Timing optimization by bit-level arithmetic transformations [*]

Luc Rijnders Zohair Sahraoui Paul Six Hugo De Man[†]

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

This paper describes a method to optimize the performance of data paths. It is based on bit-level arithmetic transformations, and is especially suited to optimize large adder structures inside these data paths. The multi-operand adders are identified at the bit level and the addition parts are merged even across operator boundaries. Area and delay optimizations use CSD coding and timing-driven transformations, including bit-slice adder trees and logarithmic addition. The method forms a link between data path optimizations at the word level and logic synthesis techniques at the bit level. Experiments show that starting from a very simple description of an $N \times N$ multiplier an $\mathcal{O}(\log N)$ delay is obtained with very low run times.

1 Introduction

In digital signal processing (DSP) systems, the throughput is a major design objective. One therefore designs data paths with functionality that is tuned towards a specific application. To improve the throughput of these data paths, pipelining is commonly used [4]. However if delay is optimized prior to pipelining, the need for pipeline registers and the latency of the data paths can be reduced. Special techniques that improve delay at a reasonable cost in area are thus very useful. General methods based on logic synthesis can be used [6], but for data paths specific methods are more efficient.

In [7], a technique is presented that combines delay optimization and pipelining. The delay optimization is done by optimal selection between a number of predefined implementations for each operator with different performance characteristics. This technique is specially suited when a regular layout style is used. We however use a standard cell design method, which allows for more possibilities to optimize the net list. As the presented optimization techniques take care of area and delay efficiencies, we need only one implementation for each operator. Because Place&Route tools can handle critical nets, we assume that the detailed routing has no negative effect on the results.

[*]This research was supported by project ESPRIT 2260 of the E.C.

[†]Professor at the Katholieke Universiteit Leuven.

The analysis of a number of DSP applications shows that the largest delays in the data paths are caused by ripple paths going through several bit slices. The direction of information flow through these ripple paths can be from the least to the most significant bit (e.g. adder) or vice-versa (e.g. integer-length operator).

In cases where many ripple paths present approximately the same delay, they can not be optimized efficiently with logic synthesis techniques. This is due to their optimization strategy [6]. When there is only one longest delay path, it is clear where optimizations have to be performed. The best place to optimize the delay is where this can be done at minimum hardware cost. The transformations are very local and are based on locally collapsing the circuit and resynthesizing it to obtain a better delay. This is repeated as long as a suitable optimization place can be found or until the optimization objective is reached. However, when many critical paths must be treated at the same time, a more global strategy is necessary.

For specific structures containing many critical paths, it is possible to apply specific methods to optimize them. In this paper we specificly consider adder structures. In the context of multipliers the addition of the partial products has been studied extensively [1, 2, 3, 9, 10], but the influence of connected operators has not been considered. With a simple method of describing the operators and the data paths, it is possible to recognize larger adder structures and in other contexts also. This extra knowledge of the multi-operand adders in the circuit allows arithmetic transformations to optimize both area and delay. Associativity and commutativity are applied to timing-driven optimize the operations inside bit slices. Across the bit slices the ripple paths are transformed into carry-look-ahead structures, yielding high performance at reasonable cost.

This paper is divided in 7 sections as follows. In section 2, the characteristics of the adder structures are described. These characteristics are used in section 3 to identify the adder structures inside a data path. The specific optimization techniques for area are given in section 4 while section 5 presents the delay optimization. Finally in section 6, experiments and results are presented.

2 Adder structures

In many DSP applications critical paths are created inside adder structures of the data paths. That is why we demonstrate the method in this paper on the multi-operand addition. Similar techniques can be applied on any operation satisfying the properties of associativity and commutativity. Adder structures are present inside operators such as adder/subtracter, multiplier and comparator. Even larger adder structures can be formed across operation boundaries as shown in the following examples.

Example 1

A first example is the multiply/accumulate unit of Figure 1 which performs : $A \times B + C$. The multiplier consists of two parts : the generation of partial products and correction terms; and an array of adders to add up the partial products shifted over precalculated numbers of bits.

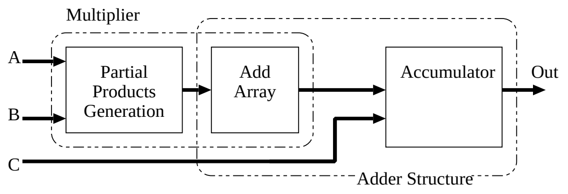

Figure 1: Adder structure of multiply/accumulate unit

The accumulator and the add-array of the multiplier form one large adder structure. The inputs to the adder structure are the partial products, the correction terms and the accumulator input C. By extending all these inputs to the word length of the accumulator output, the adder structure is simply a multi-operand adder of that word length.

Example 2

Another example is a FIR filter (see Figure 2). Also here the add-arrays of the multipliers and the multi-operand adder to generate the filter output are merged in one large adder structure.

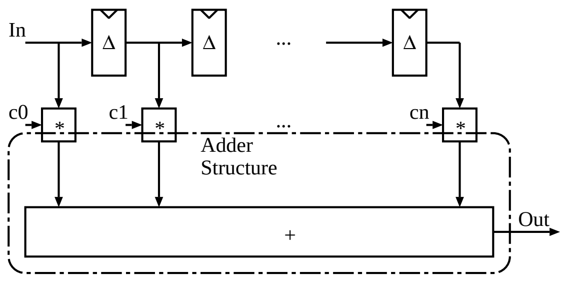

Figure 2: Adder structure of FIR filter

In case the coefficients of the filter are constant, each partial product of the multipliers is either zero or a shifted copy of the data input bits. Because of our optimization method, it does not matter if these constant coefficients are 2's-complement or Booth-encoded [1].

Characteristics

As can be seen from the two previous examples we search for structures that contain only additions. These structures should be as large as possible so that they can be optimized simultaneously. The adder structures contain several input words and one output word. More output words give rise to separated adder structures. Intermediate results of additions must thus be used exactly once. This rule holds both at the word level and at the bit level. At the bit level, this e.g. means that sign extensions have to be external to the adder structure. This can be easily done at the data path level, and although this may appear as a bad implementation, the area optimizations of the method will remove any overhead.

The functionality of the multi-operand adder structure is independent of the details of its implementation. The different addition schemes (like Dadda [3], Wallace [2], ...) are all covered by the same functionality. Therefore a simple addition scheme can be used to describe the addition parts of the operators. Different versions to obtain different performances need not to be made (and maintained), because the method will optimize area and delay.

3 Locating Adder structures

As can be seen from the examples in the previous section, adder structures do not coincide with operators. A special technique is thus required to recognize the multi-operand adder structures inside a data path.

This identification can be done at the operator level or at the bit level. The operator-level method requires for each operator the description of its adder part. It is then possible to take neighboring occurences of these adder parts together to identify the largest possible adder structures.

Another method is to identify the adder structures at the bit level. It is then required that operator descriptions use basic adder cells. Inside a data path, the occurences of the basic cells are then grouped together into adder structures.

We use the bit-level method, which will give us all information needed to perform the area and delay optimizations. The method is based on the recognition of bit slices and adder trees as we will describe them further on.

Bit slices

The problem to define *ripple path* or *bit slice* is that these terms are concepts the designer uses. When the circuit is mapped to regular layout modules these terms have a simple layout equivalent. When a standard cell implementation is used, a net list can not be simply divided into *bit slices*. Here we use the term *bit slice* in relation to the adder structures and it does not necessarily correspond to a *bit slice* of the designer.

A simple method is used to define *bit slice* by translating the designers concept to interconnections of specific ports of the basic cells : a full-adder cell has 3 inputs (A, B, CI) and 2 outputs (SUM, CARRY-OUT). The 3 data inputs are functionally equivalent. The SUM output and CARRY-OUT output are different in that the SUM remains in the bit slice of the inputs, while the CARRY-OUT output goes to the next bit slice (see Figure 3).

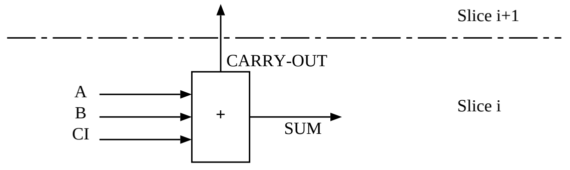

Figure 3: CARRY-OUT of basic adder cells define bit slices.

We observe thus the following : cells driven by a CARRY-OUT port of a adder cell can never belong to the same bit slice as that adder cell. This is the criterion to locate the bit slices in the circuit. Considering only the combinatorial part of a circuit, the method to divide the circuit into slices is then as follows :

1 Disconnect all CARRY-OUT connections and assign external inputs to the first slice.
2 Assign to current slice all cell instances whose inputs are driven by external inputs, or by outputs of cell instances that are already assigned to a slice.
3 Reconnect the CARRY-OUT connections of the cells belonging to the current slice.
4 Repeat from step 2 with a new slice until all cells are assigned to a slice.

Adder trees

After dividing the circuit into bit slices with this method it is determined if a connection is *horizontal* or *vertical*. Horizontal connections remain in the same bit slice, while vertical connections span more than one bit-slice. Inside a bit slice the connections between basic adder cells identify adder trees with multiple 1-bit inputs. It is required that the SUM output of an adder cell goes to exactly one data input of a next adder cell in the same bit slice. If this is not satisfied, the adder cells belong to different adder trees.

From the arithmetic properties of addition (commutative and associative) it follows that the multi-operand addition is independent of the order of the data inputs. This holds as well at the word level as at the bit level. Inside an adder tree, that is a part of a multi-operand adder, we can thus freely interchange the data inputs and transform the adder tree into an equivalent adder tree (see example of Figure 6).

When the data-inputs are interchanged, the CARRY-OUT signals of the different adder cells do not have the same functionality anymore. The multi-operand adder as a whole will however retain its functionality. It is even possible that less CARRY-OUT bits are needed as is illustrated in Figure 4 : when the 3 half-adders are replaced by 2 full-adders, there is one CARRY-OUT less. By contrast to logic synthesis, the use of arithmetic properties makes this kind of optimization possible in our method.

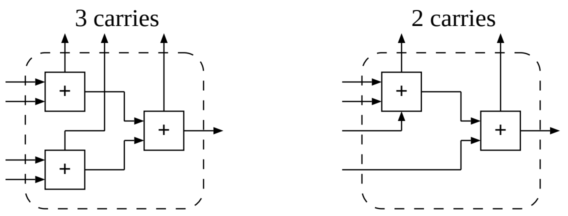

Figure 4: Arithmetic transformation can change number of carries. At the left the original adder tree with three carries and at the right a functionally equivalent adder tree with two carries.

Multi-operand adder

In order to keep the functionality of the circuit, the bit-slice adder-tree transformation above may only be applied when the adder trees form slices of a multi-operand adder. This is satisfied if CARRY-OUT bits of an adder tree are either all unconnected or are each connected to one of the data inputs of one adder tree in another slice. After locating the multi-operand adders in this way, each multi-operand adder contains slices of adder trees as shown in Figure 5.

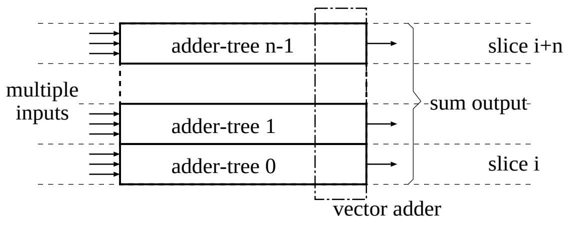

Figure 5: Multi-operand adder consists of adder trees in bit slices.

4 Area optimization

The identified multi-operand adder structures can now be optimized using arithmetic properties of addition. The first transformation is a minimization of the number of inputs into the slices of the adder. When, for instance, two equivalent signals are inputs to the same slice, they can be replaced by one input into the next slice of the multi-operand adder.

Equivalent signals

To be able to reduce the number of inputs, it is necessary to identify how input signals are related to each other. We search especially for those signals that represent logical known values (0 or 1), and for signals that are logically equivalent or inversely equivalent.

To find these relations between signals, their logical functions have to be computed and compared. Comparisons of logical functions are very easy when they are computed with Binary Decision Diagrams (BDD) [5]. The construction of BDDs can take much effort or even become impossible when the circuit is large. This is e.g. the case when the circuits contain large multipliers and these are exactly the type of circuits that we want to handle. In fact we only need the BDDs for the inputs of the adder structures. As in a data path several adder structures can be present after each other, this however means that inputs of following adder structures should be determined based on the outputs of previous adder structures.

To overcome the problem of calculating large BDDs, we can observe that calculating functions locally will detect most of the equivalent signals. When not all equivalent signals are identified, the result will still be valid although less optimal. So the BDDs are constructed with a limited depth of logical gates before the inputs of the adder structures. This depth does not have to be very large. E.g. for a Booth-multiplier [1], it is enough that the BDD covers a possible sign extension before the Booth-encoder, the Booth-encoder itself and the Booth-selector. When constant or equivalent signals are present at the inputs of the Booth-multiplier, they will also be present at the inputs of the adder structure of the multiplier.

CSD coding

The BDD construction identifies equivalent or constant inputs. These can then be used to perform the following arithmetic optimizations :

For each set of logically equivalent signals (and their inverse) the weights associated to their slices are added up and encoded with Canonical Signed Digits (CSD). This will yield the minimal number of inputs for that logical signal into the multi-operand adder. For an inverted signal the numerical value is negative and requires a correction term. E.g. an inverted input $\overline{a}$ in slice i gives :

$$2^i \times \overline{a} = -2^i \times a + -2^i$$

The CSD coding gives for each logical signal at maximum one input at maximum $n/2$ slices. When an input is negated, it is applied as an inverted input with a correction term, following the same rule.

After repeating the CSD coding for all logical equivalent signals, all correction terms together with constant signals are added. The result is one correction term which becomes a 2's-complement word input to the multi-operand adder.

When a slice has an input that is constant 1, the constant can be removed by inverting another input of the current slice, and adding the same signal to the next slice :

$$a + 1 = 2a - a + 1 = 2a + \overline{a}$$

When the slice has an even number of inputs, this can reduce the number of additions in the adder structure.

An example : signed multiplication

As an example a 4×4 signed multiplier is treated. As the output has 8 bits, sign extensions are done on the inputs outside the multiplier, so that they also contain 8 bits. When such sign extension is done outside the multiplier, there is no difference between the signed and the unsigned multiplier. The partial product terms are $p_{ij} = a_i \times b_j$ and give rise to the following add-array :

a_4	a_4	a_4	a_4	a_4	a_3	a_2	a_1	
p_{41}	p_{41}	p_{41}	p_{41}	p_{41}	p_{31}	p_{21}	p_{11}	b_1
p_{42}	p_{42}	p_{42}	p_{42}	p_{32}	p_{22}	p_{12}	0	b_2
p_{43}	p_{43}	p_{43}	p_{33}	p_{23}	p_{13}	0	0	b_3
p_{44}	p_{44}	p_{34}	p_{24}	p_{14}	0	0	0	b_4
p_{44}	p_{34}	p_{24}	p_{14}	0	0	0	0	b_4
p_{34}	p_{24}	p_{14}	0	0	0	0	0	b_4
p_{24}	p_{14}	0	0	0	0	0	0	b_4
p_{14}	0	0	0	0	0	0	0	b_4

Now for each equivalent signal, CSD coding is done. This gives e.g. for the partial product term p_{41} :

$$\begin{aligned} &p_{41} \times (2^7 + 2^6 + 2^5 + 2^4 + 2^3) = \\ &p_{41} \times (2^8 - 2^3) = \\ &p_{41} \times 2^8 + \overline{p_{41}} \times 2^3 - 2^3 \end{aligned}$$

The term $p_{41} \times 2^8$ is removed because we only retain 8 columns. The term $\overline{p_{41}} \times 2^3$ gives an inverted signal at the 4th column. The term -2^3 is a correction term. This procedure is repeated for all signals. All correction terms and constant signals are added and converted to 2's-complement. We then obtain the following add-array :

$$\begin{array}{cccccccc|c}
a_4 & a_4 & a_4 & a_4 & a_4 & a_3 & a_2 & a_1 & \\
\hline
 & & & & \overline{p_{41}} & p_{31} & p_{21} & p_{11} & b_1 \\
 & & & \overline{p_{42}} & p_{32} & p_{22} & p_{12} & & b_2 \\
 & & \overline{p_{43}} & p_{33} & p_{23} & p_{13} & & & b_3 \\
 & p_{44} & \overline{p_{34}} & \overline{p_{24}} & \overline{p_{14}} & & & & b_4 \\
1 & & & 1 & & & & &
\end{array}$$

The number of input signals into a slice is the sum of the inputs of the adder array and the carries of the previous slice. In the 8th column an even number of inputs are present, among which a constant 1. This makes possible to remove the constant by inverting the other input (which is a carry from the previous slice) and repeating the input in the next slice. This next slice is however not needed, and the adder-cell in the last column is thus replaced by an inverter. The constant 1 in the 5th column does not give any possibility for optimization in the same way as the slice contains an odd number of inputs. With redundancy removal one of the adder cells can be reduced afterwards due to this constant.

For the $N \times N$ signed multiplier a total of $N \times N + 2$ terms are added with $N \times (N - 1)$ adder cells. As can be seen from this example the sign-extensions at the inputs of the multiplier cause no overhead, because they are handled by the CSD coding of equivalent signals.

5 Delay optimization

Once the adder structures are located and the minimal number of input signals is determined, the actual adder is constructed. We use two simple but powerful techniques, namely bit-slice adder tree and linear-to-logarithmic transformation. Both are timing driven and yield area and delay efficient results.

Bit-slice tree

Inside a bit slice, the add operations are transformed using the commutativity and associativity of the addition. These transformations are timing driven and construct a tree of adder cells in such a way that the signals that arrive early are used first and those that are late are used last (see Figure 6) : if a bit has a large delay with respect to the other bits in same slice, it is for delay optimization best to add this bit as the last one to the multi-operand adder.

The transformations do not cause any area overhead. The result contains the least number of full-adders possible. The bit-slice tree gives a logarithmic delay in the number of input bits to the slice. The consequence of using this method is that there is no need to construct a *good* order of words into a multi-operand adder. The optimizations at the word level should therefore merge additions in few large adder structures without having to optimize the adder structures internally.

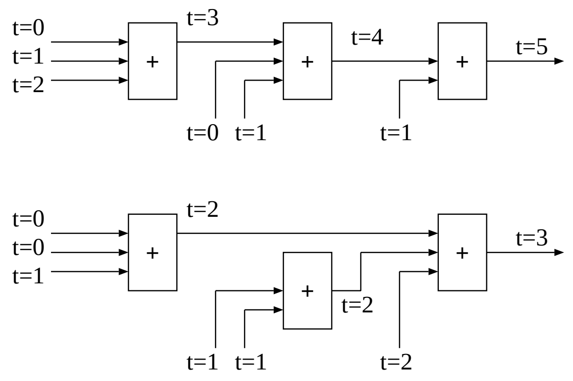

Figure 6: Timing driven optimization of adder trees reduces the total delay from 5 to 3, assuming unit delay for the adder cells.

Logarithmic addition

Taking the last adder cells of the adder trees in the bit slices of an adder structure together, a vector adder is obtained. This vector adder consists of adder cells that are connected through the CARRY-OUT pins and form thus a carry-ripple adder with two input words. When the delay of this ripple path is delay critical in the circuit, it can be speeded up by a linear to logarithmic transformation. This means that the delay of the vector adder is transformed from a linear dependency in the number of slices towards a logarithmic one.

A method similar to [8] is used to transform the ripple path into a look-ahead structure or anything in between. In contrast to [8] the arrival times of the inputs are used as they are not necessarily equal. The final look-ahead structure does not necessarily form a full binary carry-look-ahead adder and in most cases the logarithmic structure does not start at the first bits.

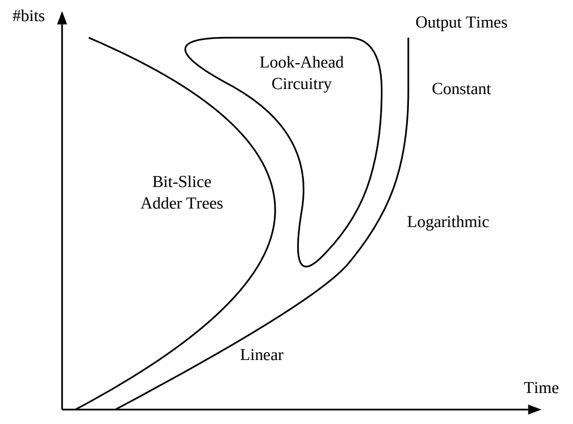

Figure 7: Timing driven look-ahead transformation for vector adder of multiplier

In the case of a multiplier, the arrival times before the vector adder are shown in Figure 7. We assume for this figure that all inputs of the multiplier arrive at the same time. After optimization with the timing driven look-ahead transformation, the first part of the final adder will remain a ripple adder, the second part becomes logarithmic, and the last part is also logarithmic but the delay of its output signal is constant, because the look-ahead circuitry fits after the arrival times of the inputs of the vector adder.

6 Experimental results

To compare the performance of the method presented in this paper with a commercially available logic synthesis tool (for which we used SYNOPSYS), unsigned multipliers have been used. This is done to avoid the impact of different correction term calculations on the results. From Figure 8 we see that the theoretically expected logarithmic delay is obtained by the presented method even for large multipliers. For small ones the result of both methods are almost the same. This result follows from the different methods that are used. We use a linear-to-logarithmic transformation making use of knowledge of the adder structures in the circuit. The logarithmic delay is obtained by the logarithmic depth of the adder trees combined with the logarithmic depth of the look-ahead transformation applied to the vector adder. Logic synthesis techniques use local transformations which can not efficiently handle many parallel ripple paths : the initial delay which is linear in the number of inputs and slices, can not be improved to obtain a logarithmic delay.

Concerning area efficiency, both methods are comparable for small multipliers, while for the large multipliers our method saves up to 15%. This is due to the larger area consumption of the logic synthesis tool when trying to speed up the large multipliers. Another advantage of the presented method is that it is very fast : it uses only information that is easy to collect and the transformations do not need any iteration. The results are produced within a few minutes, even for large adder structures, while the logic synthesis tool run for several hours to speed up the large multipliers. In the case of many parallel critical ripple paths, our method gives thus better results for both area and delay than what can be obtained with state-of-the-art logic synthesis.

7 Conclusion

A method was presented to optimize special structures in data paths of DSP applications. Although we only treated the case of adder structures, similar techniques can also be used for optimizing other kind of structures. The bitslice tree optimization can be used for all operators that satisfy the associativity and commutativity requirement. The

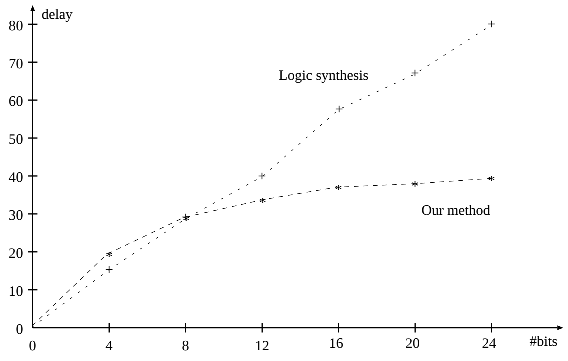

Figure 8: Delay comparison of presented method and logic synthesis for $N \times N$ multipliers

look-ahead optimization can be used for all types of ripple paths (e.g. most-significant-bit detector, equality comparator). The presented method for the adder structures requires that the operators are described with basic adder cells. Because of the provided optimizations this simplifies the descriptions of both the operators and the data paths. The results show the efficiency of the method both in terms of area, delay and CPU usage. The method guarantees a logarithmic delay regardless of the number of operands and of the word length.

References

[1] A.D. Booth : “A Signed Binary Multiplication Algorithm”, *Quarterly Journal of Mechanics and Applied Mathematics*, pp. 236-240, 1951.

[2] C.S. Wallace : “A suggestion for a fast multiplier”, *IEEE Trans. Electron. Comput.*, pp. 14-17, Feb. 1964.

[3] L. Dadda : “Some schemes for parallel multipliers”, *Alta Frequenza*, pp, 349-356, 1965.

[4] C. Leiserson, J. Saxe : “Optimizing Synchronous Circuitry by Retiming”, *Third Caltech Conference on VLSI*, Computer Science Press, 1983.

[5] R.E. Bryant : “Graph-based algorithms for boolean function manipulation”, *IEEE Transactions on Computers*, pp. 667-691, Aug. 1986.

[6] K.J. Singh, et al. : “Timing optimization of combinational logic”, *IEEE Int. Conf. on Computer-Aided Design*, pp. 282-285, 1988.

[7] S. Note, et al. : “Combined Hardware Selection and Pipelining in High-Performance Data-Path Design”, *IEEE Int. Conf. on Comp. Design*, pp. 328-331, 1990.

[8] J.P. Fishburn : “A Depth-Decreasing Heuristic for Combinational Logic”, *27th ACM/IEEE Design Automation Conference*, pp. 361-364, Jun. 1990.

[9] H.R. Srinivas, K.K. Parhi : “A Fast VLSI Adder Architecture”, *IEEE Journal of Solid-State Circuits*, pp. 761-767, May 1992.

[10] Z.-J. Mou, F. Jutand : “Overturned-Stairs Adder Trees and Multiplier Design,” *IEEE Transactions on Computers*, Vol 41, pp. 940-948, Aug. 1992.