
\alpha_1 = \text{mux}((a, c, a, c, c, a, \text{sel}) \quad \alpha_2 = \text{mux}((c, c, a, a, \text{a, sel})
\]

\[
\beta_1 = \text{mux}((b, b, b, b, a, \text{sel}) \quad \beta_2 = \text{mux}((1, 1, 1, 1, 1, \text{sel})
\]

\[
y = \left( \beta_1 (\alpha_1 \oplus \text{mux}((0, 0, 0, 0, \text{a, sel}) + \beta_1 \text{mux}((0, 0, 0, 0, \text{sel}) +
    \beta_2 (\alpha_2 \oplus \text{mux}((0, 1, 1, 0, \text{sel}) + \beta_2 \text{mux}((0, 1, 1, 0, \text{sel}) -
\text{mux}((0, 0, 1, 1, 1, \text{sel}) \oplus \text{mux}((0, 0, 1, 1, 1, \text{sel})
\]

We rely on standard synthesis tools to reduce this to:

\[
\alpha_1 = \text{sel}[0]?c:\text{a} \quad \alpha_2 = \text{sel}[0]?a:\text{c}
\]

\[
y = \left( \alpha_1 b + (\alpha_2 \oplus \text{sel}[1] \oplus \text{sel}[0]) +
(\text{sel}[1] \oplus \text{sel}[0]) - \text{sel}[1] \right) \oplus \text{sel}[1]
\]

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 \(v_r\) 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 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 \(-x = x - 1\), \(y_1\) and \(y_4\) come from [8], \(y_5\)
implies 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

<table>
<thead>
<tr>
<th>SOP</th>
<th>Delay (ns)</th>
<th>Original RTL Area (µm²)</th>
<th>Proposed RTL Area (µm²)</th>
<th>% Improvement</th>
<th>Runtime BLP (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5</td>
<td>7834</td>
<td>3359</td>
<td>57</td>
<td>0.046</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>7500</td>
<td>3779</td>
<td>50</td>
<td>0.028</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>6520</td>
<td>3463</td>
<td>47</td>
<td>0.032</td>
</tr>
<tr>
<td>4</td>
<td>2.5</td>
<td>4319</td>
<td>2917</td>
<td>32</td>
<td>0.035</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>7343</td>
<td>3735</td>
<td>49</td>
<td>0.058</td>
</tr>
<tr>
<td>6</td>
<td>3.0</td>
<td>11471</td>
<td>11449</td>
<td>0</td>
<td>2.000</td>
</tr>
<tr>
<td>7</td>
<td>3.0</td>
<td>2947</td>
<td>2641</td>
<td>10</td>
<td>0.037</td>
</tr>
</tbody>
</table>

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

Fig. 8. Benchmark SOPs.

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