Low Latency Floating-Point Division and Square Root Unit

Javier D. Bruguera, *Senior Member, IEEE*

Abstract—Digit-recurrence algorithms are widely used in actual microprocessors to compute floating-point division and square root. These iterative algorithms present a good trade-off in terms of performance, area and power. We present a floating-point division and square root unit, which implements a radix-64 floating-point division and a radix-16 floating-point square root. To have an affordable implementation, each radix-64 division iteration and radix-16 square root iteration are made of simpler radix-4 iterations: 3 radix-4 iterations in division and 2 in square root. Speculation is used between consecutive radix-4 iterations to get a reduced timing. There are three different parts in digit-recurrence implementations: initialization, digit iterations, and rounding. The digit iteration is the iterative part and it uses the same logic for several cycles. Division and square root share partially the initialization and rounding stages, whereas each one has different logic for the digit iterations. The result is a low-latency floating-point divider and square root, requiring 11, 6, and 4 cycles for double, single and half-precision division with normalized operands and result, and 15, 8 and 5 cycles for square root. One or two additional cycles are needed in case of subnormal operand(s) or result.

Index Terms—Digit-recurrence algorithms, floating-point division and square root

1 Introduction

Division and square root are some of the most representative floating-point functions in modern processors. Although they are less frequent than the two basic arithmetic operations, addition and multiplication, a poor performance when computing these operations can impact the processor global performance.

For a low precision computation of these functions, it is possible to employ direct table look-up, bipartite tables [3], [24] (table look-up and addition), or low-degree polynomial or rational approximations [12], [15], [27], [28], but the area requirements become prohibitive for table-based methods when performing medium or high precision computations (such as the single and double-precision floating-point format). More efficient alternatives are iterative algorithms [7] On one hand, digit-recurrence methods [5], [7] have linear convergence and are based on subtraction; but their linear convergence sometimes leads to long latencies and makes them inadequate methods for these computations. High-radix digit-recurrence methods result in faster but bigger designs. On the other hand, multiplicative-based methods [4], [8], [20], [21], such as the Newton-Raphson and Goldschmidt algorithms, have quadratic convergence at the expense of greater hardware requirements.

The energy efficiency of both approaches has been recently analyzed and the conclusion is that the digit-recurrence approach is much more energy efficient than the multiplicative methods [17]. In addition, for the floating-point precisions of interest, double, single and half-precision, digit-recurrence methods are faster. Multiplicative methods rely on several iterations of a fused multiply-add (FMA) operation, and the latency of a single FMA is between 3 and 6 cycles [13], [22], [25]. In some cases, this is the latency of our proposed divider for single-precision.

- *Javier D. Bruguera is with ARM Ltd, Cambridge, UK. E-mail: javier.bruguera@arm.com*

In this paper the architecture of a floating-point unit implementing a radix-64 digit-recurrence divider and a radix-16 square root is described. A radix-r digit-recurrence algorithm, being r a power of 2, is an iterative algorithm where a radix-r digit, $log_2(r)$ bits, of the result quotient or root, is obtained every iteration. To get an energy and timing efficient implementation both the division and the square root iteration are obtained by overlapping simpler radix-4 iterations. Hence, three radix-4 division iterations are overlapped in a single cycle providing 6 bits of the quotient per cycle, which is equivalent to a radix-64 iteration. Similarly, two radix-4 square root iterations are overlapped in a single cycle providing 4 bits of the root per cycle, which is equivalent to a radix-16 iteration.

A digit-recurrence division or square root algorithm with floating-point operands has three parts: pre-processing, digit iterations, and post-processing. The pre-processing includes operands unpacking, operands normalization (if required) and initialization. Digit iterations is the iterative part of the digit-recurrence algorithm. Post-processing consists of the rounding logic and right-shift in case of a subnormal result (in division only). The pre-processing and post-processing logic is mostly shared between division and square root, whereas the iterative part, the digit iterations, are specific for either operation.

To improve the timing of the division iteration we have used the divisor pre-scaling technique [6]. This is a well-known technique to simplify the quotient-digit selection function, probably the most timing critical step in a digit-recurrence algorithm. The selection function is simplified by scaling the divisor to value close enough to 1.This way, the radix-4 selection function is independent of the divisor. This scaling is carried out before the digit iterations.

Additionally, some other techniques has been used to improve the timing or latency of division and/or square root,

- To reduce the timing, speculation is used between consecutive radix-4 iterations in the cycle.
- First iteration is not carried out in the iterative part but in the pre-processing stage; this way, latency is reduced by 1 cycle for given precisions.
  - Division: The first iteration, which gives the integer digit of the result, is carried out in parallel with the operands pre-scaling, contributing to save one cycle in single-precision.
  - Square root: The first iteration is skipped and integrated in the initialization stage. The partial root and the remainder for the second iteration are easily obtained from the input value. This way, the latency is reduced by 1 cycle in double and single-precision.

The result is a low-latency unit with 11, 6, and 4 cycles latency for double-precision, single-precision and half-precision division, respectively, and 15, 8, and 5 cycles latency for double-precision, single-precision and half-precision square root. These latencies include the pre-processing and post-processing cycles and correspond to a division or square root with normalized operands and result. In case of subnormal operands, one or two additional normalization cycles are needed. Similarly, in case of a division subnormal result an additional cycle after the rounding cycle is added to right shift the result and adjust the exponent.

This unit has been implemented in a processor with a frequency of 3 GHz.

The floating-point division has been already described in [1]. In this paper, the square root implementation is discussed and some more details about the division implementation are provided. Only the operations on the mantissas are shown; the sign and exponent are obtained separately.

The rest of the paper is organized as follows. Section 2 is a brief description of the foundations of digit-recurrence division and square root. In section 3 the general architecture is presented and the main features of the proposed unit are outlined. In sections 4 and 5 the implementation of both the divider and the square root is described. Some considerations about how subnormals operands and result are processed are given in Section 6. Finally, in Section 7 the unit is compared with other implementations in actual processors, and in Section 8 the main conclusions are presented.

2 Digit-recurrence algorithms for division and square root

Digit-recurrence is a class of iterative algorithms which compute a radix-r result digit p_{i+1}, with r a power of 2, and a remainder $rem[i]$ every iteration [5], [7]. The remainder is used to obtain the next radix-r digit. Note that each radix-r digit represents $log_2(r)$ bits of the results.

The partial result before iteration i is defined as

$$R[i] = \sum_{j=0}^{i} p_j \times r^{-j} \tag{1}$$

and each algorithm iteration is described by the following equations,

$$p_{i+1} = SEL(\widehat{rem}[i], \widehat{T}[i]) \tag{2}$$

$$rem[i+1] = r \times rem[i] - p_{i+1} \times f[i+1] \tag{3}$$

being $\widehat{rem}[i]$ and $\widehat{T}[i]$ estimations with a few bits of the remainder $rem[i]$ and the divisor (in case of division) or the partial result $R[i]$ (in case of square root), respectively. The number of bits in the estimation needed for the selection function SEL depends on the radix and the operation. Term $f[i+1]$ is different for each operation.

For a fast iteration, the remainder is kept in carry-save of signed–digit redundant representation. In our implementation, we have chosen a radix-2 signed-digit representation for the remainder, with a positive and a negative word.

On the other hand, note that because of the algorithm convergence conditions and the multiplication times r in equation (3), the remainder will have several bits in the integer part; the number of integer bits depending on the radix, the digit set, and the operation.

Then, every iteration a radix-r digit of the result is obtained from the current remainder, and a new remainder is computed for the next iteration.

Then, the number of iterations is

$$it = \lceil n/\log_2(r) \rceil \tag{4}$$

being n the number of bits of the result, including the bits required for rounding.

The number of cycles is directly related to the number of iterations and to the number of iterations performed per cycle. Then, considering m iterations per cycle, the number of cycles is

$$cycles = \lceil it/m \rceil \tag{5}$$

Equations (1) to (4) can be particularized to radix-4, $r = 4$, division and square root.

2.1 Radix-4 division

The floating-point division of a dividend x and a divisor d produces a quotient $q = x/d$. The partial quotient before iteration i and the digit obtained at iteration i are called $Q[i]$ and q_{i+1} respectively, then equation (1) is rewritten as

$$Q[i] = \sum_{j=1}^{i} q_j \times 4^{-j} \tag{6}$$

It has been shown that a simpler implementation is obtained if the divisor is scaled to be close to 1 [6]. In this case the selection function is independent on the divisor, then

$$q_{i+1} = SEL(\widehat{rem}[i]) \tag{7}$$

$$rem[i+1] = 4 \times rem[i] - q_{i+1} \times d \tag{8}$$

Note that $f[i+1] = d$, and the initial value for the remainder is $rem[0] = x$.

For this implementation, it has been determined that only the 6 most-significant bits (MSB) of the remainder, three integer bits and three fractional bits, are required to select the next quotient digit with equation (7) [7].

2.2 Radix-4 square root

The floating–point square root of the operand x produces a root $s = \sqrt{x}$. The partial root before iteration i and the digit obtained at iteration i are called $S[i]$ and s_{i+1} respectively, then equation (1) is rewritten as

$$S[i] = \sum_{j=0}^{i} s_j \times 4^{-j} \qquad s_0 = 1 \tag{9}$$

The square root iteration is defined by equations

$$s_{i+1} = SEL(\widehat{rem}[i], \widehat{S}[i]) \tag{10}$$

$$rem[i+1] = 4 \times rem[i] - s_{i+1} \times (2 \times S[i] + s_{i+1} \times 4^{-(i+1)}) \tag{11}$$

For square root

$$f[i+1] = 2 \times S[i] + s_{i+1} \times 4^{-(i+1)} \tag{12}$$

then

$$rem[i+1] = 4 \times rem[i] - s_{i+1} \times f[i+1] \tag{13}$$

The initial values for remainder and partial root are $rem[0] = x - 1$ and $S[0] = 1.0$, respectively.

In the selection function (equation(10)) 9 bits of the remainder, the 4 integer bits and 5 fractional bits, and 5 bits of the partial root, the integer bit and 4 fractional bits, are checked [7].

3 General Architecture and Main Features

The two floating-point operations being performed in this unit are division and square root. The general organization is shown in Figure 1. The three parts of the digit-recurrence implementation, pre-processing, digit iterations and post-processing, are clearly marked[1]. These three parts share the registers for the partial quotient/root, remainder, and some other signals.

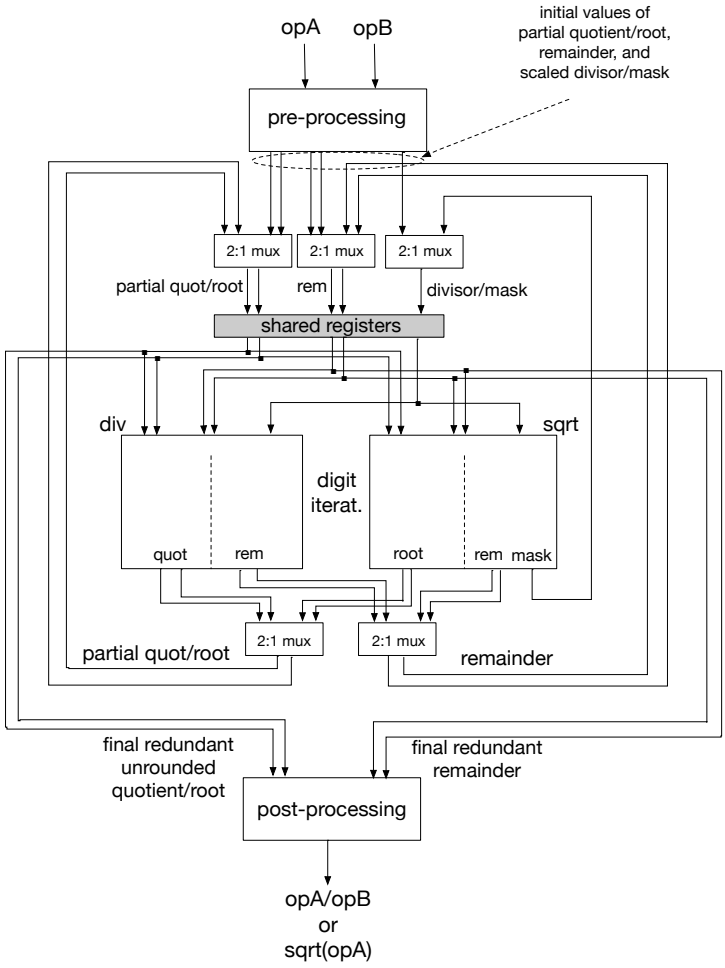

Fig. 1. General organization of the unit

In the pre-processing stage operands are unpacked, and the first iteration of division or square root is carried out. In addition, in case of division, divisor and dividend are scaled to put the divisor in a narrow interval around $+1$ so that the quotient-digit selection is simplified. Any subnormal operand is normalized in this stage; the normalization logic is used for 1 or 2 cycles depending on the number of subnormal operands. The partial quotient/root and remainder initial values are passed to the digit iterations.

The digit iterations stage is the iterative part of the digit-recurrence algorithm. Division and square root have a differentiate digit iteration logic, although the registers are shared. This logic is divided into two parts, (1) remainder calculation, and (2) quotient/root digit calculation and partial quotient/root updating. As stated in the introduction, radix-64 division and radix-16 square root iterations are obtained by overlapping simpler radix-4 iterations each cycle: three radix-4 iterations in division, and two radix-4 in square root. Every iteration a new value for the partial quotient/root and the remainder is computed. Redundant radix-2 signed-digit representation is used for both, partial result and remainder; therefore two words, positive word and negative word, are used to represent the remainder and partial result; The updated remainder and partial quotient/root are passed back for the next cycle. After the last iteration, the unrounded quotient/root and the final remainder are passed to the post-processing stage.

As said before, division and square root have separated digit iteration logic. However, there are some previous proposal combining the digit iteration logic for division and square root [2], [18]. Although the area would be smaller, in our implementation the stage timing would be affected; in particular, it would be impossible to have three radix-4 division iterations in the same cycle. As it will be shown in Sections 4.2 and 5.2, the remainder update is done with 3-to-2 carry-save adders in the division iteration, and with 4-to-2 carry-save adders in the square root iteration. In a combined unit 4-to-2 carry-save adders need to be used in both division and square root remainder update; consequently, three radix-4 iterations wouldn't fit in one cycle and the performance would be degraded. Note that the area would be smaller because part of the carry-save adders could be removed.

The post-processing stage rounds the quotient/root. Fi-

1. Pre- and post-processing can take more than 1 cycle. Pre-processing takes 1, 2 or 3 cycles whether there are none, one or two subnormal operands, respectively. Post-processing can take 2 cycles if the division quotient is subnormal. These additional pre- and post-processing cycles are not shown in the figure.

nal remainder and unrounded result need to be assimilated, that is subtract the negative word form the positive word, to get the remainders' sign and the non-redundant unrounded result, and the result is rounded to get the final quotient or root. In a floating-point division, the final quotient can be a subnormal number. In this case the mantissa needs to be right-shifted to have a IEEE standard compliant result [11].

The post-processing stage is not longer discussed because it is a traditional floating-point rounding stage.

The algorithm used for the division and square root is the radix-4 digit-recurrence algorithm with three (division) or two (square root) iterations per cycle, and with a signed-digit representation of the quotient with digit set $\{-2,-1,0,+1,+2\}$; that is, being $r=4$, $a=2$, the radix and the digit set respectively.

With the radix-4 algorithm, 2 bits of the quotient or root are obtained every iteration. As three radix-4 iterations are performed per clock cycle in division, 6 bits of the quotient are obtained every cycle, which is equivalent to a radix-64 divider. Similarly, two radix-4 iterations are performed per clock cycle in square root, and 4 bits of the root are obtained every cycle, which is equivalent to a radix-16 square root.

Let's now describe the main features of the division and square root implementations.

3.1 Floating-point division

The divider performs the floating-point division of a dividend, x, and a divisor, d, to obtain a quotient, $q=x/d$. The two operands need to be normalized, $x,d\in[1,2)$, although subnormal operands are accepted; in this case, the subnormal operands are normalized before the digit iterations.

If the two operands are normalized in $[1,2)$, the result is in $[0.5,2)$; this way two bits to the right of the least–significant bit (LSB) of the quotient are needed for rounding, the *guard* and the *round* bits. The guard bits is used for rounding when the result is normalized, $q\in[1,2)$, whereas the round bit is used for rounding when the result is not normalized, $q\in[0.5,1)$. In this latter case, the results is left-shifted by 1 bit, and the guard and round bits become the LSB and the guard bit, respectively, of the normalized result.

However, to simplify the rounding, the final quotient is forced to be in $q\in[1,2)$. Note that $q<1$ only if $x<d$. This situation is detected in pre-processing and the dividend is left-shifted by 1 bit in such a way that $q=2\times x/d$ and $q\in[1,2)$. Of course, the mantissa is the same as in x/d but the exponent needs to be decremented.

Each iteration, a digit of the quotient is obtained by means of a selection function. In order to have a quotient-digit selection function independent of the divisor, the divisor has been scaled close to 1. To preserve the result the dividend needs to be scaled by the same amount as the divisor.

In addition, note that the first quotient digit, which is the integer digit of the result, can take only values $\{+1,+2\}$, and its calculation is much simpler than the calculation of the remaining digits. Then, it is obtained in parallel with the operand pre-scaling, saving one cycle in single-precision.

3.2 Floating-point square-root

The unit computes the floating-point square-root of the operand, x, to obtain the root, $S=\sqrt{x}$. The operand needs to be normalized, $x\in[1,2)$, although a subnormal operand is accepted; in this case, the subnormal operand is normalized before the digit iterations.

The normalized operand needs to be right shifted by 1 or 2 bits before the iterations. This is because:

1) The algorithm requires the operand to be in $[0.5,1)$,
2) In case of an operand with an even exponent, after the shift to have the operand in $[0.5,1)$ the exponent becomes odd; then the mantissa is divided by 2 and the exponent is incremented to have again an even exponent; otherwise, the square-root cannot be calculated.

Then, the input to the iterations is,

$$x'=\begin{cases} x/2 & \text{if exponent is odd}\\ x/4 & \text{if exponent is even}\end{cases} \tag{14}$$

Consequently, $x'\in[0.25,1)$ and the root will be in $[0.5,1)$.

The first iteration is skipped and integrated into the initialization of the remainder and partial root, reducing the number of iterations by 1 without affecting the timing. The latency is related to the number of iterations and to the number of iterations per cycle, and skipping the first iteration results in 1-cycle latency reduction in double- and single-precision.

3.3 Early termination mode

There is an early-termination mode for exceptional operands. The early termination occurs when any of the operands, or the only square root operand, are NaN, infinity, or zero, or a power of 2 with normalized operands. In the latter case, three cases are differentiated,

1) Division by a power of 2. The result is obtained by merely incrementing or decrementing the exponent of the dividend.
2) Square root with an even power of 2. The operand is a power of 4. The result is a power of 2 and the calculation of the square root is carried out by merely halving the input exponent
3) Square root with an odd power of 2. The mantissa of the result is $\sqrt{2}$ and the exponent is obtained by halving the input exponent minus 1.

3.4 Latency

For normal operands and result, the latency is composed of 1 cycle for pre-processing , several cycles for digit iterations, and 1 cycle for post-processing. There are no cycles for operands' normalization, nor for right-shifting the result. Then, the latency is the number of cycles of the iterative part plus 2. Therefore, for practical floating-point formats, double, singe and half-precision, DP, SP and HP, respectively, the number of bits n, the number of iterations, the number of cycles (see equations (4) and (5)), and the latency are:

TABLE 1
Result digit representation

Digit	pos. word	neg. word
2	10	00
1	01	00
0	00	00
-1	00	01
-2	00	10

Division:

- DP: $n = 53$, $it = 27$, $cycles = 9$, $latency = 11$
- SP: $n = 24$, $it = 12$, $cycles = 4$, $latency = 6$
- HP: $n = 11$, $it = 6$, $cycles = 2$, $latency = 4$

Square root:

- DP: $n = 54$, $it = 27$, $cycles = 13$, $latency = 15$
- SP: $n = 25$, $it = 13$, $cycles = 6$, $latency = 8$
- HP: $n = 12$, $it = 6$, $cycles = 3$, $latency = 5$

Note that the number of bits n includes the guard bit for division; as the quotient has been forced to be in $[1, 2)$, only one additional bit for rounding is needed. On the other hand, in case of square root, the root is in $[0.5, 1)$ then two additional bits are required for rounding. In addition, the integer bit is not included in n. In division the integer bit is obtained in pre-processing stage. In square root the root has to be left-shifted by one bit and the first computed fractional bit becomes the integer bit.

3.5 Quotient/root and remainder representation

The partial result is saved in a redundant radix-4 signed-digit representation with 2 words: a positive word, $quot\_root\_pos$, storing the positive digits, and a negative word, $quot\_root\_neg$, storing the negative digits. Each iteration, the new computed result digit is concatenated to the actual partial result to get the new partial result.

For example, the final single precision result

$$quot\_root = 2\ 0\ 0\ (-1)\ 2\ 1\ (-1)\ 0\ 0\ 0\ 1\ 1\ 1$$

is represented as

$$\begin{aligned} quot\_root\_pos &= 2\ \ 0\,0\,0\ \ 2\,1\,0\ \ 0\,0\,0\ \ 1\,1\,1 \\ quot\_root\_neg &= 0\ \ 0\,0\,1\ \ 0\,0\,1\ \ 0\,0\,0\ \ 0\,0\,0 \end{aligned}$$

Note that if the quotient digit is 0, there is a 0 in both the positive and the negative words.

Therefore, four bits are used to represent each digit, 2 *positive* bits and 2 *negative* bits. The representation of each digit is shown in Table 1. Then, the binary result for the example above is

$$\begin{aligned} quot\_root\_pos &= 10.00\ 00\ 00\ 10\ 01\ 00\ 00\ 00\ 00\ 01\ 01\ 01 \\ quot\_root\_neg &= 00.00\ 00\ 01\ 00\ 00\ 01\ 00\ 00\ 00\ 00\ 00\ 00 \end{aligned}$$

The non-redundant result is obtained by subtracting the negative word form the positive word,

$$quot\_root = 1.11111110001100000010101$$

Note that in SP the last bit computed in the iterations is not part of the result, it is part of the sticky bit.Then the guard bit is the second LSB, $g = 0$

The remainder is also represented with two words, a positive word, rem_pos, and a negative word, rem_neg. This way, the remainder can be updated without any carry propagation.

4 Divider microarchitecture

In this section the divider is described. The divider microarchitecture has been already discussed in [1]. In the following the pre-processing stage, with focus on the operand pre-scaling and the calculation of the integer digit, and the digit iterations stage are presented.

4.1 Operand pre-scaling and integer digit calculation

Pre-scaling is a well-know technique to simplify the quotient digit selection function. Here we give a brief description of this technique. A detailed description can be found in [6].

The divisor is scaled to a value close to 1 so that the quotient-digit selection is independent of the divisor. As shown in [6], the selection function is quite simple: the remainder estimation is compared with a few low-precision wired selection constants.

If the divisor is not pre-scaled, the selection function in equation (7) would become $q_{i+1} = SEL(\widehat{rem}[i], d)$ Although the divisor is the same for every iteration and known in advance, it has to be taken into account for the quotient-digit selection. Consequently, a look-up table storing the selection constants for every quotient interval would be needed for the quotient-digit selection. This look-up table is not required when the divisor is pre-scaled.

It has been determined that for a radix-4 digit recurrence it is enough to have the scaled divisor in the range $[1 - 1/64, 1 + 1/8]$. The divisor is multiplied by a scaling factor $M = 1 + b \times 2^{-3}$, with $0 \leq b \leq 8$, and $b \neq 7$, which depends only on the three most-significant bits of the divisor. The pre-scaling can be implemented as the addition of the divisor plus 2 or 1 multiples of the divisor. The dividend has to be pre-scaled by the same amount to get the correct result.

The block diagram of this cycle is shown in Figure 2. During this cycle, in addition to the operands pre-scaling, the first iteration is carried out, and the operands are compared to force the result to be in $[1, 2)$]:

1) As part of the pre-scaling, redundant carry-save representations of pre-scaled divisor and dividend are obtained.
2) The redundant pre-scaled divisor and dividend are assimilated to a non-redundant representation, scaled divisor and scaled dividend in the figure, to get the remainder after the first iteration, $rem[1]$. The non-redundant divisor is used in the digit iterations as well (see equation (8)).
3) The operands are compared and the dividend is left-shifted by 1 bit if $x < d$. To save time, the comparison is carried out in parallel with the pre-scaling
4) In parallel with the operands' redundant to non-redundant conversion, the integer digit, q_1 of the quotient is obtained as well. This is a simplified digit quotient calculation, because as the quotient

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2947899, IEEE Transactions on Computers

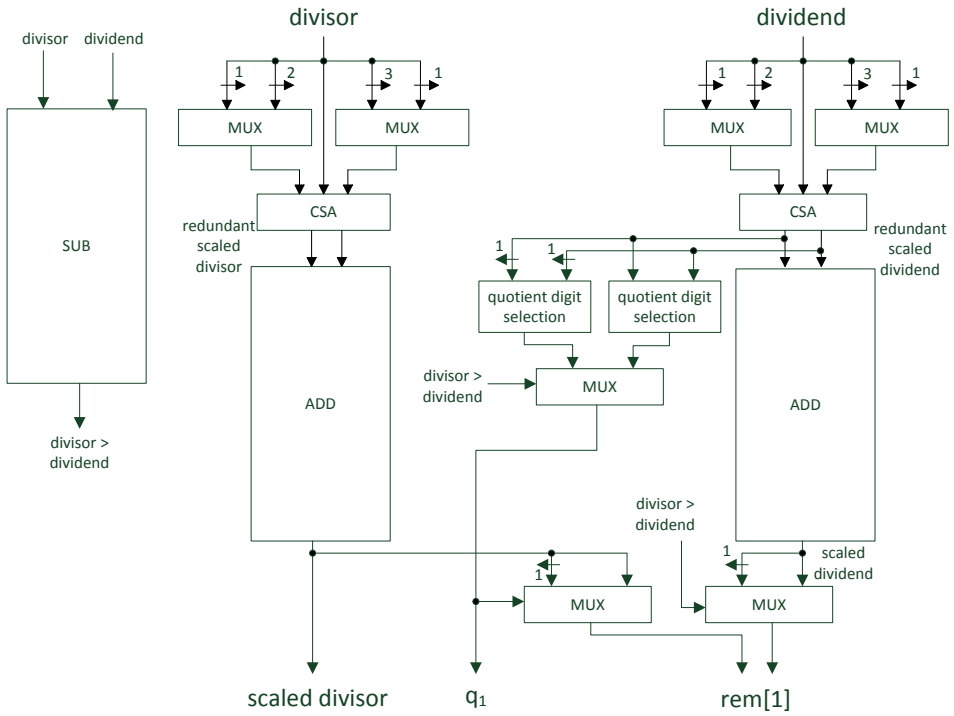

Fig. 2. Pre-processing stage for division: pre-scaling and integer quotient digit calculation

is positive and in $[1, 2)$, the integer radix-4 digit can only take values $q_1 = +1$, or $q_1 = +2$.
The integer digit calculation is replicated for $x < d$ and for $x \geq d$, obtaining two quotient digit candidates, for dividend larger than divisor and for dividend smaller than divisor. The result of the comparison selects the correct digit.

5) The next remainder, $rem[1]$, (see equation (3)) is obtained from the non-redundant scaled dividend (positive word of the remainder) and the non-redundant scaled divisor (negative word of the remainder). The operands comparison is used in the selection of the scaled dividend as well, in such a way that the scaled dividend is 1-bit left-shifted if the divisor is larger than the dividend. On the other hand, the scaled divisor is 1-bit left-shifted if the quotient digit is $+2$ and is not shifted if the quotient digit is $+1$.

4.2 Digit iteration

The actual implementation of the floating-point divider performs three radix-4 iterations per cycle. So, the logic has been optimized taking this fact into account. Figure 3 shows the block diagram of a digit iteration cycle; that is the computation of three radix-4 iterations. The block diagram in Figure 3 is split into two parts, (1) quotient digit selection and, (2) remainder calculation.

The remainders $rem[i+1]$, $rem[i+2]$ and $rem[i+3]$ are computed speculatively according to equation (8). So, five remainders are computed every iteration, one remainder for each possible value of the quotient digit; the correct remainder is selected when the digit is obtained. Note that, the remainder has to be left-shifted by two bits as part of the computation of the next remainder.

The quotient-digit selection uses an estimation of the remainder to obtain the next quotient digit (equation (7)). As said in Section 2.1, it has been determined that only the 6 most-significant bits (MSB) of the remainder are required, three integer bits and three fractional bits. The quotient digit selection function is shown in Table 2(a). The intervals $4 \times rem[i]$ for the selection of every digit has been obtained following the methodology described in [7].

For the selection of q_{i+1} the 6-bit remainder estimate is computed with the 6-bit adder in front of the first SELECT block.

To select digits q_{i+2} and q_{i+3} the 9 MSBs of the speculative $rem[i+1]$ are assimilated. Note that, the 9 MSBs are assimilated because although the selection function for q_{i+2} only needs the 6 MSBs, 2 additional bits are required for the selection of q_{i+3} because of the 2-bit left-shift of $rem[i+2]$, and the other additional bit is used to catch the carry into the least-significant position of the 8 bits.

Digit q_{i+1} selects the 9 MSBs, among the 5 speculatively calculated MSBs, that are going to be used in the selection of q_{i+2}. Note that only 6 bits are used in the selection.

The 6 MSBs so obtained may be different to the 6 MSB obtained directly from $rem[i+1]$, because the $+1$ to complete the 2's complement of the sum word in the assimilation of $rem[i+1]$ is added at a different position. In the actual implementation in Figure 3, it is added at the position of the 8th MSB whereas, in case of being obtained

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2947899, IEEE Transactions on Computers

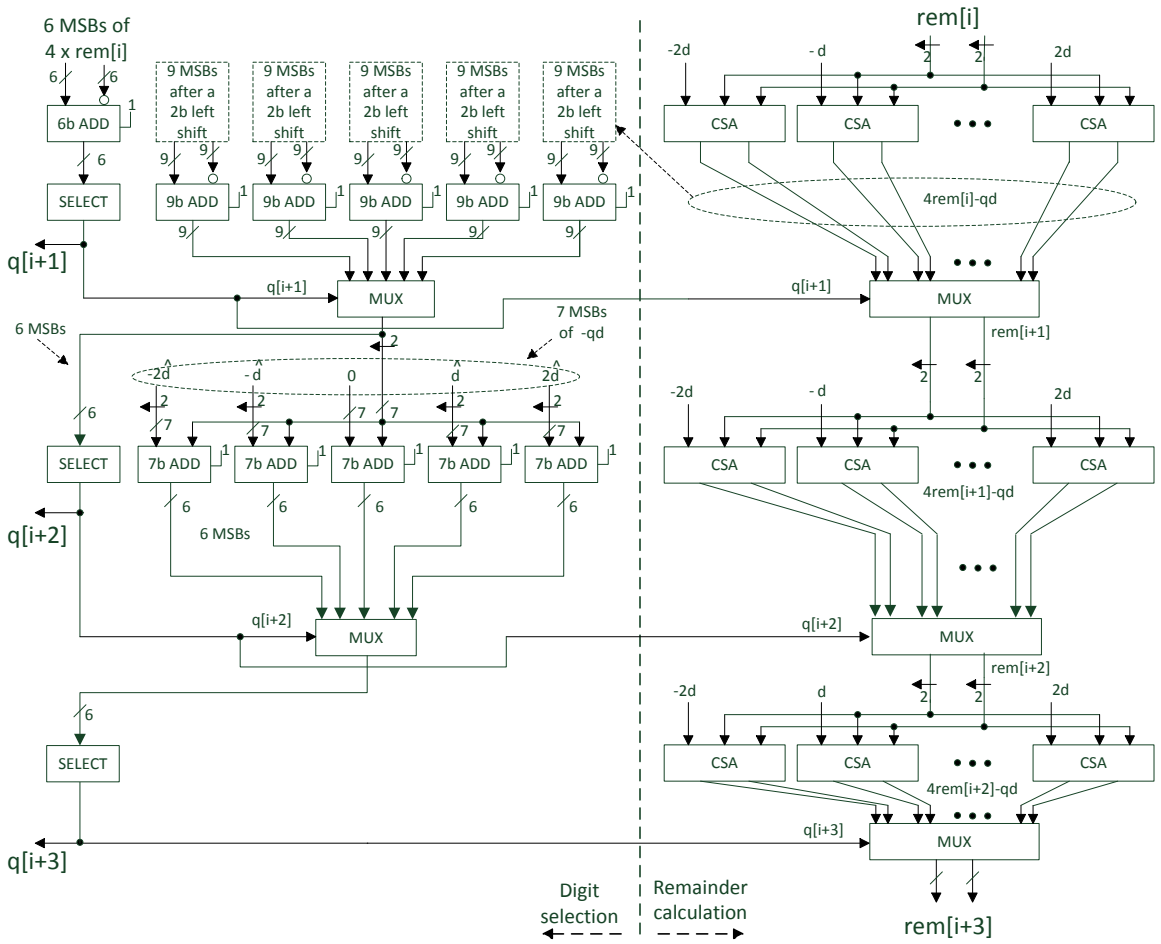

Fig. 3. Division digit iteration stage

directly from $rem[i+1]$, it would be added at the position of the 6th MSB. Consequently, the carry into the 6th MSB can be different. This difference makes the end-points of the intervals in Table 2(a) get a wrong selection when the carry into the 6th MSB bit is zero. This is corrected with the selection function shown in Table 2(b). Note that the selection of the interval end-points depends on the carry into the 6th MSB (*carry* column in the table) .

Therefore, it is clear that the carry into the 6th bit is required for the selection of the $q[i+2]$ digit. Hence, the 9–bit adder has to be split into a 6–bit adder and a 3-bit adder to get access to this carry.

In parallel with the selection of q_{i+2}, the 6 MSB to be used in the selection of digit q_{i+3} are computed speculatively for every value of q_{i+2}. Thus, the non-redundant estimation of $rem[i+2]$ is obtained in the five 7-bit adders, by adding the shifted 7 MSB of $rem[i+1]$ plus the 7 MSBs of $-q_{i+2} \times d$. Then, digit q_{i+2} is used to select the correct adder output, and q_{i+3} is selected according to Table 2(a).

This way, the delay of the logic in the cycle has been reduced with respect to a plain implementation of the three quotient-digit selection functions.

In the quotient-digit selection, *SELECT* block in the figure, the quotient digit is coded as a 1-hot 5-bit code $\{qp2, qp1, qz, qn1, qn2\}$, so that for example, $qp2 = 1$, $qp1 = qz = qn1 = qn2 = 0$ if $q_{j+1} = +2$. The logic function to get every bit in the 1-hot 5-bit code is relatively simple, a 3-level 2-input gate logic function. Each time a new digit is obtained it is concatenated to the partial quotient.

TABLE 2
Quotient-digit selection

$4 \times rem[i]$	q_{i+1}
$[13/8, 31/8]$	$+2$
$[4/8, 12/8]$	$+1$
$[-3/8, 3/8]$	0
$[-12/8, -4/8]$	-1
$[-32/8, -13/8]$	-2

(a) Standard selection intervals

$4 \times rem[i]$	carry	q_{i+1}
$31/8$	1	$+2$
$[13/8, 30/8]$	-	$+2$
$12/8$	0	$+2$
$12/8$	1	$+1$
$[4/8, 11/8]$	-	$+1$
$3/8$	0	$+1$
$3/8$	1	0
$[-3/8, 2/8]$	-	0
$-4/8$	0	0
$-4/8$	1	-1
$[-12/8, -5/8]$	-	-1
$-13/8$	0	-1
$-13/8$	1	-2
$[-32/8, -14/8]$	-	-2
$31/8$	0	-2

(b) Modified selection intervals

5 SQUARE ROOT MICROARCHITECTURE

In this section the square root implementation is described. The focus is in the skipping of the first iteration in the pre-processing stage, and in the digit iterations stage.

5.1 Latency reduction by skipping the first iteration

As explained in Section 2, the calculation of the square root is composed of several iterations, the number of iterations depending on the precision (number of bits) of the final result. The larger the number of iterations the larger the latency.

By skipping the first iteration the number of iterations is reduced by 1 and, for some floating-point precisions and organization, the latency is also reduced by 1 cycle. The first iteration is skipped and integrated in the initialization stage. The partial root and the remainder for the second iteration are easily obtained from the input value x' in equation (14).

5.1.1 New initial values for the partial root and the remainder

As the first iteration is being skipped, the initial values for the partial root and the remainder are those for the second iteration in case the first iteration is not skipped. To get these new initial values it has to be taken into account that the first digit can take only values $\{-2, -1, 0\}$, because the integer digit is 1, $s_0 = 1$, and the result is in $[0.5, 1)$. This reduced set of possible values for the first digit is still further reduced if odd and even input exponents are considered separately.

Let us consider the square root operand being

$$x = 1.x_0x_1x_2x_3x_4 \ldots x_{n-1}$$

As stated in section 3.2, this operand will be right-shifted by 1 or 2 bits, depending whether the operand's exponent is odd or even, to get x', the input to the square root digit-recurrence algorithm (see equation (14)).

In the following discussion, the initial values for remainder and partial root are $rem[0] = x' - 1$ and $S[0] = 1$, respectively (see Section 2.2)

Input operand with odd exponent

Operand x is right-shifted by 1 bit to the get x', with $x' \in [0.5, 1)$,

$$x' = 0.1x_0x_1x_2x_3x_4 \ldots x_{n-1}0$$

Then, the initial values for partial root and remainder are

$$\begin{aligned} S[0] &= 1 \\ rem[0] &= 1111.1x_0x_1x_2x_3 \ldots x_{n-1}0 \end{aligned}$$

and the shifted remainder is

$$4 \times rem[0] = 111x_0.x_1x_2x_3 \ldots x_{n-1}000$$

Under these circumstances, given the values of $4 \times rem[0]$ and $S[0]$, the only possible values for the root digit are $s_1 = \{-1, 0\}$, with

$$s_1 = \begin{cases} -1 & \text{if } x_0 = 0 \\ 0 & \text{if } x_0 = 1 \end{cases}$$

Putting together both cases, the redundant representation of the partial root is

$$\begin{aligned} S\_pos[1] &= 01.00 \\ S\_neg[1] &= 00.0\overline{x}_0 \end{aligned} \tag{15}$$

Then, the new initial remainder $rem[1]$, according to equation (11), is as follows

1) $s_1 = 0$. The partial root is obtained by the concatenation of s_1

$$S[1] = 1.00$$

and the remainder is

$$rem[1] = 4 \times rem[0]$$

In this case, as $rem[0]$ is non-redundant, the positive word of the signed-digit $rem[1]$ is $rem\_pos[1] = 4 \times rem[0]$, and the negative word is $rem\_neg[1] = 0$. Then,

$$\begin{aligned} rem\_pos[1] &= 111x_0.x_1x_2x_3 \ldots x_{n-1}000 \\ rem\_neg[1] &= 0000.000 \ldots 0 \end{aligned} \tag{16}$$

2) $s_1 = -1$. The partial root is obtained by the concatenation of s_1

$$S[1] = 0.75 = 0.11$$

and the remainder is

$$rem[1] = 4 \times rem[0] + f[1]$$

Now $rem[1]$ is the sum of $4 \times rem[0]$ and $f[1]$. Then $rem\_pos[1]$ is $4 \times rem[0]$, and $rem\_neg[1]$ is the 2's

complement of $f[1]$. From equations (12) it can be easily derived that

$$f[1] = 1.75 = 0001.110000$$

Therefore,

$$\begin{aligned} rem\_pos[1] &= 111x_0.x_1x_2x_3\ldots x_{n-1}000 \\ rem\_neg[1] &= 1110.010\ldots 0 \end{aligned} \quad (17)$$

Input operand with even exponent

Operand x is right-shifted by 2 bits to the get x', with $x' \in [0.25, 0.5)$,

$$x' = 0.01x_0x_1x_2x_3x_4\ldots x_{n-1}$$

Then,

$$\begin{aligned} S[0] &= 1 \\ rem[0] &= 1111.01x_0x_1x_2x_3\ldots x_{n-1} \end{aligned}$$

and

$$4 \times rem[0] = 1101.x_0x_1x_2x_3\ldots x_{n-1}00$$

Now, the first digit $s_1 \in \{-2, -1\}$:

$$s_1 = \begin{cases} -2 & \text{if } x_0 = 0 \\ -1 & \text{if } x_0 = 1 \end{cases}$$

Then, the redundant representation of the partial root is

$$\begin{aligned} S\_pos[1] &= 01.00 \\ S\_neg[1] &= 00.\overline{x}_0x_0 \end{aligned} \quad (18)$$

The new initial remainder $rem[1]$ is as follows

1) $s_1 = -1$. This is the same case as item 2) above, input operand with odd exponent, but with a different x' operand,

$$S[1] = 0.75 = 0.11$$

$$f[1] = 1.75 = 0001.110000\ldots$$

Then,

$$\begin{aligned} rem\_pos[1] &= 1101.x_0x_1x_2x_3\ldots x_{n-1}00 \\ rem\_neg[1] &= 1110.010\ldots 0 \end{aligned} \quad (19)$$

2) $s_1 = -2$. The partial root is

$$S[1] = 0.5 = 0.10$$

and the remainder is

$$rem[1] = 4 \times rem[0] + 2 \times f[1]$$

Now, $rem[1]$ is the sum of $4 \times rem[0]$ and $2 \times f[1]$. Then $rem\_pos[1]$ is $4 \times rem[0]$, and $rem\_neg[1]$ is the 2's complement of $2 \times f[1]$. From equation (12) it can be easily derived that

$$f[1] = f\_pos[1] - f\_neg[1] = 0001.100000\ldots 0$$

Therefore,

$$\begin{aligned} rem\_pos[1] &= 1101.x_0x_1x_2x_3\ldots x_{n-1}00 \\ rem\_neg[1] &= 1101.000\ldots 0 \end{aligned} \quad (20)$$

5.1.2 Putting it all together

The new initial values for $rem\_pos[1]$ and $rem\_neg[1]$ are given by equations (16), (17), (19) and (20). On the other hand, the initial values for $S[1]$ are given by equations (15) and (18). These are the value of the remainder and partial root for the second iteration. Now, all these equations need to be combined to get the new initial value for the remainder and the partial root.

New initial remainder

The new partial remainder is given by equations (16), (17), (19) and (20). From these equations, the negative word of the remainder can take the following values,

$$rem\_neg[1] = \begin{cases} 0000.000\ldots 0 & \text{if exp odd, } x_0 = 1 \\ 1110.010\ldots 0 & \text{if exp odd, } x_0 = 0 \\ 1110.010\ldots 0 & \text{if exp even, } x_0 = 1 \\ 1101.000\ldots 0 & \text{if exp even, } x_0 = 0 \end{cases} \quad (21)$$

The value of the negative word of the remainder, $rem\_neg[1]$, is easily set by selecting one of the four values above for the 6 most-significant bits during the unpacking or the normalization cycle, checking the less significant bit of the exponent and x_0. The remaining bits are all zero.

On the other hand, the values the positive word can take are

$$rem\_pos[1] = \begin{cases} 111x_0.x_1\ldots x_{n-1}000 & \text{if exp odd, } x_0 = 1 \\ 111x_0.x_1\ldots x_{n-1}000 & \text{if exp odd, } x_0 = 0 \\ 1101.x_0x_1\ldots x_{n-1}00 & \text{if exp even, } x_0 = 1 \\ 1101.x_0x_1\ldots x_{n-1}00 & \text{if exp even, } x_0 = 0 \end{cases} \quad (22)$$

Note that all these values corresponds to the fractional part of x' with two leading 1s and three trailing 0s:

$$11\ frac(x')\ 000$$

and can be easily obtained in the pre-processing stage.

New initial partial root

According to equations (15) and (18), the possible values of the partial root are

$$S\_pos = 01.00 \quad (23)$$

$$S\_neg = \begin{cases} 00.\overline{x}_0x_0 & \text{if exp even} \\ 00.0\overline{x}_0 & \text{if exp odd} \end{cases} \quad (24)$$

5.2 Digit iteration

The actual implementation of the floating-point square root performs two radix-4 iterations per cycle. So, the logic has been optimized taking this fact into account. Figure 4 shows the block diagram of a digit-iteration cycle; that is the computation of two radix-4 iterations. Note that, the implementation in figure is split into two parts, (1) remainder calculation, and (2) digit selection and root updating.

As for division, the remainder is speculatively computed according to equation (11). So, 5 remainders are computed every iteration, one remainder for each value of the root digit, and the correct remainder is selected when the digit has been obtained. Note that, the remainder has to be left-shifted by two bits as part of the computation of the next remainder.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2947899, IEEE Transactions on Computers

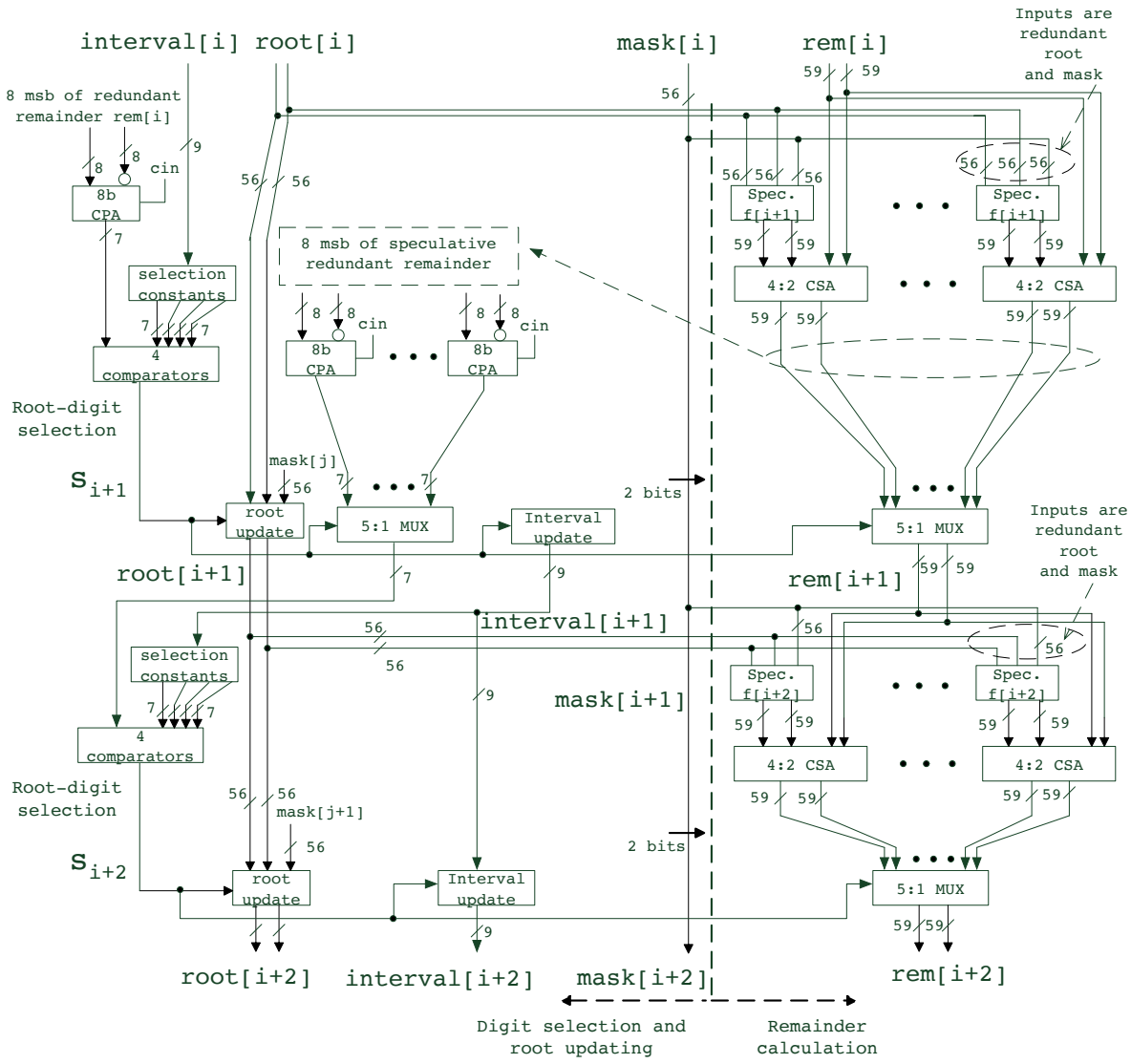

Fig. 4. Square root digit iteration stage

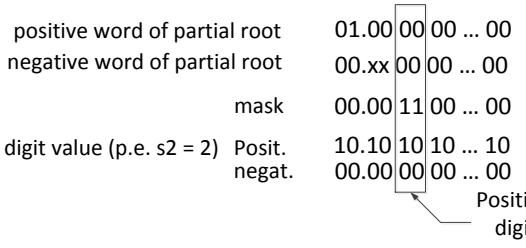

Fig. 5. Initial value for the mask and digit concatenation

The computation of remainders $rem[i+1]$ and $rem[i+2]$ involves the calculation of $f[i+1]$ and $f[i+2]$, respectively, according to equation (12).

As shown in equation (12), $f[i+1]$ is obtained by concatenating (no addition needed) the selected digit s_{i+1} to the shifted partial root $2 \times S[i]$. Note that this equation differs of the updating of the partial root (equation (9)) only in the left shifting of $S[i]$. To save the digit position where the s_{i+1} digit has to be concatenated, a mask ($mask[i]$ in the figure) is used. This mask has the same wordlength as the root, and it is initialized to $0000110\ldots0$. Note that each digit has two bits including digit s_0. As shown in Figure 5, the two 1s of this initial value are pointing to the second fractional digit, because the first iteration has been integrated into the initialization step (see Section 5.1).

Then, the 5 different speculative $f[i+1]$ are obtained, one $f[i+1]$ per digit value. The $f[i+1]$ are obtained in redundant form as well, with a positive word and a negative word. In case of a positive digit, the digit is concatenated to the shifted positive partial root word and 0s are concatenated to the shifted negative partial root word. Similarly, in case of a negative digit, the digit is concatenated to the shifted

negative partial root word and 0s are concatenated to the shifted positive partial root word. This updating is similar to the updating of the partial root shown in Figure 5, but shifting the partial root before the concatenation.

To keep the mask always updated, pointing to the position where the next digit has to be concatenated, the mask is right–shifted by 2 bits at the end of every iteration.

The left part of the figure is the digit selection and root updating, according to equations (10) and (9) respectively. The root update make use of the mask described above for the concatenation of the obtained root digit to the partial root.

The digit selection is implemented differently in the first and the second iteration in the cycle. The root digit selection requires to obtain an estimate of the remainder and of the partial root; in particular, the estimate of the partial root is required to determine the interval of the partial root [5] [7]. Both, the remainder and the partial root, are kept in redundant representation. It has been determined that the 9 MSBs of the remainder (4 integer bits and 5 fractional bits) and the 5 MSBs of the partial root (the integer bit and 4 fractional bits) are enough for the root digit selection.

The assimilation of the 5 MSBs of the partial root gives the partial root interval. Taking into account that the partial root is in $[0.5, 1)$, that is the lower half $[0, 0.5)$ of the range is not used, only 8 intervals, $I_0, \ldots, I_7$, are possible. Moreover, an additional interval I_8 is considered for the partial root being $S[i] = 1.0 \ldots 0$.

Once the interval is known, four selection constants are obtained; these selection constants correspond to the lower points of the remainder intervals for a root-digit equal to -1, 0, $+1$, and $+2$ [5], [7]. The selection constants are 7-bit width; so only the 7 most–significant bits of the remainder estimate has to be compared against these selection constants to determine the next-root digit. Then, using a set of four 7-bit comparators, the root-digit is obtained.

To speed up the digit selection in the second iteration in the cycle and to balance the delay of the remainder calculation and the digit selection in the first iteration in the cycle, the 9 most–significant bits (MSB) of the remainder $rem[i+1]$ are speculatively assimilated to its non-redundant representation. This allows to eliminate the 9-bit adder in the selection of s_{i+2}.

6 Subnormal operands and/or result

The microarchitecture described in sections 4 and 5 assumes the inputs are normal. In case of having any subnormal, the input pre-processing stage is a bit different: subnormal operands are normalized after the operand unpacking, and before the pre-scaling (division) or the skipping of the first iteration (square root). Consequently, because of the normalization, the input pre-processing needs more than one cycle.

In case of a subnormal result the post-processing stage requires two cycles: rounding and result normalization.

The digit iterations stage remains the same.

7 Evaluation

In this section we evaluate the proposed division/square root unit in terms of latency, area, and timing, and compare it with other recent divider/square root or division only units in commercial processors.

Unfortunately, the references with the description of the division and square root units of the commercial processors that are included in this comparison does not contain any information about static or dynamic power. Consequently, although power, static and dynamic, is also an important metric, it has not been included in this discussion.

TABLE 3
Division and square root latencies

	Division			Square root		
	DP	SP	HP	DP	SP	HP
Normal inputs, normal result	11	6	4	15	8	5
Normal inputs, subn result	12	7	5	–	–	–
One subn input, normal result	13	8	6	16	9	6
One subn input, subn result	14	9	7	–	–	–
Two subn input, normal result	14	9	7	–	–	–

7.1 Latency

The latency of division and square root calculation with the proposed unit is shown in Table 3. The table shows the latency for the three floating-point precisions of interest, double, single and half-precision, DP, SP and HP, respectively, and for any combination of normal and subnormal input operands and result. Note that the square root result cannot be subnormal and that it has only one operand.

The latency is the sum of the number of cycles devoted to pre-processing, plus the number of digit iterations cycles, plus the number of cycles for post-processing.

The number of pre-processing cycles depends on the number subnormals operands and on the operation. In division it can be one, three, or four pre-processing cycles. If both operands are normal unpacking and pre-scaling are done in the same cycle. In case of one subnormal operand there are three independent cycles for unpacking, normalization and pre-scaling. Finally, if both operands are subnormal there are one cycle for unpacking, two normalization cycles, and one cycle for pre-scaling.

Similarly, for square root there can be one pre-processing cycle for unpacking if the operand is normal or two pre-processing cycles, for unpacking and normalization, if the operand is subnormal.

The number of digit cycles depends on the number of result bits required. In division the number of fractional bits of the quotient the algorithm has to obtain is 53 for double precision (52 fractional bits plus the guard bit), 24 for single precision (23 fractional bits plus the guard bit) and 11 for half precision (10 fractional bits plus the guard bit). Additionally, there is an integer digit which can be 1 or 2, but this integer digit is obtained in parallel with the pre-scaling and it does not count in the digit iterations cycles.

Regarding the square root, the number of root bits is 54 for double precision, 25 for single precision and 12 for half precision.

Then, as explained in Section 3.4, the number of digit cycles in division is 9, 4, and 2 for double precision, single precision and half precision respectively; whereas, the num-

TABLE 4
Latency comparison

	Algorithm	Division			Square root		
		HP	SP	DP	HP	SP	DP
AMD K7	multiplicative	–	16	20	–	19	27
AMD Jaguar	multiplicative	–	14	19	–	16	27
IBM zSeries	radix-4	–	23	37	–	–	–
IBM z13	radix-8/4	–	18	28	–	22	37
HAL Sparc	multiplicative	–	16	19	–	22	27
Intel Penryn	radix-16	–	12	20	–	12	20
This paper	radix-64/16	4	6	11	5	8	15

ber of digit cycles in square root is 13, 6, and 3 for double, single and half precision, respectively.

Finally, the number of post-processing cycles is one if result is normal or two if the result is subnormal (only in division).

As an example, the cycles for the single precision division are shown below,

1) Normal inputs, normal result:
   PRE – DGT – DGT – DGT – DGT – RND
2) Normal inputs, subnormal result:
   PRE – DGT – DGT – DGT – DGT – RND – RSH
3) One subnormal input, normal result:
   UNP–NM–PSC–DGT–DGT–DGT–DGT–RND
4) One subnormal input, subnormal result:
   UNP–NM–PSC–DGT–DGT–DGT–DGT–RND–RSH
5) Two subnormal inputs,normal result:
   UNP–NM–NM–PSC–DGT–DGT–DGT–DGT–RND

where PRE is the pre-processing cycle with operands unpacking, pre-scaling and first iteration. This stage is split into unpacking (UNP) and normalization (NM) if any of the operands is subnormal as shown in items 3) to 5). On the other hand, DGT is a digit iteration cycle. Finally, RND and RSH are the rounding and right-shift cycles, respectively, the post-processing cycles. Note that, the RSH cycle is only need in case of a subnormal quotient.

Table 4 compares the latency of the proposed divider/square root unit with the latency of some other recent processors[2] for floating-point half, single and double precisions with normalized operands and result, Intel Penryn [2], IBM zSeries [10], IBM z13 [14], HAL Sparc [16], AMD K7 [19], AMD Jaguar [23]. The latencies shown in the table include the iteration cycles and the pre- and post-processing cycles, such as unpacking, pre-scaling and rounding. Note that no cycles for normalization are included here because it has been assumed that the operands are already normal; although, as stated previously, the proposed divider can handle subnormal inputs and output.

2. Some other modern processors, such as AMD Ryzen, Intel Skylake, and Intel Broadwell, are not included in the table because we have not been able to find the full latency figures of floating-point division and square root. However, some partial figures are given in [9]: the latency is specified with a cycle range which includes all the floating-point precisions. Thus, the floating-point division latency for Ryzen, Skylake and Broadwell is 8-15 cycles, 14-16 cycles, and 10-15 cycles, respectively. Similarly, the square root latency for Ryzen, Skylake and Broadwell is 8-20 cycles, 14-21 cycles, and 10-23 cycles, respectively

It has to be pointed out that the latency in Table 4 is in cycles, and the comparison is done only in terms of the latency without taking into account that different processors might run at different frequencies.

Most of the design in the table uses a multiplicative division algorithm, and one of them uses a radix-4 digit-recurrence implementation.

As shown in the table, our proposal gets much lower latencies. The multiplicative implementation are limited by the latency of the multiplier of multiply-and-accumulate units that, as stated in the introduction, can be very significant. On the other hand, the implementation in [10] uses a very low radix, which implies a high number of iterations, although its implementation is quite simple.

The Intel Penryn processor [2] implements a radix-16 combined division/square root unit by cascading two radix-4 iterations every cycle. Consequently, the latency is almost halved with respect to that of the radix-4 unit. As it is a combined unit, division and square root have the same latency.

Finally, the IBM z13 processor [14] has a divide and square root unit supporting single, double and quad precision, and all the hexadecimal floating-point data types. The underlying algorithm is a radix-8 division and radix-4 square root, generating 3 bits per cycle and 2 bits per cycle respectively. The major challenge was to perform a radix-8 divide or radix-4 square root step on a wide quad precision mantissa, 113 bits plus some extra rounding bits, and fit it in a single cycle.

In our implementation we have been able to put in a single cycle three and two radix 4 iterations, for division and square root respectively, by using speculation between iteration in the same cycle. In addition, there are only one pre-processing cycle before the iterations, unpacking of operands and pre-scaling, and one post-processing cycle for rounding after the iterations.

7.2 Area

The area is strongly dependent on the algorithm, digit-recurrence or multiplicative, being used for division and square root. In general, multiplicative algorithms have smaller area requirements if the multipliers or FMA units are shared with other floating-point instructions. In case of digit-recurrence algorithms, the radix has also a great impact, the larger the radix the large the area.

Therefore, the area of our division/square root unit is much larger than the area of the other units in the table; however, our focus was on obtaining a low latency division/square root unit.

The area of the rounding stage is not included in the discussion because it should be roughly the same for all the units.

7.2.1 Division/square root unit in this paper

Our unit uses a large number of 3-to-2 carry-save adders (CSA) and carry-propagate adders (CPA) in the iterative part. As for the divisor, in the digit iteration stage there are five 58-bit CSAs for iteration for a total of fifteen 58-bit CSAs, five 9-bit CPAs, and five 7-bit CPAs, plus the logic for the selection of three quotient digits and the multiplexers,

twelve 58-bit 4:1 muxes and two 5:1 small wide muxes. In addition, in the pre-scaling logic three 58-bit adders and some additional logic, two CSAs, multiplexers, and a reduced selection logic, are needed.

The digit iteration stage in the square root has ten 4-to-2 carry-save adders[3], six 8-bit adders and eight 7-bit comparators in the digit selection logic, plus some additional logic.

7.2.2 Other division/square root or division only units

The area of the radix-4 divider [10] is much smaller than the area in our proposal. The redundant partial remainder consists of a sum part of 116 bits and a carry part of 28 bits (only 1 out 4 carries are flopped); the 6 most-significant bits must be in non-redundant format because they are used for the quotient digit selection. The iteration is implemented with one stage of 116-bit 3-to-2 CSA and one stage of a 4-bit CPA; an additional 6-bit CPA is needed to deliver the 6 most-significant bits to the digit selection table.

The radix-16 division/square root unit in [2] has been obtained by concatenating two radix-4 iterations in the same cycle. Although the paper doesn't provide any area breakdown, some area information is provided in a later publication [18] by different authors. Replication is used in the remainder calculation at each radix-4 iteration; five partial remainders are computed speculatively and then one of them is selected once the digit is determined. Consequently, eight full-length 3-to-2 CSAs , four per radix-4 iteration, are used. The inputs to the 3-to-2 CSAs are the redundant partial remainder and the f-vector, which is different for division and square root; consequently, eight f-vector are speculatively generated. Each of these full-length f-vectors is obtained with a sequencer and a set of muxes.

No area data is provided for the divide and square root unit of the IBM z13 processor [14]. However, we can guess that the wider quad-precision datapath adds a large area overhead.

Finally, multiplicative division implementations [16], [19], [23] involve only modest additional cost because the existing FP multipliers are reused to perform each algorithm iteration. Only a look-up table for the initial seed and some additional logic is needed to implement the divider. The total storage required for reciprocal and reciprocal square root initial approximations in [19], [23] is 69 Kbits. In [16] the table is even smaller, 3.5 Kbits.

7.3 Timing

For the critical path delay estimation the Logical Effort model [26] is used in this section. Table 5 summarizes the delay of the basic gates (upper part) and of the main modules in Figures 2 and 3 (middle and lower parts respectively) in terms of a FO4 and its equivalent in picoseconds. We have considered a FO4 delay of $6\ ps$. The load of every signal has been taken into account, so that a fanout of n adds a delay equivalent to $\log_4 n$ FO4.

The fanout affects especially to the *SELECT* division module in Figure 3 and to the comparators in Figure 4 . The modules' output, the quotient or root digit, has a high fanout, roughly 64 gates.

3. Roughly, each 4-to-2 carry-save adder is equivalent to two CSAs

TABLE 5
Delay of basic gates and modules of the divider and square root

		FO4	ps
basic gates	inverter	1	6
	2-input gate	1.33	8
	3-input gate	1.67	10
	xor gate	2	12
	2:1 mux	2.66	16
pre-scaling	58-bit adder	14.35	86
	54-bit sub	14.35	86
	reduced *SEL* logic	4	24
	2:1 mux with load	$2.66 + \log_4(58)$*	40
digit cycle	6-bit adder	9.43	56
	7-bit adder/comp.	9.34	56
	9-bit adder	11	66
	3-to-2 CSA	4	24
	4-to-2 CSA	6	36
	5:1 mux	4.33	26
	division *SEL* logic	$5.33 + \log_4(64)$*	50
	sqrt comparators	$9.34 + \log_4(64)$*	78

*Due to fanout

7.3.1 Division

In the pre-scaling cycle there are two paths with roughly the path the same estimated delay,

54-bit sub → 2:1 mux with large fanout → 2:1 mux

and

2:1 mux → 3:2 CSA → 58-bit adder → 2:1 mux

being the delay of each path $142\ ps$. Note the large fanout in the first path 2:1 mux.

In the digit cycle, there are several candidates to be the critical path, but due to large fanout at the output of the *SEL* logic, the critical path consists of

6-bit adder → *SEL* logic → 5:1 mux → SEL *logic* → 5:1 mux → *SEL* logic → 5:1 mux

with an estimated delay of $300\ ps$

Then, in conclusion, the critical path of the divider is in the digit cycle and has a estimated delay of

$$t_{div} = 300\ ps$$

7.3.2 Square root

The critical path is in the digit cycle. As in division, due to the large fanout of the root digit the critical path is going through the selection logic of the two root digits,

8-bit adder → comparator → 5:1 mux → comparator → root update

taking into account that the root update is basically an and-or logic the estimated square root delay is

$$t_{sqrt} = 250\ ps$$

It is interesting to mention here that the combined division and square root unit in [18] implements a radix-16 division and square root iteration by overlapping two radix-4 iterations. There are quite a few similarities between the square root in this unit and in our proposal: in both units speculation is used in the calculation of the second digit in the cycle. In addition, as shown in section 5.2, our proposal uses speculation in the calculation of the remainder as well. Note that in case of not using speculation in the remainder calculation, the critical path for the square root digit cycle would have the calculation of $f[i+2]$ and a 4-to-2 CSA after the second comparator, resulting in a delay larger than that of the division cycle.

7.3.3 Unit critical path

The common division and square root critical path delay is

$$t_{unit} = max\{t_{div}, t_{sqrt}\} = 300\ ps \tag{25}$$

8 CONCLUSIONS

The architecture of a common unit for a radix-64 floating-point divider, providing 6 bits of the quotient per cycle, and radix-16 square root, providing 4 bits of the root per cycle, is presented.

To get a simple implementation and an affordable timing the radix-64 division iteration and the radix-16 floating-point square root iteration are built with 3 and 2 radix-4 iterations, respectively. Each radix-4 iteration provides 2 bits of the quotient or root for a throughput of 6-bit per cycle in division and 4 bits per cycle in square root. To improve the timing, speculation has been used between consecutive iterations in the cycle.

Additionally, some other improvements have been done in both division and square root. To have a simple digit selection logic, the divisor has been pre-scaled to a value close to 1, in such a way that the digit selection function does not depend on the divisor, it depends only on the 6 most-significant bits of the remainder. Pre-scaling has been implemented as the addition of three terms, which depend on the most-significant bits of the divisor. Of course, the dividend has to be scaled by the same amount as the divisor as well.

Further latency reductions for some floating-point division are obtained by left-shifting the dividend by 1 bit when it is larger than the divisor to have the result in $[1, 2)$, and by performing the first iteration, which gives the integer digit of the result, in parallel with the pre-scaling.

Regarding the square root, the first iteration has been skipped and integrated in the initialization stage. This way, the latency has been reduced by 1 cycle in single and double precision.

The result is a low latency floating-point digit-recurrence divider and square root unit, with latencies of 11, 6 and 4 cycles for double-precision, single-precision and half-precision division, and 15, 8 and 5 cycles for double-precision, single-precision and half-precision square root.

ACKNOWLEDGMENTS

The author would like to thank to the computer arithmetic team at ARM Inc in Austin, TX, USA, for their help and support while the author was with ARM in Austin.

REFERENCES

[1] J.D Bruguera. *Radix-64 Floating-Point Divider*. Proceedings of 25th IEEE international Symposium on Computer Arithmetic, pp. 87-94. 2018.

[2] J. Coke, H. Baliga, N. Cooray, e. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles. *Improvements in the Intels Coret Penryn Processor Family Architecture and Microarchitecture*. Intel Technology Journal, Vol. 12, N0. 3, pp. 179–192, 2008.

[3] D. DasSarma and D.W. Matula. *Faithful Bipartite ROM Reciprocal Tables*. Proceedings 12th Symposium Computer Arithmetic, pp. 17-28, 1995.

[4] M.D. Ercegovac, L. Imbert, D.W. Matula, J.M. Muller, and G. Wei, *Improving Goldschmidt Division, Square Root and SquareRoot Reciprocal*, IEEE Transactions on Computers, Vol. 49, No. 7, pp. 759-763, 2000.

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

[6] M. Ercegovac and T. Lang. *Simple Radix-4 Division with Operand Scaling*. IEEE. Transactions on Computers, Vol. 39, No. 9, pp. 1204-1208, 1994.

[7] M. Ercegovac and T. Lang,*Digital Arithmetic*. San Mateo, CA, USA: Morgan Kaufmann, 2004.

[8] M.J. Flynn. *On Division by Functional Iteration*. IEEE Transactions on Computers, Vol. 19, pp. 702-706, 1970.

[9] A. Fog. *Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. https://www.agner.org/optimize/instruction_tables.pdf, 2019.

[10] G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess. *High Performance Floating-Point Unit with 116 bit Wide Divider*. Proceedings of 16th IEEE international Symposium on Computer Arithmetic, 2003.

[11] IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-2008. IEEE Computer Soc., 2008.

[12] I. Koren, *Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations*, IEEE Transactions on Computers, Vol. 39, No. 8, pp. 1030-1037, 1990.

[13] T. Lang and J. D. Bruguera. *Floating-Point Multiply-Add-Fused with Reduced Latency*. IEEE. Transactions on Computers, Vol. 53, No. 8, pp. 988-1003, 2004.

[14] C. Lichtenau, S. Carlough, and S.M. Mueller. *Quad Precision Floating Point on ther IBM z13^{TM}*. Proceedings of 23th IEEE international Symposium on Computer Arithmetic, 2016.

[15] J.M. Muller, *Elementary Functions. Algorithms and Implementation*. Birkhauser, 2006.

[16] A. Naini, A. Dhablania. *1-GHz HAL SPARC64 Dual Floating-Point Unit with RAS Features*. Proceedings of 15th IEEE international Symposium on Computer Arithmetic, 2001.

[17] A. Nannarelli. *Performance/Power Space Exploration for Binary64 Division Units*. IEEE Transactions on Computers, Vol. 65, No. 5, pp. 1671-1677, 2016.

[18] A. Nannarelli. Radix-16 Combined Division and Square Root Unit. Proceedings of 20th IEEE international Symposium on Computer Arithmetic, 2011.

[19] S.F. Oberman. *Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor*. Proceedings of 14th IEEE international Symposium on Computer Arithmetic, 1999.

[20] J.A. Piñeiro, Javier D. Bruguera. *High-Speed Double-Precision Computation of Reciprocal, Division, Square Root, and Inverse Square Root*. IEEE Transactions on Computers, Vol. 51, No. 12, pp. 1377-1388, 2002.

[21] D. Piso and J.D. Bruguera. *Variable Latency Goldschmidt Algorithm Based on a New Rounding Method and a Remainder Estimate*. IEEE Transactions on Computers, Vol. 60, No. 11, pp. 1535-1546, 2011.

[22] J. Preiss, M. Boersma and S. M. Mueller. *Advanced Clock gating Schemes for Fused-Multiply-Add-Type Floating-Point Units*. Proceedings of 19th IEEE international Symposium on Computer Arithmetic, 2009.

[23] J. Rupley, J. King, E. Quinnell, F. Galloway, K. Patton, P. M. Seidel, J. Dinh, H. Bui, A. Bhowmik. *The Floating-Point Unit of the Jaguar x86 Core*. Proceedings of 21th IEEE international Symposium on Computer Arithmetic, 2013.

[24] M.J. Schulte and J.E. Stine, *Symmetric Bipartite Tables for Accurate Function Approximation*, Proceedings 13th Symposium Computer Arithmetic, pp. 175-183, 1997.

[25] S. Srinivasan, K. Bhudiya, R. Ramanarayanan, P. S. Babu, T. Jacob, S. K. Mathew, R. Krishnamurthy and V. Erraguntla. *Split-path Fused Floating Point Multiply Accumulate (FPMAC)*. Proceedings of 21th IEEE international Symposium on Computer Arithmetic, 2013.

[26] I. Sutherland, B. Sproull, and D. Harris *Logical Effort: Designing Fast CMOS Circuits*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 1999.

[27] N. Takagi, *Powering by a Table Look-Up and a Multiplication with Operand Modification*, IEEE Transactions on Computers, vol. 47, No. 11, pp. 1216-1222, Nov. 1998.

[28] P.T.P. Tang, *Table Look-Up Algorithms for Elementary Functions and their Error Analysis*, Argonne Nat'l Laboratory Report, MCS-P194-1190, 1991.

Javier D. Bruguera received graduate and PhD degrees from the University of Santiago de Compostela, Spain, in 1984 and 1989, respectively.

Currently Javier D. Bruguera is with ARM Ltd in Cambridge, UK, whom he joined initially for 1 year in 2015. From 2016 to 2019 he was with ARM Inc in Austin, TX, USA, and in March 2019 he returned to ARM Ltd in Cambridge.

Previously, he was an assistant professor at the Universities of Oviedo and A Corunna, Spain. In 1990, he joined the University of Santiago de Compostelaas an associate professor becoming a full professor in 1997.

Dr. Bruguera has been serving as Chair of the Department of Electronic and Computer Science of the University of Santiago de Compostela between 2006 and 2010. He was a research visitor in the Application Center of Microelectronics at Siemens, Munich, Germany; in the Department of Electrical Engineering and Computer Science at the University of California at Irvine (UCI); and in the Department of Scientific Calculation, Universite Pierre et Marie Curie (UPMC), Paris.

His research interest is in computer arithmeric, mainly on floating-point, integer and vector algorithms and microarchitectures.