A Radix-10 SRT Divider Based on Alternative BCD Codings *

Alvaro Vázquez, Elisardo Antelo
University of Santiago de Compostela
Dept. of Electronic and Computer Science
15782 Santiago de Compostela, Spain
alvaro@dec.usc.es, elisardo@dec.usc.es

Paolo Montuschi
Politecnico di Torino
Dept. of Computer Engineering
10129 Torino, Italy
montuschi@polito.it

Abstract

In this paper we present the algorithm and architecture of a radix-10 floating-point divider based on an SRT non-restoring digit-by-digit algorithm. The algorithm uses conventional techniques developed to speed-up radix-2^k division such as signed-digit (SD) redundant quotient and digit selection by constant comparison using a carry–save estimate of the partial remainder. To optimize area and latency for decimal, we include novel features such as the use of alternative BCD codings to represent decimal operands, estimates by truncation at any binary position inside a decimal digit, a single customized fast carry propagate decimal adder for partial remainder computation, initial odd multiple generation and final normalization with rounding, and register placement to exploit advanced high fanin mux-latch circuits. The rough area-delay estimations performed show that the proposed divider has a similar latency but less hardware complexity (1.3 area ratio) than a recently published high performance digit-by-digit implementation.

1. Introduction

The design of high–performance decimal floating-point units (DFUs) is becoming a topic of interest [6]. Moreover, the recent approval of the IEEE-754R standard [1] includes specifications for decimal arithmetic. In this paper we design a new algorithm and architecture for radix-10 division. We focus our attention to the class of SRT non-restoring digit-by-digit methods. Digit-by-digit methods have the characteristic of producing one digit per iteration. Good examples of radix-2^k based division algorithms can be found in [2, 7]. On the other hand, radix-10 division algorithms, which are found recently in the literature, can be classified as follows: Newton Raphson based [15] and restoring [4] or non-restoring digit-by-digit methods [8, 10]. The paper is organized as follows: in Section 2 we present the proposed algorithm and determine the selection constants and the best BCD digit coding for a fast and a simple implementation of the selection function. In Section 3 we describe the architecture and the operation sequence of the divider. Also, the decimal adder used for residual assimilation, normalization and rounding is fully detailed. In Section 4 we present the area–delay evaluation results for the proposed divider using a rough model based on logical effort [12]. We compare our design with two recent radix-10 SRT designs [8, 10] and with a software implementation [5]. In Section 5 we summarize the main conclusions of this work.

* A. Vázquez and E. Antelo supported in part by the Ministry of Science and Technology of Spain under contract TIN2004-07797-C02.

2. Radix-10 Digit-Recurrence Division

The starting point of this work is the application of well-known radix-2^k methods to improve radix-10 division. Among these methods are the use of a symmetrical redundant digit set for the quotient digits ($-a \leq q_i \leq a$, with $a \in \{5, \ldots, 10\}$), quotient digit selection using estimates of the residual and the divisor and preloaded constants, and the use of a carry-save format to represent the residual. In principle, it is not certain that the well-known methodologies currently found in the literature are the best solution, and even if they were, their application is not simple. To adapt these radix-2^k techniques to radix-10 division we follow a different approach than [8] (the best up-to-date implementation). Instead of splitting the digit selection and residual updating into two overlapped stages, we opt for a digit set that minimizes the complexity of generating divisor multiples. Specifically, this work presents the following contributions:

• Implementation with a suitable digit set that minimizes the complexity of generating divisor multiples.

• A study of alternative BCD codings to represent decimal digit operands. The dividend x, divisor d and quotient Q require a decimal range-complement non redundant representation, while the residual $w[i]$ requires a redundant representation (carry-save or signed digit). The preferred coding leads to the simpler implementation of the selection function and divisor multiples generation in terms of area-latency trade-offs.

• Obtention of the selection constants as estimates of different truncated multiples of the divisor, avoiding tables.

• Design of a decimal adder to compute the required odd divisor multiples, the assimilation of the previous residual (in parallel with the selection of the next quotient digit), and normalization and rounding, thus sharing the same hardware for different parts of the floating-point division.

2.1. Proposed Algorithm

We assume that the dividend x, the divisor d and the quotient Q are in the range[1] $[1, 10)$. The radix-10 division algorithm implements the following recurrence

$$w[i+1] = 10w[i] - q_{i+1}\, d \tag{1}$$

where $w[i]$ is the partial remainder at iteration i, d is the divisor and q_{i+1} is the digit of the quotient with weight $10^{-(i+1)}$. In order to converge, it is necessary to select q_{i+1} so that the resulting residual is bounded by

$$-\rho d \leq w[i+1] \leq \rho d \tag{2}$$

where $\rho = a/9$ is the redundancy factor.

The main drawback of recurrence (1) is the generation of odd multiples of d. One simple approach consists of implementing the recurrence as two simpler overlapped recurrences of lower radix [8]. In this work we explore an alternative, a direct implementation of (1) with the minimally redundant set $\{-5, \ldots, 0, \ldots, 5\}$ ($\rho = 5/9$). This choice minimizes the complexity of the generation of decimal divisor multiples while having a single recurrence. We only need to compute the odd multiple $3d$ using a decimal carry-propagate adder. The other multiples can be generated with simple recodings.

Since we need a carry-propagate adder for multiple generation, we designed the algorithm and the architecture to reuse this adder. Specifically, the addition required by the recurrence (1) is implemented with this adder, so that we keep the residual in non-redundant form. To make the determination of q_{i+1} independent of this carry-propagate addition (which would result in a large cycle time) we perform the digit selection using an estimation of $w[i]$ obtained from the leading digits of $10w[i-1]$ and $-q_i d$ (this is allowed by the redundancy of the digit set for q_{i+1}). From the point of view of the estimation, this is similar to the standard practice of keeping the residual in carry-save.

[1]The IEEE-754r standard does not require normalized operands, and therefore a pre-processing stage is necessary to have the values of the operands normalized.

For convergence it is necessary to assure $-(5/9)d \leq w[0] \leq (5/9)d$. This is achieved by the following initialization

$$w[0] = \begin{cases} x/20 & \text{if } (x-d) \geq 0 \\ x/2 & \text{else} \end{cases} \tag{3}$$

and $q_0 = 0$. Under this initialization $0.05 \leq q < 0.5$, so it is necessary to multiply the resultant quotient by 20 to produce the normalized quotient in the appropriate interval $[1, 10)$. For a n-digit precision rounded quotient the algorithm requires $n+2$ quotient digits q_i ($i > 0$) (including a guard and round digit). This initialization requires the computation of the sign of $x - d$, which is performed by the decimal carry-propagate adder. Moreover, the same adder is also used for the final conversion from redundant to non-redundant representation and rounding of the result quotient.

The coding of decimal digits is a key issue to provide fast redundant decimal arithmetic. A decimal digit Z_i is represented in a 4-bit weighted number system (BCD-$r_3r_2r_1r_0$) as

$$Z_i = \sum_{i=0}^{3} z_{i,j} \cdot r_j \tag{4}$$

where $z_{i,j}$ is the j^{th} bit of the BCD i^{th} digit and r_j is the weight of the j^{th} bit. Recent work [13] suggests the use of alternative BCD codes (BCD-4221 and BCD-5211) instead of the more conventional BCD code (BCD-8421) to improve decimal carry-save addition by using binary carry-save arithmetic. Moreover, the negative value of a decimal number coded in BCD-4221 or BCD-5211 can be obtained by bit-inversion and addition of one ulp (unit in the last place). Thus, negative divisor multiples are obtained from the corresponding positive ones (d, $2d$, $3d$, $4d$,$5d$) as a two's complement over the bit-vector representation. The generation of divisor multiples is detailed in Section 3.

The choice of the representation for the operands is directly related to the value of the truncation error in the estimations used to obtain the quotient digits. In this way, the BCD-5211 coding presents certain advantage with respect to BCD-4221 or BCD-8421 (see Section 2.2).

Finally, to convert the $n+2$ signed-digit quotient into the n-digit rounded non-redundant form each q_i value is recoded as $q_i = -10 \cdot s_i + q_i^*$ where

$$(s_i, q_i^*) = \begin{cases} (0, q_i) & q_i \geq 0 \\ (1, 10 - q_i) & else \end{cases} \tag{5}$$

$(s_i, q_i^*) \in (\{0,1\}, \{0, \ldots, 9\})$. This digit recoding is performed after each digit selection, so after $n+2$ iterations of (1) we have $Q^* = \sum_{i=1}^{n+2} q_i^* \cdot 10^i$ and $S = \sum_{i=2}^{n+3} s_i \cdot 10^i$, where $s_{n+3} = sign(w[n+2])$ is introduced to correct the quotient when the last residual is negative. The quotient is obtained as $Q = round(Q^* - 10 \cdot S)_n$, which requires a decimal subtraction and rounding to n digits according to the rounding specifications. This operation is performed by the decimal carry-propagate adder, adapted to include the rounding.

2.2. Selection Function

As mentioned before, the quotient digit q_{i+1} is obtained from an estimation $\widehat{10w[i]}$ of $10w[i]$ by truncating $10w[i-1]$ and $-q_i d$ and input them to the digit selection. This estimation is compared to selection constants (m_k, with $k = -4, ..., 5$) which are dependent on the leading digits of d. Specifically,

$$q_{i+1} = k \quad \texttt{if} \;\; m_k \leq \widehat{10w[i]} < m_{k+1} \tag{6}$$

As in [8] we implement the selection function by comparing the estimation of the residual with each of the selection constants. Since the estimation is composed of two words, the comparisons are performed by obtaining the sign of the difference between the estimation and the selection constant. This is performed by a three to two reduction using a decimal carry-save adder and a decimal sign detector.

The method used to obtain the selection constants is well-known [8]. However, in this work, we introduce two innovations:

• New BCD codings to reduce the estimation error.

• Estimations by truncating at the bit level instead of digit level (number of bits multiple of four), to reduce the number of bits of the estimations.

The process to obtain the number of bits of the estimations and the selection constants is not described here due to the lack of space and because it is a standard process. We obtained that by using BCD-5211 coding, one less fractional bit is needed for the estimation of the residual compared to BCD-8421 or BCD-4221. Moreover, the decimal carry-save adder is effectively implemented with BCD-5211 (very similar to the simple implementation with BCD-4221 [13]). Therefore we used BCD-5211 for representing the operands.

For BCD-5211 representation, the selection constants are obtained from the leading 12 bits of d (one integer and two fractional decimal digits). The estimation of the residual requires 9 integer (including sign) and 6 fractional bits. The selection constants require the following number of bits (integer+fractional): m_1 and $m_0 \rightarrow$ 3+6, m_2 and $m_{-1} \rightarrow$ 5+6, m_3 and $m_{-2} \rightarrow$ 6+6, m_4, m_5, m_{-3} and $m_{-4} \rightarrow$ 7+6.

A straightforward implementation to obtain the constants consists in using a look-up table with the 12 bits of d as input. Our synthesis results indicated that this approach was very costly in terms of area and time (we want to determine the constants in just one cycle). An efficient implementation is obtained by computing the constants as follows:

• For m_k with $k = 1, 2, 3, 4, 5$ compute $(k - 0.5) \times \hat{d}$ rounded up to six fractional bits. Since the sign detectors require the addition of $-m_k$, it is necessary to complement the digits of m_k (bit inversion for BCD-5211 code) and add a 1 in the least significant position (this 1 is incorporated as the carry bit in the least significant position of the resultant three to two reduction).

• For m_k with $k = -4, -3, -2, -1, -0$, we obtain $-m_k = m_{-k+1} + 1$. The addition of 1 is performed using the carry bit of the least significant position of the three to two reduction.

Therefore we compute the constants using arithmetic methods, avoiding large and slow look-up tables.

3. Divider Architecture

To represent the signed decimal operands x,d,Q and $w[i]$ we use a 10's complement representation of length $l = 4 \cdot n + 6$ bits (including a sign bit and 5 guard bits for the initial scaling by 20) for a n-digit precision quotient with decimal digits coded in BCD-5211.

The architecture for IEEE-754R Decimal64 format ($n = 16$ digits, $l = 70$ bits) is shown in Fig. 1. We only detail the architecture for significand computation, since other issues of floating-point division such as packing and unpacking from IEEE-754R floating–point format and sign and exponent calculations are straightforward. The division unit consists mainly of a 70-bit decimal adder and rounding unit (see Section 3.2), a module implementing the selection function with quotient digit recoding (detailed in Section 3.3) and a generator of divisor multiples and selection constants shown in Fig. 2. Fig. 2(a) shows the generation of positive full length divisor multiples (d, $2d$, $3d$, $4d$, $5d$). These multiples are precomputed and are not needed until after the selection of the first quotient digit q_1. We assume that the divisor d is unpacked in BCD-5211, so multiplication by 2 consists of a 1–bit wired left shift (L1shift does not require logic) followed by a digit recoding from BCD-4221 to BCD-5211 using combinational logic with no carry propagation between digits [14]. The $4d$ multiple is obtained by performing this sequence twice. Multiple $5d$ is generated first recoding from BCD-5211 to BCD-4221 and then performing a L3-shift. The generation of $3d = 2d + d$ requires a decimal carry-propagate addition that is performed in the 70–bit decimal adder. The result is stored in a latch to be available for the next iterations. The negative multiples are obtained from the corresponding positive multiples by bit inversion. This is performed by the level of XOR gates controlled by the sign of q_i (obtained in the previous cycle) and placed before the decimal adder and the selection function block in Fig. 1. In Section 2.2 we obtained that the selection function requires 15 leading bits of $-q_i d$ and so they need to be buffered. For a reduced latency implementation,

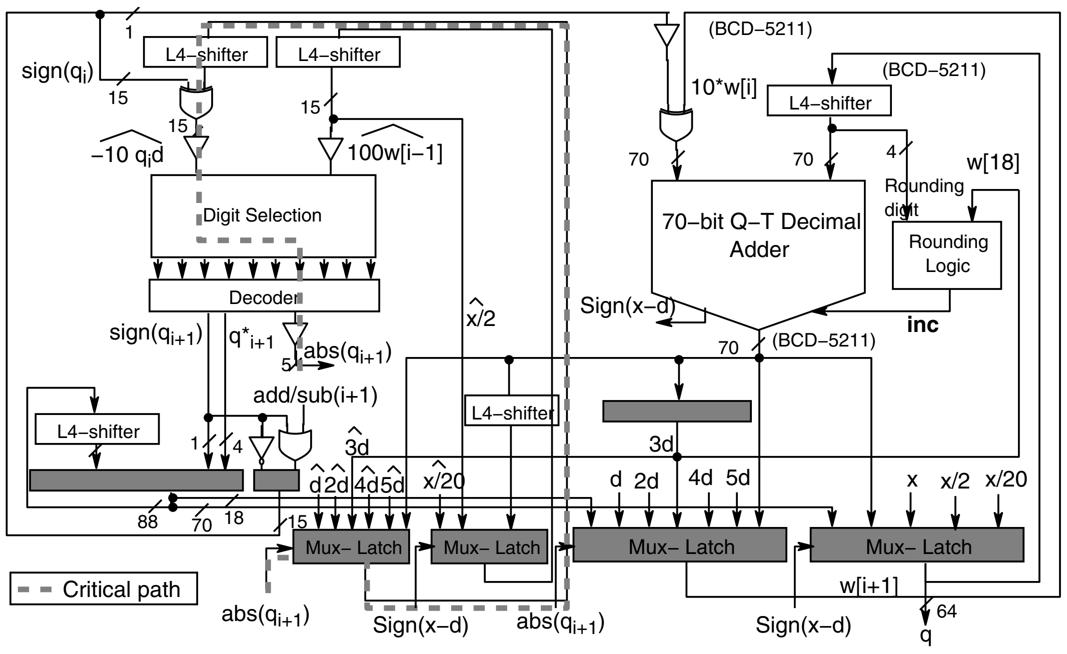
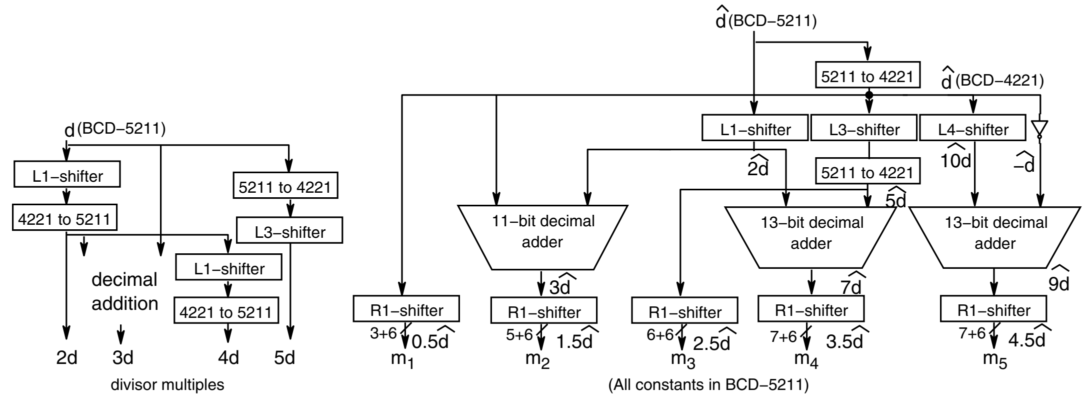


Figure 1. Datapath for the proposed radix–10 divider.


(a) Full length divisor multiples.

(b) Selection constants.

Figure 2. Generation of divisor multiples and selection constants.

the datapath (latches and multiplexes) for these leading bits is replicated and the sign bit of q_i is buffered and latched. The implementation of the precomputation of the selection constants described in Section 2.2 is shown in Fig. 2(b). The positive integer multiples of an estimate of the divisor are obtained as described before, but now one 11-bit and two 13-bit decimals adders (see Section 3.2) compute $3\widehat{d}$, $7\widehat{d}$ and $9\widehat{d}$. Selection constants $\{m_1, \ldots, m_5\}$ are obtained in BCD-5211 code from the integer multiples of $\widehat{d}$ (in BCD-4221 code) performing a 1-bit right wired shift (equivalent to $\times 0.5$). Other key components are the mux-latches that combine a wide multiplexer and a latch [11]. They are used to select and store the corresponding divisor multiples for the next iteration, the results coming from the decimal adder and the different values of initialization. The architecture was designed to fit the latches after the multiplexes to reduce latency. We now describe the operation of the divider.

3.1. Operation Sequence

The sequence of operations of the proposed architecture is as follows:

• Cycle 1. We assume $q_0 = 0$. We preloaded the narrow mux-latches with 0 and $\widehat{x/20}$ and the wide mux-latches with d and x. First, the decimal adder performs $x - d$. Then, the narrow mux-latch load 0 and $\widehat{x/20}$ (if $x - d \geq 0$, determined by examining the carry-out of the adder) or $\widehat{x/2}$ in other case. The wide mux-latches load $2d$ and d.

• Cycle 2. The selection function obtains q_1 as (q_1^*, s_1) while the decimal adder performs $2d + d = 3d$. The $3d$ multiple is stored in a dedicated latch. The wide mux-latches load $|q_1|d$ and $x/20$ ($x - d \geq 0$) or $x/2$ ($x - d < 0$). The narrow mux-latches load $\widehat{|q_1|d}$ and $\widehat{w[0]}$.

• Cycle 3 to $n + 3$ (recurrence iterations $i = 1$ to $i = n + 1$). In each iteration, the selection function performs $q_{i+1} = selection(100\widehat{w[i-1]}, -\widehat{10q_i d})$ while in parallel the decimal adder computes $w[i] = 10w[i-1] - q_i d$. At the end of the cycle $n + 3$, the quotient digits (q_1^*, s_1) to (q_{n+2}^*, s_{n+2}) are available .

• Cycle $n + 4$. The residual $w[n+2] = 10w[n+1] - q_{n+2}d$ is assimilated in the decimal adder. This cycle could be avoided by implementing a sign and zero detector at the expense of an additional hardware cost.

• Cycle $n + 5$. The decimal adder performs the subtraction $Q = Q^* - 10 \cdot S$ and rounds the result to n–digit precision (the round position is known). The rounding logic determines the +1 ulp conditional increment from the round digit, the rounding mode and the sticky bit. The BCD-5211 quotient is multiplied by 20 (5–bit left wired shift) to produce the normalized decimal quotient Q in BCD-4221. Note that only the most significant bit of q_{n+2}^* is needed , so the subtraction fits in the 70-bit wide format of the adder.

In summary, a n-digit decimal division is performed in $n + 5$ cycles. For the 16-digit divider of Fig. 1 a division is completed in 21 cycles. In the remaining Section we detail the implementation of the decimal adder and the selection function.

3.2. Architecture of the decimal adder

The algorithm for decimal addition is based on the conditional speculative method [13], but with digits in a BCD-5421 code instead of the more conventional BCD-8421. We use BCD-5421 code since recoding between BCD-5211 and BCD-5421 is much simpler than between BCD-5211 and BCD-8421 codings. A decimal digit coded in this BCD-5421 code has the following expression

$$Z_i = z_{i,3} \cdot 5 + \sum_{j=0}^{2} z_{i,j} \cdot 2^j \tag{7}$$

with the constraint $z_{i,3} = 1$ if $Z_i \geq 5$. Moreover, we adapt the architecture to operate with BCD-5211 coded numbers. Decimal adders implemented in the precomputation of the selection constants work with BCD-4221 decimal operands but its structure is basically similar to that described here. The adder performs both two-operand addition and subtraction $Z = X \pm Y = X + Y^*$, where Y^* is in 10's complement form for subtraction operations. The complement operation is performed by inverting the bits of Y coded in BCD-5211 and setting up the carry input to 1. Since the algorithm operates with BCD-5421 operands, input digits are first recoded. Then, a +3 increment is performed at each decimal digit position. This operation is carried out as a digit recoding (BCD-5421 to BCD-5421 excess3) of one input operand. This allows the computation of the decimal carries using conventional 4-bit binary carry–propagate additions, since decimal carries are equal to the binary carries at decimal positions. To obtain the decimal sum digits we compute a control signal A_i^L for each digit position i given by

$$A_i^L = \begin{cases} 1 & \texttt{If } X_i^L + (Y_i^*)^L \geq 5 \\ 0 & \texttt{Else} \end{cases} \tag{8}$$

where X_i^L, $(Y_i^*)^L$ represents the 3 less significant bits of X_i and Y_i^*. Sum digit Z_i (coded in BCD-5421) is computed speculatively by using a 4-bit binary carry-propagate addition and conditionally adding $+3$ as

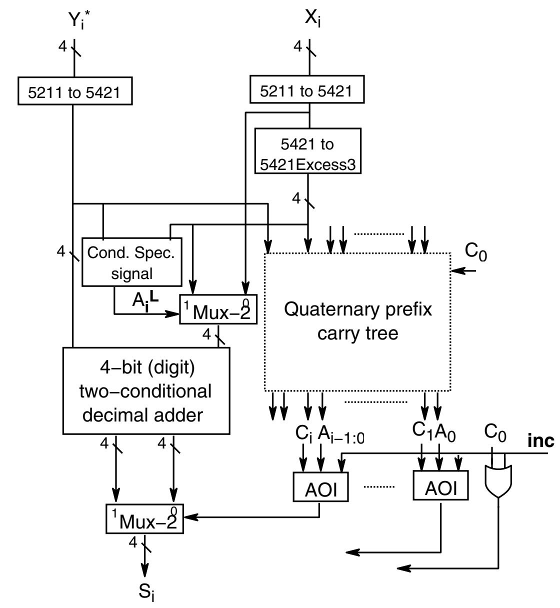
$$Z_i = \begin{cases} X_i + Y_i^* + 3 + C_i & \texttt{If}(A_i^L == 1) \\ X_i^U + Y_i^* + C_i & \texttt{Else} \end{cases} \tag{9}$$

where C_i is the decimal carry input into digit position i. This speculation produces a wrong sum digit for $C_i = 0$ and $mod_8\left(X_i^L + (Y_i^*)^L\right) = 7$ (the correct decimal sum digit is 4). The decimal correction is performed by removing the +3 value incorrectly speculated for this case. But instead of implementing this correction, we simultaneously perform a decimal correction and a digit recoding from BCD-5421 to BCD-5211. This consists of detecting the condition $C_i = 1$ and $mod_8\left(X_i^L + (Y_i^*)^L\right) = 3$ and adding $+3$. We implement it by replacing the binary carry-propagate sum expressions $Z_i^* = mod_{16}(X_i + Y_i^* + C_i)$ with the following expressions for $C_i = 1$:

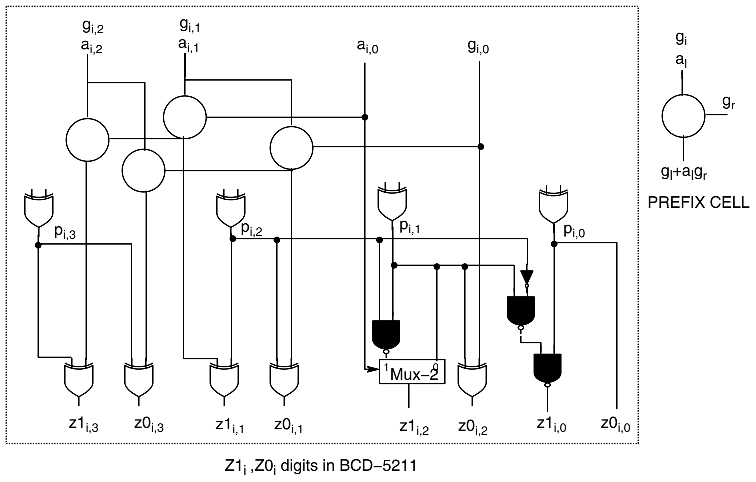
$$\begin{aligned} z_{i,0} &= \overline{p_{i,0}} \vee \overline{p_{i,2}} \cdot p_{i,1} \\ z_{i,1} &= z_{i,2}^* \\ z_{i,2} &= \overline{p_{i,2} \cdot p_{i,1}} \cdot a_{i,0} \vee p_{i,1}\overline{a_{i,0}} \\ z_{i,3} &= z_{i,3}^* \end{aligned}$$

where $\vee$ is the logical OR, $\cdot$ the logical AND, $p_{i,j}$, $a_{i,j}$ are the binary carry propagate and carry alive functions and the decimal sum digit is coded in BCD-5211. A digit slice of the 70-bit decimal adder implementing this algorithm is detailed in Fig. 3(a).

We have implemented the algorithm in a quaternary prefix tree (Q-T) topology [9]. Decimal carries are computed in a sparse prefix tree of 7 levels of logic depth. Sum digits



(a) Q-T decimal adder (digit slice).



(b) Two-conditional decimal digit sum cell.

Figure 3. Diagram block of the Q–T decimal adder.

are precomputed in parallel with carry evaluation using the array of two-conditional decimal sum cells shown in Fig. 3(b). The right sum digit is then selected by the corresponding decimal carry. These two–conditional cells are conventional 4-bit carry-select adders modified to obtain the correct BCD-5211 sum digits as described before. The black gates of Fig. 3(b) perform this decimal correction.

The architecture also implements a +1 conditional increment of the result for rounding operations. To avoid a new carry propagation, we implement a decimal late carry LC_i, which accounts for a decimal carry C_i or an input carry due to an increment operation. These late carries then select the appropriate sum digit computed in the array of two-conditional digit adders as $Z_i = Z1_i \cdot LC_i \vee Z0_i \cdot \overline{LC_i}$. A carry due to an increment operation is propagated from the less significant position to digit position i if the group alive function $A_{i-1,0} = \prod_{j=0}^{i-1} A_j$ is true [2]. The prefix adder only requires a minor modification to compute the carry alive groups $A_{i:0}$ [3]. Thus, LC_i carries are implemented as

$$LC_i = \overline{C_i} \cdot A_{i-1:0} \cdot inc \vee C_i = A_{i-1:0} \cdot inc \vee C_i \quad (10)$$

using the level of And-Or-Inverter (AOI) gates placed after the prefix tree, where signal inc controls a request from the rounding logic to increment the sum.

[2]For decimal speculative addition decimal group alive functions are computed as binary group alive functions.

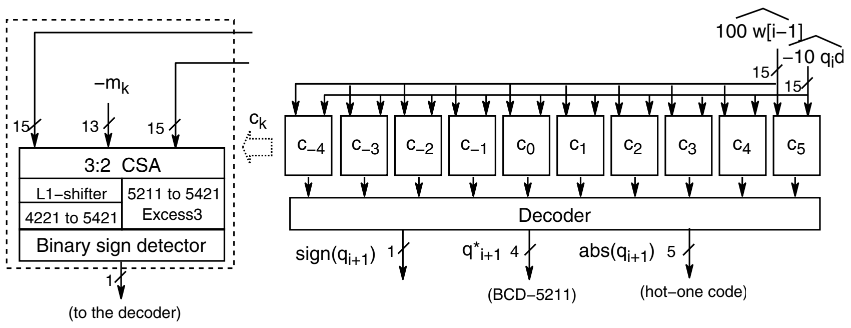
3.3. Selection function implementation

Fig. 4(a) shows the detail corresponding to the selection function part. The selection function is directly implemented using 10 decimal comparators. Each one compares the estimate of the residual $\widehat{10w[i]}$ ($\widehat{100w[i-1]}, -\widehat{10q_id}$) with one selection constant. If $\widehat{10w[i]} \geq m_k$ then the output of the comparator is 1. The value of q_{i+1} is obtained decoding the output of the comparators.

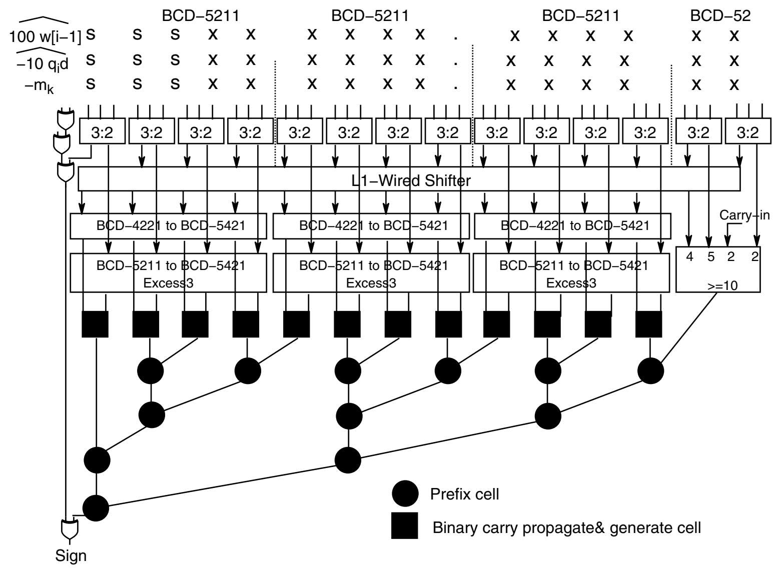
The decoder produces a sign bit $s_{i+1} = sign(q_{i+1})$ and the absolute value of q_{i+1} ($|q_{i+1}|$) in hot-one code for the control signals of the mux-latches. It also provides the q^*_{i+1} quotient digits coded in BCD-5211. The implementation of a decimal comparator is shown in Fig. 4(b). A level of binary 3 to 2 CSAs reduces the three input operands (in BCD-5211) to a two operand, a sum word $v_k[i]$ and carry word $h_k[i]$ coded in BCD-5211, that is

$$\widehat{100w[i-1]} - \widehat{10q_id} + m_k = v_k[i] + 2h_k[i] \quad (11)$$

The 2 $h_k[i]$ is performed by a L1-shift and a digit recoding from BCD-4221 to BCD-5211. The digits of $v_k[i]$ and $2h_k[i]$ are recoded to BCD–5421 and BCD-5421-excess3 to generate and propagate the decimal carries using a prefix tree and decimal speculative addition as explained in Section 3.2. The sign of the comparison is determined by the xor of the carry output of the prefix tree and the sign bits of the input operands.



(a) Digit selection block diagram.



(b) Bit level implementation of a decimal comparator.

Figure 4. Implementation of the digit selection function.

4. Evaluation and Comparison

In this Section we present the area-delay figures for the proposed architecture and compare it with two recent proposals of decimal dividers based on a radix-10 digit-recurrence algorithm [8, 10]. The delays are estimated in FO4 units (delay of an inverter with a fanout of 4 inverters) using a model based on logical effort [12]. The total stage delay is obtained as the sum of the delays of the gates on the critical path assuming equal input and output capacitances for the pipelined stage. Delay estimations take into account the buffering and the different gate loads, but not the wire delay. The hardware complexity is given as the number of equivalent minimum size NAND2 gates. The cost related to the area of a gate is a function of the number of transistors and its size (width). Although this is a rough area-delay model, it is good enough for making design decisions at gate level. Table 1 summarizes the evaluation results for the IEEE-754r Decimal64 divider using this model. Table 2 presents the area-delay figures for the architectures analyzed. We also include the delay figures of a software implementation of the IEEE-754R decimal64 FP division

Block	Area (NAND2)	Stage delay (FO4)
Sel. function (Fig. 4)	3200	22.3
Mult. generator (Fig. 2)	2000	18.4
Adder datapath (Fig. 3)	2600	21.8
Mux/latch	2700	3.0
Total	10500	25.3*

*Critical path delay (sel. function +mux/latch)
Precision digits = 16
Bits datapath=70
Cycles/division= 21

Table 1. Delay and area of the proposed divider.

Divider	Cycle time (FO4)	Cycles #	Latency (FO4)	Area (NAND2)
Proposed	25.3	21	531	10500
Ref. [10]	35.8	19	680	22600
Ref. [8]	26.8	20	536	13500
Ref. [5]*	≈ 22	294	6468	–
*Software library running in an Itanium 2 @ 1.4 GHz				

Table 2. Comparison results for area-delay (decimal64).

[5]. Since the area-delay figures for [8] are comparable to ours we have estimated its critical path delay using our area-delay model in order to provide fair comparison results. The hardware complexity in NAND2 gates was provided by the authors. For the other reference we use the figures provided by its synthesis results and expressed in terms of equivalent FO4 gate delays and number of equivalent NAND2 gates. Although the implementation proposed in [10] is for Decimal128 (n=34) we extrapolate the area figures assuming linear scaling, which is rather optimistic. From this comparison we conclude that our proposal is comparable in terms of latency to the best up-to-date implementation [8] and is more advantageous in terms of hardware complexity (about 1.3 area ratio). In addition, the software implementation analyzed [5] is an order of magnitude slower than the SRT radix-10 hardware implementations. With respect to hardware implementations based on multiplicative algorithms such as Newton-Raphson [15], comparison is difficult, since different issues should be taken into account: the use of a serial or a parallel decimal multiplier, the reuse of an existent multiplier or replication and the impact on the performance of other instructions that shares the same multiplier. However, since efficient decimal parallel multipliers have been recently reported [14], we expect that the design decisions for decimal division are close to those considered for binary division.

5. Conclusions

In this paper we present the algorithm and the architecture of a decimal division unit. The proposed radix-10 algorithm is based on the SRT digit-recurrence methods with a minimally redundant signed-digit set (ρ=5/9). The resultant implementation combines conventional methods used for high-performance radix-2^k division (SD redundant quotient and digit selection using selection constants and an estimate of the carry-save residual) with novel techniques developed in this work to exploit the particularities of radix-10 division. Among these new techniques are the use of non-conventional BCD codings to represent decimal operands, estimates by truncation at any binary position of a decimal digit and a customized decimal adder for several computations. We have implemented a 16 decimal digit divider that fits the IEEE-754R decimal64 format. Evaluation results show that our implementation presents comparable delay figures with respect to the best up-to-date implementation of a radix-10 digit-recurrence divider [8] but using less hardware complexity (1.3 area ratio).

References

[1] IEEE standard for floating–point arithmetic. IEEE Standards Committee, Oct. 2006. Available at http://754r.ucbtest.org/drafts/754r.pdf.

[2] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli. Digit-recurrence dividers with reduced logical depth. *IEEE Trans. Computers*, 54(7):837–851, 2005.

[3] N. Burgess. The flagged prefix adder and its applications in integer arithmetic. *Journal of VLSI Signal Processing*, 31(7):263–271, 2002.

[4] F. Y. Busaba et al. The IBM z900 decimal arithmetic unit. In *Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1335–1339, Nov. 2001.

[5] M. Cornea et al. A software implementation of the IEEE 754r decimal floating-point arithmetic using the binary encoding format. In 18th *Symposium on Computer Arithmetic*, pages 29–37, June 2007.

[6] M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In 16th *Symposium on Computer Arithmetic*, pages 104–111, July 2003.

[7] M. D. Ercegovac and T. Lang. *Algorithms for Division and Square Root*. Kluwer Academic Publishers, 1994.

[8] T. Lang and A. Nannarelli. A radix-10 digit–recurrence division unit: Algorithm and architecture. *IEEE Trans. Computers*, 56(6):727–739, June 2007.

[9] S. Mathew et al. A 4Ghz 130nm address generation unit with 32-bit sparse-tree adder core. *IEEE Journal of Solid-State Circuits*, 38(5):689–695, May 2003.

[10] H. Nikmehr, B. Phillips, and C.-C. Lim. Fast decimal floating-point division. *IEEE Trans. VLSI Systems*, 14(9):951–961, 2006.

[11] H.-J. Oh et al. A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor. *IEEE Journal of Solid–State Circuits*, 41(4):759–771, Apr. 2006.

[12] I. Sutherland, R. Sproull, and D. Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[13] A. Vázquez and E. Antelo. Conditional speculative decimal addition. In 7th *Conference on Real Numbers and Computers (RNC 7)*, pages 47–57, July 2006.

[14] A. Vázquez, E. Antelo, and P. Montuschi. A new family of high–performance parallel decimal multipliers. In 18th *Symposium on Computer Arithmetic*, pages 195–204, June 2007.

[15] L.-K. Wang and M. J. Schulte. Decimal floating-point division using Newton-Raphson iteration. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 84–95, Sept. 2004.