# A Hardware Algorithm for Integer Division

Naofumi Takagi, Shunsuke Kadowaki and Kazuyoshi Takagi
Department of Information Engineering, Nagoya University
email: ntakagi@is.nagoya-u.ac.jp

***Abstract*— A hardware algorithm for integer division is proposed. It is based on the digit-recurrence, non-restoring division algorithm. Fast computation is achieved by the use of the radix-2 signed-digit representation. The algorithm does not require normalization of the divisor, and hence, does not require area-consuming leading one (or zero) detection nor shifts of variable-amount. Combinational (unfolded) implementation of the algorithm yields a regularly structured array divider, where pipelining is possible for increasing the throughput. Sequential implementation yields a compact divider.**

## I. Introduction

Integer division, i.e., division of two integers yielding an integer quotient and an integer remainder, is one of the basic arithmetic operations. In modern micro-processors, integer division takes many clock cycles, even more than double precision floating-point division, as shown in Table I [1]. Furthermore, the number of clock cycles for integer division varies depending on the operands' values.

In this paper, we propose a new hardware algorithm for integer division and discuss integer dividers based on the algorithm. The algorithm is based on the digit-recurrence, non-restoring division algorithm. In order to accelerate digit-recurrence division by dedicated hardware, it is quite natural to employ carry save or signed-digit representation for representing partial remainders and to perform addition/subtractions without carry/borrow propagation [2]. However, to do so, we have to normalize the divisor, and hence, require an area-consuming leading one (or zero) detector and a barrel shifter. Actually, Wang et al. recently proposed such integer dividers [3], [4], [5]. The number of clock cycles by these integer dividers varies depending on the operands' values.

In the hardware algorithm to be proposed, although we employ the radix-2 signed-digit (SD2) representation [6] for representing partial remainders, we do not normalize the divisor. We represent each partial remainder as a pair of its sign and absolute value and represent the latter in the SD2 representation. Each quotient digit is directly obtained from the sign of the corresponding partial remainder. Since the quotient is obtained as an ordinary binary number, the area-consuming on-the-fly conversion of the quotient is not necessary. Instead, we provide a much simpler mechanism for quotient adjustment to obtain the correct remainder. The calculation of the absolute value of each partial remainder is performed in parallel with the sign detection of the partial remainder and can be started before the completion of the sign detection of the preceding partial remainder.

TABLE I
Number of clock cycles of arithmetic operations in Pentium 4.

| operation | IMUL(32) | IDIV(32) | FDIV(s) | FDIV(d) |
|---|---|---|---|---|
| # clock cycles | $15-18$ | $56-70$ | 23 | 38 |

Combinational (unfolded) implementation of the algorithm yields a regularly structured array divider. The delay of the $n$-bit array divider is proportional to $n$ and the amount of hardware is proportional to $n^2$. Pipelining is possible for increasing the throughput. Sequential implementation yields a compact divider. A divider which internally produces $k$ quotient bits per clock performs $n$-bit division in $\frac{5n}{4k}+1$ clock cycles. The delay (clock period) is proportional to $k$ but independent of $n$, and the amount of hardware is proportional to $kn$. The number of required clock cycles is a constant depending only on $n$ and $k$ and is independent of the operands' values.

We have designed several dividers based on the proposed algorithm and estimated the delay and area of them. Compared with the sequential integer dividers in Synopsys DesignWare IP library [7], our sequential integer dividers are 15 to 30 % faster.

This paper is organized as follows. We will first define integer division, then briefly explain the non-restoring division algorithm, and then show the method for calculating the sign and the absolute value of an SD2 number, in the next section. In Section III, we will propose a new hardware algorithm for integer division. In Section IV, we will discuss implementations of the algorithm. In Section V, we will give concluding remarks.

## II. Preliminaries

### A. Integer division

The integer division we consider in this paper is as follows. Given a dividend $X$ and a divisor $Y$ ($\neq 0$) as $n$-bit two's complement binary integers, and obtain the quotient $Z$ and the remainder $R$ as $n$-bit two's complement binary integers, such that $X = Y \times Z + R$ and $R$ has the same sign as $X$. (There is an alternative definition where $R$ has the same sign as $Y$, which can be implemented more easily.) We assume that $X, Y \neq -2^{n-1}$.

### B. Non-restoring integer division

The hardware algorithm to be proposed is based on the radix-2 non-restoring integer division algorithm. Here, we describe the radix-2 non-restoring integer division algorithm.

Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)
1063-6889/05 $20.00 © 2005 IEEE

**[Algorithm IDIV]**
$D := |Y|$; $R_n := X$;
for $j := n-1$ downto 0 do
  if $R_{j+1} = 0$ then do
    $Q := [q_{n-1}q_{n-2}\cdots q_{j+1}0\cdots 0]$; $R := 0$;
    go to label;
  endif;
  if $R_{j+1} < 0$ then $q_j := -1$ else $q_j := 1$ endif;
  $R_j := R_{j+1} - q_j \cdot 2^j \cdot D$;
endfor;
$Q := [q_{n-1}q_{n-2}\cdots q_0]$;
if $X > 0$ and $R_0 < 0$ then $R := R_0 + D$; $Q := Q - 1$;
  elseif $X < 0$ and $R_0 > 0$ then $R := R_0 - D$; $Q := Q + 1$;
  else $R := R_0$;
endif;
label: if $Y < 0$ then $Z := -Q$ else $Z := Q$ endif; □

$q_j$ is selected from $\{-1, 1\}$ according to the sign of $R_{j+1}$, unless $R_{j+1} = 0$. $-2^j \cdot D < R_j < 2^j \cdot D$ always holds. $Q$ is obtained as a sequence of $-1$ and $+1$, until $R_{j+1} = 0$. We can obtain the ordinary two's complement binary representation of $Q$, $P = [p_{n-1}p_{n-2}\cdots p_0]$ ($p_j \in \{0, 1\}$), easily. Let $j_0$ be the $j$ such that $R_{j+1} = 0$. If there is no such $j$, let $j_0$ be $-1$. Then, for $j \geq j_0 + 2$, $p_j = 0$ if $q_{j-1} = -1$ and $p_j = 1$ if $q_{j-1} = 1$. Namely, $p_j = (1+q_{j-1})/2$. For $j = j_0+1$, $p_j = 1$, and for $j \leq j_0$, $p_j = 0$. We can see that $P = Q$ as follows.

$$\begin{aligned} P &= -2^{n-1}p_{n-1} + \sum_{j=0}^{n-2} 2^j p_j \\ &= -2^{n-1}(1+q_{n-2})/2 + \sum_{j=j_0+2}^{n-2} 2^j(1+q_{j-1})/2 + 2^{j_0+1} \\ &= -2^{n-2}q_{n-2} + \sum_{j=j_0+1}^{n-3} 2^j q_j \\ &= -2^{n-1}q_{n-1} + 2^{n-2}q_{n-2} + \sum_{j=j_0+1}^{n-3} 2^j q_j \\ &= Q \end{aligned}$$

Note that when $X \neq 0$, $q_{n-1} \cdot q_{n-2} = -1$.

### *C. Calculation of the sign and the absolute value of an SD2 integer*

The radix-2 signed-digit (SD2) representation belongs to the class of signed-digit representations [6] and has a digit set of $\{-1, 0, 1\}$. An $n$-digit SD2 integer $A = [a_{n-1}a_{n-2}\cdots a_0]$ ($a_i \in \{-1, 0, 1\}$) has the value $\sum_{i=0}^{n-1} 2^i a_i$.

In the SD2 system, we can perform addition/subtraction without carry/borrow propagation. In the algorithm to be proposed, we use subtraction of an ordinary binary integer from an SD2 integer, producing an SD2 integer. This subtraction is carried out through two stages. In the first stage, we determine the borrow bit $b_{i+1}$ ($\in \{0, 1\}$) and the intermediate difference bit $e_i$ ($\in \{0, 1\}$) according to the rule shown in Table II(a). In the second stage, we obtain the final difference digit ($\in \{-1, 0, 1\}$) by subtracting the borrow bit generated at the next lower position, $b_i$, from the intermediate difference bit, $e_i$, without generating new borrow as shown in Table II(b).

TABLE II
THE COMPUTATION RULE OF SUBTRACTION

(a) 1st stage

| subtrahend bit \ minuend digit | −1 | 0 | 1 |
|---|---|---|---|
| 0 | 1,1 | 0,0 | 0,1 |
| 1 | 1,0 | 1,1 | 0,0 |

$b_{i+1}$, $e_i$

(b) 2nd stage

| $b_i$ \ $e_i$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | −1 | 0 |

In the algorithm to be proposed, we use sign and absolute value calculation of an SD2 integer. We adopt the method proposed in [8]. We recall the method, here.

The sign of an SD2 integer is the sign of the most significant non-zero digit of it. Therefore, we can know the sign of an SD2 integer by scanning its digits from the most significant one. While the scanned digits are zero, the sign is undetermined. Once the most significant non-zero digit is found, the sign of the integer is determined as the sign of this digit. The absolute value of an SD2 integer is its negation when it is negative, and itself otherwise. Therefore, we can obtain the absolute value of an SD2 integer as an SD2 integer by inverting the signs of its all non-zero digits when the given integer is negative. This can be carried out concurrently with the sign detection. Scanning the digits from the most significant position, once the sign is determined as negative, we invert the sign of the non-zero digits.

The procedure for calculating the sign and the absolute value of an SD2 integer $A = [a_{n-1}a_{n-2}\cdots a_0]$ is as follows. The obtained $sgn(A)$ is $-1$ or 0 or 1 accordingly as $A$ is negative or zero or positive. $abs(A)$ is obtained as a positive SD2 integer.

**[Procedure SGNABS]**
$s_n := 0$;
for $i := n-1$ downto 0 do
  if $s_{i+1} = 0$ and $a_i = -1$ then $s_i := -1$;
    elseif $s_{i+1} = 0$ and $a_i = 1$ then $s_i := 1$;
    else $s_i := s_{i+1}$;
  endif;
  if $s_i = -1$ and $a_i \neq 0$ then $b_i := -a_i$; else $b_i := a_i$ endif;
endfor;
$sgn(A) := s_0$;
$abs(A) := [b_{n-1}b_{n-2}\cdots b_0]$; □

## III. A HARDWARE ALGORITHM FOR INTEGER DIVISION

Now, we propose a hardware algorithm for integer division. It is based on the radix-2 non-restoring integer division algorithm [Algorithm IDIV] described in Subsection II.B. We assume dividend $X$ is $[x_{n-1}x_{n-2}\cdots x_0]$ and divisor $Y$ is $[y_{n-1}y_{n-2}\cdots y_0]$.

In order to make the hardware algorithm have a regular structure, we continue the computation until $j = 0$, even if $R_{j+1} = 0$. The computation proceeds according to the value of $sign(R_{j+1})$. $sign(R_{j+1})$ is $-1$ if $R_{j+1}$ is negative, and $+1$ otherwise. Note that when $R_{j+1} = 0$, $sign(R_{j+1}) = +1$ and $+1$ is selected as $q_j$.

We adopt the technique used in [8] developed for accelerating CORDIC. To derive a new algorithm, as in [8], we introduce a series of variables $\hat{R}_j$'s, where

$$\hat{R}_j = sign(R_{j+1}) \cdot R_j,$$

and represent them in the SD2 representation. By the definition of $\hat{R}_j$,

$$|\hat{R}_j| = |R_j|$$

and

$$sign(\hat{R}_j) = \begin{cases} sign(R_{j+1}) \cdot sign(R_j) & if \quad R_j \neq 0 \\ +1 & if \quad R_j = 0 \end{cases}$$

hold.[1] Therefore, we can know the sign of the original $R_j$ by

$$sign(R_j) = \begin{cases} sign(R_{j+1}) \cdot sign(\hat{R}_j) & if \quad R_j \neq 0 \\ +1 & if \quad R_j = 0 \end{cases}$$

Since $R_j = R_{j+1} + 2^j \cdot D$ when $sign(R_{j+1}) = -1$ and $R_j = R_{j+1} - 2^j \cdot D$ otherwise,

$$\hat{R}_j = |\hat{R}_{j+1}| - 2^j \cdot D.$$

We can calculate $|\hat{R}_{j+1}|$ from $\hat{R}_{j+1}$ by [Procedure SGNABS] described in Subsection II.C. Therefore, we can calculate $\hat{R}_j$ independently of $sign(R_{j+1})$. Namely, we can begin the calculation of $\hat{R}_j$, as well as $|\hat{R}_j|$, before the sign of $R_{j+1}$ ($\hat{R}_{j+1}$) is detected.

We can perform the subtraction, $|\hat{R}_{j+1}| - 2^j \cdot D$, without borrow propagation as explained in Subsection II.C. We represent $\hat{R}_j$ and $|\hat{R}_j|$ in the $n + j$ digit SD2 representation. We apply 'pseudo-overflow correction' [8] to the result of the subtraction. Actually, we calculate $\hat{R}_j$ and $|\hat{R}_j|$ only down to the $j$th position, i.e., the upper $n$ digits. The less significant digits of $|\hat{R}_j|$ are identical to the corresponding bits of $X$ or the negation of them accordingly as $sign(R_j)$ is $+1$ or $-1$.

We can obtain $sign(R_j)$ from $sign(R_{j+1})$ and the sign of the upper part of $\hat{R}_j$ down to the $j$th position $\hat{R}'_j$, i.e., $sgn(\hat{R}'_j)$ where $sgn(\ )$ is the sign detection function in [Procedure SGNABS]. Namely, we can use $sgn(\hat{R}'_j)$ instead of $sign(\hat{R}_j)$. When $sgn(\hat{R}'_j) = +1$, since $\hat{R}_j > 0$, we let $sign(R_j) = sign(R_{j+1})$. When $sgn(\hat{R}'_j) = -1$, since $\hat{R}_j < 0$, we let $sign(R_j) = -sign(R_{j+1})$. When $sgn(\hat{R}'_j) = 0$ and $sign(R_{j+1}) = +1$, since $\hat{R}_j \geq 0$, we let $sign(R_j) = sign(R_{j+1})$, i.e., $+1$. When $sgn(\hat{R}'_j) = 0$ and $sign(R_{j+1}) = -1$, since $\hat{R}_j \leq 0$, we let $sign(R_j) = -sign(R_{j+1})$, i.e., $+1$. This guarantees that $sign(R_j) = sign(R_{j+1}) \cdot sign(\hat{R}_j)$ when $R_j \neq 0$ and $sign(R_j) = +1$ when $R_j = 0$.

For all $j$'s, $-2^j \cdot D \leq R_j < 2^j \cdot D$ holds. $R_0$ may take the value of $-D$, because we continue the computation even if $R_{j+1} = 0$. Therefore, we have to modify the quotient adjustment part of [Algorithm IDIV]. We calculate $\hat{R}^*_0 = |\hat{R}_0| - D$. When $X \geq 0$, if $R_0 < 0$, i.e., $sign(R_0) = -1$, we let $Q$ be $Q - 1$ and $R$ be $|\hat{R}^*_0|$, and otherwise, we let $Q$ be $Q$ and $R$ be $|\hat{R}_0|$. When $X < 0$, if $\hat{R}^*_0 = 0$, i.e., $sgn(\hat{R}^*_0) = 0$ (implying $R_0 = -D$), we let $Q$ be $Q - 1$ and $R$ be 0 ($= -|\hat{R}^*_0|$), and if $R_0 > 0$, i.e., $sgn(\hat{R}_0) \neq 0$ and $sign(R_0) = +1$, we let $Q$ be $Q + 1$ and $R$ be $= -|\hat{R}^*_0|$, and otherwise, we let $Q$ be $Q$ and $R$ be $-|\hat{R}_0|$.

We use $p_{j+1}$ ($\in \{0, 1\}$) instead of $q_j$. Furthermore, we merge the final negation process of the quotient into the main part. Namely, when $Y < 0$, we complement $p_{j+1}$.

The hardware algorithm is as follows.

**[Hardware Algorithm IDIV-HA]**
if $y_{n-1} = 0$ then $D := Y$ else $D := \bar{Y} + 1$ endif;
$\hat{R}_n := X$;
if $x_{n-1} = 1$ then $sign(R_n) := -1$
  else $sign(R_n) := +1$
endif;
for $j := n - 1$ downto 0 do
  if $sign(R_{j+1}) = -1$ then do
    if $y_{n-1} = 0$ then $p_{j+1} := 0$ else $p_{j+1} := 1$ endif;
  else do
    if $y_{n-1} = 0$ then $p_{j+1} := 1$ else $p_{j+1} := 0$ endif;
  endif;
  $\hat{R}_j := |\hat{R}_{j+1}| - 2^j \cdot D$;
  $|\hat{R}_j| := abs(\hat{R}_j)$;
  if $sgn(\hat{R}'_j) = -1$ or ($sgn(\hat{R}'_j) = 0$ and $sign(R_{j+1}) = -1$)
    then $sign(R_j) := -sign(R_{j+1})$
    else $sign(R_j) := sign(R_{j+1})$
  endif;
endfor;
$P := [p_{n-1}p_{n-2} \cdots p_1 1]$;
if $x_{n-1} = 0$ then do
  if $sign(R_0) = -1$ then do
    $R := |\hat{R}^*_0|$;
    if $y_{n-1} = 0$ then $Z := P - 1$ else $Z := P + 1$ endif;
  else do $R := |\hat{R}_0|$; $Z := P$;
  endif;
 else do
  if $sgn(\hat{R}^*_0) = 0$ then do
    $R := -|\hat{R}^*_0|$;
    if $y_{n-1} = 0$ then $Z := P - 1$ else $Z := P + 1$ endif;
  elseif $sgn(\hat{R}_0) \neq 0$ and $sign(R_0) = +1$ then do
    $R := -|\hat{R}^*_0|$;
    if $y_{n-1} = 0$ then $Z := P + 1$ else $Z := P - 1$ endif;
  else do $R := -|\hat{R}_0|$; $Z := P$;
  endif;
endif; □

$D$ is $Y$ itself when $Y > 0$ and $\bar{Y} + 1$ otherwise, where $\bar{Y}$ is the bitwise complement of $Y$. We treat the correction $+1$ bit as the borrow bit into the least significant position (i.e., $j$th position) in the calculation of $\hat{R}_j$.

We can perform the $-1$ adjustment of $P$ very easily by changing the least significant bit, which is alway 1, to 0. On the other hand, we can perform the $+1$ adjustment of $P$ by finding the minimum $j$, $j_1$, such that $p_j = 0$ and complementing $p_j$'s for $j \leq j_1$. To obtain the remainder $R$ in the ordinary binary representation, we need SD2 to binary conversion.

Figs. 1 through 4 show examples of integer division by [Hardware Algorithm IDIV-HA]. In the figures $\bar{1}$ denotes $-1$.

Fig. 1 shows the flow of integer division of positive dividend

[1] In [8], it is mentioned that $sign(\hat{R}_j) = sign(R_{j+1}) \cdot sign(R_j)$ holds. However, this is not true when $R_{j+1} < 0$ and $R_j = 0$.

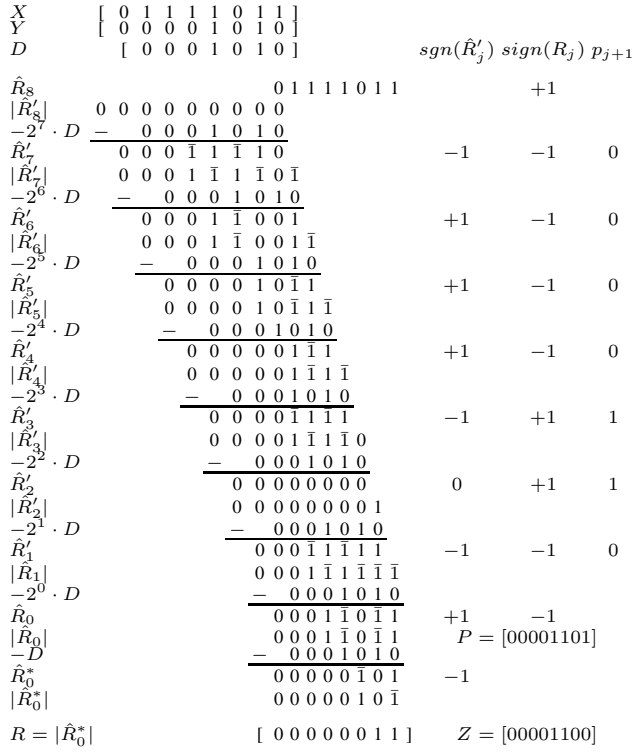


Fig. 1. An example of integer division (1): positive/positive

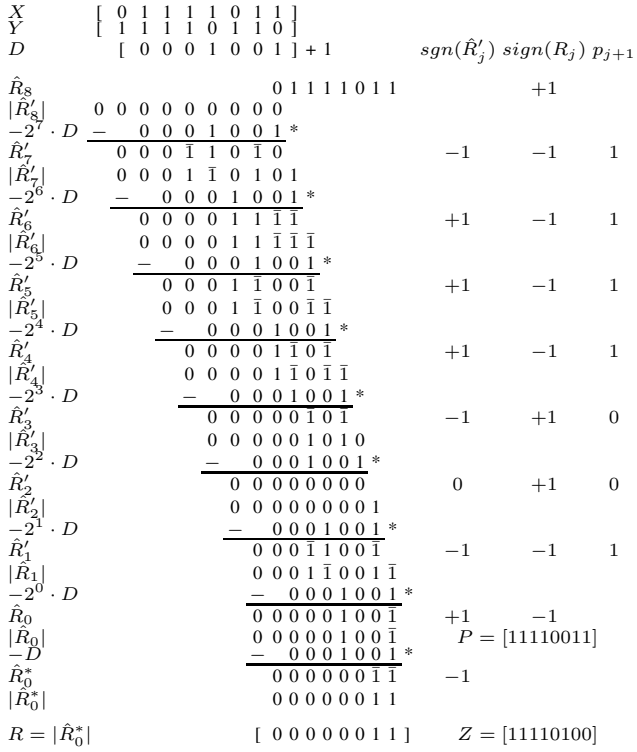


Fig. 3. An example of integer division (3): positive/negative

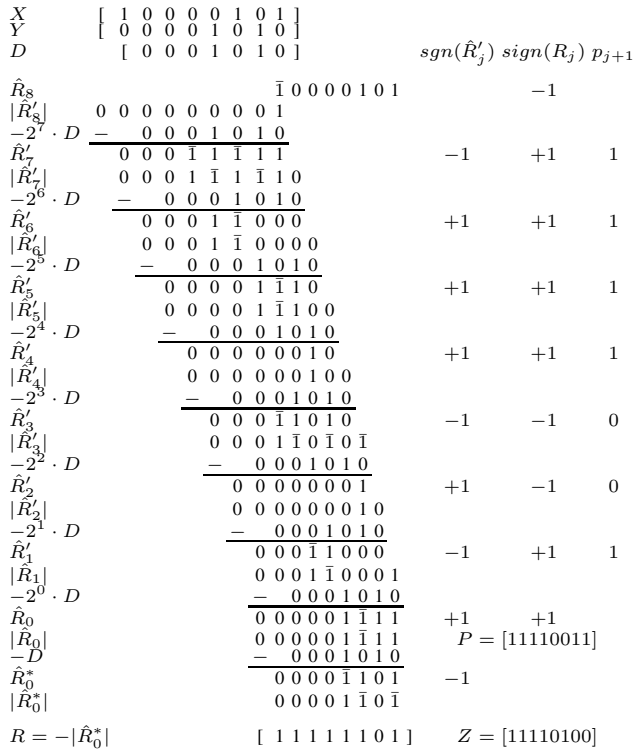


Fig. 2. An example of integer division (2): negative/positive

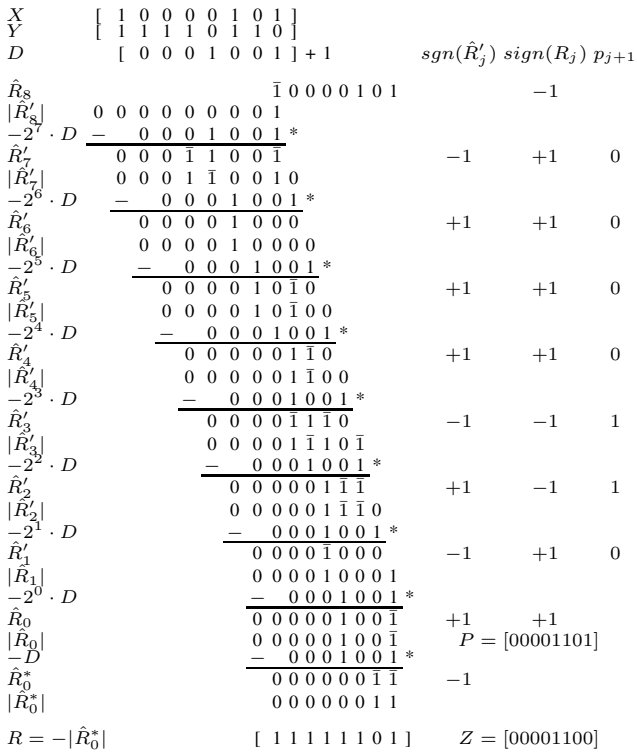


Fig. 4. An example of integer division (4): negative/negative

$X = 123$ by positive divisor $Y = 10$ yielding positive quotient $Z = 12$ and positive remainder $R = 3$.

Fig. 2 shows the flow of integer division of negative dividend $X = -123$ by positive divisor $Y = 10$ yielding negative quotient $Z = -12$ and negative remainder $R = -3$.

Fig. 3 shows the flow of integer division of positive dividend $X = 123$ by negative divisor $Y = -10$ yielding negative quotient $Z = -12$ and positive remainder $R = 3$. In Figs. 3 and 4, $*$ denotes the borrow bit input for the $+1$ correction of $D$.

Fig. 4 shows the flow of integer division of negative dividend $X = -123$ by negative divisor $Y = -10$ yielding

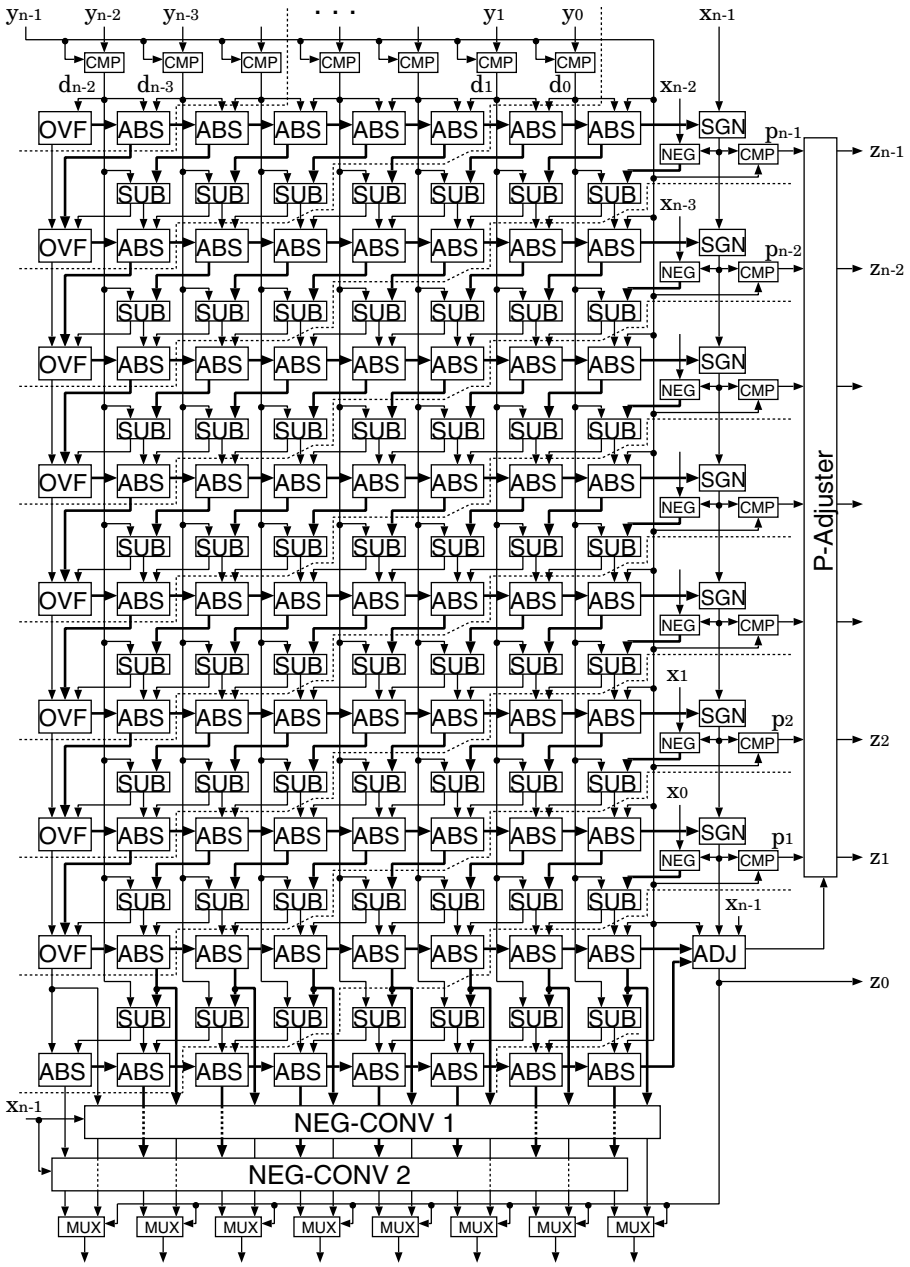


Fig. 5. A combinational implementation of the proposed integer divider

positive quotient $Z = 12$ and negative remainder $R = -3$.

## IV. Implementation consideration

### A. Combinational implementation

Combinational (or unfolded) implementation of the proposed hardware algorithm yields a regularly structured array divider as shown in Fig. 5.

The row of the $n-1$ CMP cells at the top of the array forms an complementer, which takes bit-wise complement of $Y$ when $Y$ is negative, i.e., when $y_{n-1} = 1$.

Each row of $n-1$ SUB cells and the following row of $n-1$ ABS cells, as well as the OVF cell at the leftmost and the SGN cell, the NEG cell and the CMP cell at the rightmost, form a circuit for computing one iteration of the main part of the hardware algorithm. Each SUB cell carries out the first stage of the subtraction of one bit position. Each ABS cell performs the sign and absolute calculation, as well as the second stage of the subtraction. The OVF cell compensates the pseudo-overflow. The SGN cell calculates $sign(R_j)$ and determines the quotient bit $p_j$. The NEG cell negates the dividend bit $x_{j-1}$ if $sign(R_j) = -1$. The CMP cell inverts $p_j$ if $y_{n-1} = 1$. Note that for the first iteration, a low of SUB cells is not necessary, because bits of $R_n$ down to the $n$th position are 0.

The ADJ cell generates the control signals for adjustment of the quotient and selection of the remainder.

The P-Adjuster performs $+1$ correction to $P$. We can reduce both the delay and the amount of hardware by using the fact that $p_j$'s are obtained sequentially from the most significant

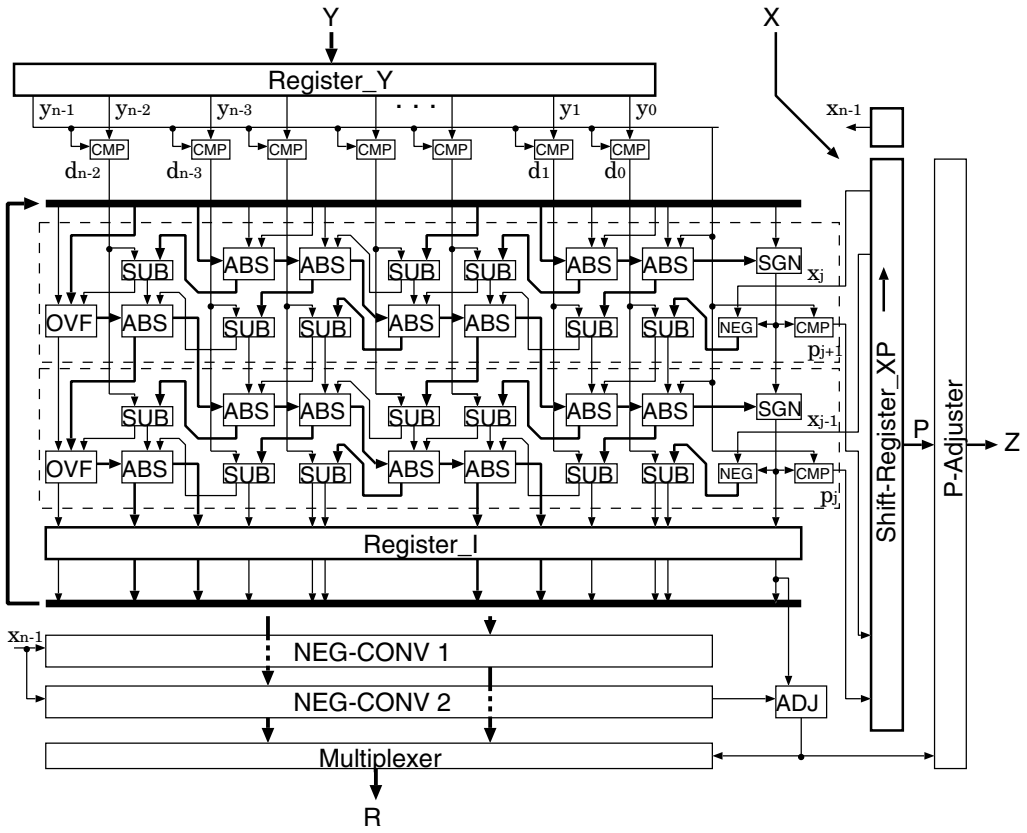


Fig. 6. A combinational implementation of the proposed integer divider

one, i.e., $p_{n-1}$ [9]. The circuit consists of several blocks of variable size. The number of blocks is proportional to $\log n$. The block size increases exponentially from the lower position. For example, block sizes are 3, 12, and 48 for $n = 64$.

NEG-CONV 1 (NEG-CONV 2) converts $|\hat{R}_0|$ ($|\hat{R}_0^*|$) itself or its negation accordingly as $x_{n-1}$ is 0 or 1, from SD2 to ordinary two's complement binary representation. The SD2 to binary converter is based on a carry-select adder. We can use the reduced-area scheme for carry-select adders [10]. We can also reduce both the delay and the amount of hardware by using the fact that digits of $|\hat{R}_0|$ ($|\hat{R}_0^*|$) are produced sequentially from the most significant one as in the P-Adjuster [9].

The delay of the $n$-bit array divider is proportional to $n$ and the amount of hardware is proportional to $n^2$.

Pipelining is possible for increasing the throughput. Each pipeline register should be inserted along one of the dotted lines in Fig. 5.

### B. Sequential implementation

Sequential implementation is also possible and yields a compact divider. Fig. 6 illustrates a sequential divider that internally produces two quotient bit per clock. Each circuit surrounded with broken lines is equivalent to the part of the circuit between two adjacent dotted lines in Fig. 5. Connecting $k$ copies of this circuit in cascade, we can construct the main part of a sequential divider which internally produces $k$ quotient bits per clock. (Fig. 6 shows the case that $k = 2$.) We can use the reduced area on-the-fly converter [11] for the NEG-CONV's for converting the remainder from SD2 to ordinary binary.

A divider, which internally produces $k$ quotient bits per clock, performs $n$-bit division in $\frac{5n}{4k} + 1$ clock cycles. This number of clock cycles is independent of the operands' values. The delay (clock period) is proportional to $k$ but independent of $n$, and the amount of hardware is proportional to $kn$.

### C. Estimation and comparison

We have designed several integer dividers based on the proposed hardware algorithm. We have described them in Verilog-HDL and have synthesized them by Synopsys Design Compiler using $0.35\mu m$ CMOS 3-metal technology provided by VLSI Design and Education Center (VDEC), the University of Tokyo, with the collaboration by Rohm Corporation [12].

Tables III(a) and (b) show the cell area, delay, number of clock cycles and computation time (delay $\times$ clock cycles) of 32-bit and 64-bit integer dividers, respectively. In the tables, '$k = 2$,''$k = 4$' and '$k = 8$ are sequential implementations of the proposed divider where 2, 4 and 8 quotient bits are internally produced per cycle, respectively. 'comb' is a combinational implementation of the proposed divider.

Tables IV(a) and (b) show the estimations of integer dividers in Synopsys DesignWare IP library [7] synthesized by using the same technology. DW_div_seq's are sequential dividers, while DW_div (rpl) and DW_div (cla) are combinational

TABLE III
ESTIMATION OF PROPOSED INTEGER DIVIDERS ($0.35\mu m$ CMOS)

(a) 32-bit

| Divider | Area[$mm^2$] | Delay[$ns$] | #cycle | comp. time[$ns$] |
|---|---|---|---|---|
| $k=2$ | 0.69 | 4.67 | 21 | 98.07 |
| $k=4$ | 0.91 | 7.74 | 11 | 85.14 |
| comb. | 4.33 | 77.54 | 1 | 77.54 |

(b) 64-bit

| Divider | Area[$mm^2$] | Delay[$ns$] | #cycle | comp. time[$ns$] |
|---|---|---|---|---|
| $k=2$ | 1.22 | 5.64 | 41 | 231.24 |
| $k=4$ | 1.75 | 8.23 | 21 | 172.83 |
| $k=8$ | 2.42 | 14.79 | 11 | 162.69 |
| comb. | 14.58 | 156.41 | 1 | 156.41 |

dividers with the ripple-carry synthesis model and the carry-look-ahead synthesis model, respectively.

TABLE IV
ESTIMATION OF INTEGER DIVIDERS OF SYNOPSIS'S DESIGNWARE ($0.35\mu m$ CMOS)

(a) 32-bit

| Divider | Area[$mm^2$] | Delay[$ns$] | #cycle | comp. time[$ns$] |
|---|---|---|---|---|
| DW_div_seq | 0.28 | 8.20 | 16 | 131.20 |
| | 0.39 | 13.40 | 8 | 107.20 |
| DW_div (rpl) | 0.65 | 246.20 | 1 | 246.20 |
| DW_div (cla) | 1.49 | 65.42 | 1 | 65.42 |

(b) 64-bit

| Divider | Area[$mm^2$] | Delay[$ns$] | #cycle | comp. time[$ns$] |
|---|---|---|---|---|
| DW_div_seq | 0.59 | 9.82 | 32 | 314.24 |
| | 0.83 | 17.83 | 16 | 285.28 |
| | 1.31 | 32.13 | 8 | 257.04 |
| DW_div (rpl) | 2.18 | 970.00 | 1 | 970.00 |
| DW_div (cla) | 5.33 | 155.14 | 1 | 155.14 |

Compared with the integer dividers in Synopsys DesignWare IP library, our sequential integer dividers are 15 to 30 % faster but about twice bigger. On the other hand, our combinational dividers are much faster but much bigger than those in the IP library with the ripple-carry synthesis model, and about the same in time but about three times bigger than those with the carry-look-ahead synthesis model. We feel that the combinational dividers in Synopsys DesignWare IP library are well optimized.

Table V shows the estimation of a 64-bit/32-bit mixed radix-16/8/4/2 integer divider proposed in [5] using TSMC $0.35\mu m$ 1P4M CMOS technology shown in Table III of [5].

TABLE V
ESTIMATION OF THE INTEGER DIVIDER OF [5] ($0.35\mu m$ CMOS)

64bit/32-bit

| Area[$mm^2$] | clock rate | #cycle | comp. time[$ns$] |
|---|---|---|---|
| 1.96 | 80MHz | 13–3 | 162.5–37.5 |

Although we cannot make a precise comparison because of the difference in the used technologies, we would like to point it out that our divider is smaller than Wang et al.'s.

## V. CONCLUDING REMARKS

We have proposed a hardware algorithm for integer division. It is based on the digit-recurrence, non-restoring division algorithm. We represent each partial remainder as a pair of its sign and absolute value and represent the latter in the SD2 representation. The calculation of the absolute value of each partial remainder is performed in parallel with the sign detection of the partial remainder and can be started before the completion of the sign detection of the preceding partial remainder. Since we do not normalize the divisor, area-consuming leading one (or zero) detection and shifts of variable-amount are not required. Each quotient digit is directly obtained from the sign of the corresponding partial remainder.

Combinational implementation of the algorithm yields a regularly structured array divider. The delay of the $n$-bit array divider is proportional to $n$ and the amount of hardware is proportional to $n^2$. Sequential implementation yields a compact divider. A divider which internally produces $k$ quotient bits per clock performs $n$-bit division in $\frac{5n}{4k}+1$ clock cycles. The delay (clock period) is proportional to $k$ but independent of $n$, and the amount of hardware is proportional to $kn$. The number of required clock cycles is a constant depending only on $n$ and $k$ and is independent of the operands' values.

## ACKNOWLEDGMENTS

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc., Cadence Design Systems, Inc., and Rohm Corporation.

## REFERENCES

[1] Intel Co.: 'A-32 Intel Architecture Optimization Reference Manual,' Order Number 248966-010, May, 2004.
[2] M. D. Ercegovac and T. Lang: "Division and Square Root – Digit-Recurrence Algorithms and Implementations," Kluwer Academic Publishers, 1994.
[3] C. Wang, C. Huang and G. Lin: 'Cell-based implementation of radix-4/2 64b dividend 32b divisor signed integer divider using the COMPASS cell library,' IEE Proceedings on Computers and Digital Techniques, vol. 147, no. 2, pp. 109–115, March 2000.
[4] C. Wang, C. Huang and I. Chang: 'Design and analysis of radix-8/4/2 64b/32b integer divider using COMPASS cell library,' VLSI System Design, vol. 11, no. 4, pp. 331–338, Dec. 2000.
[5] C. Wang, P. Lee, J. Wang and C. Huang: 'Design of a cycle-efficient 64-b/32-b integer divider using a table sharing algorithm,' IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol.11, no.4, pp.737–740, Aug. 2003.
[6] A. Avizienis: 'Signed-digit number representations for fast parallel arithmetic,' IRE Trans. Elec. Comput., vol. EC-10, no. 3, pp. 389-400, Sept. 1961.
[7] Synopsis Inc.: 'DesignWare Building Block IP,' 2003.
[8] H. Dawid and H. Meyr: 'The differential CORDIC algorithm: constant scale factor redundant implementation without correcting iterations,' IEEE Trans. Comput., vol. 45, no. 3, March 1996.
[9] N. Takagi and T. Horiyama: 'A high-speed reduced-size adder under left-to-right input arrival,' IEEE Trans. Comput., vol. 48, no. 1, pp. 76–80, Jan. 1999.
[10] A. Tyagi: 'A reduced-area scheme for carry-select adders,' IEEE Trans. Comput., vol. 42, no. 10, pp. 1163–1170, Oct. 1993.
[11] A. Nannarelli and T. Lang: 'Low-power divider,' IEEE Trans. Comput., vol. 48, no. 1, pp. 2–14, Jan. 1999.
[12] VLSI Design and Education Center (VDEC), the University of Tokyo: http://www.vdec.u-tokyo.ac.jp