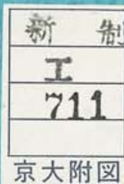
新制
工
711
京大附図

Studies on Hardware Algorithms for Arithmetic Operations with a Redundant Binary Representation

Naofumi TAKAGI

Department of Information Science
Faculty of Engineering
Kyoto University

August 1987

Studies on Hardware Algorithms for Arithmetic Operations with a Redundant Binary Representation

Naofumi TAKAGI

Department of Information Science
Faculty of Engineering
Kyoto University

August 1987

Studies on Hardware Algorithms for Arithmetic Operations with a Redundant Binary Representation

Naofumi Takagi

Abstract

Arithmetic has played important roles in human civilization, especially in the area of science, engineering and technology. With recent advances of IC (Integrated Circuit) technology, more and more sophisticated arithmetic processors have become standard hardware for high-performance digital computing systems. It is desired to develop high-speed multipliers, dividers and other specialized arithmetic circuits suitable for VLSI (Very Large Scale Integrated circuit) implementation. In order to develop such high-performance arithmetic circuits, it is important to design hardware algorithms for these operations, i.e., algorithms suitable for hardware implementation. The design of hardware algorithms for arithmetic operations has become a very attractive research subject.

In this thesis, new hardware algorithms for multiplication, division, square root extraction and computations of several elementary functions are proposed. In these algorithms a redundant binary representation which has radix 2 and a digit set $\{\bar{1},0,1\}$ is used for internal computation. In the redundant binary number system, addition can be performed in a constant time

independent of the word length of the operands. The hardware algorithms proposed in this thesis achieve high-speed computation by using this property. Since many digital systems are required to operate fast and with high reliability, not only high-speed operations and regular structures but also fault-tolerant features should be implemented in arithmetic circuits. A new design method of self-checking arithmetic circuits based on these algorithms is also proposed. Moreover, redundant coding schemes for several algebraic systems and the computational complexity of operations in the systems are also considered to give a theoretical foundation for the design of arithmetic hardware algorithms.

In Chapter 1, the backgrounds, objectives, motivations and the outline of this thesis are described.

Chapter 2 appears as an introductory chapter. The binary number system and the computation model based on which the proposed algorithms are analyzed are explained and the redundant binary representation is discussed.

From Chapter 3 through Chapter 6, new hardware algorithms for arithmetic operations including several elementary functions are proposed.

In Chapter 3, a new multiplication hardware algorithm with a redundant binary addition tree is proposed. A multiplier based on the algorithm can perform multiplication in a computation time proportional to the logarithm of the word length of the operands and has a regular cellular array structure suitable for VLSI implementation. The computation of other arithmetic operations by

means of the multiplier is also considered. The multiplier can effectively be used for computation of other arithmetic operations.

In Chapter 4, a subtract-and-shift division hardware algorithm with the redundant binary representation is proposed. A divider based on the algorithm can perform division in a computation time proportional to the word length of the operands and has a regular cellular array structure suitable for VLSI implementation.

In Chapter 5, a new subtract-and-shift square root hardware algorithm with the redundant binary representation is proposed. A square root circuit based on the algorithm can perform high-speed square root extraction and has a regular cellular array structure.

In Chapter 6, hardware algorithms for computing trigonometric, an inverse trigonometric, logarithmic and exponential functions are proposed. They are based on the CORDIC (COordinate Rotation DIgital Computer) or the STL (Sequential Table Look-up) method which are wellknown methods for computing elementary functions. The computation speed of the CORDIC and the STL method is improved by the use of the redundant binary representation for the internal computation.

In Chapter 7, a new design method for self-checking arithmetic circuits based on the proposed algorithms is shown. A logic design technique called the three-rail logic is used in the method. Arithmetic circuits based on the proposed algorithms and designed by means of the three-rail logic can perform arithmetic

operations fast, have a regular cellular array structure, and further, have a self-checking feature.

In Chapter 8, redundant coding schemes for several algebraic systems, such as a residue class, a finite Abelian group and a residue ring, and the computational complexity of operations in the systems are considered. The proposed redundant coding scheme for a residue class and the hardware algorithm for modular addition by means of the coding scheme are very useful in practice.

With the increase advances of IC technology, it will become possible to implement a special-purpose circuit solving a certain problem quickly. In the development of such a circuit, the design of a good hardware algorithm is one of the key points. Designing a hardware algorithm, we have to consider that a circuit based on it can perform high-speed computation, has a regular structure suitable for VLSI implementation and has a fault-tolerant feature. In order to design good hardware algorithms, a suitable data representation and/or structure should be employed. Especially, the use of a suitable number representation is crucial in the design of an arithmetic hardware algorithm.

Table of Contents

Abstract i

Table of Contents v

Chapter 1 Introduction

1.1 Backgrounds 1

1.2 Outline of the Thesis 4

Chapter 2 Preliminaries

2.1 Binary Number System 8

2.2 Computation Model 8

2.3 Redundant Binary Representation

2.3.1 Redundant Binary Representation 10

2.3.2 Carry-Propagation-Free Addition 11

2.3.3 Redundant Binary to Binary Conversion 15

2.3.4 Special Redundant Binary Numbers 16

Chapter 3 A Multiplication Hardware Algorithm with a Redundant Binary Addition Tree

3.1 Introduction 19

3.2 A Multiplication Hardware Algorithm

3.2.1 Algorithm 21

3.2.2 Analysis of the Algorithm 25

3.3 A Multiplier Based on the Algorithm

3.3.1 Multiplier Recoding and Partial Product Generation 27

3.3.2 A Multiplier Based on the Algorithm 30

3.3.3 The Depth and the Gate Count of the Multiplier 33

3.4 Computation of Other Arithmetic Functions Using the Multiplier

3.4.1 Redundant Binary Multiplier Recoding 35
3.4.2 Computation of Other Arithmetic Functions 36
3.5 Remarks and Discussions 39
Chapter 4 A Subtract-and-Shift Division Hardware Algorithm
4.1 Introduction 42
4.2 A Division Hardware Algorithm
4.2.1 Algorithm 44
4.2.2 Analysis of the Algorithm 47
4.3 A Divider Based on the Algorithm 51
4.4 Remarks and Discussions 56
4.A A Proof of the Correctness of the Algorithm 57
Chapter 5 A Subtract-and-Shift Square Root Hardware Algorithm
5.1 Introduction 60
5.2 A Square Root Hardware Algorithm
5.2.1 Algorithm 62
5.2.2 Analysis of the Algorithm 67
5.3 A Square Root Circuit Based on the Algorithm 69
5.4 Remarks and Discussions 70
5.A A Proof of the Correctness of the Algorithm 71
Chapter 6 Hardware Algorithms for Elementary Functions
6.1 Introduction 75
6.2 Hardware Algorithms Based on the CORDIC Method
6.2.1 Principle of the CORDIC Method 77
6.2.2 A Hardware Algorithm for Computing Sines and Cosines 80
6.2.3 A Hardware Algorithm for Computing Arctangents 87
6.3 Hardware Algorithms Based on the STL Method
6.3.1 Principle of the STL Method 91

6.3.2 A Hardware Algorithm for Computing Logarithms 92

6.3.3 A Hardware Algorithm for Computing Exponentials 97

6.4 Remarks and Discussions 101

Chapter 7 Design of Self-Checking Arithmetic Circuits by Means of the Three-Rail Logic

7.1 Introduction 103

7.2 Design of Self-Checking Arithmetic Circuits 106

7.3 A Design of a Self-Checking Multiplier 112

7.4 Remarks and Discussions 119

Chapter 8 Redundant Coding Schemes for Several Algebraic Systems

8.1 Introduction 121

8.2 Coding Schemes and Local Computability 122

8.3 A Redundant Coding Scheme for a Residue Class and a Hardware Algorithm for Modular Addition 124

8.4 Redundant Coding Schemes for Other Algebraic Systems 129

8.5 Remarks and Discussions 132

Chapter 9 Conclusion 135

Acknowledgments 138

References 139

List of Publications by the Author 145

Chapter 1

Introduction

1.1 Backgrounds

Arithmetic has played important roles in human civilization, especially in the area of science, engineering and technology. Arithmetic operations, such as addition, subtraction, multiplication, division, and so on, are the most fundamental operations in computers, process controllers, digital signal processors and various other digital computing systems. Different algorithms have been proposed for arithmetic operations and some of them have been implemented and utilized in practical systems. (See, e.g., [REIT60], [HWAN79], [CAVA84] and [SCOT85].)

Until one or two dozen years ago, in most of the digital computing systems, only addition and subtraction were implemented by hardware. Multiplication and division, as well as other elementary functions, were implemented by firmware or software using adder/subtracters and shifters. Add-and-shift multiplication algorithms and subtract-and-shift division algorithms have been widely used. They are still used in digital computing systems where a lower computation speed is tolerable.

With recent advances of IC (Integrated Circuit) technology, more and more sophisticated arithmetic processors have become standard hardware for high-performance digital computing systems.

Multiplication is now implemented by hardware in most modern computers. In such a computer, division, square root extraction and other arithmetic computations are performed using multiplication with addition and subtraction as basic operations. On-chip hardware multipliers are now available, and floating-point arithmetic processor LSI's (Large Scale Integrated circuits), digital signal processor LSI's and other LSI's with a hardware multiplier are fabricated. On-chip hardware dividers are becoming realizable. The continuous advances of IC technology are making the use of specialized arithmetic function generators more attractive. A high-performance digital computing system will be equipped with a number of hardware arithmetic processors which solve special arithmetic functions for general and dedicated applications.

In order to develop high-performance multipliers, dividers and other specialized arithmetic circuits, it is important to design good hardware algorithms for these operations, i.e., algorithms suitable for hardware implementation. Designing hardware algorithms for these operations is a very interesting research subject in the area of computer arithmetic. In order to design good hardware algorithms, we should not merely extend conventional algorithms implemented by firmware or software, but consider realization of high-speed computation by using the parallelism of hardware and achieve a regular cellular array structure suitable for VLSI (Very Large Scale Integrated circuit) implementation.

The utilization of a sophisticated number representation

seems effective to design good arithmetic algorithms. As well as the conventional radix number representations, the signed-digit number representations [AVIZ6109], the residue number representations [GARN5906] [SVOB60], the rational number representations [MATU7511] [HWANC7810] and so on have been proposed. Several arithmetic algorithms were designed by means of these representations. The signed-digit number representations and the residue number representations have mainly been used to realize high-speed computation. The rational number representations have mainly been used to realize high-precision computation systems.

In this thesis, hardware algorithms for arithmetic operations with a redundant binary representation are proposed. The redundant binary representation is one of the signed-digit number representations [AVIZ6109]. It has radix 2 and a digit set $\{\bar{1},0,1\}$. In the redundant binary number system, addition can be performed in a constant time independent of the word length of the operands. The hardware algorithms proposed in this thesis achieve high-speed computation by using this property. Several algorithms for multiplication and division with the signed-digit representation were proposed in the 60's and early 70's [AVIZ6109] [ATKI7008], which were intended to be realized by firmware or software because hardware was rather expensive at that time. The algorithms proposed in this thesis are intended to be realized as combinational circuits, and high-speed computation by using the parallelism of hardware and regular array structures suitable for hardware implementation are considered. Furthermore, hardware algorithms not only for multiplication and division but

also for several elementary functions, such as square root extraction, trigonometric functions, the logarithmic function and the exponential function are proposed.

Since many digital systems are required to operate fast and with high reliability, not only high-speed operations and regular structures but also fault-tolerant features should be implemented in arithmetic circuits. The design of self-checking, i.e., on-line error-detectable, arithmetic circuits based on the proposed algorithms is also considered in this thesis.

1.2 Outline of the Thesis

In this thesis, new hardware algorithms for arithmetic operations including several elementary functions will be proposed. In these algorithms, the redundant binary representation is used for the internal computation. These hardware algorithms can be almost directly applied to computations in the 'significand' part of the basic format of the IEEE standard for binary floating-point arithmetic [IEEE754], and can be applied, with slight modification, to computations in the mantissa part of many other floating-point number notations. However, this thesis does not delve into the details required by the IEEE standard such as rounding. A design method of self-checking arithmetic circuits based on these algorithms will also be described. Moreover, redundant coding schemes for several algebraic systems and the computational complexity of operations in the systems are

also considered to give a theoretical foundation for the design of arithmetic hardware algorithms.

Chapter 2 will appear as an introductory chapter. The binary number system and the computation model based on which the proposed algorithms are analyzed will be explained first. Then the redundant binary representation utilized in this thesis will be discussed.

From Chapter 3 through Chapter 6, hardware algorithms for arithmetic operations including several elementary functions with the redundant binary representation will be proposed.

In Chapter 3, a multiplication hardware algorithm with a redundant binary addition tree will be proposed. A multiplier based on the algorithm can perform multiplication in a computation time proportional to the logarithm of the word length of the operands and has a regular cellular array structure suitable for VLSI implementation. The computation of other arithmetic operations using the multiplier will be also considered.

In Chapter 4, a subtract-and-shift division hardware algorithm with the redundant binary representation will be proposed. A divider based on the algorithm can perform division in a computation time proportional to the word length of the operands, and has a regular cellular array structure suitable for VLSI implementation.

In Chapter 5, a subtract-and-shift square root hardware algorithm with the redundant binary representation will be proposed. Square root extraction is one of the most important elementary arithmetic functions and is used in various computa-

tions. A square root circuit based on the algorithm can perform high-speed square root extraction, and has a regular cellular array structure.

In Chapter 6, hardware algorithms for other elementary functions with the redundant binary representation will be proposed. The computations of sines and cosines, arctangents, logarithms, and exponentials will be considered. The proposed algorithms are based on the CORDIC (COordinate Rotation DIgital Computer) or the STL (Sequential Table Look-up) method which are wellknown methods for computing elementary functions. The computation speed of the CORDIC and the STL method is improved by the use of the redundant binary representation for the internal computation.

In Chapter 7, the design of self-checking arithmetic circuits based on the proposed algorithms by means of the three-rail logic will be discussed. The three-rail logic is a logic design technique in which three mutually exclusive conditions calculated in a circuit are encoded in the 1-out-of-3 code and the circuit is designed to be inverter-free. Arithmetic circuits based on the proposed algorithms and designed by means of the three-rail logic can perform arithmetic operations fast, have a regular cellular array structure, and further, have a self-checking feature.

In Chapter 8, redundant coding schemes for several algebraic systems enabling high-speed computation of the operations will be proposed and the computational complexity of the operations will be considered. The discussions will not only give a theoretical

foundation for the design of arithmetic hardware algorithms, but also produce a useful coding scheme for a residue class and an efficient hardware algorithm for modular addition.

Chapter 9 will appear as a conclusion.

Chapter 2

Preliminaries

2.1 Binary Number System

The fixed radix number systems are the most common number systems for internal use in digital computing systems. Especially, the binary number system with radix 2 and a digit set {0,1} is the most conventional and easily realizable number system.

In this thesis, arithmetic on unsigned binary numbers is considered, except in Chapter 8 where arithmetic on several algebraic systems will be considered. An n-bit unsigned binary number with a 1-bit integer part and an (n-1)-bit fraction part $X=[x_0.x_1\cdots x_{n-1}]_2$ $(x_i\in\{0,1\})$ has the value $\sum_{i=0}^{n-1} x_i\cdot 2^{-i}$. n is called the word length of the number. When x_0 is guaranteed to be 1 and hence $1\leq X<2$, X is said normalized.

2.2 Computation Model

In this thesis, combinational circuit implementation of arithmetic circuits is considered, and a combinational circuit composed of fun-in restricted computation elements (logic gates) is adopted as the computation model. A combinational circuit is a

logic circuit which is composed of computation elements of given types and has no feed-back loop in it. The fan-in (in-degree) of each computation element is restricted in a certain constant. The fan-out (out-degree) of it is not restricted. For simplicity in evaluation, it is assumed that all computation elements have the same delay and wires each of which connects computation elements have no delay. On these assumptions, the computation time on a combinational circuit is linearly proportional to the depth of the circuit. The depth of a combinational circuit is equal to the number of computation elements on the longest (directed) path from inputs to outputs in it [SAVA76]. The complexity of a combinational circuit is evaluated by its depth and its gate count. The gate count (the size) of a combinational circuit is the number of computation elements in it.

The area that an arithmetic circuit occupies on a VLSI chip is also considered. The layout rules of a circuit on a VLSI chip are as follows [YASUY8208] [BRENK8107].

(1) Each computation element occupies at least certain constant area and each wire has at least certain constant width.

(2) No computation element overlaps other computation elements or wires.

(3) At most certain constant number of wires can overlap (intersect) each other at any point on a chip.

The area of a circuit is defined by the area of the minimum rectangular region on a plane which includes the layout of the circuit.

2.3 Redundant Binary Representation

2.3.1 Redundant Binary Representation

The redundant binary representation utilized in this thesis is one of the signed-digit (SD) number representations [AVIZ6109]. It has a fixed radix 2 and a digit set $\{\bar{1},0,1\}$, where $\bar{1}$ denotes -1. An n-digit redundant binary number $A=[a_0.a_1\cdots a_{n-1}]_{SD2}$ $(a_i\in\{\bar{1},0,1\})$ has the value $\|A\|=\sum_{i=0}^{n-1} a_i\cdot 2^{-i}$. It is similar to an unsigned binary number except that a_i can be $\bar{1}$. (In Chapter 8, another notation will be used.)

The redundant binary representation allows the existence of redundancy. There are several ways to represent a number. For example, $[0.101]_{SD2}$, $[0.11\bar{1}]_{SD2}$, $[1.\bar{1}01]_{SD2}$, $[1.\bar{1}1\bar{1}]_{SD2}$ and $[1.0\bar{1}\bar{1}]_{SD2}$ all represent '0.625'. (However, '0' is uniquely represented.) Owing to the redundancy, parallel addition of two redundant binary numbers can be performed in a constant time independent of the word length of the operands, as will be mentioned in the next subsection.

One of the redundant binary numbers which have the value $-\|A\|$, where $\|A\|$ is the value of the redundant binary number $A=[a_0.a_1\cdots a_{n-1}]_{SD2}$ $(a_i\in\{\bar{1},0,1\})$, is directly derived by changing the signs of all nonzero a_i's. Namely, $\bar{A}=[\bar{a}_0.\bar{a}_1\cdots \bar{a}_{n-1}]_{SD2}$, where $\bar{a}_i$ is 1 or 0 or $\bar{1}$ accordingly as a_i is $\bar{1}$ or 0 or 1, has the value $-\|A\|$. Since this computation can be performed individually in each position, a negation of a redundant binary number can be

obtained in a constant computation time independent of the word length of the number.

2.3.2 Carry-Propagation-Free Addition

In the ordinary binary number system, parallel addition of two numbers by means of a combinational circuit requires a computation time at least proportional to the logarithm of the word length of the operands because of carry propagation. However, in the redundant binary number system, since carry propagation chains can be eliminated, parallel addition of two numbers can be performed in a constant time independent of the word length of the operands.

Let us consider addition of two n-digit redundant binary numbers $A=[a_0.a_1 \cdots a_{n-1}]_{SD2}$ ($a_i \in \{\bar{1},0,1\}$) and $B=[b_0.b_1 \cdots b_{n-1}]_{SD2}$ ($b_i \in \{\bar{1},0,1\}$). Carry-propagation-free addition is performed in two steps. In the first step (Step 1), the intermediate carry c_i ($\in \{\bar{1},0,1\}$) and the intermediate sum digit d_i ($\in \{\bar{1},0,1\}$) are determined at each position, with satisfying the equation $a_i+b_i=2c_i+d_i$, so that both d_i and c_{i+1} are not 1's nor they are $\bar{1}$'s. In the second step (Step 2), the sum digit s_i ($\in \{\bar{1},0,1\}$) is obtained at each position by the addition of d_i (the intermediate sum digit) and c_{i+1} (the intermediate carry from the next-lower-order position) without generating a new carry.

In Step 1, when one of a_i and b_i is 1 and the other is 0, c_i and d_i are determined as follows. (Note that both $[01]_{SD2}$ and $[1\bar{1}]_{SD2}$ represent '1'.)

(1) If there is a possibility of a 1-carry (a positive carry) from the next-lower-order position, $[c_i, d_i]$ is assigned $[1, \bar{1}]$.

(2) If there is a possibility of a $\bar{1}$-carry (a negative carry) from the next-lower-order position, $[c_i, d_i]$ is assigned $[0, 1]$.

(3) If there is no possibility of a carry from the next-lower-order position, $[c_i, d_i]$ is assigned either $[1, \bar{1}]$ or $[0, 1]$.

Similarly, when one of a_i and b_i is $\bar{1}$ and the other is 0, $[c_i, d_i]$ is assigned $[0, \bar{1}]$ if there is a possibility of a 1-carry from the next-lower-order position, and assigned $[\bar{1}, 1]$ if there is a possibility of a $\bar{1}$-carry. The possibility of a carry from the next-lower-order position can be seen from the augend and the addend digit, a_{i+1} and b_{i+1}, at the next-lower-order position. When both a_{i+1} and b_{i+1} are 1's or one of them is 1 and the other is 0, there is a possibility of a 1-carry. When both of them are $\bar{1}$'s or one of them is $\bar{1}$ and the other is 0, there is a possibility of a $\bar{1}$-carry. In the other cases, there is no possibility of a carry. Thus, c_i and d_i can be determined by examining a_i, b_i, a_{i+1} and b_{i+1}.

When c_i and d_i are determined as stated above, no carry generates in the addition of d_i and c_{i+1} in Step 2. Thus, each sum digit s_i can be computed from a_i, b_i, a_{i+1}, b_{i+1}, a_{i+2} and b_{i+2}. Namely, s_i depends on only these 6 digits. This fact is the key to the high-speed computation.

Table 2-1 shows a computation rule for carry-propagation-free addition. In Step 1, at each position, when one of a_i and b_i is 1 and the other is 0, $[c_i, d_i]$ is assigned $[1,\bar{1}]$ or $[0,1]$ accordingly, as both a_{i+1} and b_{i+1} are nonnegative or not. Similarly, when one of a_i and b_i is $\bar{1}$ and the other is 0, $[c_i, d_i]$ is assigned $[0,\bar{1}]$ or $[\bar{1},1]$ accordingly, as both a_{i+1} and b_{i+1} are nonnegative or not. a_n and b_n, i.e., the augend and the addend digit at the next-lower-order position of the least significant position, are assumed to be 0's. Fig. 2-1 shows an example of carry-propagation-free addition in accordance with the rule. (Take notice of the computation at the second and the third least significant position.)

Table 2-1 A computation rule for carry-propagation-free addition

(a) Step 1

c_i, d_i

a_i \ b_i	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}$,0	0,$\bar{1}$/$\bar{1}$,1*	0,0
0	0,$\bar{1}$/$\bar{1}$,1*	1,$\bar{1}$	1,$\bar{1}$/0,1*
1	1,$\bar{1}$	1,$\bar{1}$/0,1*	1,0

(b) Step 2

s_i

d_i \ c_{i+1}	$\bar{1}$	0	1
$\bar{1}$	---	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	---

* : Both a_{i+1} and b_{i+1} are nonnegative. / Otherwise.

a_i : augend digit, c_i : intermediate carry, s_i : sum digit

b_i : addend digit, d_i : intermediate sum digit

Thus, in the redundant binary number system, carry propagation chains can be eliminated from addition, and therefore, parallel addition of two numbers by means of a combinational circuit is performed in a constant time independent of the word length of the numbers. Namely, the depth of an n-digit redundant binary adder is a constant independent of n. The gate count of it is proportional to n.

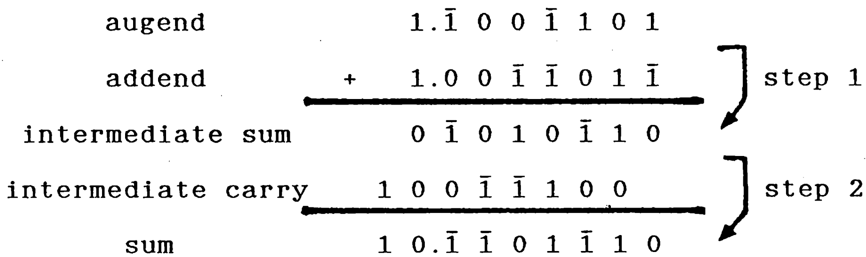

Fig. 2-1 An example of carry-propagation-free addition

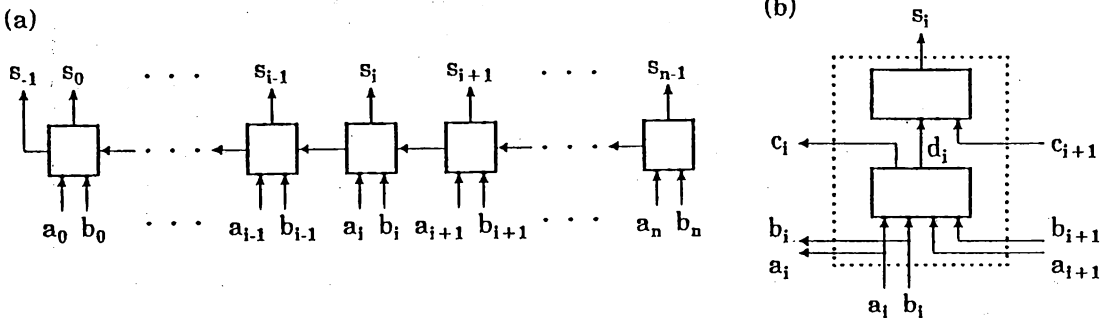

Fig. 2-2 A block diagram of a redundant binary adder

(a) A block diagram of a redundant binary adder

(b) A block diagram of a redundant binary addition cell

Fig. 2-2 (a) shows a block diagram of a redundant binary adder, i.e., a carry-propagation-free adder. □ denotes a redundant binary addition cell, which consists of two modules as shown in Fig. 2-2 (b). The structure of the adder is just like that of an ordinary ripple-carry adder.

The addition method described above is an example. There are various other carry-propagation-free addition methods in the redundant binary number system [CHOWR7810].

Subtraction of two redundant binary numbers is done by adding the minuend and the negation of the subtrahend. As stated in the previous subsection, a negation of a redundant binary number is derived directly by changing the sign of all nonzero digits in the number. Therefore, parallel subtraction can also be performed in a constant time independent of the word length of the operands.

2.3.3 Redundant Binary to Binary Conversion

An unsigned binary number is itself regarded as a redundant binary number whose each digit is nonnegative, i.e., either 0 or 1. Therefore, no computation is required to convert an unsigned binary number into one of the equivalent redundant binary numbers, where the equivalence implies that they have the same value.

An n-digit redundant binary number $A=[a_0.a_1 \cdots a_{n-1}]_{SD2}$ ($a_i \in \{\bar{1},0,1\}$) is converted into the equivalent binary number by subtracting A^- from A^+, where A^+ and A^- are n-bit unsigned binary numbers formed from the positive digits and the negative digits

in A, respectively. For example, $A=[1.\bar{1}01]_{SD2}$ is converted to $[0.101]_2$ by computing $[1.001]_2-[0.100]_2$. In general, since a redundant binary number can be either positive or negative, the sign of the binary number must be considered. However, in the algorithms proposed later, since the results are guaranteed to be positive, the sign of the number need not be considered. This conversion can be performed in a computation time proportional to $\log n$ or n accordingly, as using a carry-look-ahead adder (CLA) or a ripple-carry adder (RCA). The gate count of either adder is proportional to n [UNGE7704].

2.3.4 Special Redundant Binary Numbers

As stated in the previous subsection, an unsigned binary number can be regarded as a special redundant binary number with nonnegative digits.

Assume that a redundant binary number $A=[a_0.a_1\cdots a_{n-1}]_{SD2}$ is a special redundant binary number with nonnegative digits, i.e., each a_i is either 0 or 1. All the digits of the redundant binary number $\bar{A}=[\bar{a}_0.\bar{a}_1\cdots\bar{a}_{n-1}]_{SD2}$ which has the value $-\|A\|$ (see Subsection 2.3.1) are nonpositive. The redundant binary number $A=[\bar{1}\tilde{a}_0.\tilde{a}_1\cdots\tilde{a}_{n-1}]_{SD2}$, where $\tilde{a}_i$ is 1 or 0 accordingly as a_i is 0 or 1, has the value $-\|A\|-2^{-n+1}$. All the digits of A except the most significant one are nonnegative. Similarly, the redundant binary number $\bar{A}=[1\bar{\tilde{a}}_0.\bar{\tilde{a}}_1\cdots\bar{\tilde{a}}_{n-1}]_{SD2}$, where $\bar{\tilde{a}}_i$ is $\bar{1}$ or 0 accordingly as a_i is 0 or 1, has the value $\|A\|+2^{-n+1}$. All the digits of $\bar{A}$ except the most significant one are nonpositive. These facts

will be utilized to reduce the computation time and the amount of hardware of arithmetic circuits based on the algorithms proposed in the following chapters.

In addition of redundant binary numbers A and B, if the addend B is a special redundant binary number with nonnegative digits, the addition rule is simpler than that in the general case shown in Table 2-1 in Subsection 2.3.2. The intermediate carry c_i $(\in\{0,1\})$ and the intermediate sum digit d_i $(\in\{\bar{1},0\})$ can be determined by examining only the augend digit a_i $(\in\{\bar{1},0,1\})$ and the addend digit b_i $(\in\{0,1\})$ at each position, in the first step. Since d_i is either $\bar{1}$ or 0 and c_{i+1} is either 0 or 1, no new carry generates in the second step. Table 2-2 shows the addition rule for this case. Similarly, if the addend B is a special redundant binary number with nonpositive digits, the addition rule is also simpler than that in the general case. c_i $(\in\{\bar{1},0\})$ and d_i $(\in\{0,1\})$ can be determined by examining only a_i and b_i $(\in\{\bar{1},0\})$, in the first step. Table 2-3 shows the addition rule for this case. (The same rule can be used with exchanging a_i and b_i, when not B but the augend A is a special redundant binary number.) Furthermore, if the augend A is a special redundant binary number with nonnegative digits and the addend B is a special one with nonpositive digits (or vice versa), the addition rule is much simpler. The sum digit s_i can directly be obtained by calculating a_i+b_i without a carry. Table 2-4 shows the addition rule for this case.

Table 2-2 A computation rule for carry-propagation-free addition (When the addend digits are nonnegative.)

(a) Step 1

c_i , d_i

a_i \ b_i	0	1
$\bar{1}$	0, $\bar{1}$	0,0
0	0,0	1, $\bar{1}$
1	1, $\bar{1}$	1,0

(b) Step 2

s_i

d_i \ c_{i+1}	0	1
$\bar{1}$	$\bar{1}$	0
0	0	1

Table 2-3 A computation rule for carry-propagation-free addition (When the addend digits are nonpositive.)

(a) Step 1

c_i , d_i

a_i \ b_i	$\bar{1}$	0
$\bar{1}$	$\bar{1}$,0	$\bar{1}$,1
0	$\bar{1}$,1	0,0
1	0,0	0,1

(b) Step 2

s_i

d_i \ c_{i+1}	$\bar{1}$	0
0	$\bar{1}$	0
1	0	1

Table 2-4 A computation rule for carry-propagation-free addition (When the augend digits are nonnegative and the addend digits are nonpositive.)

s_i

a_i \ b_i	$\bar{1}$	0
0	$\bar{1}$	0
1	0	1

Chapter 3

A Multiplication Hardware Algorithm with a Redundant Binary Addition Tree

3.1 Introduction

Multiplication plays very important roles in various digital systems. Designing fast multipliers has long been a great theoretical and practical interest for computer scientists and engineers. Various multiplication algorithms have been proposed and some of them are practically used. Especially, with recent advances of IC technologies, many researchers have tried to develop high-speed multiplication algorithms which are suitable for VLSI implementation.

For high-speed multiplication by means of a combinational circuit, array multipliers [BRAU63] [AGRA7903] [HWAN7904] and multipliers with parallel counters [WALL6402] [STENK7710] have been proposed.

Array multipliers, i.e., parallel multipliers based on the add-and-shift method, have been widely used and some of them are implemented on commercial LSI chips. This type of multiplier has a regular cellular array structure of one type basic cells and is very suitable for LSI implementation. However, it does not operate so fast for longer operands, because its computation time is linearly proportional to the word length of the operands.

Multiplication algorithms with parallel counters, such as Wallace tree, have been proposed in order to realize high-speed multiplication. A multiplier based on this type of algorithm operates much faster than an array one for longer operands, because its computation time is proportional to the logarithm of the word length of the operands. It is adopted in large scale computers in which very high-speed multiplication is required. Although it is composed of one type of basic cells, its layout on a VLSI chip becomes rather complicated and the area for wires becomes larger [REUSK81] [LUK-V83]. It is not so suited to VLSI implementation.

It has been a challenging problem in recent years to develop a multiplier which can perform multiplication in a computation time proportional to the logarithm of the word length of the operands and has a regular cellular array structure suitable for VLSI implementation. In this chapter, a multiplication hardware algorithm for such a multiplier is proposed [TAKAY8209] [TAKAY8306a] [TAKAY8509]. In the algorithm, the redundant binary representation is used for the internal computation.

In this chapter, multiplication of two n-bit unsigned binary numbers with 1-bit integer part and (n-1)-bit fraction part is considered. The product is a 2n-bit unsigned binary number with 2-bit integer part and (2n-2)-bit fraction part.

In the next section, a multiplication hardware algorithm with a redundant binary addition tree will be proposed. In Section 3.3, a multiplier based on the algorithm will be discussed. In Section 3.4, the computation of other arithmetic

operations using the multiplier will be discussed. Some further discussions will be made in Section 3.5.

3.2 A Multiplication Hardware Algorithm

3.2.1 Algorithm

In this section, a new multiplication hardware algorithm is proposed. In the algorithm, as in the ordinary parallel multiplication, n partial products are first generated. These partial products are binary numbers and are regarded as special redundant binary numbers with nonnegative digits. Then, they are added up pairwise in a binary-tree-form and the product represented in the redundant binary representation is obtained. All intermediate results are represented in the redundant binary representation and all additions are performed in the redundant binary number system. Finally, the product is converted into binary representation.

The algorithm is as follows.

Algorithm [MUL]

<Input>

X and Y : a multiplicand and a multiplier, respectively
(n-bit unsigned binary numbers)

<Output>

Z : the product
(a 2n-bit unsigned binary number)

<Algorithm>

Step 1: for $j=0$ to $n-1$ do in parallel

$P_{0,j}=X \cdot y_j \cdot 2^{-j}$

end

Step 2: for $k=1$ to $m(=\lceil \log_2 n \rceil)$ do

for $j=0$ to $\lceil n/2^k \rceil-1$ do in parallel

$P_{k,j}:=P_{k-1,2j}+P_{k-1,2j+1}$

(redundant binary addition)

end

end

Step 3: $Z \leftarrow P_{m,0}$

(redundant binary to binary conversion) □

In Step 1, n partial products ($P_{0,j}$'s) are generated. A partial product is zero or the multiplicand itself (shifted j positions to the right) accordingly, as the concerned multiplier digit (y_j) is 0 or 1. (y_j is the j-th binary digit of Y.)

In Step 2, the partial products are added up by means of a binary tree of redundant binary adders and the product represented in the redundant binary representation ($P_{m,0}$) is obtained. All intermediate results ($P_{k,j}$'s) are represented in the redundant binary representation, and all additions are performed in the redundant binary number system. All additions at each level in the redundant binary addition tree are performed in parallel.

In Step 3, the product ($P_{m,0}$) is converted into the equivalent unsigned binary number Z.

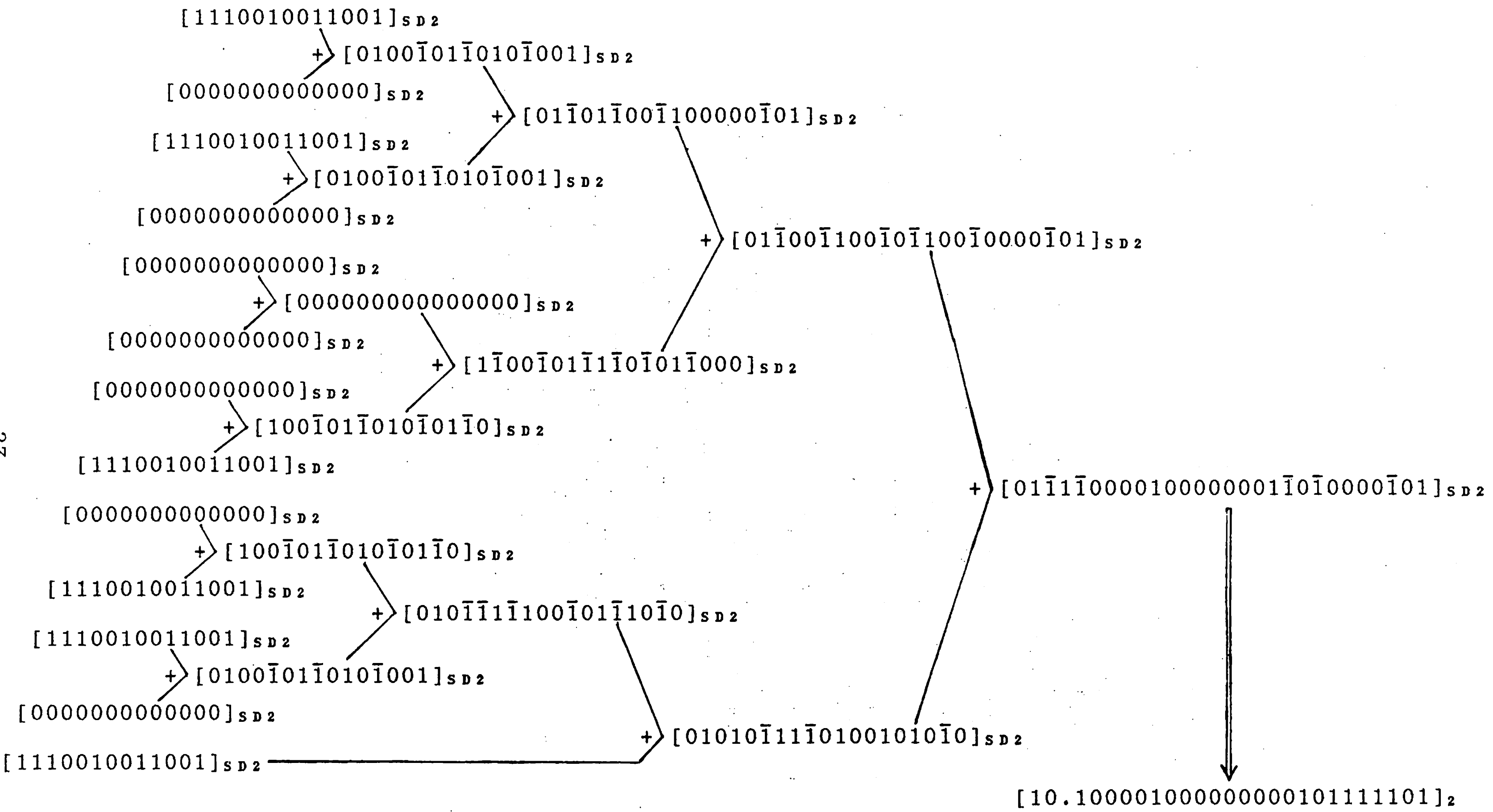

Fig. 3-1 An example of multiplication according to Algorithm [MUL]

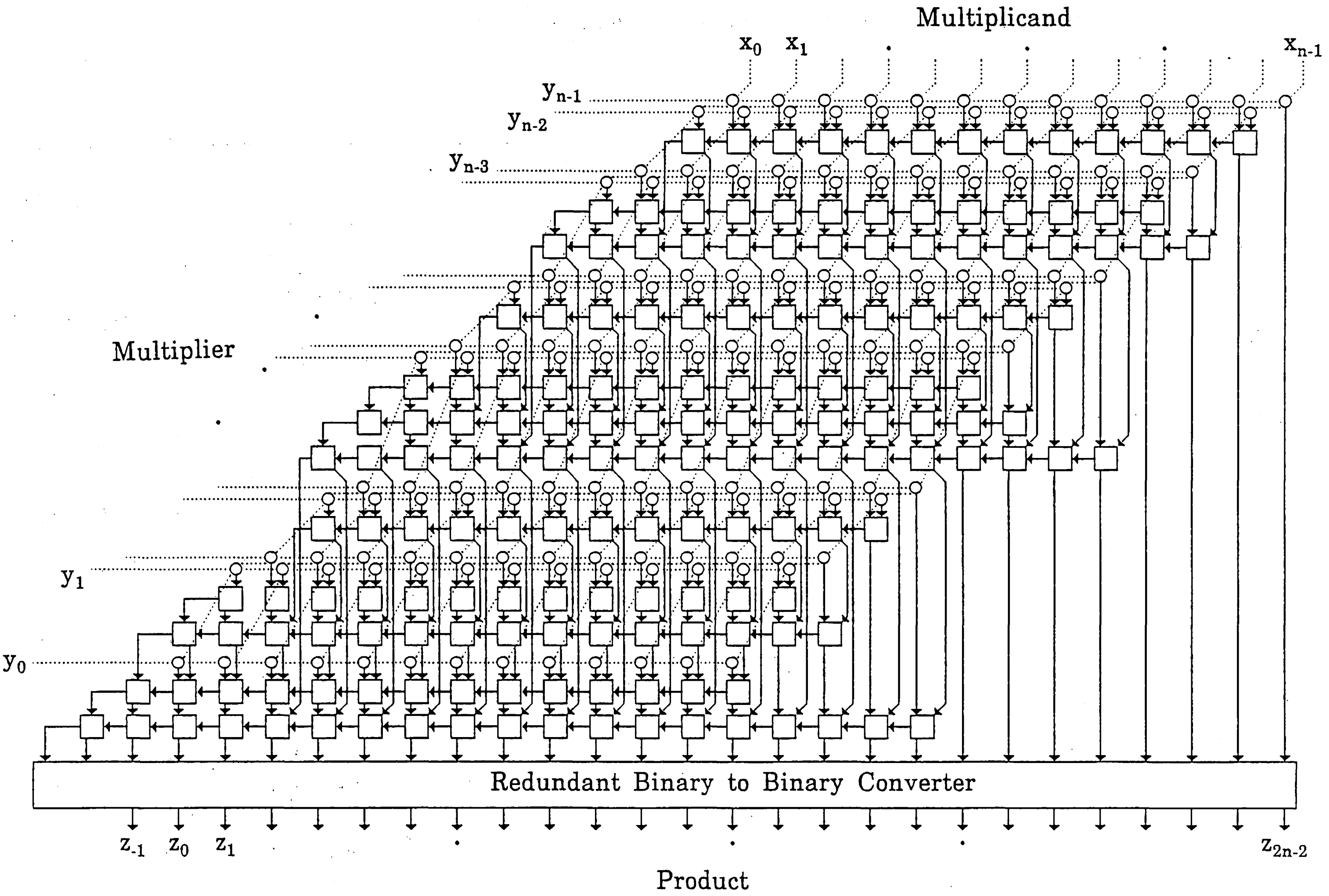

Fig. 3-2 A block diagram of a multiplier based on Algorithm [MUL]

Fig. 3-1 illustrates an example of multiplication according to the algorithm.

Fig. 3-2 shows a block diagram of a multiplier based on the algorithm. ○ denotes a partial product generation cell, and each horizontal row of ○'s forms a partial product generator. □ denotes a redundant binary addition cell, and each horizontal row of □'s forms a redundant binary adder, i.e., a carry-propagation-free adder. The redundant binary to binary converter is a modification of a carry-look-ahead adder and is easily realizable.

3.2.2 Analysis of the Algorithm

The computation in Step 1 is performed in parallel for all digits. It requires a constant computation time independent of n. Since n n-digit partial products are generated in parallel, the required gate count is proportional to n^2.

As discussed in Subsection 2.3.2, parallel addition of two redundant binary numbers can be performed in a constant time. Therefore, the computation at each level in the tree in Step 2 is performed in a constant time independent of n. Since there are $\lceil \log_2 n \rceil$ levels, the computation time for Step 2 is proportional to $\log_2 n$. The required gate count is proportional to n^2, because n-1 redundant binary adders are needed.

The conversion in Step 3 is performed in a computation time proportional to $\log n$ by means of a carry-look-ahead adder, as mentioned in Subsection 2.3.3. The required gate count is proportional to n.

Thus, we conclude that the multiplication algorithm performs

n-bit binary multiplication in a time proportional to *log*n with a gate count proportional to n^2. Namely, the depth of a multiplier based on the algorithm is $O(\log n)$, and the gate count of it is $O(n^2)$. As shown in Fig. 3-2, the multiplier has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2 \log n)$, because at most $O(\log n)$ vertical wires run between adjacent addition cells.

Table 3-1 shows a comparison of a multiplier based on the proposed algorithm, an array multiplier and a multiplier with Wallace tree (a multiplier with parallel counters) regarding the depth, the gate count, the chip area and the complexity of layout. As shown in the table, the depth of the proposed multiplier as well as that of a multiplier with Wallace tree is $O(\log n)$ in contrast to that of an array multiplier which is $O(n)$. The gate counts of the three types of multiplier are all $O(n^2)$. The chip area of the proposed multiplier is $O(n^2 \log n)$. It is the same as that of a multiplier with Wallace tree and larger than that of an array multiplier. However, the proposed multiplier has a regular cellular array structure, and therefore, its layout is simpler than that of a multiplier with Wallace tree.

Table 3-1 A comparison of three types of multiplier

	Depth	Gate Count	Area	Layout
Proposed Multiplier	$O(\log n)$	$O(n^2)$	$O(n^2 \log n)$	rather simple
Array Multiplier	$O(n)$	$O(n^2)$	$O(n^2)$	simple
Multiplier with Wallace Tree	$O(\log n)$	$O(n^2)$	$O(n^2 \log n)$	complicated

3.3 A Multiplier Based on the Algorithm

3.3.1 Multiplier Recoding and Partial Product Generation

2-bit Booth's method [WALL6402] can be effectively applied to a multiplier based on the proposed algorithm for reducing the computation time and the amount of hardware. Using 2-bit Booth's algorithm, the multiplier is recoded into the radix 4 modified signed-digit (SD4) representation with a digit set $\{\bar{2},\bar{1},0,1,2\}$ where $\bar{2}$ denotes -2, and then $\lceil (n-1)/2 \rceil+1$ partial products are generated according to the $\lceil (n-1)/2 \rceil+1$ recoded multiplier digits. Thus the number of partial products is reduced to about the half. In the recoding of the multiplier $Y=[y_0.y_1 \cdots y_{n-1}]_2$ into $\hat{Y}=[\hat{y}_0.\hat{y}_1 \cdots \hat{y}_{\lceil (n-1)/2 \rceil}]_{SD4}$, $\hat{y}_j$ is obtained by calculating $-2y_{2j-1}+y_{2j}+y_{2j+1}$ for $j=0,1,\cdots,\lceil (n-1)/2 \rceil$. (See Table 3-2.) This recoding is based on the fact that 2=4-2. The recoding can be performed in a constant time independent of n. The required gate count is proportional to n.

Table 3-2 Booth's 2-bit multiplier recoding rule

y_{2j-1}	y_{2j} \ y_{2j+1}	$\hat{y}_j$: 0	1
0	0	0	1
0	1	1	2
1	0	$\bar{2}$	$\bar{1}$
1	1	$\bar{1}$	0

Each partial product is generated by calculating $X \cdot \hat{y}_j$, where $X=[x_0 . x_1 \cdots x_{n-1}]_2$ is a multiplicand and $\hat{y}_j$ $(\in \{\bar{2}, \bar{1}, 0, 1, 2\})$ is a recoded multiplier digit. To perform the computation, negating a special redundant binary number and/or doubling a redundant binary number are needed. Twice a redundant binary number can be easily obtained by shifting the number one position to the left. As stated in Subsection 2.3.4, the redundant binary numbers $\bar{X}=[\bar{x}_0 . \bar{x}_1 \cdots \bar{x}_{n-1}]_{SD2}$ ($\bar{x}_i$ is 0 or $\bar{1}$ accordingly as x_i is 0 or 1), $X=[\bar{1}\tilde{x}_0 . \tilde{x}_1 \cdots \tilde{x}_{n-1}]_{SD2}$ ($\tilde{x}_i$ is 1 or 0 accordingly as x_i is 0 or 1) and $\bar{\bar{X}}=[1\bar{\bar{x}}_0 . \bar{\bar{x}}_1 \cdots \bar{\bar{x}}_{n-1}]_{SD2}$ ($\bar{\bar{x}}_i$ is $\bar{1}$ or 0 accordingly as x_i is 0 or 1) have the values $-\|X\|$, $-\|X\|-2^{-n+1}$, and $\|X\|+2^{-n+1}$, respectively, where $\|X\|$ is the value of X. Therefore, we can use either X or $\bar{X}-2^{-n+1}$ to represent $\|X\|$ and either $\bar{X}$ or $X+2^{-n+1}$ to represent $-\|X\|$. A partial product is generated toward $\hat{y}_j$ as shown in Table 3-3. When j is even, X and $X+2^{-n+1}$ are used to represent $\|X\|$ and $-\|X\|$, respectively. When j is odd, $\bar{X}-2^{-n+1}$ and $\bar{X}$ are used to represent them, respectively. (Some modifications may be made to simplify the addition in the first level.) The correction digits are supplied to the addition in the second or later levels of the addition tree. Take notice that all digits of X and X except the most significant digit of X are nonnegative and all digits of $\bar{X}$ and $\bar{X}$ except the most significant digit of $\bar{X}$ are non-positive. In each addition in the first level, almost all augend digits are nonnegative and almost all addend digits are nonpositive (or vice versa), and therefore, the very simple addition rule shown in Table 2-4 in Subsection 2.3.4 can be used for

the addition. The generation of partial products can be performed in a constant time independent of n. The required gate count is proportional to n^2.

Table 3-3 A rule for generating partial products

y_j	j : even	j : odd
$\bar{2}$	$[\bar{1}\ \tilde{x}_0 \tilde{x}_1 \tilde{x}_2 \cdots \tilde{x}_{n-1}\ 1\ ]_{SD2}$ 1	$[0\ \bar{x}_0 \bar{x}_1 \bar{x}_2 \cdots \bar{x}_{n-1}\ 0\ ]_{SD2}$ 0
$\bar{1}$	$[\bar{1}\ 1\ \tilde{x}_0 \tilde{x}_1 \cdots \tilde{x}_{n-2} \tilde{x}_{n-1}]_{SD2}$ 1	$[0\ 0\ \bar{x}_0 \bar{x}_1 \cdots \bar{x}_{n-2} \bar{x}_{n-1}]_{SD2}$ 0
0	$[0\ 0\ 0\ 0 \cdots 0\ 0\ ]_{SD2}$ 0	$[0\ 0\ 0\ 0 \cdots 0\ 0\ ]_{SD2}$ 0
1	$[0\ 0\ x_0 x_1 \cdots x_{n-2} x_{n-1}]_{SD2}$ 0	$[1\ \bar{1}\ \bar{\tilde{x}}_0 \bar{\tilde{x}}_1 \cdots \bar{\tilde{x}}_{n-2} \bar{\tilde{x}}_{n-1}]_{SD2}$ $\bar{1}$
2	$[0\ x_0 x_1 x_2 \cdots x_{n-1}\ 0\ ]_{SD2}$ 0	$[1\ \bar{\tilde{x}}_0 \bar{\tilde{x}}_1 \bar{\tilde{x}}_2 \cdots \bar{\tilde{x}}_{n-1}\ \bar{1}\ ]_{SD2}$ $\bar{1}$

3.3.2 A Multiplier Based on the Algorithm

The multiplier recoding method and the partial product generating method described above improve the computation speed and reduce the amount of hardware of a multiplier based on the proposed algorithm. There is also an excellent technique to reduce the amount of hardware and to increase the regularity of layout, in the additions in the addition tree.

In each addition in the k-th level of the addition tree, the addend is shifted about 2^k positions to the left from the augend. Namely, there are no augend digits at the most significant about 2^k positions and no addend digits at the least significant about 2^k positions. Let us call the positions where no addend digits exist the lower part, the positions where both augend and addend digits exist the middle part, and the positions where no augend digits exist the upper part. In the addition, in the lower part, the sum digits are let be the augend digits themselves, in the middle part, they are let be the sum digits obtained by means of the carry-propagation-free addition mentioned in Subsection 2.3.2, and in the upper part, they are let be the addend digits themselves. The carry from the most significant position of the middle part is saved separately and shall be added later.

Fig. 3-3 shows an example of multiplication based on the algorithm in consideration of the above discussions. Fig. 3-4 shows a block diagram of a multiplier in consideration of the above discussions. The computation speed is improved, the amount of hardware is reduced and the regularity of layout is increased.

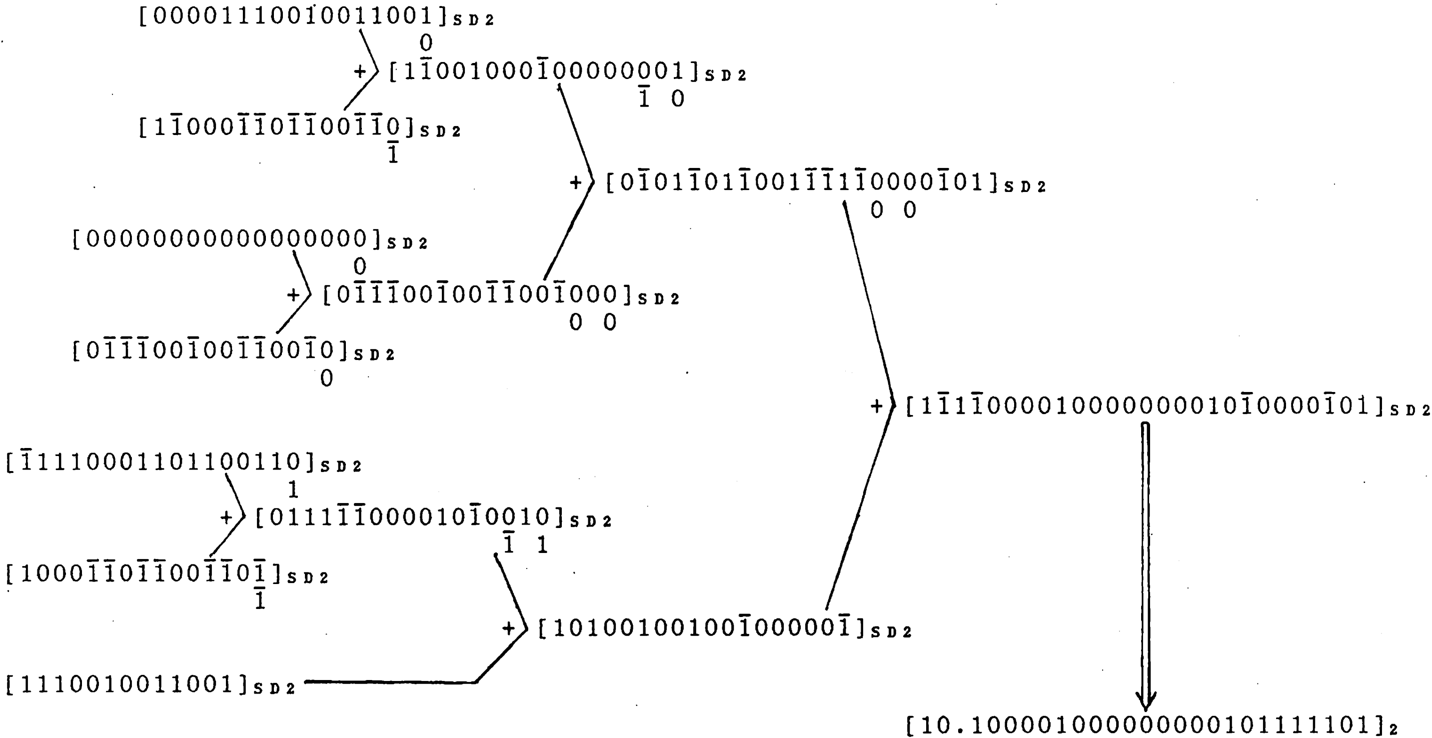

Fig. 3-3 An example of multiplication according to Algorithm [MUL] with multiplier recoding

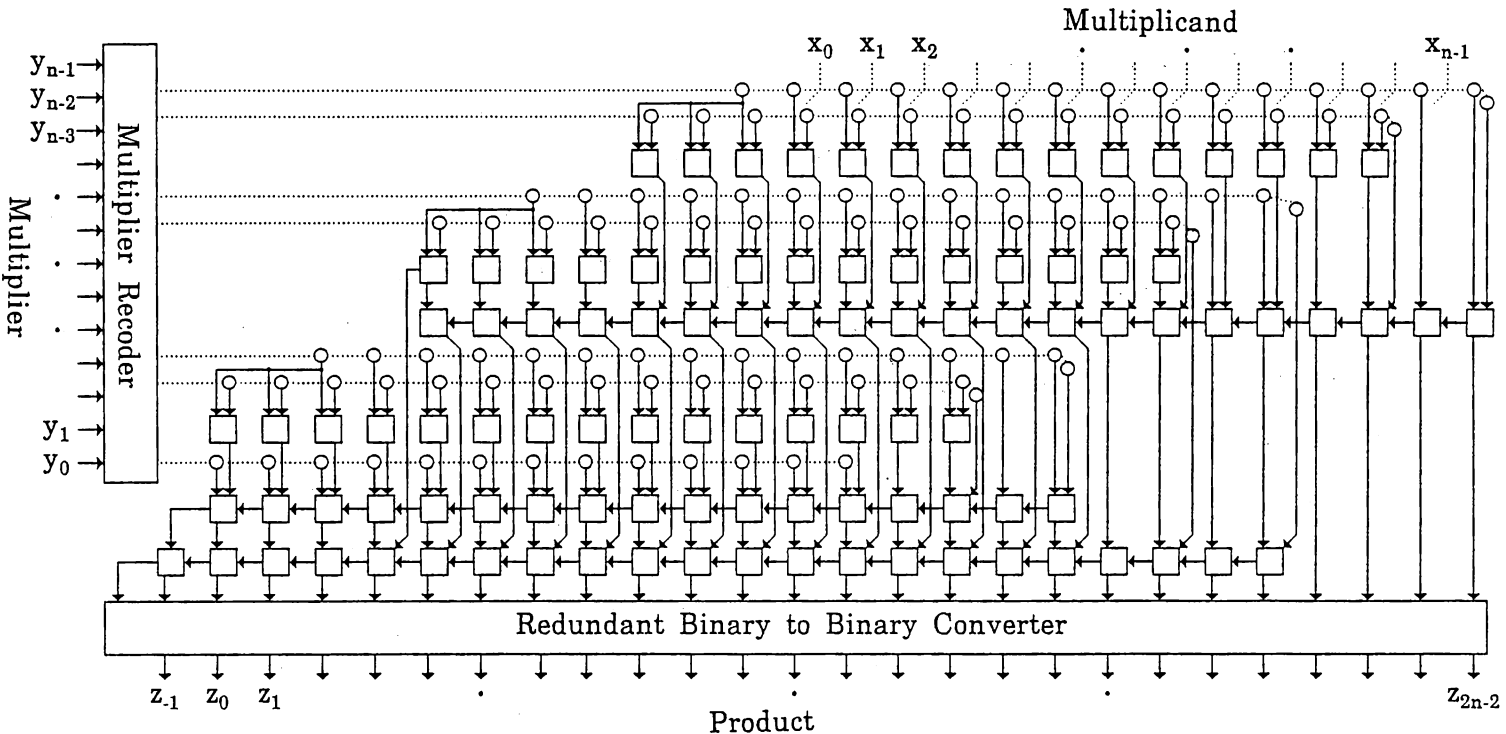

Fig. 3-4 A block diagram of a multiplier based on Algorithm [MUL] with multiplier recoding

3.3.3 The Depth and the Gate Count of the Multiplier

Table 3-4 shows an evaluation of the depth and the gate counts of a multiplier based on the proposed algorithm with the above consideration, an array multiplier and a multiplier with Wallace tree, by the use of CMOS gates (including 2-input EXOR gates) as computation elements. Each of the three multipliers uses 2-bit Booth's method for multiplier recoding and a carry-look-ahead adder for the last addition.

In the logic design of the proposed multiplier, a typical multiplier recoding cell, a typical partial product generation cell, a redundant binary addition cell for the first level of the addition tree and a typical redundant binary addition cell are composed of 5 gates (30 transistors), 2 gates (14 transistors), 3 gates (12 transistors) and 8 gates (42 transistors), respectively. Fig. 3-5 shows a CMOS logic design of the typical redundant binary addition cell. A redundant binary digit a is represented by two bits, a_s and a_d, and 11 or 00 or 01 is assigned to $a_s a_d$ accordingly, as a is $\bar{1}$ or 0 or 1.

Table 3-4 Depth and gate counts of three types of multiplier

depth / gate count (the number of transistors)

	24-bit	53-bit
Proposed Multiplier	26 / 2583 (13674)	32 / 11431 (61654)
Array Multiplier	46 / 2689 (13216)	90 / 11992 (59772)
Multiplier with Wallace Tree	29 / 2907 (14194)	37 / 12878 (63860)

As shown in Table 3-4, the depth of the proposed multiplier is similar to that of a multiplier with Wallace tree. It is smaller than that of an array multiplier, especially when the word length is longer. The 24-bit proposed multiplier is about twice faster than the 24-bit array one and the 53-bit proposed multiplier is about three times faster than the 53-bit array one. The gate count of the three types of multiplier are almost the same. Furthermore, as shown in Fig. 3-4, the proposed multiplier has a regular cellular array structure, and therefore, its layout is simpler than that of a multiplier with Wallace tree in which interconnections of cells are more complicated [YASUY8201] [REUSK81].

Thus, the proposed multiplier is excellent in both computation speed and regularity in layout.

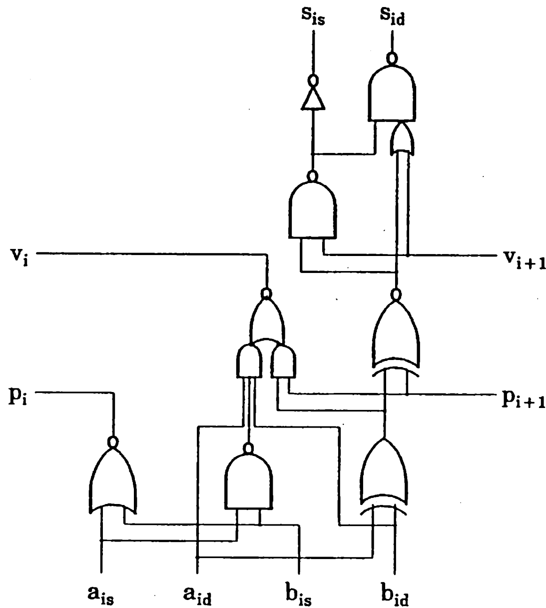

Fig. 3-5 A CMOS logic design of the typical redundant binary addition cell

3.4 Computation of Other Arithmetic Functions Using the Multiplier

3.4.1 Redundant Binary Multiplier Recoding

Various algorithms in which multiplication is used as a basic operation are proposed and practically used for division and the computation of other elementary functions. A multiplier based on the proposed algorithm can effectively be used for implementation of these algorithms. Furthermore, the computation speed for successive multiplications is improved by omitting redundant binary to binary conversion and directly applying an intermediate result represented in the redundant binary representation to the next multiplication.

Table 3-5 A redundant binary multiplier recoding rule

(a) Step 1

y_{2j-1} \ y_{2j} (c_j, d_j)	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}$,1	0,$\bar{2}$/$\bar{1}$,2*	0,$\bar{1}$
0	0,$\bar{1}$	0,0	0,1
1	0,1	1,$\bar{2}$/0,2*	1,$\bar{1}$

(b) Step 2

d_j \ c_{j+1} ($\hat{y}_j$)	$\bar{1}$	0	1
$\bar{2}$	---	$\bar{2}$	$\bar{1}$
$\bar{1}$	$\bar{2}$	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	2
2	1	2	---

* : y_{2j+1} is nonnegative. / Otherwise.

When an intermediate result represented in the redundant binary representation is applied to the next multiplication as a multiplier, there is an excellent technique to recode it to a radix 4 modified signed-digit number. (Recall the Booth's method stated in the previous section.) The recoding procedure consists of two steps. Let us consider to recode a redundant binary number $Y=[y_0 . y_1 \cdots y_{n-1}]_{SD2}$ into the equivalent SD4 number $\hat{Y}=[\hat{y}_0 . \hat{y}_1 \cdots \hat{y}_{\lceil (n-1)/2 \rceil}]_{SD4}$. In the first step, c_j ($\in\{\bar{1},0,1\}$) and d_j ($\in\{\bar{2},\bar{1},0,1,2\}$) are determined for $j=0,1,\cdots,\lceil (n-1)/2 \rceil$, with satisfying the equation $2y_{2j-1}+y_{2j}=4c_j+d_j$, so that $[d_j, c_{j+1}]$ does not become $[2,1]$ nor $[\bar{2},\bar{1}]$. In the second step, $\hat{y}_j$ ($\in\{\bar{2},\bar{1},0,1,2\}$) is obtained for each j by the addition of d_j and c_{j+1}, without generating a carry. Table 3-5 shows a recoding rule. In Step 1, when $2y_{2j-1}+y_{2j}$ is 2, $[c_j, d_j]$ is assigned $[1,\bar{2}]$ or $[0,2]$ accordingly, as y_{2j+1} is nonnegative or not. Similarly, when $2y_{2j-1}+y_{2j}$ is -2, it is assigned $[0,\bar{2}]$ or $[\bar{1},2]$ accordingly, as y_{2j+1} is nonnegative or not. No carry generates in the addition in Step 2. The depth of the recoder based on this technique is a constant independent of n and the gate count of it is proportional to n.

3.4.2 Computation of Other Arithmetic Functions

First, consider implementation of the multiplicative division algorithm based on Newton-Raphson method [HARTC78] using the multiplier. In the multiplicative division algorithm, the reciprocal of the divisor Y is calculated according to the

iteration equation $Q_j := Q_{j-1} \cdot (2 - Q_{j-1} \cdot Y)$, where Q_0 is the first approximation of 1/Y. Let us consider calculation for an iteration step by means of the circuit shown in Fig. 3-6. In the figure, 'MUL' is a multiplier based on the proposed algorithm with the above mentioned recoder, which multiplies a binary number by a redundant binary number and produces a redundant binary number. 'CONV' is a redundant binary to binary converter. 'SUB2' is a very simple redundant binary subtracter, which calculates 2-A for an input redundant binary number A. The register 'REG1' stores Y. 'REG2' stores a binary number and 'REG3' stores a redundant binary number.

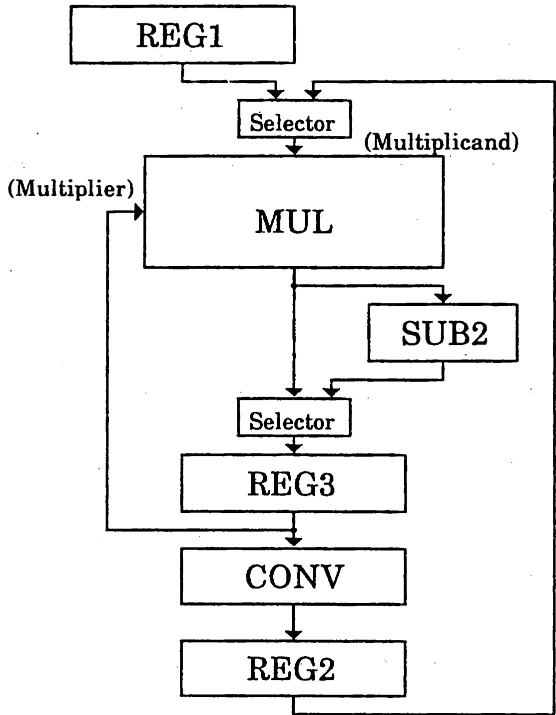

Fig. 3-6 A division circuit with the proposed multiplier

In every iteration step, at the beginning, Q_{j-1} represented in the redundant binary representation is stored in 'REG3'. The calculation for an iteration step is performed in two stages as follows.

Stage 1: Perform the following two operations in parallel.

REG3 ← MUL(<REG1>,<REG3>)

REG2 ← CONV(<REG3>)

Stage 2: REG3 ← SUB2(MUL(<REG2>,<REG3>))

'REG3 ← MUL(<REG1>,<REG3>)' means that multiplying the content of 'REG1' by that of 'REG3' and storing the result in 'REG3'. After Stage 2, Q_j is stored in 'REG3'. By adopting the recoding technique mentioned in the previous subsection, the computation time of each iteration step has been shortened.

The multiplier can also effectively be used for implementation of a multiplicative square root algorithm based on Newton-Raphson method in which the reciprocal of the square root of the radicand X is calculated according to the iteration equation $Q_j := Q_{j-1} \cdot (3 - Q_{j-1}{}^2 \cdot X)/2$.

Mathematically, bounded elementary functions can be written in the form of an infinite power series $\sum_{j=0}^{\infty} A_j \cdot X^j$. These functions are computed by calculating the polynomial (truncated power series) $\sum_{j=0}^{m} A_j \cdot X^j$ which approximates the values of the functions. The multiplier can also be used for calculation of polynomials, effectively. The polynomial $A_m \cdot X^m + A_{m-1} \cdot X^{m-1} + \cdots + A_1 \cdot X + A_0$ can be calculated according to the iteration equation $Q_j = Q_{j-1} \cdot X + A_{m-j}$ where $Q_0 = A_m$. The calculation can be performed by means of a

circuit similar to that shown in Fig. 3-6. ('SUB2' is replaced by a simple redundant binary adder 'ADD', which adds a redundant binary number and a binary number.) The register 'REG1' stores X, in this case. In every iteration step, at the beginning, Q_{j-1} represented in the redundant binary representation is stored in 'REG3'. 'ADD' is fed with A_{m-j} as the addend. The calculation for an iteration is performed as follows.

$$REG3 \leftarrow ADD(MUL(\langle REG1\rangle, \langle REG3\rangle), A_{m-j})$$

After the calculation, Q_j is stored in 'REG3'. 'CONV', 'REG2' and the selectors are not needed for the computation for the iteration step. In general, the truncated series takes the form of $\sum_{j=0}^{k} A_j \cdot X^{s+tj}$. In this case, the polynomial can be calculated in the same way as stated above, with just changing X to X^t in every iteration step.

3.5 Remarks and Discussions

A new multiplication hardware algorithm with a redundant binary addition tree has been proposed. The multiplication algorithm performs n-bit binary multiplication in a time proportional to $\log n$ with a gate count proportional to n^2. Namely, the depth of a multiplier based on the algorithm is $O(\log n)$, and the gate count of it is $O(n^2)$. It has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2 \log n)$. A multiplier recoding method (2-bit Booth's method) and an efficient partial product gener-

ating method for improving the computation speed and reducing the amount of hardware have been described.

As stated in the previous section, a multiplier based on the proposed algorithm can effectively be used for implementation of various arithmetic algorithms.

In Section 3.2 and Section 3.3, the product was computed down to the (2n-2)nd binary digit. However, in multiplication in the 'significand' part of the basic format of the IEEE standard for binary floating-point arithmetic, it is enough to obtain the correct result in the several rounding modes that the product is computed down to the n-th binary digit and the less magnitude part of it is examined whether it is positive or negative or zero. Furthermore, in multiplication in the 'significand' part of the basic format of the IEEE standard, since the multiplicand and the multiplier are normalized, i.e., their integer part is 1, the amount of hardware can be slightly reduced.

Although unsigned binary multiplication has been considered, the algorithm can also be applied to two's complement binary multiplication, directly. The multiplier recoding method (2-bit Booth's method) and the partial product generating method stated in Subsection 3.3.1 can also be applied effectively.

A. Avizienis discussed a sequential multiplication method in signed-digit number systems [AVIZ6109]. D. E. Atkins implemented a serial-parallel multiplier using the redundant binary representation in Illiac III [ATKI7008]. The algorithm proposed in this chapter is not merely an extended version of these algorithms but makes the best of parallelism of hardware.

Recently, J. E. Vuillemin developed a similar multiplication algorithm, independently [VUIL8304] [LUK-V83], in which carry save form whose each digit is 0 or 1 or 2 is used. Both the proposed algorithm and his realize high-speed computation and a regular structure. Compared with his algorithm, the proposed algorithm can adopt 2-bit Booth's method more easily, because a redundant binary number can be itself either positive or negative and it is easy to handle signed numbers. Furthermore, in the proposed multiplier, sign extension is unnecessary in addition, as shown in Section 3.3. M. Kameyama et al. mentioned design of a multiplier with the SD4 representation for digital filtering by means of a 7-valued logic circuit [KAMEH80]. It is interesting to use a multiple-valued logic circuit for design of the proposed multiplier.

A 16-bit two's complement binary multiplier based on the proposed algorithm has recently been implemented on an LSI chip [HARAN8410] [HARAN8702]. A floating-point arithmetic processor including a multiplier based on the algorithm is now under developing [KUNIN8705].

Chapter 4

A Subtract-and-Shift Division Hardware Algorithm

4.1 Introduction

Division is one of the fundamental arithmetic operations as well as addition, subtraction and multiplication. Division is not merely important by itself, but also is used as a basic operation in various algorithms for computing arithmetic functions. Various division algorithms have been proposed, and some of them are practically used. They are classified in two large groups, namely subtract-and-shift methods and multiplicative methods.

Subtract-and-shift division methods have been widely implemented by software or firmware in various digital computing systems with an adder/subtracter and a shifter. As well as the wellknown restoring division algorithm, the nonrestoring division algorithm, the SRT division algorithm [ROBE5809] and their modifications have been proposed. Carry-look-ahead adders have been adopted to improve the computation speed.

Subtract-and-shift methods are also suited to hardware implementation. Recent advances of IC technologies are making it possible to implement a subtract-and-shift divider as a combinational circuit on a VLSI chip [MCALZ8602]. A subtract-and-shift divider with ripple-carry adders has a regular cellular array structure, and therefore, is suitable for VLSI implementation.

However, it does not operate so fast for longer operands, because of carry (borrow) propagation in each addition (subtraction). It requires a computation time proportional to n^2 for n-bit division. Using carry-look-ahead adders instead of ripple-carry adders, the computation time is reduced to being proportional to $n\log n$. However, the structure of the divider becomes rather complicated.

Multiplicative division methods, such as one based on Newton-Raphson method, are implemented by software or firmware in modern digital systems with a high-speed multiplier. n-bit multiplicative division can be performed by about $2 \cdot \log_2 n$-time multiplications. If we implement a divider based on this type of algorithm as a combinational circuit using $\log n$-stage multipliers, n-bit division can be performed in a time proportional to $(\log n)^2$. However, the amount of hardware is too large to fabricate on a VLSI chip in the near future. These methods are suited to software or firmware implementation in digital systems with a high-speed multiplier. A multiplier based on the algorithm proposed in Chapter 3 can effectively be used for implementation of these algorithms, as mentioned in Section 3.4.

In this chapter, a subtract-and-shift division algorithm with the redundant binary representation will be proposed [TAKAY8306b] [TAKAY8404]. A divider based on the algorithm can perform n-bit division in a time proportional to n, and further, has a regular cellular array structure suitable for VLSI implementation.

It is assumed that the dividend X and the divisor Y are n-bit normalized binary numbers. Namely, $X=[1.x_1 \cdots x_{n-1}]_2$ ($x_i \in \{0,1\}$) and $Y=[1.y_1 \cdots y_{n-1}]_2$ ($y_i \in \{0,1\}$). Since $1 \leq X < 2$ and $1 \leq Y < 2$, the quotient Z satisfies $1/2 < Z < 2$. We compute the quotient down to the n-th binary digit. Therefore, the quotient is an (n+1)-bit unsigned binary number with 1-bit integer part and n-bit fraction part.

In the next section, a subtract-and-shift division hardware algorithm will be proposed. In Section 4.3, a divider based on the algorithm will be discussed. Some further discussions will be made in Section 4.4. Section 4.A will appear as an appendix, in which a proof of the correctness of the algorithm will be shown.

4.2 A Division Hardware Algorithm

4.2.1 Algorithm

As well as conventional (radix 2) subtract-and-shift division algorithms, the proposed algorithm is described by the following iteration equation.

$$R_j = R_{j-1} - q_j \cdot 2^{-j} \cdot Y$$

q_j is the quotient digit in the j-th binary position. Y is the divisor. R_{j-1} is the partial dividend before the determination of q_j (the partial remainder from the previous step). R_j is the partial remainder after the determination of q_j.

In the algorithm, each R_j is represented by an (n+1)-digit redundant binary number whose most significant digit is located

at the (j-1)st binary position. q_j is selected from a digit set $\{\bar{1},0,1\}$ by evaluating the most significant three digits of R_{j-1}, i.e., r^{j-1}_{j-2}, r^{j-1}_{j-1} and r^{j-1}_{j}. The calculation of the iteration equation is performed in the redundant binary number system.

The algorithm is as follows.

Algorithm [DIV]

<Input>

X and Y : a dividend and a divisor, respectively

(n-bit normalized binary numbers)

<Output>

Z : the quotient

(an (n+1)-bit unsigned binary number)

<Algorithm>

Step 1: $q_0 := 1$

$R_0 := X - Y$

(redundant binary subtraction)

Step 2: for j:=1 to n do

begin

$$q_j := \begin{cases} \bar{1} & \text{if } [r^{j-1}_{j-2} r^{j-1}_{j-1} r^{j-1}_{j}]_{SD2} < 0 \\ 0 & \text{if } [r^{j-1}_{j-2} r^{j-1}_{j-1} r^{j-1}_{j}]_{SD2} = 0 \\ 1 & \text{if } [r^{j-1}_{j-2} r^{j-1}_{j-1} r^{j-1}_{j}]_{SD2} > 0 \end{cases}$$

$R_j := R_{j-1} - q_j \cdot 2^{-j} \cdot Y$

(redundant binary addition / subtraction)

end

Step 3: $Z \leftarrow Q_n (= [q_0 . q_1 \cdots q_n]_{SD2})$

(redundant binary to binary conversion) □

In Step 1, q_0 is let be 1 and $R_0 := X-Y$ is calculated in the redundant binary number system. Since X and Y are binary numbers, i.e., special redundant binary numbers with nonnegative digits, the calculation of R_0 is very simple. The computation can be done using the computation rule shown in Table 2-4 in Subsection 2.3.4.

In Step 2, quotient digits q_j's are obtained one by one. Each R_j is represented by an (n+1)-digit redundant binary representation whose most significant digit is located at the (j-1)st binary position. Each quotient digit q_j is selected from the digit set $\{\bar{1},0,1\}$ by evaluating the most significant three digits of R_{j-1}. The calculation of the iteration equation is performed in the redundant binary number system. Since each digit of Y is nonnegative, the addition (subtraction) is simpler than a general case as stated in Subsection 2.3.4. In Step 2, these computations are performed n times (for j=1 to n).

It can be proved that in the computation according to the algorithm, each R_j satisfies $-2^{-j} \cdot Y < R_j < 2^{-j} \cdot Y$. (See [Lemma 4.2] in Section 4.A.) Hence, each R_j satisfies $-2^{-j+1} < R_j < 2^{-j+1}$, and therefore, can be represented by a redundant binary representation whose most significant digit is located at the (j-1)st binary position. In order to let the digit at the (j-2)nd binary position of each R_j be 0, in the addition (subtraction) to obtain R_j, a special computation rule has to be applied at the (j-2)nd and the (j-1)st binary position. Indeed, there exists such a rule.

In Step 3, the quotient $(Q_n = [q_0 . q_1 \cdots q_n]_{SD2})$ is converted

into the equivalent unsigned binary number Z.

Performing division according to the algorithm, the following theorem holds.

[Theorem 4]

The difference between the obtained quotient Z and X/Y is smaller than 2^{-n}. Namely, $|Z-X/Y|<2^{-n}$ holds.

This theorem will be proved in Section 4.A.

Fig. 4-1 shows an example of division in accordance with the algorithm. (In the additions $R_{j-1}+2^{-j}\cdot Y$ and $R_{j-1}+(-2^{-j}\cdot Y)$, the addition rule shown in Table 2-2 and that shown in Table 2-3 in Section 2.4 are used, respectively.)

Fig. 4-2 shows a block diagram of a divider based on the algorithm. ▣ denotes a quotient digit determination cell, and □ denotes a redundant binary addition / subtraction cell. The redundant binary to binary converter can be either a ripple-carry adder or a carry-look-ahead adder.

4.2.2 Analysis of the Algorithm

The computation time for Step 1 is constant independent of n. The required gate count is proportional to n.

In Step 2, the determination of each q_j can be done in a constant time independent of n, because it is carried out by evaluating only three digits of R_{j-1}. The calculation of the iteration equation can also be performed in a constant time independent of n, because a negation of a redundant binary number

$$[1.0100001000]_2 \; / \; [1.0110100001]_2$$

$$
\begin{array}{ll}
 & 10100001000 \\
q_0=1 & -\ 10110100001 \\
\hline
 & \underline{00}0\bar{1}0\bar{1}01001\bar{1} \\
q_1=0 & \\
\hline
 & \underline{000}\bar{1}0\bar{1}0100\bar{1}0 \\
q_2=0 & \\
\hline
 & \underline{00\bar{1}}0\bar{1}0100\bar{1}00 \\
q_3=\bar{1} & +\ 10110100001 \\
\hline
 & \underline{000}100\bar{1}0\bar{1}001 \\
q_4=0 & \\
\hline
 & \underline{001}00\bar{1}0\bar{1}0010 \\
q_5=1 & -\ 10110100001 \\
\hline
 & \underline{00\bar{1}}00\bar{1}00010\bar{1} \\
q_6=\bar{1} & +\ 10110100001 \\
\hline
 & \underline{001}\bar{1}01\bar{1}1\bar{1}0\bar{1}1 \\
q_7=1 & -\ 10110100001 \\
\hline
 & \underline{0\bar{1}0}1\bar{1}1\bar{1}1\bar{1}11\bar{1} \\
q_8=\bar{1} & +\ 10110100001 \\
\hline
 & \underline{00\bar{1}}100000\bar{1}\bar{1}1 \\
q_9=\bar{1} & +\ 10110100001 \\
\hline
 & \underline{010}0\bar{1}1\bar{1}0\bar{1}0\bar{1}1 \\
q_{10}=1 & -\ 10110100001 \\
\hline
 & \underline{01\bar{1}}0\bar{1}001\bar{1}11\bar{1} \\
q_{11}=1 &
\end{array}
$$

$$[1.00\bar{1}01\bar{1}1\bar{1}\bar{1}11]_{SD2} \Longrightarrow [0.11100100111]_2$$

Fig. 4-1 An example of division according to Algorithm [DIV]

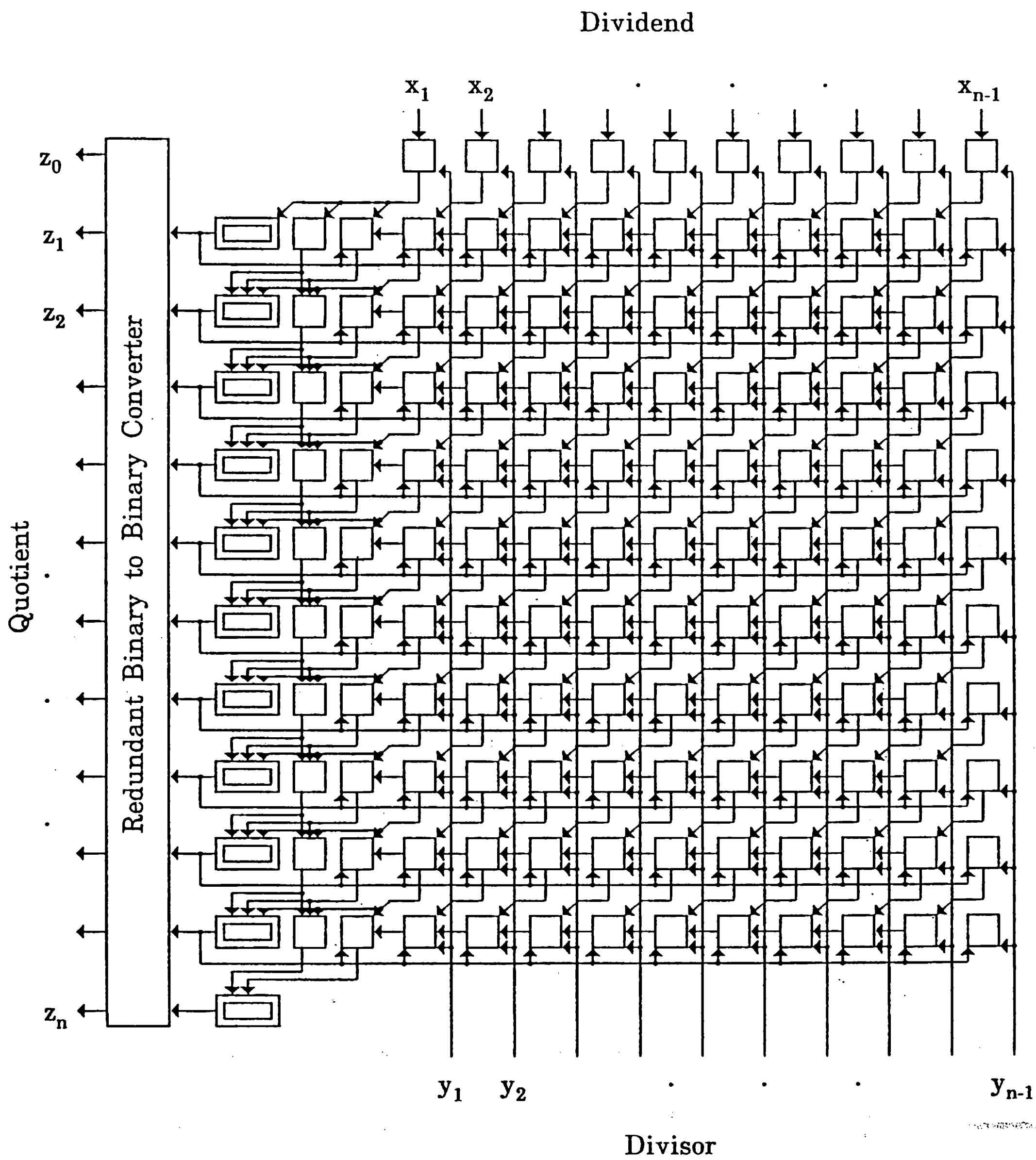

Fig. 4-2 A block diagram of a divider based on Algorithm [DIV]

can be obtained in a constant time independent of the word length of the number and parallel addition (subtraction) of two redundant binary numbers can also be performed in a constant time. Since these computations are performed n times, the computation time for Step 2 is proportional to n. The required gate count is proportional to n^2.

The conversion in Step 3 can be performed in a computation time proportional to $\log n$ by means of a carry-look-ahead adder or in a computation time proportional to n by means of a ripple-carry adder, as mentioned in Section 2.3. The gate count of either adder is proportional to n.

Thus, we conclude that n-bit division can be performed in a computation time proportional to n with a gate count proportional to n^2. Namely, the depth of a divider based on the algorithm is $O(n)$, and the gate count of it is $O(n^2)$. As shown in Fig. 4-2, the divider has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2)$.

Table 4-1 A comparison of three types of subtract-and-shift divider

	Depth	Gate Count	Area	Layout
Proposed One	$O(n)$	$O(n^2)$	$O(n^2)$	simple
with RCA's	$O(n^2)$	$O(n^2)$	$O(n^2)$	simple
with CLA's	$O(n \cdot \log n)$	$O(n^2)$	$O(n^2 \log n)$	rather complicated

Table 4-1 shows a comparison of subtract-and-shift dividers regarding the depth, the gate count, the chip area and the complexity of layout. As shown in the table, the depth of a divider based on the proposed algorithm is O(n), which is smaller than those of the other two dividers. The gate counts of three types of divider are all $O(n^2)$. The chip area of the proposed divider, as well as that of the one with ripple-carry adders, is $O(n^2)$ and is smaller than that of the one with carry-look-ahead adders. Furthermore, the proposed divider has a regular cellular array structure similar to the one with ripple-carry adders and its layout is much simpler than that of the one with carry-look-ahead adders.

4.3 A Divider Based on the Algorithm

In the calculation to obtain R_j in Step 2, the amount of hardware can be reduced by using $\bar{Y}+2^{-n+1}$ as $-Y$, where $\bar{Y}=[\bar{1}0.\tilde{y}_1\cdots\tilde{y}_{n-1}]_{SD2}$. ($\tilde{y}_i$ is 1 or 0 accordingly as y_i is 0 or 1.) (Recall the discussion in Section 2.3.) Namely, $R_{j-1}+2^{-j}\cdot Y$ or $R_{j-1}+0$ or $R_{j-1}+2^{-j}\cdot(\bar{Y}+2^{-n+1})$ is calculated accordingly, as q_j is $\bar{1}$ or 0 or 1. Then the calculation is reduced to redundant binary addition in which all addend digits except the one at the (j-1)st binary position are nonnegative. The addition rule shown in Table 2-2 in Section 2.4 can be used at the j-th and the less binary positions. The computation rule at the (j-1)st binary position is shown in Table 4-2. Doing the calculation according to these

rules, the digit at the (j-2)nd binary position of R_j surely becomes 0. (Since $-2^{-j+2}<R_{j-1}<2^{-j+2}$ (see Section 4.A), $r^{j-1}_{j-2}r^{j-1}_{j-1}$ is neither 11 nor $\bar{1}\bar{1}$. Furthermore, when $r^{j-1}_{j-2}r^{j-1}_{j-1}$ is 10, r^{j-1}_{j} is either 0 or $\bar{1}$, and when $r^{j-1}_{j-2}r^{j-1}_{j-1}$ is $\bar{1}0$, r^{j-1}_{j} is either 1 or 0.)

Some of less significant digits in R_j's for $j>n/2$ have no effect on the quotient. Therefore, the computation for these digits can be omitted.

Fig. 4-3 shows an example of division according to the algorithm in consideration of the above discussions.

We can make a fast redundant binary to binary converter with rather small amount of hardware by utilizing the fact that the quotient digits q_j's are obtained one by one from the most significant digit.

Table 4-2 A computation rule at the (j-1)st binary position

		r^{j}_{j-1}		
r^{j-1}_{j-2}	r^{j-1}_{j-1} \ r^{j-1}_{j}	$\bar{1}$	0	1
$\bar{1}$	0	---	$\bar{1}$	$\bar{1}$
$\bar{1}$	1	$\bar{1}$	0	0
0	$\bar{1}$	$\bar{1}$	0	0
0	0	0	0	0
0	1	0	0	1
1	$\bar{1}$	0	0	1
1	0	1	1	---

$$[1.0100001000]_2 \ / \ [1.0110100001]_2$$

$$
\begin{array}{lrl}
 & & 10100001000 \\
q_0=1 & - & 10110100001 \\ \hline
 & & \underline{00}0\bar{1}0\bar{1}0100\bar{1} \\
q_1=0 & + & 000000000000(+0) \\ \hline
 & & \underline{000}\bar{1}0\bar{1}1\bar{1}00\bar{1}0 \\
q_2=0 & + & 000000000000(+0) \\ \hline
 & & \underline{00\bar{1}}00\bar{1}\bar{1}00\bar{1}00 \\
q_3=\bar{1} & + & 010110100001(+0) \\ \hline
 & & \underline{001}\bar{1}00\bar{1}0\bar{1}01\bar{1} \\
q_4=1 & + & \bar{1}01001011110(+1) \\ \hline
 & & \underline{0\bar{1}0}000010011\bar{1} \\
q_5=\bar{1} & + & 010110100001(+0) \\ \hline
 & & \underline{0\bar{1}1}0\bar{1}1001\bar{1}0\bar{1} \\
q_6=\bar{1} & + & 010110100001(+0) \\ \hline
 & & \underline{000}10101\bar{1}\bar{1}00 \\
q_7=0 & + & 0000000000 \\ \hline
 & & \underline{01\bar{1}}1\bar{1}0\bar{1}\bar{1}0 \\
q_8=1 & + & \bar{1}0100101 \\ \hline
 & & \underline{000}\bar{1}000 \\
q_9=0 & + & 000000 \\ \hline
 & & \underline{00\bar{1}}00 \\
q_{10}=\bar{1} & + & 0101 \\ \hline
 & & \underline{001} \\
q_{11}=1 & &
\end{array}
$$

$$[1.00\bar{1}1\bar{1}\bar{1}010\bar{1}1]_{SD2} \Longrightarrow [0.11100100111]_2$$

Fig. 4-3 An example of division according to Algorithm [DIV] with several considerations

Table 4-3 shows an evaluation of the depth and the gate counts of a divider based on the proposed algorithm with the above consideration, a subtract-and-shift divider with ripple-carry adders and one with carry-look-ahead adders, by the use of CMOS gates as computation elements.

Table 4-3 Depth and gate counts of three types of subtract-and-shift divider

depth / gate count (the number of transistors)

	24-bit	53-bit
Proposed One	118 / 2412 (13778)	263 / 11118 (65162)
with RCA's	622 / 2968 (16656)	2863 / 14979 (84352)
with CLA's	216 / 4598 (22336)	581 / 24078 (117120)

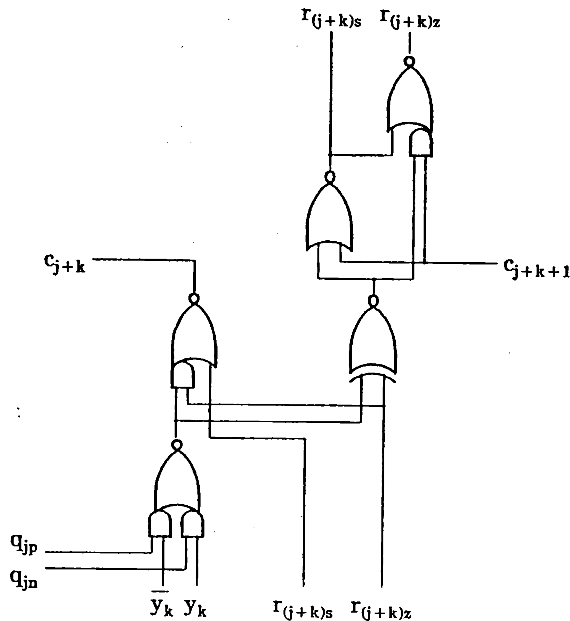

Fig. 4-4 A CMOS logic design of the typical redundant binary addition/subtraction cell

In the logic design of the proposed divider, a typical quotient determination cell and a typical redundant binary addition / subtraction cell (in which the addend digit is nonnegative) are composed of 2 gates (20 transistors) and 5 gates (30 transistors), respectively. Fig. 4-4. shows a CMOS logic design of the typical redundant binary addition / subtraction cell. A quotient digit q_j is represented by two bits, q_{jn} and q_{jp}, and 10 or 00 or 01 is assigned to $q_{jn}q_{jp}$ accordingly as q_j is $\bar{1}$ or 0 or 1. A partial remainder digit r_i^j is represented by two bits, r_{is}^j and r_{iz}^j, and 10 or 01 or 00 is assigned to $r_{is}^j r_{iz}^j$ accordingly as r_i^j is $\bar{1}$ or 0 or 1.

As shown in Table 4-3, the depth (i.e., the computation time) of the proposed divider is smaller than those of the other two dividers, especially when the word length is longer. The 24-bit proposed divider is about five times faster than the 24-bit one with ripple-carry adders, and the 53-bit proposed divider is about ten times faster than the 53-bit one with ripple-carry adders. The proposed divider is twice or more faster than the one with carry-look-ahead adders. The gate count of the proposed divider is similar to that of the one with ripple-carry adders and smaller than that of the one with carry-look-ahead adders. Furthermore, as shown in Fig. 4-2, the proposed divider has a regular cellular array structure similar to the one with ripple-carry adders, and therefore, its layout is simpler than that of the one with carry-look-ahead adders.

Thus, the proposed divider is excellent in computation speed, the amount of hardware and regularity in layout.

4.4 Remarks and Discussions

A subtract-and-shift division hardware algorithm with the redundant binary representation has been proposed. The division algorithm performs n-bit binary division in a time proportional to n with a gate count proportional to n^2. Namely, the depth of a divider based on the algorithm is $O(n)$, and the gate count of it is $O(n^2)$. It has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2)$.

In the previous sections, the quotient was computed down to the n-th binary digit, but the final remainder was not considered. The absolute error of the quotient was guaranteed to be smaller than 2^{-n}. However, in division in the 'significand' part of the basic format of the IEEE standard for binary floating-point arithmetic, in order to obtain the correct result in the several rounding modes, the quotient has to be computed down to the (n+1)st binary digit and the final remainder has to be examined whether it is positive or negative or zero. The proposed divider can easily be modified to fit the standard.

D. E. Atkins proposed a subtract-and-shift division algorithm using signed-digit number representations and a higher-radix method [ATKI6810] and implemented a divider based on the algorithm as a sequential circuit in Illiac III [ATKI7008]. The algorithm can also be implemented as a combinational circuit. A hardware divider based on the algorithm realizes high-speed computation and a regular cellular array structure suitable for

VLSI implementation. The amount of hardware of it will be smaller than that of the divider proposed in this chapter, because each quotient digit is determined from $\{\bar{2},\bar{1},0,1,2\}$ and at most one addition (subtraction) is needed for an according quotient digit. However, the computation time is not shorter because of the complex computation for determination of a quotient digit. A high-speed subtract-and-shift divider with a regular cellular array structure like the proposed one can be obtained by adopting carry-save adders in a divider based on a modification of the SRT method [TAYL8506] [FAND8705]. Recently, a VLSI chip of such a divider has been fabricated [MCALZ8602].

A divider based on the proposed algorithm can be realized as a combinational circuit on a VLSI chip using today's IC technology. A floating-point arithmetic processor including a divider based on the algorithm is now under developing [KUNIN8705].

The division algorithm proposed in this chapter can also be implemented by software or firmware in a digital system with a redundant binary adder/subtracter and a shifter. Division can be performed rather efficiently with a small amount of hardware.

4.A A Proof of the Correctness of the Algorithm

In this section [Theorem 4] is proved. Namely, the fact that the quotient Z obtained by the proposed division algorithm, Algorithm [DIV], satisfies $|Z-X/Y|<2^{-n}$ is shown. In order to show the fact, the following two lemmas are proved first. In the

following, Q_j denotes $[q_0.q_1\cdots q_j]_{SD2}$.

[Lemma 4.1]

$R_j=X-Q_j\cdot Y$ holds for all j's ($0\leqq j\leqq n$).

<Proof>

The proof can be established by induction over j.

(1) When j=0, since $R_0=X-Y$ and $Q_0=1$, $R_0=X-Q_0\cdot Y$ holds.

(2) Assume that $R_{j-1}=X-Q_{j-1}\cdot Y$ holds. Then,

$$R_j=R_{j-1}-q_j\cdot 2^{-j}\cdot Y$$
$$=(X-Q_{j-1}\cdot Y)-q_j\cdot 2^{-j}\cdot Y$$
$$=X-(Q_{j-1}+q_j\cdot 2^{-j})\cdot Y$$
$$=X-Q_j\cdot Y.$$

Thus, $R_j=X-Q_j\cdot Y$ holds.

From (1) and (2), $R_j=X-Q_j\cdot Y$ holds for all j's ($0\leqq j\leqq n$).

Q.E.D.

[Lemma 4.2]

$-2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$ holds for all j's ($0\leqq j\leqq n$).

<Proof>

The proof can be established by induction over j.

(1) When j=0, since $R_0=X-Y$, $-2^0\cdot Y<R_0<2^0\cdot Y$ holds.

(Recall that $1\leqq X<2$ and $1\leqq Y<2$.)

(2) Assume that $-2^{-j+1}\cdot Y<R_{j-1}<2^{-j+1}\cdot Y$ holds. Let us consider the following three cases.

Case 1: $q_j=\bar{1}$

$$-2^{-j+1}\cdot Y<R_{j-1}<0,\quad R_j=R_{j-1}+2^{-j}\cdot Y$$

$$\therefore\ -2^{-j+1}\cdot Y+2^{-j}\cdot Y<R_j<0+2^{-j}\cdot Y$$

$\therefore\ -2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$

Case 2: $q_j=0$

$-2^{-j}<R_{j-1}<2^{-j}$, $R_j=R_{j-1}+0$

$\therefore\ -2^{-j}+0<R_j<2^{-j}+0$

$\therefore\ -2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$ $\quad(\because\ Y\geqq 1)$

Case 3: $q_j=1$

$0<R_{j-1}<2^{-j+1}\cdot Y$, $R_j=R_{j-1}-2^{-j}\cdot Y$

$\therefore\ 0-2^{-j}\cdot Y<R_j<2^{-j+1}\cdot Y-2^{-j}\cdot Y$

$\therefore\ -2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$

Thus $-2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$ holds in any case.

From (1) and (2), $-2^{-j}\cdot Y<R_j<2^{-j}\cdot Y$ holds for all j's ($0\leqq j\leqq n$).

Q.E.D.

From [Lemma 4.2], $-2^{-j+1}<R_j<2^{-j+1}$ holds, and therefore, R_j can be represented by a redundant binary representation whose most significant digit is located at the (j-1)st binary position. Indeed, it can, when we compute as stated in Section 4.3.

Now, [Theorem 4] can be proved.

[Theorem 4]

$|Z-X/Y|<2^{-n}$ holds.

<Proof>

$Z=Q_n$

$Q_n-X/Y=-R_n/Y$ (from [Lemma 4.1])

$-2^{-n}<R/Y<2^{-n}$ (from [Lemma 4.2])

$\therefore\ -2^{-n}<Z-X/Y<2^{-n}$

Hence, $|Z-X/Y|<2^{-n}$ holds.

Q.E.D.

Chapter 5

A Subtract-and-Shift Square Root Hardware Algorithm

5.1 Introduction

Extraction of the square root of a number is one of the most important elementary arithmetic functions. It is used in various computations, such as calculation of the distance between two points, calculation of the roots of a quadratic equation, and so on. For square root extraction, similar to the case of division, subtract-and-shift methods and multiplicative methods have been developed and practically used.

Subtract-and-shift square root methods have been widely implemented by software or firmware in various digital computing systems with an adder/subtracter and a shifter. The restoring square root algorithm [LENA5507], the nonrestoring one [COWG6404] and their modifications [METZ6504] have been proposed. These methods are also suited to hardware implementation. However, a square root circuit based on this type of algorithm does not operate so fast for longer operands, because of carry (borrow) propagation in each addition (subtraction).

Multiplicative square root methods, such as one based on Newton-Raphson method, are implemented by software or firmware in modern digital systems with a high-speed multiplier. However, if we implement a square root circuit based on this type of

algorithm as a combinational circuit, the amount of hardware becomes too large to fabricate on a VLSI chip in the near future. These methods are suited to software or firmware implementation in digital systems with a high-speed multiplier. A multiplier based on the algorithm proposed in Chapter 3 can effectively be used for implementation of these methods, as mentioned in Section 3.4.

In this chapter, a new subtract-and-shift square root algorithm with the redundant binary representation will be proposed [TAKAY8306b] [TAKAY8601a]. A square root circuit based on the algorithm can perform high-speed square root extraction, and further, has a regular cellular array structure suitable for VLSI implementation.

It is assumed that the radicand X is an (n+1)-bit unsigned binary number with 2-bit integer part and (n-1)-bit fraction part, and satisfies $1 \leq X < 4$. This assumption is sound in the computation of 'significand' part of the basic format of the IEEE standard for binary floating-point arithmetic. The square root Z satisfies $1 \leq Z < 2$. We compute the square root down to the (n-1)st binary digit. Therefore, the square root is a n-bit normalized binary number.

In the next section, a new subtract-and-shift square root algorithm will be proposed. In Section 5.3, a square root circuit based on the algorithm will be discussed. Some further discussions will be made in Section 5.4. Section 5.A will appear as an appendix, in which a proof of the correctness of the algorithm will be shown.

5.2 A Square Root Hardware Algorithm

5.2.1 Algorithm

As well as conventional (radix 2) subtract-and-shift square root algorithms, the proposed algorithm is described by the following iteration equations.

$$R_j = R_{j-1} - q_j \cdot 2^{-j} \cdot (2 \cdot Q_{j-1} + q_j \cdot 2^{-j})$$

$$Q_j = Q_{j-1} + q_j \cdot 2^{-j}$$

q_j is the square root digit in the j-th binary position. R_{j-1} is the partial radicand before the determination of q_j (the partial remainder from the previous step). R_j is the partial remainder after the determination of q_j. Q_j $(=[q_0 . q_1 \cdots q_j])$ denotes the truncated square root down to the j-th binary digit.

In the algorithm, each R_j is represented by an (n-j+2)-digit (for $j<n/2$) or (j+3)-digit (for $j \geq n/2$) redundant binary number whose most significant digit is located at the (j-2)nd binary position. q_j is selected from a digit set $\{\bar{1},0,1\}$ by evaluating the most significant three digits of R_{j-1}, i.e., r^{j-1}_{j-3}, r^{j-1}_{j-2} and r^{j-1}_{j-1}. The calculation of the iteration quations is performed in the redundant binary number system.

The algorithm is as follows.

Algorithm [SQR]

<Input>

X : a radicand $(1 \leq X < 4)$

(an (n+1)-bit binary number with 2-bit integer part)

<Output>

Z : the square root

(an n-bit normalized binary number)

<Algorithm>

Step 1: $q_0 := 1$

$R_0 := X-1$

$Q_0 := [1]_{SD2}$

Step 2: for j:=1 to n-1 do

begin

$$q_j := \begin{cases} \bar{1} & \text{if } [r^{j-1}_{j-3} r^{j-1}_{j-2} r^{j-1}_{j-1}]_{SD2} < 0 \\ 0 & \text{if } [r^{j-1}_{j-3} r^{j-1}_{j-2} r^{j-1}_{j-1}]_{SD2} = 0 \\ 1 & \text{if } [r^{j-1}_{j-3} r^{j-1}_{j-2} r^{j-1}_{j-1}]_{SD2} > 0 \end{cases}$$

$R_j := R_{j-1} - q_j \cdot 2^{-j} \cdot (2 \cdot Q_{j-1} + q_j \cdot 2^{-j})$

(redundant binary addition / subtraction)

$Q_j := [1.q_1 \cdots q_{j-1} q_j]_{SD2}$

end

Step 3: $Z \leftarrow Q_{n-1}$

(redundant binary to binary conversion) □

In Step 1, q_0 is let be 1. In the calculation of R_0, only the integer part is calculated.

In Step 2, square root digits q_j's are obtained one by one. Each R_j is represented by an (n-j+2)-digit (for j<n/2) or (j+3)-digit (for j≧n/2) redundant binary number whose most significant digit is located at the (j-2)nd binary position. Each square root digit q_j is selected from the digit set $\{\bar{1},0,1\}$ by evaluating the most significant three digits of R_{j-1}. The calculation of the iteration equation is performed in the redundant binary number

system. For $j<n/2$, calculation is done for only down to the 2j-th binary position. In Step 2, these computations are performed n-1 times (for j=1 to n-1).

It can be proved that in the computation according to the algorithm, each R_j satisfies $-2^{-j+1}\cdot Q_j+2^{-2j}<R_j<2^{-j+1}\cdot Q_j+2^{-2j}$. (See [Lemma 5.3] in Section 5.A.) Hence, each R_j satisfies $-2^{-j+2}<R_j<2^{-j+2}$, and therefore, can be represented by a redundant binary number whose most significant digit is located at the (j-2)nd binary position. In order to let the digit at the (j-3)rd binary position of each R_j be 0, in the addition (subtraction) to obtain R_j, a special computation rule has to be applied at the (j-3)rd and the (j-2)nd binary position. Indeed, there exists such a rule.

In Step 3, the square root Q_{n-1} $(=[q_0 . q_1 \cdots q_{n-1}]_{SD2})$ is converted into the equivalent unsigned binary number Z.

Performing square root extraction according to the algorithm, the following theorem holds.

[Theorem 5]

The difference between the obtained square root Z and $\sqrt{X}$ is smaller than 2^{-n+1}. Namely, $|Z-\sqrt{X}|<2^{-n+1}$ holds.

This theorem will be proved in Section 5.A.

Fig. 5-1 shows an example of square root extraction in accordance with the algorithm.

$$[01.1100100110]_2$$

	01.1100100110
$q_0=1$	$-\ \ 1$
	$\underline{00}.11$
$q_1=0$	
	$\underline{00.1}100$
$q_2=1$	$+\ \ \bar{1}0\ \bar{1}$
	$\underline{0.1\bar{1}}0\bar{1}10$
$q_3=1$	$+\ \ \bar{1}0\bar{1}\ \bar{1}$
	$.\underline{00\bar{1}}01\bar{1}01$
$q_4=\bar{1}$	$+\ \ 1011\ \bar{1}$
	$\underline{001}000010$
$q_5=1$	$+\ \ \bar{1}0\bar{1}\bar{1}1\ \bar{1}$
	$\underline{00\bar{1}}10\bar{1}1\bar{1}$
$q_6=\bar{1}$	$+\ \ 1011\bar{1}1\ \bar{1}$
	$\underline{01\bar{1}}10000\bar{1}$
$q_7=1$	$+\ \ \bar{1}0\bar{1}\bar{1}1\bar{1}1\ \bar{1}$
	$\underline{000}1\bar{1}0100\bar{1}$
$q_8=0$	
	$\underline{001}\bar{1}0100\bar{1}00$
$q_9=1$	$+\ \ \bar{1}0\bar{1}\bar{1}\bar{1}\bar{1}1\bar{1}0\ \bar{1}$
	$\underline{0\bar{1}0}10010\bar{1}00\bar{1}$
$q_{10}=\bar{1}$	

$$[1.011\bar{1}1\bar{1}101\bar{1}]_{SD2} \Longrightarrow [1.0101011001]_2$$

Fig. 5-1 An example of square root extraction according to Algorithm [SQR]

Fig. 5-2 shows a block diagram of a square root circuit based on the algorithm. ▣ denotes a square root digit determination cell and □ denotes a redundant binary addition / subtraction cell. The redundant binary to binary converter can be either a ripple-carry adder or a carry-look-ahead adder.

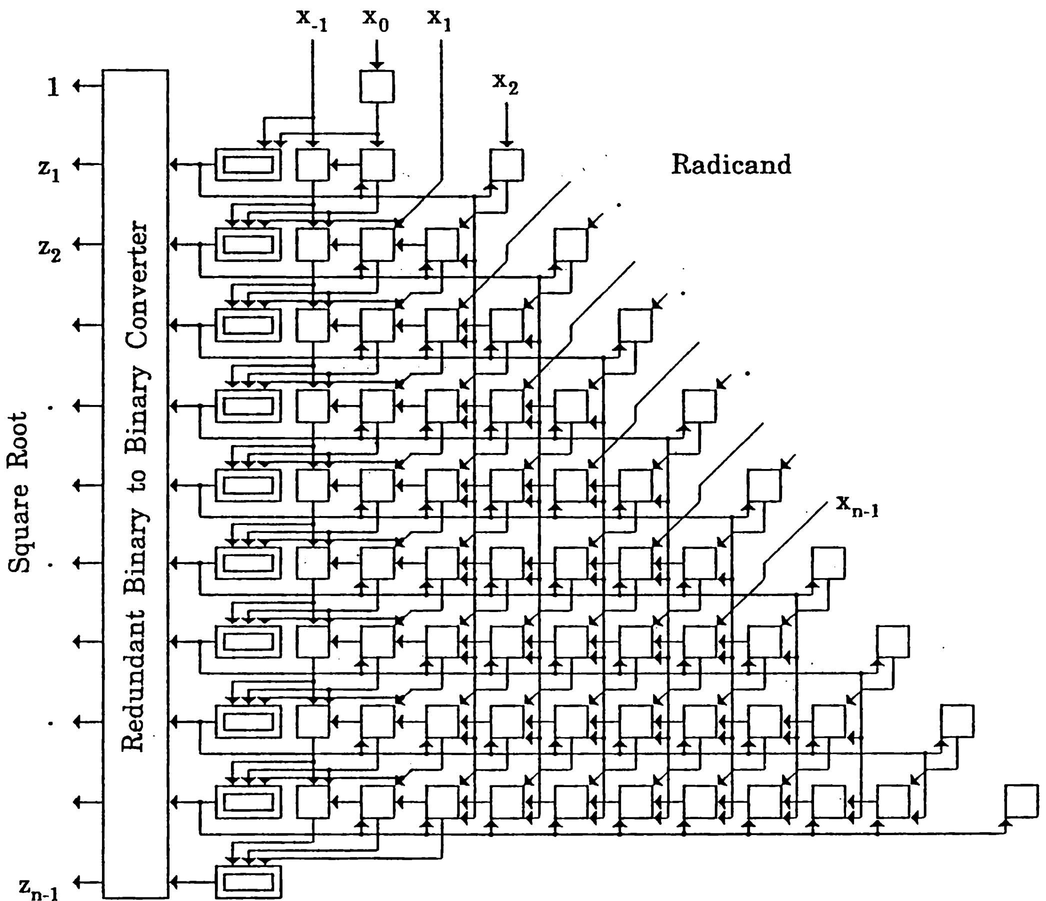

Fig. 5-2 A block diagram of a square root circuit based on Algorithm [SQR]

5.2.2 Analysis of the Algorithm

The computation time for Step 1 is constant independent of n. The required gate count is also constant independent of n.

In Step 2, the determination of each q_j can be done in a constant time independent of n, because it is carried out by evaluating only three digits of R_{j-1}. The calculation of the iteration equation can also be performed in a constant time independent of n, because a negation of a redundant binary number can be obtained in a constant time independent of the word length of the number and parallel addition of two redundant binary numbers can also be performed in a constant time, as discussed in Section 2.3. The required gate count is proportional to j. Since these computations are performed n-1 times, the required computation time for Step 2 is proportional to n. The required gate count is proportional to n^2.

The conversion in Step 3 can be performed in a computation time proportional to $\log n$ by means of a carry-look-ahead adder or in a computation time proportional to n by means of a ripple-carry adder, as mentioned in Section 2.3. The gate count of either adder is proportional to n.

Thus, we conclude that n-bit square root extraction can be performed in a computation time proportional to n with a gate count proportional to n^2. Namely, the depth of a square root circuit based on the algorithm is $O(n)$, and the gate count of it is $O(n^2)$. As shown in Fig. 5-2, the square root circuit has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2)$.

Table 5-1 shows a comparison of subtract-and-shift square root circuits regarding the depth, the gate count, the chip area and the complexity of layout. As shown in the table, the depth of the square root circuit based on the proposed algorithm is $O(n)$, which is smaller than those of the other two square root circuits. The gate counts of three types of circuit are all $O(n^2)$. The chip area of the proposed square root circuit, as well as that of the one with ripple-carry adders, is $O(n^2)$ and is smaller than that of the one with carry-look-ahead adders. Furthermore, the proposed square root circuit has a regular cellular array structure similar to the one with ripple-carry adders and its layout is much simpler than that of the one with carry-look-ahead adders.

Table 5-1 A comparison of three types of subtract-and-shift square root circuit

	Depth	Gate Count	Area	Layout
Proposed One	$O(n)$	$O(n^2)$	$O(n^2)$	simple
with RCA's	$O(n^2)$	$O(n^2)$	$O(n^2)$	simple
with CLA's	$O(n \cdot \log n)$	$O(n^2)$	$O(n^2 \log n)$	rather complicated

5.3 A Square Root Circuit Based on the Algorithm

Some of less significant digits in R_j's for $j>3(n-1)/4$ have no effect on the square root. Therefore, the computation for these digits can be omitted.

We can make a fast redundant binary to binary converter with rather small amount of hardware by utilizing the fact that the square root digits q_j's are obtained one by one from the most significant digit.

Redundant binary addition cells for least significant two positions in each calculation of R_j are much simpler than a typical addition cell.

By a rough estimation, a 24-bit square root circuit based on the algorithm with considering the above discussions is about four times faster than a 24-bit one with ripple-carry adders, and a 53-bit proposed one is about eight times faster than a 53-bit one with ripple-carry adders. The proposed square root circuit is twice or more faster than one with carry-look-ahead adders. The gate count of the proposed square root circuit is similar to that of the one with ripple-carry adders and smaller than that of the one with carry-look-ahead adders. Furthermore, as shown in Fig. 5-2, the proposed square root circuit has a regular cellular array structure similar to the one with ripple-carry adders, and therefore, its layout is simpler than that of the one with carry-look-ahead adders.

Thus, the proposed square root circuit is excellent in computation speed, the amount of hardware and regularity in layout.

5.4 Remarks and Discussions

A new subtract-and-shift square root hardware algorithm with the redundant binary representation has been proposed. The square root algorithm performs n-bit square root extraction in a time proportional to n with a gate count proportional to n^2. Namely, the depth of a square root circuit based on the algorithm is $O(n)$, and the gate count of it is $O(n^2)$. It has a regular cellular array structure, and therefore, it is suitable for VLSI implementation. The chip area of it is $O(n^2)$.

In the previous sections, the square root was computed down to the (n-1)st binary digit, but the final remainder was not considered. The absolute error of the square root was guaranteed to be smaller than 2^{-n+1}. However, in square root extraction in the 'significand' part of the basic format of the IEEE standard, in order to obtain the correct result in the several rounding modes, the square root has to be computed down to the n-th binary digit and the final remainder has to be examined whether it is positive or negative or zero. The proposed square root circuit can easily be modified to fit the standard.

Several researches have been done for speeding up square root extraction by using a carry save adder for computing each partial remainder [MAJE8508] [FAND8705]. These algorithms operate slower and require larger amount of hardware than the proposed algorithm, because the truncated square root (Q_{j-1}) has to be converted in the ordinary binary representation.

A square root circuit based on the proposed algorithm can be

realized as a combinational circuit on a VLSI chip, using today's IC technology.

The square root algorithm proposed in this chapter can also be implemented by software or firmware in a digital system with a redundant binary adder/subtracter and a shifter. Square root extraction can be performed rather efficiently with a small amount of hardware.

5.A A Proof of the Correctness of the Algorithm

In this section, [Theorem 5] is proved. Namely, the fact that the square root Z obtained by the proposed square root algorithm, Algorithm [SQR], satisfies $|Z-\sqrt{X}|<2^{-n+1}$ is shown. In order to prove the fact, the following three lemmas are proved first.

[Lemma 5.1]

$R_j=X-Q_j^2$ holds for all j's $(0\leq j\leq n-1)$.

<Proof>

The proof can be established by induction over j.

(1) When j=0, since $R_0=X-1$ and $Q_0=1$, $R_0=X-Q_0^2$ holds.

(2) Assume that $R_{j-1}=X-Q_{j-1}^2$ holds. Then,

$$R_j=R_{j-1}-q_j\cdot2^{-j}\cdot(2\cdot Q_{j-1}+q_j\cdot2^{-j})$$
$$=X-Q_{j-1}^2-2\cdot Q_{j-1}\cdot q_j\cdot2^{-j}-(q_j\cdot2^{-j})^2$$
$$=X-(Q_{j-1}+q_j\cdot2^{-j})^2$$
$$=X-Q_j^2.$$

Thus, $R_j = X - Q_j^2$ holds.

From (1) and (2), $R_j = X - Q_j^2$ holds for all j's $(0 \leqq j \leqq n-1)$.

Q.E.D.

[Lemma 5.2]

$Q_j \geqq 1$ and especially when $R_j < 0$, $Q_j \geqq 1 + 2^{-j}$ hold for all j's $(0 \leqq j \leqq n-1)$.

<Proof>

$Q_0 = 1$ and $R_0 \geqq 0$. (from Step 1)

When $R_{k-1} \geqq 0$ $(1 \leqq k \leqq n-1)$, $q_k = 0$ or 1. (from Step 2)

When $R_{k-1} > 0$ and $R_k < 0$ $(1 \leqq k \leqq n-1)$, $q_k = 1$. (from Step 2)

Hence, $q_1 = 0$ or 1.

When $R_j < 0$ $(1 \leqq j \leqq n-1)$, there exists k $(1 \leqq k \leqq j)$ such that $q_k = 1$ and $q_i = 0$ for all i's $(1 \leqq i < k)$.

Therefore,

when $R_j < 0$, $Q_j \geqq 1 + 2^{-j}$ holds for all j's $(0 \leqq j \leqq n-1)$. ---(1)

Furthermore, for all j's $(1 \leqq j \leqq n-1)$,

(a) if there exists k $(1 \leqq k \leqq j)$ such that $q_k = \bar{1}$, $Q_j \geqq 1 + 2^{-j}$ since $Q_k \geqq 1 + 2^{-k}$, and

(b) if there does not exist such k, evidently $Q_j \geqq 1$.

Therefore, $Q_j \geqq 1$ holds for all j's $(0 \leqq j \leqq n-1)$. ---(2)

From (1) and (2), $Q_j \geqq 1$ and especially when $R_j < 0$, $Q_j \geqq 1 + 2^{-j}$ hold for all j's $(0 \leqq j \leqq n-1)$.

Q.E.D.

[Lemma 5.3]

$-2^{-j+1} \cdot Q_j + 2^{-2j} < R_j < 2^{-j+1} \cdot Q_j + 2^{-2j}$ holds for all j's $(0 \leqq j \leqq n-1)$.

<Proof>

The proof can be established by induction over j.

(1) When $j=0$, since $Q_0=1$ and $R_0=X-1$, $-2^1 \cdot Q_0+2^0<R_0<2^1 \cdot Q_0+2^0$ holds.

(Recall that $1 \leq X<4$.)

(2) Assume that $-2^{-j+2} \cdot Q_{j-1}+2^{-2j+2}<R_{j-1}<2^{-j+2} \cdot Q_{j-1}+2^{-2j+2}$ holds. Let us consider the following three cases.

Case 1: $q_j=\bar{1}$

$$-2^{-j+2} \cdot Q_{j-1}+2^{-2j+2}<R_{j-1}<0$$

$$R_j=R_{j-1}+2^{-j} \cdot (2 \cdot Q_{j-1}-2^{-j}),\quad Q_j=Q_{j-1}-2^{-j}$$

$$\therefore\ -2^{-j+2} \cdot Q_{j-1}+2^{-2j+2}+2^{-j} \cdot (2 \cdot Q_{j-1}-2^{-j})<R_j<0+2^{-j} \cdot (2 \cdot Q_{j-1}-2^{-j})$$

$$\therefore\ -2^{-j+1} \cdot Q_{j-1}+2^{-2j+2}-2^{-2j}<R_j<2^{-j+1} \cdot Q_{j-1}-2^{-2j}$$

$$\therefore\ -2^{-j+1} \cdot Q_j+2^{-2j}<R_j<2^{-j+1} \cdot Q_j+2^{-2j}$$

Case 2: $q_j=0$

$$-2^{-j+1}<R_{j-1}<2^{-j+1}$$

$$R_j=R_{j-1}+0,\quad Q_j=Q_{j-1}$$

Case 2-1: $-2^{-j+1}<R_{j-1}<0$

$$-2^{-j+1} \cdot Q_j+2^{-2j}=-2^{-j+1} \cdot Q_{j-1}+2^{-2j}$$

$$\leq -2^{-j+1} \cdot (1+2^{-j+1})+2^{-2j} \qquad \text{(from [Lemma 5.2])}$$

$$<-2^{-j+1}<R_j$$

Case 2-2: $0 \leq R_{j-1}<2^{-j+1}$

$$2^{-j+1} \cdot Q_j+2^{-2j}=2^{-j+1} \cdot Q_{j-1}+2^{-2j}$$

$$\geq 2^{-j+1}+2^{-2j} \qquad \text{(from [Lemma 5.2])}$$

$$>2^{-j+1}>R_j$$

$$\therefore\ -2^{-j+1} \cdot Q_j+2^{-2j}<R_j<2^{-j+1} \cdot Q_j+2^{-2j}$$

Case 3: $q_j=1$

$$0<R_{j-1}<2^{-j+2} \cdot Q_{j-1}+2^{-2j+2}$$

$R_j = R_{j-1} - 2^{-j} \cdot (2 \cdot Q_{j-1} + 2^{-j})$, $Q_j = Q_{j-1} + 2^{-j}$

$\therefore\ 0 - 2^{-j} \cdot (2 \cdot Q_{j-1} + 2^{-j}) < R_j < 2^{-j+2} \cdot Q_{j-1} + 2^{-2j+2} - 2^{-j} \cdot (2 \cdot Q_{j-1} + 2^{-j})$

$\therefore\ -2^{-j+1} \cdot Q_{j-1} - 2^{-2j} < R_j < 2^{-j+1} \cdot Q_{j-1} + 2^{-2j+2} - 2^{-2j}$

$\therefore\ -2^{-j+1} \cdot Q_j + 2^{-2j} < R_j < 2^{-j+1} \cdot Q_j + 2^{-2j}$

Thus $-2^{-j+1} \cdot Q_j + 2^{-2j} < R_j < 2^{-j+1} \cdot Q_j + 2^{-2j}$ holds in any case.

From (1) and (2), $-2^{-j+1} \cdot Q_j + 2^{-2j} < R_j < 2^{-j+1} \cdot Q_j + 2^{-2j}$ holds for all j's ($0 \leq j \leq n-1$).

Q.E.D.

From [Lemma 5.3], since $Q_j < 2 - 2^{-j}$, $-2^{-j+2} < R_j < 2^{-j+2}$ holds, and therefore, R_j can be represented by a redundant binary number whose most significant digit is located at the (j-2)nd binary position.

Now, [Theorem 5] can be proved.

[Theorem 5]

$|Z - \sqrt{X}| < 2^{-n+1}$ holds.

<Proof>

$Z = Q_{n-1}$

$R_{n-1} = X - Q_{n-1}^2$ (from [Lemma 5.1])

$-2^{-n+2} \cdot Q_{n-1} + 2^{-2n+2} < R_{n-1} < 2^{-n+2} \cdot Q_{n-1} + 2^{-2n+2}$ (from [Lemma 5.3])

$\therefore\ -2^{-n+2} \cdot Z + 2^{-2n+2} < X - Z^2 < 2^{-n+2} \cdot Z + 2^{-2n+2}$

$\therefore\ (Z - 2^{-n+1})^2 < X < (Z + 2^{-n+1})^2$

$\therefore\ Z - 2^{-n+1} < \sqrt{X} < Z + 2^{-n+1}$

Hence, $|Z - \sqrt{X}| < 2^{-n+1}$ holds.

Q.E.D.

Chapter 6

Hardware Algorithms for Elementary Functions

6.1 Introduction

In this chapter, the computation of several elementary functions, is considered. Trigonometric and inverse trigonometric functions, such as sines, cosines, arctangents, and so on are used in various digital systems, especially in graphic systems. Logarithmic function and exponential function are also important elementary functions in digital systems. First order convergence methods, such as the CORDIC (COordinate Rotation DIgital Computer) method [VOLD5909] [WALT7105] and the STL (Sequential Table Look-up) method [CANTE6204] [SPEC6501] [CHEN7207], have been developed for computing these elementary functions.

By means of the CORDIC method, trigonometric and inverse trigonometric functions are computed by iteration of simple calculations, i.e., shift, addition / subtraction and recall of prepared constants. The CORDIC method has been implemented by software or firmware in various digital computing systems with an adder/subtracter and a shifter. A CORDIC arithmetic processor in which the method is implemented by firmware has been fabricated on an LSI chip [HAVIT8002]. By means of the STL method, logarithms and exponentials are also computed by iteration of simple calculations, as the case of the CORDIC method. The STL

method has also been implemented by software or firmware in digital computing systems with an adder/subtracter and a shifter. Several improved algorithm have been proposed [DELU7006], which are suited to firmware implementation with small amount of hardware.

In this chapter, combinational circuit implementation of circuits for computing elementary functions is considered. The CORDIC method and the STL method are also suited to combinational circuit implementation. An array-structured elementary function circuit has been proposed [TAMAK8303]. However, the computation speed is not so fast because of carry propagation in additions. In this chapter, new high-speed algorithms for computing sines and cosines, for computing arctangents, for computing logarithms and for computing exponentials suitable for combinational circuit implementation will be proposed [TAKAA8501] [TAKAA8606] [TAKAY8402] [TAKAY8601b]. The former two are based on the CORDIC method and the latter two are based on the STL method. They all use the redundant binary representation for the internal computation.

In the next section, hardware algorithms for computing sines and cosines and for computing arctangents based on the CORDIC method will be proposed. In Section 6.3, hardware algorithms for computing logarithms and for computing exponentials based on the STL method will be proposed. Some further discussions will be made in Section 6.4.

6.2 Hardware Algorithms Based on the CORDIC Method

6.2.1 Principle of the CORDIC Method

Let us consider a right-angled triangle OPP' on the X-Y plane, as shown in Fig. 6-1, where O is the origin of coordinate axes, the angle OPP' is a right angle and the angle POP' is $arctan2^{-k}$ (rad). ($arctan$R denotes the arctangent of R.) Let the coordinates of P and those of P' be (X,Y) and (X',Y'), respectively. Then, $X'=X-2^{-k}\cdot Y$ and $Y'=Y+2^{-k}\cdot X$ hold. These equations can be considered to represent a movement of a point from P to P', where the vector OP' can be obtained by rotating OP by $arctan2^{-k}$ (rad) and extending it by $(1+2^{-2k})^{1/2}$ $(=1/cos(arctan2^{-k}))$ times. (cosR denotes the cosine of R.)

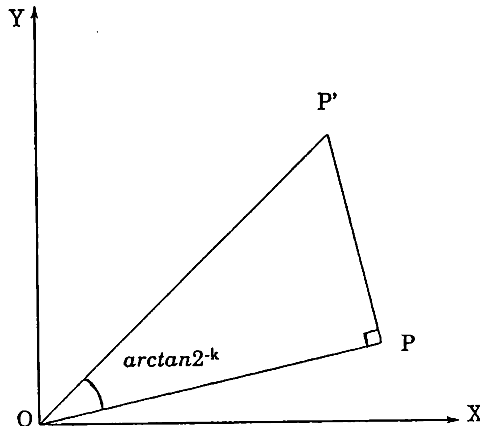

Fig. 6-1 A rotation by the angle of $arctan2^{-k}$

In the computation of the sine and the cosine of U (rad) by the conventional CORDIC method, successive movements of a point from P_0 to P_n via P_1, P_2, and so on, as shown in Fig. 6-2 are considered. A_j (the angle of $P_{j-1}OP_j$) is selected from $\{arctan2^{-j}, -arctan2^{-j}\}$. (A positive and a negative rotation are defined as a counterclockwise and a clockwise rotation, respectively.) Z_j which denotes the remaining angle is introduced. The coordinates of P_0 (X_0, Y_0) and Z_0 are let be (1/K',0) and U, respectively, where K' is the magnifying factor. The computation proceeds according to the following equations. ((X_j, Y_j) is the coordinates of P_j.)

$$X_j = X_{j-1} - q_j \cdot 2^{-j} \cdot Y_{j-1}$$

$$Y_j = Y_{j-1} + q_j \cdot 2^{-j} \cdot X_{j-1}$$

$$Z_j = Z_{j-1} - q_j \cdot arctan2^{-j}$$

q_j is selected from $\{\bar{1}, 1\}$, so that Z_n approaches to 0. (q_j denotes the direction of the j-th rotation.) Consequently, X_n and Y_n are close to *cos*U and *sin*U, respectively. (*sin*R denotes the sine of R.) $arctan2^{-j}$'s are constants and are prepared. The computation of the equations can be done by shift, addition / subtraction and recall of the prepared constant. Since vector OP_j is $(1+2^{-2j})^{1/2}$ times as long as vector OP_{j-1}, X_0 is let be 1/K' in order that the length of OP_n becomes 1. Note that OP_n is K' times as long as vector OP_0, irrespective of the directions of rotations in the computation.

In the computation of the arctangent of V based on the conventional CORDIC method, successive movements of a point from P_1 to P_n as shown in Fig. 6-3 are considered. (X_1, Y_1) and Z_1 are

let be (1,V) and 0, respectively, and the computation proceeds according to the same equations as the case of the sine-cosine computation. q_j is selected from $\{1, \bar{1}\}$, so that Y_n approaches to 0. Consequently, Z_n is close to *arctan*V. Note that the length of the vector can be left out of consideration.

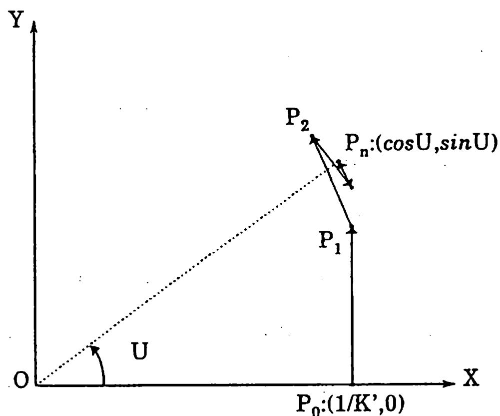

Fig. 6-2 Calculation of a sine and a cosine by the CORDIC method

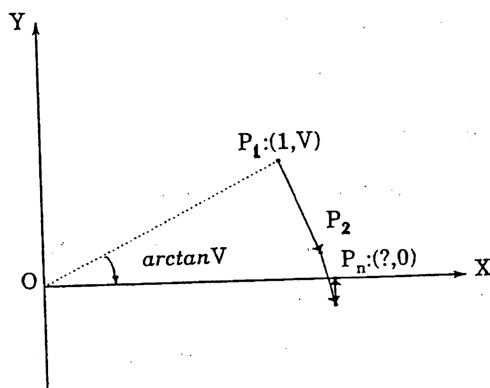

Fig. 6-3 Calculation of an arctangent by the CORDIC method

6.2.2 A Hardware Algorithm for Computing Sines and Cosines

For any real number R, *sin*R and *cos*R can easily be computed from *sin*U and *cos*U, where $0 \leqq U \leqq \pi/4$ [HITAC7807]. Hence, we consider the computation of the sine and the cosine of U, where the operand U (rad) is an n-bit unsigned binary number and satisfies $0 \leqq U \leqq \pi/4$. We compute *sin*U and *cos*U down to the (n-1)st binary position. Since $0 \leqq U \leqq \pi/4$, $0 \leqq \sin U \leqq 1/\sqrt{2}$ and $1 \geqq \cos U \geqq 1/\sqrt{2}$.

In the proposed algorithm, non-rotation as well as a positive and a negative rotation is introduced, and each rotation is performed by a combination of two sub-rotations. A negative rotation, non-rotation and a positive rotation are performed by two negative sub-rotations, one negative and one positive sub-rotation and two positive sub-rotations, respectively. Each sub-rotation is a rotation-extension operation mentioned in the previous subsection. Note that the vector length is increased by the same times, irrespective of the direction of a rotation.

The algorithm is described by the following iteration equations.

$$X_j = X_{j-1} - q_j \cdot 2^{-j} \cdot Y_{j-1} - p_j \cdot 2^{-2j-2} \cdot X_{j-1}$$

$$Y_j = Y_{j-1} + q_j \cdot 2^{-j} \cdot X_{j-1} - p_j \cdot 2^{-2j-2} \cdot Y_{j-1}$$

$$Z_j = Z_{j-1} - q_j \cdot 2 \cdot \arctan 2^{-j-1}$$

Each X_j and Y_j are represented by redundant binary numbers with 1-digit integer part, and each Z_j is represented by one whose most significant digit is located at the j-th binary position. The calculation of the iteration equations is performed in the redundant binary number system. (The above equations are obtained by combining two sets of equations which describe the two sub-

rotations with the angle $arctan 2^{-j-1}$.)

The algorithm is as follows.

Algorithm [SINCOS]

<Input>

U : an operand $(0 \leqq U \leqq \pi/4)$

(an n-bit binary number)

<Output>

X and Y: the cosine and the sine, respectively

(n-bit binary numbers)

<Algorithm>

Step 1: $X_0 := 1/K \quad (K = \prod_{j=1}^{n-1}(1+2^{-2j-2}))$

$Y_0 := 0$

$Z_0 := U$

Step 2: for j:=1 to n-1 do

begin

$$(q_j, p_j) := \begin{cases} (\bar{1}, 1) & \text{if } [z_{j-1}^{j-1} z_j^{j-1} z_{j+1}^{j-1}]_{SD2} < 0 \\ (0, \bar{1}) & \text{if } [z_{j-1}^{j-1} z_j^{j-1} z_{j+1}^{j-1}]_{SD2} = 0 \\ (1, 1) & \text{if } [z_{j-1}^{j-1} z_j^{j-1} z_{j+1}^{j-1}]_{SD2} > 0 \end{cases}$$

$X_j := X_{j-1} - q_j \cdot 2^{-j} \cdot Y_{j-1} - p_j \cdot 2^{-2j-2} \cdot X_{j-1}$

$Y_j := Y_{j-1} + q_j \cdot 2^{-j} \cdot X_{j-1} - p_j \cdot 2^{-2j-2} \cdot Y_{j-1}$

$Z_j := Z_{j-1} - q_j \cdot 2 \cdot arctan 2^{-j-1}$

(redundant binary addition / subtraction)

end

Step 3: $X \leftarrow X_{n-1}$

$Y \leftarrow Y_{n-1}$

(redundant binary to binary conversion) □

In Step 1, X_0, Y_0 and Z_0 are let be $1/K$ ($K=\prod_{j=1}^{n-1}(1+2^{-2j-2})$), 0 and U, respectively. (Since vector OP_j is $(1+2^{-2j-2})$ times as long as vector OP_{j-1}, we let X_0 be $1/K$.)

In Step 2, (q_j, p_j) is selected from $\{(\bar{1},1),(0,\bar{1}),(1,1)\}$ by evaluating the most significant three digits of Z_{j-1}, i.e., z^{j-1}_{j-1}, z^{j-1}_{j} and z^{j-1}_{j+1}. q_j denotes the direction of the j-th rotation. The calculation of the iteration equations is performed in the redundant binary number system. $2\cdot\arctan 2^{-j-1}$ is a constant and is prepared. In Step 2, these computations are performed n-1 times.

It can be proved by induction over j that each Z_j satisfies $-2\cdot\arctan 2^{-j-1}<Z_j<2\cdot\arctan 2^{-j-1}$. (In the proof, the fact that $0<\arctan 2^{-j}<2\cdot\arctan 2^{-j-1}<2^{-j}$ is used.) Hence, $-2^{-j}<Z_j<2^{-j}$ holds and Z_j can be represented by a redundant binary number whose most significant digit is located at the j-th binary position. In order to let the digit at the (j-1)st binary position of each Z_j be 0, in the addition (subtraction) to obtain Z_j in Step 2, a special computation rule has to be applied at the (j-1)st and the j-th binary position. Indeed, there exists such a rule.

In Step 3, the cosine X_{n-1} and the sine Y_{n-1} are converted into the equivalent unsigned binary numbers X and Y, respectively.

Computing a sine and a cosine according to the algorithm, the following theorem holds.

[Theorem 6.1]

When the rounding off errors in the computation are not considered, the errors of the obtained sine Y from sinU and the obtained cosine X from cosU are both smaller than 2^{-n+1}. Namely, $|Y-sinU|<2^{-n+1}$ and $|X-cosU|<2^{-n+1}$ hold.

<Proof>

$|Y-sinU|$ and $|X-cosU|$ are less than the length of the arc of a circle with a radius of 1 and an angle of $|Z_{n-1}|$. Since the length of the arc is $1\cdot|Z_{n-1}|$ and $|Z_{n-1}|<2^{-n+1}$, $|Y-sinU|<2^{-n+1}$ and $|X-cosU|<2^{-n+1}$ hold.

Q.E.D.

Considering the rounding off errors in the computation in Step 2, in order to calculate the sine and the cosine down to the (n-1)st binary digit with the computational error smaller than 2^{-n+1}, X_j's, Y_j's and Z_j's have to be calculated down to more than n-th binary position. Since the numbers of additions to obtain X_{n-1}, Y_{n-1} and Z_{n-1} are about 2n, 2n and n, respectively, they should be calculated down to about $(n+\lceil \log_2 n\rceil)$th binary digit.

Fig. 6-4 shows an example of the computation of a sine and a cosine in accordance with the algorithm. ($2\cdot arctan2^{-j-1}$ $(=2^{-j}-2^{-3j+2}/3+\cdots)$ is represented by a redundant binary number with the most significant digit of 1 located at the j-th binary position and nonpositive digits at the less significant positions.)

$U=[0.0110010]_2$

	Z_j	X_j	Y_j
	$\underline{0.01}10010$	0.1110110000	0.0000000000
$(q_1,p_1)=(1,1)$	$-\ 100000\bar{1}0\bar{1}0$	$-\ 0111011$	$+\ 0111011000$
	$\underline{00\bar{1}}010\bar{1}010$	$0.111000\bar{1}0\bar{1}\bar{1}$	0.0111011000
$(q_2,p_2)=(\bar{1},1)$	$+\ 10000000\bar{1}$	$+\ 001110110$	$-\ 0111000\bar{1}0$
		$-\ 01110$	$-\ 00111$
	$\underline{010}1\bar{1}1001$	$1.0000\bar{1}00\bar{1}1\bar{1}$	$0.0100\bar{1}11\bar{1}0\bar{1}$
$(q_3,p_3)=(1,1)$	$-\ 10000000$	$-\ 00100\bar{1}11$	$+\ 10000\bar{1}00$
		$-\ 100$	$-\ 001$
	$\underline{001}\bar{1}1001$	$1.000\bar{1}00\bar{1}1\bar{1}0$	$0.10\bar{1}00\bar{1}00\bar{1}0$
$(q_4,p_4)=(1,1)$	$-\ 1000000$	$-\ 010\bar{1}00\bar{1}$	$+\ 1000\bar{1}00$
		$-\ 1$	$-\ 0$
	$\underline{\bar{1}1\bar{1}}1001$	$1.00\bar{1}1\bar{1}001\bar{1}0$	$1.\bar{1}00\bar{1}0\bar{1}\bar{1}1\bar{1}0$
$(q_5,p_5)=(\bar{1},1)$	$+\ 100000$	$+\ 1\bar{1}00\bar{1}0$	$-\ 100\bar{1}1\bar{1}$
	$\underline{0\bar{1}1}001$	$1.00\bar{1}10\bar{1}0000$	$1.\bar{1}0\bar{1}0010\bar{1}1\bar{1}$
$(q_6,p_6)=(\bar{1},1)$	$+\ 10000$	$+\ 1\bar{1}0\bar{1}0$	$-\ 100\bar{1}1$
	$\underline{010}01$	$1.000\bar{1}00\bar{1}0\bar{1}0$	$1.\bar{1}0\bar{1}00000\bar{1}0$
$(q_7,p_7)=(1,1)$		$-\ 1\bar{1}0\bar{1}$	$+\ 1000$
		$1.000\bar{1}0\bar{1}010\bar{1}$	$1.\bar{1}0\bar{1}001\bar{1}0\bar{1}0$
		$\Downarrow$	$\Downarrow$
		$X=[0.1110110]_2$	$Y=[0.0110001]_2$
		$(\cos U)$	$(\sin U)$

Fig. 6-4 An example of computation of the sine and the cosine according to Algorithm [SINCOS]

The computation time for Step 1 is constant independent of n. In Step 2, the determination of each (q_j, p_j) can be done in a constant time independent of n. The calculation of the iteration equations can also be performed in a constant time independent of n. Since these computations are performed n-1 times, the computation time required for Step 2 is proportional to n. The conversion in Step 3 can be performed in a computation time proportional to $\log n$ or n.

Thus, we conclude that n-bit sine-cosine computation can be performed in a time proportional to n. The required gate count is proportional to n^2. Namely, the depth of a sine-cosine computing circuit based on the algorithm is $O(n)$, and the gate count of it is $O(n^2)$. The circuit is composed of a few type basic cells. However, because of wires for shifting X_{j-1} and Y_{j-1} by j or 2j-2 positions in Step 2, the chip area of it is $O(n^3)$.

There is an excellent technique to improve the computation speed and reduce the amount of hardware of a sine-cosine computing circuit based on the proposed algorithm. According to the algorithm, Z_j $(-2^{-j}<Z_j<2^{-j})$ denotes the remaining angle after the j-th rotation. Therefore, when we let K be $\prod_{k=1}^{j-1}(1+2^{-2k-2})$, the following equations hold.

$$\cos U = X_j \cdot \cos Z_j - Y_j \cdot \sin Z_j$$

$$\sin U = Y_j \cdot \cos Z_j + X_j \cdot \sin Z_j$$

For a j such that $j>(n-1)/2$, since $\cos Z_j$ $(=1-Z_j^2/2+\cdots)$ and $\sin Z_j$ $(=Z_j-Z_j^3/3+\cdots)$ are close to 1 and Z_j, respectively, the above equations can be rewritten as follows.

$$\cos U = X_j - Y_j \cdot Z_j$$

$sin U = Y_j + X_j \cdot Z_j$

Hence, Step 2 of the algorithm can be rewritten as follows.

Step 2-1 Do the computation of Step 2 for j=1 to n/2, and obtain $X_{n/2}$, $Y_{n/2}$ and $Z_{n/2}$.

Step 2-2 $X_n := X_{n/2} - Y_{n/2} \cdot Z_{n/2}$

$Y_n := Y_{n/2} + X_{n/2} \cdot Z_{n/2}$

K must be $\prod_{j=1}^{n/2}(1+2^{-2j-2})$, in Step 1. The computations for j>n/2 in Step 2 are replaced by two multiplications, a subtraction and an addition, and therefore, the computation speed is improved and the amount of hardware is reduced. (Note that $Z_{n/2}$ has only n/2+1 digits.) The computation speed is further improved and the amount of hardware is further reduced, when the multiplication in Step 2-2 is performed by means of a redundant binary adder tree as in the multiplication algorithm proposed in Chapter 3 and the multiplier recoding technique proposed in Section 3.4 is used.

Recently, the author and his colleagues have developed an improved algorithm, in which only one rotation-extension operation is required in every step and an extra rotation-extension is required in every m steps where m can be an arbitrary constant [ASADT8703b]. By the improved algorithm, sines and cosines can be obtained in shorter time with smaller amount of hardware than by Algorithm [SINCOS]. The speeding up technique stated above can also be applied to the improved algorithm.

6.2.3 A Hardware Algorithm for Computing Arctangents

For a real number R, $\mathit{arctan}R$ can be obtained by rather easy computation from $\mathit{arctan}V$, where $0 \leq V \leq 2-\sqrt{3}$ [HITAC7807]. Hence, in this section, we consider the computation of the arctangent of V, where the operand V is an n-bit unsigned binary number and satisfies $0 \leq V \leq 2-\sqrt{3}$. We compute the arctangent of V down to the (n-1)st binary position. Since $0 \leq V \leq 2-\sqrt{3}$, $0 \leq \mathit{arctan}V \leq \pi/12$.

In the proposed algorithm, as well as in the conventional CORDIC method, X_1, Y_1 and Z_1 are let be 1, V and 0, respectively, and the computation is forwarded according to the following equations.

$$X_j = X_{j-1} - q_j \cdot 2^{-j} \cdot Y_{j-1}$$

$$Y_j = Y_{j-1} + q_j \cdot 2^{-j} \cdot X_{j-1}$$

$$Z_j = Z_{j-1} + q_j \cdot \mathit{arctan} 2^{-j}$$

Each X_j is represented by a redundant binary number with 1-digit integer part, each Y_j is represented by one whose most significant digit is located at the j-th binary position, and each Z_j is represented by one without integer part. The calculation of the iteration equations is performed in the redundant binary number system.

The algorithm is as follows.

Algorithm [ATAN]

<Input>

V : an operand $(0 \leq V \leq 2-\sqrt{3})$

(an n-bit binary number)

<Output>

Z : the arctangent

(an n-bit binary number)

<Algorithm>

Step 1: $X_1 := 1 \ (= [1.0 \cdots 0]_{SD2})$

$Y_1 := V$

$Z_1 := 0$

Step 2: for $j := 2$ to $n-1$ do

begin

$$q_j := \begin{cases} 1 & \text{if } [y_{j-1}^{j-1} y_j^{j-1} y_{j+1}^{j-1}]_{SD2} < -1 \\ 0 & \text{if } [y_{j-1}^{j-1} y_j^{j-1} y_{j+1}^{j-1}]_{SD2} = -1 \text{ or } 0 \text{ or } 1 \\ \bar{1} & \text{if } [y_{j-1}^{j-1} y_j^{j-1} y_{j+1}^{j-1}]_{SD2} > 1 \end{cases}$$

$X_j := X_{j-1} - q_j \cdot 2^{-j} \cdot Y_{j-1}$

$Y_j := Y_{j-1} + q_j \cdot 2^{-j} \cdot X_{j-1}$

$Z_j := Z_{j-1} + q_j \cdot \boldsymbol{arctan} 2^{-j}$

(redundant binary addition / subtraction)

end

Step 3: $Z \leftarrow Z_{n-1}$

(redundant binary to binary conversion) □

In Step 1, X_1, Y_1 and Z_1 are let be 1, V and 0, respectively. In Step 2, q_j is selected from $\{1, 0, \bar{1}\}$ by evaluating the most significant three digits of Y_{j-1}. The calculation of the iteration equations is performed in the redundant binary number system. $\boldsymbol{arctan} 2^{-j}$ is a constant and is prepared. It can be proved by induction over j that $1 \leq X_j \leq 1 + \sum_{k=2}^{j} 2^{-2k+1}$ and $-2^{-j} < Y_j < 2^{-j}$ hold. Therefore, X_j can be

represented by a redundant binary number with 1-digit integer part and Y_j can be represented by one whose most significant digit is located at the j-th binary position. In Step 3, the arctangent Z_{n-1} is converted into the equivalent unsigned binary number Z.

Computing an arctangent according to the algorithm, the following theorem holds.

[Theorem 6.2]

When the rounding off errors in the computation are not considered, the error of the obtained arctangent Z from $\mathit{arctan}V$ is smaller than 2^{-n+1}. Namely, $|Z-\mathit{arctan}V|<2^{-n+1}$ holds.

<Proof>

$$|Z-\mathit{arctan}V|=\mathit{arctan}(|Y_{n-1}|/X_{n-1})<\mathit{arctan}2^{-n+1}<2^{-n+1}$$

Q.E.D.

Considering the rounding off errors in the computation in Step 2, in order to calculate the arctangent down to the (n-1)st binary digit with the computation error smaller than 2^{-n+1}, X_j's Y_j's and Z_j's should be calculated down to about the $(n+\lceil \log_2 n \rceil)$th binary digit.

Fig. 6-5 shows an example of the computation of an arctangent in accordance with the algorithm. ($\mathit{arctan}2^{-j}$ $(=2^{-j}-2^{-3j}/3+\cdots)$ is represented by a redundant binary number with the most significant digit of 1 located at the j-th binary position and nonpositive digits at the less significant positions.)

According to the algorithm, n-bit arctangent computation can be performed in a time proportional to n with a gate count proportional to n^2. The chip area of an arctangent computing circuit based on the algorithm is $O(n^3)$.

$V=[0.0110101]_2$

	Y_j	X_j	Z_j
	$.\underline{011}0101$	1.0000000000	0.0000000000
$q_2=1$	$-\ 100000000$	$+\ 0110101$	$-\ 100000\bar{1}0\bar{1}$
	$\underline{010}101000$	$1.0010\bar{1}1\bar{1}1\bar{1}0$	$0.0100000\bar{1}0\bar{1}$
$q_3=1$	$-\ 10010\bar{1}1\bar{1}$	$+\ 010101$	$-\ 10000000$
	$\underline{001}00\bar{1}0\bar{1}$	$1.01\bar{1}000000\bar{1}$	$0.10\bar{1}0000\bar{1}0\bar{1}$
$q_4=0$			
	$\underline{010}0\bar{1}0\bar{1}$	$1.01\bar{1}000000\bar{1}$	$0.10\bar{1}0000\bar{1}0\bar{1}$
$q_5=1$	$-\ 101\bar{1}00$	$+\ 01$	$-\ 100000$
	$\underline{00\bar{1}}00\bar{1}$	$1.01\bar{1}0000000$	$1.\bar{1}0\bar{1}1\bar{1}00\bar{1}0\bar{1}$
$q_6=0$			
	$\underline{0\bar{1}0}0\bar{1}$		$1.\bar{1}0\bar{1}1\bar{1}00\bar{1}0\bar{1}$
$q_7=\bar{1}$			$-\ 1000$
			$1.\bar{1}0\bar{1}1\bar{1}\bar{1}1\bar{1}0\bar{1}$
			⇓
			$Z=[0.0110010]_2$
			($arctan V$)

Fig. 6-5 An example of computation of the arctangent according to Algorithm [ATAN]

There is an excellent technique to improve the computation speed and reduce the amount of hardware of an arctangent computing circuit based on the proposed algorithm. According to the algorithm, $\mathit{arctan}V=Z_j+\mathit{arctan}(Y_j/X_j)$ holds. Since $\mathit{arctan}T=T-T^3/3+\cdots$ and $Y_j/X_j<2^{-j}$, $\mathit{arctan}(Y_j/X_j)$ is close to Y_j/X_j, for j's such that $j>(n-1)/3$. Hence, we can rewrite Step 2 of the algorithm as follows.

Step 2-1 Do the computation of Step 2 for $j=2$ to $n/3$, and obtain $X_{n/3}$, $Y_{n/3}$ and $Z_{n/3}$.

Step 2-2 $X_n:=Z_{n/3}+Y_{n/3}/X_{n/3}$

The computations for $j>n/3$ in Step 2 are replaced by a division and an addition.

6.3 Hardware Algorithms Based on the STL Method

6.3.1 Principle of the STL Method

For any P, Q (>0) and A (>0), $P+\mathit{ln}Q=(P-\mathit{ln}A)+\mathit{ln}(Q\cdot A)$ holds. ($\mathit{ln}X$ denotes $\log_e X$, where e is Napier's number.) Let $P'=P-\mathit{ln}A$ and $Q'=Q\cdot A$, then $P+\mathit{ln}Q=P'+\mathit{ln}Q'$ holds. In the STL method, the transformation $P_0+\mathit{ln}Q_0=P_1+\mathit{ln}Q_1=\cdots=P_m+\mathit{ln}Q_m$, which can be described by the iteration equations $P_j=P_{j-1}-\mathit{ln}A_j$ and $Q_j=Q_{j-1}\cdot A_j$ is considered. In the computation of $\mathit{ln}X$ for a given X, P_0 and Q_0 are let be 0 and X, respectively, and A_j is selected from $\{1,1-2^{-k}\}$ by evaluating the value of Q_{j-1}, so that Q_m approaches

to 1. Consequently, $0+ln X \cong P_m + ln 1$, i.e., $ln X \cong P_m$ holes, and hence, P_m is close to $ln X$. In the computation of $exp X$ for a given X, P_0 and Q_0 are let be X and 1, respectively, and A_j is selected from $\{1, 1+2^{-k}\}$ by evaluating the value of P_{j-1}, so that P_m approaches to 0. ($exp X$ denotes e^X.) Consequently, $X + ln 1 \cong 0 + ln Q_m$, i.e., $X \cong ln Q_m$ holds, and hence, Q_m is close to $exp X$. In either case, $ln(1+2^{-k})$'s or $ln(1-2^{-k})$'s are constants and are prepared and the computation of iteration equations is performed by shift and addition / subtraction and recall of the prepared constant.

6.3.2 A Hardware Algorithm for Computing Logarithms

For a positive number $R = X \cdot 2^E$, $ln R = ln(X \cdot 2^E) = ln X + E \cdot ln 2$. When R is represented in the basic format of the IEEE standard for binary floating-point arithmetic and the 'exponent' part and the 'significand' part are E and X ($1 \leq X < 2$) respectively, $ln R$ can be obtained by computing $ln X$ and then adding $E \cdot ln 2$ to it. Since multiplication by a constant and addition are easy, we consider the computation of $ln X$, where the operand X is an n-bit unsigned binary number and satisfies $1 \leq X < 2$. We compute $ln X$ down to the (n-1)st binary digit. Since $1 \leq X < 2$, $0 \leq ln X < ln 2$.

The proposed algorithm is described by the following iteration equations, as the case of the conventional STL method.

$$P_j = P_{j-1} - ln A_j$$

$$Q_j = Q_{j-1} \cdot A_j$$

P_0 and Q_0 are let be 0 and X, respectively. A_j is selected by evaluating the value of Q_{j-1} so as to bring Q_j closer to 1. P_j is the j-th approximation of $ln X$. In the algorithm, new variables

R_j's each of which denotes Q_j-1 are introduced. R_j is represented by a redundant binary number whose most significant digit is located at the j-th binary position, and each P_j is represented by one with a 1-digit integer part. A_j is regarded as $1+a_j \cdot 2^{-j}$ and a_j is selected from $\{\bar{1},0,1\}$ by evaluating the most significant three digits of R_{j-1}. The calculation of the iteration equations is performed in the redundant binary number system.

The algorithm is as follows.

Algorithm [LN]

<Input>

X : an operand

(an n-bit normalized binary number)

<Output>

Z : the logarithm to the base e

(an n-bit binary number)

<Algorithm>

Step 1: $R_0 := X-1$ $(=[0.x_1 \cdots x_{n-1}]_{SD2})$

$P_0 := 0$

Step 2: for j:=1 to n-1 do

begin

$$a_j := \begin{cases} 1 & \text{if} \quad [r^{j-1}_{j-1} r^{j-1}_{j} r^{j-1}_{j+1}]_{SD2} < 0 \\ 0 & \text{if} \quad [r^{j-1}_{j-1} r^{j-1}_{j} r^{j-1}_{j+1}]_{SD2} = 0 \text{ or } 1 \\ \bar{1} & \text{if} \quad [r^{j-1}_{j-1} r^{j-1}_{j} r^{j-1}_{j+1}]_{SD2} > 1 \end{cases}$$

$R_j := R_{j-1} + a_j \cdot 2^{-j} \cdot (1+R_{j-1})$

$P_j := P_{j-1} + (-ln(1+a_j \cdot 2^{-j}))$

(redundant binary addition / subtraction)

end

Step 3: $Z \leftarrow P_{n-1}$

(redundant binary to binary conversion) □

In Step 1, R_0 and P_0 are let be X-1 and 0, respectively. In Step 2, a_j is selected by evaluating the most significant three digits of R_{j-1}. The calculation of the iteration equations is performed in the redundant binary number system. $-ln(1-2^{-j})$ and $-ln(1+2^{-j})$ are constants and are prepared. It can be proved by induction over j that each R_j satisfies $-2^{-j}+2^{-j-2}<R_j<2^{-j}$. Therefore, R_j can be represented by a redundant binary number whose most significant digit is located at the j-th binary position. In Step 3, the logarithm P_{n-1} is converted into the equivalent unsigned binary number Z.

Computing a logarithm according to the algorithm, the following theorem holds.

[Theorem 6.3]

When the rounding off errors in the computation are not considered, the error of the obtained logarithm Z from lnX is smaller than 2^{-n+1}. Namely, $|Z-lnX|<2^{-n+1}$ holds.

<Proof>

$Z=P_{n-1}$, $0+lnX=P_{n-1}+ln(1+R_{n-1})$, $-2^{-n+1}+2^{-n-1}<R_{n-1}<2^{-n+1}$

$\therefore Z-lnX=-ln(1+R_{n-1})$

$\therefore -2^{-n+1}+2^{-2n+1}-\cdots<Z-lnX<2^{-n+1}-2^{-n-1}+(2^{-n+1}-2^{-n-1})^2/2+\cdots$

Hence, $|Z-lnX|<2^{-n+1}$ holds.

Q.E.D.

Considering the rounding off errors in the computation in Step 2, in order to calculate the logarithm down to the (n-1)st binary digit with the computational error smaller than 2^{-n+1}, R_j's and P_j's should be calculated down to about the $(n+\lceil \log_2 n \rceil)$th binary digit.

Fig. 6-6 shows an example of the computation of a logarithm in accordance with the algorithm. ($-ln(1-2^{-j})$ $(=2^{-j}+2^{-2j-1}+\cdots)$ is represented by a redundant binary number with the most significant digit of 1 at the j-th binary position and nonnegative digits at the less significant positions. $-ln(1+2^{-j-1})$ $(=-2^{-j}+2^{-2j-1}-\cdots)$ is represented by one with the most significant digit of $\bar{1}$ at the j-th binary position and nonnegative digits at the less significant positions.)

According to the algorithm, n-bit logarithm computation can be performed in a time proportional to n with a gate count proportional to n^2. The chip area of a logarithm computing circuit based on the algorithm is $O(n^3)$.

There is a very excellent technique to improve the computation speed and reduce the amount of hardware of an logarithm computing circuit based on the proposed algorithm. According to the algorithm, $lnX=P_j+ln(1+R_j)$ holds. For a j such that $j>(n-1)/2$, since $ln(1+R_j)$ $(=R_j-R_j{}^2/2+\cdots)$ is close to R_j, $lnX=P_j+R_j$ holds. Hence, Step 2 of the algorithm can be rewritten as follows.

Step 2-1 Do the computation of Step 2 for j=1 to n/2, and obtain $R_{n/2}$ and $P_{n/2}$.

Step 2-2 $P_{n-1} := P_{n/2} + R_{n/2}$

The computations for j>n/2 in Step 2 are replaced by an addition, and therefore, the computation speed is much improved and the amount of hardware is much reduced.

$X=[1.0110101]_2$

	R_j	P_j
	$\underline{0.01}10101$	0.0000000000
$a_1=0$		
	$\underline{011}0101000$	0.0000000000
$a_2=\bar{1}$	$-\ 101101010$	$+\ 100100110$
	$\underline{001}00000\bar{1}0$	$0.1\bar{1}01\bar{1}010\bar{1}0$
$a_3=0$		
	$\underline{010}000\bar{1}0$	$0.1\bar{1}01\bar{1}010\bar{1}0$
$a_4=\bar{1}$	$-\ 1\ \ 0100$	$+\ 1000010$
	$\underline{000}\bar{1}1\bar{1}0$	$1.\bar{1}\bar{1}10\bar{1}1\bar{1}000$
$a_5=0$		
	$\underline{00\bar{1}}1\bar{1}0$	$1.\bar{1}\bar{1}10\bar{1}1\bar{1}000$
$a_6=1$	$+\ 1$	$+\ \bar{1}0000$
	$\underline{1\bar{1}1}\bar{1}0$	$0.10\bar{1}\bar{1}10\bar{1}000$
$a_7=\bar{1}$		$+\ 1000$
		$1.\bar{1}\bar{1}10\bar{1}00000$
		$\Downarrow$
		$Z=[0.0101100]_2$
		$(\ln X)$

Fig. 6-6 An example of computation of the logarithm according to Algorithm [LN]

6.3.3 A Hardware Algorithm for Computing Exponentials

For a real number $R=E\cdot ln2+X$, $expR=exp(E\cdot ln2+X)=2^{E}\cdot expX$. Therefore, we can compute $expR$ for a given binary number R by dividing R by $ln2$ to obtain the quotient E and the remainder X ($0\leq X<ln2$), computing $expX$, and then shifting it E bits. Since $0\leq X<ln2$, $1\leq expX<2$. If $expR$ will be represented in the basic format of the IEEE standard, the 'exponent' part is E and the 'significand' part is $expX$. Since division by a constant is easy, we consider the computation of $expX$, where the operand X is an n-bit unsigned binary number and satisfies $0\leq X<ln2$. We compute $expX$ down to the (n-1)st binary digit.

The algorithm is described by the following iteration equations, as the case of the conventional STL method.

$$P_j=P_{j-1}-lnA_j$$

$$Q_j=Q_{j-1}\cdot A_j$$

P_0 and Q_0 are let be X and 1, respectively. A_j is selected by evaluating the value of P_{j-1} so as to bring P_j closer to 0. Q_j is the j-th approximation of $expX$. In the algorithm, each P_j is represented by a redundant binary number whose most significant digit is located at the j-th binary position, and each Q_j is represented by one with 2-digit integer part. A_j is regarded as $1+a_j\cdot 2^{-j}$ and a_j is selected from $\{\bar{1},0,1\}$ by evaluating the most significant three digits of P_{j-1}. The calculation of the iteration equations is performed in the redundant binary number system.

The algorithm is as follows.

Algorithm [EXP]

<Input>

X : an operand $(0 \leq X < \ln 2)$

(an n-bit binary number)

<Output>

Z : the exponential

(an n-bit normalized binary number)

<Algorithm>

Step 1: $P_0 := X$

$Q_0 := 1 \ (= [1\bar{1}.0]_{SD2})$

Step 2: for j:=1 to n-1 do

begin

$$a_j := \begin{cases} \bar{1} & \text{if } [p^{j-1}_{j-1} p^{j-1}_{j} p^{j-1}_{j+1}]_{SD2} < -1 \\ 0 & \text{if } [p^{j-1}_{j-1} p^{j-1}_{j} p^{j-1}_{j+1}]_{SD2} = -1 \text{ or } 0 \\ 1 & \text{if } [p^{j-1}_{j-1} p^{j-1}_{j} p^{j-1}_{j+1}]_{SD2} > 0 \end{cases}$$

$P_j := P_{j-1} + (-\ln(1 + a_j \cdot 2^{-j}))$

$Q_j := Q_{j-1} + a_j \cdot 2^{-j} \cdot Q_{j-1}$

(redundant binary addition / subtraction)

end

Step 3: $Z \leftarrow Q_{n-1}$

(redundant binary to binary conversion) □

In Step 1, P_0 and Q_0 are let be X and 1 $(=[1\bar{1}.0]_{SD2})$, respectively. In Step 2, a_j is selected by evaluating the value of the most significant three digits of P_{j-1}. The calculation of the iteration equations is performed in the redundant binary number system. $-\ln(1-2^{-j})$ and $-\ln(1+2^{-j})$ are constants and are

prepared. It can be proved by induction over j that each P_j satisfies $-2^{-j}<P_j<2^{-j}-2^{-j-2}$. Therefore, P_j can be represented by a redundant binary number whose most significant digit is located at the j-th binary position. In Step 3, the exponential Q_n is converted into the equivalent unsigned binary number Z.

Computing an exponential according to the algorithm, the following theorem holds.

[Theorem 6.4]

When the rounding off errors in the computation are not considered, the error of the obtained exponential Z from $exp X$ is smaller than $2^{-n+1}+2^{-2n+2}$. Namely, $|Z-exp X|<2^{-n+1}+2^{-2n+2}$ holds.

<Proof>

$Z=Q_{n-1}$, $X+0=P_{n-1}+ln Q_{n-1}$, $-2^{-n+1}<P_{n-1}<2^{-n+1}-2^{-n-1}$

$\therefore\ exp(X-2^{-n+1}+2^{-n-1})<Z<exp(X+2^{-n+1})$

$\therefore\ exp(-2^{-n+1}+2^{-n-1})-1<(Z-exp X)/exp X<exp(2^{-n+1})-1$

$\therefore\ (-2^{-n+1}+2^{-n-1})+(-2^{-n+1}+2^{-n-1})^2/2+\cdots$

$<(Z-exp X)/exp X<2^{-n+1}+2^{-2n+1}+\cdots$

Hence, $|Z-exp X|/exp X<2^{-n+1}+2^{-2n+2}$ holds.

Since $1\leq exp X<2$, $|Z-exp X|<2^{-n+1}+2^{-2n+2}$ holds.

Q.E.D.

Considering the rounding off errors in the computation in Step 2, in order to calculate the exponential down to the (n-1)st binary digit with the computational error smaller than 2^{-n+1}, P_j's and Q_j's should be calculated down to about the

$(n+\lceil \log_2 n \rceil)$th binary digit.

Fig. 6-7 shows an example of the computation of an exponential in accordance with the algorithm. ($-\ln(1+2^{-j})$ and $-\ln(1-2^{-j})$ are represented in the same way as in Fig. 6-6.)

According to the algorithm, n-bit exponential computation can be performed in a time proportional to n with a gate count proportional to n^2. The chip area of an exponential computing circuit based on the algorithm is $O(n^3)$.

$X=[0.0101100]_2$

	P_j	Q_j
	$\underline{0.01}01100$	$1\bar{1}.0$
$a_1=1$	$+\ \bar{1}001100001$	$+\ 1.\bar{1}0$
	$\underline{0\bar{1}1}1000001$	$10.\bar{1}0$
$a_2=0$		
	$\underline{\bar{1}1}1000001$	$10.\bar{1}000$
$a_3=0$		
	$\underline{\bar{1}10}00001$	$10.\bar{1}000000$
$a_4=\bar{1}$	$+\ 1000010$	$-\ 10\bar{1}00000$
	$\underline{000}010\bar{1}$	$10.\bar{1}0\bar{1}1\bar{1}00000$
$a_5=0$		
	$\underline{000}10\bar{1}$	$10.\bar{1}0\bar{1}1\bar{1}00000$
$a_6=0$		
	$\underline{0010}\bar{1}$	$10.\bar{1}0\bar{1}1\bar{1}00000$
$a_7=1$		$+\ 10\bar{1}0\bar{1}$
		$10.\bar{1}0\bar{1}10\bar{1}0\bar{1}0\bar{1}$
		⇓
		$Z=[1.0110101]_2$
		$(expX)$

Fig. 6-7 An example of computation of the exponential according to Algorithm [EXP]

There is an excellent technique to improve the computation speed and reduce the amount of hardware of an exponential computing circuit based on the proposed algorithm. According to the algorithm, $\mathit{exp}X=Q_j \cdot \mathit{exp}P_j$ holds. For a j such that j>(n-1)/2, since $\mathit{exp}P_j$ $(=1+P_j+P_j{}^2/2+\cdots)$ is close to $1+P_j$, $\mathit{exp}X=Q_j \cdot (1+P_j)$ holds. Hence, Step 2 of the algorithm can be rewritten as follows.

Step 2-1 Do the computation of Step 2 for j=1 to n/2, and obtain $P_{n/2}$ and $Q_{n/2}$.

Step 2-2 $Q_{n-1} := Q_{n/2} + Q_{n/2} \cdot P_{n/2}$

The calculations for j>n/2 in Step 2 are replaced by a multiplication and an addition, and therefore, the computation speed is improved and the amount of hardware is reduced.

6.4 Remarks and Discussions

New hardware algorithms for computing sines and cosines, arctangents, logarithms and exponentials with the redundant binary representation have been proposed. The former two are based on the CORDIC method and the latter two are based on the STL method.

Table 6-1 shows a comparison of three sine-cosine computing circuits, i.e., one based on the proposed algorithm, one with ripple-carry adders based on the conventional CORDIC method and

one with carry-look-ahead adders based on the conventional CORDIC method, regarding the depth and the gate count. As shown in the table, the depth of the proposed sine-cosine computing circuit is O(n), which is smaller than those of the other two circuits. The gate counts of three types of circuit are all $O(n^2)$. Comparisons of circuits for the other three functions are the same as that shown in the table.

The CORDIC method can also be applied to the computation of hyperbolic functions [WALT7105]. The author and his colleagues have developed high-speed hardware algorithms for computing sine-hyperbolic and cosine-hyperbolic and for computing arctangent-hyperbolic based on the CORDIC method in which the redundant binary representation is used for internal computation [ASADT8703a].

The proposed two algorithms for computing logarithms and for computing exponentials are based on the same principle and have much in common. We can make an efficient logarithm and exponential computing circuit based on these algorithms.

Table 6-1 A comparison of three types of sine-cosine computing circuit based on the CORDIC methods

	Depth	Gate Count
Proposed One	$O(n)$	$O(n^2)$
with RCA's	$O(n^2)$	$O(n^2)$
with CLA's	$O(n \cdot \log n)$	$O(n^2)$

Chapter 7

Design of Self-Checking Arithmetic Circuits by Means of the Three-Rail Logic

7.1 Introduction

As stated in Chapter 1, arithmetic operations play very important roles in various digital systems. Since many digital systems are required to operate fast with high reliability, not only high-speed operations and regular structures but also fault-tolerant features should be implemented in arithmetic circuits. In this chapter, the design of self-checking, i.e., on-line error-detectable, arithmetic circuits based on the algorithms proposed in previous chapters is considered. The three-rail logic [TAKAY8505a] [TAKAY8505b] is used for the design. The three-rail logic is a logic design technique in which three mutually exclusive conditions calculated in a circuit are encoded in the 1-out-of-3 code and the circuit is designed to be inverter-free.

In this chapter an arithmetic circuit is designed as a combinational circuit composed of AND and OR gates with limited fan-in and inverters. A fault set F of unidirectional stuck-at faults on gate output lines is assumed. Namely, a fault f in F sticks multiple gate output lines at the same value (logical 0 or logical 1), permanently. Note that F includes both stuck-at-0 faults and stuck-at-1 faults. Note also that a single stuck-at

fault is a special case of a unidirectional stuck-at fault. It is also assumed that faults build up with time, and they are denoted by a fault sequence $\langle f_1, f_2, \cdots, f_m \rangle$ where each f_j is a member of F. A combination of faults, $\bigcup_{j=1}^{k} f_j$, is not always a unidirectional stuck-at fault. It is assumed that once a line is stuck at 0 or 1, it remains stuck at that value, and hence, a fault can not change the value of lines which have already been stuck by one of the preceding faults. It is also assumed that faults occur one at a time and between any two faults a sufficient time elapses so that all input vectors are applied to the arithmetic circuit. In other words, since faults do not occur so frequently, all input vectors are applied to the arithmetic circuit between the occurrences of any two faults.

The following terms are defined regarding self-checking features and refer to a combinational circuit G with input code space A, output code space B and an assumed fault set F [ANDEM7303] [SMITM7806]. (In many self-checking circuits, their input and output are encoded in some error checking code. During normal operation, G receives members of A, and produces members of B. Usually, it is assumed that all the members of A are input. It may be a case that only the members of a subset of B are produced, due to the function of G. Members of a code space are called code words. Under faults, noncode words may be produced.)

(1) G is 'fault secure' with respect to F, if for all faults in F and all code word inputs, the output is either correct or is a noncode word, i.e., for all $f \in F$ and for all $x \in A$, $G(x,f)=G(x,\lambda)$ or $G(x,f) \notin B$. (G(x,f) denotes the output of G

under input x and a fault (or a combination of faults) f. $G(x,\lambda)$ is the output under the fault-free condition $f=\lambda$.)

(2) G is 'self-testing' with respect to F, if for all faults in F, there is at least one code word input that produces a noncode word output, i.e., for all $f \in F$, there is an $x \in A$ such that $G(x,f) \notin B$.

(3) G is 'totally self-checking' (TSC) with respect to F, if it is fault secure and self-testing with respect to F.

(4) G is 'code disjoint', if it maps code word inputs into code word outputs and noncode word inputs into noncode word outputs when it is fault-free, i.e., for all $x \in A$, $G(x,\lambda) \in B$ and for all $x \notin A$, $G(x,\lambda) \notin B$.

(5) G is 'strongly fault secure' (SFS) with respect to F, if G is SFS with respect to all fault sequences whose members belong to F. G is SFS with respect to a fault sequence $\langle f_1, f_2, \cdots, f_m \rangle$, if for all $x \in A$, either $G(x, \bigcup_{j=1}^{k} f_j) = G(x,\lambda)$ or $G(x, \bigcup_{j=1}^{k} f_j) \notin B$, where k is the smallest integer for which there is an $x \in A$ such that $G(x, \bigcup_{j=1}^{k} f_j) \neq G(x,\lambda)$. (Let $k=m$ if there is no such k.)

In a SFS circuit, the first erroneous output caused by faults is a noncode output, i.e., the TSC goal is achieved, on the assumption that faults occur one at a time and all code inputs are applied to the circuit between the occurrences of two faults.

In the next section, the design of self-checking arithmetic circuits based on the proposed algorithms will be considered. In

Section 7.3, a logic design of a self-checking multiplier based on the algorithm proposed in Chapter 3 is shown as an example. In Section 7.4, some further discussions will be made.

7.2 Design of Self-Checking Arithmetic Circuits

Fig. 7-1 shows a block diagram of a self-checking arithmetic circuit based on one of the algorithms proposed in previous chapters designed by means of the three-rail logic. The self-checking arithmetic circuit receives an n_I-bit vector as an input and produces an n_O-bit vector and a bit-pair as a data output and an error indicator, respectively. It consists of an input encoder, a functional block and an error checker.

The input encoder receives n_I primary input bits. It encodes each of them in the 1-out-of-2 code, and produces n_I bit-pairs. It is composed of n_I inverters. The remainder part of the arithmetic circuit is designed to be composed of only AND and OR gates, i.e., designed to be inverter-free.

The functional block receives the n_I bit-pairs produced by the input encoder, and produces n_O bit-pairs each of which represents a digit of the result encoded in the 1-out-of-2 code. The final n_O-bit data output of the circuit is directly derived from these bit-pairs. The functional block is designed to be inverter-free by means of the three-rail logic and the two-rail logic [SELLH68]. The functional block consists of a redundant binary calculator and a redundant binary to binary converter. To

apply the three-rail logic, each redundant binary digit which has one of the three values is encoded in the 1-out-of-3 code. The redundant binary calculator which is mainly composed of redundant binary adder/subtracters produces a redundant binary number as the intermediate result whose each digit is encoded in the 1-out-of-3 code. The redundant binary to binary converter converts the intermediate result into the equivalent binary number whose each digit is encoded in the 1-out-of-2 code.

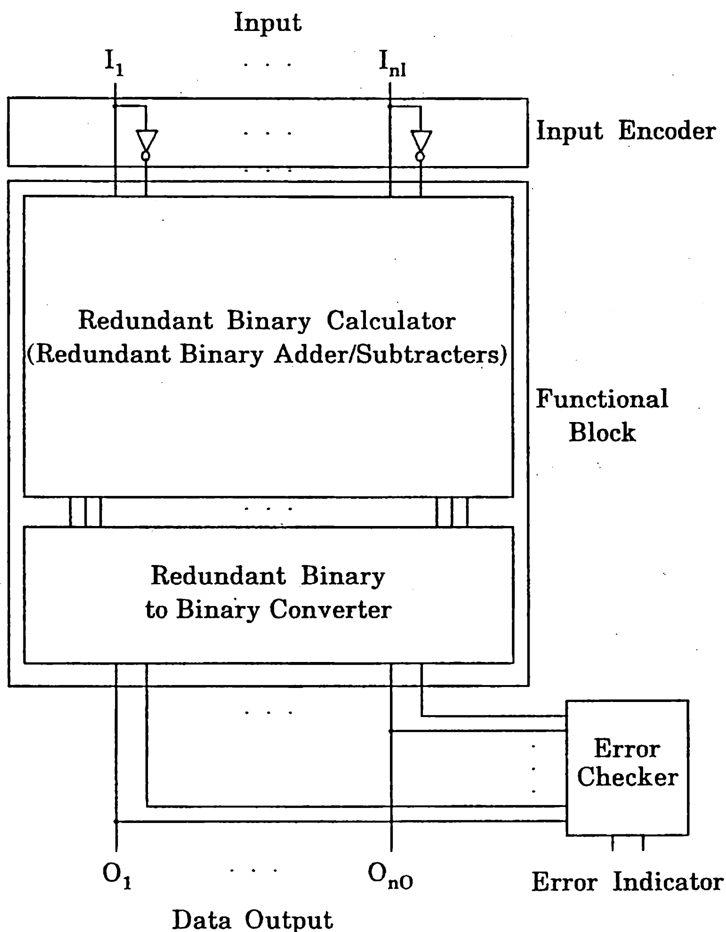

Fig. 7-1 A block diagram of a self-checking arithmetic circuit

The error checker receives the n_o bit-pairs produced by the functional block, and produces a bit-pair as the error indicator. The error indicator indicates nonerror-status when it is either <01> or <10>, and indicates error-status when it is either <00> or <11>. The TSC two-rail checker [WAKE78] is used for the error checker. It maps n_o input bit-pairs into one output bit-pair such that the output bit-pair is in the 1-out-of-2 code if and only if each of the n_o input bit-pairs is in the 1-out-of-2 code.

As will be shown below, the circuit is self-checking with respect to unidirectional stuck-at faults on multiple gate output lines, if the error checker is self-testing for the input code words which are actually input to it.

Any stuck-at fault on the output line of an inverter in the input encoder can be modeled as a combination of stuck-at faults (stuck at the same value as the modeled fault) on input lines of gates in the functional block that are directly connected with the output line of the inverter. Therefore, we consider the circuit consists of the functional block and the error checker, and assume unidirectional stuck-at faults on gate input and/or output lines in it. The circuit never receives a noncode word input. From the fault assumption, it receives all the members of the input code space $\{\langle 01\rangle,\langle 10\rangle\}^{n_I}$, between the occurrences of any two faults.

The functional block is inverter-free. Its output code space is $\{\langle 01\rangle,\langle 10\rangle\}^{n_o}$, and hence, unordered. (A code space is unordered if no member of it covers any other member. A binary vector x covers a binary vector y if x has 1 in every position

where y has 1.) From Theorem 2 and Theorem 4 in [SMITM7806], it is straightforwardly derived that any inverter-free circuit with an unordered output code space is strongly fault secure with respect to unidirectional stuck-at faults on multiple gate input and/or output lines. Hence, the functional block is SFS with respect to unidirectional stuck-at faults on multiple gate input and/or output lines.

The error checker is obviously fault secure with respect to unidirectional stuck-at faults on gate input and/or output lines. It is obviously code disjoint, too. However, it is not obvious that the error checker is self-testing. It receives only the members of a subset of input code space $\{\langle 01\rangle,\langle 10\rangle\}^{n_o}$, because its input, i.e., the output of the functional block, represents a result of the operation. There may exist some n_o-bit numbers which can not be a result of the operation. In order to show that the error checker is self-testing, it must be shown that all the assumed faults in it can be detected by the code words which are actually input to it. Here, we assume that the error checker is self-testing for the input code words which are actually input to it.

Let us consider a fault sequence $\langle f_1, f_2, \cdots, f_m\rangle$ whose each member f_j is a unidirectional stuck-at fault on multiple gate input and/or output lines. Let k be the smallest integer for which there is a stuck line in the error checker and/or there is a code word input such that the functional block produces an incorrect output. Namely, suppose that before the fault f_k occurs, the data output is always correct and the error indicator

always indicates nonerror-status. If there is no such k, the data output of the arithmetic circuit is always correct and the error indicator always indicates nonerror-status for the fault sequence. If there is such a k, after f_k occurs, the following three cases have to be considered; (1) there is no stuck line in the error checker, but there is a code word input such that the functional block produces an incorrect output, (2) the functional block produces a correct code word output for all code input but there is a stuck line in the error checker, and (3) there is a stuck line in the error checker and there is a code word input such that the functional block produces an incorrect output.

In case (1), since the functional block is SFS and error checker is code disjoint, the error indicator indicates nonerror-status whenever the data output is correct and indicates error-status whenever the data output is erroneous. Therefore, the error indicator informs us the occurrence of the first erroneous data output, immediately. Furthermore, the error indicator surely indicates error-status before the next fault f_{k+1} occurs, because there is a code word input such that the functional block produces a noncode word output. Note that if f_{k+1} occurs, there could be an undetectable error caused by $\bigcup_{j=1}^{k+1} f_j$.

In case (2), since the functional block always produces a correct code word and the error checker is TSC for the input code words which are actually input to it (from the assumption), the data output is always correct and the error indicator indicates error-status due to the fault in the error checker before f_{k+1} occurs. Therefore, the data output is always correct and the

error indicator informs us the existence of faults in the circuit before f_{k+1} occurs.

In case (3), since the functional block is SFS, it always produces either a correct code word or a noncode word. Suppose that f_k is a stuck-at-d (either 0 or 1) fault on multiple lines. Then, any noncode word produced by the functional block includes at least one bit-pair whose value is $\langle dd \rangle$ but no bit-pair whose value is $\langle \bar{d}\bar{d} \rangle$. ($\bar{d}$ denotes the logical inverse of d.) Since all the faulty lines in the error checker are stuck at d, it produces the noncode word $\langle dd \rangle$ whenever it receives such a noncode word. Therefore, the error indicator indicates error-status whenever the data output is erroneous. As in case (2), there may also be a case that while the data output is correct, the error indicator indicates error-status due to the fault in the error checker. Hence, the error indicator informs us the existence of faults in the circuit not later than the first erroneous data output is produced. Furthermore, the error indicator surely indicates error-status before f_{k+1} occurs.

Thus, in any case, the error indicator informs us the existence of faults in the circuit not later than the first erroneous data output is produced and before f_{k+1} occurs.

Therefore, the arithmetic circuit is self-checking with respect to unidirectional stuck-at faults on multiple gate output lines, if the error checker is self-testing for the input code words which are actually input to it. Namely, if the error checker is self-testing for the input code words which are actually input to it, any error in the data output and/or the

error indicator caused by unidirectional stuck-at faults on multiple gate output lines is detected in normal operation, by observation of the error indicator.

7.3 A Design of a Self-Checking Multiplier

In this section a logic design of a self-checking multiplier based on the algorithm proposed in Chapter 3, i.e., Algorithm [MUL], by means of the three-rail logic is described [TAKAY8505a] [TAKAY8505b]. The multiplier receives a 2n-bit vector (an n-bit multiplicand and an n-bit multiplier) as an input, and produces a 2n-bit vector (a 2n-bit product) and a bit-pair as a data output and an error indicator, respectively.

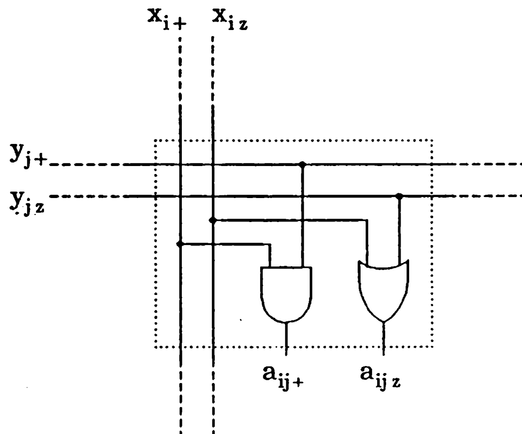

Fig. 7-2 A logic design of the partial product generation cell

The input encoder is composed of 2n inverters. The functional block consists of a redundant binary calculator and a redundant binary to binary converter. The redundant binary calculator consists of n partial product generators and a redundant binary adder tree.

Each partial product generator generates a partial product from a multiplier digit and the multiplicand. Each digit of the multiplicand, the multiplier and the partial products is encoded in the 1-out-of-2 code. An inverter-free partial product generator can easily be designed by means of the two-rail logic. Fig. 7-2 shows a logic design of a partial product generation cell.

The redundant binary adder tree produces the product represented in the redundant binary representation, by adding up the partial products. It is constructed by forming redundant binary adders in a binary tree form. An inverter-free redundant binary adder tree can be designed by means of the three-rail logic. Each redundant binary digit a_i is represented by three bits, a_{i-}, a_{iz}, and a_{i+}, and 100 or 010 or 001 is assigned to $a_{i-}a_{iz}a_{i+}$ accordingly, as a_i is $\bar{1}$ or 0 or 1. Fig. 7-3 (a) and (b) show logic designs of redundant binary addition cells for redundant binary adders at the first level in the adder tree and for adders at the other levels, respectively. The former one is simpler than the latter one, because none of its input digit is $\bar{1}$. A different addition rule from that stated in Section 2.3 is used here [CHOWR7810]. In the first step, the intermediate carry c_i ($\in\{0,1\}$) and the intermediate sum digit d_i ($\in\{\bar{1},0\}$) are

determined at each position, satisfying the equation $2c_i+d_i=a_i+b_i-2t_i+t_{i+1}$, as shown in Table 7-1 (a), where a_i and b_i are the augend and the addend digit, respectively, and t_i ($\in\{\bar{1},0\}$) is determined from only a_i and b_i. In the second step, the sum digit s_i ($\in\{\bar{1},0,1\}$) is determined at each position, by computing d_i+c_{i+1} without generating a carry, as shown in Table 7-1 (b).

Fig. 7-4 shows a block diagram of the redundant binary to binary converter. It is a modification of a carry-look-ahead adder. In the converter, three mutually exclusive conditions, i.e., one for generating a carry, one for propagating a carry and one for neither generating nor propagating a carry, at each position are first calculated, and then the three conditions at groups of adjacent positions are calculated in a binary tree form [UNGE7704] [BRENK8205]. The three conditions at each group (I_g, I_p and I_e, respectively) are encoded in the 1-out-of-3 code. Logic designs of the basic cells of the converter are shown in Fig. 7-5.

The error checker can be constructed by forming basic cells each of which is designed as shown in Fig. 7-6 in a binary tree form.

From the discussion in the previous section, the multiplier is self-checking with respect to unidirectional stuck-at faults on multiple gate output lines, if the error checker is self-testing for the input code words which are actually input to it. It can be shown that the error checker is self-testing for the input code words which are actually input to it, by the fact that

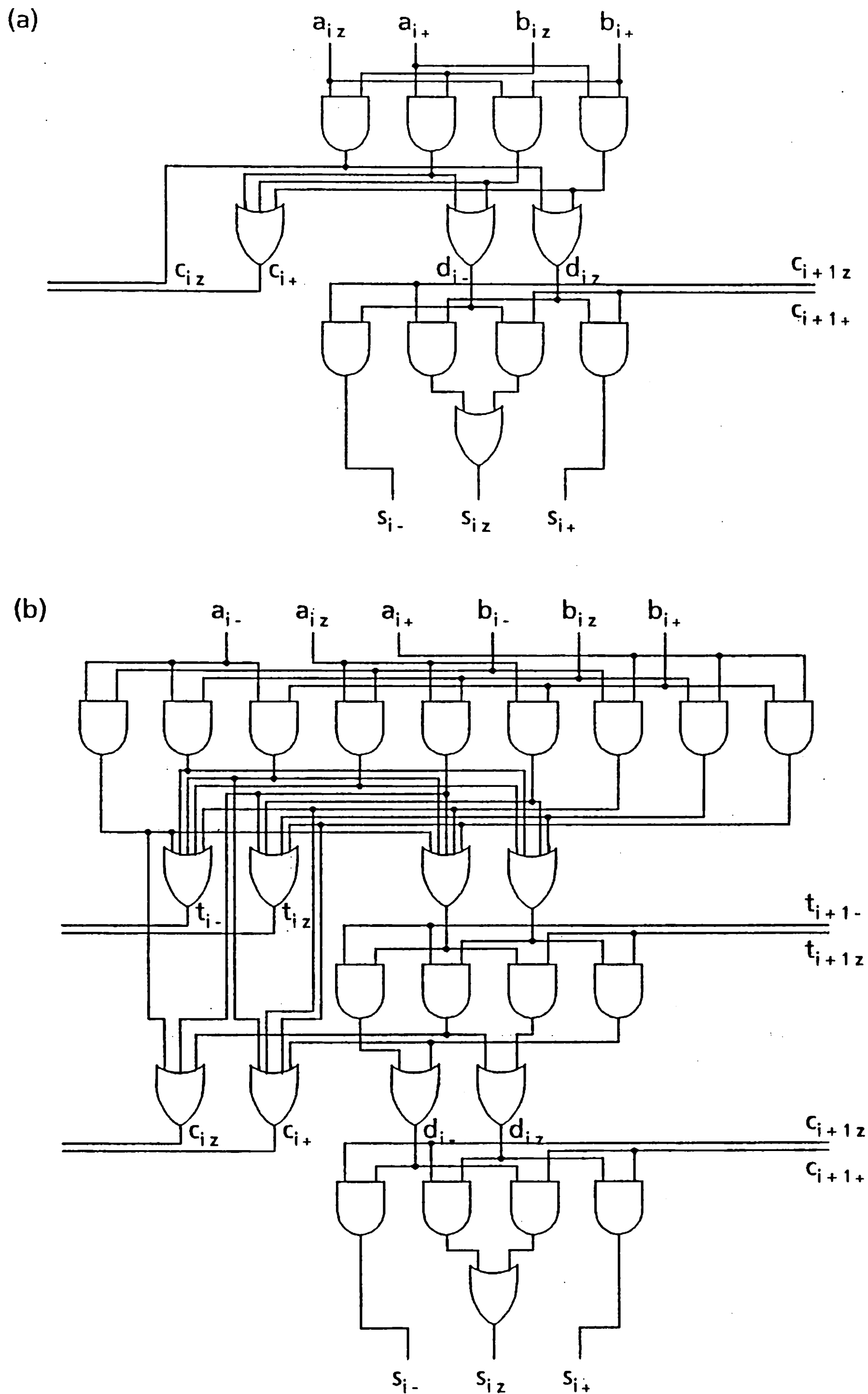

Fig. 7-3 Logic designs of the redundant binary addition cells
(a) For the first level (b) For the other levels

Table 7-1 Another computation rule for carry-propagation-free addition

(a) Step 1

t_i

a_i \ b_i	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$
0	$\bar{1}$	0	0
1	$\bar{1}$	0	0

$t_{i-1}=\bar{1}$

c_i, d_i

a_i \ b_i	$\bar{1}$	0	1
$\bar{1}$	0, $\bar{1}$	0,0	1, $\bar{1}$
0	0,0	0, $\bar{1}$	0,0
1	1, $\bar{1}$	0,0	1, $\bar{1}$

$t_{i-1}=0$

c_i, d_i

a_i \ b_i	$\bar{1}$	0	1
$\bar{1}$	0,0	1, $\bar{1}$	1,0
0	1, $\bar{1}$	0,0	1, $\bar{1}$
1	1,0	1, $\bar{1}$	1,0

(b) Step 2

s_i

d_i \ c_{i+1}	0	1
$\bar{1}$	$\bar{1}$	0
0	0	1

all the basic cells in the error checker (shown in Fig. 7-6) receive all of <0101>, <0110>, <1001> and <0101>, during, for example, the following 2^{n+1} code words are input; code words which represent the results of 0×0, 1×1, 1×2, $\cdots$, $1\times(2^n-1)$, $2^{n-1}\times2$, $2^{n-1}\times3$, $\cdots$, $2^{n-1}\times(2^n-1)$, $(2^{n-1}+2^{n-2}+2^{n-3})\times(2^{n-1}+2^{n-2})$, and $(2^{n-1}+2^{n-2}+2^{n-3})\times(2^{n-1}+2^{n-2}+2^{n-3})$.

Therefore, the multiplier is self-checking with respect to unidirectional stuck-at faults on multiple gate output lines.

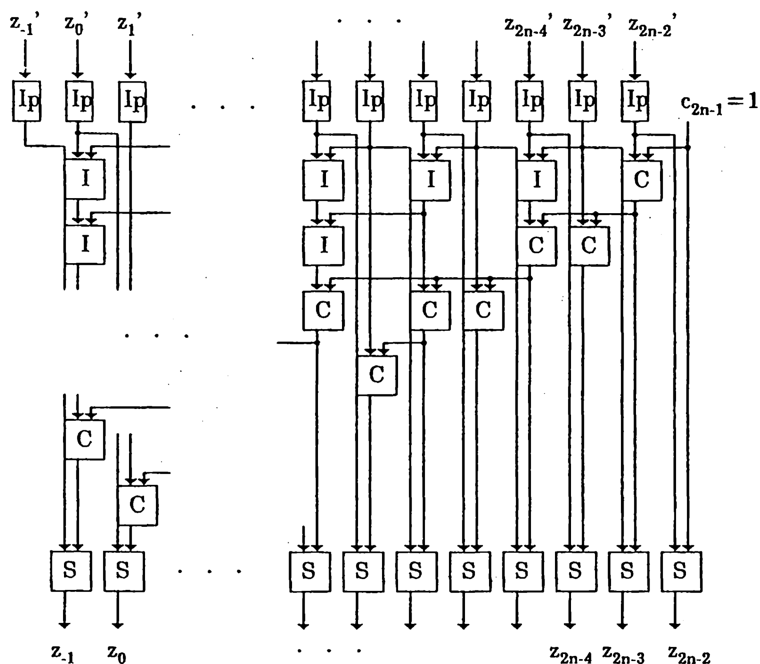

Fig. 7-4 A block diagram of the redundant binary to binary converter

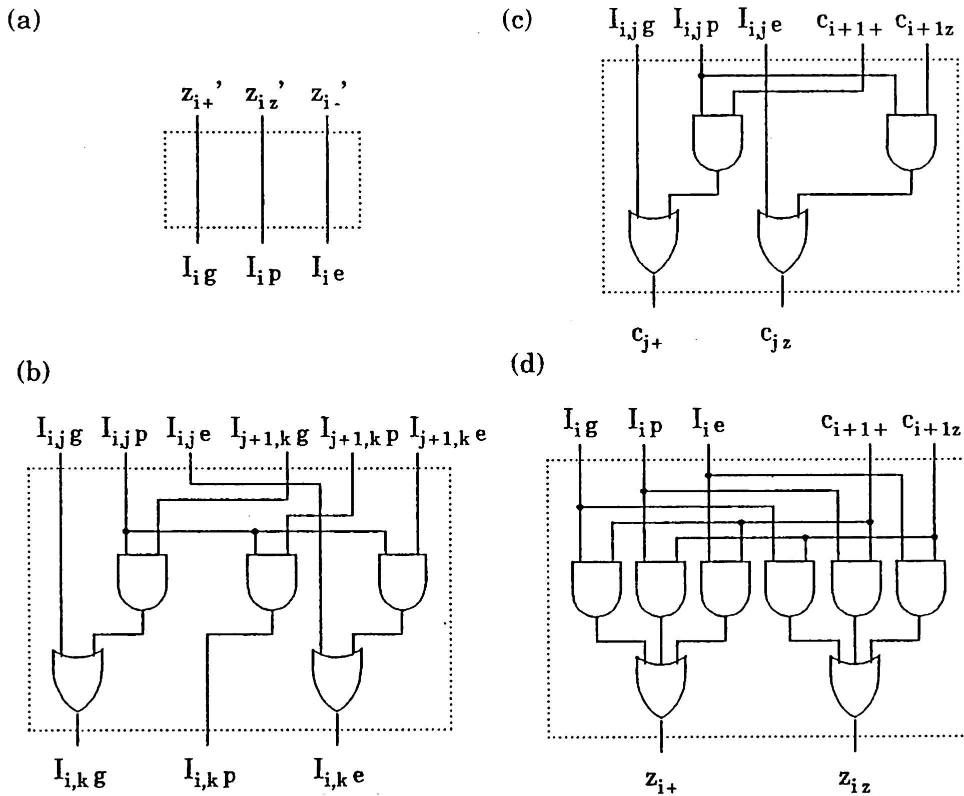

Fig. 7-5 Logic designs of the cells in the converter
(a) Ip-cell (b) I-cell (c) C-cell (d) S-cell

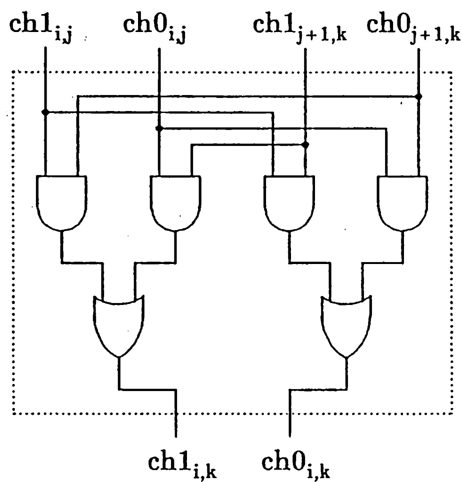

Fig. 7-6 A logic design of the checker cell

7.4 Remarks and Discussions

The design of self-checking arithmetic circuits based on the algorithms proposed in previous chapters by means of the three-rail logic has been considered. The three-rail logic is a logic design technique in which three mutually exclusive conditions calculated in a circuit are encoded in the 1-out-of-3 code and the circuit is designed to be inverter-free. A logic design of a self-checking multiplier based on the algorithm proposed in Chapter 3 has been shown as an example. The multiplier can perform n-bit multiplication in a time proportional to $log\,n$ and has a regular cellular array structure suitable for VLSI implementation, and furthermore, any error caused by unidirectional stuck-at faults on multiple gate output lines in it can be detected in normal operation.

In the previous sections, it is assumed that all inputs are applied to the arithmetic circuit between the occurrences of any two faults. However, if f_{j+1} sticks lines at the same value as f_j does, f_{j+1} can occur even immediately after f_j occurs, because the composition of f_j and f_{j+1} is a member of the assumed fault set. Furthermore, even if f_{j+1} sticks lines at the opposite value to that f_j does, it is enough that all the members of a subset of input vectors which test the arithmetic circuit for all detectable faults are input between the occurrences of f_j and f_{j+1}. Although unidirectional stuck-at faults on multiple gate output lines are assumed, many errors caused by other faults can also be detected.

The arithmetic circuit holds the self-checking feature, even if the functional block is designed to be inverter-free in a different way from the proposed one. However, if it is designed in another way, for example, using only the two-rail logic, the amount of hardware will be larger than the above design.

In widely used device technologies, such as TTL, MOS, ECL and so on, NAND and/or NOR gates are the basic gates. The AND-OR structure in the above logic design can be replaced by the NAND-NAND or the NOR-NOR structure. In either case, errors caused by faults that are equivalent to the faults assumed in the above can be detected.

We should design efficient hardware algorithms in due consideration of fault-tolerant features, as well as high-speed operations and regular structures, in order to develop high-performance arithmetic circuits.

Chapter 8

Redundant Coding Schemes for Several Algebraic Systems

8.1 Introduction

In this chapter, redundant coding schemes for several algebraic systems and the computational complexity of operations in the systems are considered to give a theoretical foundation for the design of arithmetic hardware algorithms. The concepts of 'coding scheme' and 'local computability' are first defined. Then, a redundant coding scheme for a residue class which uses the redundant binary representation and a hardware algorithm for modular addition by means of the coding scheme are proposed [YASUT8703a] [YASUT8703b]. Modular addition can be performed in a constant time independent of the magnitude of the moduli. The computations of the operation on a finite Abelian (commutative) group and operations on a residue ring of integers where two operations are defined are also considered and redundant coding schemes for these algebraic systems are discussed.

Researches on modular arithmetic and its related problems are also important and interesting in practice. Modular arithmetic is used in the residue number system (RNS) [GARN5906] [SVOB60], where high-speed computation can be achieved because addition and multiplication can be performed by independent modular additions and multiplications for the moduli,

respectively. Various researches have been made on the RNS and its application to digital systems [TAYL8405]. Especially, with recent advances of IC technologies, digital signal processors adopting RNS are proposed. The RNS is also useful for error detection and/or correction [JENK8304]. Modular addition plays important roles not only in the RNS but also in conversion of an RNS number into the ordinary representation. The proposed coding scheme for a residue class and the hardware algorithm for modular addition are very useful in practice.

In the next section, the concepts of 'coding scheme' and 'local computability' will be defined. In Section 8.3, a redundant coding scheme for a residue class and a hardware algorithm for modular addition will be proposed. In Section 8.4, redundant coding schemes for several algebraic systems will be discussed. In Section 8.5, some further discussions will be made.

8.2 Coding Schemes and Local Computability

A coding scheme C for a finite set S on an alphabet Γ with length n is defined as a mapping which satisfies the following two conditions.

(1) $C : \Gamma^n \rightarrow S \cup \{\lambda\}$, where $\lambda \notin S$.

(2) For any element s in S, there is at least one element x in Γ^n such that $C(x)=s$.

x is called a code of s. $|S|$ denotes the number of elements in S. Γ is a finite set of symbols. It is assumed that the size of Γ,

$|\Gamma|$, is larger than 1. Γ^n represents a set of strings on Γ with length n. Only fixed-length codes are considered. If C is redundant, there is an element in S which has two or more codes as shown in Fig. 8-1. (In this chapter, we mainly consider redundant coding schemes using the redundant binary representation. Namely, Γ is $\{\bar{1},0,1\}$ and $|\Gamma|$ is three.) The efficiency of the coding scheme C is defined as $(\log_{|\Gamma|}|S|)/n$. Assume that S is closed under a binary operation * defined on it. A binary operation # defined on Γ^n is stated to correspond to *, if C(x#y)=C(x)*C(y) holds for any codes x and y, i.e., (S,*) is homomorphic to $(\{x|C(x)\in S\},\#)$. If the coding scheme C is redundant, there may be several operations on Γ^n which correspond to *.

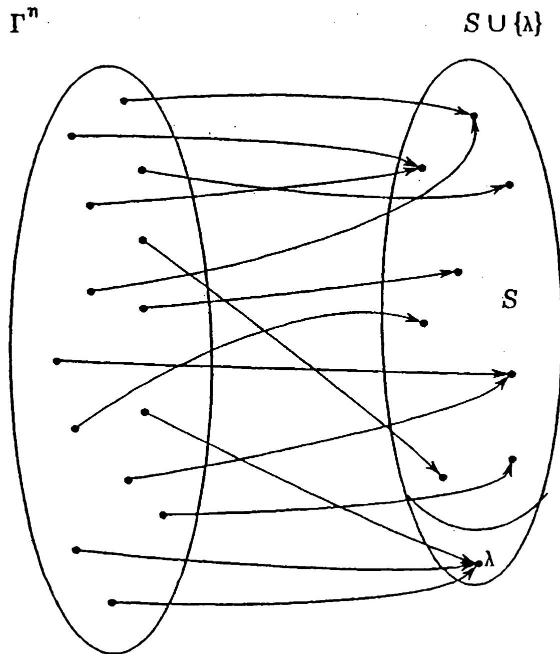

Fig. 8-1 A redundant coding scheme

A concept of local computability is introduced for clarifying the effect of a coding scheme on the computation speed. The local computability is defined as the maximum number of digits of operands required to determine each digit of the result, where the operands and results are encoded by a certain coding scheme. When every digit of the result depends on at most k digits of the operands, the operation is stated k-locally computable under the coding scheme. An operation * on a set S is said k-locally computable under a coding scheme C on an alphabet Γ with length n, if there is a k-locally computable function F which specifies an operation # on Γ^n corresponding to *.

8.3 A Redundant Coding Scheme for a Residue class and a Hardware Algorithm for Modular Addition

In this section a redundant coding scheme for the residue class modulo m in the integer ring, i.e., $Z_m=\{0,1,\cdots,m-1\}$ is considered. It is assumed that m is not a power of two. (When it is, modular addition can be performed easily by the ordinary irredundant binary coding.) $X=[x_1x_2\cdots x_n]$ ($x_i\in\{\bar{1},0,1\}$) is used to represent an element of Z_m, where n is $\lceil\log_2 m\rceil+1$. X can be regarded as an n-digit redundant binary integer. (Since integers are handled, a different notation from that used in the previous chapters is used. An n-digit redundant binary integer $X=[x_1x_2\cdots x_n]_{SD2}$ has the value $\|X\|=\sum_{i=1}^{n} x_i\cdot 2^{n-i}$.)

The proposed coding scheme is as follows. ($\|X\|$ denotes the

value of the redundant binary integer $X=[x_1 x_2 \cdots x_n]_{SD2}$.)

Coding Scheme [RB]

$$RB : \{\bar{1},0,1\}^n \rightarrow Z_m \cup \{\lambda\} \qquad (n=\lceil \log_2 m \rceil+1)$$

$$RB([x_1 x_2 \cdots x_n]) = \begin{cases} \|X\| & \text{if } 0 \leqq \|X\| < m, \\ \|X\|+m & \text{if } -m < \|X\| < 0, \\ \lambda & \text{otherwise,} \end{cases}$$

The efficiency of the coding scheme [RB] is $\lceil \log_3 m \rceil / n = \lceil \log_3 m \rceil / (\lceil \log_2 m \rceil + 1) = O(1)$. This coding scheme has double redundancy. Namely, an element s of Z_m is represented by a redundant binary integer whose value is either s or s-m, and for both s and s-m there may be several redundant binary integers which have these values.

The ordinary irredundant binary representation of an element s of Z_m is itself one of the redundant representations of s. Therefore, no computation is needed for conversion of the irredundant binary representation to the redundant representation. A redundant representation of s, X, can be converted to the equivalent n-bit two's complement binary integer X" by ordinary subtraction of two n-bit unsigned binary integers, as mentioned in Section 2.3. When X" is nonnegative, i.e., its most significant bit is 0, the irredundant representation X' of s is X" itself. When X" is negative, i.e., its most significant bit is 1, X' can be obtained by adding X" and M where M is an n-bit binary integer which has the value m. The conversion of the redundant representation to the irredundant binary representation

can be performed in a computation time proportional to $log n$ by means of carry-look-ahead adders or in a time proportional to n by means of ripple-carry adders. (Since an element of Z_m is represented by an (n-1)-bit binary integer in the ordinary irredundant binary representation, in the conversion, a digit '0' is attached to or deleted from the most significant position.)

The hardware algorithm proposed below makes modular addition under the coding scheme [RB] be performed in a constant time independent of the word length.

Algorithm [MODADD]

<Input>

X and Y : an augend and an addend, respectively

(n-bit redundant binary integers, $-m < \|X\|, \|Y\| < m$)

<Output>

Z (=X#Y) : the sum modulo m

(an n-bit unsigned binary integer, $-m < \|Z\| < m$)

<Algorithm>

Step 1: U:=X+Y ($U=[u_0 u_1 u_2 \cdots u_n]_{SD2}$)

(redundant binary addition)

Step 2:

$$Z := \begin{cases} U+M & \text{if } [u_0 u_1 u_2]_{SD2} < 0 \\ U+0 & \text{if } [u_0 u_1 u_2]_{SD2} = 0 \\ U+\bar{\bar{M}} & \text{if } [u_0 u_1 u_2]_{SD2} > 0 \end{cases}$$

(redundant binary addition)

(M and $\bar{\bar{M}}$ are n-digit redundant binary integers whose values are m and -m, respectively.) □

In Step 1, the addition X+Y is performed in the redundant binary number system, and an (n+1)-digit redundant binary integer U is obtained. The addition rule shown in Table 2-1 in Section 2.3 can be used. The required computation time is constant independent of n. The required gate count is proportional to n.

In Step 2, Z is obtained by calculating U+M or U+0 or $U+\bar{\bar{M}}$ accordingly as the most significant three digits of U is negative or zero or positive. When we let M and $\bar{\bar{M}}$ be the n-digit redundant binary integer with the value of m whose all digits are nonnegative and the one with the value -m whose most significant digit is $\bar{1}$ and the other all digits are nonnegative, respectively, the addition rule shown in Table 2-2 in Section 2.3 can be used for the addition except at the most significant two

Table 8-1 An addition rule at the most significant two positions in Step 2 of Algorithm [MODADD]

u_0	u_1 \ u_2	z_1: $\bar{1}$	0	1
$\bar{1}$	0	---	$\bar{1}$	$\bar{1}$
$\bar{1}$	1	$\bar{1}$	0	0
0	$\bar{1}$	$\bar{1}$	0	0
0	0	0	0	0
0	1	0	0	1
1	$\bar{1}$	0	0	1
1	0	1	1	---

positions where the special addition rule shown in Table 8-1 is used. The required computation time is constant independent of n. The required gate count is proportional to n.

Fig. 8-2 shows examples of modular addition in accordance with the algorithm.

According to the algorithm, $-m<\|Z\|<m$ holds, and therefore, $C(Z)=C(X)+_m C(Y)$. (Take notice that $2^{n-2}<m<2^{n-1}$, and since $-m<\|X\|<m$ and $-m<\|Y\|<m$, $-2m<\|U\|<2m$.) Namely, the operation # defined by Algorithm [MODADD] corresponds to the addition $+_m$ on Z_m, i.e., the modular addition with modulus m.

augend	X	$1\ \bar{1}\ 0\ 0\ \bar{1}\ 1\ 0\ 1$	
addend	Y	$+\ 1\ 0\ 0\ \bar{1}\ \bar{1}\ 0\ 1\ \bar{1}$	step 1
	U	$\underline{1\ 0\ \bar{1}}\ \bar{1}\ 0\ 1\ \bar{1}\ 1\ 0$	
	$+\bar{M}$	$+\ \bar{1}\ 0\ 0\ 0\ 1\ 0\ 0\ 0$	step 2
sum	Z	$1\ \bar{1}\ \bar{1}\ 1\ 0\ 0\ \bar{1}\ 0$	

augend	X	$1\ \bar{1}\ 0\ 0\ \bar{1}\ 1\ 0\ 1$	
addend	Y	$+\ \bar{1}\ 0\ 0\ 1\ 1\ 0\ \bar{1}\ 1$	step 1
	U	$\underline{0\ 0\ \bar{1}}\ 0\ 1\ 0\ 1\ 0\ 0$	
	$+M$	$+\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0$	step 2
sum	Z	$0\ 1\ 0\ 1\ 0\ \bar{1}\ 0\ 0$	

Fig. 8-2 Examples of modular addition according to Algorithm [MODADD] (m=120)

The following theorem can be proved.

[Theorem 8.1]

The addition $+_m$ on Z_m is 16-locally computable under the coding scheme [RB].

<Proof>

Each digit of U is depends on 6 digits of X and Y. (Recall the discussion in Section 2.3.) The addend in Step 2 is determined by examining the most significant three digits of U which depend on 8 digits of X and Y. After the addend is determined, each digit of Z depends on 2 digits of U which depend on 8 digits of X and Y. Therefore, each digit of Z depends on at most 16 digits of X and Y (8 digits of each).

Q.E.D.

Thus, modular addition can be performed in a constant computation time independent of n with a gate count proportional to n.

Since m is a given constant, simpler addition cells can be used for Step 2 by adopting appropriate redundant binary integers to represent m and -m.

8.4 Redundant Coding Schemes for Other Algebraic Systems

In this section, redundant coding schemes for a finite Abelian (commutative) group and for a finite ring are considered.

Since any cyclic group of order m is isomorphic to the residue class Z_m, the following corollary is derived from [Theorem 8.1].

[Corollary 8.1]

For any finite cyclic group $(G,*)$, there is a redundant coding scheme on the alphabet $\Gamma=\{\bar{1},0,1\}$ with the efficiency of $O(1)$ under which $*$ is $O(1)$-locally computable.

It is well known in the group theory that any Abelian group can be decomposed into a Cartesian product of cyclic groups. Hence, the following theorem can be proved.

[Theorem 8.2]

For any finite Abelian group $(G,*)$, there is a redundant coding scheme on the alphabet $\Gamma=\{\bar{1},0,1\}$ with the efficiency of $O(1)$ under which $*$ is $O(1)$-locally computable.

<Proof>

Assume that G is decomposable into cyclic groups G1, G2, $\cdots$, Gh. There is a coding scheme in which an element X in G can be encoded as follows: (1) X is represented by a vector $[X1,X2,\cdots,Xh]$ where Xj is an element of Gj, and (2) an element of each Gj is encoded using the redundant coding scheme [RB]. Under the coding scheme, the binary operation $*$ on G can be computed by elementwise operations $*_{(j)}$'s on Gj's. Namely, $[X1,X2,\cdots,Xh]*[Y1,Y2,\cdots,Yh]=[X1*_{(1)}Y1,X2*_{(2)}Y2,\cdots,Xh*_{(h)}Yh]$. From [Corollary 8.1], each $*_{(j)}$ is $O(1)$-locally computable, and

therefore, $*$ is O(1)-locally computable. The efficiency of this coding scheme is also a constant independent of the order of G and the number of decomposed cyclic groups.

Q.E.D.

Concerning to a residue ring of integer modulo m, i.e., Z_m, on which the modular addition $+_m$ and the modular multiplication $*_m$ are defined, the following theorem holds.

[Theorem 8.3]

For a residue ring modulo m, i.e., $(Z_m, +_m, *_m)$, there is a redundant coding scheme with the efficiency of O(1) such that $+_m$ is O(1)-locally computable and $*_m$ is $O(\log(\max q_j))$-locally computable, when m is a product of q_j's $(j=1,2,\cdots,h)$ which are relatively prime.

<Proof>

There is a coding scheme in which an element X in Z_m is encoded as follows: (1) X is represented by a vector $[X1, X2, \cdots, Xh]$ where $Xj = X$ modulo q_j and an element of Z_{q_j}, and (2) an element of each Z_{q_j} is encoded using the redundant coding scheme [RB]. Under the coding scheme, the addition and the multiplication on Z_m can be realized by elementwise additions and multiplications on Z_{q_j}'s, respectively. Namely, $[X1, X2, \cdots, Xh] +_m [Y1, Y2, \cdots, Yh] = [X1 +_{q_1} Y1, X2 +_{q_2} Y2, \cdots, Xh +_{q_h} Yh]$ and $[X1, X2, \cdots, Xh] *_m [Y1, Y2, \cdots, Yh] = [X1 *_{q_1} Y1, X2 *_{q_2} Y2, \cdots, Xh *_{q_h} Yh]$. From [Corollary 8.1], each $+_{q_j}$ is O(1)-locally computable, and therefore, $+_m$ and $*_m$ are O(1) and $O(\log(\max q_j))$-locally

computable, respectively. The efficiency of this coding scheme is O(1).

Q.E.D.

From [Theory 8.3], the following corollary is obtained.

[Corollary 8.2]

For a residue ring modulo m, i.e., $(Z_m, +_m, *_m)$, there is a redundant coding scheme with the efficiency of O(1) such that $+_m$ is O(1)-locally computable and $*_m$ is $O(\log\log m)$-locally computable, when m is a product of the smallest h primes. Moreover, we can construct an adder with the depth of O(1) and the gate count of $O(\log m)$, and a multiplier with the depth of $O(\log\log\log m)$ and the gate count of $O((\log m)^2)$ under the coding scheme.

8.5 Remarks and Discussions

Redundant coding schemes for several algebraic systems and the computational complexity of operations in the systems have been considered. The concept of 'local computability' has been introduced for clarifying the effect of a coding scheme on the computation speed. The coding scheme for a residue class which uses the redundant binary representation and the modular addition algorithm by means of the coding scheme proposed in Section 8.2 are very useful in practice. Modular addition can be performed in

a constant time independent of the magnitude of the moduli. They are effective in applications in which additions are successively carried out in a residue class with a rather large modulus. The author has applied them in a hardware algorithm for RNS to binary conversion based on the Chinese Remainder Theorem [TAKA08606]. Redundant coding schemes for a finite Abelian group and for a residue ring of integers have also been considered.

In the previous sections, coding schemes using the redundant binary representation have been considered. Namely, the alphabet $\{\bar{1},0,1\}$ whose size is three has been used. When the size of an alphabet Γ is large, the following redundant coding scheme [SDr] can be used instead of the coding scheme [RB]. It is assumed that $|\Gamma|>6$. Γ is regarded as $\{-(r/2+1),\cdots,-1,0,1,\cdots,(r/2+1)\}$ where $r=|\Gamma|-3$, when $|\Gamma|$ is odd. It is regarded as $\{-(r/2+1),\cdots,-1,0,1,\cdots,r/2+1,\gamma\}$ where $r=|\Gamma|-4$, when $|\Gamma|$ is even. γ is not used for the coding. The length of a code n is $\lceil \log_r m \rceil+1$. The coding scheme is as follows. ($\|X\|$ denotes the value of the signed digit number with radix r, $X=[x_1 x_2 \cdots x_n]_{SDr}$. Namely, $\|X\|=\sum_{i=1}^{n} x_i \cdot r^{n-i}$.)

Coding Scheme [SDr]

$$SDr : \Gamma^n \rightarrow Z_m \cup \{\lambda\}$$

$$SDr([x_1 x_2 \cdots x_n]) = \begin{cases} \|X\| & \text{if } 0 \leq \|X\| < m, \\ \|X\|+m & \text{if } -m < \|X\| < 0, \\ \lambda & \text{otherwise,} \end{cases}$$

The efficiency of the coding scheme [SDr] is $\lceil \log_{|\Gamma|} m \rceil / n$

$=\lceil \log_{|\Gamma|} m \rceil / (\lceil \log_{|\Gamma|-3} m \rceil + 1) = O(1)$. It can be proved that the addition $+_m$ on Z_m is 12-locally computable under the coding scheme [SDr]. [Theorem 8.1] can be extended as follows.

[Theorem 8.4]

For any residue class modulo m in the integer ring, i.e., Z_m, and for any alphabet Γ ($|\Gamma|>1$), there is a redundant coding scheme on Γ with the efficiency of O(1) under which addition $+_m$ is O(1)-locally computable.

<Proof>

In the cases that $|\Gamma|=3$ and $|\Gamma|>6$, the coding scheme [RB] and [SDr] can be used, respectively. When $|\Gamma|=2$, we can apply the coding scheme [RB] by encoding each element of $\{\bar{1},0,1\}$ by two bits. When $|\Gamma|$ is 4 or 5 or 6, we can easily apply the coding scheme [RB] by using only three symbols in Γ. In any case, the efficiency of the coding scheme is a constant independent of m and $|\Gamma|$.

Q.E.D.

It is interesting to make researches on redundant coding schemes for other algebraic systems, such as semigroups, fields and so on.

Chapter 9

Conclusion

Hardware algorithms for arithmetic operations with the redundant binary representation and their related problems have been discussed.

New hardware algorithms for multiplication, division, square root extraction and computation of several elementary functions have been proposed. A multiplier based on the algorithm proposed in Chapter 3 can perform multiplication in a computation time proportional to the logarithm of the word length of the operands and has a regular cellular array structure suitable for VLSI implementation. The multiplier can effectively be used for the computation of other arithmetic operations. A divider based on the algorithm proposed in Chapter 4 and a square root circuit based on the algorithm proposed in Chapter 5 can perform division and square root extraction fast, respectively, and are very suited to VLSI implementation. As shown in Chapter 6, the computation speed of the CORDIC and the STL method can be improved by the use of the redundant binary representation for internal computation.

Since the redundant binary representation uses three values, i.e., 0, 1 and $\bar{1}$ for each digit, three valued computation elements may effectively be used. It is also interesting to use higher radix signed-digit representations.

A new design method for self-checking arithmetic circuits based on the proposed algorithms have been proposed, in Chapter 7. The logic design technique called the three-rail logic is used in the method. Arithmetic circuits based on the proposed algorithms and designed by means of the three-rail logic can perform arithmetic operations fast, have a regular cellular array structure, and further, have a self-checking feature.

Since in arithmetic circuits based on the proposed algorithms, interconnection of computation elements is regular and the fun-out of most of the computation elements is small, they may have an easily testable feature.

In Chapter 8, redundant coding schemes for several algebraic systems and the computational complexity of operations in the systems have been considered to give a theoretical foundation for the design of arithmetic hardware algorithms. The redundant coding scheme for a residue class and the hardware algorithm for modular addition are very useful in practice.

As shown in this thesis, there is possibility that redundant coding schemes can be used to achieve local computability, (and hence, high-speed computation,) in the design of various arithmetic circuits.

With the advances of IC technology, it becomes possible to implement a special-purpose circuit solving a certain problem quickly. In the development of such a circuit, design of a good hardware algorithm is one of the key points. It is similar to the case of the development of software in which algorithm design is one of the key points [AHO-H74]. Designing a hardware algorithm,

we have to consider that a circuit based on the algorithm can perform high-speed computation, have a regular structure and have fault-tolerant features. In order to design a good hardware algorithm, a suitable data representation and/or structure should be employed. Especially, in the design of an arithmetic hardware algorithm, the use of a suitable number representation is crucial.

Acknowledgments

I would like to express my appreciation and thanks to all those who helped in the preparation of this thesis. In particular, I wish to express my sincere appreciation and thanks to Professor Shuzo Yajima of Kyoto University for his continuous guidance, invaluable suggestions and support.

I am grateful to thank Associate Professor Hiroto Yasuura of Kyoto University who introduced me to the research field of hardware algorithms and has been giving me invaluable suggestions, accurate criticisms and encouragements throughout this research.

I also acknowledge the interesting comments that I have received from Professor Yahiko Kambayashi of Kyushu University, Associate Professor Hiromi Hiraishi of Kyoto University and Associate Professor Kazuo Iwama of Kyoto Sangyo University. I would also like to express my thanks to Mr. Tohru Asada, Mr. Yasuo Okabe and other members of Yajima Laboratory, Department of Information Science, Kyoto University, for their kind discussions.

Finally, I would like to give special thanks to my parents for their continuing love and affection throughout this research.

References

[AGRA7903]

D. P. Agrawal, "High-speed arithmetic arrays," IEEE Trans. Comput., vol. C-28, no. 3, pp. 215-224, Mar. 1979.

[AHO-H74]

A. Aho, J. Hopcroft and J. Ullman, 'The Design and Analysis of Computer Algorithms,' Addison-Wesley, 1974.

[ANDEM7303]

D. A. Anderson and G. Metze, "Design of totally self-checking check circuits for m-out-of-n codes," IEEE Trans. Comput., vol. C-22, no. 3, pp. 263-269, Mar. 1973.

[ATKI6810]

D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," IEEE Trans. Comput., vol. C-17, no. 10, pp. 925-934, Oct. 1968.

[ATKI7008]

D. E. Atkins, "Design of the arithmetic units of Illiac III : Use of redundancy and higher radix methods," IEEE Trans. Comput., vol C-19, no. 8, pp. 720-733, Aug. 1970.

[AVIZ6109]

A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," IRE Trans. Electron. Comput., vol. EC-10, no. 3, pp. 389-400, Sept. 1961.

[BRAU63]

E. L. Braun, 'Digital Computer Design,' Academic Press, 1963.

[BRENK8107]

R. P. Brent and H. T. Kung, "The area-time complexity of binary multiplication," Journal of the ACM, vol. 28, no. 3, pp. 521-534, July 1981.

[BRENK8205]

R. P. Brent and H. T. Kung, "A regular layout for parallel adders," IEEE Trans. Comput., vol. C-31, no. 3, pp. 260-264, Mar. 1982.

[CANTE6204]

D. Cantor, G. Estrin and R. Turn, "Logarithmic and exponential

function evaluation in a variable structure digital computer," IRE Trans. Electron. Comput., vol. EC-11, no. 4, pp. 155-164, Apr. 1962.

[CAVA84]

J. J. F. Cavanagh, 'Digital Computer Arithmetic / Design and Implementation,' McGraw-Hill, 1984.

[CHEN7207]

T. C. Chen, "Automatic computation of exponentials, logarithms, ratios, and square roots," IBM Journal of Research and Development, vol. 16, no. 4, pp. 380-388, July 1972.

[CHOWR7810]

C. Y. Chow and J. E. Robertson, "Logical design of a redundant binary adder," Proc. 4th Symp. Comput. Arithmetic, pp. 109-115, Oct. 1978.

[COWG6404]

D. Cowgill, "Logic equations for a built-in square root method," IEEE Trans. Electron. Comput., vol. EC-13, no. 2, pp. 156-157, Apr. 1964.

[DELU7006]

B. G. DeLugish, "A class of algorithms for automatic evaluation of certain elementary functions in a binary computer," Report no. 399, Dept. of Computer Science, Univ. of Illinois, June 1970.

[FAND8705]

J. Fandrianto, "Algorithm for high speed shared radix 4 division and radix square-root," Proc. 8th Symp. Comput. Arithmetic, pp. 73-79, May 1987.

[GARN5906]

H. L. Garner, "The residue number system," IRE Trans. Electron. Comput., vol. EC-8, no. 2, pp. 140-147, June 1959.

[HARTC78]

J. F. Hart et al., 'Computer Approximations,' R. F. Drenick et al. Eds., 'SIAM Series of Applied Mathematics,' John Wiley & Sons, 1978.

[HAVIT8002]

G. L. Haviland and A. A. Tuszynski, "A CORDIC arithmetic

processor chip," IEEE Trans. Comput., vol. C-29, no. 2, pp. 68-79, Feb. 1980.

[HITAC7807]
Hitachi, 'HITAC Manual no. 8080-3-218-10 : VOS1/VOS2/VOS3 Mathematical Functions,' July 1978.

[HWAN79]
K. Hwang, 'Computer Arithmetic / Principles, Architecture, and Design,' John Wiley & Sons, 1979.

[HWAN7904]
K. Hwang, "Global and modular two's complement array multipliers," IEEE Trans. Comput., vol. C-28, no. 4, pp. 300-306, Apr. 1979.

[HWANC7810]
K. Hwang and T. P. Chang, "A new interleaved rational / radix number system for high-precision arithmetic computations," Proc. 4th Symp. Comput. Arithmetic, pp. 15-24, Oct. 1978.

[IEEE754]
"IEEE Standard for Binary Floating-Point Arithmetic," IEEE Standard 754, 1985, IEEE Computer Society.

[JENK8304]
W. K. Jenkins, "The design of error-checkers for self-checking residue number arithmetic," IEEE Trans. Comput., vol. C-32, no. 4, pp. 388-395, Apr. 1983.

[KAMEH8006]
M. Kameyama and T. Higuchi, "Design of radix-4 signed-digit arithmetic circuits for digital filtering," Proc. 10th Int. Symp. Multiple-Valued Logic, pp. 272-277, June 1980.

[LENA5507]
E. H. Lenaerts, "Automatic square rooting," Electron. Eng., vol. 27, no. 329, pp. 287-289, July 1955.

[LUK-V83]
W. K. Luk and J. E. Vuillemin, "Recursive implementation of optimal time VLSI integer multipliers," Proc. VLSI 83, pp. 155-168, F. Anceau and E. J. Aas, Eds., Elsevier Science, 1983.

[MAJE8508]

S. Majerski, "Square-rooting algorithms for high-speed digital circuits," IEEE Trans. Comput., vol. C-34, no. 8, pp. 724-733, Aug. 1985.

[MATU7511]

D. W. Matula, "Fixed-slash and floating-slash rational arithmetic," Proc. 3rd Symp. Comput. Arithmetic, pp. 90-91, Nov. 1975.

[MCALZ8602]

W. E. McAllister et al., "An NMOS 64b floating-point chip set," Proc. 1986 IEEE Int. Solid-State Circuit Conf., pp. 34-35, Feb. 1986.

[METZ6504]

G. Metze, "Minimal square rooting," IEEE Trans. Electron. Comput., vol. EC-14, no. 2, pp. 181-185, Apr. 1965.

[REIT60]

G. W. Reitwiesner, "Binary arithmetic," in F. L. ALT et al. Eds. 'Advances in Computers, vol. 1,' pp. 231-308, Academic Press, 1960.

[REUSK81]

P. Reusens, W. H. Ku and Y. Mao, "Fixed-point high-speed parallel multipliers in VLSI," in H. T. Kung, B. Sproull and G. Steele, Eds., 'VLSI Systems and Computations,' pp. 301-310, Computer Science Press, 1981.

[ROBE5809]

J. E. Robertson, "A new class of digital division methods," IRE Trans. Electron. Comput., vol. EC-7, no. 3, pp.218-222, Sept. 1958.

[SAVA76]

J. E. Savage, 'The Complexity of Computing,' John Wiley & Sons, 1976.

[SCOT85]

N. R. Scott, 'Computer Number Systems and Arithmetic,' Prentice-Hall, 1985.

[SELLH68]

F. F. Sellers, M. Hsiao and L. W. Bearnson, 'Error Detecting Logic for Digital Computers,' McGraw-Hill, 1968.

[SMITM7806]

J. E. Smith and G. Metze, "Strongly fault secure logic networks," IEEE Trans. Comput., vol. C-27, no. 6, pp. 491-499, June 1978.

[SPEC6501]

W. H. Specker, "A class of algorithms for ln x, exp x, sin x, cos x, $\tan^{-1}x$ and $\cot^{-1}x$," IEEE Trans. Electron. Comput., vol. EC-14, no. 1, pp. 85-86, Jan. 1965.

[STENK7710]

W. J. Stenzel, W. J. Kubitz and G. H. Garcia, "A compact high-speed parallel multiplication scheme," IEEE Trans. Comput., vol. C-26, pp. 948-957, Oct. 1977.

[SVOB60]

A. Svobada, 'Digitale Informationswander,' Vieweg and Sohn, 1960.

[TAMAK8303]

K. Tamaru and K. Kanehara, "Array-structured elementary function circuits for VLSI processors," Trans. IECE, vol. J66-D, no. 3, pp. 309-315, Mar. 1983. (in Japanese)

[TAYL8405]

F. J. Taylor, "Residue arithmetic: a tutorial with examples," IEEE Computer, vol. 17, no. 5, pp. 50-62, May 1984.

[TAYL8506]

G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," Proc. 7th Symp. Comput. Arithmetic, pp. 64-71, June 1985.

[UNGE7704]

S. Unger, "Tree Realization of Iterative Circuit," IEEE Trans. Comput., vol. C-26, no. 4, pp. 365-383, Apr. 1977.

[VOLD5909]

J. E. Volder, "The CORDIC trigonometric computing technique," IRE Trans. Electron. Comput., vol. EC-8, no. 3, pp.330-334, Sep. 1959.

[VUIL8304]

J. E. Vuillemin, "A very fast multiplication algorithm for VLSI implementation," Integration, VLSI Journal, vol. 1,

no. 1, pp. 39-52, Apr. 1983.

[WAKE78]

J. Wakerly, 'Error Detecting Codes, Self-Checking Circuits and Applications,' North-Holland, 1978.

[WALL6402]

C. S. Wallace, "A suggestion for a fast multiplier," IEEE Trans. Electron. Comput., vol. EC-13, no. 1, pp. 14-17, Feb. 1964.

[WALT7105]

J. S. Walther, "A unified algorithm for elementary functions," Proc. AFIPS 1971 SJCC, pp. 379-385, May 1971.

[YASUY8201]

H. Yasuura and S. Yajima, "Embedding problems of combinational circuits into VLSI," Report of Technical Group on Automata and Languages, IECE, AL81-97, Jan. 1982. (in Japanese)

[YASUY8208]

H. Yasuura and S. Yajima, "On the area of logic circuits in VLSI," Trans. IECE, vol. J65-D, no. 8, pp. 1080-1087, Aug. 1982. (in Japanese)

List of Publications by the Author

Major Publications

[YASUT8202]

H. Yasuura and N. Takagi, "A high-speed sorting circuit using parallel enumeration sort," Trans. IECE, vol. J65-D, no. 2, pp. 179-186, Feb. 1982. (in Japanese)

[YASUT8212]

H. Yasuura, N. Takagi and S. Yajima, "The parallel enumeration sorting scheme for VLSI," IEEE Trans. Comput., vol. C-31, no. 12, pp. 1192-1201, Dec. 1982.

[TAKAY8306a]

N. Takagi, H. Yasuura and S. Yajima, "A VLSI-oriented high-speed multiplier using a redundant binary addition tree," Trans. IECE, vol. J66-D, no. 6, pp. 684-690, June 1983. (in Japanese)

[TAKAY8404]

N. Takagi, H. Yasuura and S. Yajima, "A VLSI-oriented high-speed divider using redundant binary representation," Trans. IECE, vol. J67-D, no. 4, pp. 450-457, Apr. 1984. (in Japanese)

[TAKAY8405a]

N. Takagi, H. Yasuura, K. Taima, H. Hayata and S. Yajima, "An implementation and evaluation of the parallel enumeration sorting circuit," Trans. IECE, vol. J67-D, no. 5, pp. 623-624, May 1984. (in Japanese)

[HARAN8410]

Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa and N. Takagi, "High-speed multiplier using redundant binary adder tree," Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers ICCD'84, pp. 165-170, Oct. 1984.

[TAKAW8501]

N. Takagi and C. K. Wong, "A hardware sort-merge system," IBM Journal of Research and Development, vol. 29, no. 1, pp.49-67, Jan. 1985.

[TAKAY8505a]

N. Takagi and S. Yajima, "On-line error-detectable high-speed multiplier with a redundant binary adder tree," Proc. Int. Symp. on Circuits and Systems : ISCAS 85, pp. 1321-1324, May 1985.

[TAKAY8509]
N. Takagi, H. Yasuura and S. Yajima, "High-speed VLSI multiplication algorithm with a redundant binary addition tree," IEEE Trans. Comput., vol. C-34, no. 9, pp. 789-796, Sep. 1985.

[TAKAY8601a]
N. Takagi and S. Yajima, "A square root hardware algorithm using redundant binary representation," Trans. IECE, vol. J69-D, no. 1, pp. 1-10, Jan. 1986. (in Japanese)

[TAKAY8601b]
N. Takagi and S. Yajima, "Hardware algorithms for computing exponentials and logarithms using redundant binary representation," Trans. IECE, vol. J69-D, no. 1, pp. 11-20, Jan. 1986. (in Japanese)

[TAKAA8606]
N. Takagi, T. Asada and S. Yajima, "A hardware algorithm for computing sine and cosine using redundant binary representation," Trans. IECE, vol. J69-D, no. 6, pp. 841-847, June 1986. (in Japanese)

[HARAN8702]
Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa and N. Takagi, "A high-speed multiplier using a redundant binary adder tree," IEEE Journal of Solid-State Circuits, vol. SC-22, no. 1, pp. 28-34, Feb. 1987.

[YASUT8703a]
H. Yasuura, N. Takagi and S. Yajima, "On high-speed parallel algorithms using redundant coding," Trans. IEICE, vol. J70-D, no. 3, pp. 525-533, Mar. 1987. (in Japanese)

[YASUT8703b]
H. Yasuura, N. Takagi and S. Yajima, "Redundant coding for local computability," in W. Rheinboldt et al. Eds., 'Perspectives in Computing,' vol. 15 : D. S. Johnson et al. Eds., 'Discrete Algorithms and Complexity : Proc. Japan-US Joint

Seminar 1986,' pp. 145-159, Academic Press, Mar. 1987.
[KUNIN8705]
S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," Proc. 8th Symp. Comput. Arithmetic, pp. 80-86, May 1987.

Technical Reports

[YASUT8102]
H. Yasuura and N. Takagi, "Design of high-speed sorting circuit using parallel enumeration sort algorithm," Report of Technical Group on Automata and Languages, IECE, AL80-76, Feb. 1981. (in Japanese)
[TAKAY8209]
N. Takagi, H. Yasuura and S. Yajima, "A VLSI-oriented O(log n) stage high-speed multiplier using a redundant binary addition tree," Report of Technical Group on Automata and Languages, IECE, AL82-31, Sep. 1982. (in Japanese)
[TAKAY8306b]
N. Takagi, H. Yasuura and S. Yajima, "Hardware algorithms for division and square rooting internally using redundant binary representation," RIMS Koukyuroku 494, pp. 223-235, Research Institute for Mathematical Sciences, Kyoto Univ., June 1983.
[TAKAY8402]
N. Takagi and S. Yajima, "Hardware algorithms for exponential and logarithmic functions using redundant binary representation," Report of Technical Group on Automata and Languages, IECE, AL83-70, Feb. 1984. (in Japanese)
[TAKAY8405b]
N. Takagi and S. Yajima, "On hardware algorithms for exponential and logarithmic function," RIMS Koukyuroku 522, pp. 251-265, Research Institute for Mathematical Sciences, Kyoto Univ., May 1984. (in Japanese)
[TAKAA8501]
N. Takagi, T. Asada and S. Yajima, "A hardware algorithm for

trigonometric functions using redundant binary representation," Report of Technical Group on Automata and Languages, IECE, AL84-59, Jan. 1985. (in Japanese)

[YASUT8503]
H. Yasuura, N. Takagi and S. Yajima, "High-speed parallel computation of an operation on finite groups using redundant representation," Report of Technical Group on Automata and Languages, IECE, AL84-75, Mar. 1985. (in Japanese)

[YASUT8504]
H. Yasuura, N. Takagi and S. Yajima, "Redundant coding and local computability in parallel computation," RIMS Koukyuroku 556, pp. 93-103, Research Institute for Mathematical Sciences, Kyoto Univ., Apr. 1985.

[TAKAY8505b]
N. Takagi and S. Yajima, "On-line error-detectable high-speed multiplier by redundant binary three-rail logic," Report of Technical Group on Fault Tolerant Systems, IECE, FTS85-3, May 1985. (in Japanese)

[OHKUY8512]
M. Ohkubo, H. Yasuura, N. Takagi and S. Yajima, "On unification hardware using UNION-FIND memory," Report of Technical Group on Automata and Languages, IECE, AL85-49, Dec. 1985. (in Japanese)

[OKABT8603]
Y. Okabe, N. Takagi and S. Yajima, "O(log n) depth n-bit binary divider using residue number system," Report of Technical Group on Automata and Languages, IECE, AL85-89, Mar. 1986. (in Japanese)

[TAKAY8603]
N. Takagi and S. Yajima, "High-speed binary division and square rooting methods using a redundant binary multiplier," Report of Technical Group on Automata and Languages, IECE, AL85-90, Mar. 1986. (in Japanese)

[TAKAO8606]
N. Takagi, Y. Okabe, H. Yasuura and S. Yajima, "Modulo m addition using redundant representation and its application to

residue-number / binary conversion," Report of Technical Group on Computation, IECE, COMP86-14, June 1986. (in Japanese)

[KAGAT8610]
T. Kagatani, N. Takagi and S. Yajima, "Algorithms for generating prime implicants of a logic function suitable for a vector processor," Report of Technical Group on Design Automation, IPSJ, 34-4, Oct. 1986. (in Japanese)

[ISHIT8703]
N. Ishiura, N. Takagi and S. Yajima, "Sorting on vector processors," Report of Technical Group on Computation, IEICE, COMP86-88, Mar. 1987. (in Japanese)

Convention Records (Referred in this thesis)

[TAKAA8603]
N. Takagi, T. Asada and S. Yajima, "An algorithm for computing arctangent based on CORDIC using redundant binary representation," Record of 1986 IECE National Convention, 1447, Mar. 1986. (in Japanese)

[TAKAY8703]
N. Takagi and S. Yajima, "On a partial product generating method for a binary multiplier with a redundant binary addition tree," Record of 34th IPSJ National Convention, 3N-3, Mar. 1987. (in Japanese)

[ASADT8703a]
T. Asada, N. Takagi and S. Yajima, "Algorithms based on CORDIC for calculating hyperbolic functions using redundant binary representation," Record of 34th IPSJ National Convention, 3N-4, Mar. 1987. (in Japanese)

[OKABT8703]
Y. Okabe, N. Takagi and S. Yajima, "Log depth circuits for elementary functions using residue number system," Record of 34th IPSJ National Convention, 3N-5, Mar. 1987. (in Japanese)

[ASADT8703b]
T. Asada, N. Takagi and S. Yajima, "Acceleration of a hardware algorithm for calculating sine and cosine using redundant

binary representation," Record of the 70th Anniversary IEICE National Convention, S7-5, Mar. 1987. (in Japanese)

IECE : The Institute of Electronics and Communication Engineers of Japan

IEICE : The Institute of Electronics, Information and Communication Engineers

IPSJ : Information Processing Society of Japan