数理解析研究所講究録
494巻 1983年 223-235

Hardware Algorithms for Division and Square Rooting Internally Using Redundant Binary Representation

Naofumi TAKAGI, Hiroto YASUURA and Shuzo Yajima
高木 直史　安浦 寛人　矢島 脩三
(Fucalty of Engineering, Kyoto University)

1.Introduction

Arithmetic operations are the most fundamental and significant operations in computer systems and other digital systems. Many hardware algorithms for arithmetic operations have been proposed, and some of them have been implemented and utilized in practical systems.(1) Especially, with the advances of technology of integrated circuits, high-speed hardware algorithms for arithmetic operations and their corresponding cellular arrays have been proposed.(2) In this paper, we propose VLSI-Oriented hardware algorithms for division and square rooting internally using redundant binary representation.

Many division algorithms have been proposed.(1) They are classified in two large groups, namely subtract-shift methods and multiplicative methods. Since combinational circuits for division based on subtract-shift methods have regular cellular array structures, they are suitable for VLSI implementation. However, they do not operate so fast for longer operands, because of borrow propagation in each subtraction. Multiplicative division can be performed by repetition of multiplication. Combinational circuits based on these methods operate rather fast. However, the amount of

hardware become too large. In Chapter 4, we propose a new high-speed subtract-shift division algorithm internally using the redundant binary representation. It requires the computation time of O(n) for n-bit division. A divider based on the algorithm has a regular cellular array structure and is suitable for VLSI implementation.

For square rooting, subtract-shift methods and multiplicative methods have been developed, similar to the case of division.(1) Combinational circuits based on conventional subtract-shift methods have regular cellular array structures and are suitable for VLSI implementation, but their computation speed is not so fast. Combinational circuits based on multiplicative methods are superior in computation speed, but they require a large amount of hardware. In Chapter 5, we propose a new subtract-shift method for square rooting internally using redundant binary representation. It requires the computation time of O(n) for n-bit square rooting. A square rooting circuit based on the algorithm has a regular cellular array structure and is suitable for VLSI implementation.

2.Preliminaries

Each hardware algorithm proposed in this paper is intended to be implemented as a combinational circuit. A combinational circuit is a logic circuit, which is constructed from given logic elements and has no feed back loop in it. We assume that fan-in of each logic element is restricted in a certain constant and fan-out is not restricted. We also assume that the computation time of the combinational circuit is linearly proportional to the depth of it. The depth of a combinational circuit is the number of logic

elements on the maximum path in it. The size of a combinational circuit is the number of logic elements in it. We also consider the area of a circuit on a VLSI chip. We assume that no logic elements overlap with each other and at most v ($v \geq 2$, a constant) wires can overlap with each other at any point on the chip.(3)

3 Redundant Binary Representation

The redundant binary representation utilized in this chapter is one of signed digit (SD) representations proposed by Avizienis.(4) It has a fixed radix r=2 and a digit set $\{\bar{1},0,1\}$, where $\bar{1}$ denotes -1. A redundant binary number $Y=[y_0.y_1\cdots y_n]_{SD2}$ has the value $\sum_{i=0}^{n} y_i*2^{-i}$. This is similar to a binary number. However, the redundant binary representation allows the existence of redundancy. There are several redundant binary representations to represent a number.

An unsigned binary number $X=[.x_1\cdots x_n]_2$ (x_i $\{0,1\}$) and a redundant binary number $Y=[.x_1\cdots x_n]_{SD2}$ have the same value $\sum_{i=0}^{n} x_i*2^{-i}$. Therefore, no computation is required to convert an unsigned binary number into an equivalent redundant binary number, where the equivalence implies that they have the same value. A redundant binary number Y, which is guaranteed to be positive, can be converted into the equivalent unsigned binary number by adding two binary numbers, Y^+ and Y^-, where Y^+ and Y^- are formed from the positive digits and the negative digits in Y, respectively.

In binary number system, parallel addition of two numbers by a combinational circuit requires computation time at least proportional to the logarithm of the word length of operands, because of the carry propagation.(1) However, in the redundant

binary number system, since carry propagation in addition can be eliminated by use of the redundancy, parallel addition of two numbers can be performed in constant time independent of the word length of operands.(4)(5) Parallel subtraction can be also performed in the constant time.

4 Division

In this chapter, we propose a new division algorithm internally using redundant binary representation, and consider a combinational circuit based on the algorithm. We are concerned with n-bit binary fraction division. We assume that the dividend X and the divisor Y are both normalized, i.e. $\frac{1}{2} \leq X < 1$ and $\frac{1}{2} \leq Y < 1$. The quotient Q satisfies $\frac{1}{2} < Q < 2$. We compute the quotient down to the n-th binary digit. We are not concerned with the final remainder.

Our division algorithm is one of subtract-shift methods. Subtract-shift divisions can be described by the following recursion formula:

$$R_{j+1} = 2*R_j - q_j*D,$$

where q_j is the quotient digit in the j-th binary position, $2*R_j$ is the partial dividend before the determination of q_j, and R_{j+1} is the partial remainder after the determination of q_j.

In the algorithm, we represent each R_j by a redundant binary representation $[r_0^j.r_1^j \cdots r_n^j]_{SD2}$, and select q_j from the digit set $\{\bar{1},0,1\}$ by examining the three most significant digits of R_j, and perform the computation of the recursion formula in the redundant binary number system. The algorithm is as follows.

Algorithm 1 (Division)

Step 1: Convert the dividend X and the divisor Y into redundant binary numbers R_0 and D.

Step 2: $q_0 := 1$,

$R_1 := R_0 - D$

(compute in the redundant binary number system)

Step 3: for j:=1 step 1 until n do

begin

$$q_j := \begin{cases} \bar{1} \text{ if } [r_0^j . r_1^j r_2^j]_{SD2} < 0 \\ 0 \text{ if } [r_0^j . r_1^j r_2^j]_{SD2} = 0 \\ 1 \text{ if } [r_0^j . r_1^j r_2^j]_{SD2} > 0 \end{cases}$$

$R_{j+1} := 2 * R_j - q_j * D$

(compute in the redundant binary number system)

end

Step 4: Convert $[q_0 . q_1 \cdots q_n]_{SD2}$ into the equivalent binary number Q.

Q is the quotient.

The conversion in Step 1 requires no computation. The computation in Step 2 can be performed in constant time independent of n. In Step 3, since q_j is determined by examining only the three most significant digits of R_j, the determination of each q_j can be done in constant time independent of n. The computation of the recursion formula also can be performed in constant time independent of n, because in the redundant binary number system, parallel addition / subtraction can be performed in constant time independent of the word length of operands. These computations are performed n times, so the computation time for Step 3 is O(n). The conversion in Step 4 can be performed in

$X = [\,0.1\,0\,0\,1\,1\,1\,0\,1\,]_2$

$Y = [\,0.1\,1\,0\,0\,0\,1\,0\,1\,]_2$

$q_0 = 1$ $\quad 0.1\,0\,0\,1\,1\,1\,0\,1$

$\quad +)\ 0.\bar{1}\,\bar{1}\,0\,0\,0\,\bar{1}\,0\,\bar{1}$

$q_1 = \bar{1}$ $\quad 0.0\,\bar{1}\,0\,1\,1\,0\,0\,0$

$\quad +)\ 0.1\,1\,0\,0\,0\,1\,0\,1$

$q_2 = 1$ $\quad 0.1\,0\,0\,\bar{1}\,1\,\bar{1}\,0\,1$

$\quad +)\ 0.\bar{1}\,\bar{1}\,0\,0\,0\,\bar{1}\,0\,\bar{1}$

$q_3 = 0$ $\quad 0.0\,0\,1\,0\,0\,1\,1\,\bar{1}$

Shift

$q_4 = 1$ $\quad 0.0\,1\,0\,0\,1\,1\,\bar{1}\,0$

$\quad +)\ 0.\bar{1}\,\bar{1}\,0\,0\,0\,\bar{1}\,0\,\bar{1}$

$q_5 = \bar{1}$ $\quad 0.\bar{1}\,1\,0\,1\,0\,0\,0\,\bar{1}$

$\quad +)\ 0.1\,1\,0\,0\,0\,1\,0\,1$

$q_6 = 1$ $\quad 0.1\,0\,\bar{1}\,0\,1\,\bar{1}\,1\,1$

$\quad +)\ 0.\bar{1}\,\bar{1}\,0\,0\,0\,\bar{1}\,0\,\bar{1}$

$q_7 = 0$ $\quad 0.0\,0\,0\,0\,0\,0\,1\,\bar{1}$

Shift

$q_8 = 0$ $\quad 0.0\,0\,0\,0\,0\,1\,\bar{1}\,0$

This part has no effect on the quotient.

$Q_8 = [\,1.\bar{1}\,1\,0\,1\,\bar{1}\,1\,0\,0\,]_{SD2}$

$\Downarrow$

$Q = [\,0.1\,1\,0\,0\,1\,1\,0\,0\,]_2$

Fig.1 An Example of Division

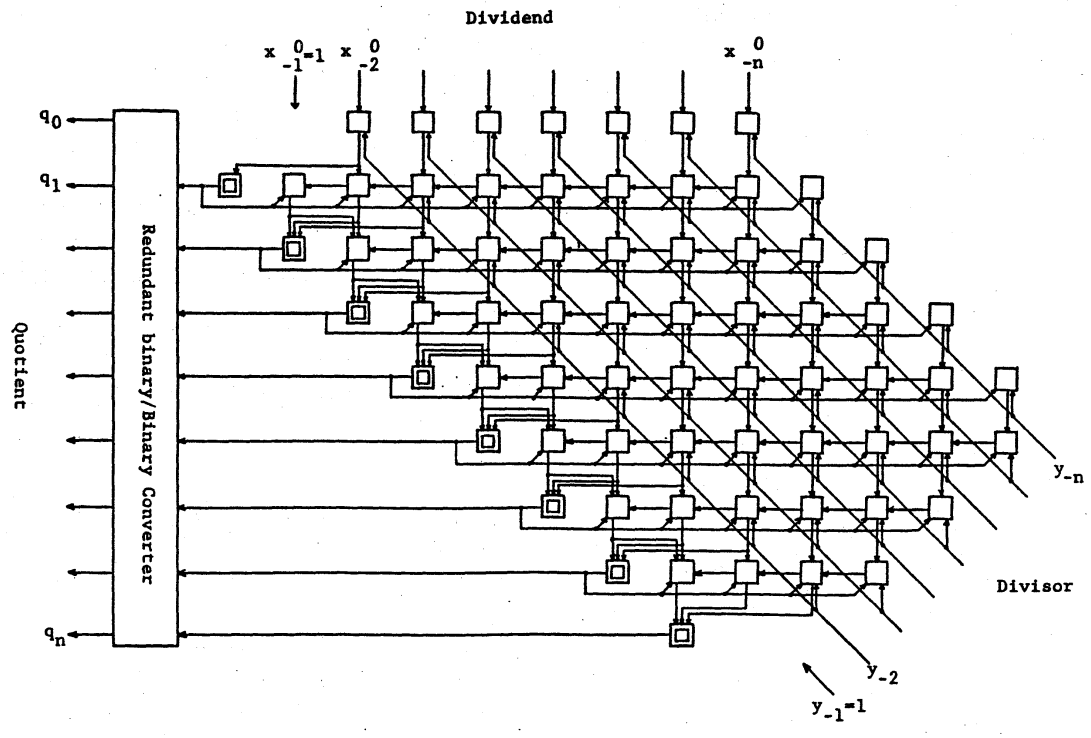

Fig.2 A Block Diagram of Our Divider

computation time of O(n) if a ripple carry adder is used, and in computation time of O(log n) if a carry-look-ahead adder is used. Thus, n-bit division can be performed in computation time of O(n).

Through the algorithm, each R_j satisfies $-D<R_j<D$. The difference between the obtained quotient Q and X/Y is smaller than 2^{-n}.

Fig.1 shows an example of division in accordance with the algorithm. Some of less significant digits in R_j $(j>\frac{n}{2})$ have no effect on the quotient, and the computation for these digits can be omitted.

Fig.2 shows a block diagram of a divider based on the algorithm. ⧈ denotes a cell for determining a quotient digit, and ☐ denotes a redundant binary add/subtract cell. The redundant binary / binary converter can be either a ripple-carry adder or a carry-look-ahead adder. The size of the divider is $O(n^2)$. As shown in Fig.1, the divider has a regular cellular array structure and is suitable for VLSI implementation. The chip area of the divider is $O(n^2)$.

Table 1 shows a comparison of several dividers. Table 2 shows examples of the depth and the size of them with use of 4-input NOR/OR's as logic elements. As shown in these tables, on the depth (i.e. on the computation time), our divider is superior to most of other dividers except a multiplicative divider using multipliers with parallel counters. On the amount of hardware, our divider is similar to a subtract-shift divider using ripple carry adders, and much superior to multiplicative dividers. Furthermore, since our divider has a regular cellular array structure as shown in Fig.2, it is more suitable for VLSI implementation, as well as a subtract-shift divider using ripple carry adders.

		Computation time (Depth)	Size	Area
Our divider		$O(n)$	$O(n^2)$	$O(n^2)$
Subtract-shift method	Ripple-carry adder	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Carry-look-ahead adder	$O(n \log n)$	$O(n^2 \log n)$ $O(n^2)$	$O(n^2 \log n)$
Multiplicative method	Array multiplier	$O(n \log n)$	$O(n^2 \log n)$	$O(n^2 \log n)$
	Multiplier using parallel counters	$O(\log^2 n)$	$O(n^2 \log n)$	$O(n^2 \log^2 n)$

Table 1 Comparison of Dividers

n		8	16	32	64	128
Our divider		36 685	70 2758	136 10939	264 43435	522 172936
Subtract-shift method	Ripple-carry adder	127 597	511 2717	2047 11565	8191 47693	32767 193677
	Carry-look-ahead adder	64 673	128 3103	320 14361	640 64475	1536 284421
Multiplicative method	Array multiplier	87 2640	244 16352	625 88128	1518 440704	3563 2106624
	Multiplier using parallel counters	66 4075	96 20573	150 89685	204 411653	280 1850355

4 input NOR/OR $\frac{\text{Depth}}{\text{Size}}$

Table 2 Depth and Size of Dividers

5 Square rooting

In this chapter, we propose a new subtract-shift method for square rooting internally using redundant binary representation, and consider a combinational circuit based on the algorithm. We are concerned with computing the square root Q of an n-bit

unsigned binary fraction X. We assume that X satisfies $\frac{1}{4}\leqq X<1$. Therefore, the square root Q satisfies $\frac{1}{2}\leqq Q<1$. We compute the square root down to the n-th binary position.

Our square rooting algorithm is in accordance with the following recursion formula:

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

where p_j is the square root digit in the j-th binary position, R_{j+1} is the partial remainder after the determination of p_j, and Q_j denotes $\sum_{i=1}^{j} q_j$.

In the algorithm, we represent each R_j by a redundant binary representation $[r_{j-2}^j r_{j-1}^j \cdots r_{2j}^j]_{SD2}$, and select p_j from the digit set $\{\bar{1},0,1\}$ by examining the three most significant digits of R_j, and perform the computation of the recursion formula in the redundant binary number system. The algorithm is as follows.

Algorithm 2 (Square Rooting)

Step 1: Convert X into a redundant binary number R_1.

Step 2: $p_1:=1$, $q_1:=2^{-1}$, $Q_1:=[0.1]_{SD2}$

$R_2:=R_1-2^{-2}$

(compute in the redundant binary number system)

Step 3: for j:=2 step 1 until n do

begin

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

$q_j:=2^{-j}*p_j$

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

(compute in the redundant binary number system)

$Q_j := Q_{j-1} + q_j$

end

Step 4: Convert Q_n into the equivalent binary number Q.

Q is the square root.

The conversion in Step 1 requires no computation. The computation in Step 2 can be performed in constant time independent of n. In Step 3, since p_j is determined by examining only the three most significant digits of R_j, the determination of each p_j can be done in constant time independent of n. The computation of the recursion formula also can be performed in constant time independent of n. These computations are performed n-1 times, so the computation time for Step 3 is O(n). The conversion in Step 4 can be performed in time of O(n) if a ripple-carry adder is used, and in time of O(log n) if a carry-look-ahead adder is used. Thus, n-bit square rooting can be performed in computation time of O(n).

The difference between the obtained square root Q and $\sqrt{X}$ is smaller than 2^{-n}.

Fig.3 shows an example of square rooting in accordance with the algorithm. Some of less significant digits in R_j ($j > \frac{3}{4}n$) have no effect on the square root, and the computation for these digits can be omitted.

Fig.4 shows a block diagram of a square rooting circuit based on the algorithm. ◘ denotes a cell for determining a square root digit, and □ denotes a redundant binary add/subtract cell. The redundant binary / binary converter can be either a ripple-carry

$X = [0.10001101]_2$

$P_1 = 1$ 0.10001101
+) 0.0$\bar{1}$
$P_2 = 1$ 0.0100
+) $\bar{1}0\bar{1}$
$P_3 = 0$ Shift 000$\bar{1}$11
$P_4 = \bar{1}$ 00$\bar{1}$1101
+) 1100$\bar{1}$
$P_5 = 1$ 011$\bar{1}$0000
+) $\bar{1}\bar{1}$010$\bar{1}$
$P_6 = 1$ 1$\bar{1}$01$\bar{1}$0$\bar{1}$00
+) $\bar{1}\bar{1}$01$\bar{1}$0$\bar{1}$ This part has
$P_7 = 1$ 010$\bar{1}$00$\bar{1}$00 no effect on
+) $\bar{1}\bar{1}$01$\bar{1}\bar{1}$0$\bar{1}$ the square root.
$P_8 = 0$ 000000$\bar{1}$

$Q_8 = [0.110\bar{1}1110]_{SD2}$

⇓

$Q = [0.10111110]_2$

Fig.3 An Example of Square Rooting

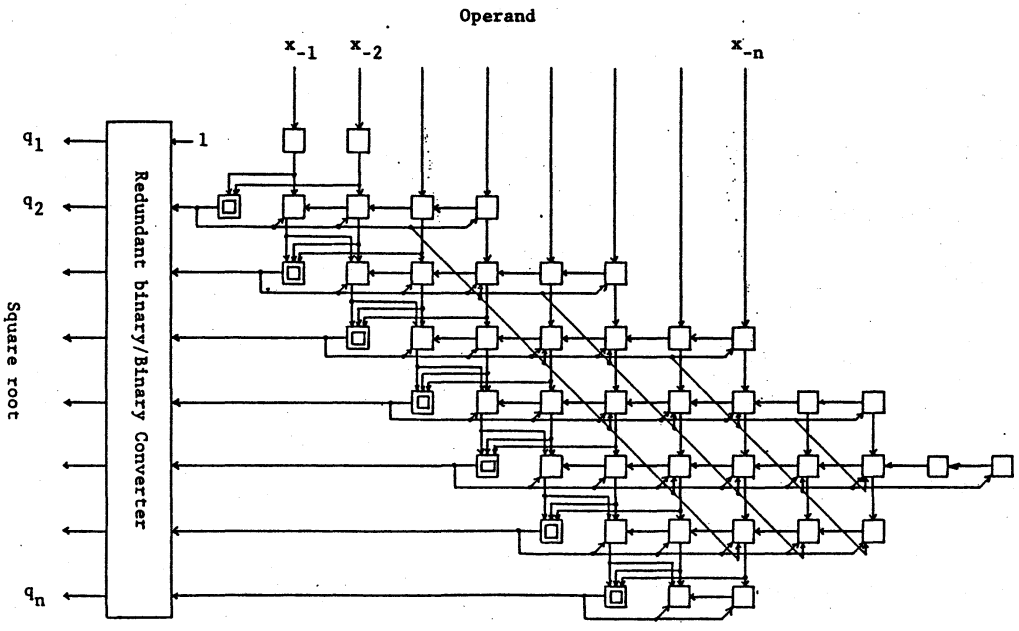

Fig.4 A Block Diagram of Our Square Rooting Circuit

		Computation time (Depth)	Size	Area
Our square rooting circuit		$O(n)$	$O(n^2)$	$O(n^2)$
Subtract-shift method	Ripple-carry adder	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Carry-look-ahead adder	$O(n \log n)$	$O(n^2 \log n)$ $O(n^2)$	$O(n^2 \log n)$
Multiplicative method	Array multiplier	$O(n \log n)$	$O(n^2 \log n)$	$O(n^2 \log n)$
	Multiplier using parallel counters	$O(\log^2 n)$	$O(n^2 \log n)$	$O(n^2 \log^2 n)$

Table 3 Comparison of Square Rooting Circuits

adder or a carry-look-ahead adder. The size of the square rooting circuit is $O(n^2)$. Since the circuit has a regular cellular array structure as shown in Fig.4, it is suitable for VLSI implementation. The chip area of the circuit is $O(n^2)$.

Table 3 shows a comparison of several square rooting circuits. As shown in the table, our square rooting circuit is superior to conventional subtract-shift ones in the depth (i.e. in the computation time), asymptotically. By a rough estimation, the depth of our square rooting circuit is smaller than those of conventional subtract-shift ones for $n \geq 8$. Some of multiplicative square rooting circuits are superior to ours on the computation time, asymptotically. But it seems that for practical n's, the depth of our square rooting circuits are smaller than that of multiplicative ones. On the amount of the hardware, our square rooting circuit is similar to a conventional subtract-shift one consisting of ripple carry adders and much smaller than those of multiplicative square rooting circuits.

6 Conclusion

We proposed VLSI-oriented hardware algorithms for division and square rooting internally using the redundant binary representation. A divider and a square rooting circuit based on these algorithms are excellent in both the computation speed and the regularity. Each of them can be also implemented as a sequential circuit with a redundant binary adder/subtracter and a shifter, and they operates rather efficiently.

The redundant binary representation can be used in hardware algorithms for other arithmetic operations.(5)

References

(1) K.Hwang : "Computer Arithmetic / Principles, Architecture and Design", John-Wiley & Sons, 1979.

(2) D.E.Agrawal : "High-Speed Arithmetic Arrays", IEEE Trans. Comput., vol.C-28, no.3, pp.215-224, Mar. 1979.

(3) H.Yasuura and S.Yajima : "On the Area of Logic Circuits in VLSI", Trans. IECEJ, vol.J65-D, no.8, pp.1080-1087, Aug. 1982. (in Japanese)

(4) A.Avizienis : "Signed-Digit Number Representations for Fast Parallel Arithmetic", IRE Trans. Elec. Comput., vol.EC-10, no.9, pp.389-400, Sept. 1961.

(5) N.Takagi, H.Yasuura and S.Yajima : "A VLSI-Oriented O(log n) Stage High-Speed Multiplier Using a Redundant Binary Addition Tree", Paper of Technical Group on Automata and Languages, AL82-31, Sept. 1982. (in Japanese)