Signal Representation Guided Synthesis Using Carry-Save Adders For Synchronous Data-path Circuits *

Zhan Yu
Integrated Circuits and Systems Lab.
University of California
Los Angeles, CA 90095
zhanyu@icsl.ucla.edu

Meng-Lin Yu
Circuit and Systems Lab.
Agere Systems
Holmdel, NJ 07733
myu@agere.com

Alan N. Willson, Jr.
Integrated Circuits and Systems Lab.
University of California
Los Angeles, CA 90095
willson@icsl.ucla.edu

ABSTRACT

Arithmetic transformations using carry-save adders have been exploited recently in design automation but existing transformation approaches only optimize combinatorial functions. Most applications need synchronous circuits and it is known that techniques that move the positions of the registers, such as retiming, can significantly reduce the cycle time of a synchronous circuit. However, retiming disregards arithmetic transformations and its power is limited by the circuit topology. This work is the first to exploit carry-save arithmetic transformations together with the moving of the register positions. To enable such transformations, we first propose the use of a new *multiple-vector* signal representation. Next, we use *multiple-vector* signal representation as a common guide for all of our simultaneous carry-save arithmetic transformations with the moving of the register positions. Specifically, we propose, *operation forward* and *operation backward* carry-save transformations, which are transformations across register boundaries. We also propose *operation duplicate* and *operation merge* transformations to exploit the resource sharing and timing trade-offs in the implementation of a multiple-fanout network. Finally, we propose an efficient and effective heuristic that selectively applies a sequence of transformations to optimize the timing and the area of a synchronous circuit. Experimental results show that the proposed techniques significantly out-perform previous approaches.

1. INTRODUCTION

Arithmetic functions are key building blocks in VLSI data-path circuits. Among them, an efficient implementation of addition is especially important since it is the fundamental ingredient of other operations such as subtraction and multiplication. An *n*-bit carry-save adder (CSA) is shown in Fig. 1-(a), which is composed of *n* disjoint full adders (FAs). It takes three binary vectors (*X*, *Y* and *Z*) as inputs and its output is represented by two binary vectors (the *sum* vector *S* and the *carry* vector *C*, which we call a *carry-save* representation.) Instead of propagating the carry signals to the MSBs and generating the output in *vector-merge* representation as the vector-merge adder (VMA) in Fig. 1-(b), a CSA "saves" the carry signals in a *carry* vector *C*. Since carry-save addition avoids the time-costly carry-propagation operation, it has only one full-adder delay and is particularly suitable for high-speed implementation of arithmetic operations.

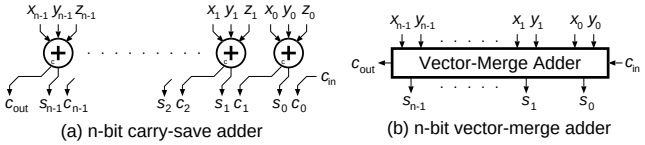

Figure 1: Carry-save adder and vector-merge adder.

Arithmetic transformations using CSAs have been exploited recently in [1]. Variations of the techniques proposed in [1] have also been reported [2, 3]. Carry-save transformations across non-addition operators were proposed in [2]. The timing and area trade-offs of carry-save implementation for multiple addition trees were exploited in [3]. However, all these transformation techniques [1, 2, 3] only optimize combinatorial circuits. They are obviously limited by the register boundaries and cannot be applied to optimize synchronous data-path circuits.

Retiming [4] is an important technique for optimizing synchronous circuits and it can be formulated as a mixed-integer linear programming (MILP) problem. Since a carry-propagation operation is more time-costly than carry-save operations, [5] separates them by introducing *carry-save* signal representation in the joint module selection and retiming optimization. The techniques proposed in [5] are capable of producing faster and smaller circuits in comparison with similar techniques that handle vector-merge signal representation alone, however it is limited by retiming, which assumes static circuit topology and disregards arithmetic transformations.

To further illustrate the limitations of performing retiming and arithmetic transformations separately, we examine the direct-form FIR filter example in Fig. 2-(a). Because of its circuit topology, a direct application of the retiming with carry-save representation technique in [5] on the Fig. 2-(a) filter produces an implementation of all the addition operators along the critical path using carry-save arithmetic and a VMA at the output as in Fig. 2-(b). Such an implementation has a critical path delay of 11 CSA delays and one VMA delay (we assume each multiplier has three partial-products). On the other hand, if a carry-save arithmetic transformation on combinatorial blocks is applied first, as in [1], the addition part of the multipliers and the chain of adders (as circled in Fig. 2-(a)) will be identified and reconstructed as a balanced CSA tree followed by a VMA, as shown in Fig. 2-(c). Next, however, even if we then attempt retiming, the critical path will still consist of six CSA delays and one VMA delay. The transposed-form FIR filter of Fig. 2-(d) (which has a much shorter critical path–three CSA delays and one VMA delay) cannot be generated from Fig. 2-(a) unless carry-save transformations and

*This research was supported by a grant from Lucent Technologies and MICRO 00-104.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA

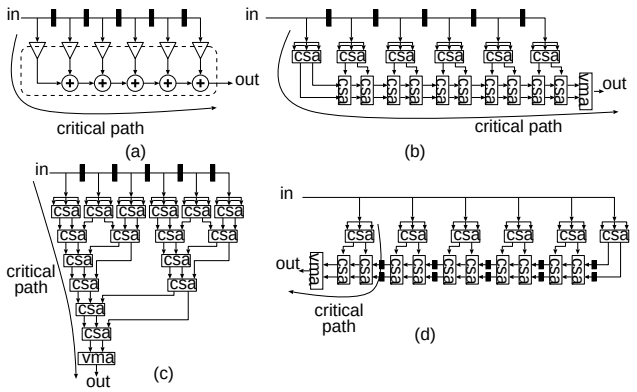

Figure 2: Motivation Example.

retiming are applied simultaneously.

In this work, we improve the existing transformation-based and retiming-based algorithms by simultaneous transformation and retiming. This is achieved by a set of carry-save transformations across register boundaries and algorithms to use these transformations to optimize a synchronous circuit. Note that a simultaneous transformation and retiming technique has been exploited in the logic synthesis domain [6]. However, there are sharp differences between logic functions and arithmetic functions (including their different delay models and different sets of applicable transformations). Therefore, logic synthesis techniques cannot be applied directly. This work is the first that exploits carry-save arithmetic transformations for synchronous data-path circuits.

Beyond the handling of transformations and the moving of register positions separately, all existing approaches have another hidden limitation: only trade-offs between carry-save arithmetic and carry-propagation operations are exploited. The joint module selection and retiming techniques in [5] can only insert registers between carry-save and carry-propagation operations by distinguishing *carry-save* and *vector-merge* signal representations; they are not capable of inserting registers inside a CSA tree. The transformation-based techniques in [3] only exploit the resource sharing of the VMAs among different addition trees but fail to exploit the resource sharing flexibilities inside a CSA tree.

We observe that the essence of using CSAs lies in a signal representation property: the number of binary vectors used to represent a signal. We will further discuss this point and propose the use of *multiple-vector* signal representation as a guide for our arithmetic transformations. We will show that using the signal representation property overcomes all the above limitations of existing techniques.

We will first introduce, in Section 2, the *multiple-vector* signal representation concept (and its advantages) as a guide for our transformations. Next, in Section 3, we will present *multiple-vector* signal representation guided carry-save arithmetic transformations across register boundaries, and for multiple-fanout networks. We will discuss the circuit timing and area trade-offs using these transformations. In Section 4, we will show how to use these transformations to optimize the timing and the area of a synchronous circuit. Our experimental results in Section 5 will show that the proposed method outperform existing combinatorial transformation techniques [1] and the retiming with carry-save representation technique [5]. Finally, we summarize our conclusions in Section 6.

2. MULTIPLE-VECTOR SIGNAL REPRESENTATION

To explain the concept of a *multiple-vector* signal representation, we first consider the task of adding together n binary vectors. For an addition operator that takes these n (assume $n \geq 3$) binary vectors as input, we can simply pass the n binary vectors directly to the operator's output and view the output signal as being represented by these n binary vectors. Alternatively, we can assign a CSA to any three of the vectors and convert them into $n-1$ vectors. Each time we allocate a CSA, we reduce the number of vectors that represent the output by one. This implies that we can actually use up to n vectors to represent a signal (which we call the *multiple-vector* signal representation), and view the addition operation as a signal representation conversion operation. This signal representation conversion view is not limited to addition operations; many arithmetic operations, such as multiplication and subtraction can be transformed into addition operations [1]. For simplicity, we only discuss addition operations in this paper.

We now define the *size* of a multiple-vector signal representation. If a signal is represented by n binary vectors, we say that the size of this representation is n. Therefore, a carry-save representation has size 2 and a vector-merge representation is of size 1. We will use signal representation and the size of signal representation interchangeably when there is no ambiguity. We refer to *increasing* and *decreasing* a signal representation as increasing or decreasing the size of a signal representation.

There is a circuit cost and a delay associated with signal representation conversion. Assume an addition operator has n input binary vectors and an output signal representation of size m ($n \geq m$). For $m > 1$, $n-m$ CSAs must be allocated to implement the addition operation. If $m = 1$, a VMA must be allocated, which usually has higher cost and larger circuit delay in comparison with a CSA. Given n input vectors of an addition operator, along with their signal arrival times, and the size of the output signal representation m, the timing-optimal construction of a CSA tree can be generated by a polynomial-time greedy algorithm similar to the construction of a Huffman tree [7].

A synchronous data-flow graph (DFG) $G(V,E)$ is commonly used to represent a synchronous circuit. It is a directed graph, where each vertex $X \in V$ represents an operation and each edge $e \in E$ represents a connection between two operations. A non-negative weight $w(e)$ is associated with each edge e, and it corresponds to the number of registers on e. The size of the signal representation is expressed as a positive integer $s(e)$ associated with each edge e. We call a DFG with the signal representation property a signal-representation flow-graph (SFG).

With the multiple-vector signal representation property associated with each edge in the SFG, we can now overcome the limitations of the existing approaches discussed in Section 1:

1. Using "multiple-vector" signal representation enables carry-save arithmetic transformations across register boundaries. The size of the signal representation $s(e)$ on a registered edge $\{e : w(e) > 0\}$ affects the computation distribution between the head and tail vertices of e. Therefore, we can perform transformations across the register boundaries by changing $s(e)$.

2. Using "multiple-vector" instead of carry-save and vector-merge signal representations allows us to insert registers inside a CSA tree. The MILP technique in [5] limits $s(e)$ to be 1 or 2, which implies a registered signal can only have carry-save or vector-merge signal representation. This may separate CSA trees from VMAs. With $s(e) > 2$, we now allow word-level pipelining inside a CSA tree.

3. Using multiple-vector signal representation gives us resource sharing flexibilities inside a CSA tree. Given an operator with multiple fanouts, allowing its output signal representation to be 1 or 2 exploits the resource sharing of a VMA on different fanouts. Allowing its output signal representation to be larger than 2 enables us

to share part of the CSA tree and to vary the extent of the sharing as needed.

With multiple-vector signal representation, the traditional signal data-arrival-time and required-time concepts should be modified accordingly. For edge e with $s(e) = n$, each of the n vectors has its own data-arrival-time. It is obviously not optimal to use one data-arrival-time (the largest one) for all the vectors. We define a *data-arrival-time profile* of the signal on e, which is an ordered list of the data-arrival-times of the n vectors. The *required-time profile* of a signal should also be an ordered list of the required-times that correspond to each vector. We will say that a signal is timing-critical or critical if its data-arrival-time profile is not upper bounded by its required-time profile.

3. SIGNAL REPRESENTATION GUIDED TRANSFORMATIONS FOR SYNCHRONOUS CIRCUITS

We will show, in this section, that multiple-vector signal representation allows us to perform carry-save transformations across register boundaries, exploit area and timing trade-offs in the implementation of multiple-fanout networks, and perform arithmetic transformations across non-addition operators. We also examine the circuit timing and area trade-offs associated with the multiple-vector signal representation property.

3.1 Transformations Across Register Boundaries

Carry-save transformations across register boundaries are achieved by changing the signal representation $s(e)$ on a registered edge e : $w(e) > 0$, since $s(e)$ determines the computation's distribution between the head and tail vertices of e. For example, in the DFG of Fig. 3-(a), edge $e : w(e) = 1$ connects the addition operators X and Y. The signal representations of all edges are given except for edge e. Figs. 3-(b) and (c) show two possible solutions of $s(e)$, where each arrow represents a binary vector and the numbers in parenthesis are the data-arrival-times of each binary vector. A CSA is assumed to have a sum and carry delay of 2. A circle represents an operator with its implementation illustrated in detail. In Fig. 3-(b), $s(e) = 3$, while $s(e) = 2$ in Fig. 3-(c). Increasing $s(e)$ moves part of the signal representation conversion operation in X *forward* into operator Y, while decreasing $s(e)$ moves part of the signal representation conversion operation in Y *backward* into X. We emphasize that the transformation from Fig. 3-(b) to Fig. 3-(c) cannot be obtained by retiming alone, since it involves the reconstruction of the CSA trees in X and Y. A transformation from Fig. 3-(b) to Fig. 3(c) is a carry-save arithmetic transformation across register boundaries, and it is achieved by changing $s(e)$. The circuit area and timing trade-offs associated with $s(e)$ are also reflected in Fig. 3. The solution in Fig. 3-(c), in comparison with Fig. 3-(b), requires higher hardware cost and longer circuit delay in X, but lower circuit cost and earlier data-arrival-time at the output of Y. Overall, reducing $s(e)$ results in smaller circuit area since the number of registers needed on edge e is reduced.

3.2 Transformations for Multiple-fanout Networks

A multiple-fanout network is a subgraph of an SFG $G(V,E)$, which includes an operator X with more than one fanout, the output edges of X and the fanout operators of X. For example, Fig. 4-(a) shows a multiple-fanout network, where addition operator X fans out to addition operators Y and Z. A carry-save transformation for this multiple-fanout network is achieved by changing the output signal

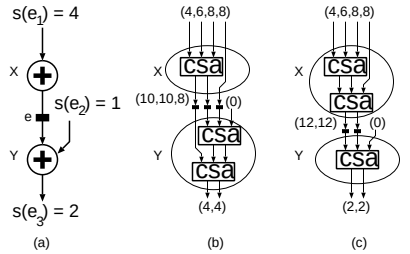

Figure 3: Signal Representation on Registered Edge.

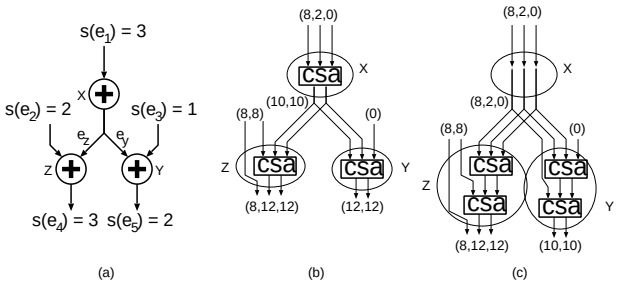

Figure 4: Signal Representation of Multiple-fanout.

representation of X, since this signal representation affects the computation's distribution among X and its fanout operators. In Fig. 4-(a), the signal representations on all edges are given except for the output edges of X: e_y and e_z. Figs. 4-(b) and (c) show two possible solutions for $s(e_y)$ and $s(e_z)$. In Fig. 4-(b), $s(e_z) = s(e_y) = 2$ while $s(e_z) = s(e_y) = 3$ in Fig. 4-(c). Increasing $s(e_z)$ and $s(e_y)$ moves part of the signal representation conversion operation out of X and *duplicates* this operation in fanout operators Y and Z. Conversely, decreasing $s(e_z)$ and $s(e_y)$ would *merge* part of the computation in Y and Z back into X. In general, increasing the output signal representation of an operator with multiple fanouts requires more circuit area, but may improve the timing of its fanout operators. This is illustrated in Fig. 4-(c), where Y has an earlier output data-arrival-time profile in comparison with Y in Fig. 4-(b).

The technique proposed in [3] exploits trade-offs similar to those described here, however, they are limited to the sharing of the VMAs by allowing $s(e_z), s(e_y) = 1,2$. In this work, we extend the resource sharing to the inside of a CSA tree by allowing $s(e_z), s(e_y) > 2$ and by allowing transformations across register boundaries.

3.3 Transformations Across Non-addition Operators

Carry-save transformations across a non-addition operator X are achieved by changing the signal representation of the input edge e of X. Since allowing $s(e) > 1$ usually implies a significant hardware increase in X, we only allow $s(e) = 1,2$. That is, we allow the flexibility of performing carry-propagation at the input of X or not doing so. In general, increasing the input signal representation of non-addition operators may trade circuit area for timing improvement. Such transformations have been discussed in [2]. Our work puts this kind of transformation into the framework of signal representation guided transformations and allows transformations across register boundaries.

4. OPTIMIZATION ALGORITHMS

We now present algorithms that use signal representation guided transformations to optimize a synchronous circuit. We first define, in Section 4.1, some terms used in later discussions. Next, in Section 4.2, we discuss the algorithms that perform signal representation

guided transformations in a synchronous data-path circuit. Finally, in Section 4.3, we propose an efficient and effective local search algorithm that applies these transformations to optimize the clock period T and the area of a synchronous circuit.

4.1 Background

We define the following notation. We denote the type of vertex X as $type(X)$. Functions $i(X)$ and $o(X)$ give, respectively, the number of input and output vectors of vertex X respectively. A vertex X is a tail of a signal path if all its output edges have non-zero weight. A vertex X is a head of a signal path if at least one of its input edges has non-zero weight. To retime a vertex X by $+r$ is to reduce the weight of each output edge of X by r and increase the weight of each input edge of X by r. To retime a vertex X by $-r$ is simply the opposite. When we disconnect an edge e from a vertex and reconnect it to another vertex, its weight $w(e)$ and signal representation $s(e)$ remain unchanged unless mentioned otherwise. Assuming each vertex has only one output, an edge e is a multiple (single) fan-out edge if there does (not) exist another e' with $tail(e') = tail(e)$. An SFG $G_0(V_0, E_0)$ is T-feasible with clock period T if the signal data-arrival-time profiles of all vertices $X \in V_0$ are upper bounded by T.

Given a design represented by a DFG $G(V,E)$, we first initialize it into a *canonical* SFG $G_0(V_0,E_0)$ with signal representation properties, in which there exists no zero-weight single-fanout edge connecting two addition operators, and where a timing-optimal CSA tree [7] is used to implement each addition operator. The initialization algorithm identifies addition trees in the combinatorial blocks of a synchronous circuit according to the rules in [1], then merges all addition operators of the same addition tree into one addition operator vertex. The signal representation properties on each edge are then assigned accordingly. Since the algorithm in [1] does not recognize registers, we assign all registered edges $e : w(e) > 0$ a signal representation $s(e) = 1$.

With carry-save arithmetic, if a transformation affects the input data-arrival-times of an addition operator X, we must reconstruct the CSA tree that implements X. This is because we want to implement each addition operator vertex using a timing-optimal CSA tree [7] which is constructed according to the data-arrival-times of its inputs. After each transformation, we need to update the implementation and timing information of all affected vertices.

During our transformations, we always seek to maintain the SFG in its canonical form, so that the largest combinatorial addition tree is identified and implemented with a single CSA tree to obtain the best timing performance.

4.2 Signal Representation Guided Transformation Algorithms

The problem of minimizing the clock period T using transformations is approached by generating a sequence of SFGs that are T-feasible for decreasing values of T. Transformations are applied to the timing critical vertices in an SFG to eliminate critical paths. The problem of minimizing the circuit area under a given clock period T assumes that we already have a T-feasible SFG $G_0(V_0,E_0)$. Local transformations are applied to reduce the circuit area. In this section, we discuss how to identify a candidate vertex for a signal representation guided transformation and how to apply a transformation for timing or area optimization.

4.2.1 Operation Forward

The operation forward transformation could be used to optimize the timing or the area of a synchronous circuit. Due to space limitations, we only elaborate on the use of operation forward for timing optimization.

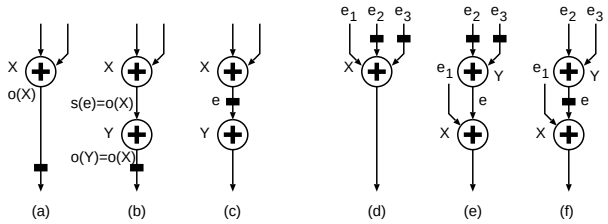

Figure 5: Operations Forward and Backward.

A candidate vertex X for the operation forward transformation in timing optimization is a tail vertex of a critical path in the SFG. If X is an addition operator, operation forward seeks to increase the output signal representation of X until X is no longer timing-critical. If X is a non-addition operator, operation forward simply retimes X by $+1$. The following algorithm describes the operation forward transformation for timing optimization with a target clock period T.

ALGORITHM 1 (OP-FWD(X,T))

1. Given a target clock period T, and vertex X which is a tail of a critical path and has an output signal representation $o(X)$.
2. **if** $type(X) \neq addition$
   retime X by $+1$, goto 4.
   endif
3. **if** $type(X) = addition$
   Substitute X with addition vertices X, Y and edge $e : X \xrightarrow{e} Y$, with $w(e) = 0$, $s(e) = o(X)$.
   Retime Y by $+1$.
   Transform the SFG to its canonical form.
   while (($s(e) \leq i(X)$) or (X is timing-critical))
   $s(e) = s(e) + 1$.
   Update implementation and timing of affected operators.
   endwhile
   endif
4. If no new critical path is created, accept transformation.

An example of the operation forward transformation for timing optimization is given in Figs. 5-(a), (b) and (c). Fig. 5-(a) shows a candidate addition operator X with output signal representation $o(X)$. We first replace X with $X \xrightarrow{e} Y$, as in Fig. 5-(b), where Y is an addition operator which passes all of its input vectors directly to its output. Next, we retime Y by $+1$ to obtain Fig. 5-(c), and transform the SFG into its canonical form if possible. Finally, we reduce $s(e)$ until X is no longer timing critical.

4.2.2 Operation Backward

The operation backward transformation can similarly be used for both timing and area optimization. A candidate vertex X for operation backward is a head node of a critical path. If X is not an addition operator, we simply examine whether we can retime X by -1. An example of the operation backward transformation for timing optimization with candidate addition vertex X is shown in Fig. 5-(d). We first allocate a new addition operator Y. We disconnect all registered input edges of X (e_2 and e_3), and reconnect them to the input of Y. Next, we connect the output of Y to the input of X via zero-weight edge e. Let Y pass all of its input vectors directly to its output, i.e., $s(e) = o(Y) = i(Y) = s(e_2) + s(e_3)$. The result of this step is shown in Fig. 5-(e). We then retime Y by -1 to obtain Fig. 5-(f), and transform the SFG into its canonical form if possible. Finally, we reduce $s(e)$ and seek to find the minimum $s(e)$ so that X is not timing-critical, and Y is not a new critical vertex.

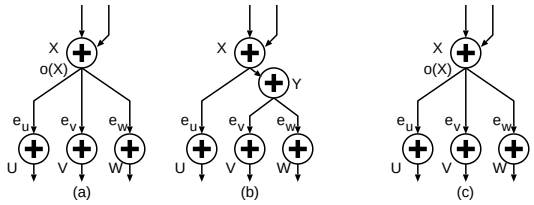

Figure 6: Operation Duplicate and Merge.

4.2.3 Operation Duplicate

The operation duplicate transformation is used for timing optimization and its candidate operator X is a timing-critical addition operator with multiple fanouts. An example of using operation duplicate for timing optimization is shown in Fig. 6-(a) where the candidate operator X has three addition operator fanouts U, V and W. For fanout operator U of X we increase $o(X)$ and find the minimum $o(X)$ that makes U non-critical. We record this value as $s(e_u)$. We do the same to V and W. In this example, let us assume we obtained $s(e_u) = 4$ and $s(e_v) = s(e_w) = 3$ for Fig. 6-(a). Next, we set the new value of $o(X) = \max\{s(e_u), s(e_v), s(e_w)\} = 4$, and we allocate a new addition operator Y with $o(Y) = s(e_v) = s(e_w) = 3$ and connect it with X, V and W as in Fig. 6-(b). We then update the implementation of affected vertices to conclude our operation duplicate transformation. If we combine operation duplicate with operation forward, we will have a transformation across register boundaries. We will not elaborate due to space limitations.

4.2.4 Operation Merge

The operation merge transformation targets a non-critical addition operator X with multiple fanouts. It is mainly used for area optimization. For the multiple-fanout network in Fig. 6-(c), we assume X and its fanout operators U, V and W are not timing-critical. Opposite to the operation duplicate transformation, the operation merge transformation first decreases $o(X)$ to find the minimum signal representation for each fanout edge e_u, e_v and e_w of X so that U, V and W are not timing-critical. Next, we reconstruct the multiple-fanout network according to $s(e_u)$, $s(e_v)$ and $s(e_w)$. If we allow the change of signal representation on registered edges ($w(e_u)$, $w(e_v)$ or $w(e_w)$ are non-zero), we have an operation merge transformation across register boundaries.

4.3 Efficient Search Algorithm

A transformation-based optimization method applies a series of transformations to optimize the timing or the area of an SFG. If we are restricted to one kind of transformation at a time, we may soon find that the timing or area of an SFG may not be further improved by this transformation alone, while it actually can be improved by other transformations or combinations of two (or more) transformations. However, if we consider multiple transformations or combined transformations at a single time, we expand the local search space but this requires more CPU time and more memory to store temporary SFGs. To solve this problem efficiently, we propose an efficient yet effective local search algorithm using the following strategies:

1. *Use combined transformations.* Combined transformations are used in both timing and area optimizations.

   (a) In timing optimization, we consider all transformations that could improve the timing of a timing-critical vertex. As we apply a transformation to optimize the cycle time of a synchronous circuit, we usually need to check whether the transformation creates new critical vertices and we can only accept transformations that reduce the number of critical paths. However, we would not like to limit our local search space by this criterion. Instead, if a transformation creates new critical paths that can be eliminated using another transformation, we apply the combination of the two transformations to increase the scope of our local search.

   (b) In circuit area optimization, we consider all transformations that could reduce the circuit area under clock period T. If a transformation creates new critical paths, we seek to eliminate these critical paths by using transformations for timing optimization.

   (c) The combination of operation duplicate and operation merge can be used to find the optimal signal representation for each fanout edge of a multiple fanout operator. When operation duplicate is applied on a critical addition vertex with multiple fanouts and a non-critical addition vertex with multiple fanouts is created, we can apply operation merge to reduce the circuit area. It can be shown that operation merge does not create new critical paths in an SFG.

2. *Rank the priority of different transformations.* During timing optimization, operations forward, backward, duplication, and transformations across non-addition operator boundaries, can all be applied to a timing-critical vertex in an SFG. However, we give these transformations different priorities. In general, operation forward and operation backward are likely to be able to eliminate critical paths with lower cost than operation duplicate or allowing carry-save input representation to multipliers. (Intuitively, operation duplicate increases circuit area in all fanout vertices of the candidate vertex, and allowing carry-save input representation vs. a vector-merge representation for a multiplier will cost us approximately another multiplier circuit.) Therefore, we give a higher priority to the use of operation forward and operation backward to prevent the circuit area from increasing too fast. Such priority is achieved by searching for acceptable operation forward and operation backward transformations first. If such transformations are found, we choose one with minimum area cost. Otherwise, we search for acceptable operation duplicate transformations and increasing input signal representations of non-addition operators.

Using these strategies, we overcome the limitations of local search by expanding the local search space selectively using our combined transformations and by reducing the local search space selectively with prioritized transformations.

Table 1: Hardware Module Library

Module	Cost	Delay (ns)
16-bit carry-propagate adder	252	10
16-bit carry-save adder	128	2
16-bit register	128	0

5. EXPERIMENTAL RESULTS

We have applied our method to several digital signal processing applications. Sample hardware delay and cost models are summarized in Table 1; they have been extracted from the Synopsys LSI_10k logic library.

We first initialized the combinatorial blocks between registers using the algorithm of [1] and then applied our algorithm to reduce the clock period of the design. The synthesis job took several seconds of CPU time on a Sparc Ultra-10 workstation and the synthesis results for timing optimization are summarized in Table 2. The proposed transformations across register boundaries significantly reduced the cycle time of the circuits generated using the algorithm in [1]. We also compare, in Table 2, with the result of joint retiming

Table 2: Timing Optimization

Design	Initial [1]		MILP model [5]		Multiple-vector		Compare with [1]		Compare with [5]	
	Timing	Area	Timing	Area	Timing	Area	Timing	Area	Timing	Area
BIQUAD2	70	21108	56	25460	46	25712	-34.3%	+21.8%	-17.9%	+1.0%
FIR16	22	36672	14	39420	12	36860	-45.5%	+0.5%	-14.3%	-6.5%
FIR8D	32	17276	50	17276	22	17404	-31.3%	+0.7%	-56.0%	+0.7%
EWF5	94	24004	78	35404	66	38472	-29.8%	+60.3%	-15.4%	+8.7%
EWF	66	14804	56	18772	50	21592	-24.2%	+48.9%	-10.7%	+15.0%
JAU4	64	12124	46	17512	40	18652	-37.5%	+53.8%	-13.0%	+6.5%
LAT4	116	11360	90	18920	78	20068	-32.7%	+76.6%	-13.3%	+6.1%
LWDF	34	6120	28	9712	28	8812	-29.0%	+35.5%	0%	-9.3%
LWDF2	62	12244	48	15952	44	16596	-17.6%	+44.0%	-8.3%	+4.0%
SS	28	18804	24	27892	20	27892	-28.6%	+48.3%	-16.7%	0%

Table 3: Area Comparison

Design	Clock Period T	MILP model [5]	Multiple-vector	Area Comparison
BIQUAD2	66	23412	21360	-8.8%
	56	25460	21360	-16.1%
EWF5	92	25664	23744	-7.5%
	82	33096	28984	-12.4%
	78	35404	30264	-14.5%
EWF	64	16592	15052	-9.3%
	60	18512	16976	-8.3%
	56	18772	19028	+1.4%
JAU4	54	13536	12372	-8.6%
	46	17512	16344	-6.7%
LAT4	100	16876	15200	-9.9%
	92	18796	17120	-8.9%
LWDF	30	8172	8300	+1.6%
	28	9712	8812	-9.3%
LWDF2	60	12360	12364	0%
	52	15828	14416	-8.9%
SS	26	23412	19188	-18.0%
	24	27892	23412	-16.1%

with carry-save representation using mixed-integer linear programming (MILP) [5]. These results show that our signal representation guided transformation method is capable of generating faster circuits than those produced by the MILP method with a small area penalty.

The following comments are directed toward the relative usefulness of the proposed transformations in timing optimization. The proposed operation forward and backward transformations are most powerful in minimizing the clock period for designs that have few operators with multiple fanouts, such as BIQUAD2, FIR16, FIR8D. For designs that contain many operators with multiple fanouts, such as EWF5, EWF, JAU4, LAT4, LWDF, LWDF2, SS, the use of operations forward and backward alone cannot successfully improve the circuit timing. The power of operation duplicate and its combination with operations forward and backward is reflected in these design examples and results.

The advantage of using signal representation guided carry-save transformations across register boundaries is clearest for clock period optimization, as shown in Table 2. However, our algorithm is also capable of producing circuit implementations having relatively small area. We compare, in Table 3, the circuit area optimization results with those of the MILP model [5] under various clock period T constraints. Since transformations relieve the timing bottlenecks, the proposed algorithm using signal representation guided transformations achieves lower or similar circuit area cost when compared with [5]. However, since our method is based on local transformations, rather than global optimization using mixed-integer linear programming [5], there are cases where we obtain a circuit with larger area than the result obtained using MILP. But such area increases, when they occur, are very limited, as indicated in Table 3.

6. CONCLUSIONS

In this work we have exploited carry-save arithmetic transformations together with the moving of the register positions. To enable such transformations, we have proposed a new *multiple-vector* signal representation and have shown the properties of a signal represented by *multiple-vectors*. We have used *multiple-vector* signal representation as a common guide for all our simultaneous carry-save arithmetic transformations with the moving of the register positions. Specifically, we have proposed *operation forward* and *operation backward* carry-save transformations, which are transformations across the register boundaries. We have also proposed *operation duplicate* and *operation merge* transformations to exploit the timing and area trade-offs for operators with multiple fanouts. These transformations are also performed across register boundaries. To use signal representation guided transformations to optimize a synchronous circuit, we have proposed an efficient local search algorithm for timing and area optimization. Experimental results have shown that the proposed techniques out-perform previous approaches.

7. REFERENCES

[1] T. Kim, W. Jao, and S. Tjiang, "Arithmetic optimization using carry-save-adders," in *Proc. Design Automation Conf.*, Jun. 1998, pp. 433–438.

[2] T. Kim and J. Um, "A timing-driven synthesis of arithmetic circuits using carry-save-adders," in *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2000, pp. 313–316.

[3] Y. Kim and T. Kim, "An accurate exploration of timing and area trade-offs in arithmetic optimization using carry-save adder cells," in *Proc. Midwest Synposium on Circuit and Systems*, Aug. 2000.

[4] C.E. Leiserson, F.M. Rose, and J.B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Third Caltech Conf. on Very Large Scale Integration*, Mar. 1983, pp. 87–116.

[5] Z. Yu, K.-Y. Khoo, and A. N. Willson Jr., "The use of carry-save representation in joint module selection and retiming," in *Proc. Design Automation Conf.*, Jun. 2000, pp. 768–773.

[6] G. De Micheli, "Synchronous logic synthesis: Algorithms for cycle-time minimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, pp. 63–73, Jan. 1991.

[7] J. Um, T. Kim, and C. L. Liu, "Optimal allocation of carry-save adders in arithmetic optimization," in *Proc. International Conf. on Computer-Aided Design*, Nov. 1999, pp. 410–413.