社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－ARC－171 (1)
2007－EMB－ 3 (1)
2007／1／22

動作合成による倍精度浮動小数点型加算器の設計事例

原祐子† 冨山宏之† 本田晋也† 高田広章† 石井克哉‡
† 名古屋大学 大学院情報科学研究科
‡ 名古屋大学 情報連携基盤センター

概要 近年，FPGAの容量の増加に伴い，浮動小数点型演算IPをFPGAに組み込めるようになってきた．しかし，依然としてそのコストが高いことが問題視されている．本論文では，動作合成を用いて，倍精度浮動小数点型の加算器および加減算器を設計した事例を報告する．動作合成において様々な工夫を行い，同じCプログラムから15通りの加算器と21通りの加減算器を設計する．実験結果から，動作合成の有効性ならびに動作合成における各種最適化技術の有効性を示す．

Behavioral Synthesis of Double-Precision Floating-Point Adders

Yuko Hara† Hiroyuki Tomiyama† Shinya Honda† Hiroaki Takada† Katsuya Ishii‡
† Graduate School of Information Science, Nagoya University
‡ Information Technology Center, Nagoya University

Abstract Recently, the continuously growing capacity of FPGAs has enabled us to place floating-point arithmetic IPs on FPGAs. The required area for floating-point computations, however, is still high. This paper presents several techniques to design double-precision floating-point adders and adder/subtracters for FPGAs through behavioral synthesis. We generate totally 15 adders and 21 adder/subtracters from the same addition and subtraction functions written in C. From the experimental results, we show the effectiveness of behavioral synthesis techniques for complex arithmetic circuits.

1 Introduction

Traditionally, floating-point arithmetic units have rarely been used in FPGAs due to their high cost. A designer had to convert floating-point numbers in system-level specification into fixed-point ones before starting hardware design. However, the conversion from floating-point numbers into fixed-point ones is very time-consuming and error-prone.

Recently, the continuously growing capacity of FPGAs has enabled us to place single-precision floating-point arithmetic IPs, or even double-precision ones, on FPGAs. In most cases, floating-point arithmetic IPs are provided in the form of gate-level netlist or register-transfer level description in HDL. With such gate- or RT-level IPs, it is often impossible to satisfy application-specific design requirements such as area, clock frequency, latency, and so on. Although RT-level IPs are customizable or modifiable to some extent, it is not easy to significantly change their latency or area.

In this paper, we generate adders and adder/subtracters dealing with double-precision floating-point formats for FPGAs through behavioral synthesis. Behavioral synthesis is a technology which automatically generates an RT-level circuit from a sequential program [1]. Behavioral synthesis techniques have extensively been studied for more than two decades, and behavioral synthesis tools are now being used in practice, particularly in Japanese industry [2, 3]. Using behavioral synthesis, various circuits with different area and performance can be generated from the same sequential program by specifying different synthesis options and constraints.

In this case study, we have designed totally 15 adders and 21 adder/subtracters for double-precision floating-point numbers with using a behavioral synthesis tool. Thus, a designer can select the most appropriate design for application-specific requirements. Goals of the experiments described in this paper are two-fold. One is to test the effectiveness of behavioral synthesis techniques for complex arithmetic circuits. The other is to develop a set of guidelines for design space exploration using behavioral synthesis.

The rest of this paper is organized as follows. Section 2 explains experimental environments used in Sections 3 and 4. Section 3 shows a case study on behavioral synthesis of adders. Section 4 studies synthesis of adder/subtracters. Section 5 concludes this paper with a summary.

Table. 1: Experimental results for adders

Clock const. (MHz)	25		50		100	
Design goal	Perf.	Area	Perf.	Area	Perf.	Area
States	3	21	5	22	10	25
Area (slices)	6,039	5,654	4,913	5,875	3,667	5,875
Clock freq. (MHz)	23.6	29.4	29.4	26.3	49.6	37.3
Clock cycles	3	21	5	22	10	25
Exec. time (ns)	127.3	723.2	169.9	835.4	201.6	671.0
Area-delay ($\times 10^3$)	768.7	4,088.7	834.9	4,918.0	739.3	4,107.0

2 Design Environment

This section describes the design environments used in the experiments of Section 3 and 4.

2.1 Design Tools

We use a commercial behavioral synthesis tool eXCite from YXI [4]. A C program is input to eXCite with several optimization options and design constraints. Then, an RT-level description in VHDL or Verilog-HDL is generated. The optimization options and design constraints include clock frequency, the number of functional units, and so on. For logic synthesis and place-and-route, we use Synplify Pro from Synplicity [5] and XST from Xilinx [6], respectively. Logic synthesis and place-and-route are optimized for performance. Xilinx Spartan 3 is specified as a target device.

2.2 Double-Precision Floating-Point Addition Program in C

We use a double-precision floating-point addition function *double_add* from the SoftFloat suite [7]. The SoftFloat suite is an open-source software implementation of the IEC/IEEE Standard for binary floating-point arithmetic. It includes several fundamental arithmetic operations such as addition, subtraction, multiplication, division, and so on, supporting both single-precision and double-precision floating-point formats. In this paper, we select double-precision floating-point addition due to its high computational complexity and importance.

Note that the size of the double-precision floating-point addition program is relatively large compared with DSP kernels which were often used in the past literature on behavioral synthesis. The addition program consists of more than 600 lines of C code. After behavioral level optimization such as common subexpression elimination and dead code elimination, there exist 298 arithmetic and logic operations, 307 assignments, 77 *if* statements, 26 *goto/return* statements, and so on.

2.3 Evaluation Metrics

We evaluate the quality of designs with three metrics, i.e., *area*, *execution time* and *area-delay product*. Area is measured by the number of slices occupied for the designs. Execution time is defined as the product of clock period and execution cycles. Area-delay product is defined as the product of area and execution time. Since in general area and execution time are in trade-off relation, area-delay product is useful to evaluate the overall quality of the designs.

3 Synthesis of Adders

In this section, we first show a case study on synthesizing adders for double-precision floating numbers. Then, we employ three techniques to improve the quality of adders.

3.1 Simple Synthesis of Adders

First, we synthesized adders from the double-precision floating-point addition function explained in Section 2.2. The clock frequency constraint was set to be 25, 50 and 100 MHz. For each clock frequency, we specified two types of synthesis goal to behavioral synthesis tool eXCite: one is the performance maximization and the other is the area minimization by sharing components as much as possible.

Then, we executed logic synthesis and place-and-route to evaluate area and clock frequency of the designs. The results are shown in Table 1. The row "Design goal" of Table 1 represents the synthesis goal.

When the synthesis goal is the area minimization, the number of required components is smaller than that for performance maximization since several components are temporally shared [1]. The total area, however, is larger as imposing a more severe constraint on clock frequency. This is because more multiplexers and registers are required, and this area overhead is larger than the area saving obtained by reduced components. As a severe clock constraint is given, the control path becomes compilicated and its area is increased. Moreover, the execution time becomes

[1] Information on the types and the numbers of components requried by each design are omitted due to the limited space.

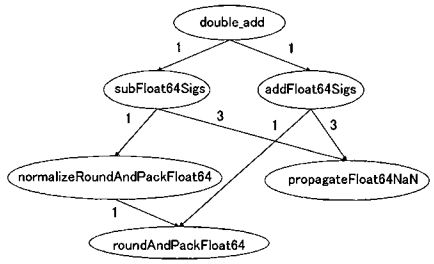

Fig. 1: A call graph of double-precision floating-point addition function

longer, which results in severe performance degradation. In terms of the area-delay product, the synthesis goal for performance yields better designs than that for area.

3.2 Synthesis of Adders with Goto Conversion

The double-precision floating-point addition function *double_add* from the SoftFloat suite consists of multi-level function calls. A call graph for *double_add* is shown in Fig. 1. In Fig. 1, *addFloat64Sigs* directly calls *propagateFloat64NaN* three times and *roundAndPackFloat64* once. *subFloat64Sigs* directly calls *propagateFloat64NaN* three times, while it indirectly calls *roundAndPackFloat64* once via *normalizeRoundAndPackFloat64*. *double_add* has two arguments *a* and *b*. If both of their signs are same, *double_add* calls *addFloat64Sigs*, otherwise *double_add* calls *subFloat64Sigs*. In addition to the functions shown in Fig. 1, there exist more than ten functions, but they are omitted here since they are small and to be inlined.

Unless specific options are given, eXCite inlines all callee functions and generates one large function. When synthesizing *double_add* in Fig. 1, for example, all the functions are inlined into *double_add*. In general, functional units can be shared among the inlined functions, which leads to a small circuit area. When large functions which are called multiple times are inlined, however, the number of states is increased. This makes its control path complicated, leading to inefficient designs. This problem is avoided by applying *goto conversion* to such functions. Goto conversion is a transformation to replace function calls with goto statements, and has been used in some behavioral synthesis tools such as [2, 3].

An example of goto conversion is shown in Fig. 2. Fig. 2 (a) is an original C source code. In this example, there exist two function calls to *func1* in *func2*.

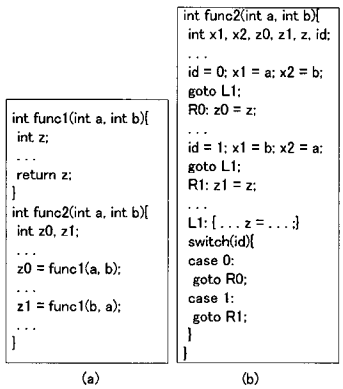

Fig. 2: (a) An emample of an original program (b) The rewritten program with goto conversion

Without goto conversion, the body of *func1* is inlined twice. This might lead to large number of states, which results in the complicated control logic. In Fig. 2 (b), only *func2* is rewritten with goto conversion. First, when a goto statement for label *L1* is executed above label *R0*, the control flow jumps to label *L1* and calls *func1*. After executing *func1*, the control flow jumps back to label *R0* from a *switch* statement described below label *L1* since *id* is zero. When the control flow executes a goto statement for label *L1* above label *R1*, it behaves as same as above. In the program in Fig. 2 (b), the body of *fun c1* is inlined only once in spite of being executed at two locations in the C source code. In addition, the components required by *func1* can be shared with other operations as same as inlining.

In this section, goto conversion is applied to synthesis of double-precision floating-point adders. The candidate functions for goto conversion are *propagateFloat64NaN* and *roundAndPackFloat64* since they are relatively large and called several times. Using goto conversion, we have designed three adders as follows, for each clock constraint.

RG: goto conversion is applied to *roundAndPackFloag64* with inlining *propagateFloat64NaN*

PG: goto conversion is applied to *propagateFloat64NaN* with inlining *roundAndPackFloag64*

RPG: goto conversion is applied to both *roundAndPackFloat64* and *propagateFloat64NaN*

We set three constraints on clock frequency, i.e., 25, 50 and 100 MHz. Based on the results in Section 3.1, we specified performance maximization as our synthesis goal. The experimental results are shown in Table

Table. 2: Experimental results for adders with goto conversion

Clock const. (MHz)	25			50			100		
Method	RG	PG	RPG	RG	PG	RPG	RG	PG	RPG
States	3	3	3	5	5	5	10	10	10
Area (slices)	5,503	4,799	5,854	4,541	4,614	5,158	3,578	5,468	5,034
	(0.91)	(0.80)	(0.97)	(0.87)	(0.94)	(1.05)	(0.98)	(1.49)	(1.37)
Clock freq. (MHz)	27.6	21.7	28.7	26.7	25.6	29.2	50.1	43.9	49.2
	(1.03)	(0.72)	(1.19)	(1.10)	(1.01)	(1.09)	(0.99)	(1.13)	(1.01)
Clock cycles	3	3	3	5	5	5	10	10	10
Exec. time (ns)	108.5	138.4	104.7	187.5	171.4	184.5	199.8	227.9	103.3
	(0.83)	(1.09)	(0.82)	(1.10)	(1.01)	(1.09)	(0.99)	(1.13)	(1.01)
Area-delay ($\times 10^3$)	597.3	664.2	613.0	851.2	791.0	951.9	715.0	1,246.2	1,023.5
	(0.78)	(0.86)	(0.80)	(1.02)	(0.95)	(1.14)	(0.97)	(1.69)	(1.38)

2. Numbers in parentheses in Table 2 are normalized values where baseline is "Perf." in Table 1.

When the clock constraint is 25 or 100 MHz, it is the best to apply goto conversion only to *roundAndPackFloag64* in terms of area-delay product. When the clock constraint is 50 MHz, applying goto conversion only to *propagateFloat64NaN* is the best. When employing RPG on 50 MHz, PG on 100 MHz or RPG on 100 MHz, the areas are larger than those without goto conversion. This is mainly because, with use of goto conversion, the number of registers is largely increased by gate-level retiming. Particularly as more severe clock constraint is given, larger numbers of registers are required. Then, the total area was also increased due to the increase of registers.

In terms of area-delay product, the best design is generated with RG on 25 MHz clock constraint among all the results in Table 2.

4 Synthesis of Adder/Subtracters

In this section, we design adder/subtracters from an addition and subtraction functions which supports the double-precision floating-point format. Adder/subtracter is an arithmetic circuit which computes both addition and subtraction. In Section 4.1, we show the results with a simple method to merge these two functions. Then, in Section 4.2, we employ goto conversion to generate improved designs compared to the simple design.

4.1 Simple Synthesis of Adder/Subtracters

The double-precision floating-point subtraction function *double_sub* from the SoftFloat suite has a similar structure as addition function *double_add*. *double_sub* takes two arguments *a* and *b*. If both the signs of *a* and *b* are same, *subFloat64Sigs* is called, otherwise *addFloat64Sigs* is called. A call graph of *double_add* and *double_sub* is shown in Fig. 3. *double_addsub* is a new function which is defined to merge *double_add* and *double_sub*. An adder/subtracter for

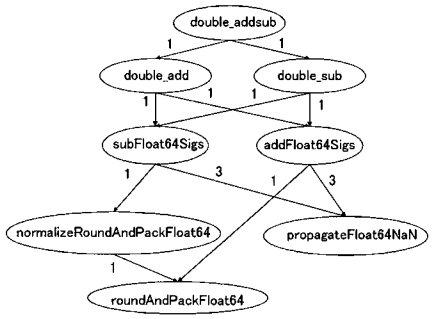

Fig. 3: A call graph of double-precision floating-point addition and subtraction functions

```
typedef unsigned long long float64;

float64 double_addsub
            (float64 a, float64 b, char id){
  float64 z;   /* output */

  if(id == 0) z = double_add(a, b);
  else z = double_sub(a, b);
  return z;
}
```

Fig. 4: A new function which is defined to merge *double_add* and *double_sub*

double-precision floating-point format can be generated from *double_addsub*.

First, in this section, *double_sub* was singly synthesized, and then, the new function *double_addsub* was synthesized. *double_addsub* has three input values; two arguments *a* and *b*, and a 1-bit *id*. This program is partially shown in Fig. 4. The bodies of *double_add* and *double_sub* are omitted here. Variables defined as *double* type are automatically converted to *float64* type, which is in actual *unsigned long long* type. The bitwidth of *id* is reduced to one by an option of eX-Cite although it is originally defined as eight bits in the C source code. If *id* is equal to zero, *double_add* is called, otherwise *double_sub* is called. The experimental results for *double_sub* and a function which merges *double_add* and *double_sub* are shown in Table 3.

For adder/subtracters, the area is almost same

Table. 3: Experimental results for subtracters and adder/subtracters

Function	Subtracters			Adder/subtracters		
Clock const. (MHz)	25	50	100	25	50	100
States	3	5	10	3	5	10
Area (slices)	5,041	5,184	3,737	8,145	7,381	9,696
Clock freq. (MHz)	21.0	19.6	50.4	16.1	20.5	32.6
Clock cycles	3	5	10	3	5	10
Exec. time (ns)	142.6	254.5	198.5	186.6	244.2	306.5
Area-delay ($\times 10^3$)	768.7	1,319.3	741.9	1,519.8	1,802.6	2,971.4

Table. 4: Experimental results for adder/subtracters with components sharing

Clock const. (MHz)	25	50	100
States	7	11	21
Area (slices)	8,998	8,264	5,943
	(1.11)	(1.12)	(0.61)
Clock freq. (MHz)	15.4	20.9	22.0
	(1.04)	(0.98)	(0.72)
Clock cycles	4	6	11
Exec. time (ns)	259.9	287.3	242.1
	(1.39)	(1.18)	(0.79)
Area-delay ($\times 10^3$)	2,338.2	2,374.0	1,439.0
	(1.54)	(1.32)	(0.48)

as the sum of areas of *double_add* and *double_sub*. The number of components required by an adder/subtracter is also same as the sum of those of *double_add* and *double_sub*. This is because *double_add* and *double_sub* were speculatively executed even though execution of *double_add* and *double_sub* must be exclusive. Therefore, the components are hardly shared between the two functions.

Next, in order to prevent from the speculative execution of *double_add* and *double_sub*, we explicitly inserted a clock boundary after the condition test in Fig. 4 so that *double_add* and *double_sub* are mutually executed. This helps components be shared. The experimental results are shown in Table 4. Values in parentheses in Table 4 are normalized by "Adder/subtracters" in Table 3.

In terms of area in Table 4, when the clock constraint is 25 or 50 MHz, the areas are increased compared to the results in Table 3. This is mainly because the number of multiplexers is increased to share the components. When the clock constraint is 100 MHz, on the other hand, the area is reduced since the number of registers is significantly reduced. Note that, in general, the speculative execution requires more registers.

4.2 Synthesis of Adder/Subtracters with Goto Conversion

As explained above, *double_add* and *double_sub* have very similar structures. Fig. 3 shows that both *double_add* and *double_sub* call *addFloat64Sigs* and *subFloat64Sigs*. In the experiments in Table 3, both *addFloat64Sigs* and *subFloat64Sigs* are inlined twice

```
float64 double_add( float64 a, float64 b ){
  int aSign, bSign;

  aSign = extractFloat64Sign(a);
  bSign = extractFloat64Sign(b);
  if ( aSign == bSign ) {
    return addFloat64Sigs(a, b, aSign);
  }else {
    return subFloat64Sigs(a, b, aSign);
  }
}
```

(a)

```
float64 double_sub( float64 a, float64 b ){
  int aSign, bSign;

  aSign = extractFloat64Sign(a);
  bSign = extractFloat64Sign(b);
  if ( aSign == bSign ) {
    return subFloat64Sigs(a, b, aSign);
  }else {
    return addFloat64Sigs(a, b, aSign);
  }
}
```

(b)

```
float64 double_addsub
          (float64 a, float64 b, char id){
  float64 z;   /* output */
  int aSign, bSign;

  aSign = extractFloat64Sign(a);
  bSign = extractFloat64Sign(b);
  if((aSign == bSign && id == 0)
      || (aSign != bSign && id != 0)){
    z = addFloat64Sigs(a, b, aSign);
  }else{
    z = subFloat64Sigs(a, b, aSign);
  }
  return z;
}
```

(c)

Fig. 5: A new function which is defined to merge *double_add* and *double_sub*

since the functions are called by both *double_add* and *double_sub*. This makes designs large. To avoid inlining large functions such as *addFloat64Sigs* and *subFloat64Sigs*, firstly, we employ goto conversion to *addFloat64Sigs* and *subFloat64Sigs*. This design is denoted as **ASG**.

The C source code of *double_add* and *double_sub* are shown in Fig. 5 (a) and (b), respectively. Fig. 5 (a) and (b) have little differences except a condition in the *if* statement to determine a function to be called. Note that *extract64Sign* is a small function which gets a sign bit of an argument, and *aSign* and *bSign* represents the signs of a and b, respectively. Then, we define a new function whose condition of *if* statement is rewritten from *if* statements of *double_add* and *double_sub* in order to directly call *addFloat64Sigs* and *subFloat64Sigs* from the new function as shown in Fig. 5 (c). In this function, *addFloat64Sigs* is called when the signs of a and b are same and *id* is zero or when *double_add* and *double_sub* are not same and *id* is not zero, i.e., *id* is one, otherwise *subFloat64Sigs* is called. Goto conversion is not applied to any functions. This design is denoted as **NGT**.

Next, goto conversion is applied to *roundAndPack-*

Float64 and *propagateFloat64NaN* for *double_addsub* described in Fig. 5 (c). Then, we obtain three designs **NRG**, **NPG** and **NRPG**. The five designs are summarized as follows.

ASG: goto conversion is applied to *addFloat64Sigs* and *subFloat64Sigs*

NGT: *addFloat64Sigs* and *subFloat64Sigs* are directly called in a new function without goto conversion

NRG: goto conversion is applied to *roundAndPackFloag64* in addition to a technique **NGT**

NPG: goto conversion is applied to *propagateFloat64NaN* in addition to a technique **NGT**

NRPG: goto conversion is applied to *roundAndPackFloat64* and *propagateFloat64NaN* in addition to a technique **NGT**

Table. 5: Experimental results for adder/subtracters with goto conversion

Clock const. (MHz)	25				
Technique	ASG	NGT	NRG	NPG	NRPG
States	3	3	3	3	3
Area (slices)	7,937	6,049	5,446	4,865	5,787
	(0.97)	(0.74)	(0.67)	(0.60)	(0.71)
Clock freq. (MHz)	21.1	23.7	30.3	23.0	23.2
	(0.76)	(0.68)	(0.53)	(0.70)	(0.69)
Clock cycles	3	3	3	3	3
Exec. time (ns)	142.4	126.8	99.1	130.6	129.4
	(0.76)	(0.68)	(0.53)	(0.70)	(0.69)
Area-delay ($\times 10^3$)	1,130.3	766.8	539.9	635.4	748.6
	(0.74)	(0.51)	(0.36)	(0.42)	(0.49)
Clock const. (MHz)	50				
Technique	ASG	NGT	NRG	NPG	NRPG
States	5	5	5	5	5
Area (slices)	7,190	4,408	4,702	4,732	4,691
	(0.97)	(0.60)	(0.64)	(0.64)	(0.64)
Clock freq. (MHz)	22.3	28.7	27.4	30.6	26.7
	(0.92)	(0.71)	(0.75)	(0.67)	(0.77)
Clock cycles	5	5	5	5	5
Exec. time (ns)	224.5	174.0	182.7	163.4	187.4
	(0.92)	(0.71)	(0.75)	(0.67)	(0.77)
Area-delay ($\times 10^3$)	1,613.9	766.8	858.8	773.4	879.0
	(0.90)	(0.43)	(0.48)	(0.43)	(0.49)
Clock const. (MHz)	100				
Technique	ASG	NGT	NRG	NPG	NRPG
States	10	10	10	10	10
Area (slices)	5,382	4,054	4,393	5,823	5,227
	(0.56)	(0.42)	(0.45)	(0.60)	(0.54)
Clock freq. (MHz)	38.5	44.3	47.9	47.9	47.0
	(0.85)	(0.74)	(0.68)	(0.68)	(0.69)
Clock cycles	10	10	10	10	10
Exec. time (ns)	259.5	225.9	208.7	208.9	212.6
	(0.85)	(0.74)	(0.68)	4(0.68)	(0.69)
Area-delay ($\times 10^3$)	1,396.8	915.9	916.8	1,216.4	1,111.5
	(0.47)	(0.31)	(0.31)	(0.41)	(0.37)

The experimental results are shown in Table 5. Values in parentheses in Table 5 are normalized by "Adder/subtracters" in Table 3. in Table 3. All the results with ASG have better results than the results in Table 3. These results, however, were the worst among with five techniques in Table 5. When the clock constraint is 25 MHz, NRG has the best result. In this case, all the techniques with goto conversion, i.e., NRG, NPG and NRPG have better results than the one with NGT. When the clock constraint is 50 or 100 MHz, NGT which does not employ goto conversion has the best results. This is mainly because of gate-level retiming during logic synthesis. With goto conversion, gate-level retiming easily increases registers, which results in an increase in the circuit area.

4.3 Discussion

Through the case study, we have found the following observations.

- Reducing the number of components does not always lead to area reduction because of the increased multiplexers and registers.
- Goto conversion is useful in order to reduce the area.
- However, goto conversion does not always lead to area reduction because of the increased registers through gate-level retiming.

We have seen so far that the quality of designs obtained by behavioral synthesis is affected by a number of factors such as clock constraint, resource constraint, optimization options and so on. Therefore, it is not easy but very important to establish a systematic methodology for behavioral synthesis.

5 Conclusions

In this paper, we have presented several techniques on behavioral synthesis of double-precision floating-point adders and adder/subtracters. We have generated totally 15 adders and 21 adder/subtracters with several techniques such as goto conversion.

The future works are considered in two directions. One is to develop other arithmetic IPs for double-precision floating-point computations. The other is to establish a systematic methodology for behavioral synthesis of complex arithmetic circuits.

Reference

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[2] K. Wakabayashi and T. Okamoto, "C-based SoC Design Flow and EDA Tools: An ASIC and System Bendor Perspective," *IEEE Trans. CAD*, vol. 19, no. 12, Dec. 2000.

[3] K. Wakabayashi, "CyberWorkBench: Integrated Design Environment Based on C-based Behavior Synthesis and Verification," *Int. Symp. VLSI Design, Automation and Test*, 2005.

[4] Y Explorations, Inc., http://www.yxi.com/.

[5] Synplicity Inc., http://www.synplicity.com/.

[6] Xilinx Inc., http://www.xilinx.com/.

[7] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html.