A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture

Tomás Lang, *Member*, *IEEE Computer Society*, and
Alberto Nannarelli, *Member*, *IEEE Computer Society*

Abstract—In this work, we present a radix-10 division unit that is based on the digit-recurrence algorithm. The previous decimal division designs do not include recent developments in the theory and practice of this type of algorithm, which were developed for radix-2^k dividers. In addition to the adaptation of these features, the radix-10 quotient digit is decomposed into a radix-2 digit and a radix-5 digit in such a way that only five and two times the divisor are required in the recurrence. Moreover, the most significant slice of the recurrence, which includes the selection function, is implemented in radix-2, avoiding the additional delay introduced by the radix-10 carry-save additions and allowing the balancing of the paths to reduce the cycle delay. The results of the implementation of the proposed radix-10 division unit show that its latency is close to that of radix-16 division units (comparable dynamic range of significands) and it has a shorter latency than a radix-10 unit based on the Newton-Raphson approximation.

Index Terms—Decimal arithmetic, digit-recurrence division, decimal division, algorithms and architectures for floating-point arithmetic.

1 Introduction

The need for floating-point decimal arithmetic in hardware has been reported in several publications (some recent references being [1], [2], [3], [4]). Moreover, radix-10 units are included in some processors (although not floating-point) [5] and some recent designs for adders, multipliers, dividers, and square-root units have been described in [3], [4], [5], [6], [7].

In this work, we present a radix-10 division unit that is based on the digit-recurrence algorithm. Digit-recurrence algorithms for division and square root give probably the best trade-off in delay area, do not require the use of a multiplier, and easily provide exact rounding. Previous designs of decimal dividers are reported in [4], [5] and in several patents [8], [9], [10], [11], [12], [13]. The previous digit-recurrence designs are relatively old and do not include recent developments in the theory and practice of this type of algorithm. In particular, the recent IBM z900 processor includes a decimal divider using the restoring algorithm [5]. On the other hand, [4] describes a recent implementation based on the Newton-Raphson approximation of the reciprocal. We compare with this implementation in Section 5 and show that the digit-recurrence approach might produce a lower latency.

The design described in this paper includes the following novelties, with respect to previous designs:

- Adaptation of the techniques developed for radix-2^k dividers such as
  1. signed-digit redundant quotient,
  2. carry-save representation of the residual,
  3. decomposition of the quotient digit,
  4. realization of the quotient-digit selection by preloading the selection constants and then comparing them with an estimate of the residual,
  5. overlapped implementation of the selection of both components of the quotient digit,
  6. retiming and separate implementation of the most significant digits (slice), and
  7. on-the-fly conversion, normalization, and rounding.
- More specifically, decomposition of the quotient digit into one radix-5 component and one radix-2 component, with the radix-5 component having values $\{-2, -1, 0, 1, 2\}$. This results in a redundancy factor of $7/9$ and requires only multiples of the divisor d, $2d$, and $5d$.
- Radix-2 algorithm and implementation of the most significant slice to reduce the critical path delay. The retiming ensures that the critical path is in this slice so that an efficient binary implementation reduces the cycle time.

With respect to the floating-point format, several recent papers discuss the issue [1], [2]. Of particular interest is the specification that has been added to the revision of the IEEE 754 Standard [14]. In this paper, we assume normalized and fractional significands for operands and result. Moreover, we use a binary coded decimal (BCD) coding for the decimal digits. For other representations and/or coding, it is possible to use conversions or to adapt the algorithm and implementation. Specifically, [4] describes an implementation of the modules required for compliance with the IEEE 754 Standard.

- *T. Lang is with the Department of Electrical Engineering and Computer Science, Engineering Tower, Room 602, University of California, Irvine, CA 92697. E-mail: tlang@uci.edu.*
- *A. Nannarelli is with the Department of Informatics and Mathematical Modeling, Technical University of Denmark, Richard Petersens Plads-Building 321, DK-2800 Kongens Lyngby, Denmark. E-mail: an@imm.dtu.dk.*

Manuscript received 4 apr. 2006 revised 25 Sept. 2006; accepted 8 Jan. 2007; published online 27 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0131-0406.
Digital Object Identifier no. 10.1109/TC.2007.1038.

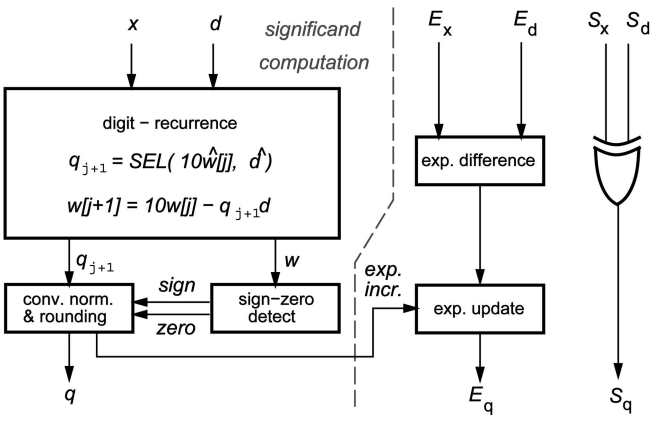

Fig. 1. Structure of the floating-point radix-10 division.

We show the proposed algorithm and architecture for the divider unit. We then implement the design by using a 90-nm complementary metal-oxide semiconductor (CMOS) standard cells library and Synopsys tools. Moreover, we use the same design tools and cells to implement two radix-16 units, namely, the basic retimed radix-16 and a faster radix-16 unit, as presented in [15], and show that the radix-10 divider has a latency similar to that of the basic retimed radix-16 unit.

Finally, we compare with the design based on Newton-Raphson iterations [4] and conclude that the digit-recurrence approach that we propose offers the shortest latency for decimal division among the units presented so far.

2 Operation and Algorithm

We perform the division operation $Q = X/D$ in which the operands and result have a floating-point representation such that $X = (-1)^{S_x} x \cdot 10^{E_x}$, $D = (-1)^{S_d} d \cdot 10^{E_d}$, and $Q = (-1)^{S_q} q \cdot 10^{E_q}$. The corresponding structure of the operation is shown in Fig. 1. We now concentrate on the digit-recurrence algorithm to obtain the significand of the quotient. To be specific, we perform decimal fractional division, with x and d normalized such that $0.1 \leq x, d < 1$ (see Table 1 for the notation used). We also produce a normalized quotient. The modification of the algorithm and the implementation for other cases are straightforward.

2.1 Background

We now summarize the theory and development of the radix-r digit-recurrence algorithm, as described in more detail, for example, in [16]. This will be the basis for the radix-10 case that we propose.

Given the significands of the dividend x and of the divisor d, we produce the significand of the quotient q such that

$$x = q \cdot d + rem.$$

If the quotient has a maximum relative error of 1 *ulp*, then $rem \leq d \cdot ulp$. Moreover, rounding (round-half-even mode) produces a quotient with a maximum relative error of 0.5 *ulp*.

In the digit-recurrence algorithm, the quotient is produced one digit per iteration of the following radix-r recurrence on the residual:

$$w[j+1] = rw[j] - q_{j+1}d, \tag{1}$$

with initial condition $w[0] = x$ and with the quotient digit obtained by the selection function

$$q_{j+1} = SEL(\widehat{rw[j]}, \widehat{d}), \tag{2}$$

where $\widehat{rw[j]}$ and $\widehat{d}$ are estimates of $rw[j]$ and d, respectively.

TABLE 1
Symbols and Notation

radix-10	radix-2	name
x		dividend
d		divisor
q		quotient
$r = 10$		radix
q_{j+1}		quotient digit at iteration j
a		$q_{j+1} \in [-a, a]$
$\rho = a/(r-1)$		redundancy factor
q_{Hj+1} and q_{Lj+1}		$q_{j+1} = 5q_{Hj+1} + q_{Lj+1}$, $q_H = \{-1, 0, 1\}$ and $q_L = \{-2, -1, 0, 1, 2\}$
a_L		$q_{Lj+1} \in [-a_L, a_L]$
$w[j]$		residual at iteration j
w_s, w_c	W_s, W_c	carry-save representation of w
e		error in the estimate $\widehat{w}[j]$ due to using a carry-save representation
$w_{(i)}$		digit of weight 10^{-i}
$v[j]$		$v[j] = 10w[j] - 5q_{Hj+1}d$
$[L_k, U_k]$		selection interval for $q_{j+1} = k$
$m_k(i)$		selection constant for $q_j = k$ and the subinterval $[d_i, d_{i+1})$
$\widehat{(...)}$		estimate of (...)
$trunc_3(...)$		(...) truncated up to third fractional digit
$w_m[j]$	$W[j]$	$10^3(trunc_3(w[j])) = w_{(1)}w_{(2)}w_{(3)}$
$y_m[j]$	$Y[j]$	$10w_m[j] + w_{(4)}$
$(q_{Hj} \cdot 5d)_m$	$(q_{Hj}5D)$	$10^3(trunc_3(q_{Hj} \cdot 5d))$
$(q_{Lj}d)_m$	$(q_{Lj}D)$	$10^3(trunc_3(q_{Lj}d))$
b_j		BCD digit converted from quotient digit q_j
$b_{(i)}$		BCD digit in position i in normalized quotient Q_{norm}

	$\cdots$	$w_{s(i-1)}$	$w_{s(i)}$	$w_{s(i+1)}$	$\cdots$
$w_s[j]$:	$\cdots$	ssss	ssss	ssss	$\cdots$
$w_c[j]$:	$\cdots$	c	c	c	$\cdots$
	$\cdots$	$w_{c(i-1)}$	$w_{c(i)}$	$w_{c(i+1)}$	$\cdots$

Fig. 2. Radix-10 carry-save representation of w (ssss corresponds to a digit in BCD representation).

To allow for the use of estimates in the selection function, it is necessary to use a redundant digit set for the quotient digit. For a symmetrical (and continuous) digit set $-a \leq q_j \leq a$, the redundancy factor is defined as $\rho = a/(r-1)$. To assure convergence, the selection function should satisfy the range of the residual, namely, $|w[j]| \leq \rho d$. Moreover, to speed up the subtraction, a redundant adder is used.

For the selection function, the divisor is partitioned into subranges $[d_i, d_{i+1})$ such that, for subrange i, the selection function is defined by the selection constants $m_k(i)$ so that the quotient digit has value k if

$$m_k(i) \leq \widehat{rw} < m_k(i+1).$$

The selection constants are selected so that the range of the residual is bounded. This is achieved by the condition

$$L_k(d_{i+1}) \leq m_k(i) \leq U_{k-1}(d_i) - e,$$

where $[L_k, U_k]$ is the selection interval for $q_j = k$, with $L_k(d_i) = (k-\rho)d_i$ and $U_k = (k+\rho)d_i$. Moreover, e is the maximum error introduced by using $\widehat{rw}$ instead of a truncated $rw[j]$. This expression is specific to the case in which the estimate is obtained by truncating the carry-save representation of $rw[j]$. In this case, $0 \leq trunc(rw[j]) - \widehat{rw[j]} \leq e$.

2.2 Proposed Algorithm

We now describe the algorithm that we propose. As is usually done for algorithms using signed values and an addition or subtraction-based recurrence, we use a range-complement (equivalent to two's complement in radix 2) representation of these signed values. For the radix-10 case, to represent a signed value with n decimal digits, the representation has n decimal digits and one additional digit (bit) with a value of 0 or 1 so that a negative fraction x is represented by $2 - |x|$.

To reduce the delay of the subtraction in the recurrence, we use a radix-10 carry-save representation of the residual, as shown in Fig. 2, with $w_{s(i)} \in [0, 9]$ and $w_{c(i)} \in [0, 1]$. As a consequence of this representation, the subtraction is done using a radix-10 carry-save adder, which adds a carry-save operand and a conventional operand to produce a carry-save result, and the delay of the addition corresponds to the delay of one radix-10 digit.

The direct implementation of the radix-10 recurrence has two complications: 1) The selection function is complex and has a large delay and 2) the divisor multiples required for the recurrence are complicated to generate. To overcome these issues, we decompose the quotient digit as

$$q_j = 5q_{Hj} + q_{Lj},$$

with digit sets

$$q_{Hj} \in \{-1, 0, 1\} \quad \text{and} \quad q_{Lj} \in \{-a_L, \ldots, a_L\}$$

so that the recurrence is performed as follows:

$$w[j+1] = [10w[j] - 5q_{Hj+1}d] - q_{Lj+1}d = v[j] - q_{Lj+1}d. \quad (3)$$

We choose $a_L = 2$, which produces $-7 \leq q_j \leq 7$, and a redundancy factor $\rho = 7/9$. This choice of the digit set of q_L simplifies the generation of the multiples $q_{Lj}d$.

Since, for convergence, $|w[j]| \leq (7/9)d$, we initialize $w[0] = x/100$, which satisfies the range for the worst case, where $x = 1.0$ and $d = 0.1$.

Therefore, the recurrence is performed in the following steps (we later show how we can implement this faster):

- Compute
$$q_{Hj+1} = SEL_H(\widehat{10w[j]}, \widehat{d}),$$
where $\widehat{10w[j]}$ and $\widehat{d}$ are estimates of $10w[j]$ and d, respectively, as discussed further later.
- Select $5q_{Hj+1}d = q_{Hj+1} \cdot (5d)$ from the precomputed $-5d$, 0, and $+5d$ and compute
$$v[j] = 10w[j] - q_{Hj+1} \cdot (5d),$$
with $v[j]$ also in radix-10 carry-save format.
- Compute $q_{Lj+1} = SEL_L(\widehat{v[j]}, \widehat{d})$.
- Select $q_{Lj+1}d$ from the precomputed $-2d$, $-d$, 0, $+d$, and $+2d$ and compute
$$w[j+1] = v[j] - q_{Lj+1}d.$$

The above-mentioned recurrence requires the following:

- selection functions for q_{Hj+1} and q_{Lj+1} (described in Section 2.2.1),
- precomputation of $5d$ and $2d$, and
- radix-10 carry-save subtractions.

The multiples $2d$ and $5d$ are easy to generate. Specifically, the generation of $2d$ requires a carry propagation of only one digit and the generation of $5d$ can be performed by multiplying by 10 (a wired left shift) and dividing by 2, which requires a *carry* to the adjacent less significant position.

Although, in general, the range-complement representation of $w[j]$ would require an additional bit for the sign, in this case, since the most significant digit of the magnitude of $w[j]$ is 7, the most significant bit of the most significant digit acts as the sign bit.

2.2.1 Selection Function

We now determine the selection functions, namely,

$$q_H = SEL_H(\widehat{10w}, \widehat{d})$$
$$q_L = SEL_L(\widehat{v}, \widehat{d}).$$

The notation and process follow [16] for the general radix-r case.

The estimates $\widehat{10w}$ and $\widehat{v}$ are obtained by using a limited number of digits of the carry-save representation. Although, in principle, this number could be different for $10w$ than for v, we use the same number because, as described later, we use a radix-2 representation for this slice of digits. Let us call t the number of fractional digits of the estimates.

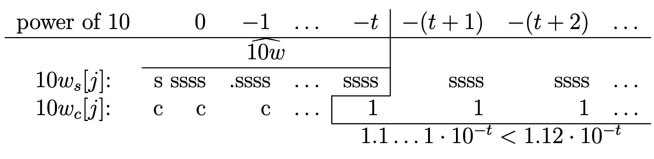

Fig. 3. Error in carry-save estimate.

Moreover, as we will see in the implementation, we do not include in the estimate the carry bit of digit t. Then, the error due to the estimate is bounded by 1.12×10^{-t}, as shown in Fig. 3. Then, the conditions for the selection constants m_k are

$$L_k(d_{i+1}) \leq m_k(i) \leq U_{k-1}(d_i) - 1.12 \times 10^{-t}, \qquad (4)$$

where $[L_k, U_k]$ are the corresponding selection intervals.

To obtain the selection intervals, we need the ranges of $10w[j]$ and $v[j]$. The range of $10w[j]$ is $[-10\rho d, +10\rho d]$, which, for the selected digit set, becomes $[-(70/9)d, +(70/9)d]$. The range of $v[j]$ is obtained from $|v[j] - q_L d| \leq \rho d$, which results in $|v[j]| \leq (\rho + a_L)d$. For the selected digit set $(a_L = 2)$, this is $|v[j]| \leq (25/9)d$.

Note that, to simplify the notation, in the development that follows, we use the same notation, $[L_k, U_k]$, for both q_H and q_L although they have different values and the index k corresponds to different sets.

Selection intervals for q_H. We now obtain the selection intervals $[L_k, U_k]$ for $q_H = k$, with $k \in \{-1, 0, 1\}$. From the recurrence,

$$v[j] = 10w[j] - 5q_{Hj+1}d$$

and, since $v[j] \leq (25/9)d$, by replacing $10w[j]$ by the upper limit of the selection interval, we obtain

$$(25/9)d = U_k - k(5d),$$

resulting in

$$U_k = (5k + 25/9)d.$$

Similarly, since $v[j] \geq -(25/9)d$, and L_k is the lower limit of the interval,

$$-(25/9)d = L_k - k(5d) \qquad \rightarrow \qquad L_k = (5k - 25/9)d.$$

Selection intervals for q_L. Similarly, we obtain the interval $[L_k, U_k]$ for $q_L = k$, with $k \in \{-2, -1, 0, 1, 2\}$. From the recurrence,

$$w[j+1] = v[j] - q_L d.$$

Introducing $w[j+1] \leq (7/9)d$ and the upper limit of the interval U_k, we obtain

$$(7/9)d = U_k - kd \qquad \rightarrow \qquad U_k = (k + 7/9)d$$

and, with $w[j+1] \geq -(7/9)d$ and the lower limit L_k,

$$-(7/9)d = L_k - kd \qquad \rightarrow \qquad L_k = (k - 7/9)d.$$

Bound for t. To obtain a bound on t, the number of fractional digits of $\widehat{10w}$, we use the minimum overlap condition:

$$U_{k-1}(d_i) - L_k(d_{i+1}) \geq 1.12 \times 10^{-t},$$

which occurs for the first subrange of d and the maximum k. If this subrange has a width of $10^{-\delta}$, then we obtain, for q_H,

$$(25/9) \cdot 0.1 - (20/9) \cdot (0.1 + 10^{-\delta}) \geq 1.12 \times 10^{-t},$$

so that, for integer t and for $\delta \geq 2$, we obtain $t \geq 2$.

Similarly, for q_L,

$$(16/9) \cdot 0.1 - (11/9) \cdot (0.1 + 10^{-\delta}) \geq 1.12 \times 10^{-t},$$

again resulting in $t \geq 2$.

Since the delay of the selection function directly depends on t, we choose the minimum value $t = 2$. Moreover, as explained next, we adjust the interval width $\Delta d = 10^{-\delta}$ to reduce the number of intervals of d.

Selection constants. We now determine suitable selection constants. The usual method is to divide the interval of d into subintervals of width Δd and then obtain selection constants for each interval. However, in this design, we proceed as follows:

- Define selection constants that are integers. That is, we define the selection function by the integer selection constants such that

  $$q_{Hj+1} = k \text{ if } m_{Hk} \leq 10^2 \times \widehat{10w}[j] < m_{Hk+1}$$

  and

  $$q_{Lj+1} = k \text{ if } m_{Lk} \leq 10^2 \times \widehat{v}[j] < m_{Lk+1}.$$

- Obtain the selection constants from (4) transformed so that the selection constants are integers and for $t = 2$. Moreover, from the possible values satisfying the condition, select the selection constant that is a multiple of 2^p for the largest possible p. This reduces the number of bits of the binary representation of the selection constant, which is convenient for the radix-2 implementation presented later.
- Make $m_{-k+1} = -m_k$ (for both H and L) so that only the constants for positive k have to be determined from $\hat{d}$. This way, for each $\hat{d}$, we need to compute $m_H(i) = \{m_{H1}\}$ and $m_L(i) = \{m_{L2}, m_{L1}\}$.
- Use the same subintervals for both functions and, beginning from $d = 0.1$, use the largest possible subinterval. This minimizes the number of subintervals and, therefore, the number of products in the implementation.

This procedure produces the constants, as described in Table 2.

3 Divider Architecture

We now describe the architecture of the divider. Of particular interest are 1) the retiming of the recurrence, 2) the organization of the selection function, and 3) the radix-2 implementation of the most significant slice. We now discuss these aspects.

3.1 Retiming of the Recurrence

To be able to reduce the delay of the most significant slice, we start with a retimed version of the iteration in which q_j and $w[j-1]$ are inputs and q_{j+1} and $w[j]$ are outputs. That

TABLE 2
Selection Constants for q_H and q_L

$[d_i,\ d_{i+1})$	q_H		q_L			
	m_{H1}	m_{H0}	m_{L2}	m_{L1}	m_{L0}	m_{L-1}
0.100, 0.106	26	-26	16	4	-4	-16
0.106, 0.120	28	-28				
0.12, 0.13	32	-32	20	8	- 8	-20
0.13, 0.14	34	-34				
0.14, 0.15	36	-36				
0.15, 0.17	40	-40	24			-24
0.17, 0.20	46	-46	28			-28
0.20, 0.22	52	-52	32			-32
0.22, 0.25	58	-58	36			-36
0.25, 0.30	68	-68	40			-40
0.30, 0.35	80	-80	48	16	-16	-48
0.35, 0.42	96	-96	56			-56
0.42, 0.50	114	-114	68			-68
0.50, 0.60	136	-136	84			-84
0.60, 0.70	164	-164	104	32	-32	-104
0.70, 0.84	188	-188	112			-112
0.84, 1.00	224	-224	124			-124

is, an iteration is described by the following expressions (see the 16-digit implementation in Fig. 4):

$$\begin{aligned} v[j] &= 10w[j-1] - q_{Hj} \cdot (5d), \\ w[j] &= v[j] - q_{Lj} d, \\ q_{Hj+1} &= SEL(\widehat{10w}[j], m_H(i)), \\ q_{Lj+1} &= SEL(\widehat{v}[j+1], m_L(i)). \end{aligned} \tag{5}$$

As explained in Section 3.2, $\widehat{v}[j+1]$ is computed speculatively inside the SEL block (see Fig. 5). It is necessary to initialize w and q. As indicated before, to ensure convergence, we initialize $w[0] = x/100$. Moreover, from the recurrence and $q_0 = 0$, we obtain $q_1 = SEL(\widehat{10w}[0], m(i))$. Consequently, in Fig. 4, this initialization is performed in the first cycle by selecting $x/100$ and initializing $q_H = q_L = w_c = 0$. The intialization effectively scales the dividend by 10^{-2} and therefore produces a quotient in the range $(10^{-3}, 10^{-1})$, that is, $q = q_0.q_1q_2\ldots = 0.0xxx\ldots$ or $0.00xxx\ldots$. Moreover, in the redundant (signed-digit) representation of the quotient, since the range of a digit is -7 to $+7$, the first fractional digit needs to be 0 or 1 (the value of 1 is needed when the second fractional digit of the final quotient is larger than 7). During the conversion to nonredundant representation (Section 4), this value 1 is converted to 0.

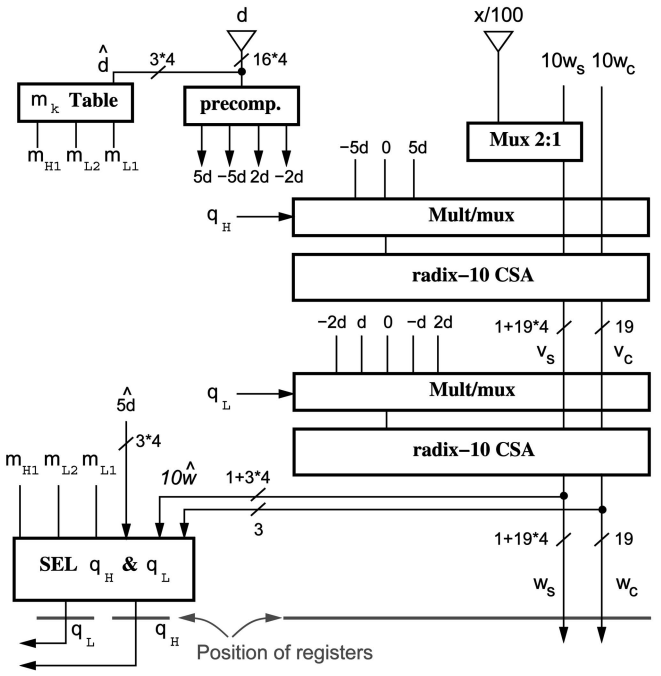

Fig. 4. Basic implementation of the radix-10 recurrence.

Table 3 shows some iterations of the division algorithm executed in the unit in Fig. 4.

3.2 Organization of the Selection Function

We consider now two issues in the organization of the selection function, namely, 1) the preloading of the selection constants and selection by comparison and 2) the overlapping of a conditional q_L with q_H. That is, as shown in Fig. 5,

1. As done in [17] and [15] for the binary case, we implement the selection function by preloading the selection constants, depending on the value of $\widehat{d}$,

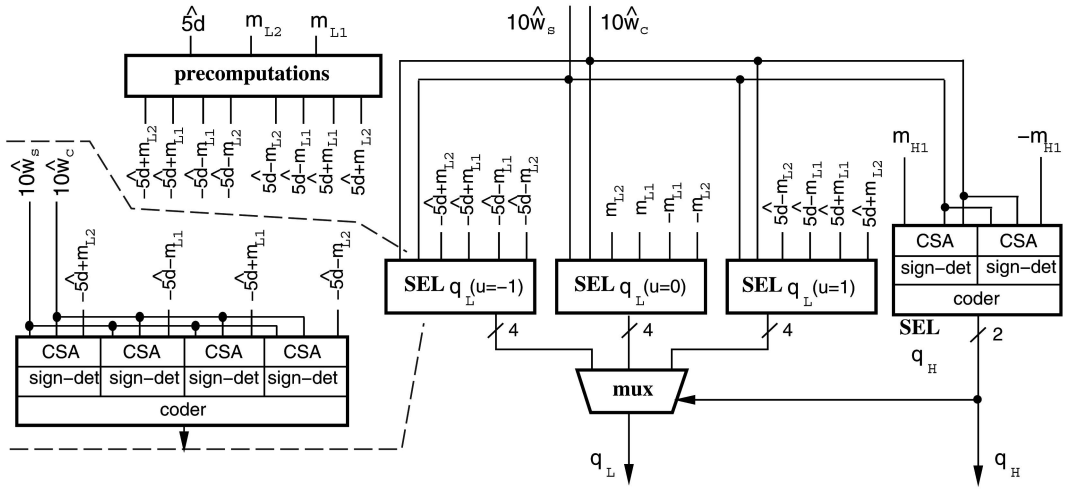

Fig. 5. Implementation of the selection function with overlapping and speculation.

TABLE 3
Example of Radix-10 Division (Unit in Fig. 4)

x = 0.85884596, d = 0.10637878, q = 8.073470667740310			
$\Rightarrow\ \widehat{d} = 0.106$, $m_{H1} = -m_{H0} = 26$, $m_{L2} = -m_{L-1} = 16$, $m_{L1} = -m_{L0} = 4$,			
	$10^2(\widehat{10w[j]})$	$w[j]$	$q[j]$
$w_s[0] = \frac{x}{100}$	0 008	0.008584596000000000	
$w_c[0]$	000	0.000000000000000000	$q[0] = 0.0000000000000000000$
	8	$\rightarrow q_H = 0,\ 10^2\widehat{v}[1]^* = 8 \rightarrow q_L = 1 \rightarrow q_1 = 1$	
$10w_s$		0.085845596000000000	
$10w_c$		0.000000000000000000	
$-q_H(5d)$		0.000000000000000000	
v_s		0.085845596000000000	
v_c		0.000000000000000000	
$-q_L d$		9.893621219999999999	
$w_s[1]$	1 878	9.878405705999999999	
$w_c[1]$	101	0.101100110000000001	$q[1] = 0.1000000000000000000$
	-21	$\rightarrow q_H = 0,\ 10^2\widehat{v}[2]^* = -21 \rightarrow q_L = -2 \rightarrow q_2 = -2$	
$10w_s$		8.784057059999999990	
$10w_c$		1.011001100000000010	
$-q_H(5d)$		0.000000000000000000	
v_s		9.795058159999999900	
v_c		0.000000000000000100	
$-q_L d$		0.212757560000000000	
$w_s[2]$	1 907	9.907705619999999000	
$w_c[2]$	100	0.100110100000001000	$q[2] = 0.0800000000000000000$
	7	$\rightarrow q_H = 0,\ 10^2\widehat{v}[3]^* = 7 \rightarrow q_L = 1 \rightarrow q_3 = 1$	
$10w_s$		9.077056199999990000	
$10w_c$		1.001101000000010000	
$-q_H(5d)$		0.000000000000000000	
v_s		0.078157199999900000	
v_c		0.000000000000100000	
$-q_L d$		9.893621219999999999	
$w_s[3]$	1 861	9.861778308888999999	
$w_c[3]$	110	0.110000111111000001	$q[3] = 0.0810000000000000000$
	-29	$\rightarrow q_H = -1,\ 10^2\widehat{v}[4]^* = 24 \rightarrow q_L = 2 \rightarrow q_4 = -3$	
$10w_s$		8.617783088889999990	
$10w_c$		1.100001111110000010	
$-q_H(5d)$		0.531893900000000000	
...	...	...	...
$\widehat{v}[j]^*$ is computed speculatively (see Figure 5)			

and we perform the selection of q_{Hj+1} and q_{Lj+1} by comparing the corresponding constants with $\widehat{10w[j]}$ and $\widehat{v}[j+1]$, respectively.

The preloading of the constants takes the corresponding delay out of the delay of the iteration. Moreover, the comparisons are done by a carry-save subtraction and a sign detection. This allows the balancing of the paths for delay reduction.

2. As is usually done for radix-16 dividers, we overlap the computation of a conditional q_L with the computation of q_H and then select the correct q_L. Therefore, the inputs to the SELs q_L are

$$\widehat{10w_s} + \widehat{10w_c} - u \cdot (\widehat{5d}) \qquad \text{with } u = \{-1, 0, 1\}.$$

Moreover, we can precompute

$$-u \cdot (\widehat{5d}) - m_{Lk}$$
$$\text{with } u = \{-1, 1\} \text{ and } k = \{-1, 0, 1, 2\} \ \text{(eight values)},$$

and, remembering that $m_{L-1} = -m_{L2}$ and $m_{L0} = -m_{L1}$, set the inputs for each of the SELs q_L as (see Fig. 5):

$$\text{SEL } q_L(u = -1) : \widehat{10w_s}, \widehat{10w_c}, (-\widehat{5d} + m_{L2}),$$
$$(-\widehat{5d} + m_{L1}), (-\widehat{5d} - m_{L1}), (-\widehat{5d} - m_{L2})$$
$$\text{SEL } q_L(u = 0) : \widehat{10w_s}, \widehat{10w_c}, (+m_{L2}), (+m_{L1}),$$
$$(-m_{L1}), (-m_{L2})$$
$$\text{SEL } q_L(u = 1) : \widehat{10w_s}, \widehat{10w_c}, (\widehat{5d} + m_{L2}), (\widehat{5d} + m_{L1}),$$
$$(\widehat{5d} - m_{L1}), (\widehat{5d} - m_{L2}).$$

3.3 Radix-2 Implementation of the Most Significant Digits

Since the estimate used in the selection function is obtained from the three most significant digits of $w[j]$, we separately implement the slice consisting of those digits. This has been done previously for binary division: first, in the 1960s, when the recurrence was using a carry-propagate adder [18] and then recently to be able to reduce the energy consumption of the not-critical part [19] and/or to reduce the delay by balancing the delay paths [15]. Moreover, we implement the most significant slice by using a radix-2 representation. This reduces the delay of the additions (both carry-save and carry-propagate) and reduces the width of the slice.

In the description of the most significant slice, it is convenient to refer to integer values. Specifically, since the most significant slice has three fractional digits, we define

integer variables as follows (and identify them with the subscript m):

$$(\ldots)_m = 1,000(trunc_3(\ldots)),$$

where $(\ldots)$ refers to a generic variable and $trunc_3(\ldots)$ corresponds to the carry-save representation of $(\ldots)$ truncated so as to include up to the third fractional digit.

As illustrated in Fig. 6 (upper part), the recurrence for this slice is then

$$\begin{aligned} y_m[j-1] &= 10w_m[j-1] + w_{(4)}[j-1], \\ v_m[j] &= y_m[j-1] - (q_{Hj} \cdot 5d)_m + v_{c(3)}[j], \\ w_m[j] &= v_m[j] - (q_{Lj}d)_m + w_{c(3)}[j], \end{aligned}$$

where $w_{(4)}[j-1]$ is the integer corresponding to the fractional digit 4 of $w[j-1]$, and $v_{c(3)}[j]$ and $w_{c(3)}[j]$ are the carries of $v[j]$ and $w[j]$ into the fractional digit 3.

Combining the two previous expressions, we get

$$\begin{aligned} w_m[j] = 10w_m[j-1] + w_{(4)}[j-1] - (q_{Hj} \cdot 5d)_m - (q_{Lj}d)_m \\ + v_{c(3)}[j] + w_{c(3)}[j]. \end{aligned} \tag{6}$$

A block diagram of the implementation of the most significant slice is shown in the lower part of Fig. 6, also including the selection function. Note that, in this slice, we do not explicitly compute $(v[j])_m$ since this value is computed speculatively in the selection function for q_L.

The critical path of this implementation is large, mainly because of the delay of the radix-10 addition in the most significant slice (most likely implemented by radix-10 carry-save adders: two in the recurrence, labeled "CSA tree" in Fig. 6, and one in the selection function). To reduce this delay, we implement this slice in radix 2. The disadvantages of this are the multiplication by 10 (which now requires a 4-to-2 adder) and the addition of $w_{(4)}[j-1]$ (which now cannot be done by concatenation). However, we will perform a retiming of the recurrence so that these operations are outside the critical path.

The radix-2 recurrence is obtained directly from (6). To distinguish from the radix-10 case, we use uppercase notation for the radix-2 variables (and eliminate the subscript m). That is,

$$\begin{aligned} W[j] = 10W[j-1] + w_{(4)}[j-1] - (q_{Hj}5D) - (q_{Lj}D) + v_{c(3)}[j] \\ + w_{c(3)}[j], \end{aligned} \tag{7}$$

where $q_{Hj}5D$ and $q_{Lj}D$ are the binary representation of $(q_{Hj} \cdot 5d)_m$ and $(q_{Lj}d)_m$, respectively.

Since $W[j]$ is the integer corresponding to the three most significant digits of the carry-save representation of $w[j]$ and $|w[j]| \leq 7/9$, the range of $W[j]$ is $[-779, +777]$, so it is represented in radix 2 (two's complement) by 11 bits.

The corresponding radix-2 implementation is shown in Fig. 7 and requires

- precomputation of the radix-2 representation of $\pm d_m$, $\pm(2d)_m$, and $\pm(5d)_m$ (that is, $\pm D$, $\pm 2D$, and $\pm 5D$, respectively),
- multiplication by 10 (this is implemented as $10z = 8z + 2z$ by using a 4-to-2 (radix-2) carry-save adder), and
- additions in the recurrence by using radix-2 carry-save adders.

There are two paths that potentially are the critical paths, namely, 1) the path in the radix-2 part corresponding to the sum of the delays in the multiplication by 10, the addition of $w_{(4)}$, the addition of the multiples of the divisor, and the selection function or 2) the path in the radix-10 part that produces $w_{c(3)}[j]$.

The path in the radix-2 part is reduced by retiming since the multiplication by 10 and the addition of $w_{(4)}$ are outside the $q_j - q_{j+1}$ loop. Consequently, they can be performed in parallel with the (previous) selection function. For the path in the radix-10 part, we perform the selection without including $w_{c(3)}[j]$ and add this value after the multiplication by 10 as $10w_{c(3)}[j]$. The selection function has already been designed to accommodate this postponement.

The radix-2 part has to be initialized with the corresponding portion of $x/100$, namely, $1,000(trunc_3(x/100)$. This is the integer corresponding to the first digit of x, that is, $x_{(1)}$.

The corresponding diagram is shown in Fig. 8 and the resulting implementation in Fig. 9.

The registers are placed in such a way as to balance the different paths. Consequently, the best placement depends on a specific implementation. Fig. 9 shows the placement obtained for the implementation presented in Section 5. The registers are called q_H, q_L, Y, w_s, and w_c. Furthermore, to keep the precomputation of $2d$, $5d$, and their radix-2 equivalents out of the critical path, we latch those values as well. Note that those precomputed values are not needed in the first iteration, as explained next.

3.4 Operation Execution

The computation of the 16-digit normalized and rounded significand of the quotient requires 19 iterations if $x \geq d$ or 20 iterations if $x < d$, as explained in Section 4.2.

To summarize, the architecture shown in Fig. 9 implements the algorithm in Table 4.

4 Conversion, Normalization, and Rounding

4.1 On-the-Fly Conversion

The on-the-fly-conversion algorithm [16] performs the conversion from the signed-digit representation of the quotient digit q_j to b_j, with the conventional representation in BCD (radix 10). The conversion is done as the digits are produced and does not require a carry-propagate adder.

The partial quotient is stored in two registers: Q, holding the converted value of the partial quotient $Q[j]$, and QM, holding $Q[j] - 10^{-j}$. The registers are updated in each iteration by shift-and-load operations and the final quotient is chosen between those two registers during the rounding.

In the radix-10 case, the implementation of the two registers Q and QM, with a precision of n digits, requires $2 \times (n \times 4)$ flip-flops. This results in a large area and a high-power dissipation due to the shifting during each iteration.

To reduce both power dissipation and area, we modify the on-the-fly algorithm, as described in [19]. Specifically:

power of 10 (s = sign)	0	.	−1	−2	−3	−4
	$w_m[j-1]$					
$10w[j-1]$	nnns	.	yyyy	yyyy	**zzzz**	xxxx
	c		y	y	**z**	x
	$y_m[j-1]$					
$-q_H(5d)$	nnns	.	yyyy	yyyy	yyyy	xxxx
$v[j]$	nnns	.	yyyy	yyyy	yyyy	xxxx
	c		y	y	z	x
$-q_L d$	nnns	.	yyyy	yyyy	yyyy	xxxx
$w[j]$	nnns	.	yyyy	yyyy	yyyy	**zzzz**
	c		y	y	z	**z**
bits marked n are not implemented			$w_m[j]$			

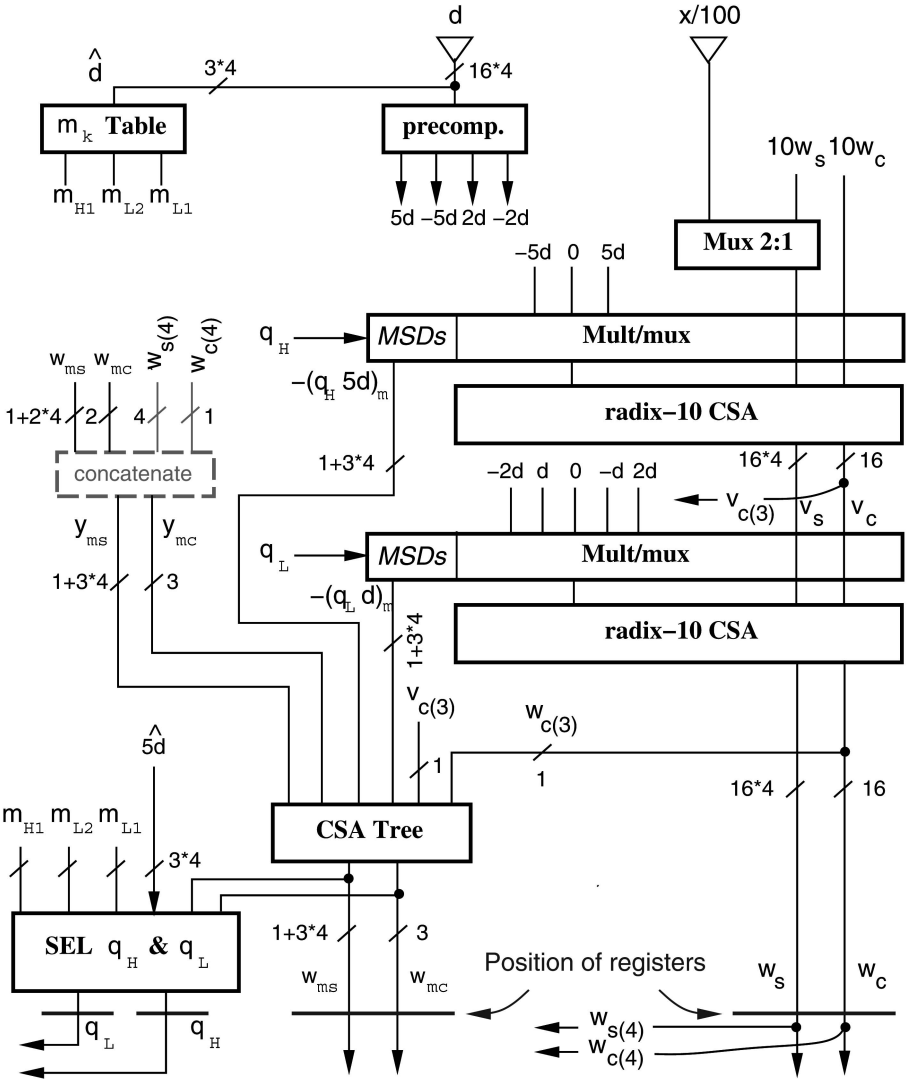

Fig. 6. A first implementation of the radix-10 recurrence.

1. We load each digit in its final position. This way, we avoid shifting digits along the registers. To determine the load position, we use an n-bit ring counter C, one bit for each digit to load.

$W[j-1]$	sssssssssss	
	ccccccccccc	
	$\times 10$	
	zzzz	$w_{s(4)}[j-1]$
	z	$w_{c(4)}[j-1]$
$Y[j-1]$	sssssssssss	
	ccccccccccc	
$-q_H 5D$	xxxxxxxxxxx	
$-q_L D$	xxxxxxxxxxx	
	z	$v_{c(3)}[j]$
	z	$w_{c(3)}[j]$
$W[j]$	sssssssssss	
	ccccccccccc	
	to selection function	

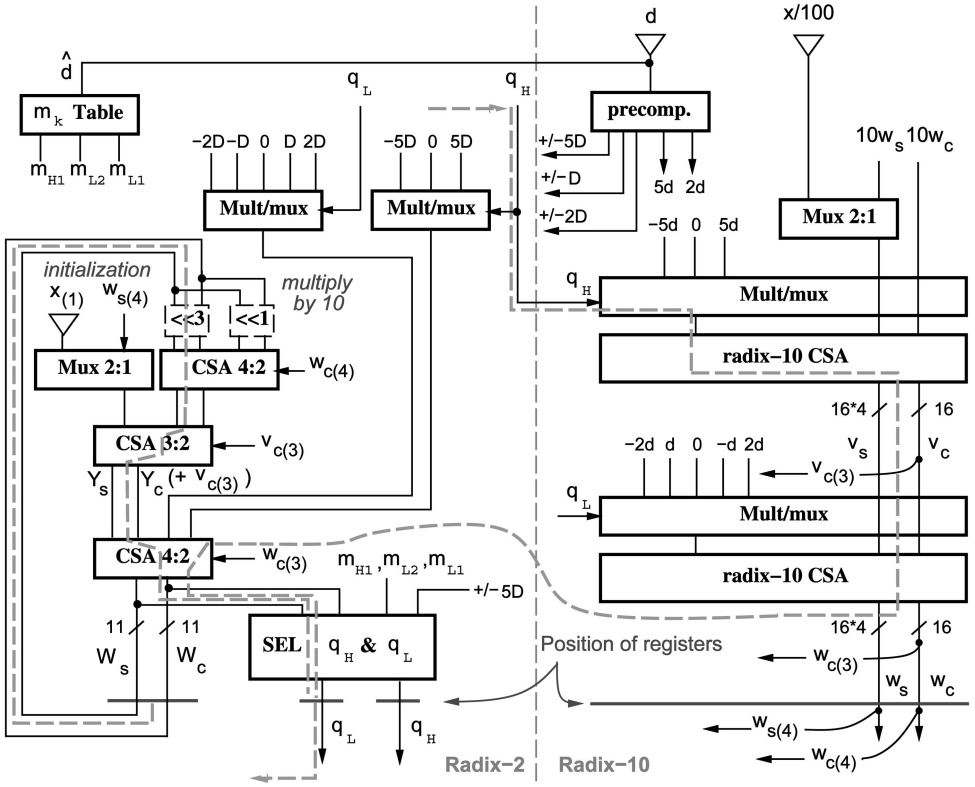

Fig. 7. Implementation of the recurrence with the radix-2 slice.

2. We reduce the partial-quotient registers from two to one by eliminating register QM and by including in register Q a digit decrementer controlled by the ring counter C (see [19] for more details).

With these modifications, we reduce the number of flip-flops in the conversion unit from $2 \times (n \times 4)$ to $n + (n \times 4)$.

4.2 On-the-Fly Normalization

The load-digit-in-place mechanism can be used to normalize[1] the quotient during the on-the-fly conversion. When dividing x/d $(0.1 \leq x, d < 1)$, we have two cases:

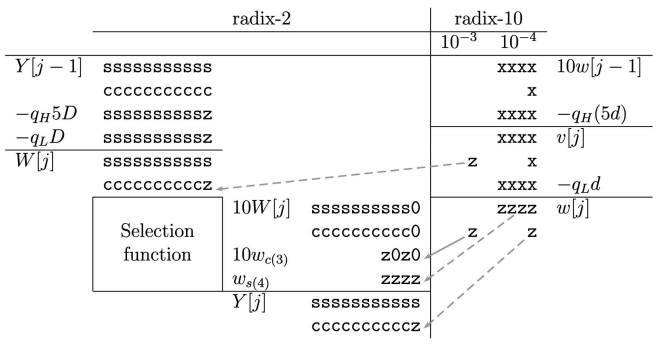

Fig. 8. Retimed operation of the radix-2 slice.

1. Although normalization is not required in the IEEE standard for decimal representation if the quotient is exact, even in this case, it might be better to provide a normalized quotient to the compliance modules. This is especially true since the normalization has no additional cost.

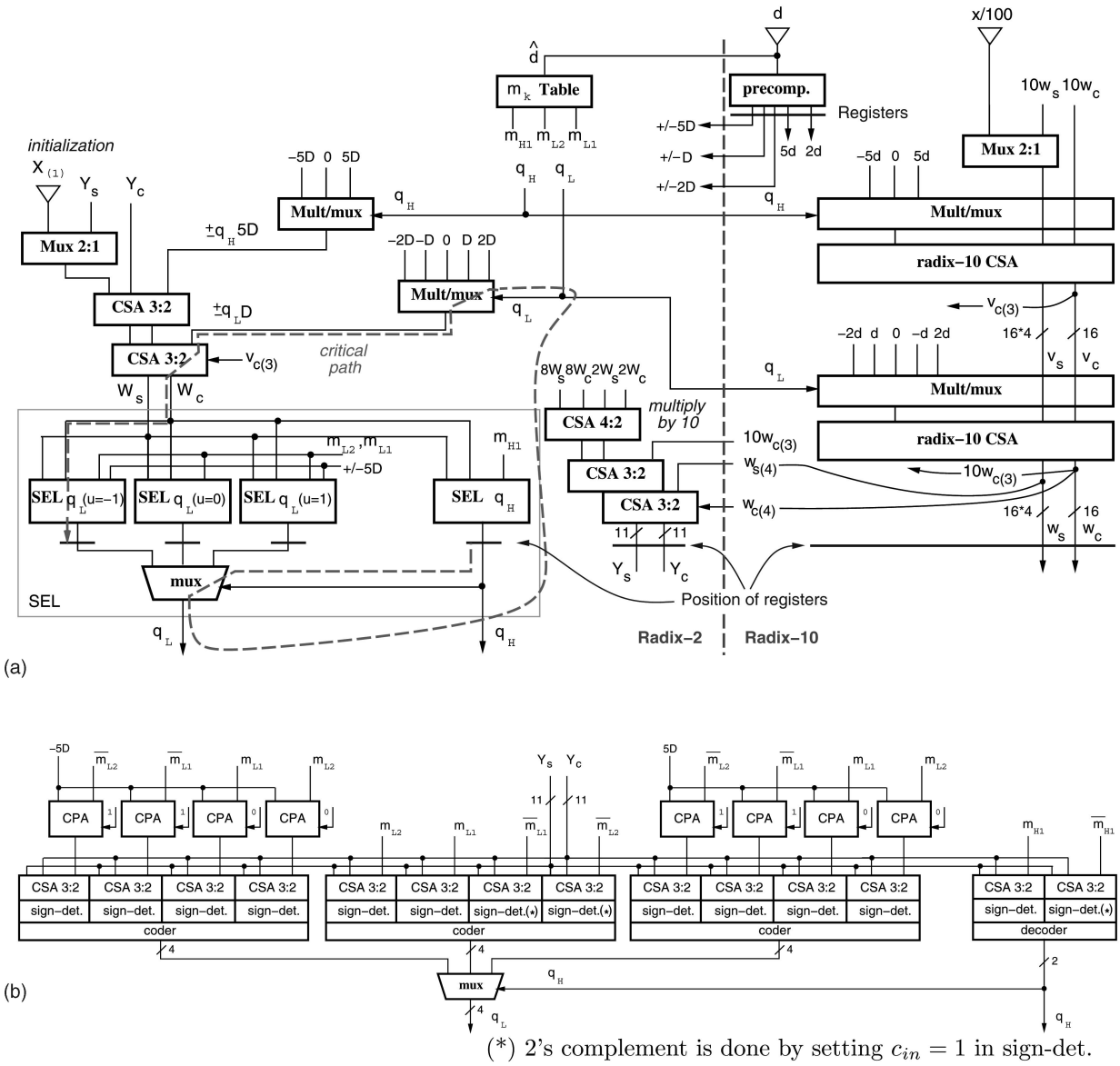

Fig. 9. Final implementation of the radix-10 recurrence and selection function. (a) Recurrence. (b) Selection function.

1. $x \geq d \quad \Rightarrow \quad 1.0 \leq q < 10.0$ (not normalized).
2. $x < d \quad \Rightarrow \quad 0.1 < q < 1.0$ (normalized).

Because of the initialization, the quotient obtained corresponds to $Q = q/100$. Consequently, the converted quotient is multiplied by 100 (shifted left, two digits), resulting in

$$\begin{array}{ll} Q & 0.0yxxx\ldots \\ q & y.xxxxx\ldots, \end{array}$$

where we indicate with y the digit of weight 10^{-2} in Q. Therefore, to obtain a normalized result, the two cases are listed as follows:

1. If $(y \neq 0) \quad Q_{norm} = 0.yxxx\ldots$, increment exponent.
2. If $(y = 0) \quad Q_{norm} = 0.xxxx\ldots$.

This could be achieved by placing y as the first fractional digit and shifting left one position when $y = 0$. We now describe how the normalization is done during the conversion, in this way avoiding the shifting.

Since the quotient obtained by the iterations is redundant with $\rho = 7/9$, q_1 of the redundant quotient can be 0 or 1. Consequently, the normalization is done as follows:

- If $q_1 = 1$, then the conversion changes it to $b_1 = 0$. The result is $q \geq 1.0\ (y \neq 0) \Rightarrow$ Case A.
- If $q_1 = 0$ and $q_2 > 1$, then $q \geq 1.0\ (y \neq 0) \Rightarrow$ Case A.
- If $q_1 = 0$ and $q_2 = 0$, then $q < 1.0\ (y = 0) \Rightarrow$ Case B.
- If $q_1 = 0$ and $q_2 = 1$, then this depends on the first nonzero digit $q_j\ (j \geq 3)$:
  - If positive, then $(y \neq 0) \Rightarrow$ Case A.
  - If negative, then $(y = 0) \Rightarrow$ Case B. This situation is shown in the example in Table 5.

In case A, the converted quotient corresponds to $0.b_2b_3b_4\ldots b_N$ and the exponent is incremented by 1, whereas, in case B, the converted quotient corresponds to $0.b_3b_4b_5\ldots b_N$.

TABLE 4
Algorithm Implemented by the Unit in Fig. 9

Initialization : $q_0 = 0$, $(w_s,\ w_c) = 0$, $Y = 0$.

Cycle 1 : precomputation of $\pm 5d$, $\pm 2d$, $\pm 5D$, $\pm 2D$, $\pm D$ (not needed in first cycle), and m_ks (needed in SEL at the end of the cycle).

$$\begin{aligned} w[0] &= x/100 + 0 \cdot d = x/100 \quad (x/100 \text{ selected by mux}) \\ W[0] &= x_{(1)} - 0 \cdot 5D - 0 \cdot D = x_{(1)} \quad (x_{(1)} \text{ selected by mux}) \\ q_1 &= SEL(W[0], m_H(i), m_L(i)) = 0 \text{ or } 1 \rightarrow q_H = 0,\ q_L = 0 \text{ or } 1 \\ Y[0] &= 10W[0] + w_4[0] = 10 \cdot x_{(1)} + x_{(2)} \end{aligned}$$

Cycle 2 :

$$\begin{aligned} w[1] &= 10w[0] - q_1 d \\ W[1] &= Y[0] - q_{H1} 5D - q_{L1} D + v_{c(3)}[1] \\ q_2 &= SEL(W[1], m_H(i), m_L(i)) \\ Y[1] &= 10W[1] + 10w_{c(3)}[1] + w_{s(4)}[1] + w_{c(4)}[1] \end{aligned}$$

...

Cycle N :

$$\begin{aligned} w[N-1] &= 10w[N-2] - q_{N-1} d \\ W[N-1] &= Y[N-2] - q_{H(N-1)} 5D - q_{L(N-1)} D + v_{c(3)}[N-1] \\ q_N &= SEL(W[N-1], m_H(i), m_L(i)) \\ Y[N-1] &= 10W[N-1] + 10w_{c(3)}[N-1] + w_{s(4)}[N-1] + w_{c(4)}[N-1] \end{aligned}$$

...

Termination (Cycle $N+1$): $N+1 = 19$ if $x \geq d$, or $N+1 = 20$ if $x < d$ (see Section 4.2).
Remainder computation: $w[N] = w_s[N] + w_c[N]$
Rounding (see Section 4.3).

TABLE 5
Example of Conversion and Normalization when $q_2 = 1$ and $q_3 = 0$

Conversion			Conversion and Normalization			
q_1	=	0 , Q = 0				
Q[1]	=	0.000000000000000000	Q[1]	=	0.00000000000000000	
q_2	=	1 , Q = 0 1				
C	=	100000000000000000	C	=	10000000000000000	
Q[2]	=	0.100000000000000000	Q[2]	=	0.10000000000000000	
q_3	=	0 , Q = 0 1 0				
C	=	010000000000000000	C	=	01000000000000000	
Q[3]	=	0.100000000000000000	Q[3]	=	0.10000000000000000	
q_4	=	0 , Q = 0 1 0 0				
C	=	011000000000000000	C	=	01100000000000000	
Q[4]	=	0.100000000000000000	Q[4]	=	0.10000000000000000	
q_5	=	0 , Q = 0 1 0 0 0				
C	=	011100000000000000	C	=	01110000000000000	
Q[5]	=	0.100000000000000000	Q[5]	=	0.10000000000000000	
q_6	=	-3 , Q = 0 1 0 0 0 $\bar{1}$				
C	=	000010000000000000	C	=	00001000000000000	← freeze
Q[6]	=	0.099970000000000000	Q[6]	=	0.09970000000000000	
q_7	=	1 , Q = 0 1 0 0 0 $\bar{1}$ 1				
C	=	000001000000000000	C	=	00001000000000000	
Q[7]	=	0.099971000000000000	Q[7]	=	0.09971000000000000	

The control of cases A and B is performed during the conversion. The normalization is achieved by freezing the position of the hot 1 in the ring counter (see the example in Table 5).

Calling L the number of digits of the final quotient, we produce $L+1$ digits (one additional quotient digit for the rounding). Consequently, the number of cycles $(N+1)$ is $L+3$ for case A and $L+4$ for case B. Specifically, for $L = 16$, we get 19 and 20 cycles, respectively.

4.3 Rounding

In the implementation of the unit, we only consider the *round-half-even* rounding mode, but the other modes can be implemented in a similar manner.

For the normalized quotient, the rounding is always performed in position $L+1$ in register Q (see Fig. 10). The rounding is performed by adding half ulp $(5 \times 10^{-(L+1)})$ to $b_{(L+1)}$ and truncating the representation of Q in position L. Moreover, when the sign of the remainder is negative $(sign = 1)$, we have to subtract 1 from $b_{(L+1)}$. Finally, if the remainder is zero $(zero = 1)$ and $b_{(L+1)} = 5$, then we

weight 10^{-k}	0	.	1	2	...	L-1	L	L+1
digits of Q:	0	.	$b_{(1)}$	$b_{(2)}$	...	$b_{(L-1)}$	$b_{(L)}$	$b_{(L+1)}$

$b_{(k)}$ indicates the BCD digit of weight 10^{-k}

Fig. 10. The rounding position in register Q.

increment $b_{(L)}$ by 1 if its least significant bit is 1 (round to even). In summary, the operation to perform on $b_{(L+1)}$ is

$$b_{(L+1)} + 5 - sign - (zero \text{ AND } \overline{\text{LSB}(b_{(L)})}). \quad (8)$$

Clearly, a borrow ($borrow_{RND}$) or a carry ($carry_{RND}$) might be generated into position L. Although the propagation of the borrow is eliminated by the on-the-fly conversion, the propagation of the carry (which occurs, for instance, when, before conversion, we have $\ldots 3\,0\,0\,0\,-1\;q_N \rightarrow \ldots 2\,9\,9\,9\,9\;b_{(L+1)}$) has to be addressed separately.

This carry propagation can be avoided if the conversion of digit L is performed including the carry from the rounding (in this case, in the conflicting situation above, the converted digit becomes 0). That is, the conversion of digit L has to be postponed one cycle. This is achieved by freezing the update of the marked digits when iteration L is reached. The update of digits in register Q is resumed only after the outcome of (8) is known. An example is shown in Table 6.

5 Implementation and Comparisons

In this section, we present the results of the evaluation of the proposed design and a comparison with a double-precision radix-16 digit-recurrence division unit which has roughly the same dynamic range for the normalized significand ($2^{53} < 10^{16} < 2^{54}$). Furthermore, we compare our results with those obtained in [4] for a divider based on the Newton-Raphson approximation.

We performed a synthesis of the decimal divider, as shown in Fig. 9, by using the STM 90-nm CMOS standard cells library [20] and Synopsys Design Compiler. From the synthesis, we estimated the critical path (including estimations at the netlist level of wire load) and the area. The critical path is highlighted in Fig. 9 (dotted line) and reported in detail in Table 7.

The results are compared with those of [15] and reported in Table 8. The data in Table 8 show that the decimal divider has a latency close to that of a standard radix-16 and its area is close to that of the faster radix-16 unit in [15].

With respect to the implementation in [4], it is quite difficult to compare between the two different algorithms. However, the implementation in [4] requires 150 cycles to compute a 16-digit quotient (as in our case) and has a critical path of 0.7 ns in a 0.11 μm standard cells library. Assuming a constant field technology scaling[2] [21], the critical path of 0.7 ns scales approximately to $0.7 \cdot (90/110) = 0.57$ ns. The resulting latency is $150 \times 0.57 = 85.5$ ns, which is more than four times the latency of the divider presented in this work. Furthermore, the expression given in [4] for the number of cycles for a generic implementation of the Newton-Raphson algorithm is n. cycles $= 10 + P(3 + 2m)$, where P is the number of cycles needed to perform a decimal multiplication and m is the number of iterations needed for the approximation of $1/d$ (Newton-Raphson iterations). Because, to our knowledge, decimal multiplication requires several cycles ($n + 4$ cycles, where n is number of digits, in [3], for example), the digit-recurrence approach proposed in this work seems quite advantageous in terms of latency.

2. In [4], the supply voltage is $V_{DD} = 1.2$ V, whereas, in the library that we used, it is $V_{DD} = 1.0$ V. Therefore, the ratio of the feature lengths $110/90 = 1.2$ equals the ratio of the supply voltages.

TABLE 6
Example of Rounding without the Carry Propagation when $q_L = -1$

		L-3	L-2	L-1	L	L+1
sequence of q_j:	...	3	0	0	-1	q_N

iteration	Register Q					Register C					
N-4	...	3	-	-	-	...	1	0	0	0	0
N-3	...	3	0	-	-	...	1	1	0	0	0
N-2	...	3	0	0	-	...	1	1	1	0	0
N-1	...	3	0	0	9	...	1	1	1	1	0
N	...	3	0	0	9	...	1	1	1	1	1

Rounding*	Register Q				comments
$(B+R) \geq 10$	...	3	0	0	0	$carry_{RND} = 1,\ borrow(L) = 1$
$0 \leq (B+R) < 10$	...	2	9	9	9	$carry_{RND} = 0,\ borrow_{RND} = 0,\ borrow(L) = 1$
$(B+R) < 0$	...	2	9	9	8	$borrow_{RND} = 1,\ borrow(L) = 1$

* $R = 5 - sign - (zero \text{ AND } \overline{\text{LSB}(b_{(L)})})$ as indicated in (8).

TABLE 7
Details of the Critical Path for the Implementation in Fig. 9

blocks in critical path							total delay
reg. q_H	buffers	mux q_L	MULT q_L	CSA 3:2	SEL q_L	set-up	
0.09	0.04	0.10	0.08	0.08	0.50	0.08	1.00
delays in $[ns]$							

TABLE 8
Summary of Results for the Synthesized Unit and Those of [15]

Unit	cycle time $[ns]$	n. cycles	latency $[ns]$	speed-up	area $[\mu m^2]$	ratio
Radix-16 (standard)	1.00	16	16.0	1.00	38000	1.00
Radix-16 [15]	0.72	16	11.5	1.40	59600	1.56
Radix-10 (this work)	1.00	20	20.0	0.80	59700	1.57

6 Conclusions

In this work, we presented a radix-10 division unit that is based on the digit-recurrence algorithm. We developed the algorithm and the divider architecture and implemented it in standard cells. The unit implemented (16-digit BCD) has a dynamic range of the significand that is comparable to that of the IEEE double-precision standard.

Since there were no comparable digit-recurrence radix-10 units, we compared with a radix-16 divider because the radices are similar. We concluded that the proposed radix-10 divider has a latency close to that of a basic retimed radix-16 divider, which a floating-point-unit designer might use when considering a fast implementation for medium-range radices. Moreover, we compared the proposed divider with a recent implementation of a decimal

divider based on the Newton-Raphson approximation [4]. Although a comparison between such different approaches depends on too many parameters, considering the timing reported for the two implementations, we are quite confident that the design presented here shows a latency (execution time for a 16-digit division) about four times shorter than that of the unit based on the Newton-Raphson approximation.

REFERENCES

[1] M. Cowlishaw, E. Schwarz, R. Smith, and C. Webb, "A Decimal Floating-Point Specification," *Proc. 15th IEEE Symp. Computer Arithmetic (ARITH 15)*, pp. 147-154, June 2001.

[2] M.F. Cowlishaw, "Decimal Floating-Point: Algorithm for Computers," *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH 16)*, pp. 104-111, June 2003.

[3] M. Erle, E. Schwarz, and M. Schulte, "Decimal Multiplication with Efficient Partial Product Generation," *Proc. 17th IEEE Symp. Computer Arithmetic (ARITH 17)*, pp. 21-28, June 2005.

[4] L.-K. Wang and M. Schulte, "Decimal Floating-Point Division Using Newton-Raphson Iteration," *Proc. 15th IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP '04)*, pp. 84-95, Sept. 2004.

[5] F. Busaba, C. Krygowski, W. Li, E. Schwarz, and S. Carlough, "The IBM z900 Decimal Arithmetic Unit," *Proc. 35th Asilomar Conf. Signals, Systems, and Computers (Asilomar '01)*, vol. 2, pp. 1335-1339, Nov. 2001.

[6] R. Kenney and M. Schulte, "High-Speed Multioperand Decimal Adders," *IEEE Trans. Computers*, vol. 54, no. 8, pp. 953-963, Aug. 2005.

[7] L.-K. Wang and M. Schulte, "Decimal Floating-Point Square Root Using Newton-Raphson Iteration," *Proc. 16th IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP '05)*, pp. 309-315, July 2005.

[8] S.A. Tague and V.S. Negi, "Data Processor Having Carry Apparatus Supporting a Decimal Divide Operation," US patent 4,384,341, Patent and Trademark Office, 1983.

[9] S. Tokumitsu, "Divider Circuit for Dividing n-Bit Binary Data Using Decimal Shifting and Summation Techniques," US patent 4,599,702, Patent and Trademark Office, 1986.

[10] H. Yabe et al., "Binary Coded Decimal Number Division Apparatus," US patent 4,635,220, Patent and Trademark Office, 1987.

[11] A. Yamaoka et al., "Coded Decimal Non-Restoring Divider," US patent 4,692,891, Patent and Trademark Office, 1987.

[12] D.E. Ferguson, "Non-Heuristic Decimal Divide Method and Apparatus," US patent 5,587,940, Patent and Trademark Office, 1996.

[13] C.F. Webb et al, "Specialized Millicode Instructions for Packed Decimal Division," US patent 6,067,617, Patent and Trademark Office, 2000.

[14] *IEEE Standard for Floating-Point Arithmetic*, http://754r.ucbtest.org/drafts/754r.pdf, Oct. 2005.

[15] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli, "Digit-Recurrence Dividers with Reduced Logical Depth," *IEEE Trans. Computers*, vol. 54, no. 7, pp. 837-851, July 2005.

[16] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[17] N. Burgess and C. Hinds, "Design Issues in Radix-4 SRT Square Root and Divide Unit," *Proc. 35th Asilomar Conf. Signals, Systems and Computers (Asilomar '01)*, pp. 1646-1650, 2001.

[18] D.E. Atkins, "Design of Arithmetic Units of ILLIAC III: Use of Redundancy and High-Radix Methods," *IEEE Trans. Computers*, vol. 19, no. 8, pp. 720-733, Aug. 1970.

[19] A. Nannarelli and T. Lang, "Low-Power Divider," *IEEE Trans. Computers*, vol. 48, no. 1, pp. 2-14, Jan. 1999.

[20] STMicroelectronics, 90nm CMOS090 Design Platform, http://www.st.com/stonline/prodpres/dedicate/soc/asic/90plat.htm, 2007.

[21] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.

Tomás Lang received the BS degree in electrical engineering from the Universidad de Chile in 1965, the MS degree from the University of California, Berkeley, in 1966, and the PhD degree from Stanford University in 1974. He is a professor in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. Previously, he was a professor in the Department of Computer Architecture at the Polytechnic University of Catalonia, Spain, and a faculty member in the Department of Computer Science at the University of California, Los Angeles. His primary research and teaching interests are in digital design and computer architecture, with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is the coauthor of two textbooks on digital systems, two research monographs, and one IEEE tutorial and the author/coauthor of research contributions to scholarly publications and technical conferences. He is a member of the IEEE Computer Society.

Alberto Nannarelli received the BS degree in electrical engineering from the University of Rome La Sapienza, Italy, in 1988 and the MS and PhD degrees in electrical and computer engineering from the University of California, Irvine, in 1995 and 1999, respectively. He is an associate professor at the Technical University of Denmark. He worked for SGS-Thomson Microelectronics and for Ericsson Telecom as a design engineer and for Rockwell Semiconductor Systems as a summer intern. From 1999 to 2003, he was with the Department of Electrical Engineering, University of Rome Tor Vergata, Italy, as a postdoctoral researcher. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE Computer Society.