Unified Digit Selection for Radix-4 Recurrence Division and Square Root

Harris, David; Stine, James; Ercegovac, Milos; Nannarelli, Alberto; Parry, Katherine; Turek, Cedar

Published in:
IEEE Transactions on Computers

Link to article, DOI:
10.1109/TC.2023.3305760

Publication date:
2023

Document Version
Peer reviewed version

Unified Digit Selection for Radix-4 Recurrence Division and Square Root

David Harris[1], James Stine[2], Milos Ercegovac[3], Alberto Nannarelli[4], Katherine Parry[5], and Cedar Turek[1]
[1]Harvey Mudd College, [2]Oklahoma State University, [3]UCLA, [4]Technical University of Denmark, [5]Carnegie Mellon University
David_Harris@hmc.edu, james.stine@okstate.edu, milos@cs.ucla.edu, alna@dtu.dk, kparry@andrew.cmu.edu, cedarturek@gmail.com

***Abstract* – Division and square root are fundamental operations required by most computer systems. They are commonly implemented in hardware using radix-4 recurrence, which produces a 2-bit result digit on each step. Unified digit selection logic chooses the next quotient or square root digit based on a residual and divisor or square root approximation. This paper presents the first derivation of digit selection constants for unified radix-4 recurrence division and square root.**

Index Terms – division, square root, SRT, minimally-redundant radix-4

I. Introduction

Most computer instruction sets provide integer division with remainder, floating-point division, and square root. They are frequent enough to require efficient hardware implementations to achieve good system performance [1]. The two major approaches are *recurrence* and *multiplicative*.

Recurrence algorithms produce one digit of the result on each step. A simple radix-2 design (such as the Intel Merom integer divider) produces one bit per cycle, while a higher performance design such as the ARM11 [2] or Intel Penryn [3] uses one or two radix-4 stages to produce two to four bits per cycle. Allowing for pre and post-processing, ARM11 takes 15-29 cycles for single/double precision floating-point divide and square root, and Penryn takes 12-21 cycles.

Multiplicative algorithms, such as Newton-Raphson or Goldschmidt, look up an initial approximation and refine it to double the number of accurate digits on each step. Multiplicative algorithms generally reuse the existing fused multiply-add (FMA) unit, saving the need for dedicated recurrence hardware. Because of the power-hungry FMA, multiplicative algorithms use 6-8x more energy per operation and are falling out of favor [4].

Recurrence division resembles long division by paper and pencil. The partial remainder, or *residual*, is initialized to the dividend. At each step, the divider compares the residual to the divisor to compute the next quotient digit. It subtracts that multiple of the divisor from the partial remainder, then shifts the partial remainder by a digit and repeats until the full quotient is ready. To avoid a slow carry-propagate adder for the subtraction, the recurrence divider keeps the residual in carry-save redundant form. To avoid a slow full-precision comparison of the residual and divisor, the recurrence divider compares a low-precision estimate of the residual to *selection constants* that depend on the divisor. The estimate is either correct or only slightly wrong. The divider allows symmetric redundant quotient digits to fix the result on a subsequent step. Square root uses a closely-related approach and can share much of the hardware [5]. Ercegovac and Lang pioneered the *unified* approach to recurrence division and square root and wrote two authoritative books on the subject [6]-[7].

Radix $R = 2^r$ units produce r bits per step. Radix-2 recurrence dividers produce one quotient bit per step using a redundant digit set of {-1, 0, 1}. Radix-4 recurrence dividers produce two quotient bits per step using a minimally redundant digit set of {-2, -1, 0, 1, 2} or a maximally redundant digit set of {-3, -2, -1, 0, 1, 2, 3}. The maximally redundant set makes digit selection simpler because it can accommodate more error but requires calculating a 3x divisor multiple $3D = D + 2D$.

This paper presents the first complete derivation of selection constants for unified minimally redundant radix-4 recurrence division and square root. Compared to previous literature [6]-[7], this work also improves notation by moving the recurrence left shift after digit selection to avoid an unnecessary preshift, defining a thermometer code C_j for square root addend generation, and defining a selection interval M_k in place of the hats and stars. It concludes with verified hardware.

II. Prior Art

Experience has shown that minimally redundant radix-4 recurrence division and square root offers attractive speed and area. This is the strategy used in various Intel, IBM, ARM, and Weitek floating-point units. Intel Penryn does not disclose implementation details [3]. The IBM z13 uses recurrence division to produce three bits per cycle for division and two for square root, but also does not disclose details [8]. ARM presents a unified divide/square root unit with radix-4 selection constants developed by experimentation rather than derivation [2], [9]. The constants also rely on one extra bit of the residual, doubling table size compared to this work.

The literature comes tantalizingly close to deriving unified radix-4 selection constants but stops just short. The seminal textbooks on recurrence division and square root [6], [7] present a combined radix-2 algorithm but lack unified radix-4. The radix-4 division constants from Table 5.10 of [7] almost work for both but lack $m_{-1}(9/8) = -14$ and $m_{-1}(12/8) = -18$ needed for square root.

Weitek [10] and Liu et al. [11] derive radix-4 division/square root digit selection tables, but they are not valid for the initial few steps. Ercegovac and Lang derive square root selection constants without an initial PLA but did not unify them with division in their original work [5]. Nannarelli and Lang [12] present a unified divide/square root unit with the same incorrect selection constant $m_{-1}(9/8) = -15$ as in [7].

Nannarelli and Liu [13]-[4] correct the unified selection constants, consistent with Table V of this work. However, they omit the derivation, and cite [6], which lacks these constants.

In summary, minimally redundant radix-4 unified recurrence division and square root is a widely used technique for floating-point arithmetic, but the open literature is missing a reliable source for the selection constants.

Minimally redundant radix-4 is not the only recurrence divide/square root in commercial use. ARM describes recent low-latency divide/square root units producing 4-6 bits per cycle. One approach uses overlapped radix-4 stages with *prescaling* [14] that simplifies digit selection at the expense of some precomputation. Another uses overlapped radix-8 stages with separate digit selection constants for division and square root [15]. These strategies are beyond the scope of this paper.

III. Recurrence Division and Square Root

This section reviews the known recurrence division and square root algorithms, which operate internally on fixed-point numbers. Define U$a.b$ to be a fixed-point representation of an unsigned number with a integer bits and b fractional bits. Define Q$a.b$ to be a fixed-point representation of a twos-complement signed number with a integer bits (including the sign) and b fractional bits. For example, the bit string 1001 is interpreted as 2.25 in U2.2, -1.75 in Q2.2, and 0.5625 in U0.4.

A. Recurrence Division Algorithm

Recurrence division is defined for normalized positive floating-point significands $X, D \in [1,2)$, producing a quotient $Q = X / D \in (1/2,2)$. The divisor D must be normalized so that quotient digit selection depends only on the most significant bits. The quotient exponent is the diffference of the dividend and divisor exponents, and is not considered further. Subnormal floating-point numbers are pre-shifted into normalized form with a 1 in the most significant bit. Unsigned integer division also uses the same recurrence algorithm after preshifting integers into normalized form with a 1 in the msb. Signed integer division is performed on absolute values and then the sign is corrected later. Hence, without loss of generality, this paper assumes normalized U1.N_f inputs with a leading 1 followed by N_f fractional bits. $N_f = 23$ for single-precision and 52 for double-precision floating-point formats, or 31 or 63 for integer words and double-words. Q is computed in U1.b format. For floating-point, $b = N_f + 2$ to give one extra bit to for a postnormalization left shift into the range [1, 2) and another bit for rounding.

Q is generated as a weighted sum of radix-R redundant digits $q_i \in \{-a, \ldots, -1, 0, 1, \ldots, a\}$ with an integer digit q_0 and n fractional digits $q_1 \ldots q_n$. For minimally redundant radix 4, $R = 4$, $a = 2$, $n = \lceil b/2 \rceil$. The *redundancy factor* is defined to be $\rho = a / (R-1) = 2/3$.

$$Q_j = \sum_{i=0}^{j} q_i R^{-i} \quad (1)$$

$$Q = Q_n = \sum_{i=0}^{n} q_i R^{-i} \quad (2)$$

The *partial remainder* $X - DQ_j$ decreases on each step as the quotient approximation Q_j improves. The *residual* $W_j = R^{j+1}[X - DQ_j]$ describes this partial remainder shifted left by the number of steps so that it remains within similar bounds on each step. The quotient approximation Q_{-1} is initialized to 0 so residual W_{-1} is initialized to X. The residual is kept in carry-save redundant form $W = WS + WC$ for fast carry-save addition. Algorithm I describes recurrence division, which takes n+1 steps to compute the quotient digits. The *quotient selection circuit* (QSLC) uses the most significant bits of the divisor (D_H) and carry-save residual (WC_H, WS_H). In each step, the divisor multiple is subtracted from the redundant residual, and then the residual is multiplied by R. *An on-the-fly converter* (OTFC) produces the quotient Q without carry-propagate addition [7] as will be described below.

$WC_{-1} = 0$, $WS_{-1} = X$
for $j = 0$ to n
  $q_j = \mathrm{QSLC}(WC_{Hj-1}, WS_{Hj-1}, D_H)$
  $F_j = -q_j D$
  $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} << r$
  $\{Q_j, QM_j\} = \mathrm{OTFC}(Q_{j-1}, QM_{j-1}, q_j)$

Algorithm I Recurrence Division

The residual bound is found from the recurrence step $W_{j+1} = R(W_j - aD)$ by setting $|W_j|, |W_{j+1}| \le |W|$. Setting these bounds equal gives $|W| \le \rho RD$. Substituting this bound into the recurrence step gives a *containment condition* describing the range of residuals W_{j-1} for which QSLC can choose a quotient digit q_j.

$$\begin{aligned} \left|R(W_{j-1} - q_j D)\right| &\le \rho RD \\ (q_j - \rho)D \le W_{j-1} &\le (q_j + \rho)D \end{aligned} \quad (3)$$

The containment condition can be rewritten in terms of lower and upper bounds L_k and U_k for which it is acceptable to select the next quotient digit $q_j = k$.

$$L_k \le W_{j-1} \le U_k \quad (4)$$

$$L_k = (k - \rho)D \quad (5)$$

$$U_k = (k + \rho)D \quad (6)$$

Section IV discusses the QSLC function. The quotient Q is computed from the redundant quotient digits q_j according to EQ (2). To avoid carry propagation addition on the critical path, OTFC uses two shift registers holding Q and $QM = Q - 1$. Q and QM are initialized to 0 and -1, respectively. At each step, the registers shift left and appends new bits depending on the next quotient digit, as shown in Algorithm II. Keeping both Q and QM avoids the need for an adder.

$Q_{-1} = 0$; $QM_{-1} = -1$
for $i = 0$ to n

if	$q_i = 2$	$Q_i = \{Q_{i-1}, 10\}$	$QM_i = \{Q_{i-1}, 01\}$
else if	$q_i = 1$	$Q_i = \{Q_{i-1}, 01\}$	$QM_i = \{Q_{i-1}, 00\}$
else if	$q_i = 0$	$Q_i = \{Q_{i-1}, 00\}$	$QM_i = \{QM_{i-1}, 11\}$
else if	$q_i = -1$	$Q_i = \{QM_{i-1}, 11\}$	$QM_i = \{QM_{i-1}, 10\}$
else if	$q_i = -2$	$Q_i = \{QM_{i-1}, 10\}$	$QM_i = \{QM_{i-1}, 01\}$

Algorithm II Minimally-redundant radix-4 division on-the-fly conversion

At step n, Q_n contains the unrounded quotient. Further postprocessing may be necessary for rounding, subnorms, remainder calculation, and shifting or adjusting the sign of integers [7].

The recurrence division algorithm in Figure 5.1 of [7] involves left shifting the residual before rather than after quotient selection and carry-save addition, introducing an extra right shift during initialization and a left shift during postprocessing. The algorithm of [7] also assumes X and D in the range of [1/2, 1), involving another shift from floating-point significands. The notation of Algorithm I is slightly simpler and produces the same answer in the end.

B. Recurrence Square Root Algorithm

Floating-point square root $S = \sqrt{X}$ involves taking the square root of the significand and dividing the exponent by 2. The exponent must be even to divide exactly, so the significand may be shifted by one bit to compensate. Plausible choices of significand ranges are [1, 4), [1/2, 2), or [1/4, 1). [1/2, 2) complicates digit selection because the square root may or may not have a leading 1 in the integer digit. [1, 4) is also troublesome because the square root will be in the range [1, 2) so the integer digit may need to be either 1 or 2 given that fractional digits may be negative. Hence, square root generally preshifts the input X right by 1 or 2 bits to the range [1/4, 1) such that the exponent is even and S is in the range [1/2, 1). S also uses U1.b format with $b = N_f + 2$ to give one extra bit for the postnormalization left shift and another for rounding. Like the quotient, it is generated as a weighted sum of radix-R redundant digits s_i with one integer and n fractional digits. Subnormal floating-point numbers and integers are preshifted into this same form.

Like division, the square root approximation S_j at step j is the weighted sum of square root digits s_i. The unrounded square root is $S = S_n$. The error $X - {S_j}^2$ decreases on each step as the approximation improves. The residual W_j again describes this error shifted left by the number of steps.

$$S_j = \sum_{i=0}^{j} s_i R^{-i} \quad (7)$$

$$W_j = R^{j+1}\left(X - {S_j}^2\right) \quad (8)$$

The square root residual recurrence is obtained by substituting an expression for the next square root approximation into the residual definition EQ (8):

$$S_j = S_{j-1} + s_j R^{-j} \quad (9)$$

$$W_j = R^{j+1}\left(X - {S_j}^2\right) = R^{j+1}\left(X - \left(S_{j-1} + s_j R^{-j}\right)^2\right) \quad (10)$$

After simplifying, W_j can be expressed in terms of W_{j-1}. The square root recurrence is similar to the division recurrence, but instead of subtracting the quotient digit times the divisor, it subtracts the square root digit times twice the square root approximation, along with a term that decrease with the step j. After subtraction, the residual is shifted left by r.

$$W_j = R\left[W_{j-1} - \left(2S_{j-1}s_j + s_j^2 R^{-j}\right)\right] \quad (11)$$

Next, consider the containment condition defining the digit selection s_j. The difference between the square root approximation S_j at step j and the exact $\sqrt{X}$ must be bounded by the largest value obtained by choosing $s_i = a$ for all subsequent digits, which is

$$\left|\sqrt{X} - S_j\right| \le \sum_{i=j+1}^{\infty} aR^{-i} = \frac{a}{R-1}R^{-j} = \rho R^{-j} \quad (12)$$

With some algebra [7], this leads to a containment condition similar to EQ (3) for division. The major differences are that D is replaced by $2S_j$ and that the containment has an extra $\rho^2 R^{-j+1}$ term related to completing the square that decreases with step j.

$$\rho^2 R^{-j+1} - 2\rho R S_j \le W_j \le \rho^2 R^{-j+1} + 2\rho R S_j \quad (13)$$

Substituting EQ (9) and simplifying gives lower and upper bounds for which it is acceptable to select the next quotient digit $q_j = k$. Again, the bounds are similar to division, with D replaced by $2S_j$ and an extra term decreasing with j.

$$L_k \le W_{j-1} \le U_k \quad (14)$$

$$L_k = 2(k-\rho)S_{j-1} + (k-\rho)^2 R^{-j} \quad (15)$$

$$U_k = 2(k+\rho)S_{j-1} + (k+\rho)^2 R^{-j} \quad (16)$$

Algorithm III performs recurrence square root using a *square root selection circuit* (SSLC) that will be described in Section IV. The final result S is in the range [1/2, 1). If the integer digit s_0 is 0, the fractional digits can add up to at most 2/4 + 2/16 + 2/64 + … = ρ = 2/3, which is not large enough for some results. Hence, the initial square root approximation $S_0 = s_0$ is set to 1. According to EQ (10), the residual W_0 is initialized to $R(X–1)$. This is done in 2's complement arithmetic by making the integer bits all 1 and the fractional bits equal to X, then shifting left by r. Again, the residual is kept in carry-save redundant form. Because the integer digit is known, the square root algorithm uses one fewer step than division.

$WC_0 = 0,\ WS_0 = R(X–1)$
$s_0 = 1$
for $j = 1$ to n
  $s_j = \mathrm{SSLC}(WC_{Hj\text{-}1}, WS_{Hj\text{-}1}, S_{Hj\text{-}1}, j–1)$
  $F_j = -\left(2S_{j-1}s_j + s_j^2 R^{-j}\right)$
  $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} \ll r$
  $\{S_j, SM_j\} = \mathrm{OTFC}(S_{j\text{-}1}, SM_{j\text{-}1}, s_j)$

Algorithm III Recurrence Square Root

As compared to division, the square root recurrence algorithm F term is more complicated because it depends on the approximate square root S_{j-1} and has another term $s_j^2 R^{-j}$. The OTFC thus must properly align S_j in the most significant bits rather than simply shifting. These issues are both addressed by introducing a variable $K_j = R^{-j}$ indicating where the s_j^2 term is added and where to place the next digit in S_j. K_j has a single one that shifts right by r at each step. F generation is easier in hardware if we also introduce C_j, a thermometer-coded number with 1's in all the positions left of and including K_j:

$$C_j = -K_j = -R^{-j} = 1111.11..1100\ldots00 \quad (17)$$

The square root circuitry adds a shift register to hold C_j. It is shifted right arithmetically by r on each step. K_j is readily obtained from C_j with simple Boolean logic. Algorithm IV shows the minimally-redundant radix-4 OTFC that produces S and SM. It is similar to Algorithm II, but initializes S_0 to 1 and

Table I
Minimally Redundant Radix-4 Addend Generation (FSEL)

s_i	F_j			
	Mathematical	In terms of S_{j-1}, SM_{j-1}, and j	Bit String	In terms of C
+2	$-4S_{j-1} - 4K_j$	$4\left[\overline{S_{j-1}} + 4^{-(j-1)}\right] - 4(4^{-j}) = \overline{4S_{j-1}} + 12 \times 4^{-j}$	$\overline{a_0 a_1 a_2 . a_3 \ldots a_{r(j-1)}}\ 1100$	$\overline{(S_{j-1} \ll 2)}\ \&\ (C_j \ll 2)$
+1	$-2S_{j-1} - K_j$	$2\left[\overline{S_{j-1}} + 4^{-(j-1)}\right] - 4^{-j} = \overline{2S_{j-1}} + 7 \times 4^{-j}$	$\overline{a_0 a_1 . a_2 a_3 \ldots a_{r(j-1)}} 111$	$\overline{(S_{j-1} \ll 1)}\ \&\ C_j$
0	0	0	all zeros	0
-1	$2S_{j-1} - K_j$	$2[SM_{j-1} + 4^{-(j-1)}] - 4^{-j} = 2SM_{j-1} + 7 \times 4^{-j}$	$b_0 b_1 . b_2 b_3 \ldots b_{r(j-1)} 111$	$(SM_{j-1} \ll 1) \mid \left[C_j\ \&\ \overline{(C_j \ll 3)}\right]$
-2	$4S_{j-1} - 4K_j$	$4[SM_{j-1} + 4^{-(j-1)}] - 4(4^{-j}) = 4SM_{j-1} + 12 \times 4^{-j}$	$b_0 b_1 b_2 . b_3 \ldots b_{r(j-1)}\ 1100$	$(SM_{j-1} \ll 2) \mid \left[(C_j \ll 2)\ \&\ \overline{(C_j \ll 4)}\right]$

aligns the new digit in the appropriate columns using bitwise OR with K or $K \ll 1$. Table I shows the logic for adder-free F generation [7]. It involves small constant shifts of S, SM, and C. a and b are the Boolean representation of S_{j-1} and SM_{j-1}, respectively.

$S_0 = 1$; $SM_0 = 0$; $C_0 = -1$
for $i = 1$ to n
  $C_i = C_{i-1} \ggg 2$, $K_i = C_i\ \&\ \overline{(C_i \ll 1)}$
  if ($s_i = +2$) $S_i = S_{i-1} \mid K_i \ll 1$ $SM_i = S_{i-1} \mid K_i$
  else if ($s_i = +1$) $S_i = S_{i-1} \mid K_i$ $SM_i = S_{i-1}$
  else if ($s_i = 0$) $S_i = S_{i-1}$ $SM_i = SM_{i-1} \mid (K_i \ll 1) \mid K_i$
  else if ($s_i = -1$) $S_i = SM_{i-1} \mid (K_i \ll 1) \mid K_i$ $SM_i = SM_{i-1} \mid (K_i \ll 1)$
  else if ($s_i = -2$) $S_i = SM_{i-1} \mid (K_i \ll 1)$ $SM_i = SM_{i-1} \mid K_i$

Algorithm IV Minimally-redundant radix-4 square root OTFC

C. Unified Divide and Square Root Algorithm

The divide and square root algorithms are similar enough to share most of their logic [7]. Define the output of the unified logic to be U, which is either Q or S. Similarly, UM is one less than U (in the jth digit), and u_j is the jth digit of U. Algorithm V performs unified division and square root. It initializes U to either 1 or 0 for square root or division, and starts at step 1 or 0, respectively. On each step, it performs digit selection using three upper bits of D or S_{j-1}, as will be described in Section IV. It shifts C and K, and uses C for square root addend generation. It uses K for on-the-fly conversion to place the digits in their correct positions for both division and square root.

if (sqrt)
  $u_0 = 1$, $U_0 = 1.0$, $UM_0 = 0.0$
  $WC_0 = 0$, $WS_0 = R(X-1)$
  $C_0 = -1.0$, firststep = 1
else // divide
  $U_{-1} = 0.0$, $UM_{-1} = -1.0$
  $WC_{-1} = 0$, $WS_{-1} = X$
  $C_{-1} = -R$, firststep = 0
for j = firststep to n
  if (sqrt) $A_{j-1} = \mathrm{ASEL}(U_{Hj-1}, j-1)$
  else $A_{j-1} = D_{H\text{-fractional}}$
  $u_j = \mathrm{USLC}(WC_{Hj-1}, WS_{Hj-1}, A_{j-1})$
  $C_j = C_{j-1} \ggg r$, $K_j = C_j\ \&\ \overline{(C_j \ll 1)}$
  if (sqrt) $F_j = \mathrm{FSEL}(S_{j-1}, SM_{j-1}, u_j, C_j)$
  else $F_j = -u_j D$
  $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} \ll r$
  $\{U_j, UM_j\} = \mathrm{OTFC}(U_{j-1}, UM_{j-1}, u_j, K_j)$

Algorithm V Unified Recurrence Division and Square Root

IV. Minimally Redundant Radix-4 Digit Selection

This section derives the selection intervals for minimally redundant radix-4 QSLC and SSLC digit selection functions. QSLC is straightforward, while SSLC depends on the step j and requires case analysis. The two functions intersect such that a unified USLC function works for both. To our knowledge, this section has the first correct and complete published derivation of unified digit selection. These calculations are available in a spreadsheet [16].

The quotient or square root digit is selected based on the most significant bits of the residual W_H, along with either the divisor D_H or square root approximation S_H. To minimize the logic, use as few bits as possible while covering the range of inputs and ensuring the containment condition is satisfied despite truncation errors in the inputs.

For division, the divisor D is in the range [1, 2) in U1.N_f format with a 1 in the integer bit. For square root, the square root approximation S_{j-1} is in the range [1/2, 1] in U1.b format. Digit selection uses only the most significant bits. Ercegovac and Lang show that using D_H in U1.3 form is sufficient for division, with a maximum truncation error $D–D_H$ of 1/8 [7]. Similarly, S_H in U1.4 form is sufficient for square root with a maximum truncation error of 1/16. Square root digit selection depends on 2S, so the effective error is the same as division.

According to EQ (4-6), the range of W is $[L_{-2}, U_2]$. For division, $D < 2$, so the residual range is ((-2 – 2/3)(2), (2 + 2/3)(2)) = (-16/3, 16/3), which requires 4 integer bits to represent as a 2's complement number. For square root, the upper bound occurs on the first step $j = 1$, while the lower bound occurs for large j. S is in the range [1/2, 1]. By EQ (14-16), the residual range is $[2(-2 - 2/3)(1), 2(2 + 2/3)(1) + (2 + 2/3)^2 4^{-1}]$ = [-16/3, 64/9], which also fits in 4 integer bits. It is sufficient to work with 4 fractional bits of WC_H and WS_H, in Q4.4 format [7]. Add these redundant residuals and drop the least significant bit to obtain $W_H = WC_H + WS_H$ in Q4.3 format. WC_H and WS_H each have a truncation error in the range [0, 1/16) and dropping the bottom bit of W_H may introduce another truncation error of 1/16, so the total residual error $W–W_H$ is in the range [0, 3/16).

For a given divisor or square root approximation, there are overlapping residual ranges for which a digit of either k or k–1 can be selected. Define a corresponding *selection constant* m_k, and select the largest digit k such that $W_H \geq m_k$. m_k is in Q4.3 format like W_H. For minimally-redundant radix-4, selection reduces to four comparisons.

$$u_j = \begin{cases} +2 & W_{H\,j-1} \geq m_2 \\ +1 & m_1 \leq W_{H\,j-1} < m_2 \\ 0 & m_0 \leq W_{H\,j-1} < m_1 \\ -1 & m_{-1} \leq W_{H\,j-1} < m_0 \\ -2 & W_{H\,j-1} < m_{-1} \end{cases} \quad (18)$$

Next, find the *selection interval* M_k containing possible selection constants m_k. Because $W - W_H$ is in the range [0, 3/16), if $W_H \geq L_k$, then $W \geq L_k$, so $m_k \geq L_k$. If $W_H < m_k$, then $W_H \leq m_k - 1/8$ because W_H and m_k are multiples of 1/8. $W < W_H + 3/16 < m_k + 3/16 - 1/8 = m_k + 1/16$. Hence, choose $m_k \leq U_{k-1} - 1/16$. In summary, m_k must be a multiple of 1/8 in the selection interval

$$M_k = [L_k, U_{k-1} - 1/16] \qquad (19)$$

to ensure choosing a legal digit despite error in W_H. The next sections derive M_k and viable m_k for division and square root.

Note that the M_k notation in this work differs from the hats and stars used in [7]. We believe M_k is more readable.

A. Division

Fig. 1 shows a P-D diagram graphically illustrating which quotient digit(s) may be selected for a given partial remainder W_H and divisor D_H. It shows lower and upper bounds for selecting a given quotient digit q_j. The shaded overlap is the selection interval M_k. The heavy black lines are m_k values for each D_H, taken from Table VI. They must remain within the shaded overlap so that the selected digit is valid for any divisor in that interval.

Fig. 2 shows the how the selection interval M_k is bounded by (EQ 19). The width of the rectangle represents the divisor truncation error. Digit selection must work (satisfy EQ 19) for any divisor in that range. As the figure shows,

$$M_k = \begin{Bmatrix} \left[(k-\rho)\left(D_H + \frac{1}{8}\right), (k-1-\rho)D_H - \frac{1}{16}\right] & k > 0 \\ \left[(k-\rho)D_H, (k-1-\rho)\left(D_H + \frac{1}{8}\right) - \frac{1}{16}\right] & k \leq 0 \end{Bmatrix} \quad (20)$$

Table II summarizes these division selection intervals numerically.

B. Square Root

According to (EQ 15-16, 19) the selection intervals for square root vary with the step j because of the 4^{-j} term that is significant for small j. Ercegovac and Lang observed that the possible square root approximations S_{j-1} are limited for small j, making selection easier [5]. There is no simple way to find intervals that work for all j, so this section does a case analysis for small j, considers the general case for $j \geq 4$, and finds the intersection of all these cases. Each case is described below, and the numerical results are summarized in Table III.

For $j = 0$, Algorithm III always picks an initial digit $s_0 = 1$, so no selection function is needed.

For $j = 1$, $S_0 = 1.0$. The residual $W_0 = 4(X-1)$ is always nonpositive, so the selected digit must be nonpositive as well: $s_1 \in \{0, -1, -2\}$. This follows intuitively because the square root of a number between [1/4, 1) is never larger than 1. Moreover, W_0 is initialized in nonredundant form so the truncation error in W_{H0} is only 1/8 instead of 3/16. Therefore, the selection interval extends all the way up to U_{k-1} rather than $U_{k-1} - 1/16$:

$$M_k = [L_k, U_{k-1}] \quad (21)$$

Using EQ (15-16, 21) with $j = 1$, $S_{j-1} = 1.0$, $\rho = 2/3$, $R = 4$ gives

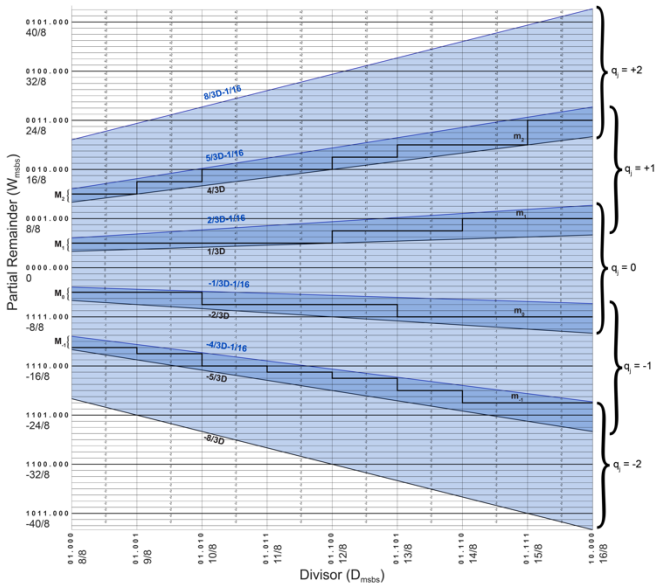

Fig. 1 Minimally redundant radix-4 division P-D diagram

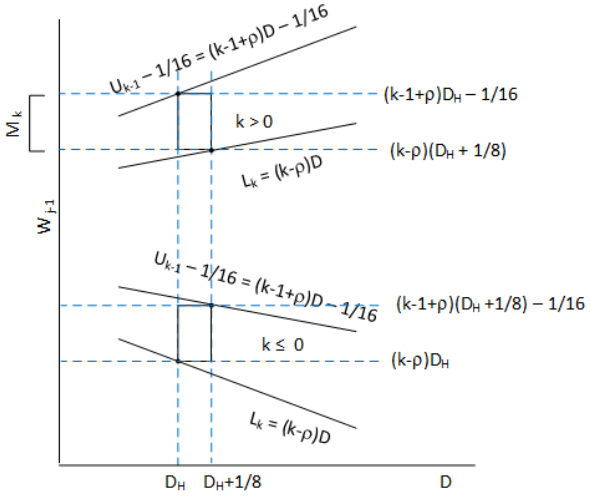

Fig. 2 Division selection intervals

$$s_1 = \begin{cases} 0 & -\frac{704}{576} \leq W_{H0} \leq 0 \\ -1 & -\frac{1520}{576} \leq W_{H0} < -\frac{368}{576} \\ -2 & W_{H0} < -\frac{1280}{576} \end{cases} \quad (22)$$

which is summarized in the $j = 1$ portion of Table III.

For $j = 2$, the possible square root approximations are $S_1 = 1.0 + s_1/4 \in \{8/16, 12/16, 16/16\}$. Only exact multiples of a quarter need to be considered. Furthermore, when $S_1 = 8/16$, s_2 will never be negative because the square root is in the range [1/2, 1). Similarly, when $S_1 = 16/16$, s_2 will never be greater than 0. The $j = 2$ portion of Table III shows selection intervals for these cases using EQ (15-16, 19).

For $j = 3$, the possible square root approximations are $S_2 = 1.0 + s_1/4 + s_2/16 \in \{8/16, 9/16, 10/16, 11/16, 12/16, 13/16, 14/16, 15/16, 16/16\}$. Only exact multiples of a sixteenth need to be considered. Again, when $S_2 = 8/16$, s_3 will never be

Table II
Minimally Redundant Radix-4 Division Selection Intervals (in units of 576ths)

D_H	1.000 8/8	1.001 9/8	1.010 10/8	1.011 11/8	1.100 12/8	1.101 13/8	1.110 14/8	1.111 15/8
M_2	[864, 924]	[960, 1044]	[1056, 1164]	[1152, 1284]	[1248, 1404]	[1344, 1524]	[1440, 1644]	[1536, 1764]
M_1	[216, 348]	[240, 396]	[264, 444]	[288, 492]	[312, 540]	[336, 588]	[360, 636]	[384, 684]
M_0	[-384, -252]	[-432, -276]	[-480, -300]	[-528, -324]	[-576, -348]	[-624, -372]	[-672, -396]	[-720, -420]
M_{-1}	[-960, -900]	[-1080, -996]	[-1200, -1092]	[-1320, -1188]	[-1440, -1284]	[-1560, -1380]	[-1680, -1476]	[-1800, -1572]

Table III
Minimally Redundant Radix-4 Square Root Selection Intervals (in units of 576ths)

$S_{H\,j-1}$	0.1000 8/16	0.1001 9/16	0.1010 10/16	0.1011 11/16	0.1100 12/16	0.1101 13/16	0.1110 14/16	0.1111 15/16	1.0000 16/16
$j = 1$									
M_2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
M_1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
M_0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	[-704, -368]
M_{-1}	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	[-1520, -1280]
$j = 2$									
M_2	[832, 1024]	n/a	n/a	n/a	[1216, 1504]	n/a	n/a	n/a	n/a
M_1	[196, 364]	n/a	n/a	n/a	[292, 556]	n/a	n/a	n/a	n/a
M_0	n/a	n/a	n/a	n/a	[-560, -320]	n/a	n/a	n/a	[-752, -416]
M_{-1}	n/a	n/a	n/a	n/a	[-1340, -1124]	n/a	n/a	n/a	[-1820, -1508]
$j = 3$									
M_2	[784,949]	[880, 1069]	[976, 1189]	[1072, 1309]	[1168, 1429]	[1264, 1549]	[1360, 1669]	[1456, 1789]	n/a
M_1	[193, 352]	[217, 400]	[241, 448]	[265, 496]	[289, 544]	[313, 592]	[337, 640]	[361, 688]	n/a
M_0	n/a	[-428, -251]	[-476, -275]	[-524, -299]	[-572, -323]	[-620, -347]	[-668, -371]	[-716, -395]	[-764, -419]
M_{-1}	n/a	[-1055, -884]	[-1175, -980]	[-1295, -1076]	[-1415, -1172]	[-1535, -1268]	[-1655, -1364]	[-1775, -1460]	[-1895, -1556]
$j \geq 4$									
M_2	[864, 924]	[960, 1044]	[1056, 1164]	[1152, 1284]	[1248, 1404]	[1344, 1524]	[1440, 1644]	[1536, 1764]	n/a
M_1	[216, 348]	[240, 396]	[264, 444]	[288, 492]	[312, 540]	[336, 588]	[360, 636]	[384, 684]	n/a
M_0	[-383, -252]	[-431, -276]	[-479, -300]	[-527, -324]	[-575, -348]	[-623, -372]	[-671, -396]	[-719, -420]	[-767, -420]
M_{-1}	[-953, -900]	[-1073, -996]	[-1193, -1092]	[-1313, -1188]	[-1433, -1284]	[-1553, -1380]	[-1673, -1476]	[-1793, -1572]	[-1914, -1572]
All j									
M_2	[864, 924]	[960, 1044]	[1056, 1164]	[1152, 1284]	[1248, 1404]	[1344, 1524]	[1440, 1644]	[1540, 1764]	
M_1	[216, 348]	[240, 396]	[264, 444]	[288, 492]	[312, 540]	[336, 588]	[360, 636]	[385, 684]	
M_0	[-383, -252]	[-428, -276]	[-476, -300]	[-524, -324]	[-560, -348]	[-620, -372]	[-668, -396]	[-716, -420]	
M_{-1}	[-953, -900]	[-1055, -996]	[-1175, -1092]	[-1295, -1188]	[-1340, -1284]	[-1535, -1380]	[-1655, -1476]	[-1775, -1572]	

Table IV
Unified Division and Square Root Selection Intervals (in units of 576ths)

A	000	001	010	011	100	101	110	111
M_2	[864, 924]	[960, 1044]	[1056, 1164]	[1152, 1284]	[1248, 1404]	[1344, 1524]	[1440, 1644]	[1536, 1764]
M_1	[216, 348]	[240, 396]	[264, 444]	[288, 492]	[312, 540]	[336, 588]	[360, 636]	[384, 684]
M_0	[-383, -252]	[-428, -276]	[-476, -300]	[-524, -324]	[-560, -348]	[-620, -372]	[-668, -396]	[-716, -420]
M_{-1}	[-953, -900]	[-1055, -996]	[-1175, -1092]	[-1295, -1188]	[-1340, -1284]	[-1535, -1380]	[-1655, -1476]	[-1775, -1572]

Table V
Unified Division and Square Root Selection Constants (in units of 8ths)

A	000	001	010	011	100	101	110	111
m_2	12	14	15, 16	16, 17	18, 19	19, 20, 21	20, 21, 22	22, 23, 24
m_1	3, 4	4, 5	4, 5, 6	4, 5, 6	5, 6, 7	5, 6, 7, 8	5, 6, 7, 8	6, 7, 8, 9
m_0	-5, -4	-5, -4	-6, -5	-7, -6, -5	-7, -6, -5	-8, -7, -6	-9, -8, -7, -6	-9, -8, -7, -6
m_{-1}	-13	-14	-16	-17	-18	-21, -20	-22, -21	-24, -23, -22

negative. The $j = 3$ portion of Table III shows selection intervals for these cases using EQ (15-16, 19).

For $j \geq 4$, the square root estimates are not exact multiplex of a sixteenth. Instead, we use the four msbs of S. Thus, S_H has a truncation error of up to 1/16. 4^{-j} has lower and upper bounds of [0, 1/256], so that term is minor.

Fig. 3 shows the selection intervals for $j \geq 4$. Each upper and lower bound has two parallel lines, one for $j = 4$, and the other for j approaching infinity. S is truncated to 4 fractional bits, introducing an error of up to 1/16. For positive digits ($k > 0$), the lower bound is set by the lower right corner of the box and the upper bound by the upper left corner. For negative digits, the left and right bounds are reversed.

L_k for $k > 0$ is an interesting case that can be relaxed slightly by considering specific values of j. The possible square root estimates are discrete, so the width of the rectangle is actually $1/16 - 4^{(j-1)}$; for example, at $j = 4$, the granularity of S is 1/64, so the width is 3/64. Hence, L_k is precisely

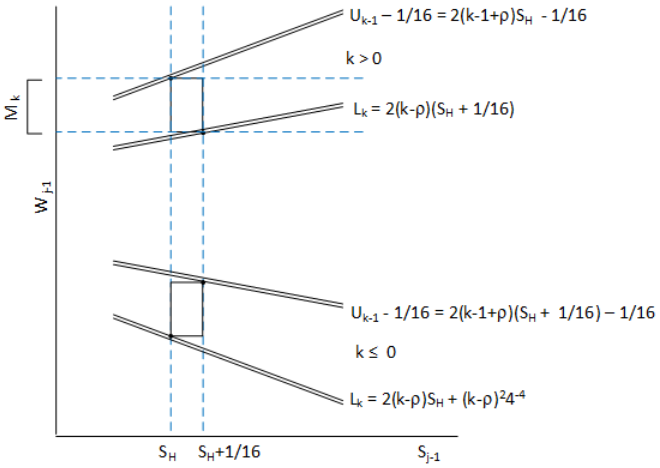

Fig. 3 Square root selection intervals for $j \geq 4$

$$L_k = 2(k-\rho)\left(S_H + \tfrac{1}{16} - 4\times 4^{-j}\right) + (k-\rho)^2 4^{-j} \quad (23)$$

The maximum value of L_k occurs as j approaches infinity, so Fig. 3 uses this conservative bound for all $j \geq 4$.

$$L_k = 2(k-\rho)\left(S_H + \tfrac{1}{16}\right) \quad (24)$$

Table III ($j \geq 4$) lists the selection intervals obtained using Fig. 3. Lower bounds for m_{-1} ending in .75 are rounded to the tighter integer bound. The case of S exactly equal to 16/16 has no error in S, so the bounds are

$$L_k = 2(k-\rho) + (k-\rho)^2 4^{-4} \quad (25)$$
$$U_{k-1} = 2(k-1+\rho) \quad (26)$$

Examining the M_k intervals in Table III, $j = 1$ is peculiar, but the intervals overlap for larger j. Table III (All j) shows a unified set of intervals taken as the maximum of the lower bounds and minimum of the upper bounds for $j = 2$, 3, and 4 or more. It uses the more restrictive of 15/16 and 16/16 for the 0.1111 column.

Returning to the $j = 1$ case, the viable selection constants are a superset of those for the 0.1101 column, so this column can be used for digit selection for $j = 1$.

In summary, square root digit selection uses the intervals from Table III (all j) by using the 0.1101 column for $j = 1$, the 0.1111 column for $S = 1.0$, $j > 1$, and otherwise picking the column based on the most significant fractional bits of S.

C. Unified Digit Selection

The square root selection intervals from Table III (all j) resemble those for division from Table II because the containment conditions are similar except for the 4^{-j} term. Both tables have 8 columns. Both tables have identical entries for M_2 and M_1 except in the rightmost column. For the remaining entries, the upper bounds are identical, and the lower bounds are tighter for square root. Hence, division and square root can share the digit selection logic by using the tighter square root selection intervals. Selection uses the most significant bits of D or S_{j-1}. The integer bit of D is always 1 and can be ignored. As described above, the case of $S = 1.0000$ is merged with $S = 0.1111$, so the most significant fractional bit of S can be treated as 1 and ignored as well. This leaves 3 interesting bits of D or S. To make a combined table, index the columns with a U0.3 variable A defined in Algorithm VI [7]. Table IV lists the unified digit selection intervals, which are identical to the square root entries.

Division: A = three most significant fractional bits of D
Square Root: Step $j = 1$ $A = 101$
Step $j > 1$ $A = 111$ if $S_{j-1} = 1.0$
A = three most significant fractional bits of $2S_{j-1}$ otherwise

Algorithm VI Divisor or square root bits to index unified digit selection table

Recall that the selection constants m_k must be multiples of 1/8 falling within the M_k intervals. They are typically scaled by the common denominator of 8 for brevity. If M_k spans the range $[a, b]$, compensating for the 576ths and 8ths, m_k is bounded by

$$\left\lceil \frac{8a}{576} \right\rceil \leq m_k \leq \left\lfloor \frac{8b}{576} \right\rfloor \quad (27)$$

The viable choices of m_k are listed in Table V. Choices of m_k affect the complexity of the digit selection logic. Table VI lists simple selection constants that repeat adjacent entries and have zeros in the lsbs where possible to simplify the logic. Table VII lists alternate selection constants that are symmetric for positive and negative entries, except for $m_{-1}(000)$ [13] .

Table VI
Simple Selection Constants (in units of 8ths)

A	000	001	010	011	100	101	110	111
$\mathbf{m_2}$	12	14	16	16	18	20	20	24
$\mathbf{m_1}$	4	4	4	4	6	6	8	8
$\mathbf{m_0}$	-4	-4	-6	-6	-6	-8	-8	-8
$\mathbf{m_{-1}}$	-13	-14	-16	-17	-18	-20	-22	-22

Table VII
Nearly Symmetric Selection Constants (in units of 8ths)

A	000	001	010	011	100	101	110	111
$\mathbf{m_2}$	12	14	16	17	18	20	22	23
$\mathbf{m_1}$	4	4	6	6	6	8	8	8
$\mathbf{m_0}$	-4	-4	-6	-6	-6	-8	-8	-8
$\mathbf{m_{-1}}$	-13	-14	-16	-17	-18	-20	-22	-23

V. Implementation

Fig. 4 shows the unified divide/square root unit. It contains registers to hold the redundant residual WS/WC, the OTFC approximate results U/UM, and the thermometer code C, along with the most significant bits of the divisor. The registers are initialized on the Start cycle for either divide or square root. On each step, the ASEL mux chooses A based on the operation and whether this is the first step of a square root. The USLC unified selection block computes $W_H = WS_H + WC_H$ and chooses the result digit u_j based on 7 bits of W_H in Q4.3 and 3 bits of A. USLC is coded as a lookup table, but synthesis optimizes it into standard cells. The digit is encoded with four bits indicating one of {-2, -1, 1, 2}. A digit of zero is encoded with all 0s. The FSEL block generates the appropriate addend F_j for division or square root. The CSA adds F_j to the redundant residual, then shifts the residual left to multiply by 4 before the next step. Meanwhile, the OTFC inserts the result digit into the appropriate column. The thermometer code shifts in two more 1s on each step to control conversion and square root addend generation.

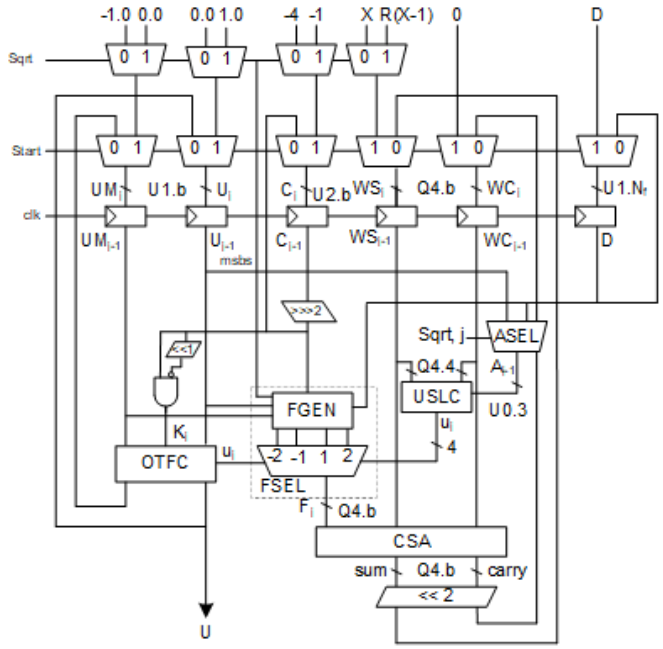

Fig. 4 Unified divide/square root hardware

Fig. 5 shows two implementations of the unified selection logic (USLC). The first is a direct implementation with an adder + table. The most significant bits of the redundant residual WS/WC_H in Q4.4 form are summed, and the least significant bit is discarded to produce the nonredundant residual W_H in Q4.3 form. Then a 1024-entry x 4-bit selection table is indexed with W_H and the three bits of A to produce u_j.

The second implementation receives the four selection constants $m_{2:-1}$ rather than A. It compares W_H to each selection constant using an 8-bit 3:2 CSA and sign-detection logic, then uses a priority encoder to select the digit.

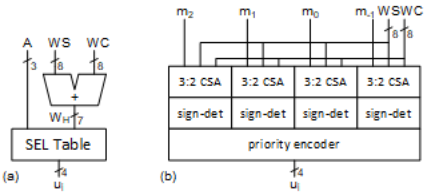

Fig. 5 USLC implementations: (a) adder + table vs. (b) comparators

Table VIII shows synthesis results targeting a 45 nm CMOS cell library with a 65 ps FO4 inverter delay and 1.06 μm^2 NAND-2 cell area. The comparator approach is 22% faster at the expense of almost twice the area and energy.

Table VIII
USLC Synthesis Results

Design	Delay (ps)	Area (μm^2)	Energy (pJ)
adder + table	514	370	0.10
comparators	400	715	0.18

The unified divide/square root unit using selection constants from Table VI is integrated into a pipelined RISC-V processor described in synthesizable SystemVerilog [17]. It is parameterized to support half, single, double, and quad precisions, and combinations thereof. It has been verified using the following test cases: Berkeley TestFloat [18] divide and square root for half, single, double, quad; IBM FPGen [19] divide and square root for single, double; and exhaustive test of single-precision square root. By modifying the selection constants, it also shows that five different constants recommended in previous division publications do not work for unified divide/square root.

VI. Conclusion

Recurrence division and square root are a subtle but essential and mature branch of computer arithmetic. Surprisingly, the literature is missing a proven table of digit selection constants for the widely-used case of unified minimally-redundant radix-4 division and square root. This paper fills the void, presenting unified digit selection logic supporting both operations.

Acknowledgment

This work was supported by the Clay-Wolkin Fellowship, and by a grant from Qualcomm.

References

[1] S. Oberman and M. Flynn, "Division algorithms and implementations," *IEEE Trans. Computers*, 46(8):833-854, 1997.
[2] N. Burgess and C. Hinds, "Design of the ARM VFP11 divide and square root synthesizable microcell," *IEEE Symp. Computer Arithmetic*, pp. 87-96, 2007.
[3] J. Coke et al, "Improvements in the Intel Core2 Penryn processor family architecture and microarchitecture," *Intel Technology Journal*, vol. 12, no. 3, pp. 179-192, 2008.
[4] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Trans. Computers*, vol. 61, no. 8, pp. 1059-1070, 2012.
[5] M. Ercegovac and T. Lang, "Radix-4 square root without initial PLA," *IEEE Trans. Computers*, 39(8):1016-1024, 1990.
[6] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.
[7] M. Ercegovac and T. Lang, *Digital Arithmetic*, Elsevier, 2004.
[8] C. Lichtenau, S. Carlough, and S. Mueller, "Quad precision floating point on the IBM z13," *IEEE Symp. Computer Arithmetic*, pp. 87-94, 2016.
[9] N. Burgess and C. Hinds, "Design issues in radix-4 SRT square root & divide unit," *Asilomar Conf. on Signals, Systems, and Computers*, pp. 1646-1650, 2001.
[10] J. Fandrianto, "Algorithm for high speed radix 4 division and radix 4 square root," *IEEE Symp. Computer Arithmetic*, pp. 73-79, 1987.
[11] Z. Liu, X. Song, Z. Wang, Y. Wang, and J. Zhou, "Constructing high radix quotient digit selection tables for SRT division and square root," *IEEE Trans. Computers*, TO APPEAR, 2023.
[12]A. Nannarelli and T. Lang, "Low-power radix-4 combined division and square root," *IEEE Intl. Conf. Computer Design*, pp. 236-242, 1999.
[13] A. Nannarelli, "Radix-16 combined division and square root," *IEEE Symp. Computer Arithmetic*, pp. 169-176, 2011.
[14]J. Bruguera, "Low latency floating-point division and square root unit," *IEEE Trans. Computers*, 69(2):274-287, 2020.
[15] J. Bruguera, "Low-latency and high-bandwidth pipelined radix-64 division and square root unit," *IEEE Symp. Computer Arithmetic*, pp. 10-17, 2022.
[16] D. Harris, "Unified minimally redundant radix 4 divsqrt selection intervals and constants," bit.ly/3fLR81z
[17] CORE-V Wally https://github.com/openhwgroup/cvw
[18] Berkeley TestFloat www.jhauser.us/arithmetic/TestFloat.html
[19] E. Guralnik, M. Aharoni, A. Birnbaum, and A. Koyfman, "Simulation-based verification of floating-point division," *IEEE Trans. Computers*, 60(2):176-188, 2011.