Progressive Decomposition: A Heuristic to Structure Arithmetic Circuits

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Philip Brisk
Philip.Brisk@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Federale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

ABSTRACT

Despite the impressive progress of logic synthesis in the past decade, finding the best architecture for a given circuit still remains an open problem and largely unsolved. In most of the arithmetic circuits the outcome of the synthesis tools depends on the input description of the circuit. In other words, logic synthesis optimisations hardly change the architecture of the given circuit. However, once the input description belongs to the right architecture, logic synthesis does an excellent job in optimising the circuit locally. This is the reason why designers still rely on well studied architectures. The main difficulty in finding the suitable architecture for an arithmetic circuit is the high fan-in dependencies between inputs and outputs (i.e., each output bit depends on a large portion of input bits). Hence, imposing hierarchy and structure is the key to find the best architecture. Although factorisation is one potential solution for this problem, the computational complexity of Boolean factorisation and poor performance of algebraic factorisation make this solution impractical in most cases of interest. In this paper we present a novel approach which progressively decomposes the input circuits into building blocks and constructs hierarchy among these blocks. We show that our approach optimises the critical path delay by 15–30% at the cost of marginal or no area penalty. In some cases, it even improves the area. Qualitatively we observed that our approach found the best known architecture for some circuits without any a priori knowledge about the functionality of the circuit.

Categories and Subject Descriptors

B.2.4 [**Arithmetic and Logic Structures**]: High Speed Arithmetic -algorithms, cost/performance

General Terms

Algorithms, Performance, Design

Keywords

Progressive Decomposition, Boolean Ring, Factorisation

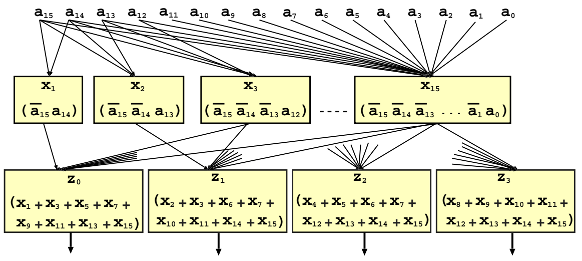
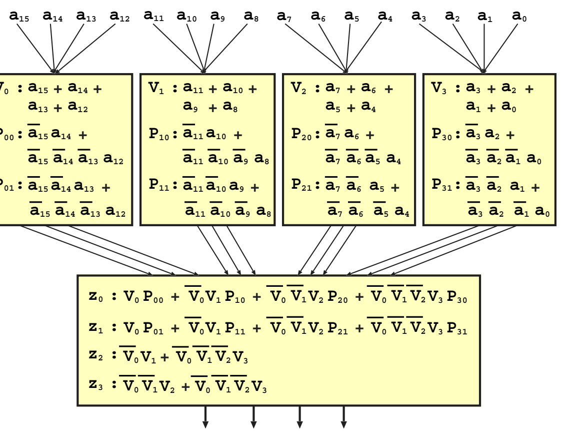

a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
x_1 $(\overline{a}_{15} a_{14})$
x_2 $(\overline{a}_{15} \overline{a}_{14} a_{13})$
x_3 $(\overline{a}_{15} \overline{a}_{14} \overline{a}_{13} a_{12})$
x_{15} $(\overline{a}_{15} \overline{a}_{14} \overline{a}_{13} \dots \overline{a}_1 a_0)$
z_0 $(x_1 + x_3 + x_5 + x_7 + x_9 + x_{11} + x_{13} + x_{15})$
z_1 $(x_2 + x_3 + x_6 + x_7 + x_{10} + x_{11} + x_{14} + x_{15})$
z_2 $(x_4 + x_5 + x_6 + x_7 + x_{12} + x_{13} + x_{14} + x_{15})$
z_3 $(x_8 + x_9 + x_{10} + x_{11} + x_{12} + x_{13} + x_{14} + x_{15})$

Figure 1: Illustration of high fan-in dependencies in LZD.

1. MOTIVATION

A classic example of the circuits where hierarchy matters is the *Leading Zero Detector* (LZD). The LZD, which is used in the floating-point normalisation process, takes an integer as input and returns another integer which tells the position of the first nonzero bit from the left in the input integer. Fig. 1 shows a straightforward implementation of the 16-bit LZD. In this implementation, the Boolean variables $x_1, x_2, \dots x_{15}$ are computed, where x_i is true if the leading nonzero entry is at the i^{th} position. The values of x_i's are used to determine the output bits as shown in Fig. 1. For example, the first least significant bit of the output will be one if the first nonzero entry occurs at an odd position (i.e., one of the variables $x_1, x_3, \dots x_{15}$ is true). As we can see, there is a huge number of interconnections between the primary inputs and the blocks computing x_i's, as well as between x_i's and the blocks computing z_i's. Such a high number of interconnections makes this implementation complex and inferior. On the other hand, Fig. 2 shows a design proposed by Oklobdzija [8] for the same circuit. This method divides the 16 bit integer into four blocks, each containing four bits. For each block the variables V_i, P_{i0}, and P_{i1} are computed. The variable V_i is true if any of the four bits of the corresponding block is true, and the the bits P_{i0} and P_{i1} are such that the two-bit integer $\langle P_{i1} P_{i0} \rangle$ gives the position of leading nonzero entry in the corresponding block. At the next level the values of V_i, P_{i0}, and P_{i1} are used to compute the output bits. For example, if V_0 is true, then the output integer will be $\langle 00 P_{01} P_{00} \rangle$; if V_0 is false, and V_1 is true, then the output will be $\langle 01 P_{11} P_{10} \rangle$, and so on. As we can see, each of the four blocks which computes V_i, P_{i0}, and P_{i1} uses only four bits of the inputs. This makes the number of interconnections extremely small compared to the previous design, resulting in a regular, structured, and low fan-in circuit.

For exploiting the regularity and structure in a circuit, designers rely on manual techniques, which makes these techniques applicable only to limited circuits. What is desired, is to have an auto-



a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
$V_0 : a_{15} + a_{14} + a_{13} + a_{12}$
$P_{00} : \overline{a}_{15} a_{14} + \overline{a}_{15}\overline{a}_{14}\overline{a}_{13} a_{12}$
$P_{01} : \overline{a}_{15}\overline{a}_{14} a_{13} + \overline{a}_{15}\overline{a}_{14}\overline{a}_{13} a_{12}$
$V_1 : a_{11} + a_{10} + a_9 + a_8$
$P_{10} : \overline{a}_{11} a_{10} + \overline{a}_{11}\overline{a}_{10}\overline{a}_9 a_8$
$P_{11} : \overline{a}_{11}\overline{a}_{10} a_9 + \overline{a}_{11}\overline{a}_{10}\overline{a}_9 a_8$
$V_2 : a_7 + a_6 + a_5 + a_4$
$P_{20} : \overline{a}_7 a_6 + \overline{a}_7\overline{a}_6\overline{a}_5 a_4$
$P_{21} : \overline{a}_7\overline{a}_6 a_5 + \overline{a}_7\overline{a}_6\overline{a}_5 a_4$
$V_3 : a_3 + a_2 + a_1 + a_0$
$P_{30} : \overline{a}_3 a_2 + \overline{a}_3\overline{a}_2\overline{a}_1 a_0$
$P_{31} : \overline{a}_3\overline{a}_2 a_1 + \overline{a}_3\overline{a}_2\overline{a}_1 a_0$
$z_0 : V_0 P_{00} + \overline{V}_0 V_1 P_{10} + \overline{V}_0\overline{V}_1 V_2 P_{20} + \overline{V}_0\overline{V}_1\overline{V}_2 V_3 P_{30}$
$z_1 : V_0 P_{01} + \overline{V}_0 V_1 P_{11} + \overline{V}_0\overline{V}_1 V_2 P_{21} + \overline{V}_0\overline{V}_1\overline{V}_2 V_3 P_{31}$
$z_2 : \overline{V}_0 V_1 + \overline{V}_0\overline{V}_1\overline{V}_2 V_3$
$z_3 : \overline{V}_0\overline{V}_1 V_2 + \overline{V}_0\overline{V}_1\overline{V}_2 V_3$

Figure 2: A hierarchical implementation of LZD built on smaller blocks. Note the low fan-in dependencies compared to straightforward implementation.

mated tool which takes the specification of input circuit in any form and gives an hierarchical implementation (if there is one) built of smaller blocks. In this paper we present a method which we call *Progressive Decomposition*.

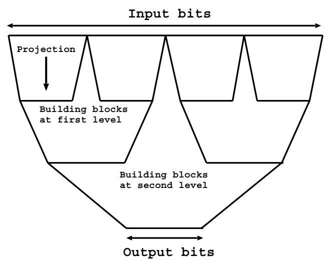
The rest of the paper is organised as follows: In the next section, we discuss some earlier work related to this topic. Section 3 discusses the main idea to find an appropriate architecture for the given circuit, and also argues about its applicability on a wide range of circuits. In Section 4, we present some results from abstract algebra, which we will use in our algorithm. Section 5 presents the algorithm in detail. Finally in Section 6, we show the results of our experiments, followed by concluding remarks in Section 7.

2. STATE OF THE ART

To date, most circuit optimisations have been performed manually using ad-hoc techniques that could, in principle, be generalised. Ad-hoc techniques have been applied to virtually all circuits of interest [9]; some particular notable examples are Oklobdzija's Leading Zero Detector circuit [8] and Wallace's use of Carry-Save adders for multi-input addition [13].

Algorithmic approaches for circuit optimisation are limited to a small domain, such as the variable group size *Carry-Lookahead Adder* [7] and irregular partial-product compressors for multipliers [10]. There has also been work on circuit rewriting techniques [11] to merge previously disjoint addition operations into multi-input adders implemented efficiently with a compressor tree. Verma and Ienne [12] have also developed a tool that enumerates all possible architectures for a circuit via exhaustive search; however, the high computational complexity limits the applicability of the technique to small circuits.

Multi-level optimisation [4] attempts to find suitable architectures for circuits by extracting kernels and cokernels using factorisation techniques proposed by Brayton [2, 3]. The kernels are similar in principle to the building blocks discussed here; however, the method for kernel extraction is based on algebraic division applied to Boolean functions in sum-of-product form. Most arithmetic circuits, in contrast, are XOR-dominated, exposing a weakness of algebraic division. Boolean division [3], on the other hand, is a superior method for finding blocks, but has a prohibitive computational complexity (if used indiscriminately). Our approach, in contrast, uses the Reed-Muller form [3] (XOR-of-product form) and uses a combination of algebraic and Boolean division.



Input bits
Projection
Building blocks at first level
Building blocks at second level
Output bits

Figure 3: Illustration of the main idea to optimise the circuit by imposing hierarchy and structure.

3. BUILDING BLOCK APPROACH: IDEA AND APPLICABILITY

The main idea in our approach is to build the circuit using building blocks, as shown in Fig. 3. This means the input bits are divided into several groups of size k, for some fixed k. Next, for each of these groups some *leader expressions* are computed, such that any information about the bits of a group must be accessed through its leader expressions (In other words, to evaluate the value of whole expression one does not need to know the value of all input bits, but only of leader expressions). Note that this immediately restricts the high fan-out load on primary inputs.

Once the leader expression for all groups are computed, they are considered as new input bits, and their groups are formed to compute new leader expressions. This process is continued till the final leader expressions correspond to the output bit expressions. These leader expressions are our building blocks. It is easy to see that the implementation generated by building block approach is structured and hierarchical.

At this point one might think that application of this approach is very limited, because there are very few circuits which can be implemented using building blocks. However, the next theorem says that this approach is applicable in a wide range of applications. We define *online algorithm* for computing an expression an algorithm which assumes that the input bits are provided serially and not all at once. An *effective online algorithm* is an online algorithm which takes constant amount of time (independent from the precision) to compute the output since the last input bit has arrived.

THEOREM 1. *Any circuit which has an effective online algorithm, will also have a hierarchical structure, and can be implemented using leader expressions.*

Due to lack of space we omit the proof of above theorem and explain it using a simple example. For the sake of simplicity let us assume that there is a single output bit. The fact that the time taken to compute the output since the last input bit arrival is constant implies that the corresponding online algorithm must have a constant number of precomputed expressions of the earlier bits which are used for the computation of the output bit and the new set of precomputed expressions. Assume that this constant is bounded by c. Using this fact one can construct a hierarchical implementation to compute the output bit.

To make the situation even simpler, assume that $c = 1$. In other words to evaluate the expression one needs to know only one bit of information from the set of first m input bits (for any arbitrary m). Now if we divide the input bits into groups of k bits in the order they arrive, each group will need exactly one bit information from the previous group, and will pass exactly one bit information to the next group as shown in Fig. 4. The conditioned values (according

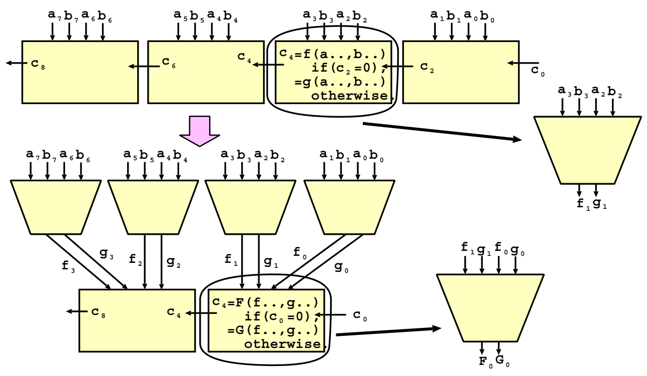


Figure 4: Any circuit having an effective online algorithm also has a hierarchical structure.

```
progressiveDecomposition (List $\mathcal{L}$) {
// The function takes the list of input expressions
// builds the hierarchy to implement them.
  identities = $\phi$;
  while (true) {
    G = findGroup($\mathcal{L}$, k);
    (B, C) = findBasis($\mathcal{L}$, G, identities);
    // This function returns a basis and coefficients
    // of its elements.
    (B, C) = minimizeBasisUsingLinearDependence(B, C);
    (B, C) = improveBasisUsingSizeReduction(B, C);
    identities = identities $\cup$ findIdentities(B);
    B = ReduceBasisUsingIdentities(B, identities);
    $\mathcal{L}$ = rewriteExpr($\mathcal{L}$, B);
    identities = rewriteExpr(identities, B);

    if(all elements in $\mathcal{L}$ are literals) break; } }
```

Figure 5: The Progressive Decomposition algorithm.

to the incoming bit) of this output bit are the only informations that one needs to know from the k input bits of the corresponding group. Hence the conditioned values can be treated as leader expressions at first level. In a similar way the leader expressions at second and subsequent levels can be formed as shown in Fig. 4 resulting in a structured and hierarchical implementation.

Since most of the arithmetic circuits have effective online algorithms as shown in [5], we can say that the building block approach has application to a wide range of circuits.

4. SOME RESULTS FROM ALGEBRA

In this section we discuss some well known results from algebra [6], which we use in our algorithm.

Boolean Ring and its consequences: As we mentioned in Section 2, before doing any operation on the Boolean expressions corresponding to the input circuit, we convert them into Reed-Muller form (i.e., XOR-of-product form). There are several advantages of Reed-Muller form over the sum-of-product form: First the Reed-Muller form of an expression is unique, hence, the output of our algorithm is independent of the input description of the circuit. The second advantage is that Boolean expressions in Reed-Muller form form a ring under the operations XOR and AND (known as Boolean Ring). This means we can use several properties of the ring to optimise the Boolean expressions, e.g., a set of Boolean expressions is linearly dependent if one of these expressions can be written as the XOR of a subset of the rest of the expressions (note that there are efficient ways to check linear dependence in rings). This property is useful because it helps in minimising the number of leader expressions. For example, if the leader expressions corresponding to a group of inputs bits are linearly dependent, then the one which can be written as XOR of others can be removed from the set of leader expressions.

Null-Space of a Boolean Expression: For a given expression P, all the expressions X which satisfy $PX = 0$ belong to the null-space of P, denoted as $N(P)$. For example if $P = a \oplus b$, then ab will belong to the null-space of P ($ab(a \oplus b) = 0$). It can be easily seen that the null-space of an expression forms a ring. One can also define $N(P) \oplus N(Q)$ and $N(P) \cdot N(Q)$ as follows:

$$N(P) \oplus N(Q) = \{x \oplus y \mid x \in N(P), y \in N(Q)\},$$
$$N(P) \cdot N(Q) = \{xy \mid x \in N(P), y \in N(Q)\}.$$

Here are two important properties about null-spaces:

$$N(P) \oplus N(Q) \text{ also forms a ring, and}$$
$$N(P) \cdot N(Q) \subseteq N(P \oplus Q).$$

Using these properties one can factorise Boolean expressions more effectively, resulting in better building blocks. For example, consider the expression $X = (a \oplus b)(p \oplus cd) \oplus (c \oplus d)(p \oplus ab)$. In terms of algebraic factorisation this expression is irreducible; however, using the properties of null-space, this expression can be factorised as shown below:

$$ab \in N(a \oplus b)$$
$$\Rightarrow (a \oplus b)(p \oplus cd) = (a \oplus b)(p \oplus ab \oplus cd) \text{ and}$$
$$cd \in N(c \oplus d)$$
$$\Rightarrow (c \oplus d)(p \oplus ab) = (c \oplus d)(p \oplus ab \oplus cd).$$

Hence, $X = (a \oplus b \oplus c \oplus d)(p \oplus ab \oplus cd)$. In general, if $X = PQ \oplus RS$ and $(Q \oplus S) \in (N(P) \oplus N(R))$, then the expression X can be factorised as $X = (P \oplus R)T$, for some expression T. Since $(N(P) \oplus N(R))$ is a ring, the membership of $(Q \oplus S)$ can be checked easily using the *Ideal Membership Problem* [1]. However, we need to know the null-spaces of various expressions, which we compute incrementally. In the above case, we compute the null-space of $(P \oplus R)$, which is the new factor of expression X using $N(P)$ and $N(R)$. According to the second property mentioned above, $N(P \oplus R)$ will contain the smallest ring containing $N(P) \cdot N(R)$, denoted as rC$(N(P) \cdot N(R))$. However, it might contain some other elements also, but we can conservatively assume the two identical.

5. ALGORITHM

We have seen in Section 3 a method to impose hierarchy on a given circuit. However, there are certain problems with the method proposed there: First it requires the prior knowledge of an effective online algorithm for the circuit, which is not available for general circuits. The second problem is that there is no attempt to minimise or optimise building blocks. As a result, the imposed hierarchical structure may be much worse than the optimal circuit design. For these reasons we propose a new method to build the hierarchy, and call it *Progressive Decomposition*. The new method does not require any knowledge about the input circuit, and also tries to optimise the set of building blocks.

The overall description of Progressive Decomposition is shown in Fig. 5. The algorithm takes a list of expressions as input and iteratively finds building blocks to impose the structure and hierarchy. The algorithm stops when all the output expression are reduced to a literal. In each iteration it selects a group of k bits, and finds the minimal number leader expressions of the k bits. From now on, we call this set of leader expressions *basis*. After finding the first basis, a set of procedures are applied to optimise the basis and finally the original expressions are rewritten by replacing occurrences of the basis elements by new variables. We discuss each of these procedures individually in the coming sections.

5.1 Finding a Group

This is the first procedure, which takes a list of expressions $\mathcal{L}$ and an integer k, and returns a group of k bits. One possibility to implement this function can be to try all possible k-bit groups, and choose the one for which the rewritten expression at the end of the iteration has the smallest size in terms of number of literals.

However we use a different approach in order to reduce the time complexity. In most cases the input bits correspond to the bits of integers, where normally building blocks are built on contiguous bits. In our method, if the circuit has r input integers and some of these input bits are still visible in the expressions, then we choose $\lfloor k/r \rfloor$ least significant available bits from each integer (note that this might leave us with a group of size less than k). Once all the primary input bits are exhausted, then to find the group of k bits we try all possible combinations of k-bits as a group. Since at this point the size of the expressions is significantly smaller than the size of original expressions, even the exhaustive search for groups does not increase the time complexity significantly. In our experiments we always use $k = 4$ but different values of k can be used.

5.2 Finding a Basis

This procedure takes a list of expressions, a group of variables, and a set of Boolean identities, and returns the basis containing only the given variables. This procedure is very similar to the kernel extraction algorithm in algebraic factorisation. Yet, here the expressions are in XOR-of-product form and we also have a set of identities. We explain this procedure using an example.

Suppose that the input expression is

$$X = ad \oplus aef \oplus bcd \oplus abe \oplus ace \oplus bcef \oplus xy.$$

Also assume that the group of variables we are interested in is $\{a, b, c\}$. In the given expression consider all the product terms, which contain any of the variables $\{a, b, c\}$. For each of these product terms we make a pair by splitting the product term into two product terms: one containing only variables from $\{a, b, c\}$, and the other containing only the rest of the variables. All these pairs are stored in a list. In the above example this list $\mathcal{A}$ will look like this:

$$\mathcal{A} = \{(a, d), (a, ef), (bc, d), (ab, e), (ac, e), (bc, ef)\}.$$

Next we try to reduce the size of this list by merging several pairs into one. If the set of given identities is null, then merging corresponds to replacing two pairs (α, γ) and (β, γ) by a single pair $(\alpha \oplus \beta, \gamma)$ as well as replacing (α, β) and (α, γ) by a single pair $(\alpha, \beta \oplus \gamma)$. This will lead us to reduce the above list to

$$\mathcal{A}' = \{(a \oplus bc, d \oplus ef), (ab \oplus ac, e)\}.$$

Finally the set containing the first element of each pair is returned as a basis. In this case the basis is $\{a \oplus bc, ab \oplus ac\}$.

On the other hand when we the set of identities is nonnull, then we can use the null-space properties mentioned in previous section to merge the pairs of list more aggresively. As an example suppose the original expression is

$$X = ap \oplus bp \oplus cp \oplus ax \oplus ay \oplus by \oplus bz \oplus cx \oplus cz.$$

The group variables are $\{a, b, c\}$ and the set of identities is $\{az = 0, bx = 0, cy = 0\}$. Based on these identities we find the null-space of each of the variable in the group. In this case $N(a) =$ rC(z), $N(b) =$ rC(x), and $N(c) =$ rC(y). Before doing any merging, the list of pairs looks like:

$$\mathcal{A} = \{(a,p), (a,x), (a,y), (b,p), (b,y), (b,z), (c,p), (c,z), (c,x)\}, \text{ and}$$
$$\mathcal{A}' = \{(a, p \oplus x \oplus y), (b, p \oplus y \oplus z), (c, p \oplus x \oplus z)\}.$$

We check now if any of the two pairs can be merged. Let's consider the first two pairs. Since $(p \oplus x \oplus y) \oplus (p \oplus y \oplus z) = (x \oplus z) \in (N(a) \oplus N(b))$, according to the proposition of the previous section $a(p \oplus x \oplus y) \oplus b(p \oplus y \oplus z)$ can be factorised as $(a \oplus b)(p \oplus x \oplus y \oplus z)$, i.e., the two pairs can be merged into a single pair $(a \oplus b, p \oplus x \oplus y \oplus z)$. Also, now we have to update the null-space of $(a \oplus b)$. Once again, according to the property of Section 4, we conseratively assign $N(a \oplus b) =$ rC(xz). Using the same approach for the remaining two pairs finally we reduce the list to

$$\mathcal{A}_{\text{final}} = \{(a \oplus b \oplus c, p \oplus x \oplus y \oplus z)\}.$$

Although, the above mentioned procedure computes the basis of a single expression, the same technique can be used when the input list has more than one expression. Suppose the input list has m expressions named $P_1, P_2, \ldots, P_m$, then we first construct a single expression $X = K_{P_1}P_1 \oplus K_{P_2}P_2 \oplus \cdots \oplus K_{P_m}P_m$, where $K_{P_1}, K_{P_2}, \ldots, K_{P_m}$ are new variables. Next we find the basis for this new expression, which is the basis of the list of expressions.

5.3 Minimizing the Basis using Linear Dependencies

In the above procedure, if the final list is such that either the set of first elements or the set of second elements of the pairs is linearly dependent, then the size of the basis can be reduced. More generally, suppose that the final list output by above procedure is

$$\mathcal{A} = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_m, Y_m)\}.$$

Now, if the set $\{X_1, X_2, \ldots X_m\}$ is linearly dependent, then one of its element can be written as XOR of some other elements of the set. Without loss of generality we can assume that X_1 can be written as the XOR of $X_2, X_3, \ldots X_n$. In this case the list $\mathcal{A}$ can be modified to

$$\mathcal{A} = \{(X_2, Y_1 + Y_2), (X_3, Y_1 + Y_3), \ldots, (X_n, Y_1 + Y_n), \ldots, (X_m, Y_m)\}.$$

In this case we have reduced the size of basis by one. In a similar way if the set $\{Y_1, Y_2, \ldots Y_m\}$ is linearly dependent, and Y_1 can be written as XOR of $Y_2, Y_3, \ldots Y_n$, then the final list can be modified in the following way resuling in unit reduction in the size of basis:

$$\mathcal{A} = \{(X_1 + X_2, Y_2), (X_1 + X_3, Y_3), \ldots, (X_1 + X_n, Y_n), \ldots, (X_m, Y_m)\}.$$

An example of this kind of minimisation occurs in the leading zero detector: the original basis found by the above procedure is $\{V_0, P_{00}, P_{01}, V_0 + P_{00}, V_0 + P_{01}\}$, V_0, P_{00}, and P_{01} are the expressions defined in Section 1. Using linear dependencies among the original basis, it can be reduced to a new basis $\{V_0, P_{00}, P_{01}\}$.

5.4 Improving the Basis via Size Reduction

Sometimes when the expression is not very regular, it might be possible that, in the final list of pairs, two pairs cannot be merged just because of some small difference between the first or second elements of the pair. The following example illustrates such case. Assume that the final list output by the above procedures is

$$\mathcal{A} = \{(a, p \oplus q \oplus r \oplus s \oplus t), (b, p \oplus q \oplus r \oplus s)\}.$$

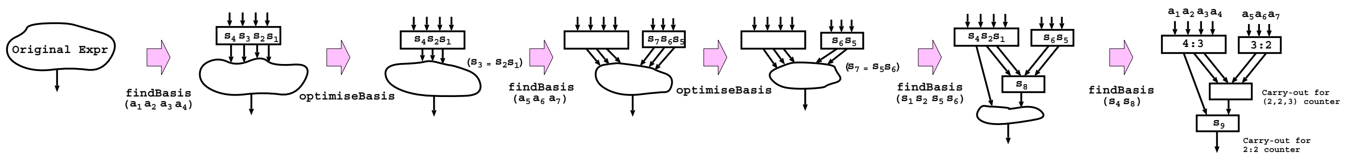


Figure 6: Execution of Our algorithm on the 7-bit majority function. The function findBasis is the same as the procedure mentioned in Section 5.2. On the other hand optimiseBasis is the cumulative form of optimisations mentioned in Sections 5.3, 5.4 and 5.5.

In this case neither the set of first elements $\{a, b\}$ is linearly dependent, nor is the set of second elements $\{p \oplus q \oplus r \oplus s \oplus t, p \oplus q \oplus r \oplus s\}$. Hence, the basis minimisation based on linear dependence will not work in this case. However the list can be modified in the following way to reduce its size:

$$\mathcal{A}' = \{(a \oplus b, p \oplus q \oplus r \oplus s), (a, t)\}.$$

This optimisation is some kind of local minimisation of the list (which effectively minimises the size of the factorised expression). In a more general way, for this kind of optimisation we take any two pairs from the list and check whether modifying them reduces the size of the list in terms of number of literals. Assume that the two pairs under consideration are (X_1, Y_1) and (X_2, Y_2); the modification of these pairs to $(X_1 \oplus X_2, Y_1)$ and $(X_2, Y_1 \oplus Y_2)$ is useful if doing so reduces the cumulative size of the two pairs in terms of number of literals.

5.5 Finding Identities and Their Application

This procedure takes a set of expressions and finds the relations between them. Although finding all the relations among a set of expressions is a difficult task, if the size of the set is small, we can enumerate all possible expression trees and check whether it results zero or one in any case; if so, we get an identity. For example if we have two expressions X and Y, then we only need to check if one the two expressions $X \oplus Y$ and XY is zero or one. In case of more expressions we use a conservative approach to find a subset of all identities. In that case, we enumerate all expression trees with depth smaller than some constant, set according to available computation resources.

This is one of the crucial steps as the identities output by this procedure are used in the next iteration for basis computation. The following example finds the identities among the basis obtained for the majority function of 7 inputs $a_1, a_2, \ldots, a_7$:

$$X = \bigoplus a_{i_1} a_{i_2} a_{i_3} a_{i_4},$$

where (i_1, i_2, i_3, i_4) takes all possible values of 4-tuples chosen from the integers $\{1, 2, \ldots, 7\}$.

The procedure *findBasis* with the input $\{X\}$ as the list of expressions, $\{a_1, a_2, a_3, a_4\}$ as the group variables, and $\emptyset$ as set of identities generates the following basis:

$$\begin{aligned} B = \{ & s_1 = a_1 \oplus a_2 \oplus a_3 \oplus a_4, \\ & s_2 = a_1a_2 \oplus a_1a_3 \oplus a_1a_4 \oplus a_2a_3 \oplus a_2a_4 \oplus a_3a_4, \\ & s_3 = a_1a_2a_3 \oplus a_1a_2a_4 \oplus a_1a_3a_4 \oplus a_2a_3a_4, \\ & s_4 = a_1a_2a_3a_4\}. \end{aligned}$$

When we give this basis to find the identities, we find the following identities:

$$s_3 \oplus s_1s_2 = 0,\ s_1s_4 = 0,\ s_2s_4 = 0, \text{ and } s_3s_4 = 0.$$

Note that by using the first identity we can reduce the basis to $\{s_1, s_2, s_4\}$, as s_3 can be written using s_1 and s_2. It is easy to see that the three output bits of a 4-bit counter with inputs a_1, a_2, a_3, and a_4 have the same expressions as s_1, s_2, and s_4. In other words, this helps us finding the hidden instance of a 4-bit parallel counter in the majority function. The next two identities can be used in the next iteration to find the new basis, as they define the null-spaces of s_1, s_2, and s_4.

Right now we are concentrating only these two kind of identities: the first, where one of the variables can be written as a function of the others (which helps in reducing the size of basis) and the second, where the product of two expressions is zero (as it helps in the computation of null-spaces of some variables, which are used in the basis computation in the next iteration).

After optimising the basis we rewrite the input expressions by replacing each occurences of basis elements by a new variable and proceed to the next iteration. An execution of our algorithm to find the hierarchy in the 7-bit majority function is shown in Fig. 6.

16-bit LZD/LOD		
Unoptimised (SOP)	$426.8\mu m^2$	$0.36ns$
Progressive Decomposition	$392.3\mu m^2$	$0.30ns$
32-bit LOD		
Unoptimised (SOP)	$1691.7\mu m^2$	$0.54ns$
Progressive Decomposition	$1062.7\mu m^2$	$0.43ns$
15-bit Majority function		
Unoptimised (SOP)	$2353.5\mu m^2$	$0.79ns$
Progressive Decomposition	$765.5\mu m^2$	$0.58ns$
16-bit Counter		
Unoptimised (using adder tree)	$1251.1\mu m^2$	$0.86ns$
Progressive Decomposition	$1427.3\mu m^2$	$0.74ns$
TGA	$1066.2\mu m^2$	$0.71ns$
16-bit Adder		
Unoptimised (Ripple Carry Adder)	$1866.2\mu m^2$	$0.56ns$
Progressive Decomposition	$1836.9\mu m^2$	$0.54ns$
DesignWare	$1375.5\mu m^2$	$0.58ns$
15-bit Comparator		
Unoptimised (progressive comparator)	$514.9\mu m^2$	$0.40ns$
Progressive Decomposition	$466.6\mu m^2$	$0.33ns$
Carry out of Subtracter	$577.2\mu m^2$	$0.40ns$
12-bit Three-Input Adder		
Unoptimised ($A + B + C$)	$2058.0\mu m^2$	$1.09ns$
RCA(RCA(A, B), C)	$2426.1\mu m^2$	$1.11ns$
Progressive Decomposition	$1772.8\mu m^2$	$0.75ns$
CSA + Adder	$1646.8\mu m^2$	$0.70ns$

Table 1: Optimisation results for all our benchmarks.

6. EXPERIMENTS

We have implemented Progressive Decomposition in Maple 10, which is used as a front-end to Synopsys Design Compiler. We synthesise the input circuit directly, or we feed it to Progressive Decomposition to build the hierarchy and then synthesise the structured circuit. All the circuits are synthesised using a common standard cell library for UMC $0.13\mu m$ CMOS technology. The results of all of our benchmarks are shown in Table 1.

As we can see there are qualitatively 6 different circuits. The first circuit corresponds to the leading zero detector, mentioned in Section 1. The *Leading One Detector* (LOD) is the similar circuit to LZD, the only difference is that it looks for the first zero bit from the left, in the given integer. Since the Reed-Muller expression corresponding to LOD circuit is significantly smaller compared to the expression corresponding to LZD ciruit, we can even optimize a 32-bit LOD using our tool, which is not possible for 32-bit LZD, due to its large size in Reed-Muller form. In the case of LZD and LOD, the unoptimised expression corresponds to the expression for each output bit written in sum-of-product form, which is the input description of LZD mentioned in Section 1. As the results show, Progressive Decomposition not only reduces the critical path delay, but also gains in terms of cell area. Qualitatively the output generated for 16-bit LZD by Progressive Decomposition is exactly identical to the one suggested in [8].

The next circuit is the 15-bit majority function. A straightforward implementation of majority function is to consider all 8-bit combinations of the 15 input bits. If, for any combination, all the 8 bits turns out to be one, then the majority will be one; otherwise it will be zero. This gives a very intuitive description of the majority function in sum-of-product form. When we give this dscription to Design Compiler, it produces a circuit with delay 0.79 ns and cell area $2353.5\mu m^2$. Our tool instead finds the hidden instances of parallel counters inside the circuit and implements the circuit by first counting the number of ones and then compare them with 8. The output generated by our tool is not only 25% faster compared to the original circuit, but also has a size almost one third of the original circuit.

The unoptimised expression for the 16-bit parallel counter corresponds to the input written as a sum of sixteen 5-bit integers whose four most significant bits are zero and the least significant bit is the corresponding input bit. Once again, we can see that our tool outperforms the circuit output by direct synthesis. However, the circuit generated by the *Three Greedy Approach* (TGA) [10] is slightly faster than our circuit. This is due to the fact that TGA not only builds the circuit using $3 : 2$ counter blocks, but also keeps the proper interconnection between the block to optimise the delay. In our tool the main emphasis is finding the structure and not finding the optimal scheduling.

In the case of a 16-bit adder, the input description is in the form of *Ripple Carry adder* (RCA). As we can see, the circuit generated by our algorithm has almost the same performance as the one generated by direct synthesis. This is because in the case of an adder, the basis constructed by using only algebraic factorisation is the same basis constructed by our algorithm. Hence, the techniques based on algebraic factorisation such as kernel extraction algorithm also perform well on this circuit. However, as soon as we increase the number of operands of the adder from two to three, Design Compiler is unable to find effective kernels, because this would require Boolean division. Hence, in this case, direct synthesis results in a circuit which is almost 50% slower and is almost 1.5 times as large as the circuit generated by Progressive Decomposition. Note that the performance of the circuit generated by our tool in this case is almost the same as the circuit designed manually using a *Carry-Save Adder* (CSA) followed by a final adder.

The 15-bit comparator function takes two 15-bit integers A and B, and returns true if $A > B$ and 0 otherwise. One possible way to implement the comparator function is to compare the most significant bits, if they are unequal then it is clear which of the two integers is greater; however, if they are equal, then one compares the second most significant bits and so on. When we give this description to Design Compiler and our tool, our tool recognises that the comparator function is the same as the sign of the subtraction of B from A, which can be computed in a carry-lookahead fashion. This results in a circuit 20% faster compared to direct synthesis.

7. CONCLUSIONS AND FUTURE WORK

In this work we show that logic synthesis tools are unable to recognise the regularity in general circuits, due to shortcomings of algebraic division. We present a novel approach to tackle this problem which builds the circuit progressively oot of smaller building blocks and hence imposes some hierarchy on the circuit without implementing Boolean division explicitly. This makes the resulting circuit more regular and having low fan-in dependencies. Our tool generates the circuits with quality equivalent to manually designed ones without any prior knowledge of the circuit.

We have already mentioned that in some cases the input description of a circuit in Reed-Muller is so large that it cannot be handled by our algorithm. However, the Reed-Muller form is important for our optimisations as, in this form, Boolean expressions form a ring and exhibit useful properties. Hence, a good inspiration for future work would be to search a representation for Boolean expressions which does not blow up the size of the original expression but also follows the properties of a ring.

8. REFERENCES

[1] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer, New York, 1993.

[2] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[3] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, Feb. 1990.

[4] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[5] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kauffman Publishers, San Francisco, 2004.

[6] I. N. Herstein. *Topics in Algebra*. John Wiley and Sons, 2001.

[7] B. D. Lee and V. G. Oklobdzija. Improved CLA scheme with optimized delay. *Journal of VLSI Signal Processing*, 3:265–74, Oct. 1991.

[8] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-2(1), Mar. 1994.

[9] A. R. Omondi. *Computer Arithmetic Systems*. Prentice Hall, New York, 1994.

[10] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, C-47(3):273–85, Mar. 1998.

[11] A. K. Verma and P. Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 791–98, San Jose, Calif., Nov. 2004.

[12] A. K. Verma and P. Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 445–50, San Francisco, Calif., July 2006.

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