Automatic Synthesis of Compressor Trees: Reevaluating Large Counters

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

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

ABSTRACT

Despite the progress of the last decades in electronic design automation, arithmetic circuits have always received way less attention than other classes of digital circuits. Logic synthesisers, which play a fundamental role in design today, play a minor role on most arithmetic circuits, performing some local optimisations but hardly improving the overall structure of arithmetic components. Architectural optimisations have been often studied manually, and only in the case of very common building blocks such as fast adders and multi-input adders, ad-hoc techniques have been developed. A notable case is multi-input addition, which is the core of many circuits such as multipliers, etc. The most common technique to implement multi-input addition is using compressor trees, which are often composed of carry-save adders (based on $(3:2)$ counters, i.e., full adders). A large body of literature exists to implement compressor trees using large counters. However, all the large counters were built by using full and half adders recursively. In this paper we give some definite answers to issues related to the use of large counters. We present a general technique to implement large counters whose performance is much better than the ones composed of full and half adders. Also we show that it is not always useful to use larger optimised counters and sometimes a combination of various size counters gives the best performance. Our results show 15% improvement in the critical path delay. In some cases even hardware area is reduced by using our counters.

1. INTRODUCTION

Compressor trees are one of the key components in arithmetic circuits, as these are the main constituents of parallel multiplier and multi-input adders. Hence, improving the speed of a compressor tree results in significant speedup of the circuit. Unfortunately logic synthesis tools do a lousy job in optimising XOR-intensive circuits due to the shortcoming of algebraic factoring. Hence, the direct synthesis of compressor trees (which are heavily XOR-dominated) results in poor quality circuits. In fact, finding the optimum implementation of a compressor tree still remains a challenging task. Several attempts has been made to generate the optimal compressor trees. The most common strategy to implement a compressor tree is by using carry save adders. A carry save adder takes three integers and returns two integers such that the sum of the inputs equals the sum of the outputs. The carry save adder uses full adders (i.e., $(3:2)$ counter) in parallel to reduce the three bits in i^{th} bit position into two bits at positions i and $i+1$. Many algorithms have been proposed to use $(3:2)$ counters in an effective way: some of them are layout constrained such as the compressor trees by Wallace [14] and Dadda [2], which exploit the regularity of the structure; however, some other methods such as Three Greedy Approach by Oklobdzija et al. [5, 7] are greedy algorithms to find the optimal interconnections among the various $(3:2)$ counters.

It is also possible to use large counters instead of $(3:2)$ counter, e.g., using a $(7:3)$ counter one can reduce 7 bits at bit position i into 3 bits at positions i, $i+1$ and $i+2$. It has been observed that using a $(7:3)$ counters is advantageous compared to using only $(3:2)$ counters. In fact, as we increase the counter size, the speed of the compressor tree increases. However, later it was noticed that all the large counters were implemented using $(3:2)$ counter, but having a proper interconnections among the $(3:2)$ blocks. Other components used for multi-input additions are $(p:q)$ compressors. In contrast to counters, compressors use a horizaontal path also for carry propagation. However, compressors also use full adder (FA) and half adder (HA) as their building blocks, and hence the special advantage of compressors can also be achieved by implementing proper interconnections among FA and HA blocks.

As we have already mentioned, the most effective algorithm to find the best interconnections among the $(3:2)$ counters was given by Oklobdzija et al [5] and is known as Three Greedy Approach (TGA). In TGA, each bit position is considered individually from right to left and the bits in a column are reduced to three or less using $(3:2)$ counters. While choosing the inputs of a $(3:2)$ counter, one sums those bits whose input arrival time is least among all the bits. Once each bit position has three or less bits, a final sequence of $(3:2)$ counters is used to reduce the three integers into two integers which are finally added by using an appropriate hybrid adder. A similar approach at word level was suggested by Kim et al. in their work [11] for the optimal allocation of the inputs to cascaded Carry-Save Adders.

In this paper we address the following questions:

- Is the implementation of a large counter using $(3:2)$ counters (full adders) and $(2:2)$ counters (half adders) the best implementation of it? If not, how to obtain the best implementation of a counter?
- If building optimal counters of arbitrary size is not feasible, is it sufficient to use only primal counters (a primal counter reduces 2^n-1 input bits into an n-bit word)? Does it always pay off to use the largest available counters instead of smaller counters (e.g., should we ever use a $(3:2)$ counter when the number of bits is ≥ 7, and a $(7:3)$ can be used)?

In the rest of the paper we answer these questions and show that it is not always possible to obtain the best implementation of a counter using smaller counters. We also propose a new method to get the optimal implementation of counters and compressor trees.

The rest of the paper is organised as follows. In the next section we discuss some earlier work done on this topic. In Section 3 we explain why the large counters and compressors built of full and half adders are not optimal. Section 4 discusses a novel approach to optimise counters and its limitations. In Section 5 we formulate the problem of optimising counters and compressor trees, and present a method to solve it using Integer Linear Programming. Finally in Section 6 we present the results of our experiments followed by conclusions in Section 7.

2. STATE OF THE ART

The problem of multi-input addition is not a new problem in the arithmetic community as it appears often in many arithmetic circuits such as multipliers, etc. Not only in multipliers, but also in some other applications which do not seem to contain multi-input additions, some appearances of multi-input additions can be found by clustering adders separated by logic operations as shown in [12] and [10]. The first breakthrough in this direction was by Wallace [14] when he introduced the notion of carry save adder (constructed by using $(3:2)$ counters). Using the chain of carry save adders, the inputs of multi-input addition can be reduced significantly faster than that by doing serial addition of inputs. The notion of Wallace tree was generalised by Dadda [1] and he proposed algorithms to minimise the number of counters.

Since the compressor trees proposed by Wallace and Dadda were very regular and layout constrained, the interconnections between the $(3:2)$ counters were fixed, irrespective of the arrival times of the inputs. It was later realized that by implementing proper interconnections between various $(3:2)$ counter blocks, large counters and compressors (such as $(7:3)$ counter, $(4:2)$ compressor, etc.) can be generated. Since the large counters consider input arrival times to interconnect the $(3:2)$ blocks up to some extent (at least among its constituents $(3:2)$ counters), the compressor trees built on large counters have usually smaller delays.

The notion of large counters and compressors was initiated by Weinberger [15] who introduced $(4:2)$ compressors. The use of larger compressors and counters were explored by Song and De Micheli [6]. Almost all large counters in literature are made of full adders; however, discussion of faster quasi-digital counters also exists in literature as in the work of Swartzlander [9]. Unfortunately these quasi-digital counters are extremely complex and prone to problems due to drift, etc. For the first time Oklobdzija presented algorithmic methods (TGA) in his fundamental work [5, 7] to find the optimal interconnections among the $(3:2)$ counter blocks, which consider the different input arrivals of inputs. Since all the large counters used previously were built using $(3:2)$ counters as a basic block, after the introduction of the Three Greedy Approach the use of larger counters appeared unnecessary. The present work advances with respect to TGA and shows that it is advantageous to use large optimised counters (not necessarily built using $(3:2)$ counter as a basic block).

Other than optimising the compressor tree, some work has been done to optimise the final adder to add the two output words of compressor trees. The choice of an optimal adder depends on the delay profiles of bits of two output words. An example of such optimisation is presented in the work of Fadavi-Ardekani [3] which optimises the final adder of the compressor tree used to reduce partial products bit array. The adder generated by [3] uses various stages of carry-select adders. However, the proposed method assumes that all inputs of compressor tree are available at the same time, which is not always true in general. The optimisation of final adder by considering an hybrid adder was also explored by Oklobdzija [8]. In this work, we use the latter approach to optimise the final adder.

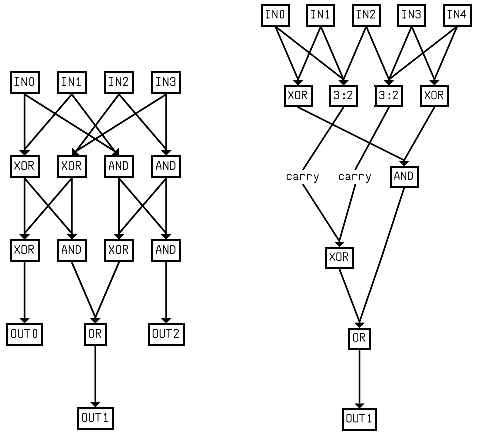


Figure 1: Best implementation of the 2^{nd} most significant bit of (a) four-bit counter, and, (b) five-bit counter (the one which typically belongs to the critical path). Both implementations cannot be obtained by any combination of full and half adders.

Note that all the works mentioned above assume that the inputs of a $(3:2)$ counter are independent of each other, which is not the case in general as we have shown previously [13]. This work is based on expliting the dependencies among the inputs of an XOR gate, which are the core elements of full adder, to improve the speed of multipliers. However, the method is computationally expensive and remains practical only for smaller circuits. We will discuss the application of this technique to counters in Section 4.

3. NON-OPTIMALITY OF COUNTERS BUILT ON FULL AND HALF ADDERS

As we have mentioned in the previous section, sometimes the inputs of a XOR gate are correlated expressions and using this fact the XOR gate can be simplified into simple gates such as NAND or OR. As an example the expression for the second most significant bit of a four-bit counter $(4:3)$ with input bits x_0, x_1, x_2, and x_3 can be written as follows:

$$\begin{aligned} out_1 &= x_0x_1 \oplus x_0x_2 \oplus x_0x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_2x_3 \\ &= ((x_0 \oplus x_1)(x_2 \oplus x_3)) \oplus (x_0x_1 \oplus x_2x_3). \end{aligned}$$

Note that the expressions $((x_0 \oplus x_1)(x_2 \oplus x_3))$ and $(x_0x_1 \oplus x_2x_3)$ cannot be true simultaneously for any values of x_0, x_1, x_2, and x_3 and hence the XOR of the two expressions will be same as the OR of the two expressions. Using this fact the expression of out_1 can be simplified:

$$out_1 = ((x_0 \oplus x_1)(x_2 \oplus x_3)) + (x_0x_1 \oplus x_2x_3).$$

This implementation of a four-bit counter is shown in Fig. 1(a). In a similar way, in the expression of the second bit of a five-bit counter, one XOR can be converted into one OR, resulting in the implementation shown in Fig. 1(b).

3.1 Explicit expression for a Counter

To understand the nature of the correlation among the various inputs of XOR gates used in a counter circuit, we should look at the exact expression of a counter circuit. A $(n:k)$ counter takes n bits $x_0, x_1, ..., x_{n-1}$ as inputs, and returns k bits $c_0, c_1, ..., c_{k-1}$ such that vector $(c_{k-1}, c_{k-2}, ..., c_0)$ corresponds to the binary representation of the sum of n input bits. The following theorem tells the exact expression for c_i's in terms of $x_0, x_1, ..., x_{n-1}$.

THEOREM 1. *If $(c_{k-1}, c_{k-2}, ..., c_0)$ is the binary representation of the sum of the bits $x_0, x_1, ..., x_{n-1}$, then the expression for c_i can be given as follows:*

$$c_i = \bigoplus x_{k_1} x_{k_2} \cdots x_{k_{2^i}},$$

where $(k_1, k_2, \cdots, k_{2^i})$ runs over all $\binom{n}{2^i}$ of 2^i integers from the set $\{0, 1, \cdots, n-1\}$.

In order to illustrate the simplification of some counter circuits using correlation we define two terms.

$$c(n, r) = \bigoplus x_{k_1} x_{k_2} \cdots x_{k_r},$$
$$d(n, r) = \sum x_{k_1} x_{k_2} \cdots x_{k_r}.$$

Note that $c_i = c(n, 2^i)$. It is easy to see that $d(n, r)$ corresponds to the same expression as $c(n, r)$ with each XOR gate replaced by OR gate. Hence, often the delay of a circuit corresponding to $d(n, r)$ can be significantly less than that of corresponding to $c(n, r)$. Also note that in some cases $c(n, r)$ can be expressed very easily in terms of $d(n, r)$. Some such cases are stated in the next theorem.

THEOREM 2. *The following holds true for all n and i:*

$$(2^i < n < 2^{i+1}) \Rightarrow (c(n, 2^i) = d(n, 2^i)).$$
$$(2^{i+1} \leq n < 3 \times 2^i) \Rightarrow (c(n, 2^i) = d(n, 2^i)\overline{d(n, 2^{i+1})}).$$
$$(n \, mod \, 2 = 1) \Rightarrow (c(n, n-1) = d(n, n-1)).$$
$$(n \, mod \, 2 = 0) \Rightarrow (c(n, n-1) = d(n, n-1)\overline{d(n, n)}).$$

Note that the first statement tells that for the calculation of the most significant bit of a counter we do not need any XOR gate. Also note that a full adder is better than implementing a $(3 : 2)$ counter using half adders because it can use the property (3) mentioned above (that $c(3, 2) = d(3, 2)$; in other words $ab \oplus bc \oplus ac = ab + bc + ac$). Similarly, the expression for the second most significant bit of a four-bit counter can be simplified using the property (2) in the theorem above, i.e., $c(4, 2) = d(4, 2)\overline{d(4, 4)}$, which has the same performance as the one shown in Fig. 1(a). Note that these are not the only relations between $c(n, r)$ and $d(n, r)$, and it is extremely difficult to figure out which relation between $c(n, r)$ and $d(n, r)$ we should use to get the best advantage. Sometimes we might not even want to rewrite $c(n, r)$ in terms of $d(n, r)$. This is because rewriting $c(n, r)$'s improves the delay of the circuit locally, but introduces OR gates which results in the loss of ring properties such as commutativity, associativity, and distributivity properties (i.e., OR and XOR operations do not follow assocativity rules with each other); hence it hinders the factorisation of the expression.

4. EXPLOITING CORRELATION TO BUILD OPTIMAL COUNTERS

The previous section has answered negatively to our first qustion: large counters cannot always be implemented optimally out of smaller counters. It has also shown that the reason for such impossibility is the difficulty of accounting for correlations. We have mentioned in the Section 2 that the algorithm presented in [13], known as Selective Expansion improves the performance of a circuit by exploiting the correlation between the operands of XOR operation. Hence, one possibility is to write the circuit for a counter in terms of full and half adders, and then use the Selective Expansion algorithm to improve the performance of the circuit by replacing full adders and half adders with correlated inputs by simpler operators.

Counter size	Delay with FA and HA (ns)	Selective Expansion (ns)
2	(0.07, **0.12**)	(0.07, **0.12**)
3	(0.11, **0.17**)	(0.11, **0.17**)
4	(0.09, **0.33**, 0.21)	(0.09, **0.22**, 0.21)
5	(0.18, **0.34**, 0.26)	(0.18, **0.30**, 0.26)
6	(0.30, **0.36**, 0.28)	(0.24, **0.33**, 0.28)
7	(0.37, **0.40**, 0.28)	(0.34, **0.38**, 0.28)
8	(0.12, **0.48**, **0.48**, 0.32)	(0.12, 0.35, **0.40**, 0.32)

Table 1: Comparison between the delays of different counters before and after using the Selective Expansion algorithm [13].

In the Selective Expansion algorithm two kinds of correlation are measured, called local correlation and global correlation. The local correlation is the correlation between the operands of an XOR operation, while global correlation is the correlation among the rest of the expression and the operands of an XOR operation. If some empirical correlation index is above a threshold value, then the XOR gate is replaced by its equivalent expression in terms of AND and OR gates as shown below:

$$A \oplus B = \overline{AB}(A + B), \text{ and}$$
$$(A \oplus B) + C = (AB \rightarrow C)(A + B + C),$$

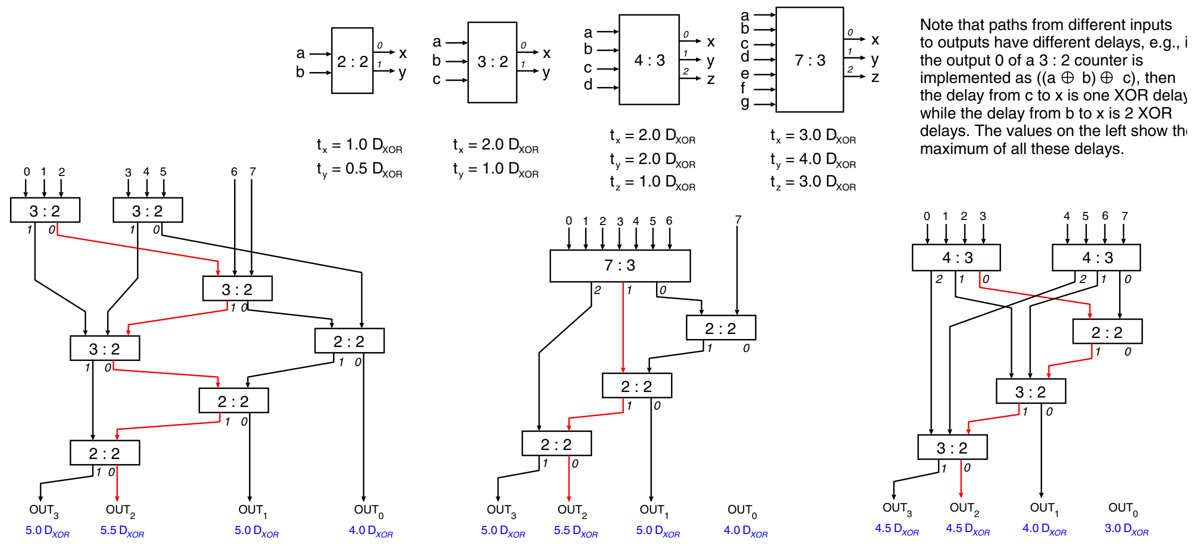
where the expression $(x \rightarrow y)$ is the same as $(\overline{x} + y)$. We used this algorithm to optimise counters, and the performance of some of the counters optimised by this algorithm is shown in Table 1. The first column shows the counter size, the second column shows the delay vector (delays of individual output bits from most significant to least significant) of the underlying counter when implemented using only full and half adders interconnected using the Three Greedy Approach. The third column shows the delay vector of the same circuit after optimising it using the Selective Expansion algorithm.

In this section we have exploited an existing technique to generate better large counters than previously possible with algorithmic approaches. The only problem with the Selective Expansion algorithm is its computational complexity: although it produces faster counter and compressor trees, it remains practical only for small size counters and compressors. We turn therefore our attention to the second set of questions mentioned in Section 1: how can we build a large counter optimally by using a combination of optimized counters up to a certain size?

5. BUILDING COMPRESSORS FROM LARGE SET OF COUNTERS

Since we cannot implement counters and compressors of all sizes using the Selective Expansion algorithm, one possibility is to find small and frequent blocks in a compressor tree, which can be optimised using Selective Expansion, and then replace those blocks by the optimised circuit corresponding to them. One way is to consider counters as building blocks and implement the compressor tree using these optimised counters, thanks to the fact that counters up to some significant sizes (e.g., 12-bit) can be optimised using Selective Expansion.

Now the question is how large should be the set of building block counters. As we have seen, large counters have more possibility to use the correlation between the operands of XORs, we should extend the set of building block counters as much as we can. At this point one might also think that we should consider only pri-



Note that paths from different inputs to outputs have different delays, e.g., i the output 0 of a 3 : 2 counter is implemented as ((a ⊕ b) ⊕ c), then the delay from c to x is one XOR delay while the delay from b to x is 2 XOR delays. The values on the left show th maximum of all these delays.

Figure 2: Implementation of an eight-bit counter using (a) only full and half adders, (b) only primal (i.e., $(2^n - 1)$ to n) and $(2 : 2)$ counters, and (c) counters smaller than eight bit. Note that the use of large counters is not always advantageous as it might leave the circuit lopsided.

.

mal counters, i.e., the counters which reduce $(2^n - 1)$ bits into n bits (e.g., $(3 : 2)$, $(7 : 3)$, etc.) and half adders. However, using only primal counters might lead to a nonoptimal solution, because it might make the circuit unbalanced, while using smaller counters which are not primal may make the circuit more balanced, and hence faster.

To understand it more clearly let us consider the example shown in Fig. 2. The first part of the figure shows the delays of various counters in terms of the delay of a two-input XOR gate (which is an approximation of the real delay values). Next, Fig. 2(a) shows the implementation of a $(8 : 4)$ counter using only full and half adders according to the Three Greedy Approach. Fig. 2(b) shows the best implementation of $(8 : 4)$ counter using only primal counters and half adders. Finally, Fig. 2(c) shows the best implementation of the same $(8 : 4)$ counter using any smaller counters. Note that in all the three circuits, the computation of the second most significant bit comes in the critical path. However, the critical path delay in the three circuits is different. In the first and second circuits the delay is $5.5D_{XOR}$, and in the last circuit it is $4.5D_{XOR}$. The reason is that when we implement the $(8 : 4)$ counter using $(4 : 3)$ counters, we exploit the correlation among the operands of XOR more than the correlation used in implementation using full and half adders, and also it is more balanced compared to the circuit built on $(7 : 3)$ counters. Although here we have used an approximate model to measure the counter delays, the same conclusion can be deduced if we had used the actual delays of optimized counters as shown in Table 1.

5.1 Approximate Delay Model and Problem Formulation

For the simplicity and without loss of generality we use only upto $(8 : 4)$ counters as building block counters, and to estimate the delay of a circuit built on these counters we consider the critical path delay of the circuit, where the values corresponding to the delay of the building block counters are taken from Table 1, i.e., the delays of counters optimised using Selective Expansion. Now we can formulate the problem as follows.

PROBLEM 1. *Given a set of input integers with not necessarily identical arrival times, find the best implementation of the compressor tree built on up to $(8 : 4)$ counters to add the input integers.*

5.2 ILP Formulation

Integer Linear Programming (ILP) has been proved to be a powerful method to solve combinatorial optimisation problems like the one mentioned above. Although theoretically solving an ILP is an NP-hard problem, however, many tools like CPLEX [4] solve sufficiently large instances of ILP within reasonable time. Any problem which can be formulated as an Integer Linear Program has two elements: constraints and objective functions. The constraints should be in the form of linear inequalities and equalities and the objective function must be a linear function of the input variables which has to be minimised or maximised. Restriction of variables to integers, Booleans, and piecewise continuous variables is also allowed. Next we show how we formulate our problem as an Integer Linear Programming problem. First we define a couple of terms which will help understanding the formulation.

DEFINITION 1. **Rank of a counter:** *If the inputs of a counter are the inputs at bit-position i, or the carries propagated from previous bit positions to this bit position, then the rank of this counter is i.*

DEFINITION 2. **Weight of a signal:** *All the input signals at bit-position i have the weight i. Also the j^{th} output (starting from 0) of a counter with rank i will have weight $(i + j)$.*

It is easy to see that all the input signals of a rank i counter will have weight i. Also the output at bit position i will have weight i. Note that, since in all counters except $(2 : 2)$ counters the number of inputs are at least one more than the number of outputs, the number of counters (excluding the $(2 : 2)$ counters) in a compressor tree with N input bits must be less than N. If we consider $(2 : 2)$ counters also, then the upper bound can be proved to be $O(N^2)$. However this is an extreme bound and we allow only cN counters in our compressor for some constant c.

For the sake of brevity we demonstrate our formulation using and, or, if-else, max, min, etc. Such operators can be easily written using ILP with additional variables and constraints. The list of variables used in the formulation and their interpretation is given below.

- $size_i$**:** This denotes the size of the i^{th} counter. Note that $size_i$ can vary from 2 to 9 and must be an integer (i.e., $2 \leq size_i \leq 9$). If the $size_i$ is 9 for a counter, that means the counter is a null counter (i.e., unused counter).
- e_{ijk}**:** It is a Boolean variable and is true if there is a connection between the k^{th} output signal of the i^{th} counter and the j^{th} counter. The value of k varies from 0 to 3, and if the i^{th} counter has less than $(k+1)$ outputs, then e_{ijk} is set false.
- p_{ij}, and q_{ijk}**:** Both are Boolean variables. p_{ij} is true if there is an edge from the i^{th} input bit to the j^{th} counter, while q_{ijk} is true if the k^{th} output signal of the i^{th} counter corresponds to the j^{th} output. Note that e_{ijk} and $q_{ij'k}$ can not be true simultaneously (i.e., $e_{ijk} + q_{ij'k} \leq 1$).
- t_{ij}**:** It is a real variable and denotes the delay of the j^{th} output of the i^{th} counter.
- r_i**:** This denotes the rank of the i^{th} counter.
- h_{ijk}**:** These are some special variables which manage the counters with inputs of different arrival times.

Next we present the list of the constraints. The constraints can be divided into three categories: I/O based constraints, constraints based on rank of counters, delay based constraints, and special constraints.

- **I/O Constraints:** The number of inputs and outputs of a counter should be consistent with its size, e.g., an $(8:4)$ counter must have 8 incoming connections and 4 outgoing connections. This constraint can be written as follows:

$$\text{if } (size_i = 8), \text{ then} \sum_{j<i}\sum_{k}(e_{jik}) + \sum_{k}(p_{ki}) = 8, \text{and}$$
$$\text{if } (size_i = 8), \text{ then} \sum_{j>i}\sum_{k}(e_{ijk}) + \sum_{j'}\sum_{k}(q_{ij'k}) = 4.$$

  Also note that some of the edge variables can be assigned zero directly as mentioned above. The following examples illustrate this kind of constraints:

$$\text{if } (size_i < 8), \text{ then } \forall j(e_{ij3} = 0).$$
$$\text{if } (size_i = 9), \text{ then } \forall (j,k)(e_{ijk} = 0).$$

- **Rank Based Constraints:** The rank of a counter must be well defined, i.e., all weight of all its input signals must be equal to the rank of this counter. As an example suppose that the m^{th} input bit was at the n^{th} bit position, then the rank of a counter which uses this input must be n and also all other input signals of this counter must have weight n. More formally:

$$\text{if } (p_{mi} = 1), \text{ then } r_i = n.$$
$$\text{if } (e_{ijk} = 1), \text{ then } r_j = r_i + k.$$

  Similar constraints for the outputs can also be applied.

- **Delay Based Constraints:** Delay based constraints put lower bounds on the delays of output signals of a counter. A typical delay constraints looks like:

$$\text{if } (size_j = 5 \text{ and } e_{ijk} = 1), \text{ then } t_{ik} + d_{5,0} \leq t_{j0},$$

  where $d_{5,0}$ is the delay to compute the 0^{th} bit of a $(5:3)$ counter. Note that there is no upper bound on t_{ij}. This is because our objective function is to minimise the delay, hence the values of t_{ij}'s will automatically be set to their lower bounds. Also the above constraint assumes uniform arrival times of the inputs of a counter, which is not true. For example if 0^{th} of a $(3:2)$ counter with inputs a, b, c is implemented like $((a \oplus b) \oplus c)$, then the delay from c to output is D_{XOR}, while the delay from b to output is $2D_{XOR}$. In order to delay be minimized c should be the one with largest arrival time. To handle these kind of cases we define the new Boolean variable h_{ijk} which is true only if, among all the inputs to j^{th} counter, the one which is coming from i^{th} has the largest arrival time. After introducing this variable the delay constraint for a $(3:2)$ counter will look as follows:

$$\text{if } (size_j = 3), \text{ then} \sum_{i<j}\sum_{k}(h_{ijk}) = 1,$$
$$\text{if } (size_j = 3 \text{ and } e_{ijk} = 1), \text{ then}$$
$$t_{ik} + d_{3,0} - d'_{3,0}h_{ijk} \leq t_{j0}.$$

  Once again, note that we have specified that the sum of all h_{ijk}'s is one; since the objective function is to minimise the delay, this will automatically set true that h_{ijk} whose corresponding input has the largest arrival time among all the inputs.

- **Special Constraints:** Other than the above constraints, we also need to enforce that each input must be used by exactly one counter and each output should correspond to exactly one output bit of a counter. In other words,

$$\forall(i) \sum_{j} p_{ij} = 1, \text{and}$$
$$\forall(j) \sum_{i}\sum_{k} q_{ijk} = 1.$$

Objective Function: One possibility can be to have exactly one output bit per bit-position, in that case the objective function will be to minimise the maximum of the delays of these output bits. However, this method enforces that the final adder used must be a ripple carry adder, because that is the only adder which can be made using only full adders and other counters but no other logic functions. Instead, we allow two temporary outputs per bit-position so that we can use an appropriate final adder. In this case our objective function corresponds to the following expression:

$$\text{minimise } \max_i\{\max(\text{tmpOut}_{i0}, \text{tmpOut}_{i1}) + d_i\}.$$

The d_i's are constants which denote the estimated delay from i^{th} temporary output bits to the the slowest output bit of final adder. A reasonable estimate of these constant values can be found by implementing the compressor tree using Three Greedy Approach. Using the above constraints and the mentioned objective function, the problem can be fed to any standard ILP solver which can find an optimal solution or an approximation after some reasonable time.

6. EXPERIMENTS

We have written a C++ program which takes the bit-width and arrival times of input integers and writes an ILP instance corresponding to the optimisation of the compressor tree used to add the integers. This instance of ILP is solved by the ILP solver CPLEX and then we write the VHDL code corresponding to the resulting circuit with an appropriate choice of final adder (the final adder is

12×12-**bit Multiplier**		
DesignWare	$8211.3\mu m^2$	$2.43ns$
Three Greedy Approach	$11033.3\mu m^2$	$1.65ns$
Optimised Counter Approach	$12464.1\mu m^2$	$1.41ns$
16×16-**bit Multiplier**		
DesignWare	$15498.4\mu m^2$	$3.12ns$
Three Greedy Approach	$23070.5\mu m^2$	$1.83ns$
Optimised Counter Approach	$24120.0\mu m^2$	$1.64ns$
24×24-**bit Multiplier**		
DesignWare	$35069.8\mu m^2$	$4.35ns$
Three Greedy Approach	$50103.4\mu m^2$	$2.22ns$
Optimised Counter Approach	$49557.3\mu m^2$	$1.95ns$
16-**bit Counter**		
Three Greedy Approach	$1143.9\mu m^2$	$0.77ns$
Optimised Counter Approach	$1092.1\mu m^2$	$0.62ns$
24-**bit Counter**		
Three Greedy Approach	$1622.6\mu m^2$	$0.91ns$
Optimised Counter Approach	$1540.9\mu m^2$	$0.78ns$
32-**bit Counter**		
Three Greedy Approach	$2609.3\mu m^2$	$1.11ns$
Optimised Counter Approach	$2886.0\mu m^2$	$0.94ns$
48-**bit Counter**		
Three Greedy Approach	$3829.2\mu m^2$	$1.25ns$
Optimised Counter Approach	$4107.6\mu m^2$	$1.10ns$

Table 2: Optimisation results for all our benchmarks.

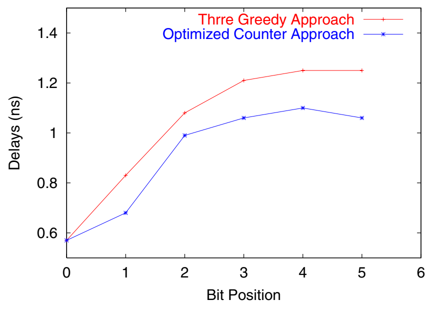


Figure 3: Comparison of arrival times of the 6 outputs of the 48-bit counter generated by the Three Greedy Approach and the Optimised Counter Approach.

chosen using the algorithm mentioned in [8]). The circuits are synthesised using a common standard-cell library for UMC $0.13\mu m$ CMOS technology.

Table 2 shows the results of our algorithm. There are two qualitatively different kind of arithmetic circuits: multipliers and counters. In case of multipliers we compare our results with the DesignWare implementation and also with the multiplier generated using the Three Greedy Approach. We have implemented 12×12, 16×16, and 24×24-bit multipliers. As we can see that the multipliers generated by our approach (Optimised Counter Approach) are the fastest ones. The multiplier generated by Optimised Counter Approach are 12–15% faster than the ones generated by the Three Greedy Approach, and the area penalty is almost negligible. In some cases, such as the 24×24-bit multiplier, the area of the Optimised Counter Approach multiplier is less than that of the Three Greedy Approach Multiplier.

The second set of benchmarks consist of counters. We have implemented $16, 24, 32$, and 48-bit counters. Here too we compare our results with the counters generated by the Three Greedy Approach using only full and half adders. Once again, the counters generated by our approach are almost 15% faster than the ones produced by the Three Greedy Approach at the cost of negligible or no area overhead. The comparison of the delay vectors of 48-bit counter generated by the two approaches is shown in Fig. 3.

7. CONCLUSIONS

In this paper we have shown that there are still chances to improve compressor trees, one of the most studied component in arithmetic circuits. We have shown that the compressor trees built on only full and half adders do not utilise the correlation among various operands and hence produce suboptimal results. Also, a compressor tree built on large size counters may be lopsided and hence slow compared to the compressor tree built on smaller counters. We have presented an approach based on Integer Linear Programming which exploits the correlation among various operands as well as tries to make the circuit as balanced as possible to improve the speed of the resulting circuit. The results show that our approach improves the speed of a compressor tree by almost 15% compared to state of art techniques.

8. REFERENCES

[1] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, XXXIV:349–56, 1965.

[2] L. Dadda and D. Ferrari. Digital multipliers: A unified approach. *Alta Frequenza*, XXXVII(11):1079–86, Nov. 1968.

[3] J. Fadavi-Ardekani. $M \times N$ Booth encoded multiplier generator using optimized Wallace trees. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-1(2):120–25, June 1993.

[4] ILOG. *CPLEX Optimization Engine*, 2006. Version 10.0.

[5] V. G. Oklobdzija, D. Villeger, and S. S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, C-45(3):294–306, Mar. 1996.

[6] P. Song and G. De Micheli. Circuit and architecture trade-offs for high speed multiplication. *IEEE Journal of Solid-State Circuits*, 26(9), Sept. 1991.

[7] 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.

[8] P. F. Stelling and V. G. Oklobdzija. Design strategies for optimal hybrid final adders in a parallel multiplier. *Journal of VLSI Signal Processing*, 14:321–31, Dec. 1996.

[9] E. E. Swartzlander, Jr. Parallel counters. *IEEE Transactions on Computers*, C-22(11):1021–24, Nov. 1973.

[10] Synopsys. *Creating High-Speed Data-Path Components—Application Note*, Aug. 2001. Version 2001.08.

[11] J. Um and T. Kim. An optimal allocation of carry-save-adders in arithmetic circuits. *IEEE Transactions on Computers*, C-50(3):215–33, Mar. 2001.

[12] 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.

[13] A. K. Verma and P. Ienne. Improving XOR-dominated arithmetic circuits by exploiting dependencies between operands. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, Jan. 2007.

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

[15] A. Weinberger. 4:2 carry-save adder module. *IBM Technical Disclosure Bulletin*, 23, Jan. 1981.