A Floating Point Divider Performing IEEE Rounding and Quotient Conversion in Parallel

Woo-Chan Park[1], Tack-Don Han[2], and Sung-Bong Yang[2]

[1] Department of Internet Engineering,
Sejong University, Seoul 143-747, Korea,
pwchan@sejong.ac.kr
[2] Department of Computer Science,
Yonsei University, Seoul 120-749 Korea,
{hantack}@kurene.yonsei.ac.kr
{yang}@cs.yonsei.ac.kr

Abstract. Processing floating point division generally consists of SRT recurrence, quotient conversion, rounding, and normalization steps. In the rounding step, a high speed adder is required for increment operation, increasing the overall execution time. In this paper, a floating point divider performing quotient conversion and rounding in parallel is presented by analyzing the operational characteristics of floating point division. The proposed floating point divider does not require any additional execution time, nor does it need any high speed adder for the rounding step. The proposed divider can execute quotient conversion, rounding, and normalization within one cycle. To support design efficiency, the quotient conversion/rounding unit of the proposed divider can be shared efficiently with the addition/rounding hardware for floating point multiplier.

1 Introduction

An FPU (Floating Point Unit) is a principal component in graphics accelerators [1,2], digital signal processors, and high performance computer systems. As the chip integration density increases due to the advances in semiconductor technology, it has become possible for an FPU to be placed on a single chip together with the integer unit, allowing the FPU to exceed its original supplementary function and becoming a principal element in a CPU [2,3,4,5]. In recent microprocessors, a floating point division unit is built on a chip to speed up the floating point division operation.

In general, the processing flow of the floating point division operation consists of SRT recurrence, quotient conversion, rounding, and normalization steps [6,7,8]. SRT recurrence has been used to perform the division operation for the fraction part and to produce the final quotient and its remainder as in a redundant representation. In the quotient conversion step, the sign bit for the final remainder can be calculated from both the carry part and the sum part of the remainder in a redundant representation. Hence, a conventionally binary represented quotient is produced using the positive part and the negative part of the

© Springer-Verlag Berlin Heidelberg 2004

redundantly represented quotient and the sign bit of the final remainder. After that, the rounding step can be performed using the results from the quotient conversion step. For the rounding step, a high speed adder for increment operation is usually required, increasing the overall execution time and occupying a large amount of chip area.

In some microprocessors, due to design efficiency, the rounding unit for a floating point divider is shared with a rounding hardware for a floating point multiplier or a floating point adder [7,9]. The reasons for this sharing are as follows. First, because the floating point division operation requires many cycles to complete its operation, it does not need to be implemented by a pipeline structure. Second, because the floating point division operation is not used more frequently than other floating point operations, additional hardware for the rounding unit in the floating point divider is not economical. Third, the hardware to support the rounding operation for floating point division can be developed simply by modifying the rounding unit for either a floating point multiplier or a floating point adder. However, this sharing must not affect any critical path of the shared one and should not complicate the control scheme. Therefore, an efficient sharing mechanism is required.

In this paper, a floating point divider performing quotient conversion and rounding in parallel is proposed by analyzing the operational characteristics of floating point division. The proposed floating point divider does not require any additional execution time, nor does it need any high speed adder for the rounding step. Also it can execute quotient conversion, rounding, and normalization within only one cycle, and can share the addition/rounding hardware logics for a floating point multiplier presented in [11] by adding several hardware logics. The additive hardware does not affect the execution time of the floating point multiplier and can be implemented with very simple hardware logics.

In [9], quotient conversion and rounding can be performed in parallel, and the addition/rounding hardware logics are shared with a floating point multiplier presented in [10]. However, it requires a more complex processing algorithm. This in turn requires additional hardware components which increase the length of the critical path on the pipeline. Such increase causes one more unit of pipeline delay in their approach and requires more hardware components than our approach.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the IEEE rounding methods and the integer SRT division method. We also illustrate how the proposed floating point divider can share an addition/rounding unit of the floating point multiplier. Section 3 suggests a hardware model which can execute rounding and quotient conversion in parallel and its implementation with respect to IEEE four rounding modes. In Section 4, conclusion is given.

2 Backgrounds and Basic Equations

In this section, the IEEE rounding methods and the SRT division algorithm are discussed. An addition and rounding circuit for a floating point multiplier is also illustrated to be shared with the proposed floating point divider.

2.1 The IEEE Rounding Modes

The IEEE standard 754 stipulates four rounding modes; they are round-to-nearest, round-to-zero, round-to-positive-infinity, and round-to-negative-infinity. These four rounding modes can be classified mainly into round-to-nearest, round-to-zero, and round-to-infinity, because round-to-positive-infinity and round-to-negative-infinity can be divided into round-to-zero and round-to-infinity according to the sign of a number.

For the sake of the IEEE rounding, two additional bits, R and Sy, are required. R is the MSB among the less significant bits other than LSB. Sy is the ORed results of all the less significant bits except with R. The following three algorithms are the results of the rounding operation with LSB, R, and Sy when the MSB is zero, which is a normalized case. "return 0" means truncation, and "return 1" indicates increment as the result of any rounding operation.

Algorithm 1 : $Round_{nearest}(LSB, R, Sy)$

```
if ($R$=0) return 0
else if ($Sy$=1) return 1
    else if ($LSB$=0) return 0
        else return 1
```

Algorithm 2 : $Round_{zero}(LSB, R, Sy)$

```
return 0
```

Algorithm 3 : $Round_{infinity}(LSB, R, Sy)$

```
if (($R$=1) or ($Sy$=1)) return 1
else return 0
```

Assume that two input significands, *divisor* d and *dividend* x, have n bits each. To simplify the notation, the binary point is to be located between the LSB and the R bit positions. Then, the R and the Sy bit positions become the fraction portion. The significand bits above them, which are the most significant $(n + 1)$ bits, are the integer portion. The integer portion is represented with subscript I and the fractional portion with subscript T. Figure 1 shows that most significant $(n+1)$ bits of H are the integer portion H_I, while R and Sy in H are the fractional portion H_T.

For the rest of this paper, '$\wedge$' denotes the boolean AND, '$\vee$' denotes the boolean OR, and '$\oplus$' denotes the boolean exclusive-OR. X denotes the "don't care" condition. If any overflow is generated from the result of Z operation then $overflow(Z)$ returns 1, otherwise it returns 0. C_k^{in} denotes the value of the carry signal from the $(k-1)$-th bit position into the k-th bit position.

$$H = \overbrace{h_n h_{n-1} h_{n-2} \cdots h_1 h_0}^{H_I} . \overbrace{RSy}^{H_T}$$

Fig. 1. The definitions of H_I and H_T.

2.2 SRT Recurrence

Division operation can be defined by the following equation [6]:

$$x = q \times d + rem,$$

where $|rem| < |d| \times ulp$. The dividend x and the divisor d are the input operands. The quotient q and the remainder rem are the results of the division operation. The unit in the last position, denoted by ulp, defines the precision of the quotient, where $ulp = r^{-n}$ for n-digit and radix-r fractional results.

The following recurrence is used at each iteration:

$$\begin{aligned} rP_0 &= x \\ P_{j+1} &= rP_j - dq_{j+1}, \end{aligned}$$

where q_{j+1} is the $(j+1)$-th quotient digit, numbered from the highest to the lowest order, and P_j is the partial remainder at iteration j. In order for the next partial remainder P_{j+1} to be bounded, the value of the quotient-digit is chosen as follow:

$$|P_{j+1}| \leq d.$$

The final quotient is the weighted sum of all of the quotient-digits selected through the iteration such that

$$Q_{final} = \sum_{j=1}^{n} q_j \times r^{-j}.$$

In general, to speed-up the recurrence, the partial remainder and the quotient are represented as in the redundant forms. That is, the partial remainder consists of the carry part and the sum part, and the quotient consists of the positive part and the negative part.

2.3 Calculating the Final Quotient, *R*, and *Sy*

Suppose that the quotient obtained after the l-th iteration is denoted by Q_l and the partial remainder is denoted by P_l. Then because P_l should be positive, the restoration step for P_l is required to produce the final quotient if P_l is negative. Therefore, considering the restoration step, Q_l and P_l can be converted as follows.

$$\begin{aligned} \tilde{Q}_l &= Q_l - restore_l \\ \tilde{P}_l &= P_l + restore_l \cdot d \\ restore_l &= \begin{cases} 1 & \text{if } P_l < 0 \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

After the completion of SRT recurrence, the quotient and the remainder values are generated in the redundant binary representations. The quotient value from the result of SRT recurrence consists of the positive part Q^P and the negative part Q^N. Then Q^P and Q^N can be shown as follows.

$$\begin{aligned} Q^P &= p_n \cdots p_0 . p_{-1} \\ Q^N &= n_n \cdots n_0 . n_{-1}, \end{aligned}$$

where Q^N is represented in the one's complement form.

The remainder value generated after the completion of SRT recurrence consists of the carry part Rem_C and the sum part Rem_S. Both Rem_C and Rem_S are represented in the two's complement form and have length of $(n + 1)$ bits each. Rem_C and Rem_S can be now represented as follows.

$$\begin{aligned} Rem_S &= s_n \cdots s_1 s_0 \\ Rem_C &= c_n \cdots c_1 c_0 . \end{aligned}$$

The integer portions of Q^P and Q^N are denoted as Q_I^P and Q_I^N, respectively. Q_I^P and Q_I^N can be then defined as follows.

$$\begin{aligned} Q_I^P &= p_n \cdots p_0 \\ Q_I^N &= n_n \cdots n_0 . \end{aligned}$$

Then, because Q^N is represented in the one's complement form, when the values in the redundant representation converted into the conventional binary representation, '1' should be added to the position of n_{-1}. Also, if the remainder is negative, '1' should be borrowed from the quotient to restore the final remainder. Therefore, H which is the conventional binary representation of the final quotient including R and Sy can be calculated as follows.

$$\begin{aligned} H &= h_n h_{n-1} \cdots h_0 . RSy \\ &= (p_n \cdots p_0) + (n_n \cdots n_0) + 0.p_{-1} + 0.n_{-1} + \overline{restore_{-1}} + 0.0Sy, \end{aligned} \qquad (1)$$

where $restore_{-1}$ is the restoration signal to the (-1)-th position of the quotient and is identical to the sign value of the result of $(Rem_S + Rem_C)$. $Sy = 0$, if the result sum bits of $(Rem_S + Rem_C)$ are all zeros, otherwise $Sy = 1$.

Then, (1) can be converted as follows.

$$\begin{aligned} H &= Q_I^P + Q_I^N + 0.p_{-1} + 0.n_{-1} + 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + C_0^{in} + 0.RSy \\ R &= p_{-1} \oplus n_{-1} \oplus \overline{restore_{-1}} \\ C_0^{in} &= overflow(p_{-1} + n_{-1} + \overline{restore_{-1}}), \end{aligned}$$

where R is the round bit and C_0^{in} is the value of the carry signal from the (-1)-th bit position to the 0-th bit position. Hence, H_I, the integer portion of H, can be represented as follows.

$$H_I = Q_I^P + Q_I^N + C_0^{in} . \qquad (2)$$

2.4 The Addition/Rounding Unit for a Floating-Point Multiplier

In general, processing floating point multiplication consists of multiplication, addition, rounding, and normalization steps. A floating point multiplier in [11, 12,13] can execute the addition/rounding operation within only one pipeline stage and hence simplify the hardware design. The hardware model presented in [11] is shown in Figure 2. The proposed floating point divider can share the addition/rounding unit for a floating point multiplier to process the quotient conversion, the rounding, and the normalization steps.

Sy which is calculated at the previous pipeline stage in parallel with a wallace tree is included in the input factors of the *predictor* logic in [11]. Because *Sy* can be generated after SRT recurrence in floating point division, *Sy* must not be included among the input elements for the *predictor* logic to perform addition and rounding in parallel. This is considered in the proposed floating point divider which is given in next section.

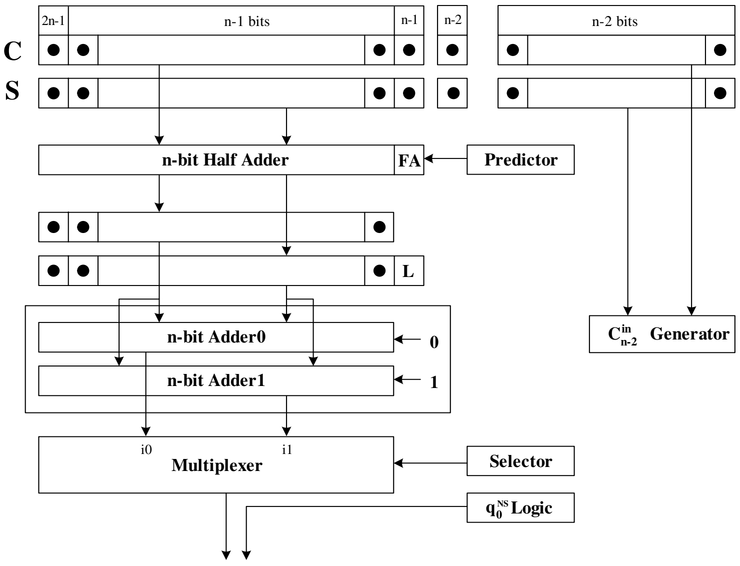

Fig. 2. The hardware model of the addition/rounding stage for a floating point multiplier.

3 The Proposed Floating-Point Divider

In this section, the operational characteristics in performing rounding and quotient conversion in parallel are illustrated. Based on these characteristics, a hardware model is presented. Finally, the proposed floating point divider is compared with respect to cycle time.

3.1 Analysis for the Quotient Conversion and Rounding Steps

Depending on the MSB of H, the shifting operation for normalization is performed. If $h_n = 1$ then no bit shifting for normalization is required, otherwise one bit shifting to the left should be performed in the normalization stage. The former case is denoted as NS (no shift) and the latter case is denoted as LS (left shift).

Normalization should be accounted for two different cases, i.e., the LS and the NS cases. In the LS case, the result value of H after normalization is denoted as O^{LS}. For the NS case, the result value of H after normalization is denoted as O^{NS}. Suppose that O_I^{LS}, O_R^{LS}, O_{Sy}^{LS} are the integer portion of O^{LS}, the R bit value, and the Sy bit value in the case of LS, respectively. Then they can be represented as follows.

$$\begin{aligned} O_I^{LS} &= h_{n-1}h_{n-2}\cdots h_0 \\ O_R^{LS} &= R \\ O_{Sy}^{LS} &= Sy. \end{aligned} \tag{3}$$

O_I^{NS}, O_R^{NS}, and O_{Sy}^{NS} are defined similarly for the NS case as follows:

$$\begin{aligned} O_I^{NS} &= h_n h_{n-1}\cdots h_1 \\ O_R^{NS} &= h_0 \\ O_{Sy}^{NS} &= R \vee Sy. \end{aligned} \tag{4}$$

Suppose that Q is the result value after the rounding step which is performed prior to the normalization stage. Then in the LS case, Q is represented by Q^{LS}. Thus, Q^{LS} can be written as follows according to (2) and (3):

$$\begin{aligned} Q^{LS} &= O_I^{LS} + Round_{mode}(h_0, R, Sy) \\ &= (h_{n-1}h_{n-2}\cdots h_0) + Round_{mode}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + C_0^{in} + Round_{mode}(h_0, R, Sy). \end{aligned} \tag{5}$$

Also, for the NS case, Q is represented by Q^{NS}. Thus, Q^{NS} can be obtained as follows according to (2) and (4).

$$\begin{aligned} Q^{NS} &= (O_I^{NS} + Round_{mode}(h_1, h_0, (R \vee Sy))) \times 2 \\ &= (h_n \cdots h_1 h_0) + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)) \\ &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)). \end{aligned} \tag{6}$$

3.2 The Proposed Hardware Model for Performing IEEE Rounding and Quotient Conversion in Parallel

A hardware model capable of performing rounding and quotient conversion in parallel is designed as shown in Figure 3. The proposed hardware model can be implemented by adding some hardware to the hardware model in Figure 2.

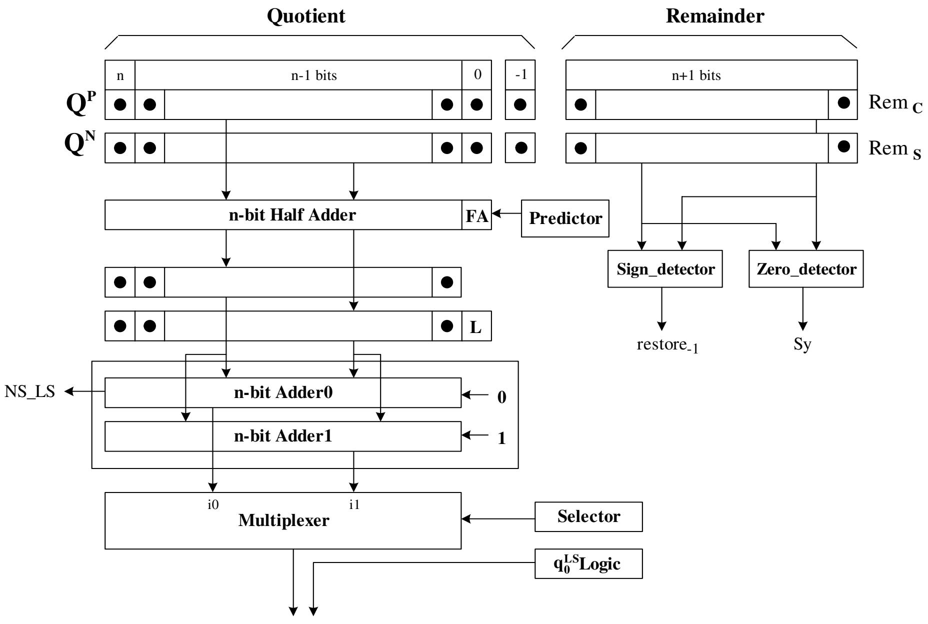

Fig. 3. The proposed hardware model for performing IEEE rounding and quotient conversion in parallel.

In *sign_detector*, the result value of $restore_{-1}$ in the (1) is calculated. The *zero_detector* logic determines whether the remainder is zero. If the result of *zero_detector* is zero, $Sy = 0$, otherwise $Sy = 1$. These logics can be implemented by adding additional logics to the C^{in}_{n-2} *generator* in Figure 2.

When Q^P_I and Q^N_I are added by the n bit HA and the one bit FA, the *predictor* bit is provided to the FA. Then the n bit carry and the $(n + 1)$ bit sum are generated. Here, the LSB of the sum is represented by L as shown in Figure 3. The n bit carry and the most significant n bit sum are added by a single carry select adder which is drawn as a dotted box in Figure 3. The *selector* selects one of the result values after executing addition and rounding from the two inputs $i0$ and $i1$. If $selector = 0$, then $i0$ is selected, otherwise $i1$ is selected as the output value of the multiplexer. The input values of $i0$ and $i1$ can be represented as follows.

$$\begin{aligned} i0 &= Q^P_I + Q^N_I + predictor \\ i1 &= Q^P_I + Q^N_I + 2 + predictor. \end{aligned} \tag{7}$$

In Figure 3, the multiplexer output may be either $(Q^P_I + Q^N_I + predictor)$ or $(Q^P_I + Q^N_I + predictor + 2)$, depending on the value of *selector*. According to (5) and (6), one of four possible cases, i.e., $(Q^P_I + Q^N_I)$, $(Q^P_I + Q^N_I + 1)$, $(Q^P_I + Q^N_I + 2)$, and $(Q^P_I + Q^N_I + 3)$ needs to be generated to perform rounding and quotient conversion in parallel. Therefore, if *predictor* and *selector* are properly selected, then the result value Q after performing addition and rounding in parallel can be generated. Note that Q is defined in Section 3.1.

To configure *predictor*, the following two factors should be considered. First, the input signals of *predictor* must be generated before any addition operation performed by the carry select adder. Second, the delay of the *selector* logic which is finally configured after the determination of the *predictor* logic should be negligible. Thus, *predictor* must be selected very carefully.

The input value of $i0$ in the multiplexer is denoted by $E = e_n e_{n-1} \cdots e_1$. Because the LSB position of E corresponds to the first bit positions of Q^P and Q^N, the integer value of E is $E_I = E \times 2$. Thus, the $(n+1)$ bit integer field can be denoted as E_I^* and $E_I^* = E_I + L$. Hence, E_I^* can be represented as follows:

$$E_I^* = E_I + L = Q_I^P + Q_I^N + predictor = e_n \cdots e_1 L. \tag{8}$$

In the next subsections, the rounding position is analyzed, and *predictor* and *selector* are determined according to all the three rounding modes.

3.3 The Round-to-Nearest Mode

In the round-to-nearest mode and the LS case, one of three possible cases, i.e., $(Q_I^P + Q_I^N)$, $(Q_I^P + Q_I^N + 1)$, and $(Q_I^P + Q_I^N + 2)$ needs to be generated according to (5) to perform rounding and quotient conversion in parallel.

In the NS case, for increment as the result of $Round_{Nearest}(h_1, h_0, (R \vee Sy))$, h_0 should be '1' according to Algorithm 1. If the result of rounding is increment and the NS case, '1' should be added to the position of h_1. But because h_0 should be '1' for increment as a result of rounding in the NS case, adding '1' to the position of h_1 has the same most significant n bit result when '1' is added to the position of h_0. Therefore, one of the three possible cases, i.e., $(Q_I^P + Q_I^N)$, $(Q_I^P + Q_I^N + 1)$, and $(Q_I^P + Q_I^N + 2)$, is required to be generated also in the NS case.

According to (7), when *predictor* is selected to '0', $(Q_I^P + Q_I^N)$ and $(Q_I^P + Q_I^N + 2)$ are generated. Because the most significant n bits of either $(Q_I^P + Q_I^N)$ or $(Q_I^P + Q_I^N + 2)$ are identical to the most significant n bits of $(Q_I^P + Q_I^N + 1)$, $(Q_I^P + Q_I^N + 1)$ can be generated. However, *predictor* is selected as follows to simplify the *selector* logic.

$$predictor = p_{-1} \wedge n_{-1}. \tag{9}$$

Then, (2) can be converted as follows.

$$\begin{aligned} H &= Q_I^P + Q_I^N + 0.p_{-1} + 0.n_{-1} + 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + overflow(p_{-1} + n_{-1}) + 0.(p_{-1} \oplus n_{-1}) + \\ &\quad 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + predictor + 0.(p_{-1} \oplus n_{-1}) + 0.\overline{restore_{-1}} + 0.0Sy \\ &= E_I + L + C_0^{in} + 0.R + 0.0Sy \\ &= E_I + C_1^{in} \times 2 + h_0 + 0.R + 0.0Sy \end{aligned} \tag{10}$$

$$\begin{aligned} R &= (p_{-1} \oplus n_{-1}) \oplus \overline{restore_{-1}} \\ C_0^{in} &= (p_{-1} \oplus n_{-1}) \wedge \overline{restore_{-1}} \\ h_0 &= L \oplus C_0^{in} \\ C_1^{in} &= L \wedge C_0^{in}. \end{aligned}$$

In the LS case, Q^{LS} can be expressed as follows according to (5) and (10).

$$Q^{LS} = E_I + C_1^{in} \times 2 + h_0 + Round_{nearest}(h_0, R, Sy). \tag{11}$$

Then, *selector* and q_0^{LS} can be produced as follows.

$$\begin{aligned} selector &= C_1^{in} \vee (h_0 \wedge Round_{nearest}(h_0, R, Sy)) \\ q_0^{LS} &= h_0 \oplus Round_{nearest}(h_0, R, Sy). \end{aligned}$$

In the NS case, Q^{NS} is as follows according to (6) and (10).

$$Q^{NS} = E_I + 2 \times C_1^{in} + h_0 + 2 \times Round_{nearest}(h_1, h_0, R \vee Sy). \tag{12}$$

Then, *selector* can be produced as follows.

$$selector = C_1^{in} \vee Round_{nearest}(h_1, h_0, R \vee Sy).$$

3.4 The Round-to-Zero Mode

It is considered that the *predictor* of the round-to-zero mode is identical to that of the round-to-nearest mode. Because $Round_{Zero}(X, X, X) = 0$ for both the LS and the NS cases, both *selector* and q_0^{LS} can be obtained by replacing both $Round_{Nearest}(h_0, R, Sy)$ and $Round_{Nearest}(h_1, h_0, R \vee Sy)$ of (11) and (12) with zeros. Therefore, *selector* and q_0^{LS} can be written as follows:

$$\begin{aligned} selector &= C_1^{in} \\ q_0^{LS} &= h_0. \end{aligned}$$

3.5 The Round-to-Infinity Mode

For a specific case such as $L = p_{-1} \oplus n_{-1} = C_0^{in} = Sy = 1$, if the *predictor* of the round-to-nearest mode is used, then $C_1^{in} = 1$, $h_0 = 0$, $R = 0$, and $Sy = 1$. Thus the rounding result in the round-to-nearest mode can be obtained by truncation according to Algorithm 1. However, because the value of Sy is equal to '1', the rounding result in the round-to-infinity mode can be obtained by increment according to Algorithm 3. Therefore, in the NS case, the case of $C_1^{in} = 1$ and $Round_{infinity} = 1$ can be occurred. Eventually, in the round-to-infinity mode, the *predictor* of the round-to-nearest mode cannot be used. Thus, the *predictor* is given as follows.

$$predictor = p_{-1} \vee n_{-1}. \tag{13}$$

Table 1. The result values of C_0^{in}, R, *predictor* according to the values of the p_{-1}, n_{-1}, $\overline{restore_{-1}}$.

p_{-1}	n_{-1}	$\overline{restore_{-1}}$	$p_{-1} \oplus n_{-1}$	C_0^{in}	R	$predictor$
0	0	0	0	0	0	0
0	1	0	1	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1
0	0	1	0	0	1	0
0	1	1	1	1	0	1
1	0	1	1	1	0	1
1	1	1	0	1	1	1

In Table 1, C_0^{in}, R, and *predictor* are illustrated as the values of p_{-1}, n_{-1}, and $\overline{restore_{-1}}$. The following two cases can be generated according to the values of $p_{-1} \oplus n_{-1}$ and $\overline{restore_{-1}}$. First, $\overline{restore_{-1}} = 0$ and $p_{-1} \oplus n_{-1} = 1$. In this case, C_0^{in} which represents the carry value to the position of h_0 is 0, both R and *predictor* are 1's. Second, the value of C_0^{in} has an identical value with *predictor* for all the cases except the first case.

In the first case, the result of rounding is increment because $R = 1$ according to Algorithm 3, $predictor = 1$ and $C_0^{in} = 0$. Then, Q^{LS} can be given as follows according to (5) and (8).

$$\begin{aligned} Q^{LS} &= Q_I^P + Q_I^N + C_0^{in} + Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + 1 = Q_I^P + Q_I^N + predictor \\ &= E_I + L. \end{aligned}$$

Then, *selector* and q_0^{LS} can be produced as follows in this case.

$$\begin{aligned} selector &= 0 \\ q_0^{LS} &= L. \end{aligned} \tag{14}$$

In the NS case, Q^{NS} can be written as follows according to (6) and (8).

$$\begin{aligned} Q^{NS} &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + 2 = Q_I^P + Q_I^N + predictor + 1 \\ &= E_I + L + 1. \end{aligned}$$

Hence, *selector* is as follows in this case.

$$selector = L.$$

In the second case, the value of C_0^{in} is identical to *predictor*. Then, Q^{LS} can be produced as follows.

$$\begin{aligned} Q^{LS} &= Q_I^P + Q_I^N + C_0^{in} + Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + predictor + Round_{infinity}(h_0, R, Sy) \\ &= E_I + L + Round_{infinity}(h_0, R, Sy). \end{aligned}$$

Then, in this case, *selector* and q_0^{LS} can be determined as follows.

$$\begin{aligned} selector &= L \wedge Round_{infinity}(h_0, R, Sy) \\ q_0^{LS} &= L \oplus Round_{infinity}(h_0, R, Sy). \end{aligned}$$

Thus, Q^{NS} is also obtained as follows.

$$\begin{aligned} Q^{NS} &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy) \\ &= Q_I^P + Q_I^N + predictor + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy) \\ &= E_I + L + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy). \end{aligned}$$

Hence, *selector* is represented as follows in this case.

$$selector = Round_{infinity}(h_1, h_0, R \vee Sy).$$

3.6 Critical Path Analysis

There are two dataflows in Figure 4. The critical path latency of the left hand side flow, denoted as L_{CP}^{div}, is ($predictor + FA +$ carry select adder $+ selector +$ multiplexer). The right hand side flow, denoted as R_{CP}^{div}, is ($zero\_detector + selector +$ multiplexer). In L_{CP}^{div} and R_{CP}^{div}, q_0^{LS} is ignored because both *selector* and q_0^{LS} can be performed simultaneously and the delay characteristic of *selector* is more complex than that of q_0^{LS}. As aforementioned in Section 2.3, the result values of $restore_{-1}$ and Sy can be calculated with the result of $Rem_C + Rem_S$. Because the latency of *zero_detector* is either equal to or longer than that of *sign_detector*, *sign_detector* is not also included in R_{CP}^{div}.

If we compare L_{CP}^{div} with R_{CP}^{div}, it seems that the carry select adder and *zero_detector* reveal similar delay characteristic, because both can be implemented with a high speed adder. Thus, the critical path of the hardware model in Figure 4 will be L_{CP}^{div}.

L_{CP}^{div} is almost similar to the critical path L_{CP}^{mul} of the hardware model in Figure 2. Only the delay characteristics of *predictor* and *selector* are somewhat different. The logic delay of *predictor* on L_{CP}^{div} is one gate delay for each rounding mode, otherwise that of L_{CP}^{mul} is two gate delays in the case of the round-to-infinity mode. For the *selector* of L_{CP}^{div}, an additional exclusive-OR gate is required due to identify the two cases in Table 1, as shown in Section 3.5. The additive gate delay of L_{CP}^{div} amount to the gate delays of exclusive-OR minus one gate delay. This additive delay is so small that it may not affect the overall pipeline latency.

3.7 Comparison with On-the-Fly Rounding

In [14], on-the-fly rounding was suggested to avoid a carry-propagation addition in rounding operation, by combining the rounding process with the on-the-fly conversion of the quotient digits from redundant to conventional binary form. The on-the-fly rounding requires one cycle for the rounding operation because the

Table 2. The execution cycles for double precision according to radix-r.

Processor	cycles	radix
PowerPC604e [3]	31	4
PA-RISC 8000 [15]	31	4
Pentium [7]	33	4
UltraSPARC [4]	22	8
R10000 [5]	19	16
Proposed floating point divider	30	4
Proposed floating point divider	21	8
Proposed floating point divider	15	16

rounded quotient is selected after the sign bit detection. This one cycle latency is equal to the latency for the rounding operation of the proposed architecture. But, the on-the-fly rounding requires four shift registers with somewhat complex parallel load operations.

On the other hand, the proposed architecture can share the addition/rounding hardware logics for a floating point multiplier presented in [14]. Thus, the proposed architecture seems to require less hardware than the on-the-fly rounding. Moreover, the proposed architecture achieves low-power consumption over the on-the-fly rounding, because the parallel loading operations for four registers could be generated at each iteration in case of on-the-fly rounding, while only one rounding operation is required in the proposed architecture.

3.8 Comparison with Other Microprocessors

To complete double precision floating point division, the SRT recurrences on the radix-4 case, on the radix-8 case, and on the radix-16 case take 29 cycles, 20 cycles, and 14 cycles, respectively. Because the proposed floating point divider can perform quotient conversion and rounding within only one cycle, to complete the floating point division operation, 30 cycles should be taken in the radix-4 case, 21 cycles for the radix-8 case, and 15 cycles for the radix-16 case, respectively. As shown in Table 2 in the radix-4 case, PowerPC604e [3] takes 31 cycles, PA-RISC 8000 [15] takes 31 cycles, and Pentium [7] takes 33 cycles to complete the floating point division operation. In the radix-8 case, UltraSPARC [4] takes 22 cycles. Also, R10000 [5] takes 19 cycles in the radix-16 case.

4 Conclusion

In this paper, a floating point divider which is capable of performing the IEEE rounding and addition in parallel is proposed. Its hardware model is provided and evaluated with the proofs for correctness of the model. The performance improvement and cost effectiveness design for floating point division can be achieved by this approach.

References

1. M. Kameyama, Y. Kato, H. Fujimoto, H. Negishi, Y. Kodama, Y. Inoue, and H. Kawai. 3D graphics LSI core for mobile phone “Z3D”. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware, pages 60–67, 2003.
2. S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. IEEE Micro, 19(2):37–48, April 1999.
3. S. P. Song, M. Denman, and J. Chang. The powerPC604 RISC microprocessor. IEEE Micro, 14(5):8–17, Oct. 1994.
4. M. Tremblay and J. M. O’Connor. Ultra SPARC I : A four-issue processor supporting multimedia. IEEE Micro, 16(2):42–50, April 1996.
5. K. C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, 16(2):28–40, April 1996.
6. M. D. Ercegovac and T. Lang. Division and square root: digit recurrence algorithms and implementations. (Kluwer Academic Publishers, 1994).
7. D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. IEEE Micro, 13(3):11–21, June 1993.
8. C. H. Jung, W. C. Park, T. D. Han, S. B. Yang, and M. K. Lee. An effective out-of-order execution control scheme for an embedded floating point coprocessor. Microprocessors and Microsystems, 27:171–180, April 2003.
9. J. A. Prabhu and G. B. Zyner. 167 MHz Radix-8 divide and square root using overlapped radix-2 stages. In Proceedings of the 12th IEEE Symposium on Computer Arithmetic, pages 155–162, July 1995.
10. J. Arjun Prabhu and Gregory B. Zyner. 167 MHZ radix–4 floating point multiplier. In Proceedings of the 12th IEEE Symposium on Computer Arithmetic, pages 149–154, July 1995.
11. W. C. Park, T. D. Han, and S. D. Kim. Floating point multiplier performing IEEE rounding and addition in parallel. Journal of Systems Architecture, 45:1195–1207, July 1999.
12. G. Even and P. M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. IEEE Transactions on Computers, 49(7):638–650, July 2000.
13. M. R. Santoro, G. Bewick, and M. A. Horowitz. Rounding algorithms for IEEE multiplier. In Proceedings of the 9th IEEE Symposium on Computer Arithmetic, pages 176-183, 1989.
14. M. D. Ercegovac and T. Lang. On-the-fly Rounding. IEEE Transactions on Computers, 41(12):1497–1503, Dec. 1992.
15. P. Solderquist and M. Leeser. Division and square root: choosing the right implementation. IEEE Micro, 17(4):56–66, Aug. 1997.