計測自動制御学会東北支部 第191回研究集会 (2000.11.18)
資料番号 191-2

進化的グラフ生成手法とその応用

Evolutionary Graph Generation and Its Application

○本間尚文*†, 陳定君*†, 青木孝文*, 樋口龍雄*

○Naofumi Homma*†, Dingjun Chen*†, Takafumi Aoki*, Tatsuo Higuchi*

*東北大学 大学院情報科学研究科
†日本学術振興会特別研究員

*Graduate School of Information Sciences, Tohoku University
†The author is also a Research Fellow of the Japan Society for the Promotion of Science.

キーワード： 回路設計 (circuit design), 進化論的計算手法 (evolutionary computation), 算術演算回路 (arithmetic circuit), 論理合成 (logic synthesis), PC クラスタ (the PCs cluster)

連絡先： 〒980-8579 仙台市青葉区荒巻字青葉05 東北大学 大学院情報科学研究科 樋口研究室
本間尚文, Tel.: (022)217-7169, Fax.: (022)263-9406, E-mail: homma@higuchi.ecei.tohoku.ac.jp

1. Introduction

Arithmetic circuits are of major importance in today's computing and signal processing systems. Numerous algorithms for arithmetic circuits have been developed and implemented since the early days of digital computers, and newer ones are still being proposed. Most of the arithmetic circuits are designed by a designer who had trained in a particular way to understand the basic arithmetic algorithms. Even the state-of-the-art logic synthesis tools can provide only limited capability to create structural details of arithmetic circuits. Correspondingly, recent high-level synthesis techniques tend to employ module libraries containing basic arithmetic functional units, which are usually designed in advance as essential resources.

This paper proposes a new approach to designing arithmetic circuits using an evolutionary optimization technique called Evolutionary Graph Generation (EGG) (see [1)–3)] for earlier discussions on this topic). The key ideas of the proposed EGG system are to employ general graph structures as individuals and introduce new evolutionary operations to manipulate the individual graph structures directly without encoding them into other indirect representations, such as bit strings (used in GA [4)]) and trees (used in GP [5)]). This makes possible the generation of the target structure efficiently.

This paper discusses the synthesis of fast constant-coefficient multipliers as a typical example of an arithmetic design problem, since the high-speed multipliers with fixed coefficients are important in many practical DSP applications [6)–8)]. For designing fast constant-coefficient multipliers, we assume the

use of special number representation called *Signed-Weight* (SW) number system [9]. The SW number system makes possible the construction of compact counter-tree architecture for fast multiplication and multiply-add operation. At present, the combination of CSD (Canonic Signed-Digit) encoding technique [10] with the SW counter-tree architecture seems to provide the best possible approach to the practical hardware implementation of fast constant-coefficient multipliers. In this paper, we show that the EGG system can naturally create the optimal constant-coefficient multipliers, whose performances are comparable (or sometimes superior) to those of the multipliers designed by using CSD and SW arithmetic algorithms.

2. EGG system for arithmetic circuit synthesis

2.1 Basic concept of EGG

The Evolutionary Graph Generation (EGG) technique can be regarded as a unique variation of evolutionary computation techniques [11]. In general, evolutionary methods mimic the process of natural evolution, the driving process for emergence of complex structures well-adapted to the given environment. The better an individual performs under the conditions the greater is the chance for the individual to live for a longer while and generate offspring. As a result, the individuals are transformed to the suitable forms on the designer's defined constraint. In the EGG system, a graph representing a specific circuit structure is modeled as an individual, and a population of individual graphs is evolved by evolutionary operations.

The EGG system employs *circuit graphs* to represent circuit structures. A circuit graph G is de-

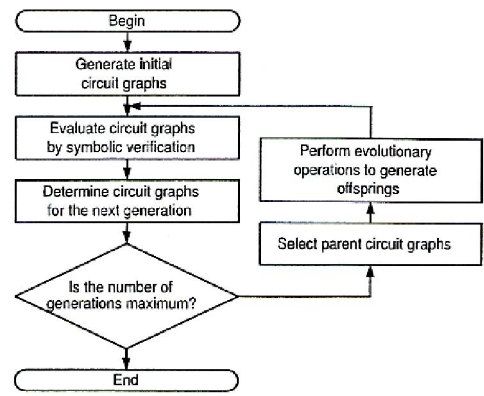


Fig. 1 EGG system flow.

fined by

$$G = (N(G), D(G)), \tag{1}$$

where $N(G)$ is the set of nodes and $D(G)$ is the set of directed edges. Nodes are of two classes: functional nodes and input/output nodes. Every node has its own name, the function type and input/output terminals. We assume that every directed edge must connect one output terminal (of a node) and one input terminal (of another node), and that each terminal has one edge connection at most. A circuit graph is said to be *complete* if all the terminals have an edge connection. In order to guarantee valid circuit structures, all the circuit graphs used in the EGG system are complete circuit graphs.

Fig. 1 shows the overall procedure of the EGG system. After the evolutionary run, every circuit graph in the population is evaluated by a symbolic model checking technique [3]. Then, the circuit graphs having higher scores are selected to perform *variation operations*, called *crossover* and *mutation*, to generate offsprings from the parents. The *crossover* operation, illustrated in Fig. 2 (a), recombines two parent graphs into two new graphs.

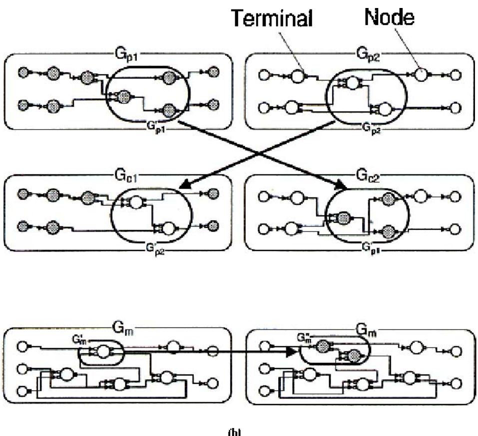


Fig. 2 Examples of evolutionary operations: (a) crossover, (b) mutation.

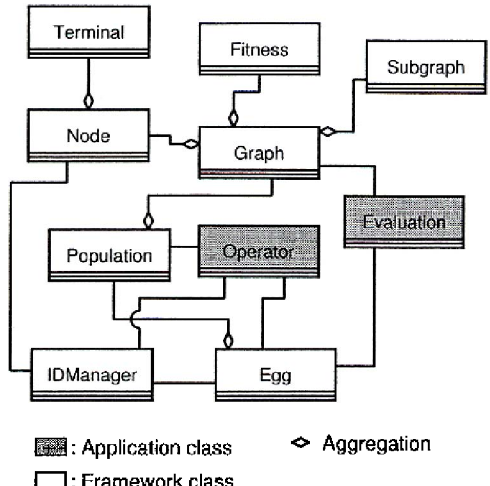


Fig. 3 Class diagram of EGG system.

When a pair of parent graphs G_{p1} and G_{p2} is selected from the population, the *crossover* operation determines a pair of subgraphs G_{p1}' and G_{p2}' to be exchanged between the parents, and generates offsprings by replacing the subgraph of one parent by that of the other parent. In this process, the system selects a subgraph G_{p1}' randomly from the mother circuit graph G_{p1}, and selects a *compatible* subgraph G_{p2}' from the father circuit graph G_{p2}, where "compatible" means that the cut sets for these subgraphs contain the same number of edges for both negative and positive directions. This ensures the completeness of the generated offsprings. The *mutation* operation, on the other hand, partially reconstructs the given circuit graph. This operation selects the subgraph randomly and replaces it with a randomly generated subgraph that is compatible with the original subgraph as shown in Fig. 2 (b).

2.2 EGG system implementation

The EGG system was developed on the basis of object-oriented programming approach in order to realize the highly flexible system that can be applied to various different applications systematically. In practice, we implement the EGG system based on a class relationship diagram as shown in Fig. 3. Note that if objects of one class contain objects of another, then the first class has an "aggregation" relationship with the second. The EGG system consists of framework (or invariable) classes and application (or variable) classes. The `Egg` class controls the overall work-flow and has an aggregation relationship with the `Population` class, which contains the basic individual model defined by the `Graph` class. The `Graph` also aggregates the `Node`, `Subgraph` and `Fitness` classes, where the `Node` contains the `Terminal` class. These seven classes forms the framework classes together with the `IDManager` class which controls the ID number of nodes.

The `Operator` and `Evaluation` classes are application-specific classes. The `Operator` class holds miscellaneous operations for handling circuit graphs. This

class also performs the functional verification of individuals. The other application class `Evaluation` gives "fitness" values to every individual. By modifying these application classes, the EGG system can be easily applied to a wide variety of design problems.

2.3 Arithmetic extension of EGG system

The EGG system described in the above subsection can be applied to a general class of graph synthesis problems. In the following, we describe an extension of the EGG system so as to manipulate arithmetic circuits efficiently. This extended version of EGG is called "Arithmetic-EGG", which employs a higher level of abstraction for arithmetic algorithms in order to reduce the size of search space. Arithmetic-EGG interprets an individual circuit graph as a data-flow graph representing specific arithmetic computation process. A directed edge in the data-flow graph represents the dependence of operands. Also, two attributes are assigned to each edge: (i) the type of number system used for operand encoding and (ii) the activated operand digits. In Arithmetic-EGG, we assume the use of positional number systems for operand representation.

A node in Arithmetic-EGG's data-flow graph, on the other hand, represents a specific arithmetic operation. Thus, the node itself has no circuit details at first. It can be transformed into a set of bit-level circuit elements only when the attributes of all the input operands are determined. Therefore, the actual interpretation of a node depends on the overall structure of the data-flow graph. Each node has a rule for generating the corresponding bit-level circuit interpretation.

3. Synthesis of fast constant-coefficient multipliers

3.1 Motivation and method of experiment

This section addresses the problem of synthesizing the architecture for multiplication in the form: $y = Rx$, where R is an integer coefficient, and x and y are the integer input and output. The reasons for choosing the constant-coefficient multiplier as a target function are as follows: (i) there are many possible choices for the multiplier structure for a specific coefficient R, and (ii) the complexity of the multiplier structure significantly varies with the coefficient value R.

One of the most important techniques in constant-coefficient multiplier design is to encode the target coefficient R by the Canonic Signed-Digit (CSD) number representation [10]. The CSD number representation is defined as a specific binary Signed-Digit (SD) number representation that contains the least number of non-zero digits. This encoding technique makes possible to reduce the number of partial products, which is equal to the number of non-zero digits. The CSD encoding combined with the fast partial product accumulation technique using parallel counter trees is widely used in practical DSP applications, such as high-frequency FIR filter architectures [6], [7].

As for compact counter tree design for partial product accumulation, the authors' group has recently proposed the Signed-Weight (SW) arithmetic [9]. The use of SW arithmetic instead of conventional two's complement arithmetic makes possible the construction of compact counter trees without using irregular arithmetic operations, such as sign extension and two's complementation. This

Table 1 Functional nodes

<table>
<tr><th>Name</th><th>Symbol</th><th>Function</th><th>Output Sign</th></tr>
<tr><td rowspan="3">SW 3-2 counter</td><td rowspan="3">3-2</td><td>3-input 2-output carry-free addition</td><td rowspan="3">Variable</td></tr>
<tr><td>3-input 2-output carry-free addition with 2-way branches</td></tr>
<tr><td>3-input 2-output carry-free addition with 3-way branches</td></tr>
<tr><td>Final stage adder</td><td>FSA</td><td>Carry-propagate addition with a bias canceling stage</td><td>Invariable</td></tr>
<tr><td>1-bit shifter</td><td>1-S</td><td>1-bit arithmetic shift</td><td>Invariable</td></tr>
<tr><td>2-bit shifter</td><td>2-S</td><td>2-bit arithmetic shift</td><td>Invariable</td></tr>
<tr><td>4-bit shifter</td><td>4-S</td><td>4-bit arithmetic shift</td><td>Invariable</td></tr>
<tr><td>Operand input</td><td>IN</td><td>Input signal</td><td>Variable</td></tr>
</table>

property was confirmed in the Field-Programmable Digital Filter (FPDF) architecture [9]. As a result, the combination of the CSD encoding technique with SW counter trees seems to provide the best possible approach to the practical hardware implementation of fast constant-coefficient multipliers at present. Thus, we have decided to compare the multipliers generated by the Arithmetic-EGG with the multipliers designed by hand employing the knowledge of the above techniques (CS-D plus SW arithmetic). The result shows that the Arithmetic-EGG can generate efficient multiplier structures whose performance and complexity are comparable with those designed by experienced designers.

Table 1 shows the six functional nodes used in our experiment. We employ SW 3-2 counters with programmable output polarity. The use of SW arithmetic makes possible to control the sign of individual data lines without using complicated two's complement arithmetic operations. The overhead of SW arithmetic is that an extra bias canceling stage is required at the output of the counter tree. We also allow the use of multiple-way (branched) outputs for every SW 3-2 counter node in order to reduce the complexity of SW counter tree by sharing hardware resources.

Table 2 Main parameter values

Parameter	Value
Population size	500
Max. num. of generations	500
Max. num. of nodes	50
Crossover rate	0.7
Mutation rate	0.2
Operand wordlength	16

The generated structures are evaluated by a combination of two different evaluation functions, *functionality* and *performance*. The functionality measure F evaluates the validity of the logical function compared with the target function. The performance measure P, on the other hand, is assumed to be the product of circuit delay D and number of inter-module interconnections A. First, we describe the functionality measure F in detail. Let R be the target coefficient given by the follow-

Table 3 DA product of multipliers: (a) the multipliers generated by Arithmetic-EGG, (b) the CSD multipliers.

Index	Coefficient	DA (a)	DA (b)	Index	Coefficient	DA (a)	DA (b)	Index	Coefficient	DA (a)	DA (b)
1	-2077	810	822	18	10075	1600	1824	35	17012	1324	1324
2	13492	2040	2250	19	2609	1280	1296	36	52	358	370
3	-20844	1548	1600	20	-17127	1564	1596	37	-29824	771	792
4	27155	2335	2375	21	5755	1556	1612	38	30321	1616	1616
5	-17614	1556	1572	22	-1749	1528	1528	39	19878	2290	2290
6	-1353	1276	1276	23	6674	1336	1336	40	-32424	1320	1320
7	10304	406	408	24	-24570	882	912	41	15315	1580	1620
8	-14338	456	458	25	-26881	1368	1376	42	30248	1336	1336
9	18639	1604	1604	26	4134	1072	1092	43	11452	1512	1576
10	-27400	1312	1320	27	14577	1372	1372	44	-15697	1572	1588
11	-4444	1256	1276	28	-1257	1260	1276	45	26204	2080	2275
12	-28961	1360	1392	29	3461	1332	1332	46	-28097	1384	1384
13	28959	1360	1392	30	-8390	1288	1312	47	22732	2245	2285
14	3548	819	819	31	14993	1560	1596	48	26605	1668	1668
15	-9566	1524	1572	32	-18597	1596	1628	49	-24213	2290	2375
16	-28565	1664	1664	33	9959	1560	1612	50	-27804	1576	1584
17	4833	1316	1316	34	-14886	1356	1356				

ing CSD representation:

$$R = r_0 2^0 + r_1 2^1 + r_2 2^2 + \cdots = \sum_{j=0}^{||R||-1} r_j 2^j \quad (2)$$

where $||R||$ denotes the length of the CSD representation of the coefficient R and $r_j \in \{-1, 0, 1\}$. As described in [3], the system checks the function of a circuit graph by symbolic verification and obtains the estimated coefficient $\hat{R}$, which may be written in the CSD notation as

$$\hat{R} = \hat{r}_0 2^0 + \hat{r}_1 2^1 + \hat{r}_2 2^2 + \cdots = \sum_{j=0}^{||\hat{R}||-1} \hat{r}_j 2^j \quad (3)$$

The similarity between R and $\hat{R}$ are evaluated by digit-coincidences for all the digit positions of the given two strings. Using the difference of the string lengths n $(= ||\hat{R}|| - ||R||)$, the correlation M_k of the two coefficient strings at the shift amount k $(0 \leq k \leq |n|)$ is defined by

$$M_k = \begin{cases} \frac{1}{||\hat{R}||} \sum_{j=0}^{||R||-1} \delta(\hat{r}_j - r_{j-k}) & n \geq 0 \\ \frac{1}{||R||} \sum_{j=0}^{||R||-1} \delta(\hat{r}_{j-k} - r_j) & n < 0, \end{cases}$$

where $\delta(x)$ is defined as

$$\delta(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0. \end{cases}$$

In the above calculation, we assume the values of the undefined digit positions to be 0 for both coefficient strings. Using this correlation function, the functionality measure F is defined as

$$F = \max_{0 \leq k \leq |n|} \left[100 M_k - C_1 k\right], \quad (4)$$

where $C_1 = 2$ in this experiment.

On the other hand, the performance measure P is defined as

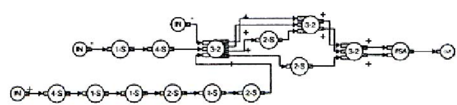
$$P = \frac{C_2}{DA}, \quad (5)$$

where D is the number of counter stages, and A is the total number of inter-module interconnections in the translated bit-level circuit. We use $F + P$ as a total fitness function, where the ratio P_{max}/F_{max} is adjusted about 5/100 by tuning the constant C_2.

3.2 Experimental results

Table 2 shows major parameters of Arithmetic-EGG used in this experiment. Table 3 shows the result of a set of evolutionary runs, in which Arithmetic-EGG generates 50 multipliers whose coefficients are ranging from −32424 to 30321. Total 50 coefficients are selected randomly as target values out of 16-bit coefficients ($-32767 \sim 32767$). In Table 3, we show the DA product (the number of counter stages × the number of inter-module interconnections) of the multipliers generated by Arithmetic-EGG (a) and that of the corresponding CSD multipliers using the optimal Wallace tree architecture (b). From this table, we can confirm that all the generated solutions exhibit almost same level of performance compared with the optimal CSD multipliers. We can see some improvements in DA complexity for evolved multipliers compared with th reference designs.

Fig. 4 (a) is the best solution generated by Arithmetic-EGG for the target coefficient $R = 10075$. Fig. 4 (b), on the other hand, shows the conventional CSD multiplier using Wallace tree architecture consisting of the least stages of SW 3-2 counters. Note that the solution (a) obtained by Arithmetic-EGG employs an SW 3-2 counter with two-way carry-save branches at the first stage, and this hardware resource is shared by the successive stages. Compared with the structure of Fig. 4 (b), this feature significantly reduces the complexity of the corresponding bit-level circuit configurations as



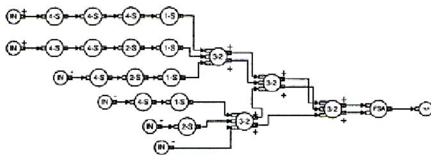
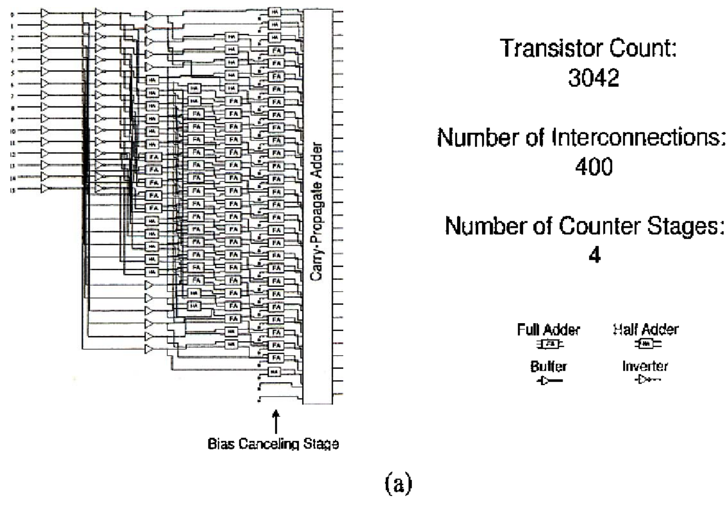


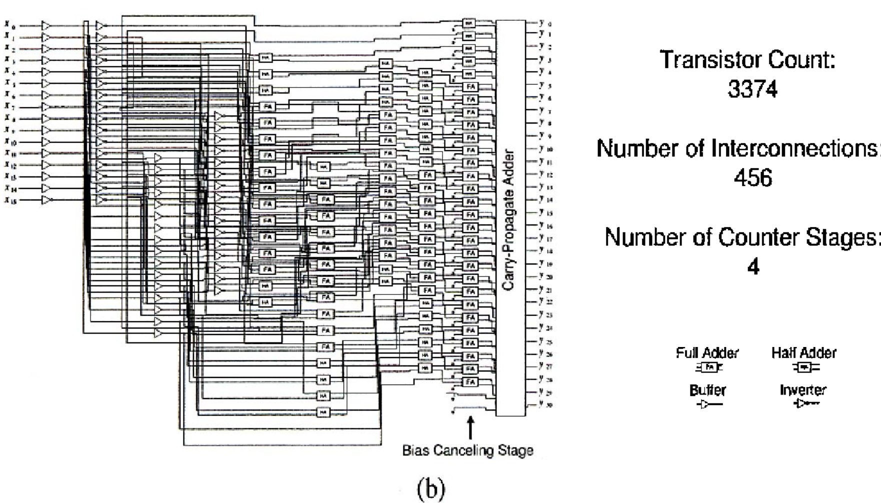
Fig. 4 Data-flow graphs for the multiplier of $R = 10075$: (a) the best solution generated by Arithmetic-EGG, (b) the CSD multiplier using Wallace tree architecture.

shown in Fig. 5. Recently, some papers have presented the method of reducing the complexity of the constant-coefficient multipliers by inserting such branches at adders [8), 12)]. Note here that the EGG system can derive the optimal structure without using the knowledge of these techniques.

An important issue to be addressed for practical application of EGG system is its computation time. In the experiment of Table 3, the time for each evolutionary run is about 3.6 hours on Sun Ultra 60 workstation (CPU: 360MHz, Memory: 1.15GB). In order to reduce the time for experiments of EGG-based circuit synthesis, we have recently introduced inexpensive COTS (Commercial Off-The-Shelf) cluster computing technique. Fig. 6 shows the 5-node Linux PC cluster designed for EGG system. Each node of the cluster is Linux PC with 700MHz Pentium III and 1GB memory. The interconnection network is a 100Base-TX Ethernet. By employing the coarse-grained model [13)], focusing simply on the latest modifications introduced in the original EGG system, we have implemented the



(a)



(b)

Fig. 5 Multiplier configurations corresponding to the graphs of Fig. 4: (a) the best solution generated by Arithmetic-EGG, (b) the CSD multiplier using Wallace tree architecture.

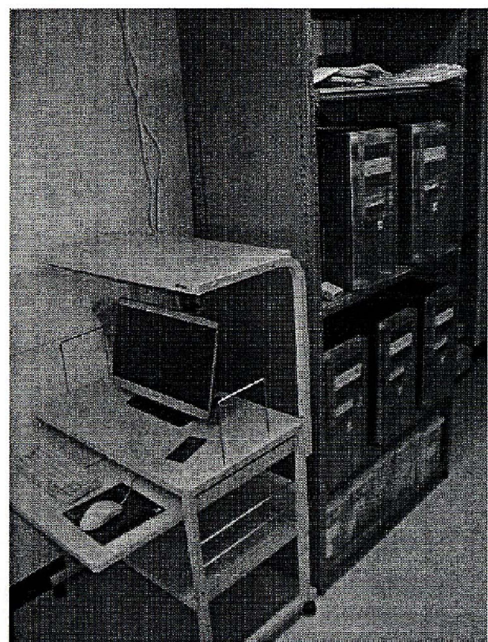


Fig. 6 5-node Linux PC cluster for EGG system.

parallel EGG system on the cluster of PCs using Message-Passing Interface (MPI). Compared with the original EGG system implemented on a single computer, the parallel EGG system achieved the speed-up of about 11 times synthesizing the same kinds of constant-coefficient multipliers. Thus, this kind of COTS parallel processing technique provides a potential possibility of building an EGG-based CAD system that can be applied to various practical circuit design problems.

4. Conclusion

In this paper, we presented an efficient graph-based evolutionary optimization technique called Evolutionary Graph Generation (EGG), and its application to the design of fast constant-coefficient multipliers. We also implemented the parallel EGG system on the cluster of PCs using Message-Passing Interface (MPI). The detail of the parallel EGG system will be reported in future papers.

References

1) N. Homma, T. Aoki, and T. Higuchi,"Design of arithmetic circuits based on evolutionary graph generation," *Proc. of the Workshop on Synthesis And System Integration of Mixed Technologies*, No. 1-3, pp. 31–38, October 1998.

2) T. Aoki, N. Homma, and T. Higuchi, "Evolutionary design of arithmetic circuits," *IEICE Trans. Fundamentals*, Vol. E82-A, No. 5, pp. 798–806, May 1999.

3) N. Homma, T. Aoki, and T. Higuchi, "Evolutionary graph generation system with symbolic verification for arithmetic circuit design," *Electronics Letters*, Vol. 36, No. 11, pp. 937–939, May 2000.

4) F. J. Miller, P. Thomson, and T. Fogarty,"Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study," *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pp. 105 – 131, September 1997.

5) R. J. Koza, H. F. III, Bennett, D. Andre, A. M. Keane, and F. Dunlap,"Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Trans. Evolutionary Computation*, Vol. 1, No. 2, pp. 109 – 128, July 1997.

6) B. C. Wong and H. Samueli,"A 200-MHz all-digital QAM modulator and demodulator in 1.2-μm CMOS for digital radio applications,"

IEEE J. Solid-State Circuits, Vol. 26, No. 12, pp. 1970 – 1979, December 1991.

7) K. Khoo, A. Kwentus, and A. N. Willson,"A programmable FIR digital filter using CS-D coefficients," *IEEE J. Solid-State Circuits*, Vol. 31, No. 6, pp. 869 – 874, June 1996.

8) R. I. Hartley,"Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, Vol. 43, No. 10, pp. 677 – 688, October 1996.

9) T. Aoki, Y. Sawada, and T. Higuchi,"Signed-weight arithmetic and its application to a field-programmable digital filter architecture," *IEICE Trans. Electronics*, Vol. E82-C, No. 9, pp. 1687–1698, September 1999.

10) K. Hwang, *Computer arithmetic: principles, architecture, and design*, John Wiley & Sons, 1979.

11) T. Back, U. Hammel, and P. H. Schwefel,"Evolutionary computation: Comments on the history and current state," *IEEE Trans. Evolutionary Computation*, Vol. 1, No. 1, pp. 3 – 13, April 1997.

12) A. G. Dempster and M. D. Macleod,"Constant integer multiplication using minimum adders," *IEE Proc. Circuits Devices Syst.*, Vol. 141, No. 5, pp. 407 – 413, October 1994.

13) J. Grefenstette,"Optimization of control parameters for genetic algorithms," *IEEE Trans. System, Man and Cybernetics*, Vol. 16, No. 1, pp. 150 – 157, 1986.